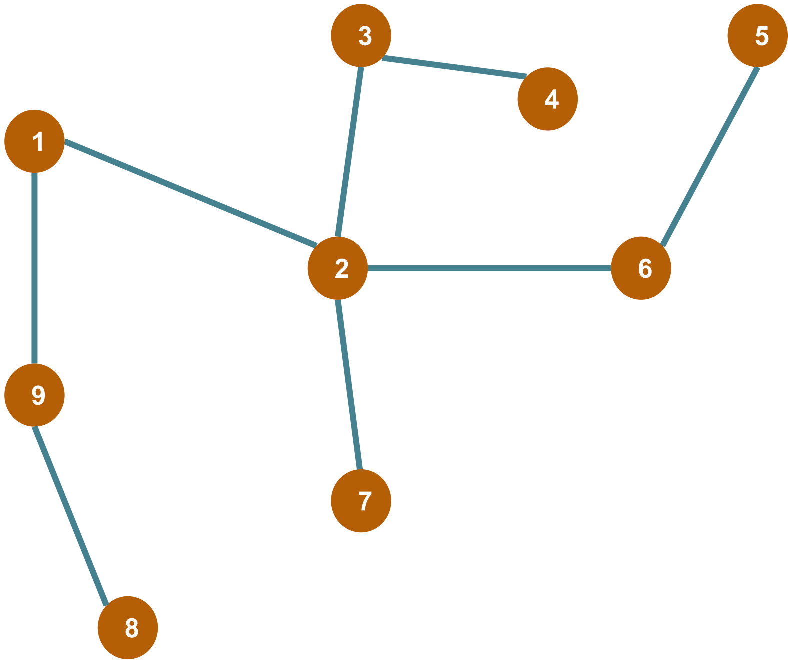
Quiz

AED3 > Clase 5 > Árboles

Árboles

Definición 1:

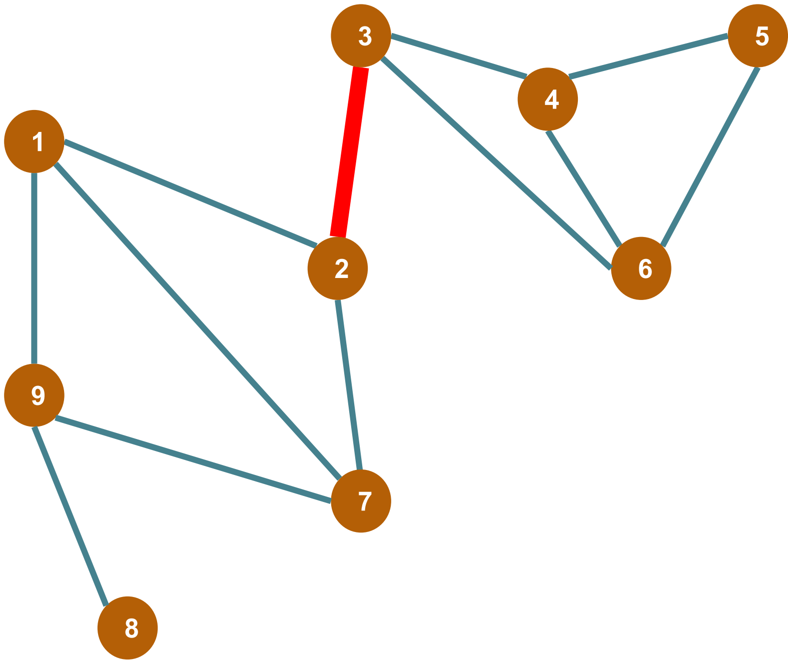
Un **árbol** es un grafo conexo sin circuitos simples.



Árboles

Definición 2:

Una arista *e* de *G* es un **puente** si *G* - *{e}* tiene más componente que G. Es decir, si la saco desconecta.



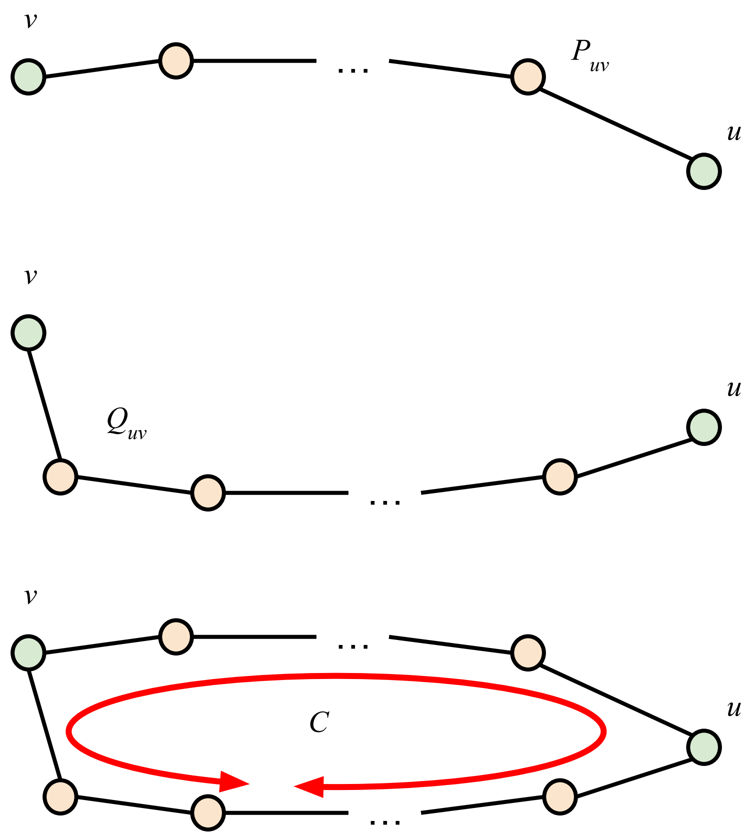
Árboles

Lema 1:

La unión entre dos caminos simples distintos entre u y v contiene un circuito simple.

P_{uv} , Q_{uv}: Caminos simples

$C : P_{uv} + Q_{vu}$: Circuito simple

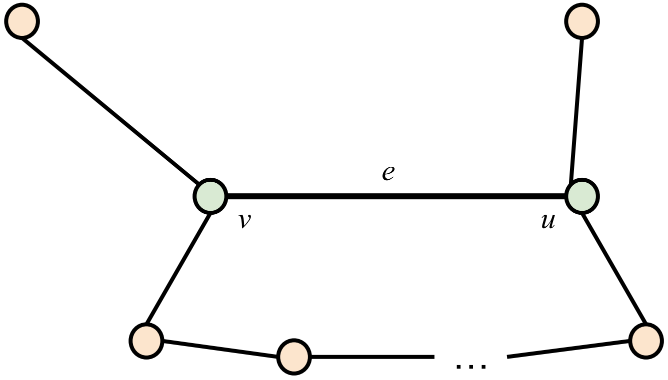


Árboles

Lema 2: Sea $G = (V, E)$ un grafo conexo y $e=(v,u) \in E$.

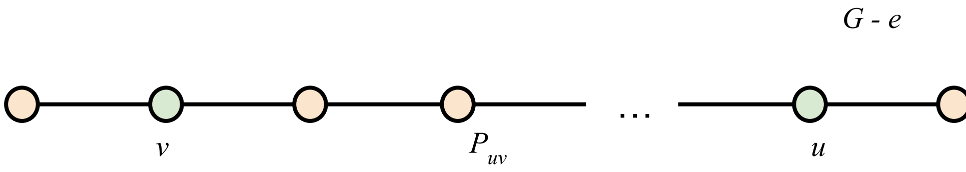
G - $e = (V, E \setminus \{e\})$ es conexo $\Leftrightarrow$ $e \in C$: circuito simple de G.

($e=(v,u) \in E$ es puente $\Leftrightarrow$ e no pertenece a un circuito simple de G).



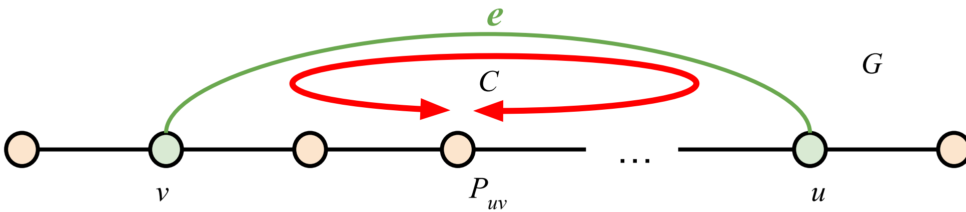
Demostración ($\Rightarrow$)**:**

G - e es conexo $\Rightarrow$



Existe un camino simple entre u y v (P_{uv}) que no usa e.

Si agrego e se forma un circuito simple C : $P_{uv} + e$

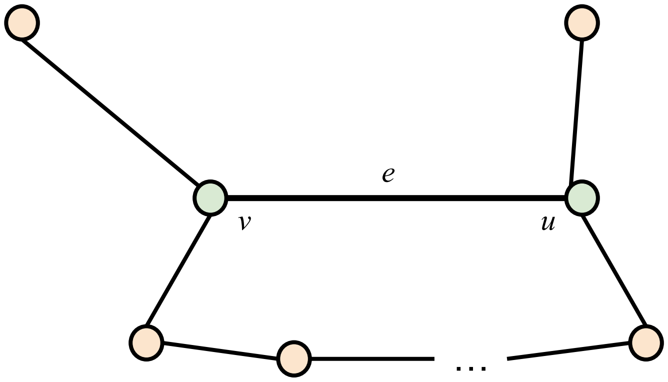


Árboles

Lema 2: Sea $G = (V, E)$ un grafo conexo y $e=(v,u) \in E$.

$G - e = (V, E \setminus \{e\})$ es conexo $\Leftrightarrow$ $e \in C$: circuito simple de G.

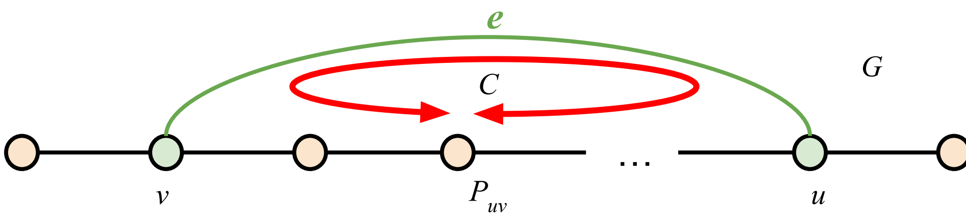
($e=(v,u) \in E$ es puente $\Leftrightarrow$ e no pertenece a un circuito simple de G).



