CURSO DE PROGRAMACIÓN COMPETITIVA
URJC - 2018

Sesión 3 (4ª Semana)

David Morán (ddavidmorang@gmail.com)
Juan Quintana (juandavid.quintana@urjc.es)
Sergio Pérez (sergioperezp1995@gmail.com)

Contenidos

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

Contenidos

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

Algoritmos Voraces

- Definición
- Partes del algoritmo
- Funcionamiento
- Problemas frecuentes

Algoritmos Voraces

Definición

- Búsqueda eligiendo la opción más prometedora en cada paso local con la esperanza de llegar a una solución general óptima
- Rutinas muy eficientes $O(n)$, $O(n^2)$
- **NO** suelen proporcionar la solución óptima

Algoritmos Voraces

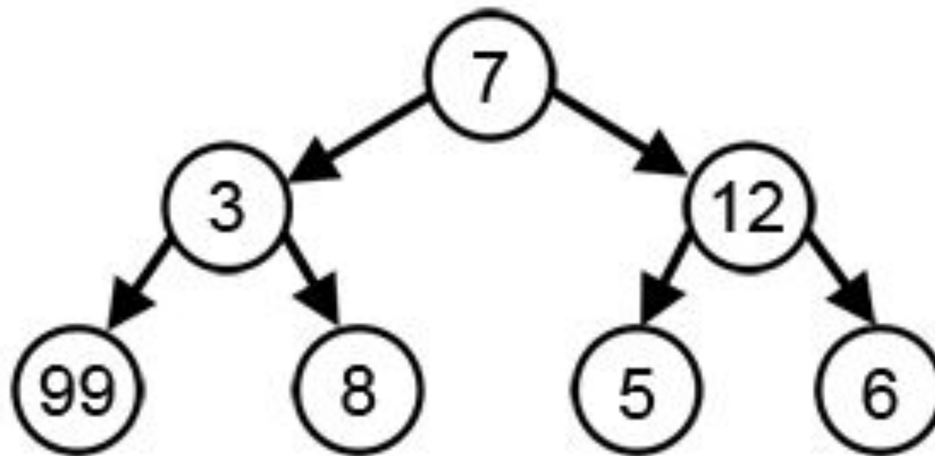
Partes del algoritmo

- **Conjunto de candidatos (*C*).** Entradas del problema
- **Función solución.** Comprueba, en cada paso, si el subconjunto actual de candidatos elegidos forma una solución
- **Función de selección.** Informa cuál es el elemento más prometedor para completar la solución
- **Función de factibilidad.** Informa si a partir de un conjunto se puede llegar a una solución.
- **Función objetivo.** Es aquella que queremos maximizar o minimizar, el núcleo del problema

Algoritmos Voraces

Funcionamiento

Algoritmo que busca el camino de mayor peso



Algoritmos Voraces

SPOJ_STAMPS

- http://www.spoj.com/problems/STAMPS/
- Lucy quiere superar en número la colección de sellos de Raymond
- Para ello, pide sellos a sus amigos
- Entrada: nº de escenarios (casos de prueba), nº de sellos necesarios para alcanzar a Raymond, nº de amigos que nos dejarán sellos, lista de los sellos que nos dejará cada uno

Algoritmos Voraces

SPOJ_STAMPS

- **Conjunto de candidatos:** lista de sellos que nos dejará cada amigo
- **Función solución:** Comprueba si hemos superado o no los sellos de Raymond
- **Función de selección:** Entre todos los amigos, escogeremos primero los que más sellos nos presten
- **Función de factibilidad:** ¿Tenemos suficientes sellos entre todos los amigos para superar a Raymond?
- **Función objetivo:** Minimizar el número de amigos necesarios para alcanzar a Raymond

Algoritmos Voraces

SPOJ_STAMPS: Ejemplo

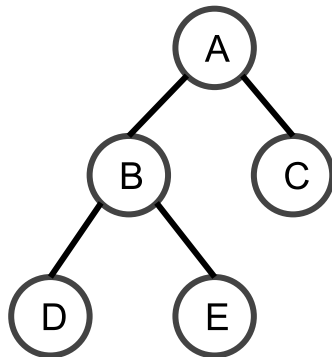
- **Objetivo:** Llegar a 100 sellos con 6 amigos (100 6)
- **Lista de candidatos:** 13 17 42 9 23 57
- **Función solución:** suma>=needed
- **Función de selección:** Escoger uno a uno los elementos del Array ordenado de mayor a menor
- **Función de factibilidad:** ¿suma<needed?
- **Función objetivo:** Minimizar el número de amigos necesarios para alcanzar a Raymond
- **Solución (Java):** https://pastebin.com/2LDxFwR9

Grafos

- ¿Qué es?
- Representación
  - Con índices
  - Con strings (utilizar map)

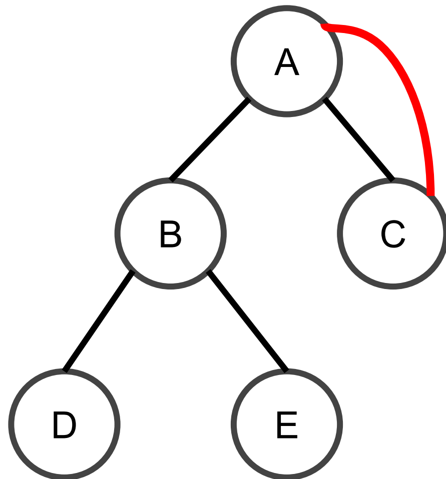
Grafos

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



Grafos

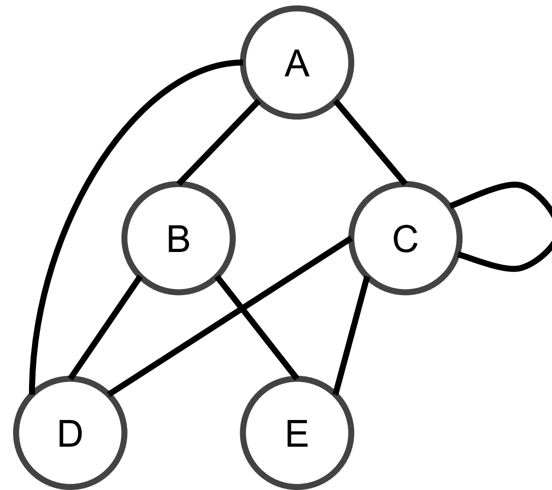
- Restricciones jerárquicas:
  - No admite ciclos



- C no puede ser padre de de su padre A

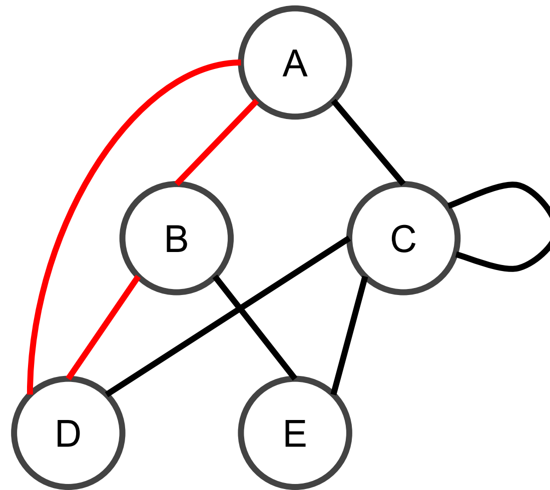
Grafos

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



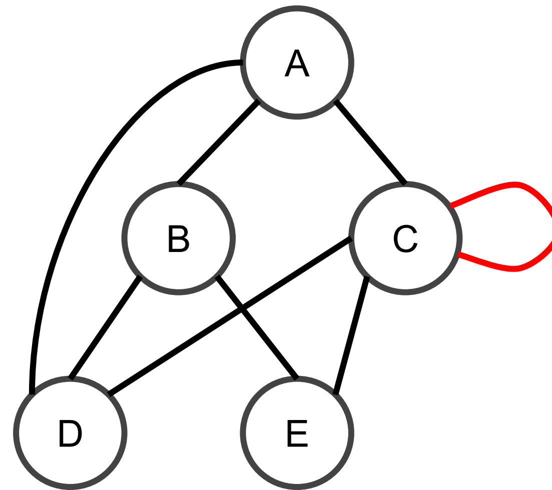
Grafos

- podemos tener ciclos



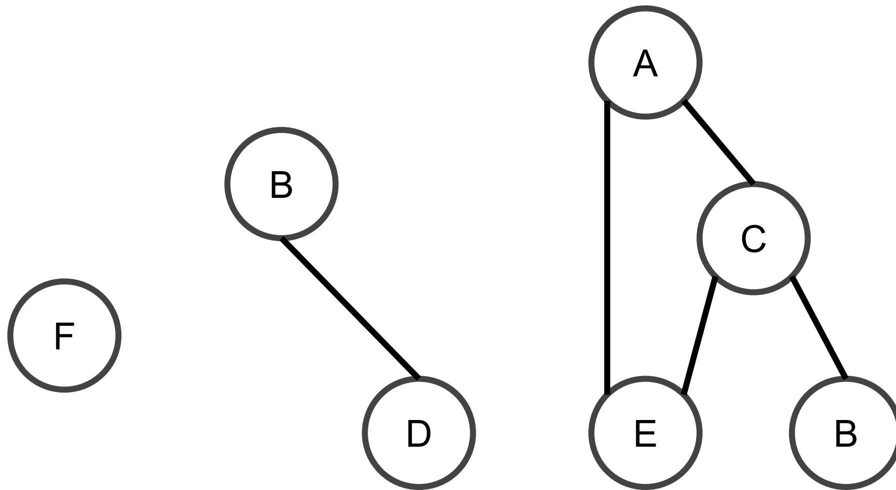
Grafos

- podemos tener bucles sobre el mismo elemento



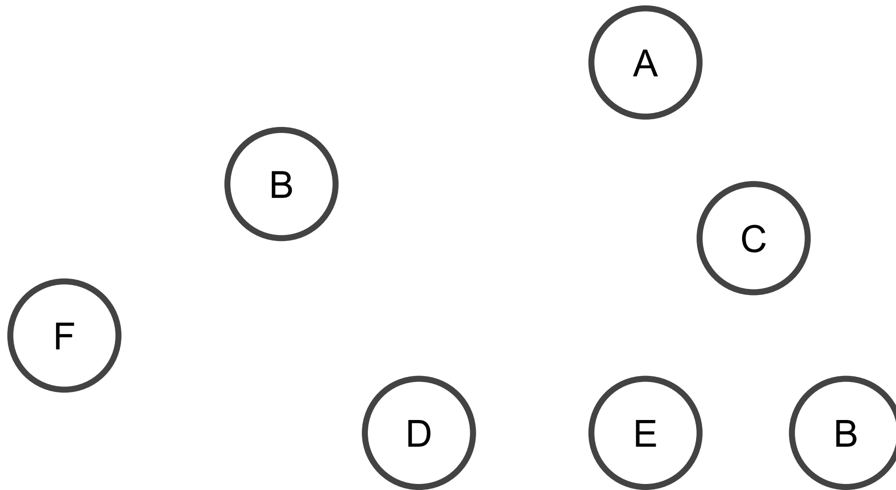
Grafos

- podemos tener grupos aislados en un mismo grafo



Grafos

- o elementos totalmente aislados entre sí



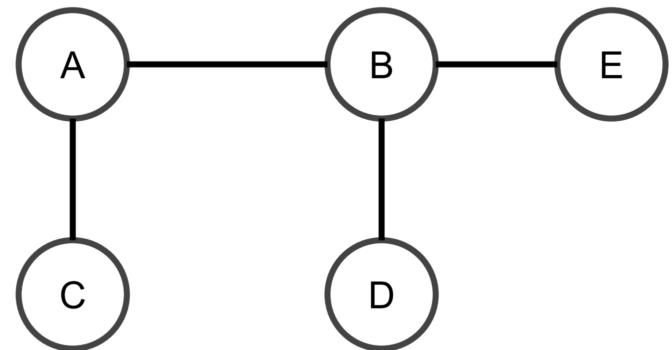
Grafos

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

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

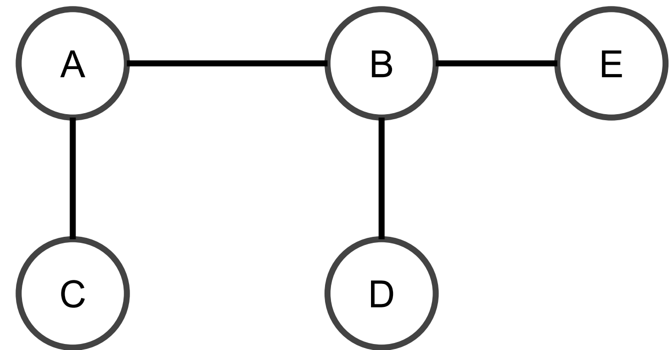
  - Conjunto de aristas:

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)}

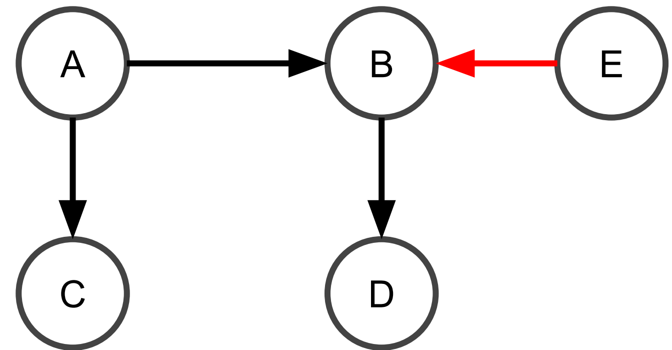
(A,B) ⇔ (B,A)

Grafos

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

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

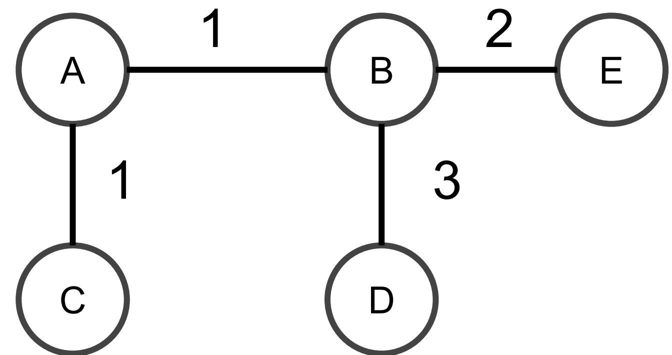
(E,B) ≠ (B,E)



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

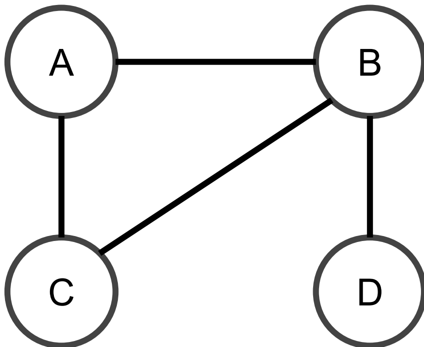
...

Grafos - Implementación

- Implementaciones
  - Matriz de adyacencia
  - Lista de adyacencia

Grafos - Matriz

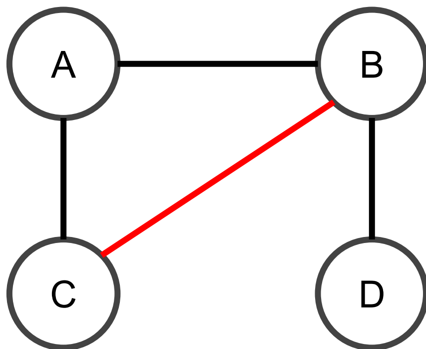
- Matriz de adyacencia
    - Array de dos dimensiones



	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

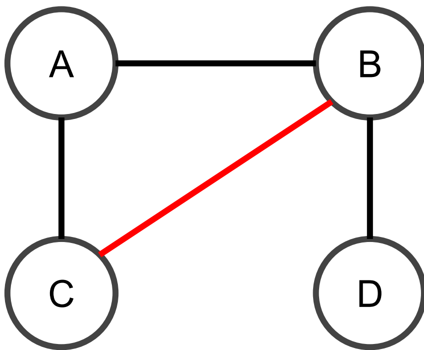
- Arista (B,C) ⇒ m[1][2] = 1



	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

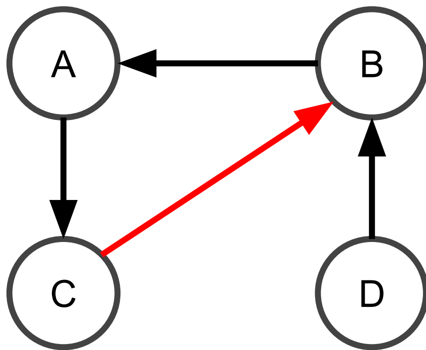
- Matriz simétrica en grafos no dirigidos
- m[fila][col]==m[col][fila]



	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

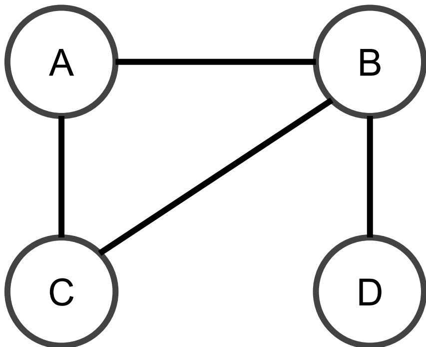
Grafos - Matriz

- 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

Grafos - Lista

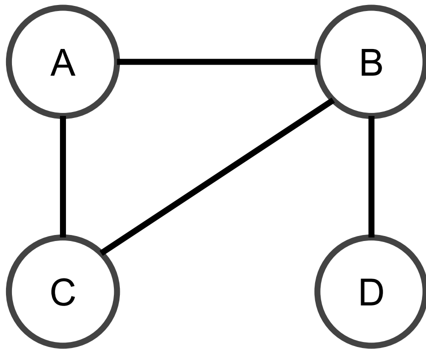
- Lista de adyacencia
  - enumerar las aristas por vértice



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

Grafos - Lista

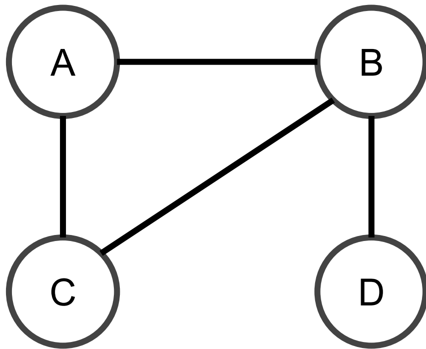
- Lista de adyacencia
  - guardar cada lista en un array



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

Grafos - Lista

- Lista de adyacencia
  - 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

- Lista de adyacencia
  - Memoria: O(|V|+|E|)
  - Acceso: O(|V|)
    - recorrer todas las aristas de la lista
  - Aristas de un vértice:
    - recorrer sus aristas sigue siendo O(|V|)
    - |Adyacentes| << |V|
- útil en grafos dispersos

Grafos - Mapas

- Lista de adyacencia
- Almacenar los adyacentes en un mapa
  - Acceso: O(1)

A	mapa={B,C}
B	mapa={A,C,D}
C	mapa={A,B}
D	mapa={B}

Grafos - Mapas

- Almacenar cada mapa dentro de un mapa…

  mapa<int, “listaAdyacencia”>

  mapa<int, mapa<int,bool>>

Grafos - Mapas

- Consultar si existe una arista (pseudocódigo)

```
grafo = mapa<int, mapa<int,bool>>
//inicializar el mapa / añadir aristas
A = 0
C = 2
adyacentesA = grafo.get(A)
existeAC = adyacentesA.get(C)
```

Grafos - Mapas

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

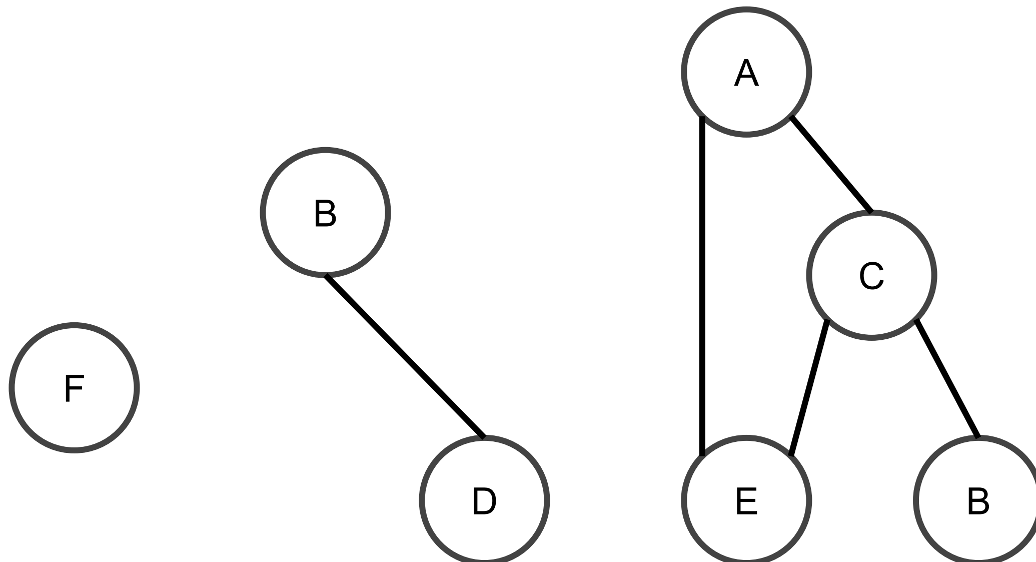
```
grafo = mapa<string, mapa<string,bool>>
adyacentes = grafo.get(“madrid”)
existeArista = adyacentes.get(“barcelona”)
```