Demostración ($\Leftarrow$):

Sea C un circuito simple que contiene a $e=(u, v)$ $\Rightarrow$

$C : P_{uv} + e$, tq P_{uv} no usa e.

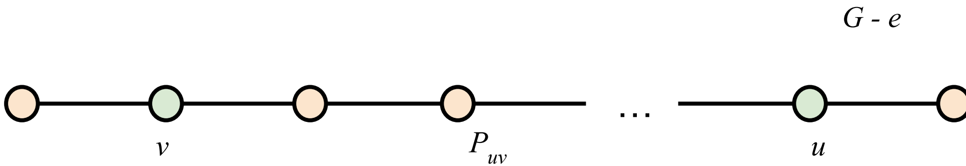


G es conexo $\Rightarrow$ Existe un camino entre todo par de vértices.

Si no usa e lo conservo en G-e .

Si usa e , la reemplazo por P_{uv} en G-e. $\Rightarrow$

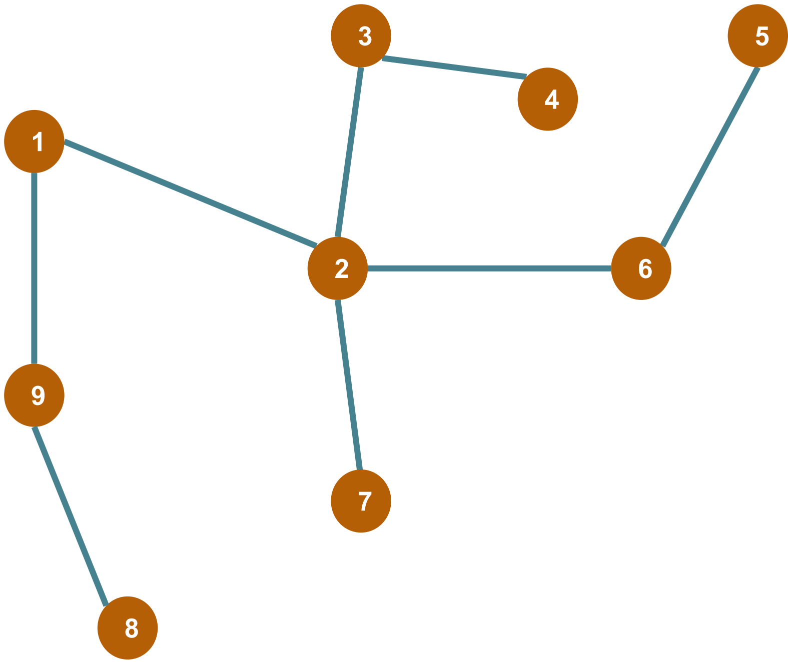
Existe un camino entre todo par de vértices en G-e. $\Rightarrow$ G-e es conexo



Árboles

Teorema: Equivalencias

1. *G* es un árbol *(grafo conexo sin circuitos simples)*.
2. *G* es un grafo sin circuitos simples y *e* una arista tq $e \notin E$. $G+e = (V, E+\{e\})$ tiene exactamente un circuito simple, y ese circuito contiene a e. *Es decir, si agrego una arista cualquiera se forma un ciclo.*
3. $\exists$ exactamente un camino simple entre todo par de nodos.
4. *G* es conexo, pero si se quita cualquier arista queda un grafo no conexo. *Es decir, si saco cualquier arista se desconecta, o toda arista es puente.*



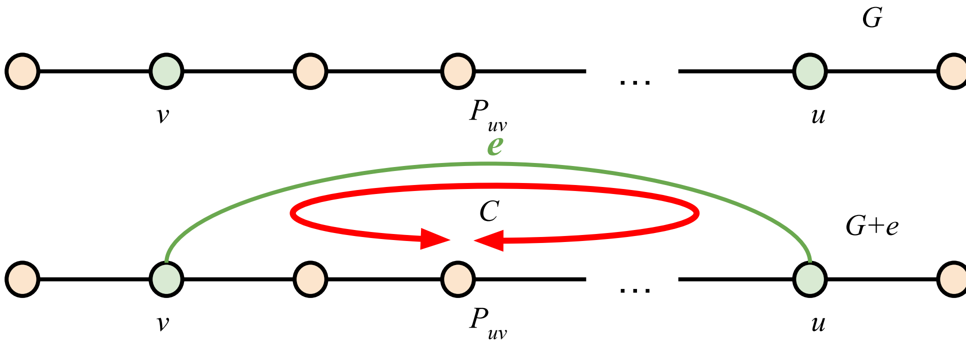
Árboles

1 ⇒ 2) G es un árbol *(grafo conexo sin circuitos simples)*. ⇒ G es un grafo sin circuitos simples y e una arista tq $e \notin E$. $G+e = (V, E+\{e\})$ tiene exactamente un circuito simple, y ese circuito contiene a e. *Es decir, si agrego una arista cualquiera se forma un ciclo.*

Demostración (1 ⇒ 2):

Como G es conexo ⇒ Existe algún camino P_{uv} entre u y v.

Como $G+e$ es conexo ⇒ Existe algún circuito C: $P_{uv} + e$.



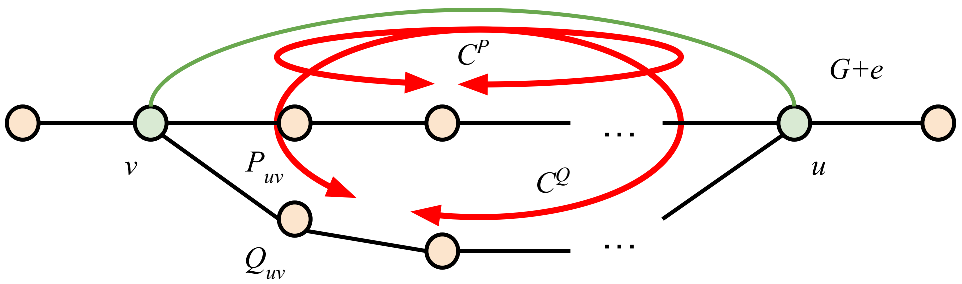
Árboles

1 ⇒ 2) G es un árbol *(grafo conexo sin circuitos simples)*. ⇒ G es un grafo sin circuitos simples y e una arista tq $e \notin E$. $G+e = (V, E+\{e\})$ tiene exactamente un circuito simple, y ese circuito contiene a e. *Es decir, si agrego una arista cualquiera se forma un ciclo.*

Demostración (1 ⇒ 2):

Como G es conexo ⇒ Existe algún camino P_{uv} entre u y v.

Como $G+e$ es conexo ⇒ Existe algún circuito C: $P_{uv} + e$.

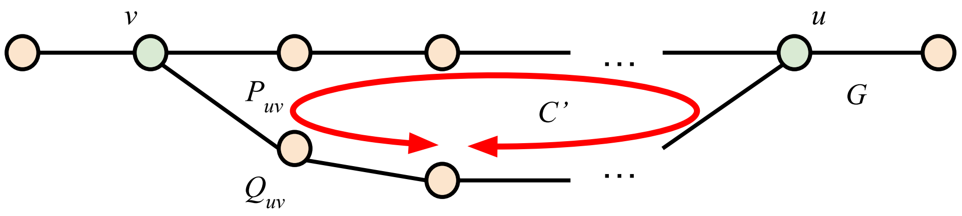


¿Por no existen más?

Supongo que existen dos C^P: $P_{uv} + e$ y C^Q: $Q_{uv} + e$ en $G+e$

⇒ Existe algún circuito C': $P_{uv} + Q_{vu}$ en $G+e$ no usa e

⇒ Existe algún circuito C': $P_{uv} + Q_{vu}$ en G



¡Absurdo!

Árboles

2 ⇒ 3) G es un grafo sin circuitos simples y e una arista tq $e \notin E$. $G+e = (V, E+\{e\})$ tiene exactamente un circuito simple, y ese circuito contiene a e. *Es decir, si agrego una arista cualquiera se forma un ciclo.* ⇒ ∃ exactamente un camino simple entre todo par de nodos.

Demostración (2 ⇒ 3):

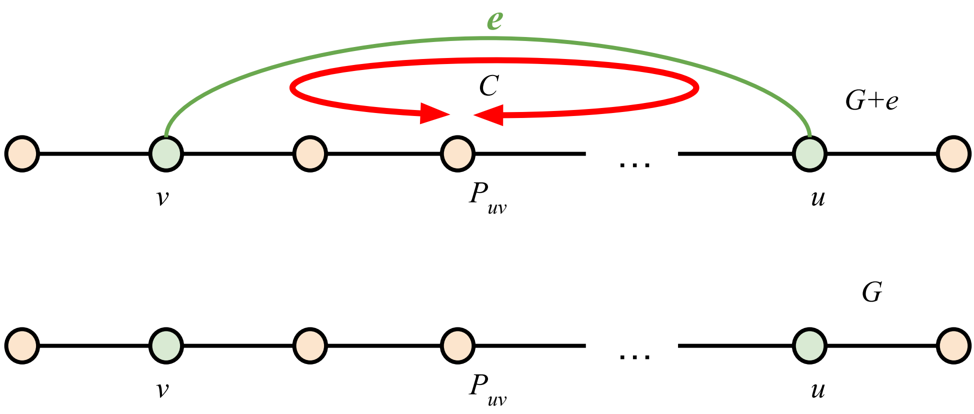
Existe algún circuito C: $P_{uv} + e$. en $G+e$

⇒ Existe algún camino P_{uv} entre u y v en $G+e-e$

⇒ Existe P_{uv} entre todo par de vértices

¿Por no existen más?

Ídem 1 ⇒ 2



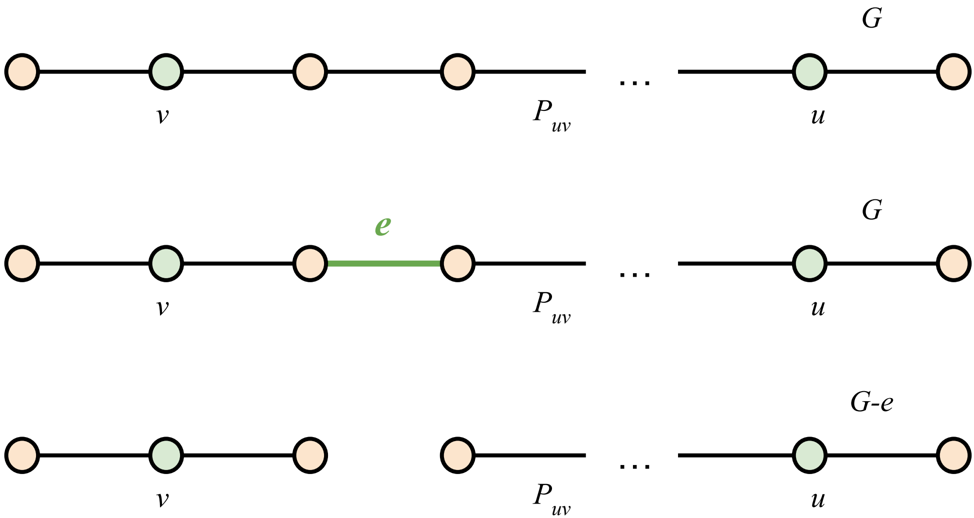
Árboles

3 ⇒ 4) $\exists$ exactamente un camino simple entre todo par de nodos. ⇒ G es conexo, pero si se quita cualquier arista queda un grafo no conexo. *Es decir, si saco cualquier arista se desconecta, o toda arista es puente.*

Demostración (3 ⇒ 4):

Existe P_{uv} entre todo par de vértices ⇒ G es conexo

P_{uv} es único ⇒ Si saco cualquier arista $e \in P_{uv}$ se desconeta



Árboles

4 ⇒ 1) G es conexo, pero si se quita cualquier arista queda un grafo no conexo. *Es decir, si saco cualquier arista se desconecta, o toda arista es puente.*⇒ G es un árbol *(grafo conexo sin circuitos simples)*.

Demostración (4 ⇒ 1):

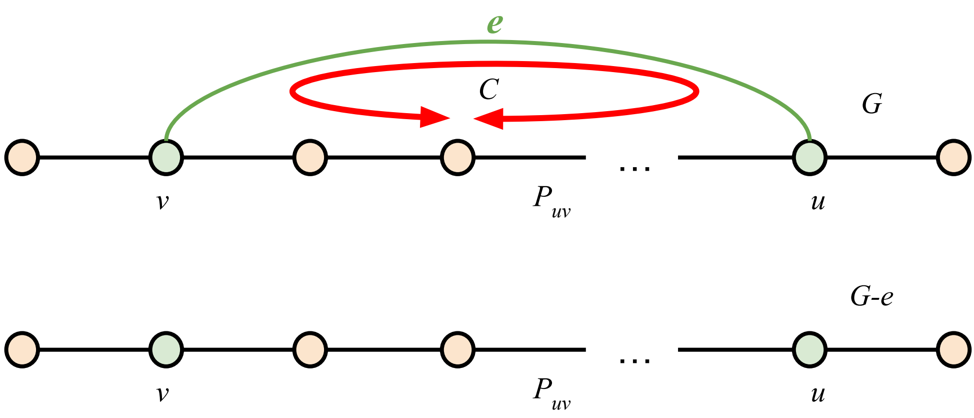
G es conexo

Si existe e tq $C : P_{uv} + e$ es circuito simple en G

⇒ Si saco e no se desconecta.

¡Absurdo!

⇒ G es conexo y sin circuitos simples (un árbol)



Árboles: Definiciones

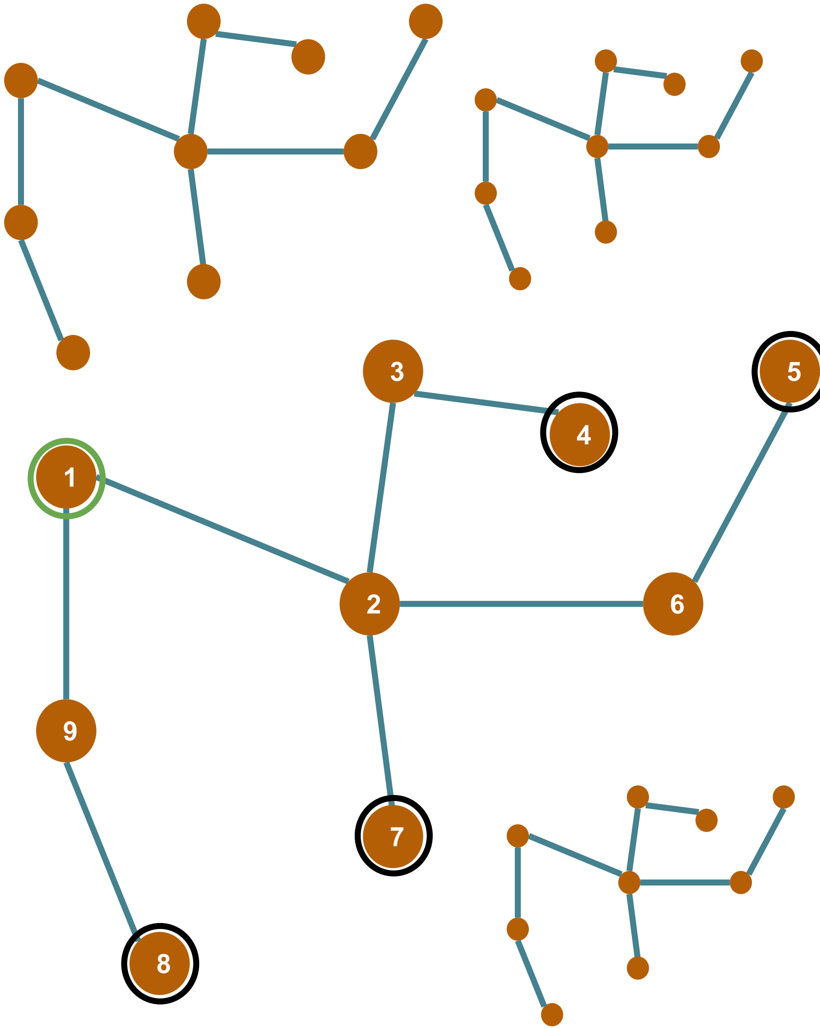
Árbol : *T*

Hoja: u tq $d(u) = 1$

Raíz: Algún vértice elegido

Bosque: Conjunto de árboles

Árbol trivial: T con $n=1$ y $m=0$



Árboles

Lema 3: Todo árbol no trivial tiene al menos dos hojas

Demostración:

$P : v_1 \ldots v_k$ es un camino simple <u>maximal</u> en T (no lo puedo extender más).

q.v.q. v_1 y v_k son hojas, es decir que $d(v_1) = 1$ y $d(v_k) = 1$

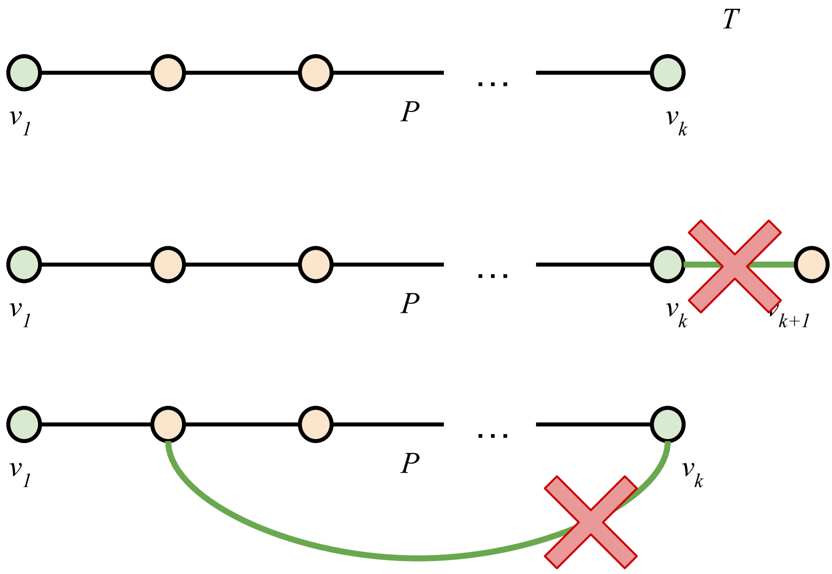
$d(v_k) > 0$ *porque conecta con* v_{k-1}

$d(v_k) > 1$??

No puedo agregar un vértice porque era maximal

No puedo ir a uno existente porque formo un circuito.

$\Rightarrow d(v_k) = 1$ *(idem* v_1 *)*



Árboles

Lema 4: Sea $G = (V, E)$ un árbol $\Rightarrow m = n - 1$

Demostración: Inducción en n.

Caso base: $n=1$ y $m=0$

Hipótesis inductiva: T' con k' vértices ($k' < k$) tiene $k' - 1$ aristas

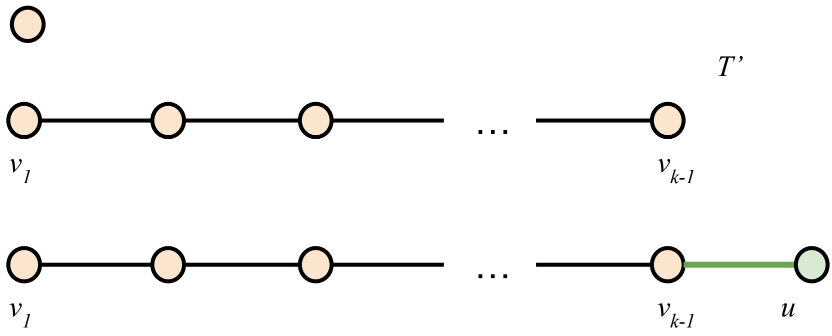
Paso inductivo: Sea u una hoja (sabemos que tiene por Lema 2).