Grafos - Recorridos

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

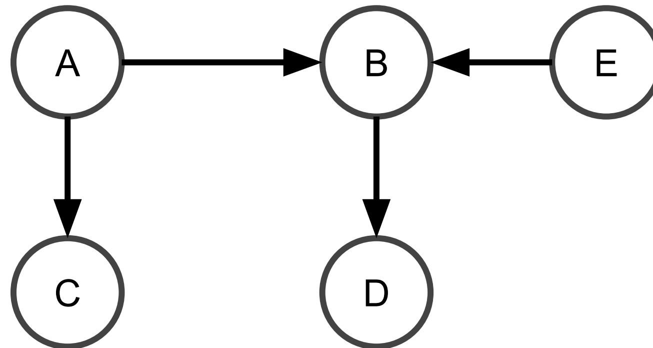
Grafos - Recorridos

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



Grafos - Recorridos

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

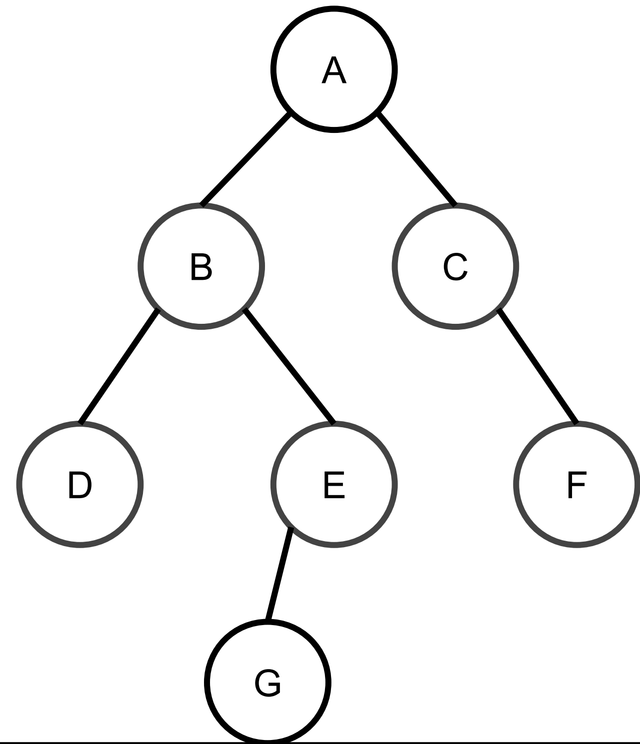


Grafos - Recorridos

- Recorrer los 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

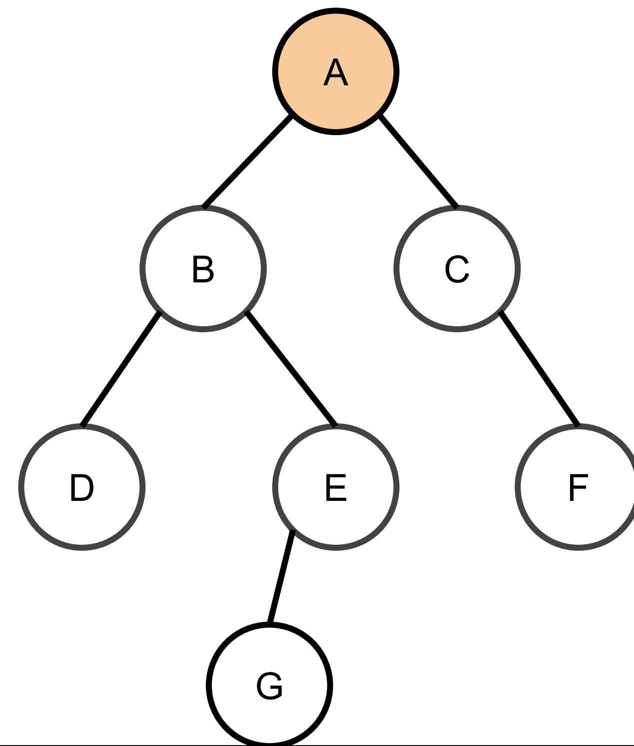
- Recorrido en anchura



Grafos - BFS

- Si empezamos desde A
- Recorrido por niveles

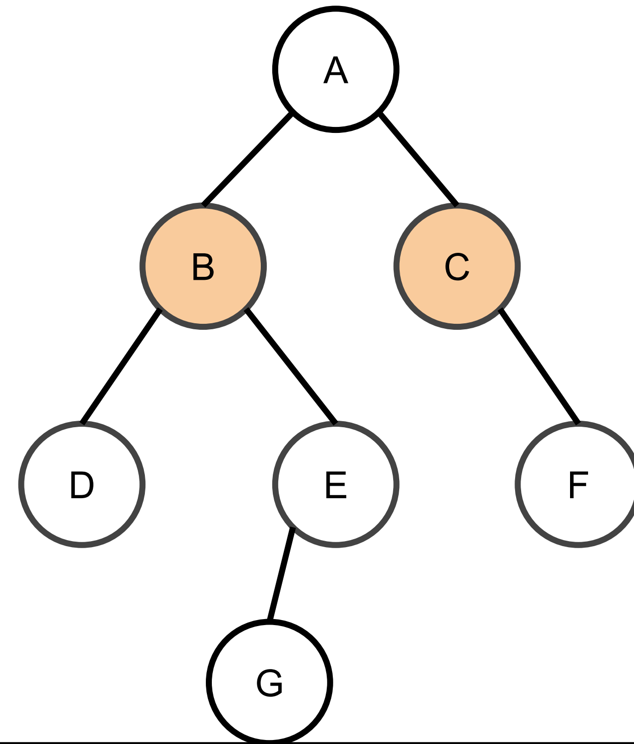
A



Grafos - BFS

- Si empezamos desde A
- Recorrido por niveles

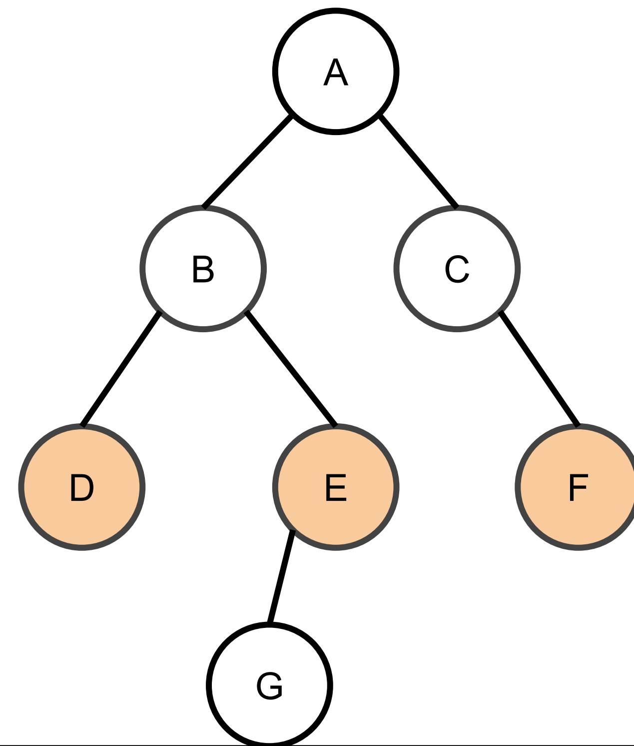
A ⇒ B ⇒ C



Grafos - BFS

- Si empezamos desde A
- Recorrido por niveles

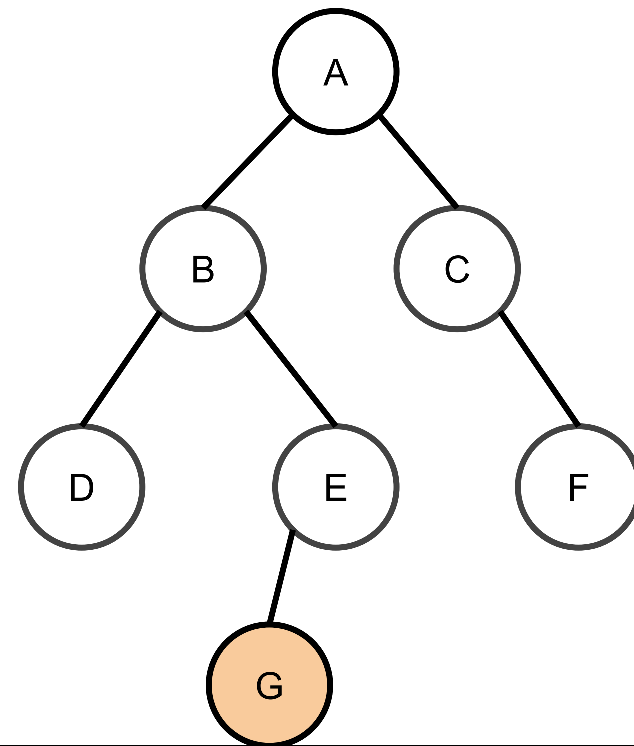
A ⇒ B ⇒ C ⇒ D ⇒ E ⇒ F ⇒ G



Grafos - BFS

- Si empezamos desde A
- Recorrido por niveles

A ⇒ B ⇒ C ⇒ D ⇒ E
⇒ F ⇒ G



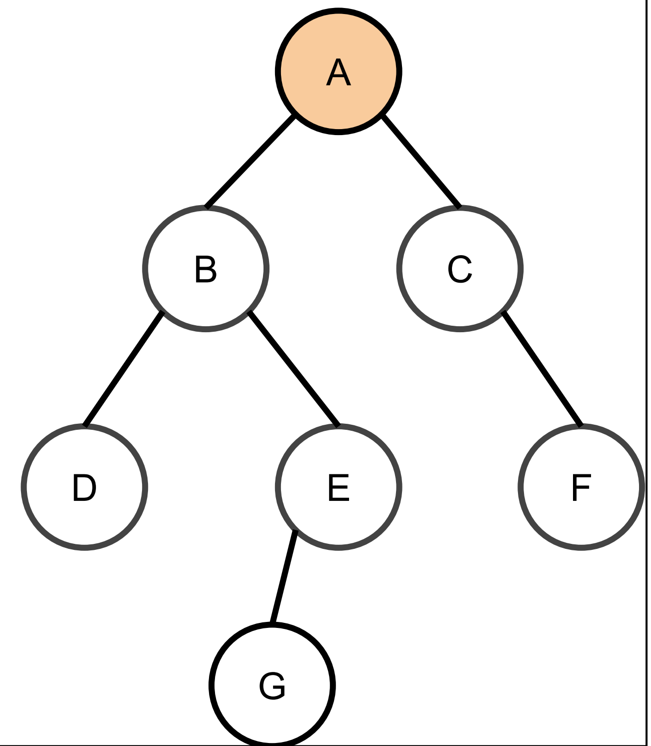
Grafos - BFS

- 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)
```



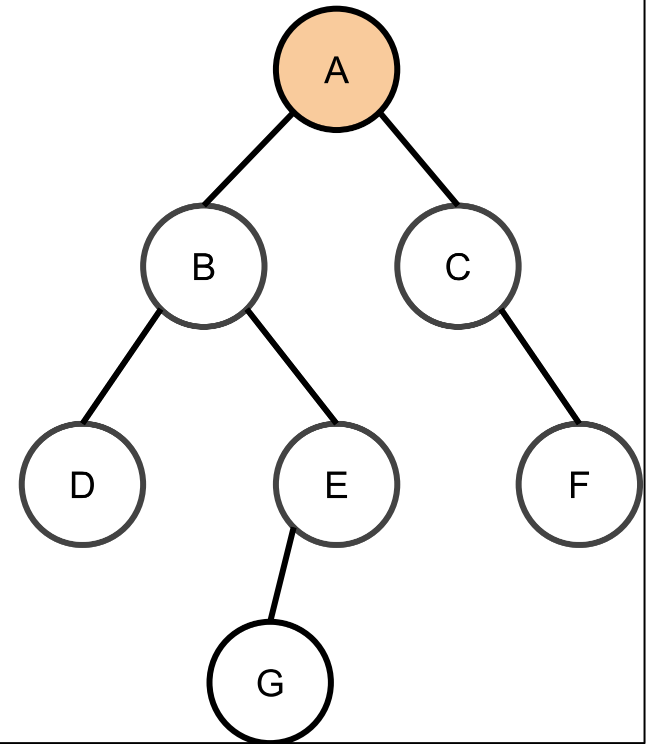
Grafos - BFS

- Cola = {A}

inicial = A

visitado[inicial]=true

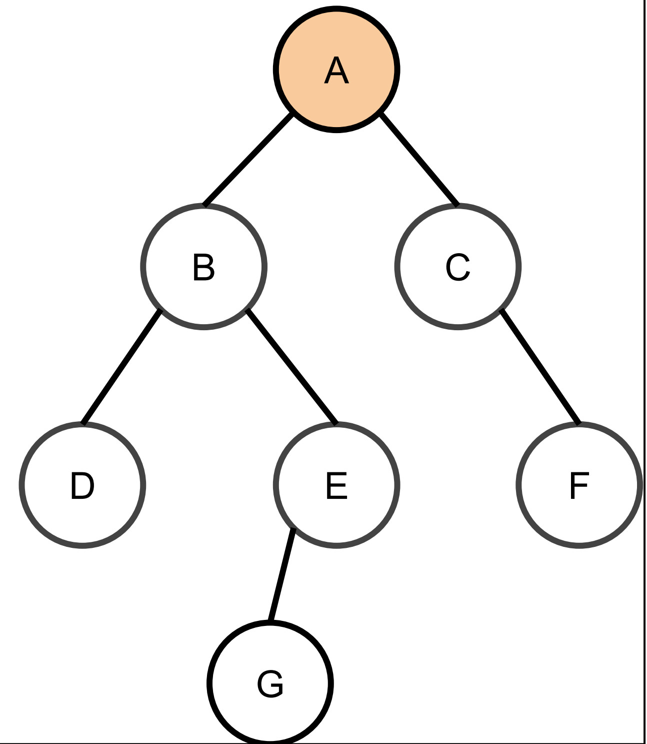
cola.add(inicial)



Grafos - BFS

- Recorremos los vértices

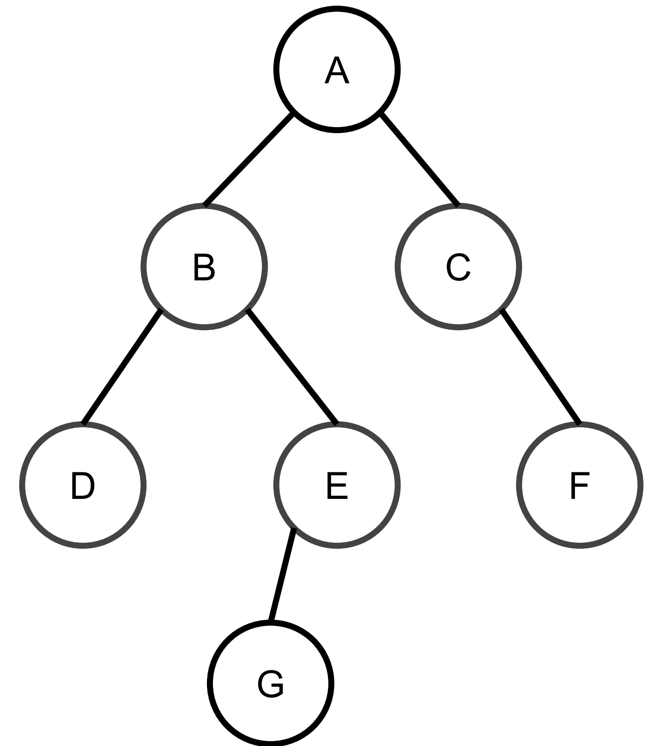
```
mientras(cola.size() > 0)
    v = cola.extraer()
    para cada ady de v:
        if(no visitado[ady])
            visitado[ady]=true
            cola.add(ady)
```



Grafos - BFS

- Cola = {}
- Sacamos A

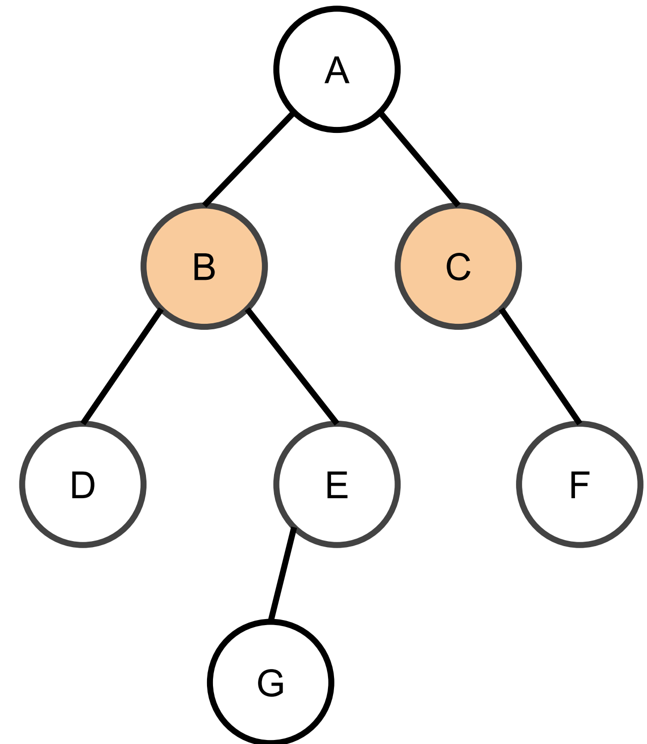
```
mientras(cola.size() > 0)
    v = cola.extraer()
    para cada ady de v:
        if(no visitado[ady])
            visitado[ady]=true
            cola.add(ady)
```



Grafos - BFS

- Cola = {B,C}

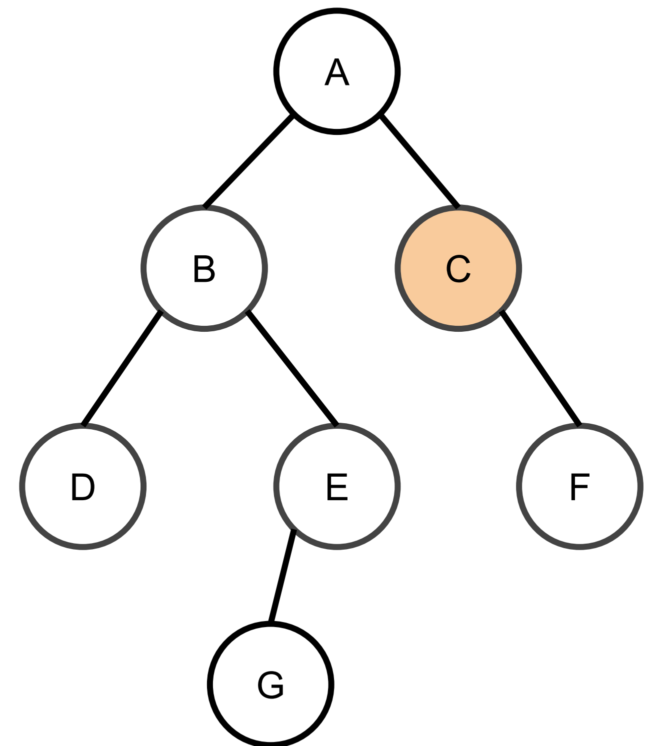
```
mientras(cola.size() > 0)
    v = cola.extraer()
    para cada ady de v:
        if(no visitado[ady])
            visitado[ady]=true
            cola.add(ady)
```



Grafos - BFS

- Cola = {C}
- Sacamos B

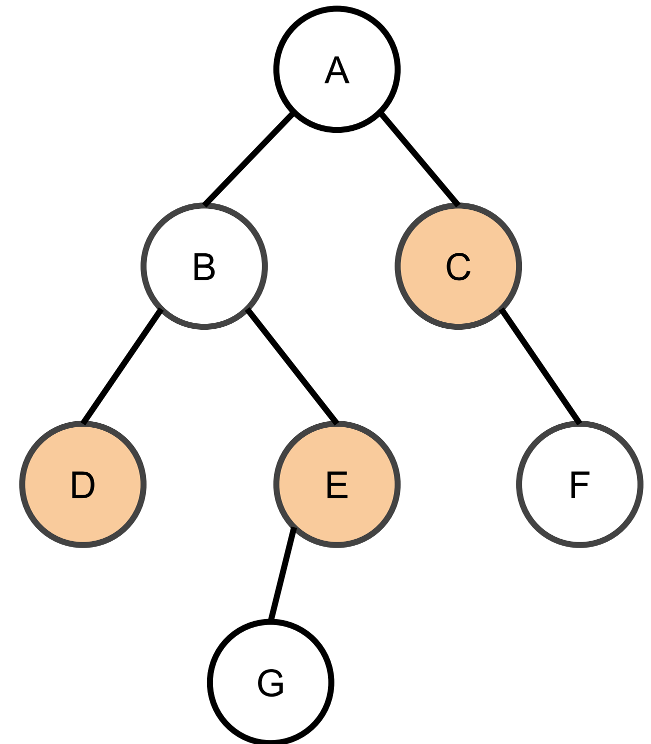
```
mientras(cola.size() > 0)
    v = cola.extraer()
    para cada ady de v:
        if(no visitado[ady])
            visitado[ady]=true
            cola.add(ady)
```



Grafos - BFS

- Cola = {C, D, E}
-

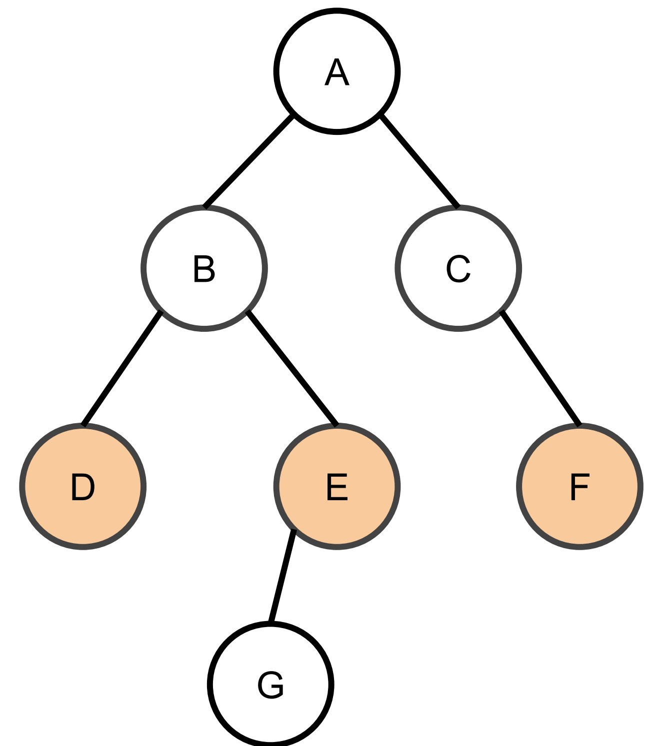
```
mientras(cola.size() > 0)
    v = cola.extraer()
    para cada ady de v:
        if(no visitado[ady])
            visitado[ady]=true
            cola.add(ady)
```



Grafos - BFS

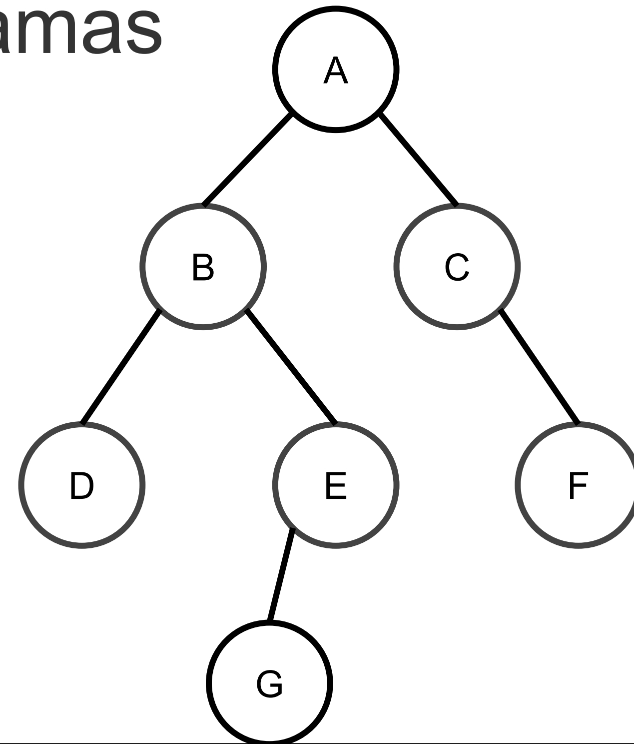
- Cola = {D, E, F}
- Sacamos C, Metemos F

```
mientras(cola.size() > 0)
    v = cola.extraer()
    para cada ady de v:
        if(no visitado[ady])
            visitado[ady]=true
            cola.add(ady)
```



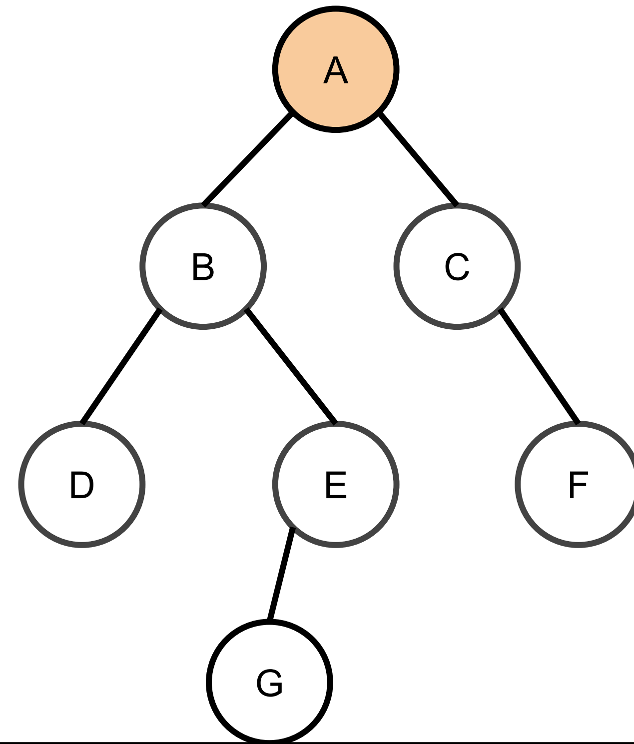
Grafos - DFS

- Recorrido en profundidad
- Equivalente a recorrer ramas



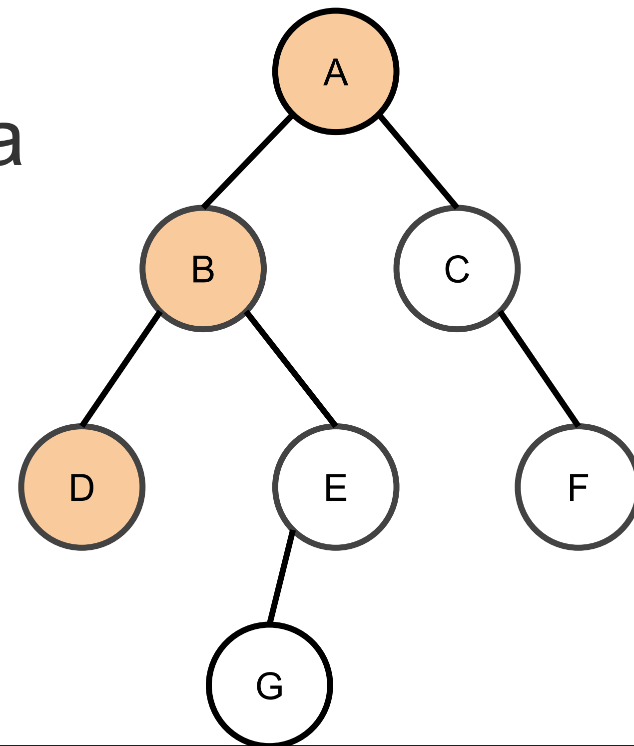
Grafos - DFS

- Recorrido en profundidad
- Seleccionamos A



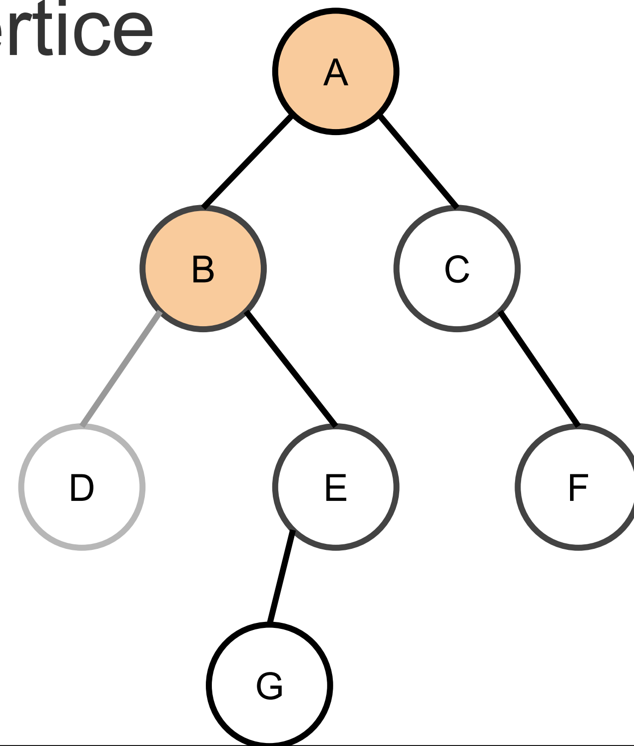
Grafos - DFS

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



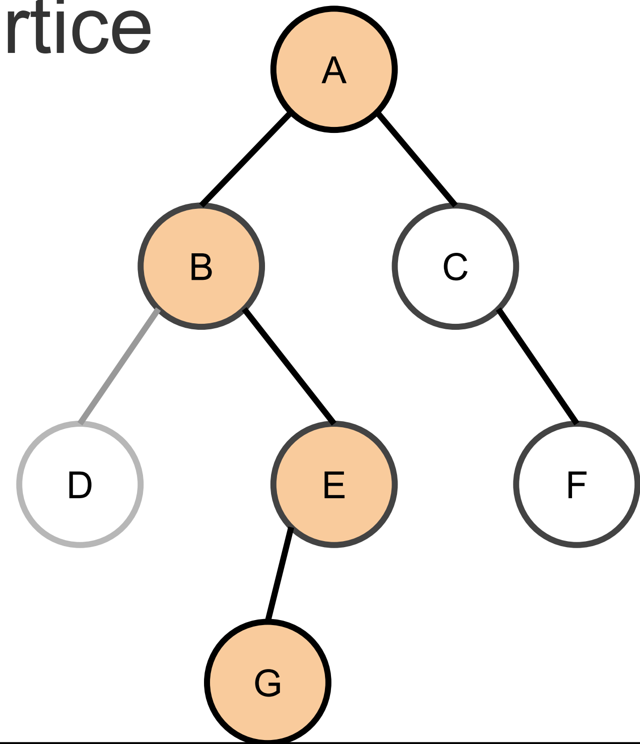
Grafos - DFS

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



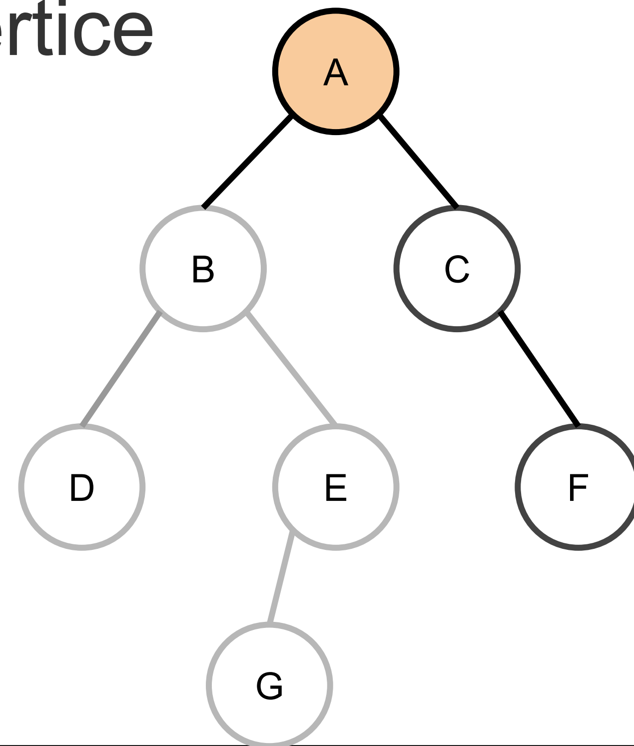
Grafos - DFS

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



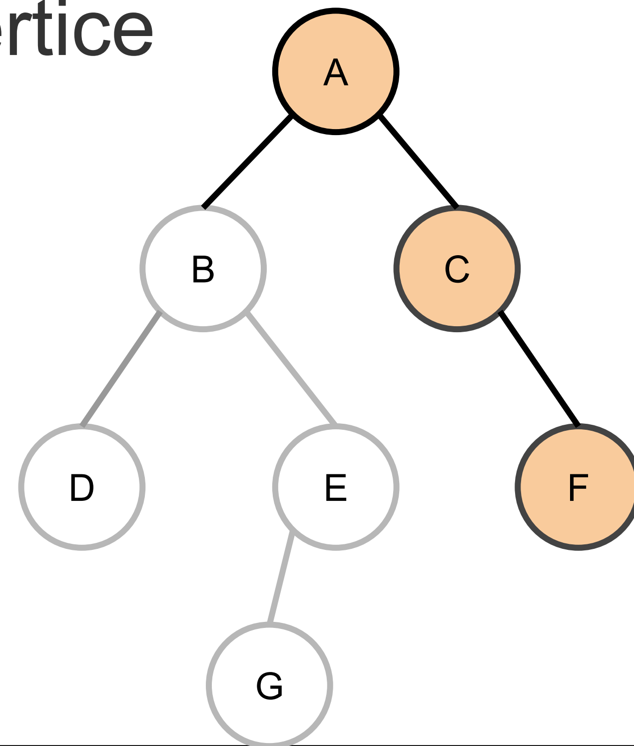
Grafos - DFS

- 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

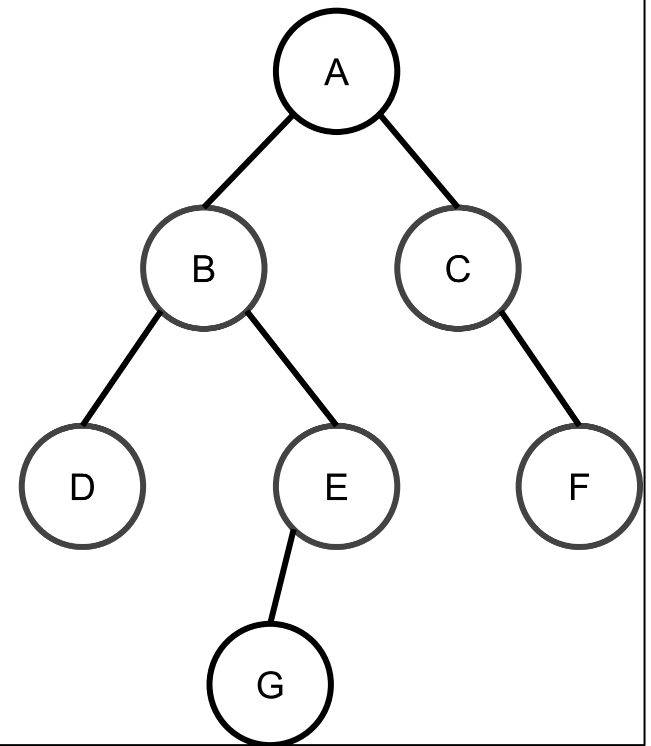
Grafos - DFS

- Inicialización: Elegimos un vértice inicial

inicial = A

visitado[inicial]=true

pila.add(inicial)



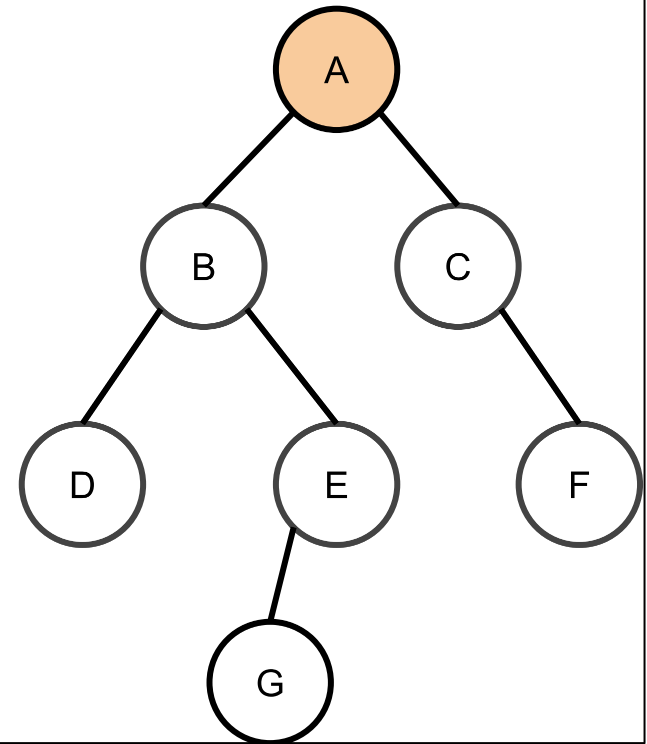
Grafos - DFS

- Pila = {A} ⇐ cima de la pila por la derecha

inicial = A

visitado[inicial]=true

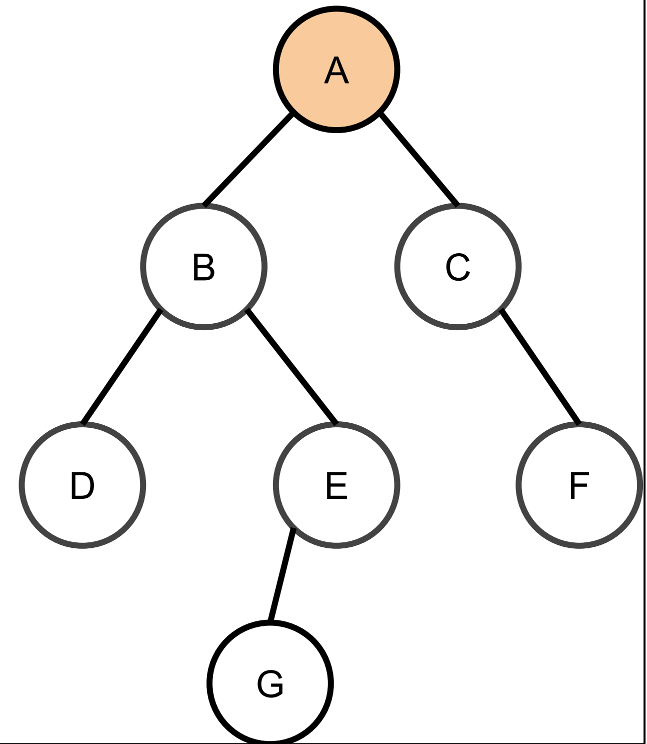
pila.add(inicial)



Grafos - DFS

- Recorremos los vértices

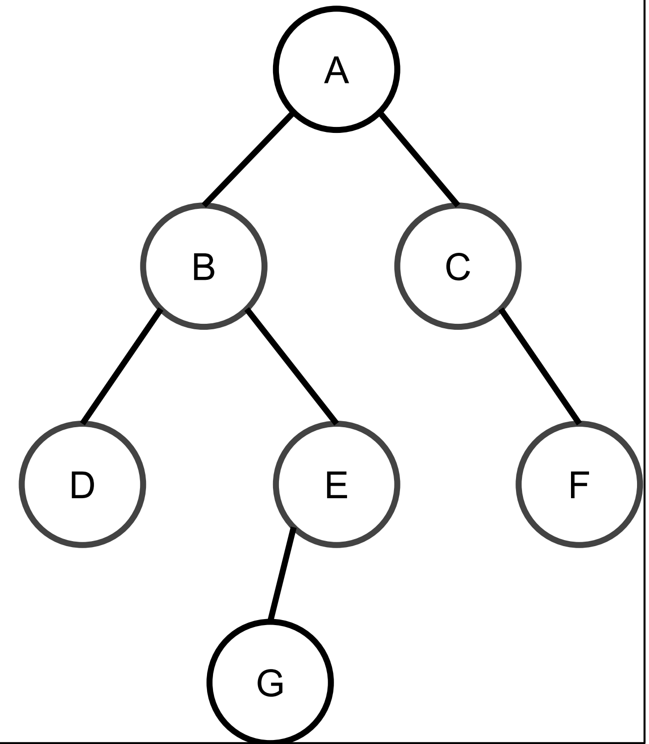
```
mientras(pila.size() > 0)
    v = pila.extraer()
    para cada ady de v:
        if(no visitado[ady])
            visitado[ady]=true
            pila.add(ady)
```



Grafos - DFS

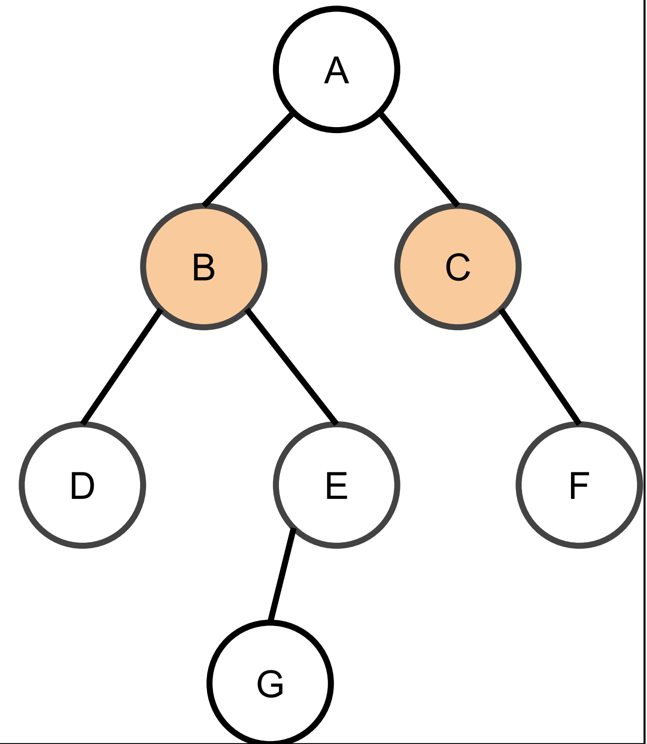
- Pila = {}
- Sacamos A