$T' = T - u = (V \setminus \{u\}, E \setminus \{(u,v) \in E, \ \forall v \in V\}$

T' es conexo y sin circuitos con $k' = k - 1$ vértices $< k$

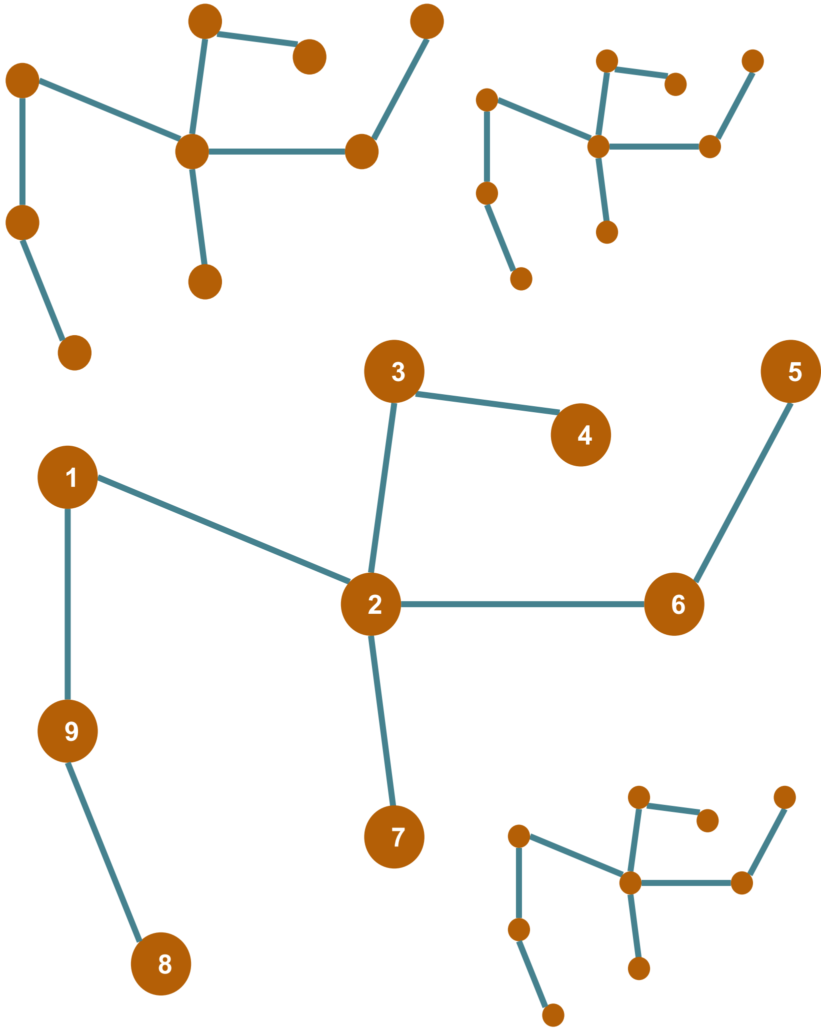
$\Rightarrow$ (Hip. ind.) tiene $k - 2$ aristas

Como u era una hoja $\Rightarrow d(u)=1 \ \Rightarrow T$ tiene $k - 2 + 1 = k - 1$ aristas



Árboles: Definiciones

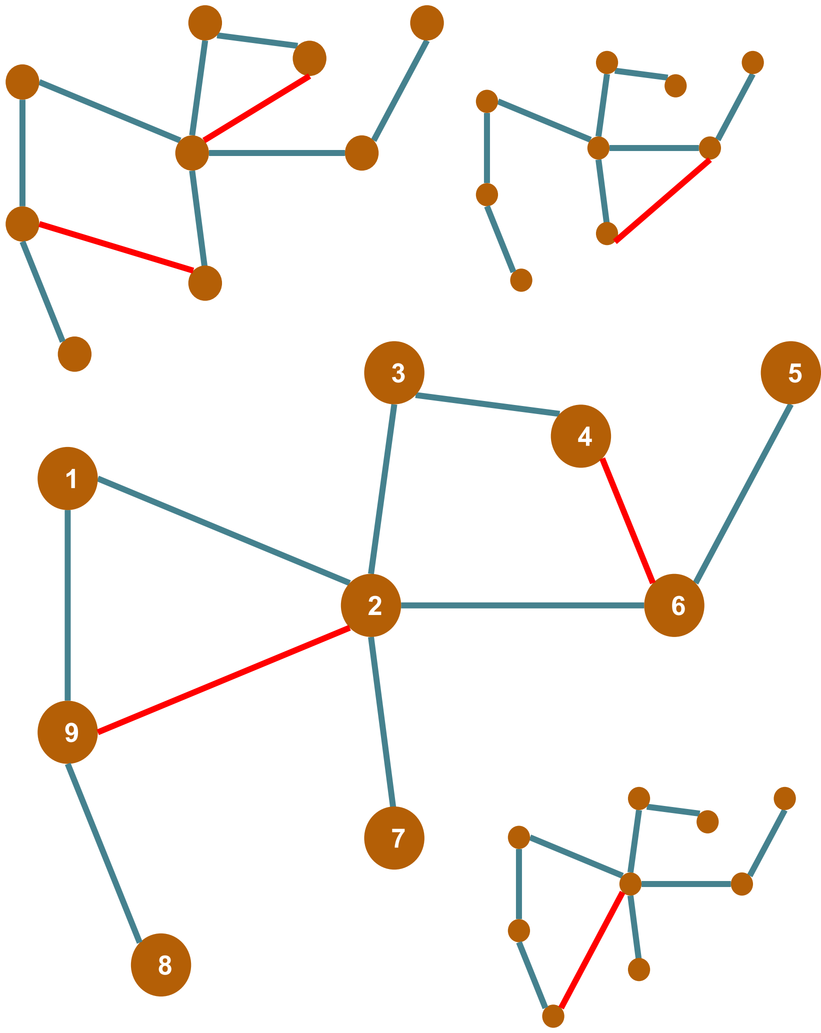
Corolario 1: Sea G un bosque con c c.c. $\Rightarrow m = n - c$



Árboles: Definiciones

Corolario 1: Sea G un bosque con c c.c. $\Rightarrow m = n - c$

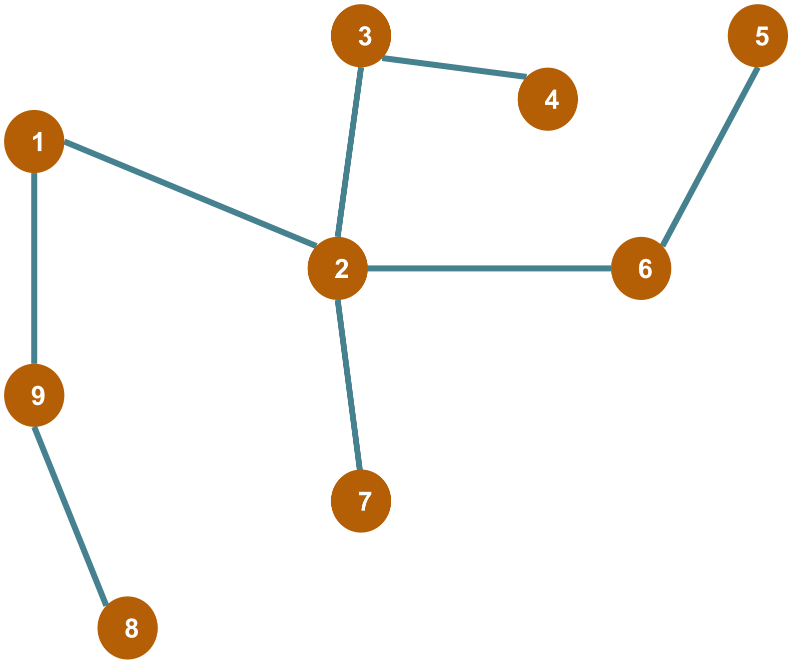
Corolario 2: Sea G un grafo con c c.c. $\Rightarrow m \geq n - c$



Árboles

Teorema 2: Equivalencias

1. *G* es un árbol *(grafo conexo sin circuitos simples)*.
2. *G* es un grafo sin circuitos simples y $m = n - 1$
3. *G* es un grafo conexo y $m = n - 1$



1 ⇒ 2) *G* es un árbol *(grafo conexo sin circuitos simples)*. ⇒ *G* es un grafo sin circuitos simples y $m = n - 1$

Árboles

Demostración (1 ⇒ 2):

(Por Lema 4)

2 ⇒ 3) *G* es un grafo sin circuitos simples y $m = n - 1 \Rightarrow G$ es un grafo conexo y $m = n - 1$

Árboles

Demostración (2 ⇒ 3):

Si tiene c c.c. $\Rightarrow m = n - c = n - 1 \Rightarrow c = 1 \Rightarrow G$ conexo

3 ⇒ 1) G es un grafo conexo y $m = n - 1 \Rightarrow G$ es un árbol

Árboles

Demostración (3 ⇒ 1):

Si G tiene un circuito simple, G conexo

$\Rightarrow$ *(por Lema 2)* $G\text{-}e$ conexo y $m_{G\text{-}e} = n - 2$ (porque $m_G = n - 1$)

¡Absurdo!

G no tiene un circuito simple, G conexo (es un árbol)

Lema 2: Sea $G = (V, E)$ un grafo conexo y $e=(v,u) \in E$.

$G - e = (V, E \setminus \{e\})$ es conexo $\Leftrightarrow e \in C$: circuito simple de G.