```
mientras(pila.size() > 0)
    v = pila.extraer()
    para cada ady de v:
        if(no visitado[ady])
            visitado[ady]=true
            pila.add(ady)
```



- Pila = {B,C}

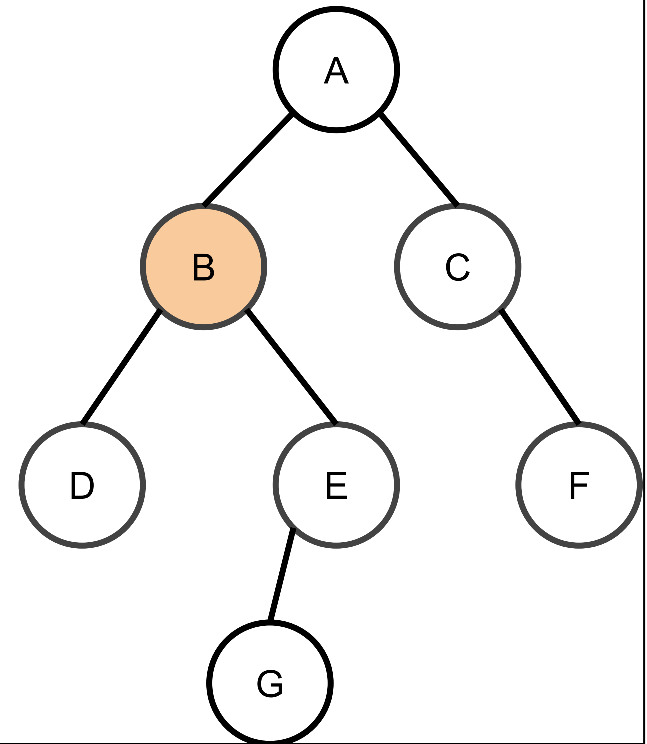
```
mientras(pila.size() > 0)
    v = pila.extraer()
    para cada ady de v:
        if(no visitado[ady])
            visitado[ady]=true
            pila.add(ady)
```



Grafos - DFS

- Pila = {B}
- Sacar C

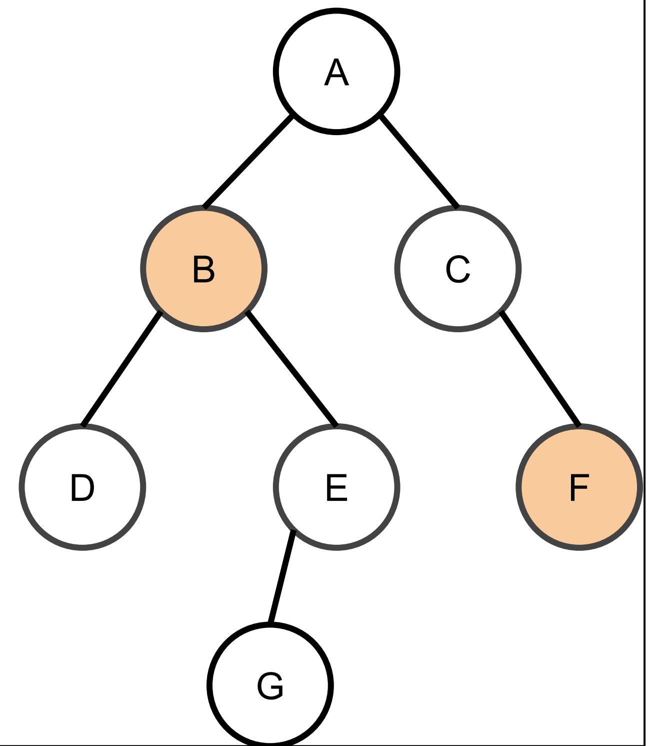
```
mientras(pila.size() > 0)
    v = pila.extraer()
    para cada ady de v:
        if(no visitado[ady])
            visitado[ady]=true
            pila.add(ady)
```



Grafos - DFS

- Pila = {B,F}

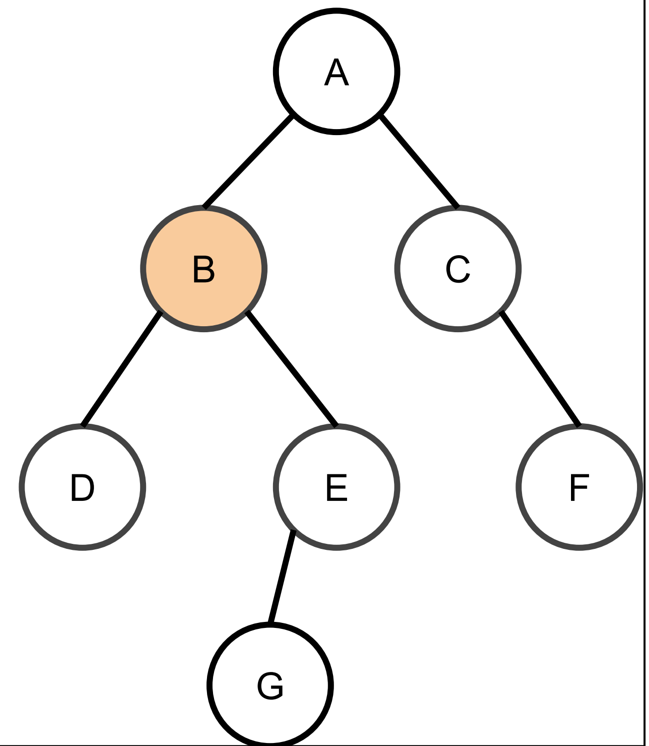
```
mientras(pila.size() > 0)
    v = pila.extraer()
    para cada ady de v:
        if(no visitado[ady])
            visitado[ady]=true
            pila.add(ady)
```



Grafos - DFS

- Pila = {B}
- Sacar F

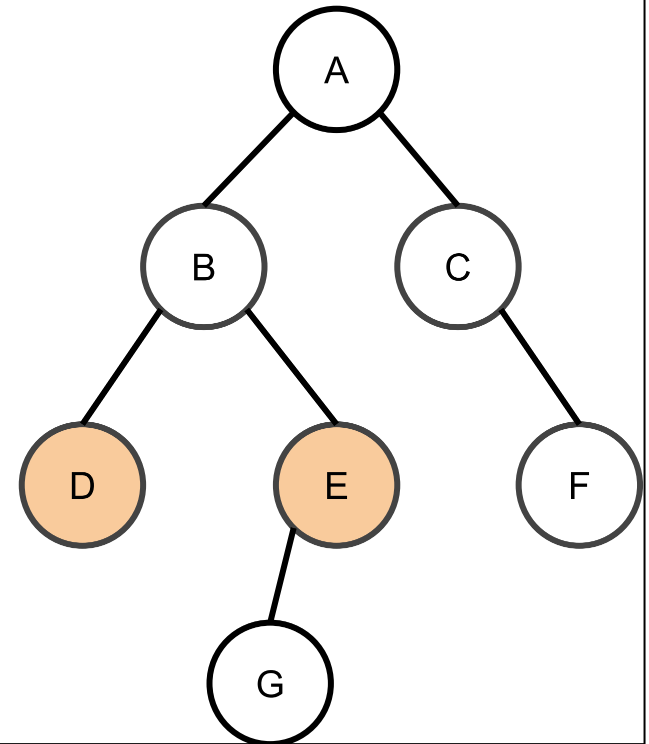
```
mientras(pila.size() > 0)
    v = pila.extraer()
    para cada ady de v:
        if(no visitado[ady])
            visitado[ady]=true
            pila.add(ady)
```



Grafos - DFS

- Pila = {D,E}
- Sacar B, Meter D,E

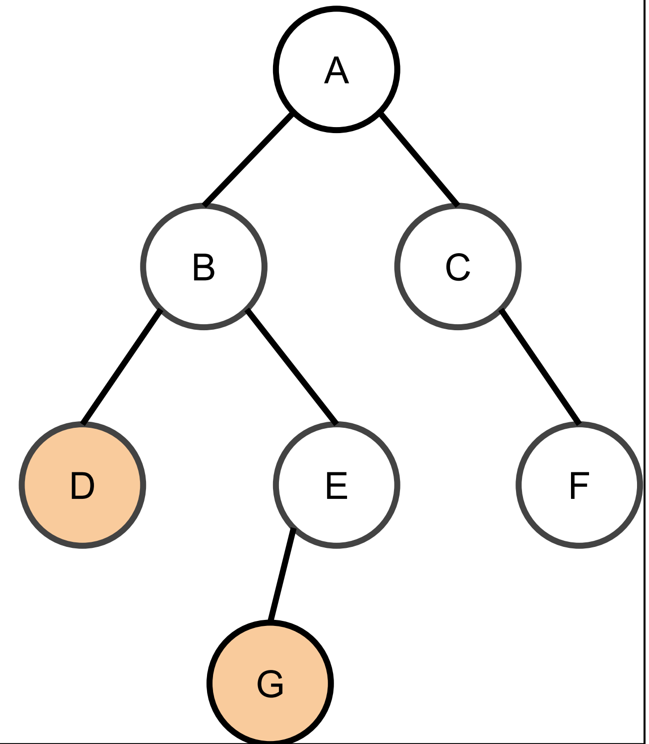
```
mientras(pila.size() > 0)
    v = pila.extraer()
    para cada ady de v:
        if(no visitado[ady])
            visitado[ady]=true
            pila.add(ady)
```



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

- Camino / Ciclo Euleriano
  - Explicar SGAME
- Camino / Ciclo Hamiltoniano
  - Máscaras de bits

- Camino / Ciclo Euleriano
- Camino / Ciclo Hamiltoniano

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

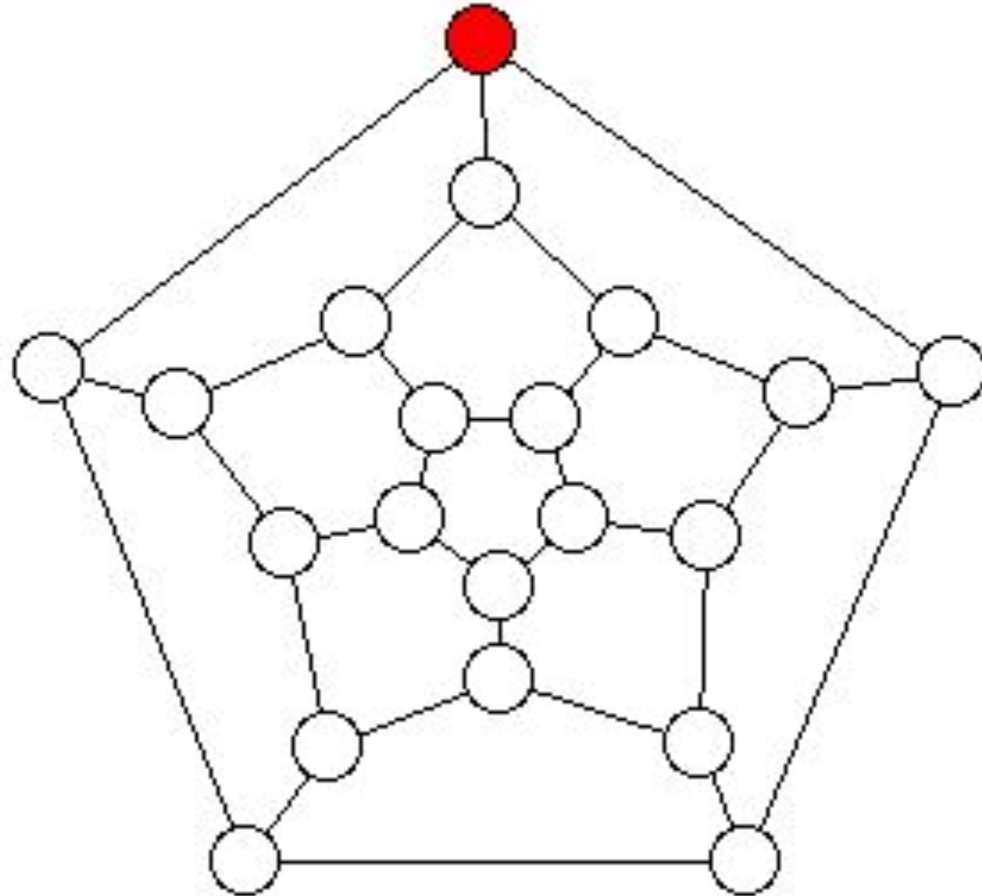
Grafos | Camino y ciclo Hamiltoniano

- 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 ^ (1<<i) donde i es el nodo

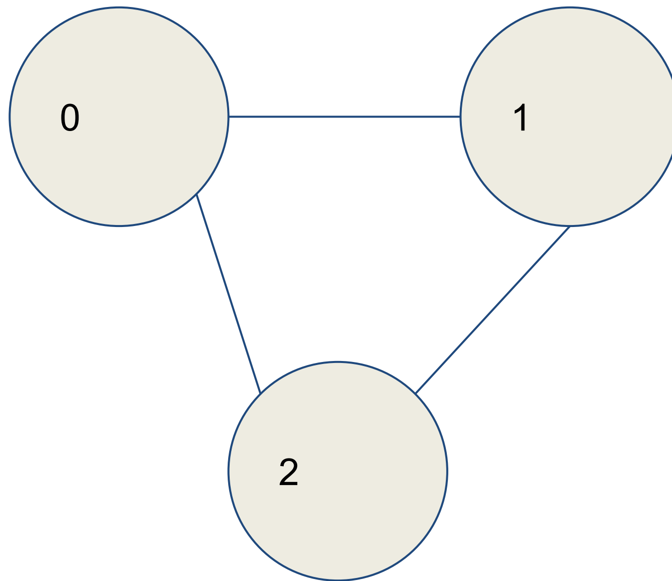
Grafos | Camino y ciclo Hamiltoniano

- 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

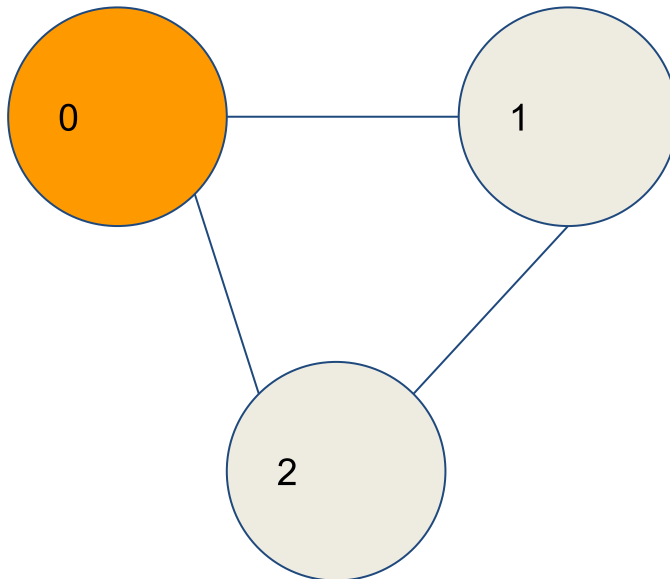


Grafos | Camino y ciclo Hamiltoniano



NODO=0,
MASK=111_2

Grafos | Camino y ciclo Hamiltoniano



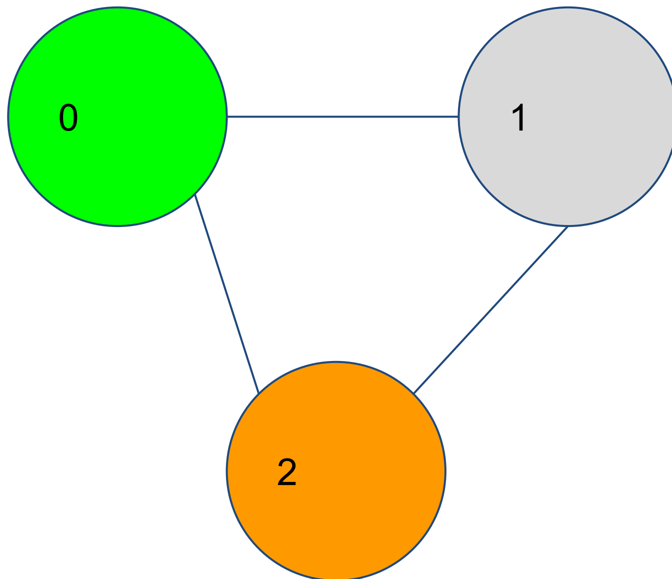
NODO=0,

MASK=111_2

$2^0 = 1$

111 & 001 > 0?

Grafos | Camino y ciclo Hamiltoniano



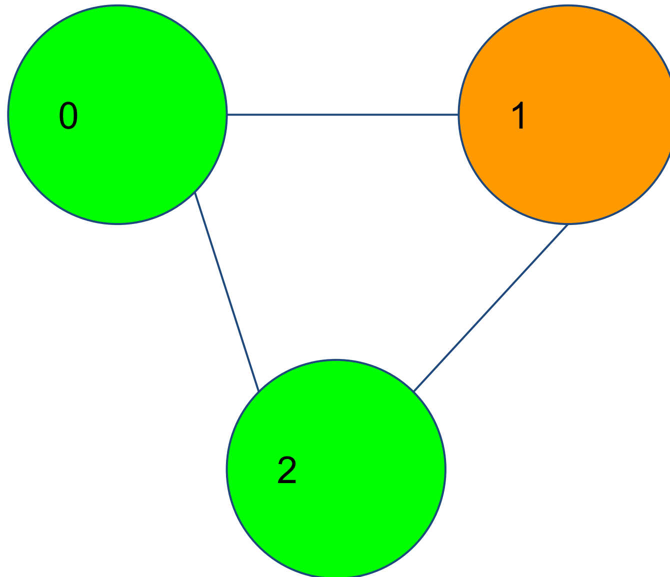
NODO=2,

MASK=110_2

$2^2 = 4$ (100)

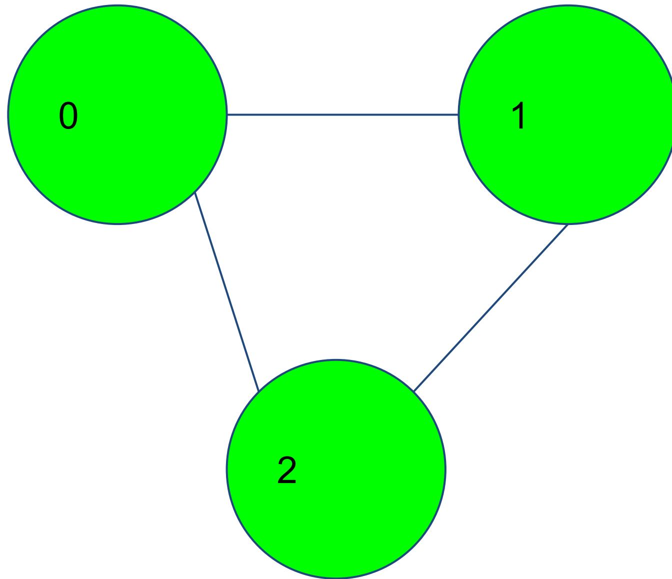
110 & 100 > 0?

Grafos | Camino y ciclo Hamiltoniano



NODO=1,
MASK=010_2
2^1 = 2 (010)
010 & 010 > 0?

Grafos | Camino y ciclo Hamiltoniano



NODO=X, MASK=000_2
Es camino Hamiltoniano

¿Es tambien ciclo?

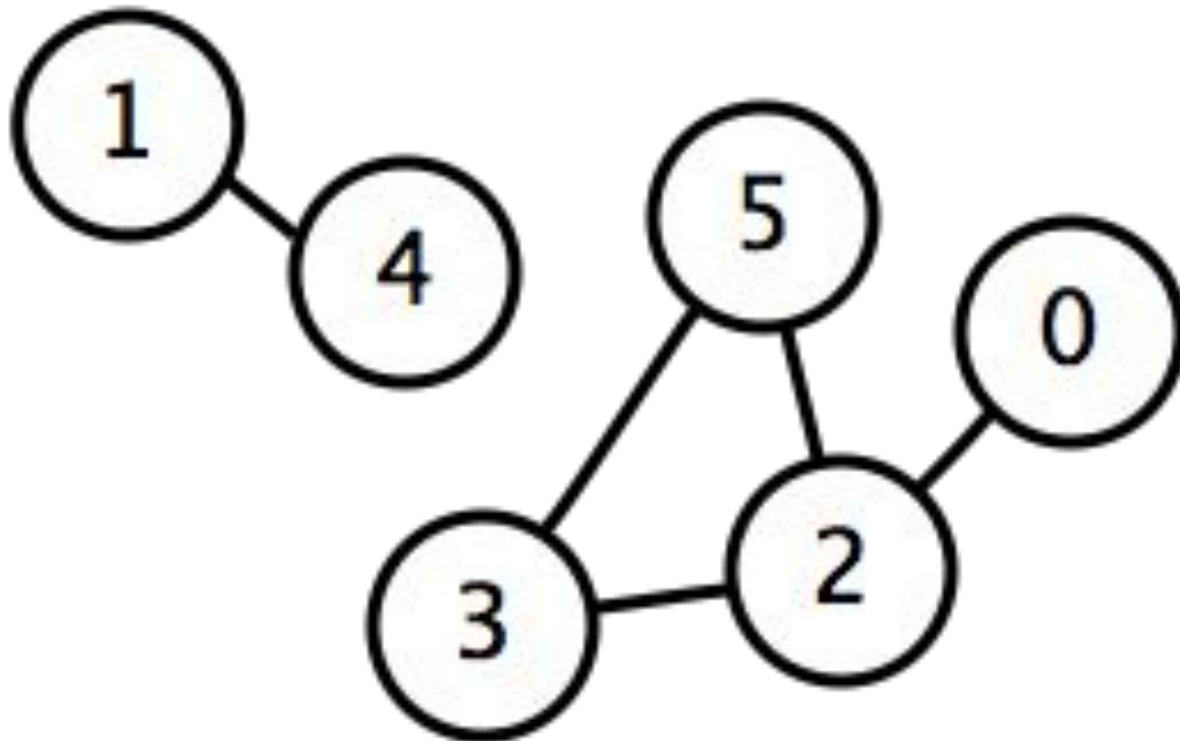
Grafos | Componentes Conexas

- 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

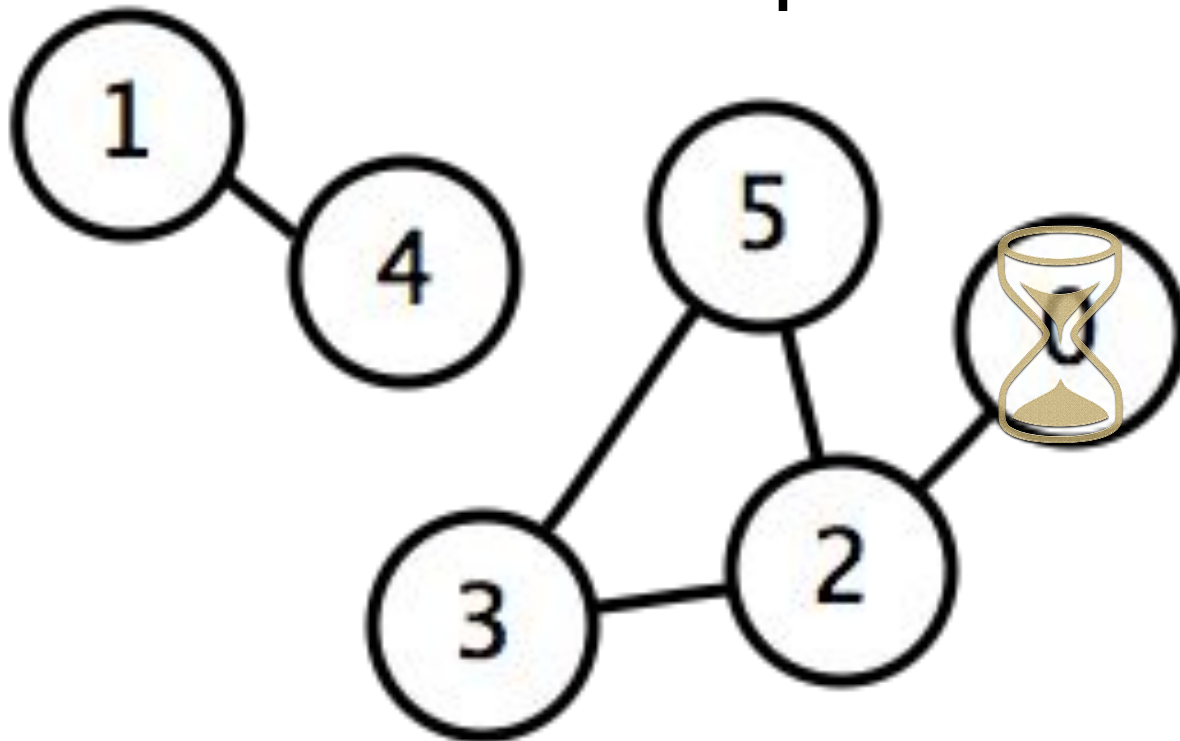
- La idea básica es hacer un BFS/DFS por cada vértice i desde 0..N siempre y cuando i no haya sido recorrido por un algoritmo anterior
- Se cuenta 1 y se recorre i

Grafos | Componentes Conexas

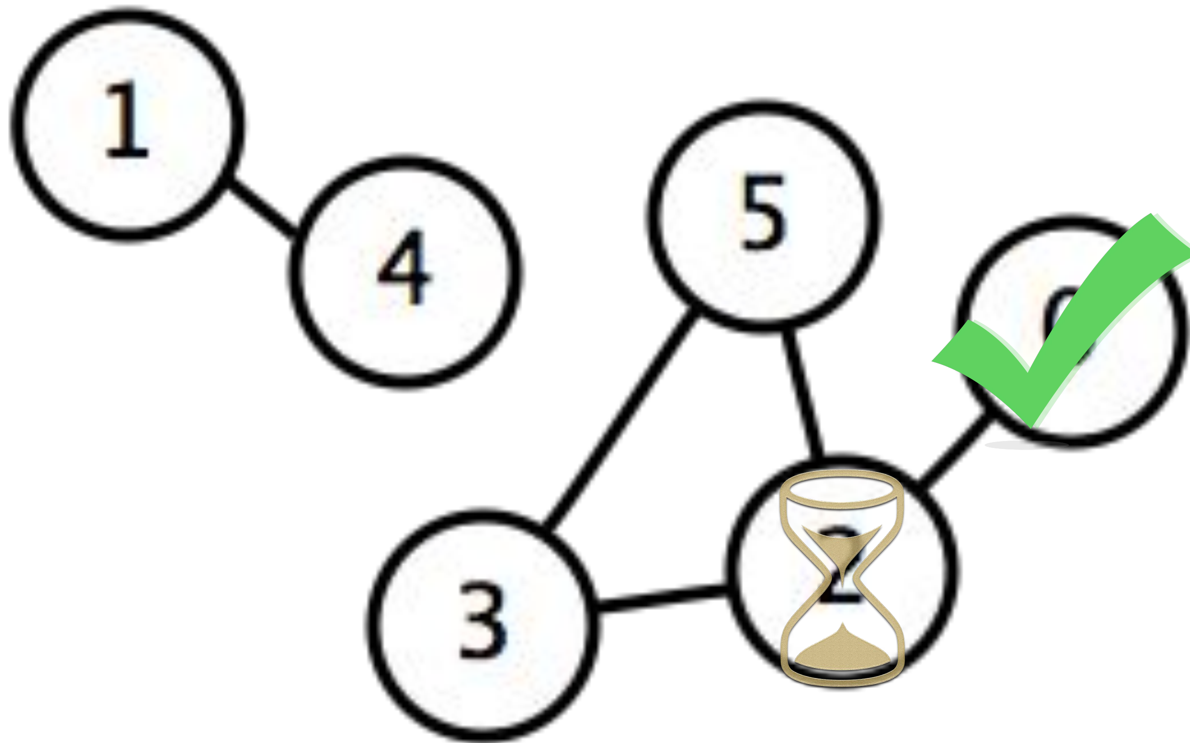


Grafos | Componentes Conexas

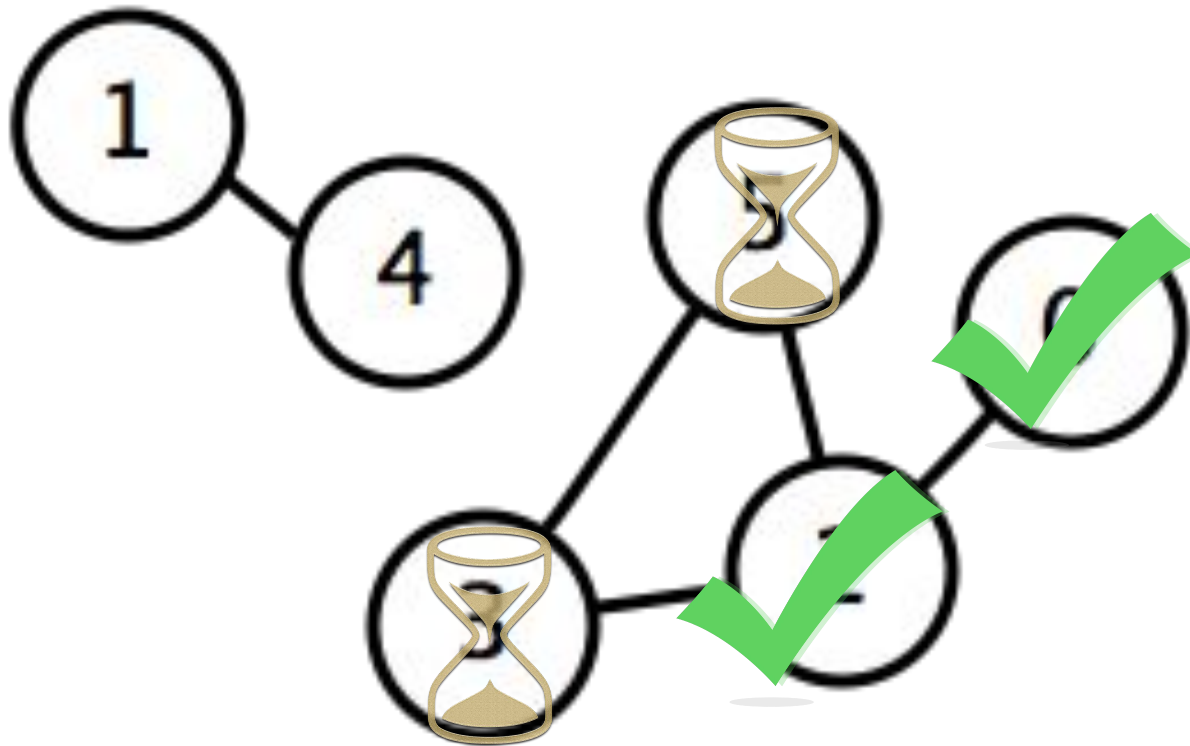
+1 Componente Conexa (1)



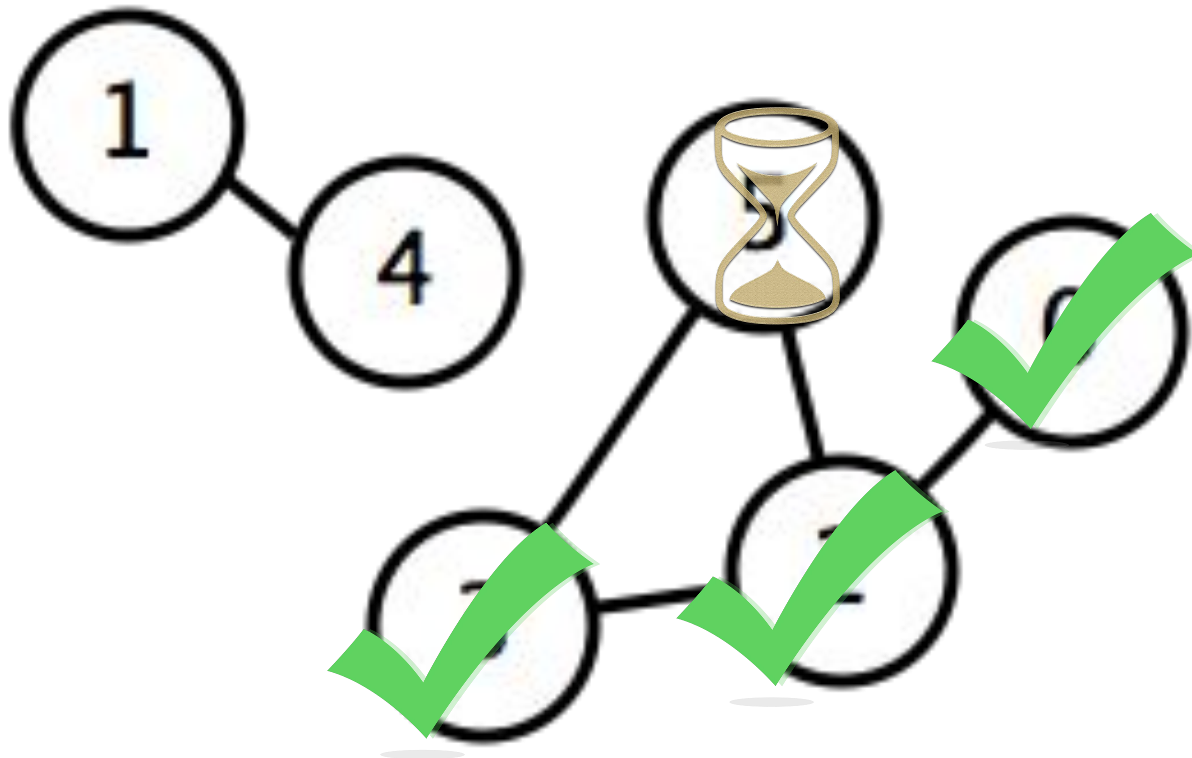
Grafos | Componentes Conexas



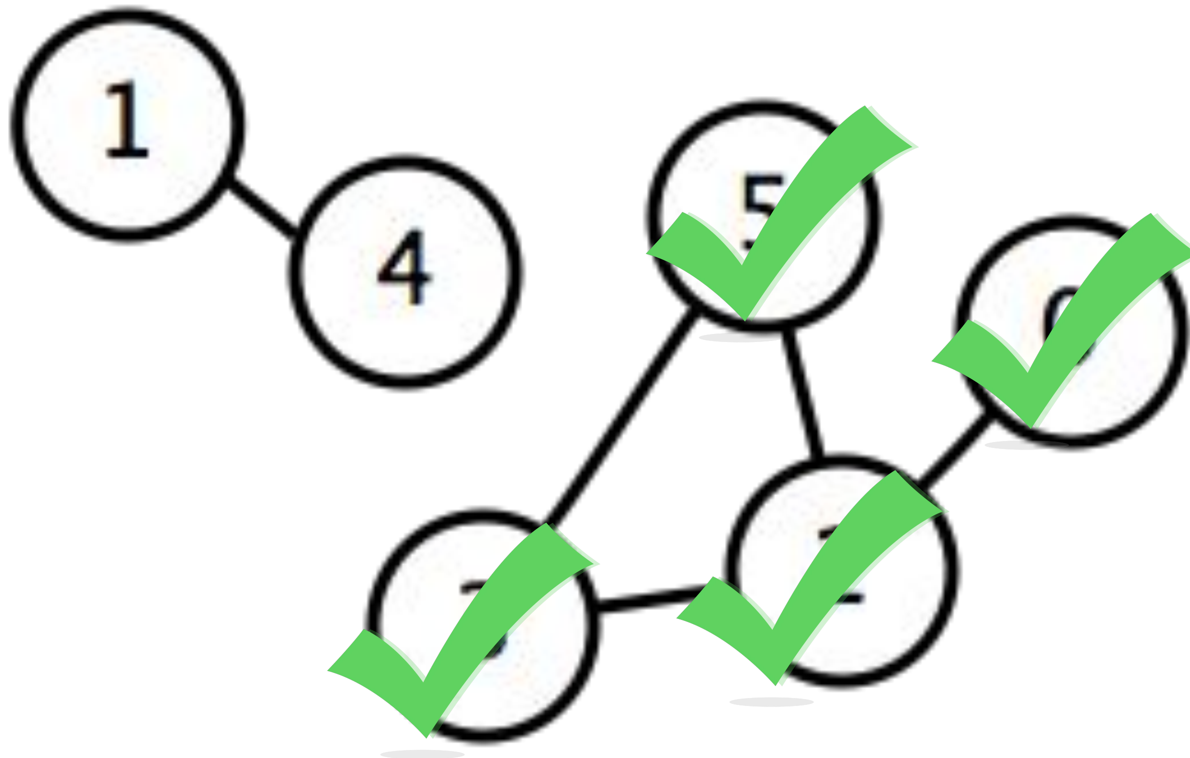
Grafos | Componentes Conexas



Grafos | Componentes Conexas

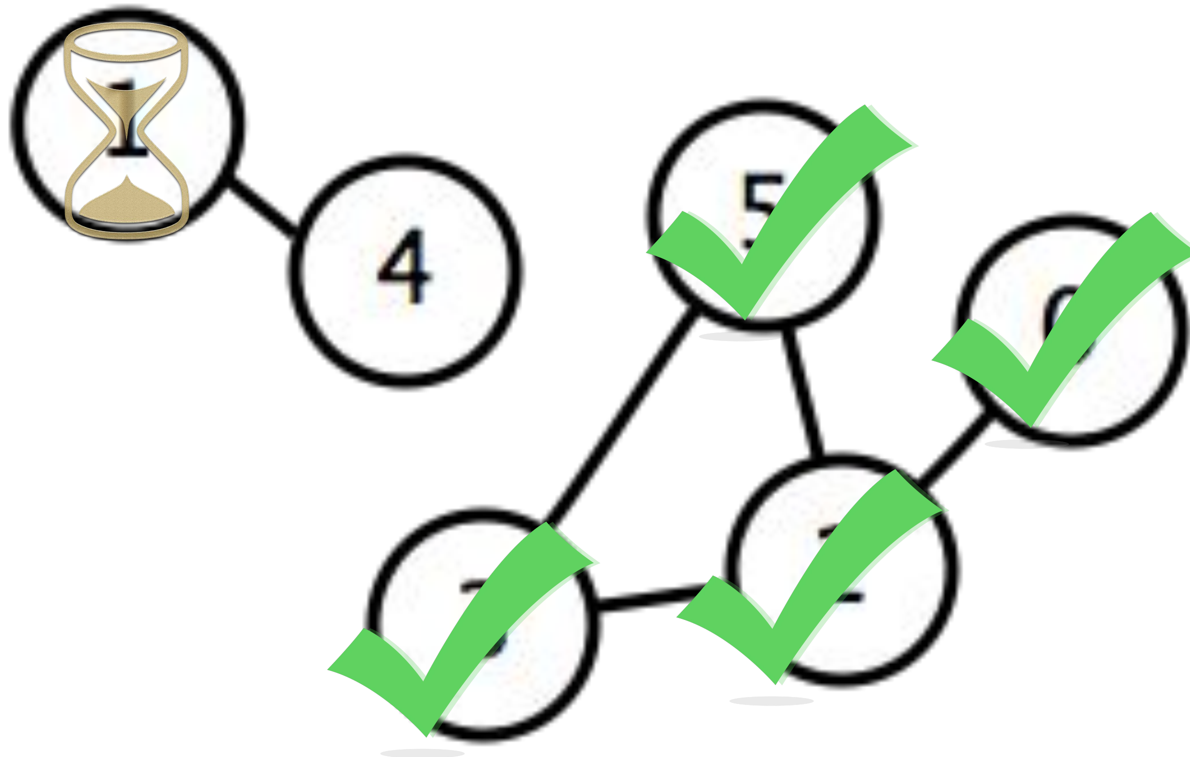


Grafos | Componentes Conexas

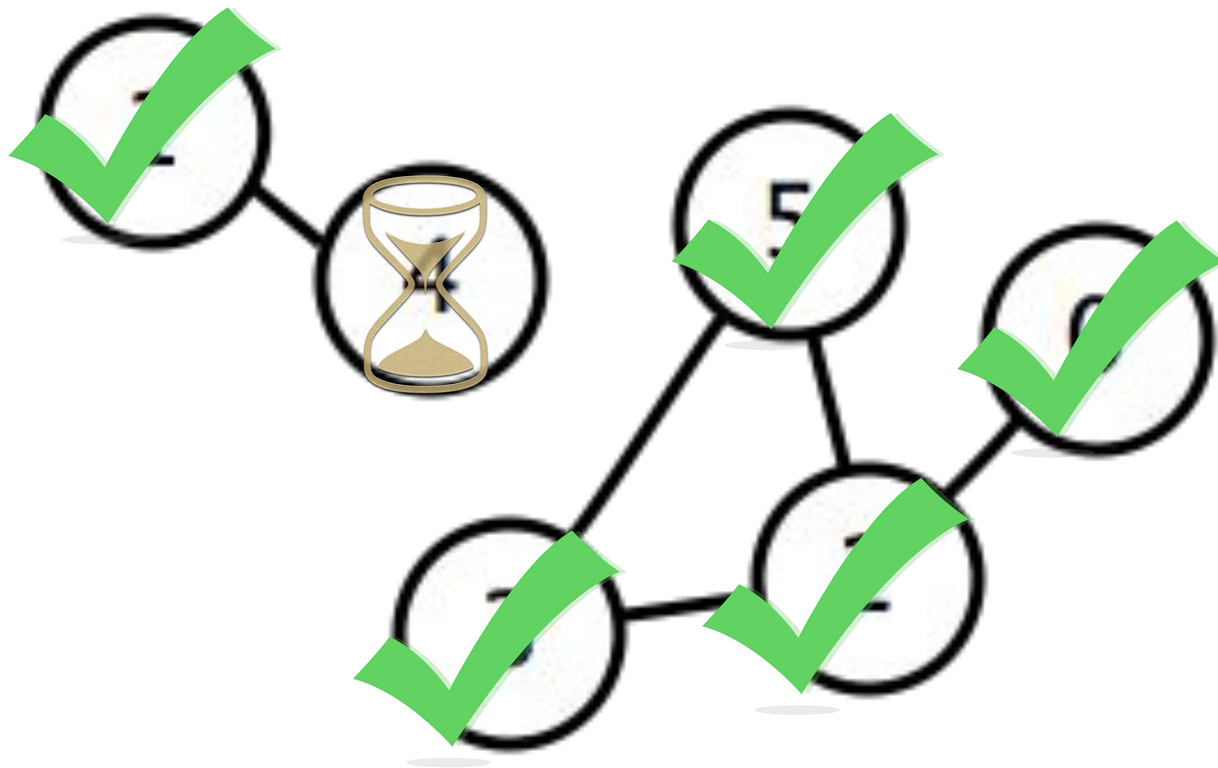


Grafos | Componentes Conexas