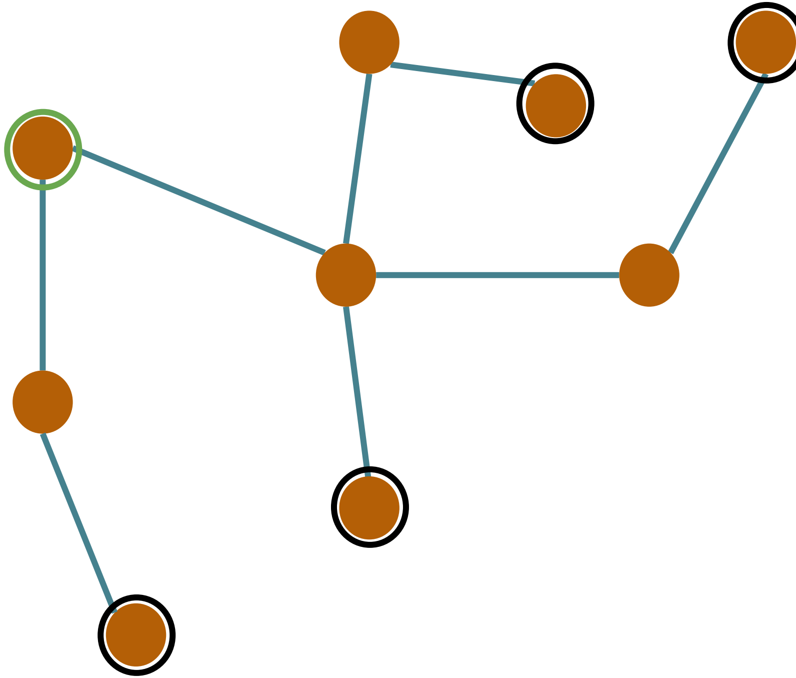
($e=(v,u) \in E$ es puente $\Leftrightarrow$ e no pertenece a un circuito simple de G).

Árboles con raíz (enraizados…)

Raíz: Algún vértice elegido

Hoja: u tq $d(u) = 1$

Árbol enraizado: Árbol con raíz



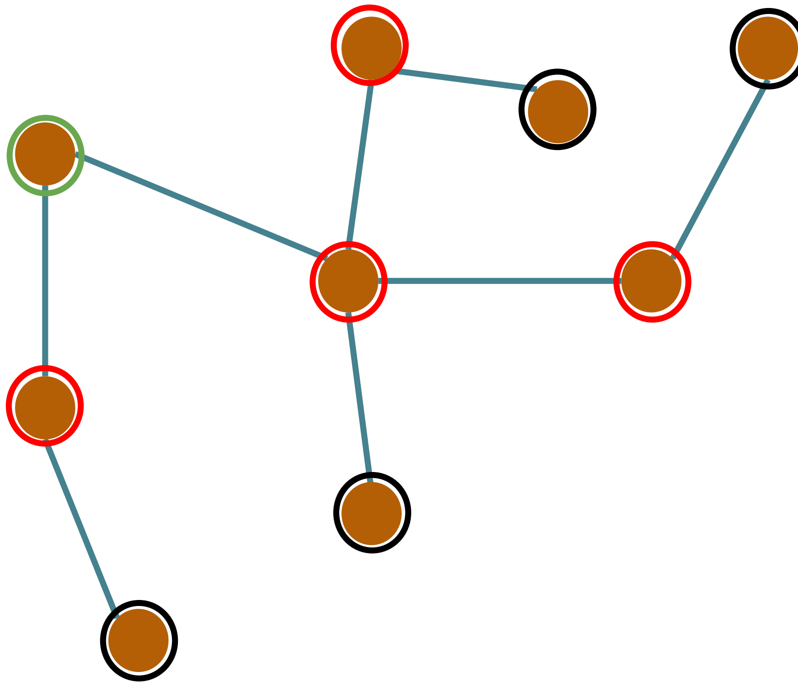
Árboles con raíz (enraizados…)

Raíz: Algún vértice elegido

Hoja: u tq $d(u) = 1$

Árbol enraizado: Árbol con raíz

Vértices internos: Ni hojas ni raíces



Árboles con raíz (enraizados…)

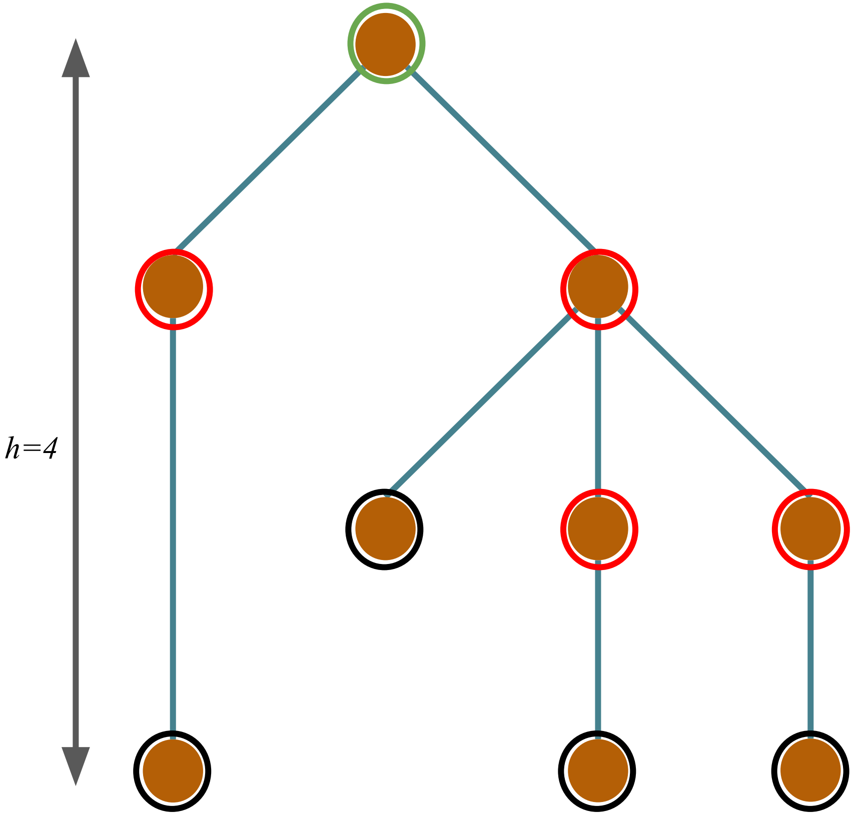
Raíz: Algún vértice elegido

Hoja: u tq $d(u) = 1$

Árbol enraizado: Árbol con raíz

Vértices internos: Ni hojas ni raíces

Altura (h): De la raíz a la hoja más lejana.



Árboles con raíz (enraizados…)

Raíz: Algún vértice elegido

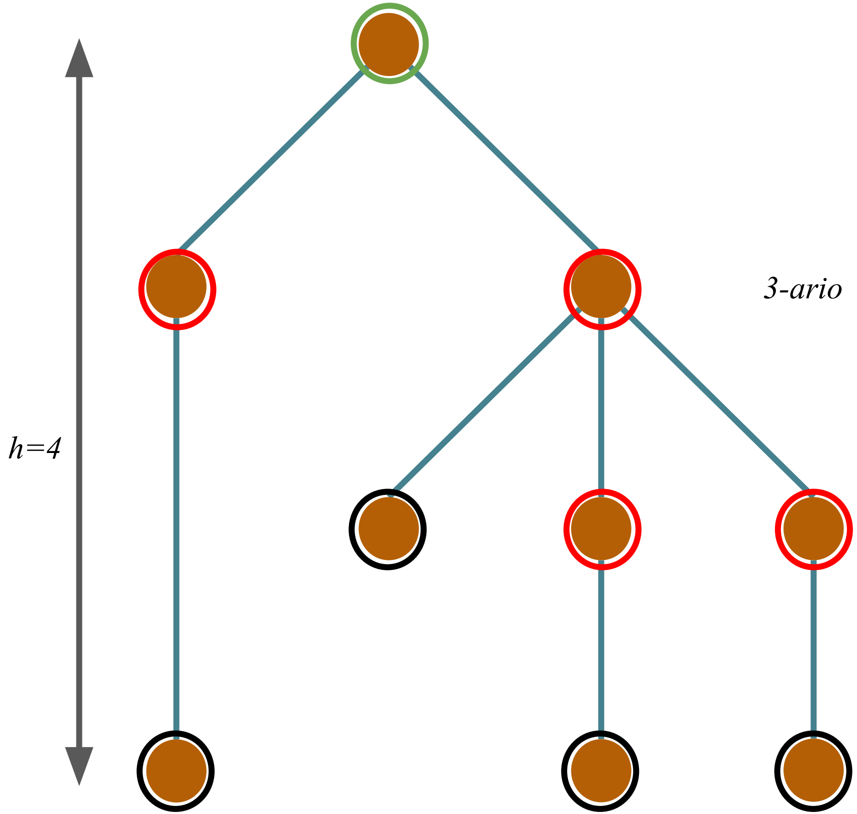
Hoja: *u* tq $d(u) = 1$

Árbol enraizado: Árbol con raíz

Vértices internos: Ni hojas ni raíces

Altura (h): De la raíz a la hoja más lejana.

Árbol m-ario: Donde *m* es el número máximo de hijos un nodo (si todos los vértices *v* tienen $d(v) \leq m + 1$ y la raíz *r* tiene $d(r) \leq m$).



Árboles con raíz (enraizados…)

Raíz: Algún vértice elegido

Hoja: *u* tq *d(u) = 1*

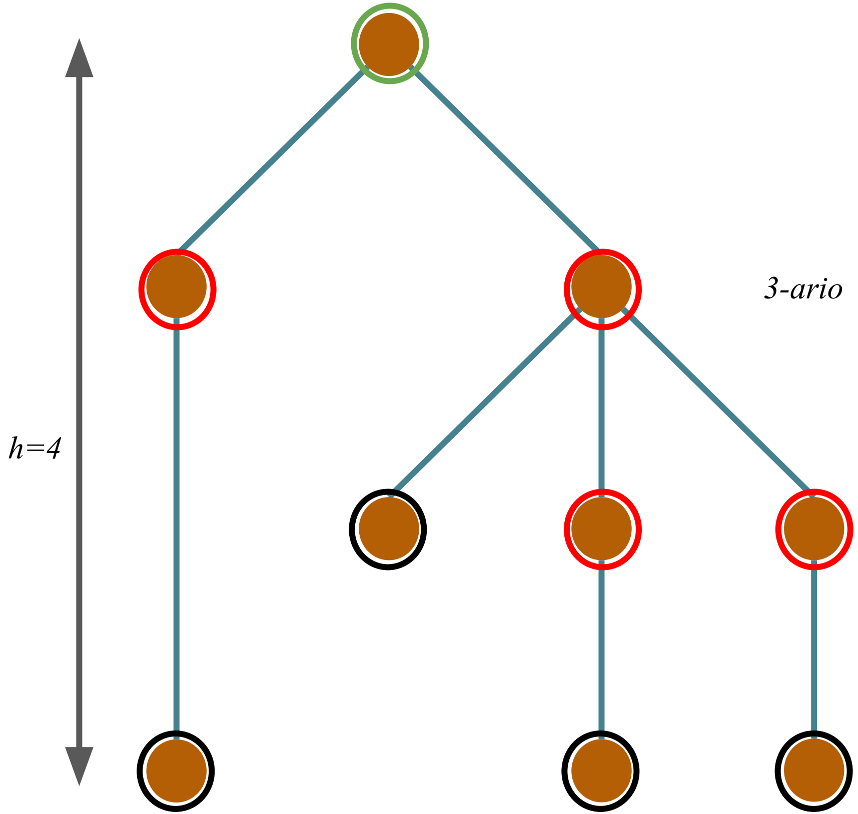
Árbol enraizado: Árbol con raíz

Vértices internos: Ni hojas ni raíces

Altura (h): De la raíz a la hoja más lejana.