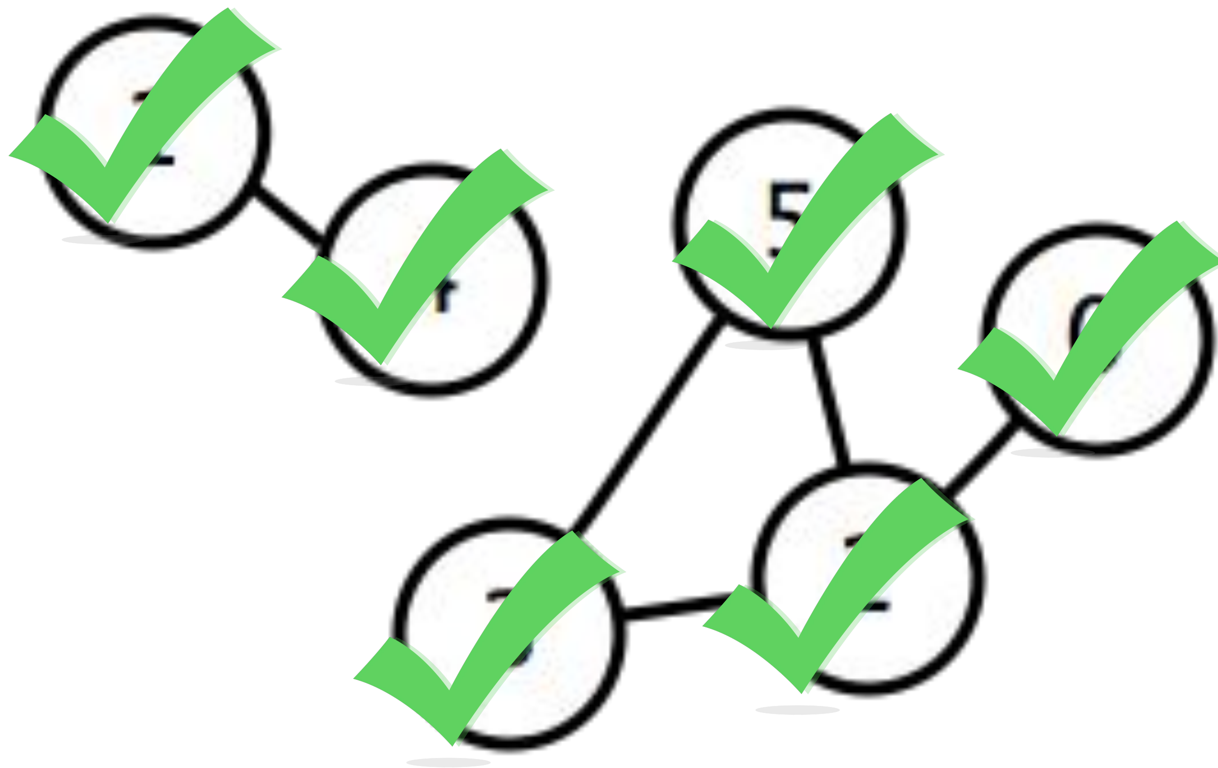
+1 Componente Conexa (2)



Grafos | Componentes Conexas



Grafos | Componentes Conexas



Grafos | Componentes Conexas

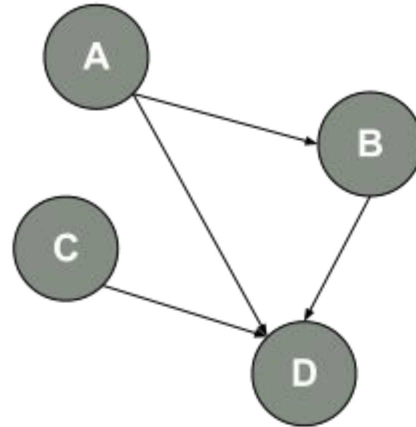
- Se empieza desde 0 y se recorre:
  - 0-2-5-3
- Al estar 1 no visitado se recorre los adyacentes de 1
  - 1-4
- Al estar 2,3,4,5 no visitado se ignoran
- Hay dos componentes conexas

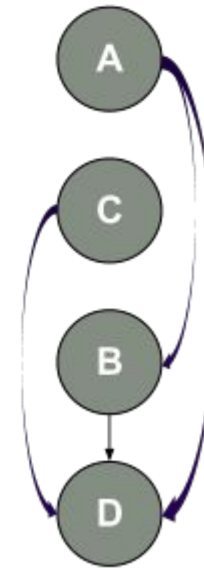
Grafos | Ordenamiento Topológico

- 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

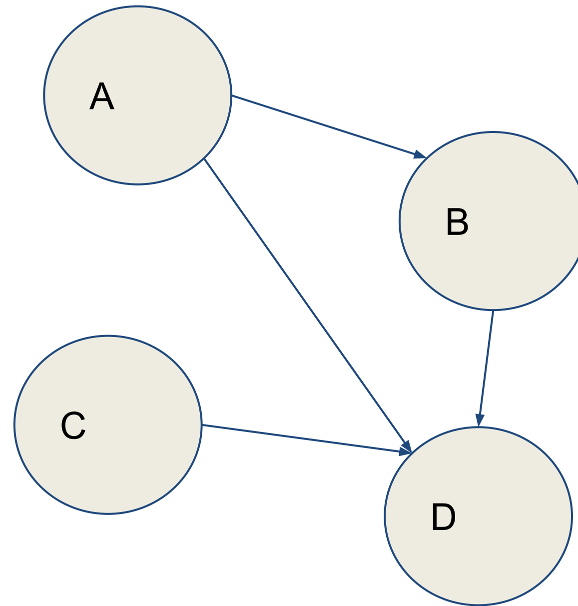
DAG AND ITS TOPOLOGICAL SORT





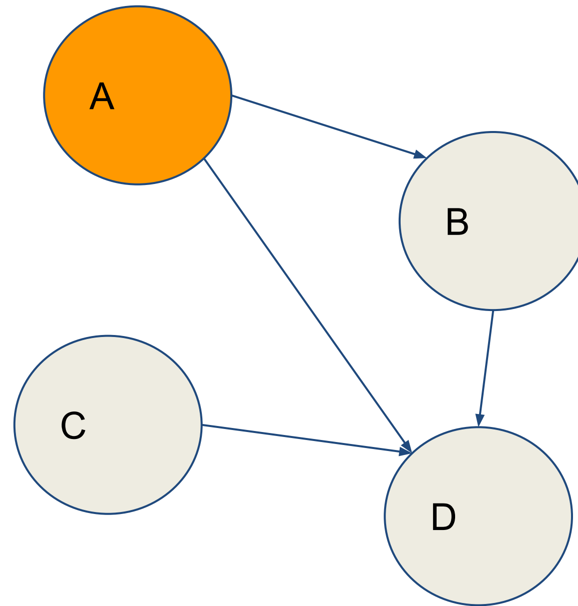
The above graph can be "sorted" as follows: [A, C, B, D]. That means that if the edge (A, B) exists, A must precede B in the final order!

Grafos | Ordenamiento Topológico



A y C no les incide ningún otro nodo

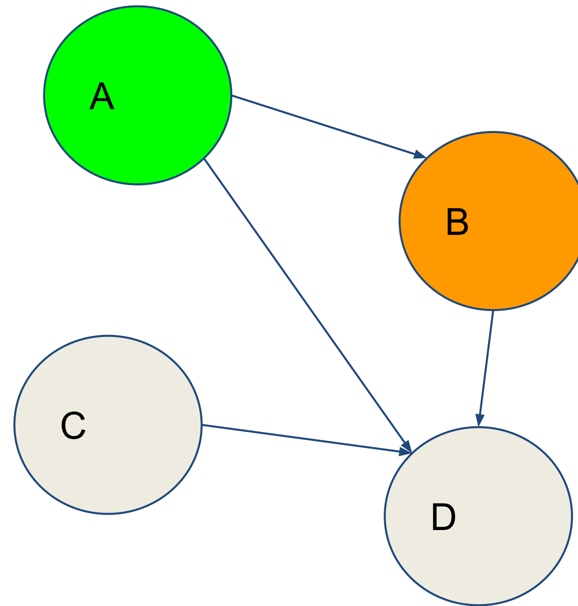
Grafos | Ordenamiento Topológico



TS = { }

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

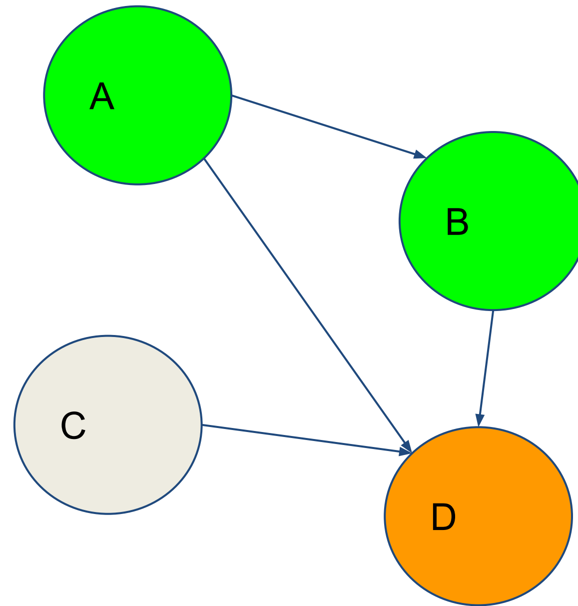
Grafos | Ordenamiento Topológico



TS = { }

DFS(B) = DFS(D)

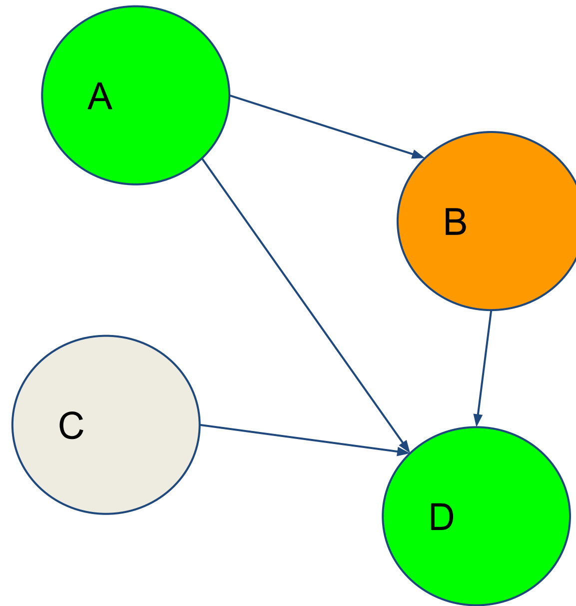
Grafos | Ordenamiento Topológico



TS = {D}

DFS(D) = ø, TS.push(D)

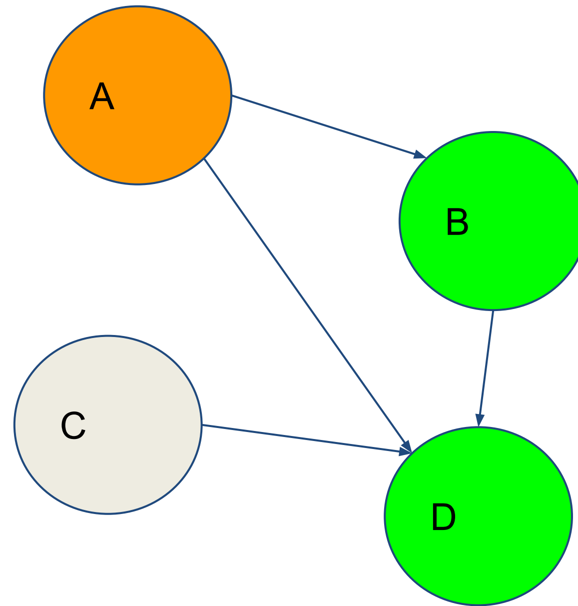
Grafos | Ordenamiento Topológico



TS = { B, D}

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

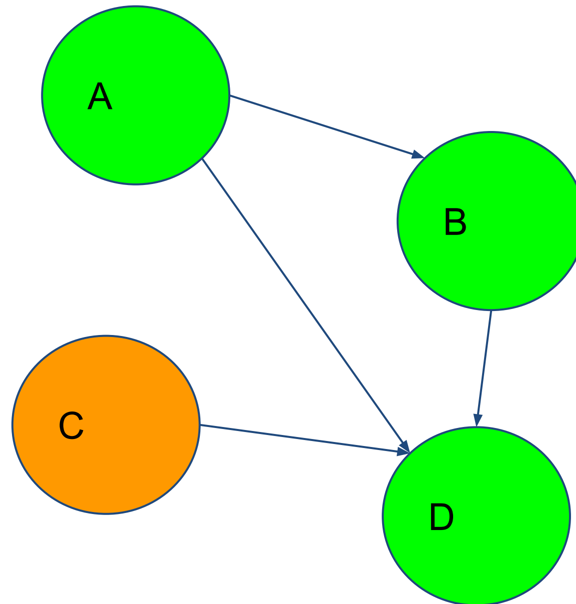
Grafos | Ordenamiento Topológico



TS = { A, B, D}

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

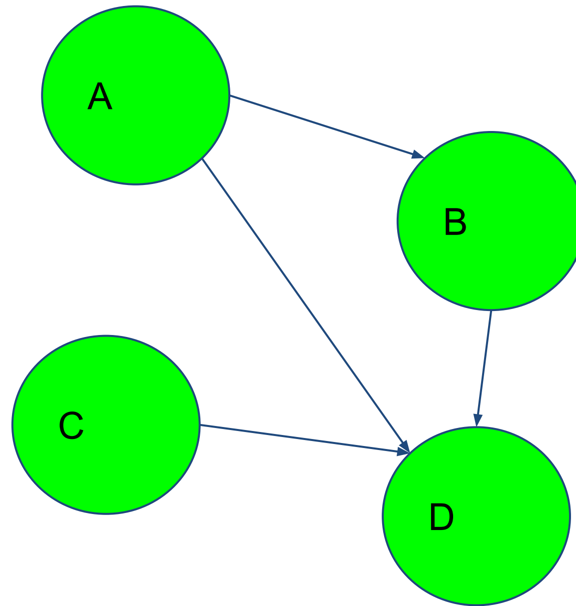
Grafos | Ordenamiento Topológico



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 => {C,A,B,D} ó {A,C,B,D}

Grafos | Ordenamiento Topológico

- Pseudocódigo