Árbol m-ario: Donde *m* es el número máximo de hijos un nodo (si todos los vértices *v* tienen $d(v) \leq m + 1$ y la raíz *r* tiene $d(r) \leq m$).

Nivel: "Altura" de un vértice o distancia a la raíz.



Árboles con raíz (enraizados…)

Raíz: Algún vértice elegido

Hoja: u tq $d(u) = 1$

Árbol enraizado: Árbol con raíz

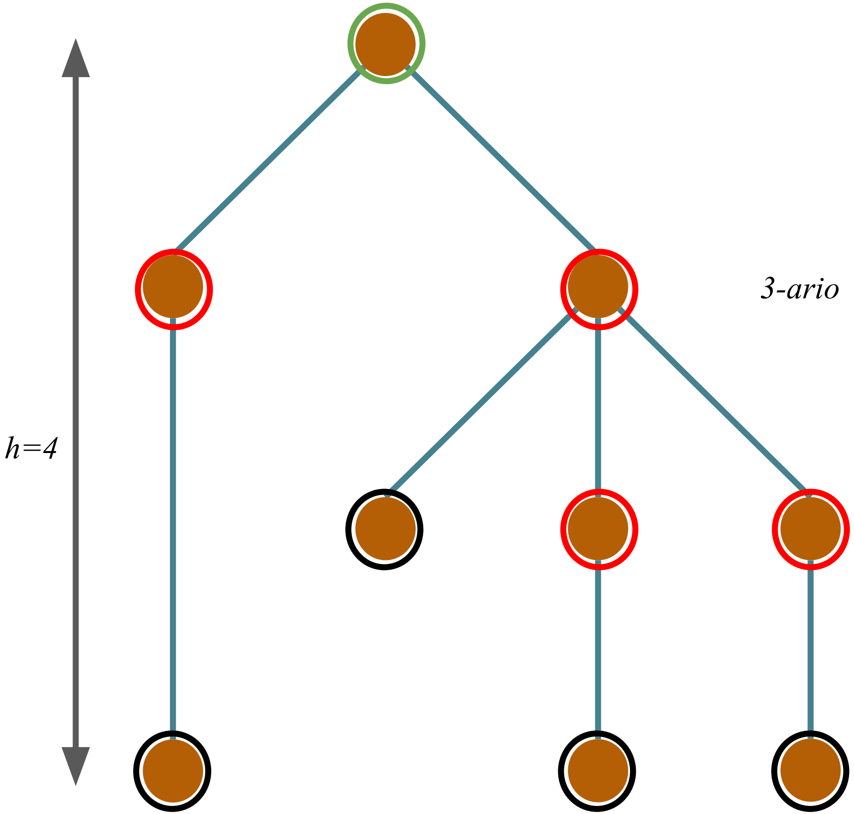
Vértices internos: Ni hojas ni raíces

Altura (h): De la raíz a la hoja más lejana.

Árbol m-ario: Donde m es el número máximo de hijos un nodo (si todos los vértices v tienen $d(v) \leq m + 1$ y la raíz r tiene $d(r) \leq m$).

Nivel: "Altura" de un vértice o distancia a la raíz.

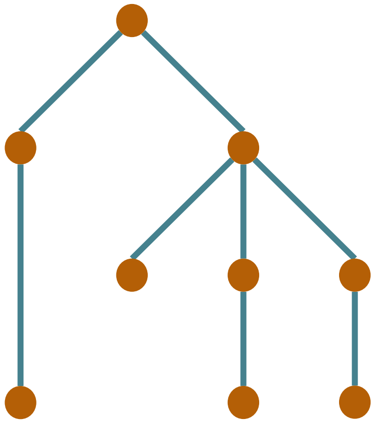
Árbol balanceado: Todas sus hojas están a nivel h (o h-1).



Árboles con raíz (enraizados…)

Teorema 3:

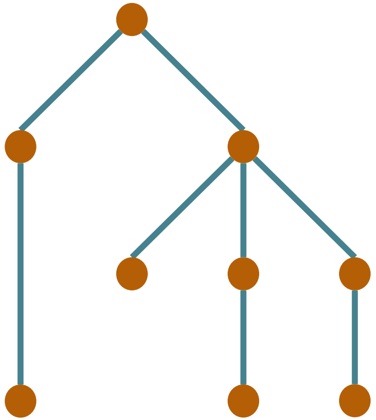
1. T es *m-ario* de altura $h \Rightarrow$ tiene a lo sumo $l=m^h$ hojas
2. T es *m-ario* con l hojas $\Rightarrow$ tiene $h \geq \lceil log_m(l) \rceil$ hojas



T es *m-ario* de altura $h \Rightarrow$ tiene a lo sumo $l = m^h$ hojas

Árboles con raíz (enraizados…)

Demostración (1):



$h=1 \Rightarrow l \leq m = m^1$

$h=2 \Rightarrow l \leq m * m^1 = m^2$

$h=3 \Rightarrow l \leq m * m^2 = m^3$

$h=4 \Rightarrow l \leq m * m^3 = m^4$

$\Rightarrow l \leq m * m^{h-1} = m^h$

Árboles con raíz (enraizados…)

Demostración (2):

$l \leq m^h$

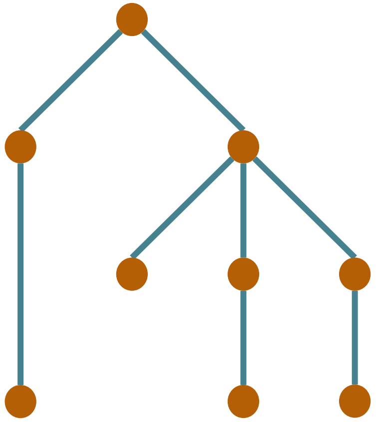
$log(l) \leq log(m^h)$

$log(l) \leq h*log(m)$

$log(l)/log(m) \leq h$

$log_m(l) \leq h$

$\lceil log_m(l) \rceil \leq h$ (por ser entero)



$h=1 \Rightarrow l \leq m = m^1$

$h=2 \Rightarrow l \leq m*m^1 = m^2$

$h=3 \Rightarrow l \leq m*m^2 = m^3$

$h=4 \Rightarrow l \leq m*m^3 = m^4$

$\Rightarrow l \leq m*m^{h-1} = m^h$

AED3 > Clase 5 > Search

Búsqueda / Recorrido en grafos

<u>Objetivos:</u>

- ➔ Encontrar caminos (mínimos)
  - ◆ Medir distancias.
  - ◆ Estimar el diámetro del grafo: Camino mínimo más largo.
- ➔ Encontrar todos los nodos alcanzables desde una fuente.
- ➔ Encontrar ciclos.
- ➔ Ordenar nodos (TOPOLOGICAL_SORT)
- ➔ Encontrar Componentes Fuertemente Conexas (c.f.c.)
- ➔ Encontrar el Árbol Generador Mínimo (AGM)
- ➔ …

Búsqueda / Recorrido en grafos

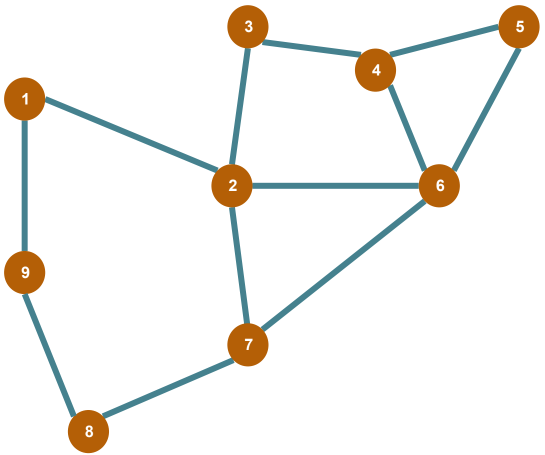
Repaso:

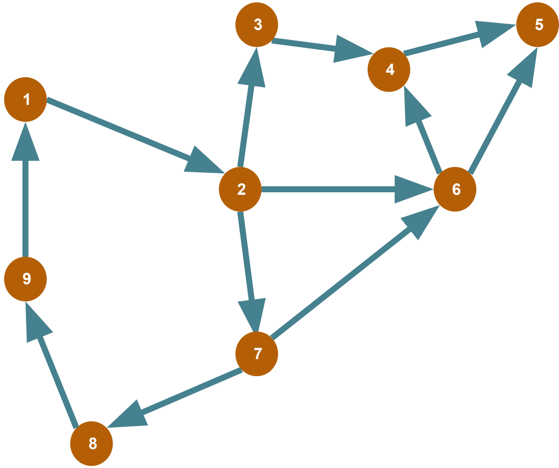
- ➔ $G = (V, E)$

Búsqueda / Recorrido en grafos

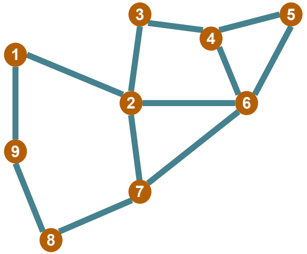
<u>Repaso:</u>

- ➔ G = (V, E)
- ➔ grafos y digrafos (lo que vamos a usar hoy)





Búsqueda / Recorrido en grafos



Repaso:

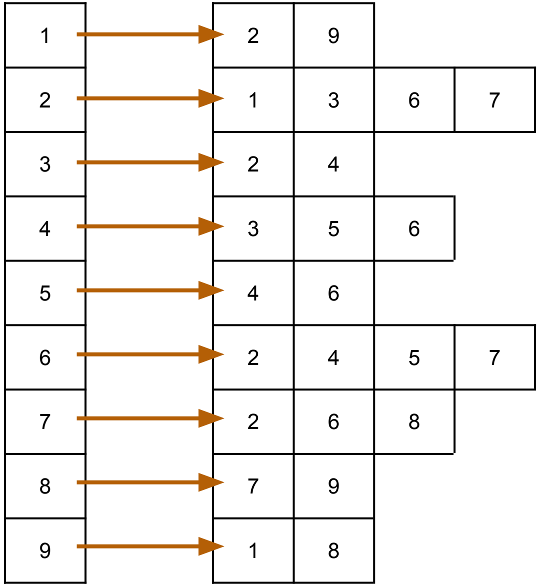
- ➔ G = (V, E)
- ➔ grafos y digrafos (lo que vamos a usar hoy)
- ➔ Listas de adyacencia

⟶ Adj[u] = vecinos de $\forall$ u $\in$ V

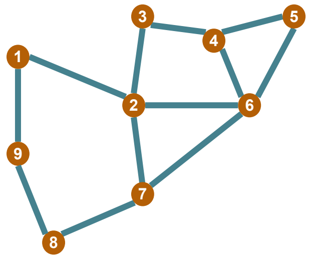
⟶ Representación rala (E ~ V)

⟶ Recorrerla lleva $\Theta(V+E)$

⟶ Es fácil definir muchos grafos con los mismos vértices



Búsqueda / Recorrido en grafos



<u>Repaso:</u>

➔ Otras representaciones

⟶ Objeto: ***u.vecinos*** (CLRS)

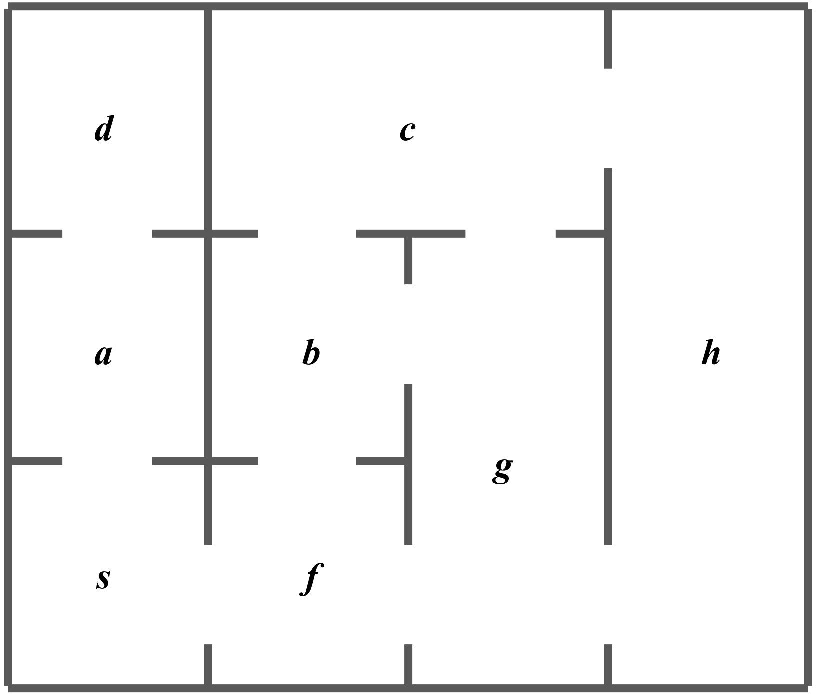
⟶ Representación implícita:

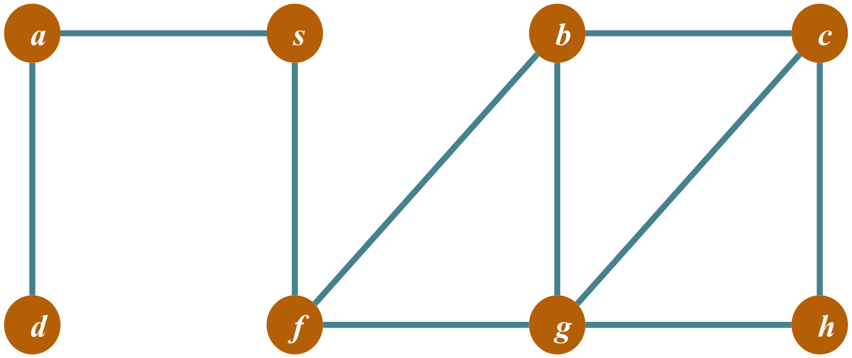
Defino una función Adj(u) o un método u.vecinos.

Ventaja: No guardo todo el grafo en memoria. Entonces, si es fácil de calcular el siguiente punto a partir del anterior (como por ej. los estados de un cubo rubik) y el grafo es muy grande, como el cubo rubik) se ahorra muchísimo espacio!!

1	2	9		
2	1	3	6	7
3	2	4		
4	3	5	6	
5	4	6		
6	2	4	5	7
7	2	6	8	
8	7	9		
9	1	8		

Breadth First Search (BFS)





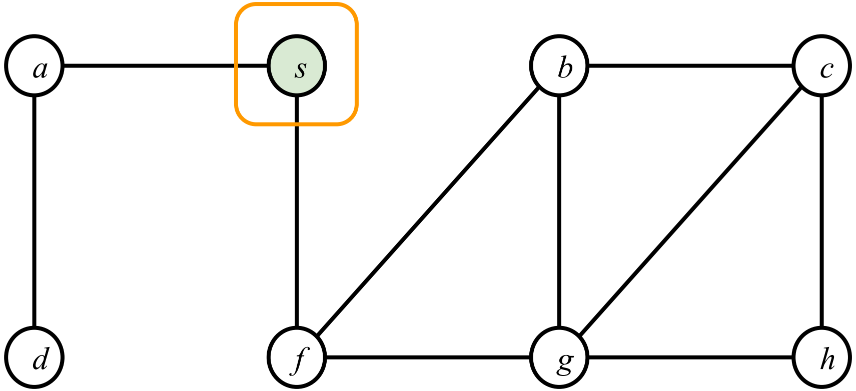
Moore (1959) "*The shortest path through a maze*" pensado para estimar el tráfico en las telecomunicaciones.

Breadth First Search (BFS)

- <u>Objetivo</u>:
  - Visitar todos los nodos accesibles (alcanzables) desde una fuente (**s**; *source*).
    - nos permite computar a distancia hasta esos nodos.
    - generar un árbol a través de los caminos generados.
  - $\Theta(V+E)$
- <u>Idea</u>:
  - Visito los vecinos de s, luego los vecinos de los vecinos, luego los vecinos de los vecinos de los vecinos, … (por capas)
    - en 0 movidas ⟶ *s*
    - en 1 movidas ⟶ *Adj[s]*
    - en 2 movidas ⟶ ...
    - …
- *Contiene las ideas generales de otros algoritmos como: Prim (Árbol Generador Mínimo) o Dijkstra (Camino Mínimo)*

Breadth First Search (BFS) iterativo

```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |     |     |     |     level [ v ] = i
|     |     |     |     |     parent [ v ] = u
|     |     |     |     |     next.append( v )
|     |     frontier = next
|     |     i += 1
```



level = {*s*: 0}

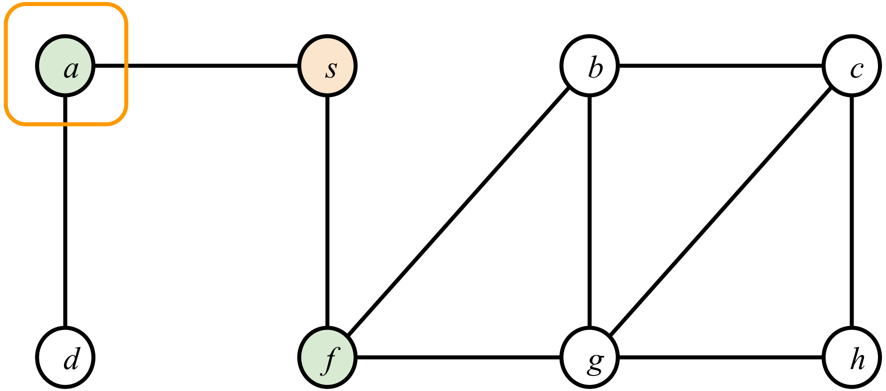
parent = {*s*: *None*}

frontier = [*s*]

Adj[*s*] = { *a*, *f*}

Breadth First Search (BFS) iterativo

```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |     |     |     |     level [ v ] = i
|     |     |     |     |     parent [ v ] = u
|     |     |     |     |     next.append( v )
|     |     frontier = next
|     |     i += 1
```