```
TopoSort(n):
  si v(n) ret ∅
  v(n) = 1
  for edge in edges(n):
    TopoSort(edge)
  TS.push(n)
  ret ∅
```

Grafos | Ordenamiento Topológico

- Pseudocódigo
- Siendo TS una pila
- Siendo 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

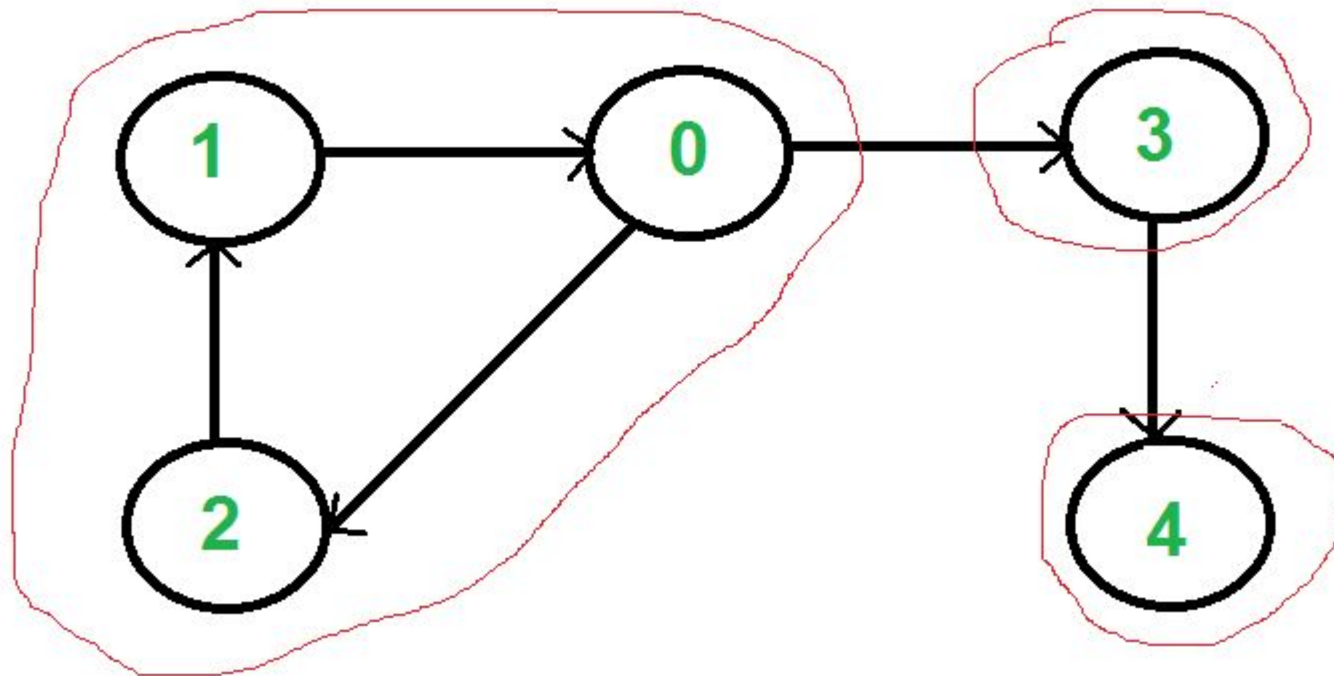
Grafos | Componentes Fuertemente Conexas

- 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

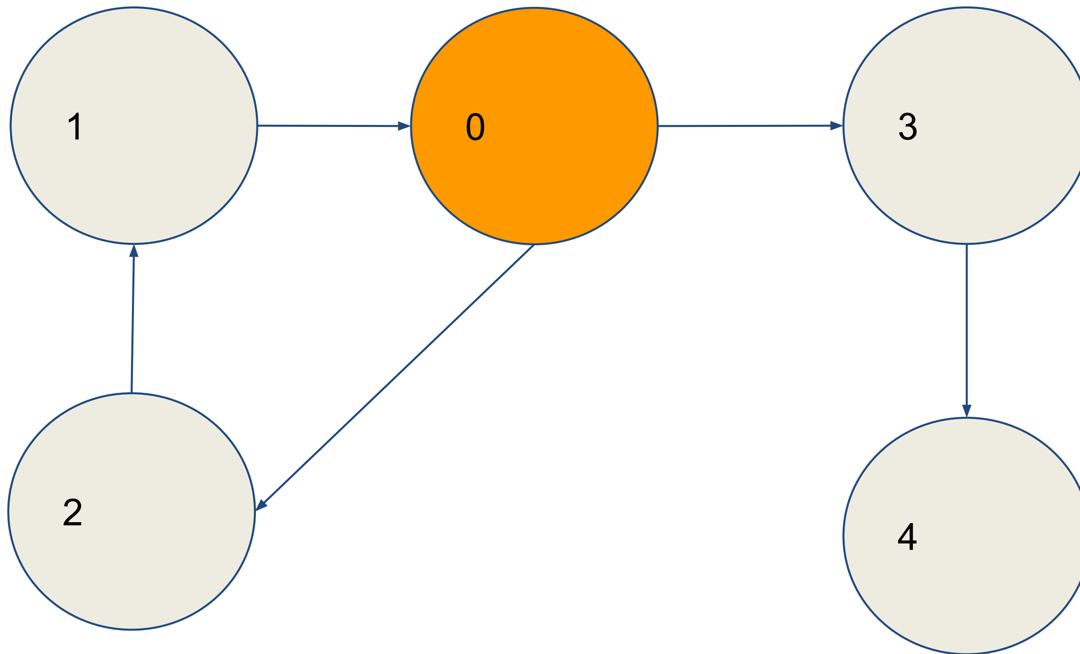
Grafos | Componentes Fuertemente Conexas

- 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



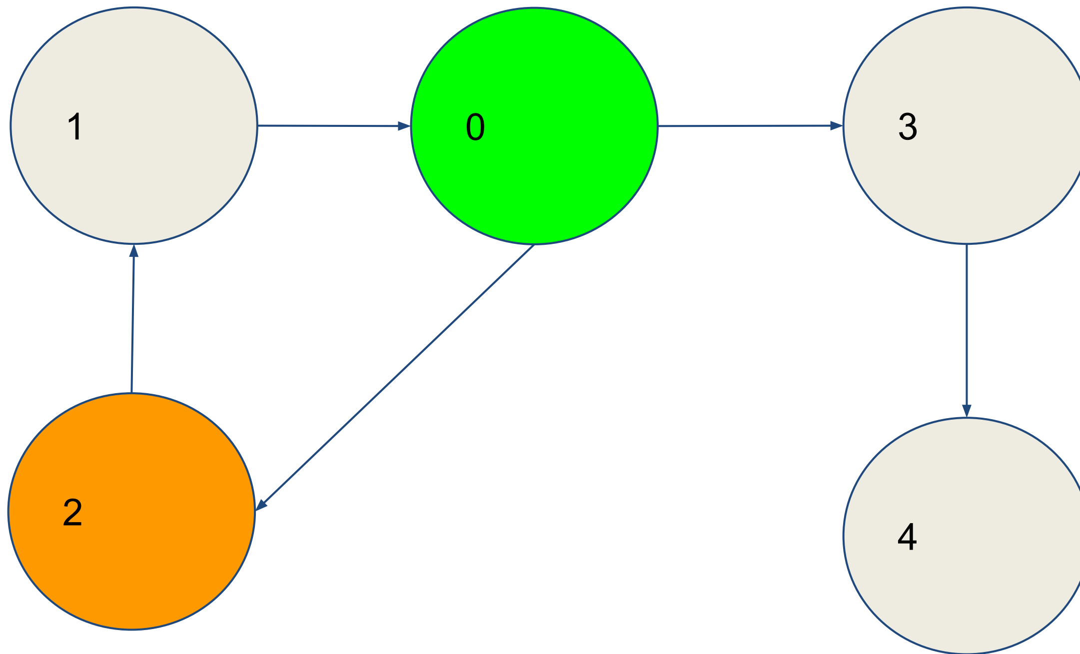
Grafos | Componentes Fuertemente Conexas



TS = { }

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

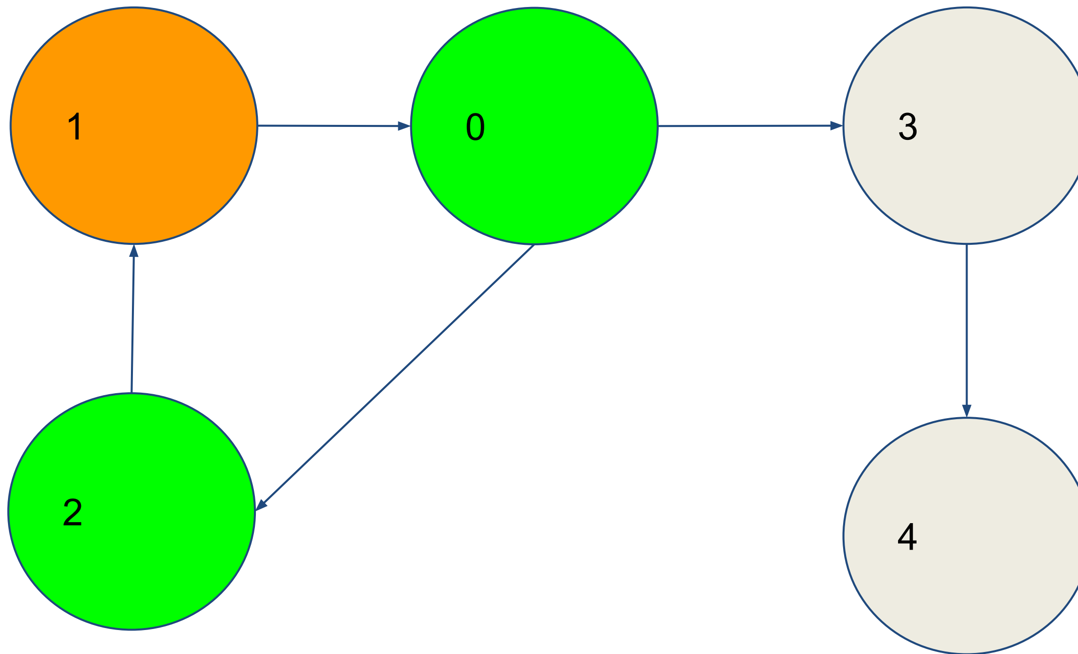
Grafos | Componentes Fuertemente Conexas



TS = { }

DFS(2) => DFS(1)

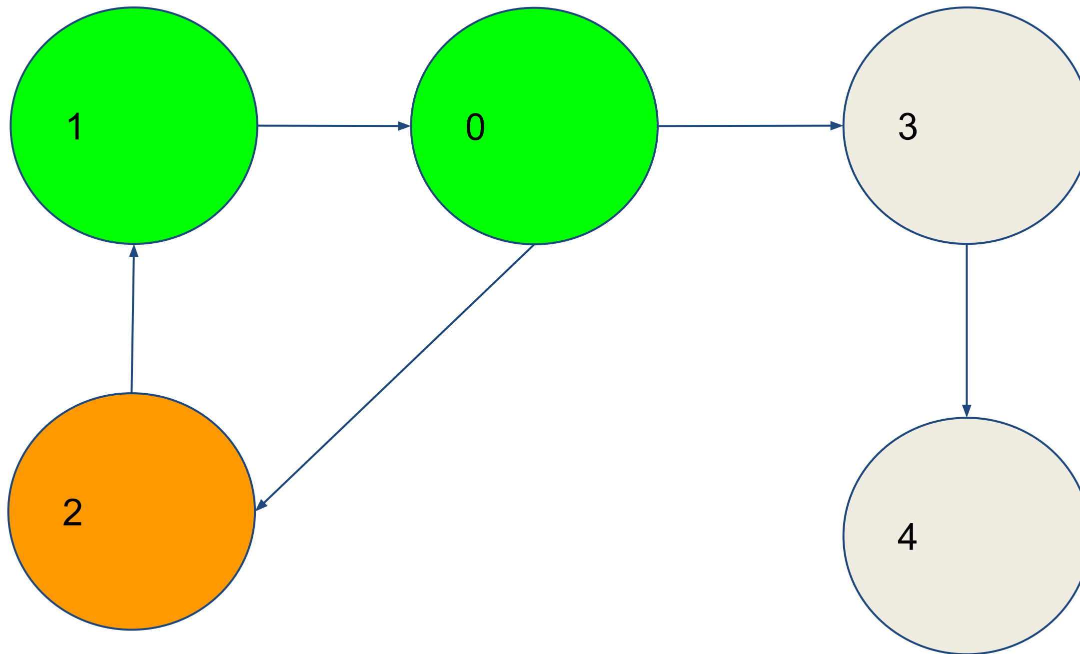
Grafos | Componentes Fuertemente Conexas



TS = { 1 }

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

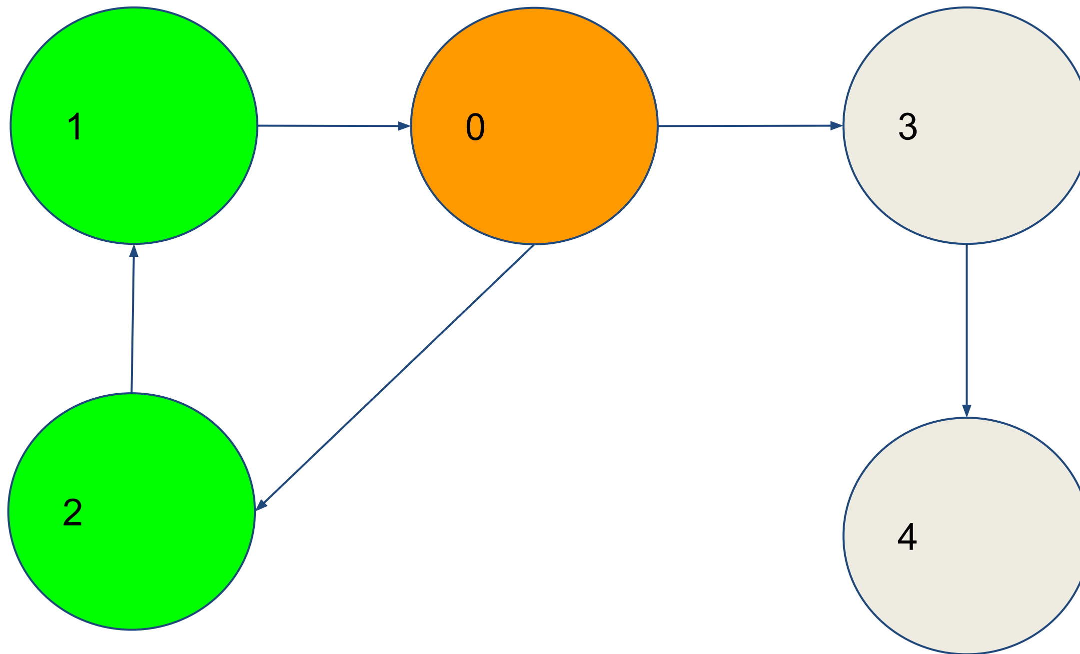
Grafos | Componentes Fuertemente Conexas



TS = {
2, 1
}

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

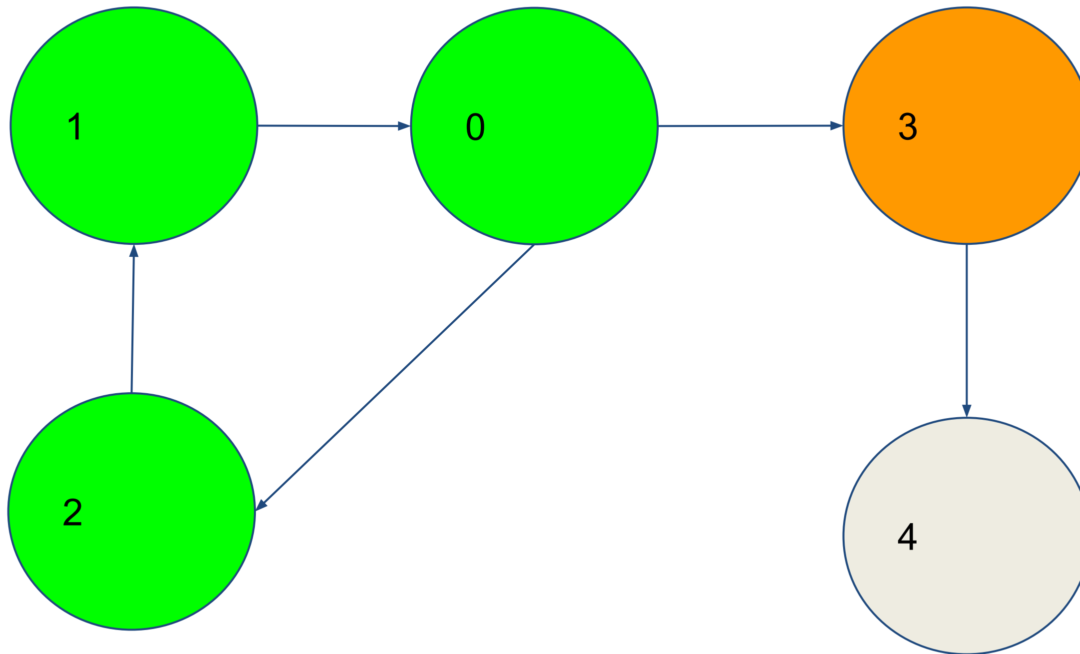
Grafos | Componentes Fuertemente Conexas



TS = {
2, 1
}

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

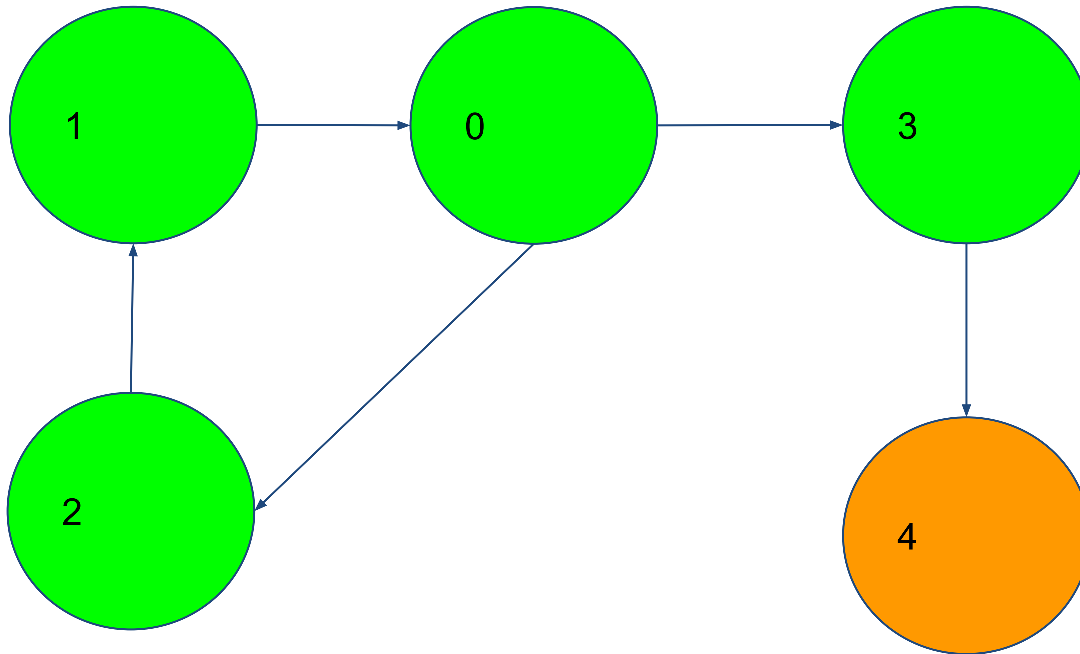
Grafos | Componentes Fuertemente Conexas



TS = {
2, 1
}

DFS(3) => DFS(4)

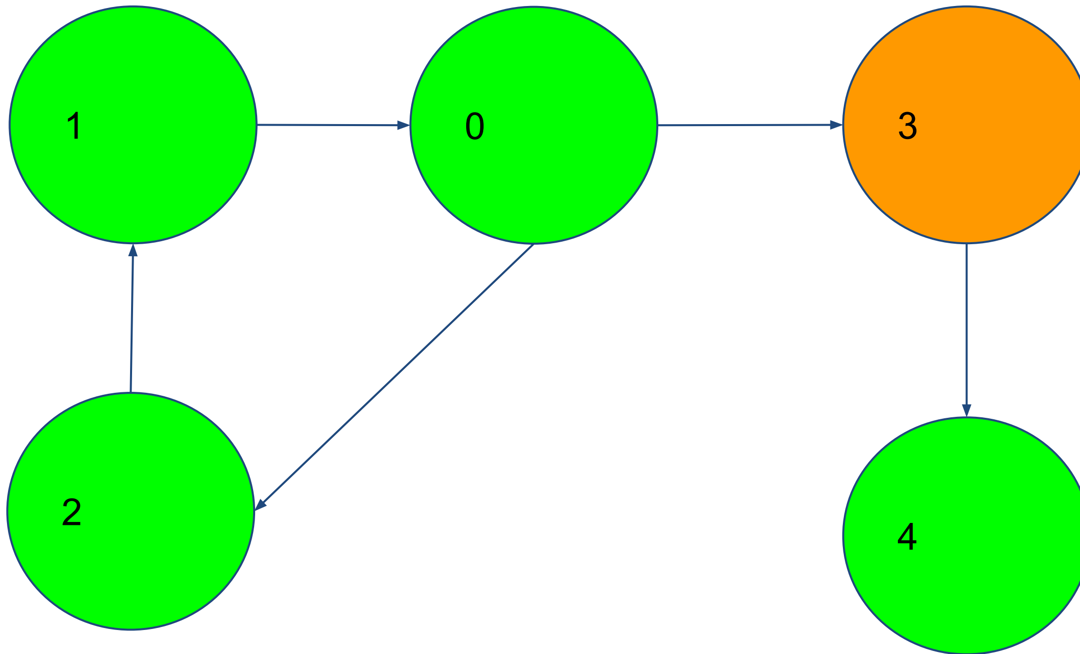
Grafos | Componentes Fuertemente Conexas



TS = {
4, 2, 1
}

DFS(4) => ø, TS.push(4)

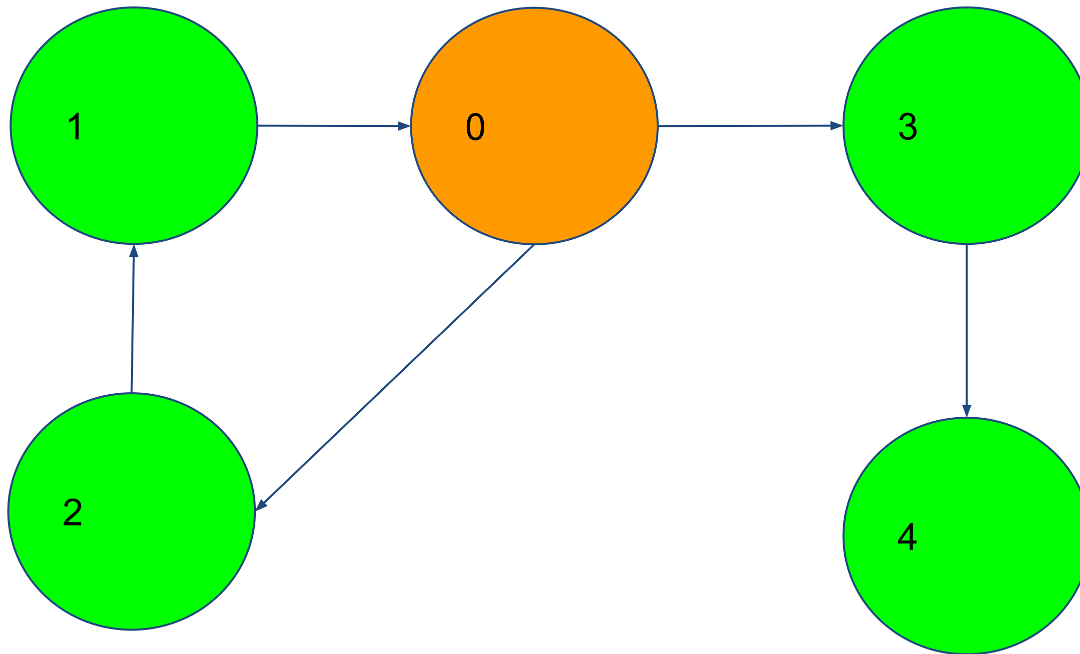
Grafos | Componentes Fuertemente Conexas



TS = {
3, 4, 2, 1
}

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

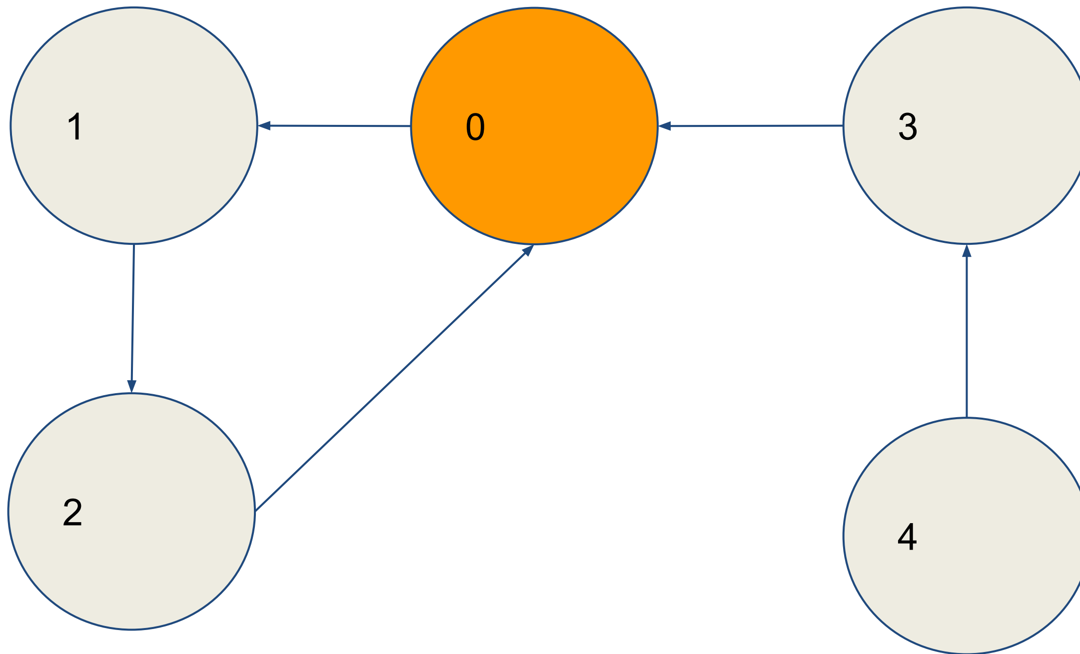
Grafos | Componentes Fuertemente Conexas



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

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

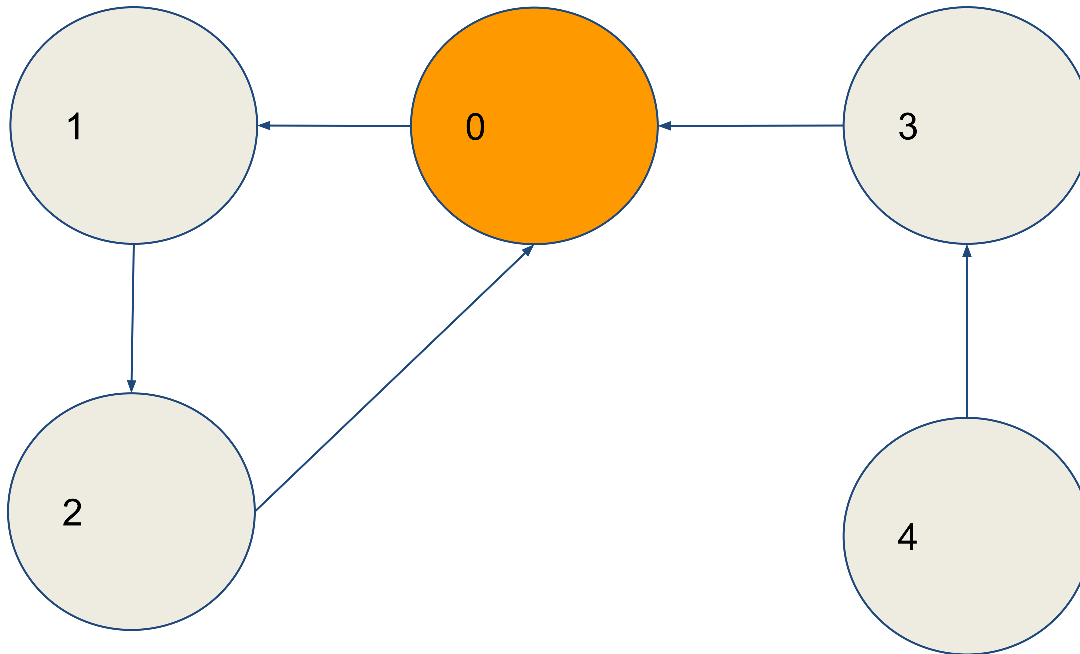
Grafos | Componentes Fuertemente Conexas



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

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

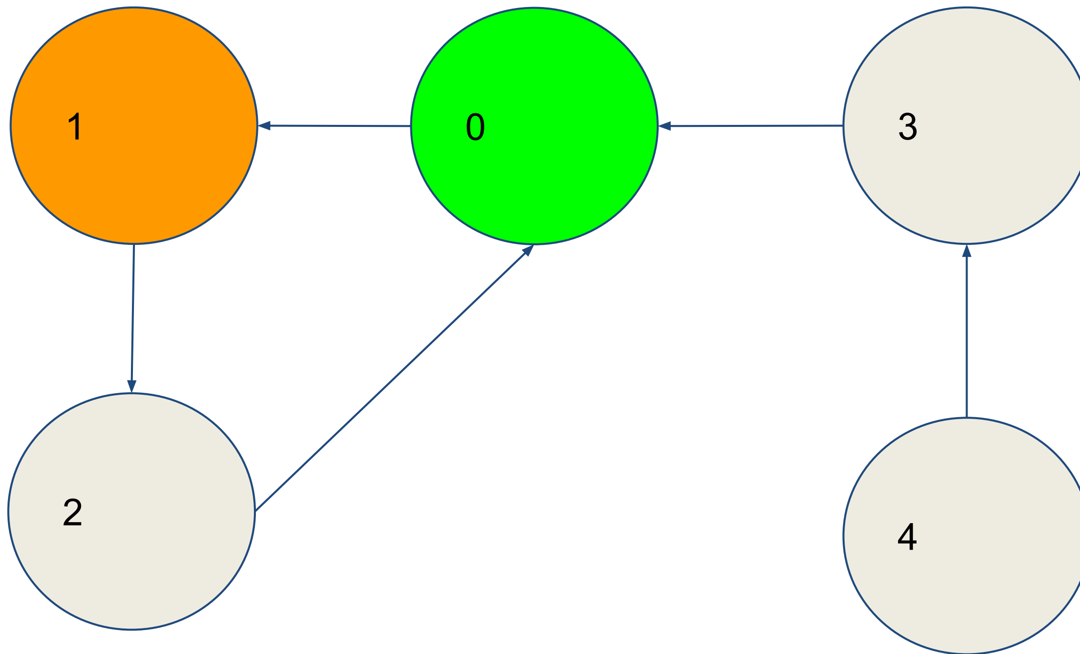
Grafos | Componentes Fuertemente Conexas



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

DFS(0), SCC+1

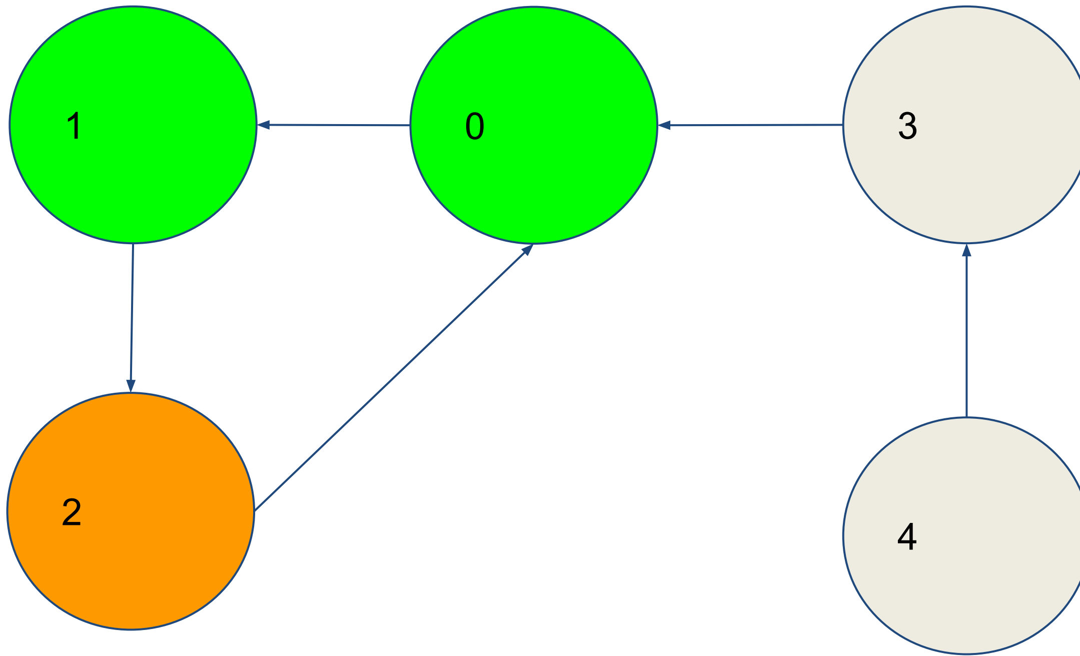
Grafos | Componentes Fuertemente Conexas



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

DFS(1)

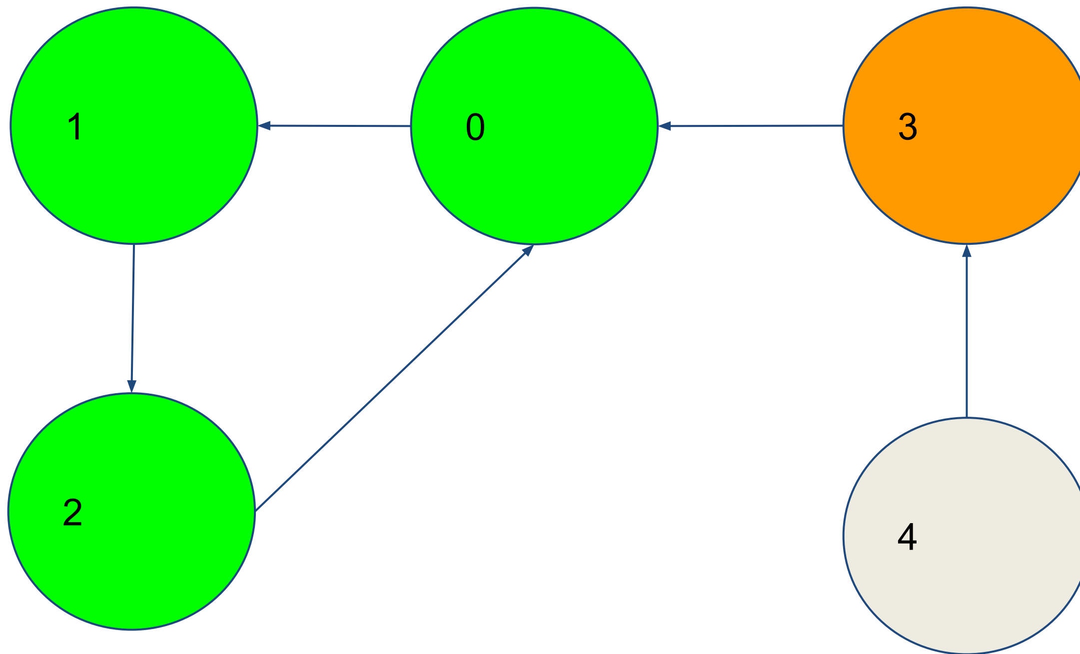
Grafos | Componentes Fuertemente Conexas



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

DFS(2)

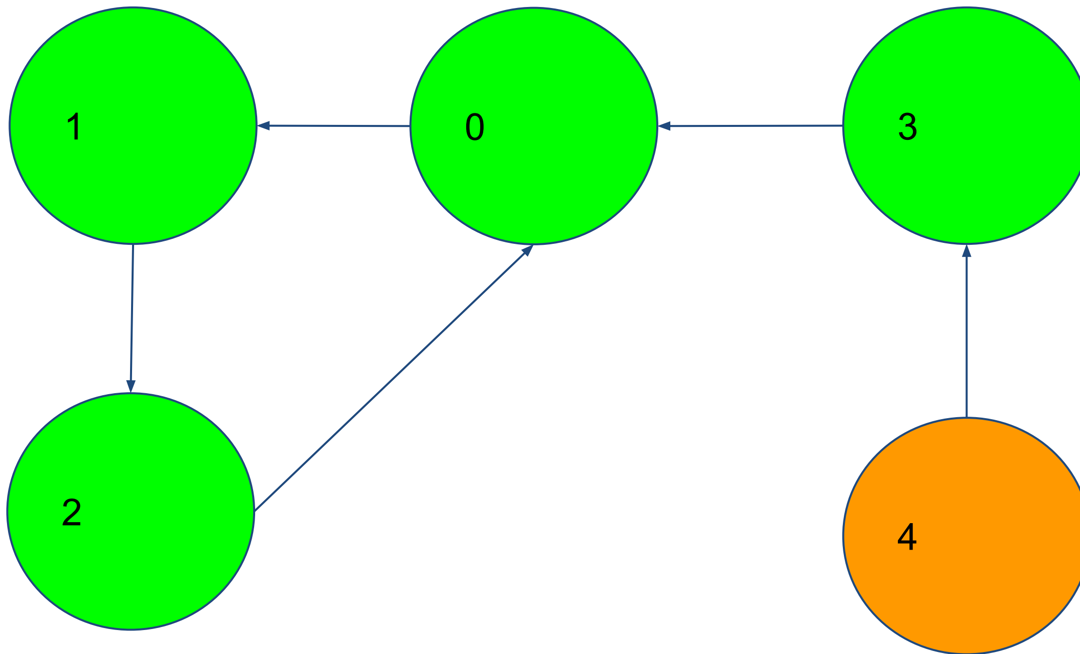
Grafos | Componentes Fuertemente Conexas



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

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

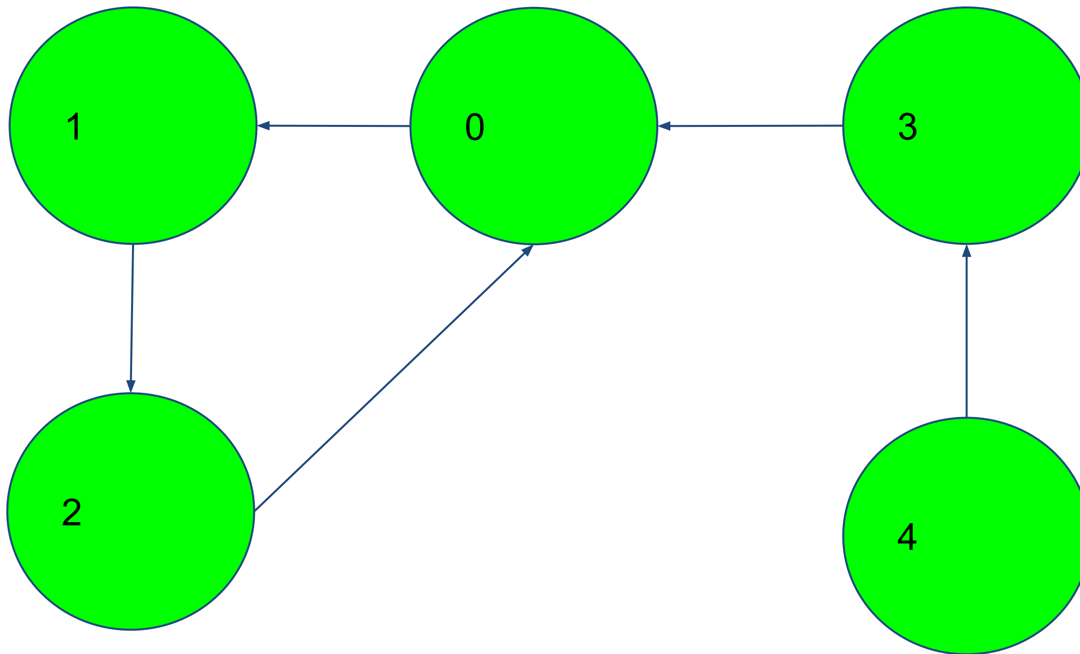
Grafos | Componentes Fuertemente Conexas



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

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

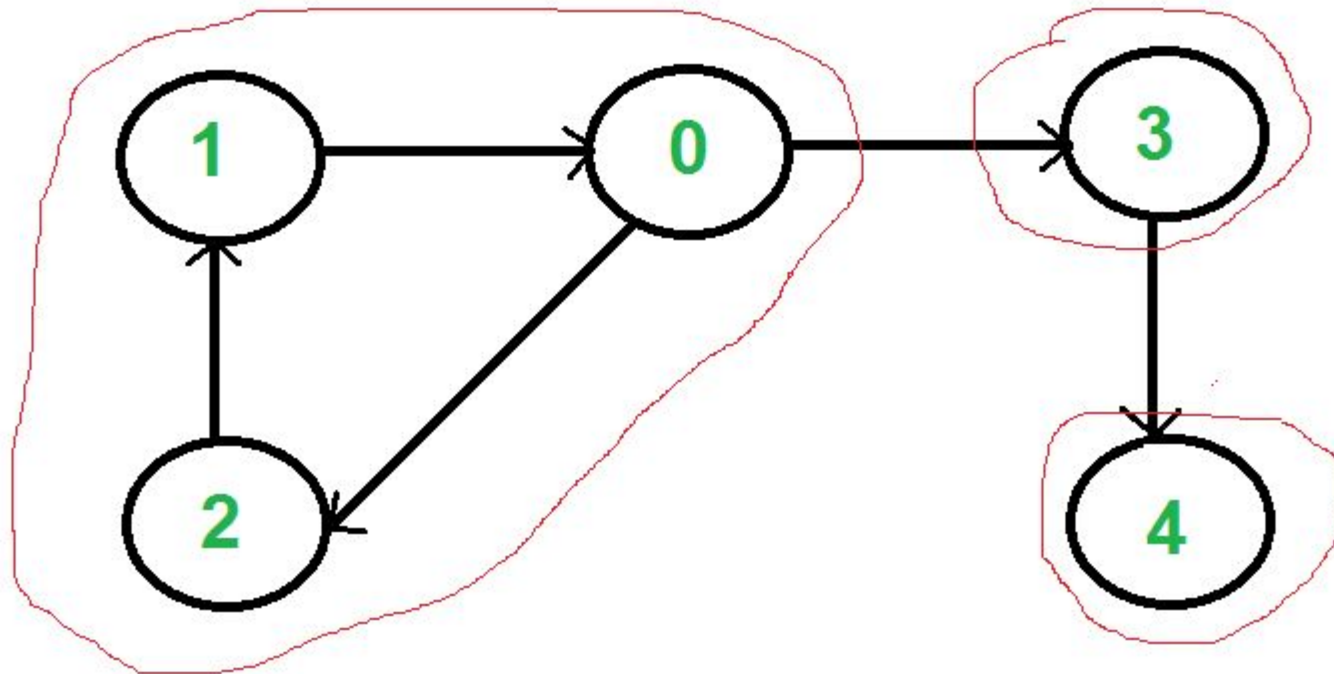
Grafos | Componentes Fuertemente Conexas



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 -> Toposort = [1,2,4,3,0]
  - Gr[1] => [1,2,0] (+1)
  - Gr[2] => [] (+0)
  - Gr[3] => [0(x)] (+1)
  - Gr[4] => [3(x)] (+1)
- 3 componentes fuertemente conexas

Semana que viene...

- Grafos (parte II)
  - Algoritmos de distancia mínima (floyd-warshall, dijkstra, bellman-ford)
  - Árboles de recubrimiento (Prim, Kruskal)
- Programación Dinámica (parte I)
  - Recursión
  - Backtracking
  - Memorización

¡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 (sergioperezp1995@gmail.com)