$level = \{s: 0, a: 1, f: 1\}$

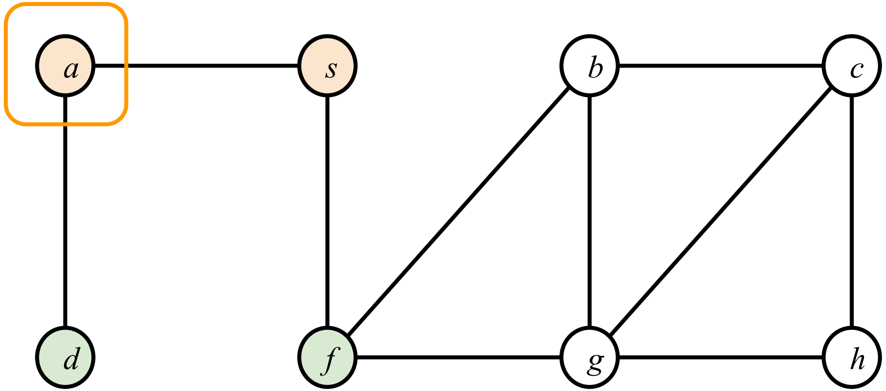
$parent = \{s: None, a: s, f: s\}$

$frontier = [\, a, f \,]$

$Adj[\, a \,] = \{\, d, s\}$

Breadth First Search (BFS) iterativo

```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |     |     |     |     level [ v ] = i
|     |     |     |     |     parent [ v ] = u
|     |     |     |     |     next.append( v )
|     |     frontier = next
|     |     i += 1
```



level = {*s*: 0, *a*: 1, *f*: 1, *d*: 2}

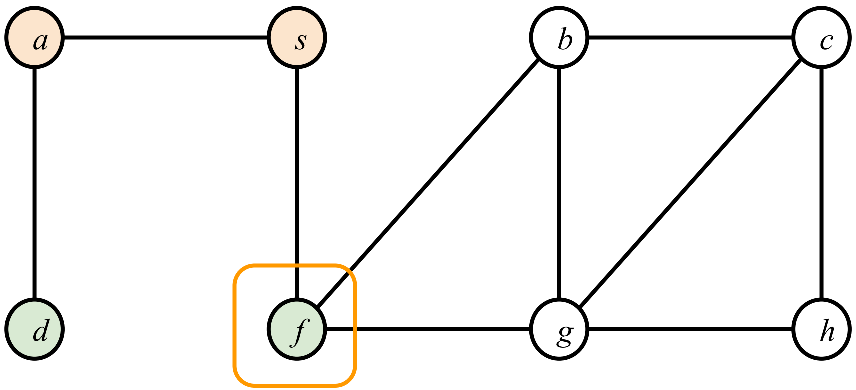
parent = {*s*: None, *a*: *s*, *f*: *s*, *d*: *a*}

frontier = [*a*, *f*]

Adj[*a*] = { *d*, *s*}

Breadth First Search (BFS) iterativo

```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |     |     |     |     level [ v ] = i
|     |     |     |     |     parent [ v ] = u
|     |     |     |     |     next.append( v )
|     |     frontier = next
|     |     i += 1
```



level $= \{s: 0, a: 1, f: 1, d: 2\}$

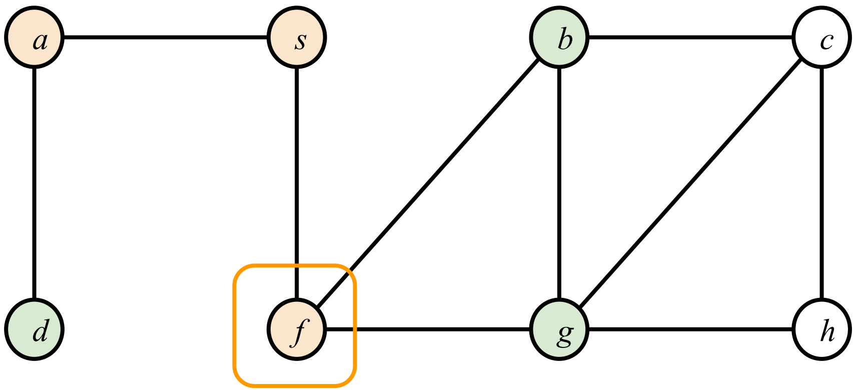
parent $= \{s: None, a: s, f: s, d: a\}$

frontier $= [\, a, f\,]$

Adj[f] $= \{\, b, g, s\}$

Breadth First Search (BFS) iterativo

```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |     |     |     |     level [ v ] = i
|     |     |     |     |     parent [ v ] = u
|     |     |     |     |     next.append( v )
|     |     frontier = next
|     |     i += 1
```



level = {*s*: 0, *a*: 1, *f*: 1, *d*: 2, *b*: 2, *g*: 2}

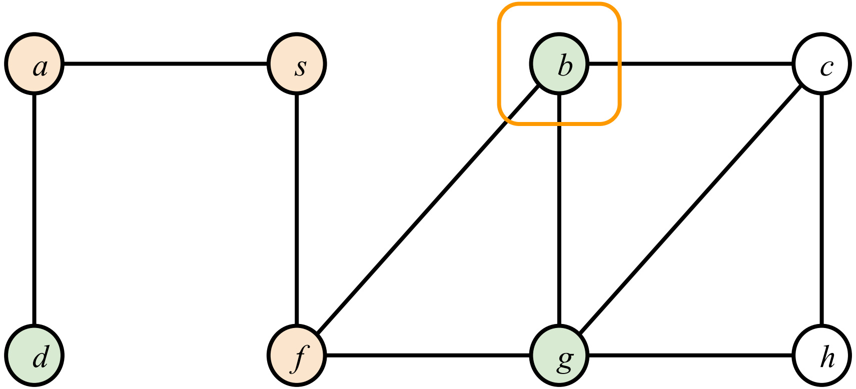
parent = {*s*: None, *a*: *s*, *f*: *s*, *d*: *a*, *b*: *f*, *g*: *f*}

frontier = [*a*, *f*]

Adj[*f*] = { *b*, *g*, *s*}

Breadth First Search (BFS) iterativo

```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |     |     |     |     level [ v ] = i
|     |     |     |     |     parent [ v ] = u
|     |     |     |     |     next.append( v )
|     |     frontier = next
|     |     i += 1
```



level = {*s*: 0, *a*: 1, *f*: 1, *d*: 2, *b*: 2, *g*: 2}

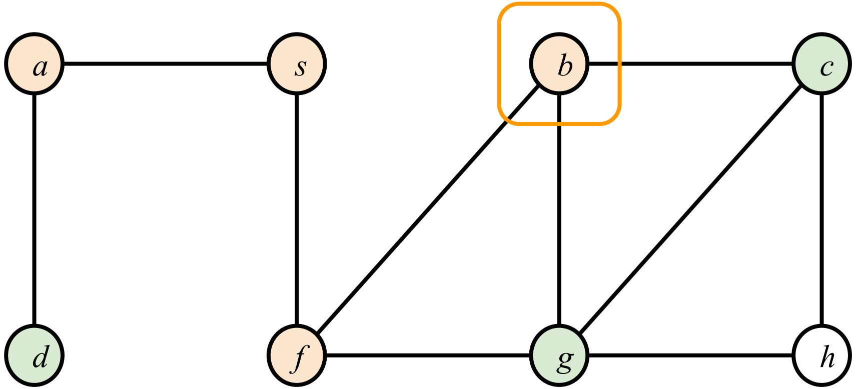
parent = {*s*: None, *a*: *s*, *f*: *s*, *d*: *a*, *b*: *f*, *g*: *f*}

frontier = [*b*, *g*, *d*]

Adj[*b*] = { *c*, *f*, *g*}

Breadth First Search (BFS) iterativo

```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |     |     |     |     level [ v ] = i
|     |     |     |     |     parent [ v ] = u
|     |     |     |     |     next.append( v )
|     |     frontier = next
|     |     i += 1
```



level = {*s*: 0, *a*: 1, *f*: 1, *d*: 2, *b*: 2, *g*: 2, *c*: 3}

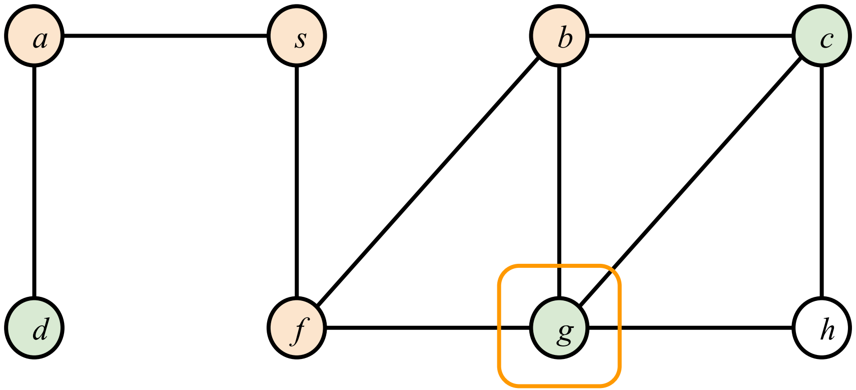
parent = {*s*: None, *a*: *s*, *f*: *s*, *d*: *a*, *b*: *f*, *g*: *f*, *c*: *b*}

frontier = [*b*, *g*, *d*]

Adj[*b*] = { *c*, *f*, *g*}

Breadth First Search (BFS) iterativo

```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |     |     |     |     level [ v ] = i
|     |     |     |     |     parent [ v ] = u
|     |     |     |     |     next.append( v )
|     |     frontier = next
|     |     i += 1
```



level = {*s*: 0, *a*: 1, *f*: 1, *d*: 2, *b*: 2, *g*: 2, *c*: 3}

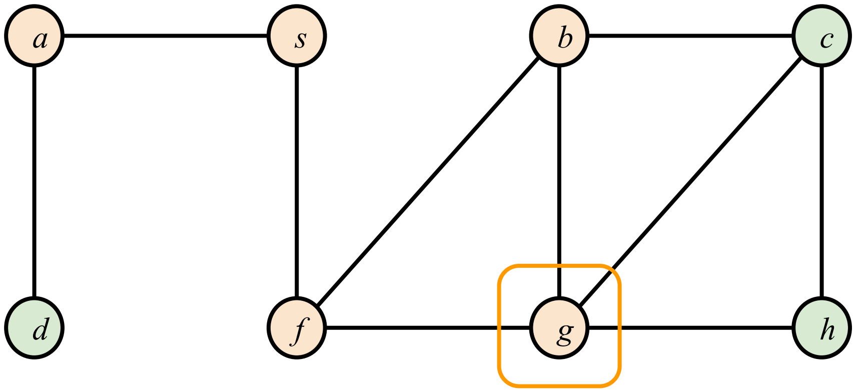
parent = {*s*: None, *a*: *s*, *f*: *s*, *d*: *a*, *b*: *f*, *g*: *f*, *c*: *b*}

frontier = [*b*, *g*, *d*]

Adj[*g*] = { *b*, *c*, *f*, *h*}

Breadth First Search (BFS) iterativo

```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |     |     |     |     level [ v ] = i
|     |     |     |     |     parent [ v ] = u
|     |     |     |     |     next.append( v )
|     |     frontier = next
|     |     i += 1
```



level = {s: 0, a: 1, f: 1, d: 2, b: 2, g: 2, c: 3, h: 3}

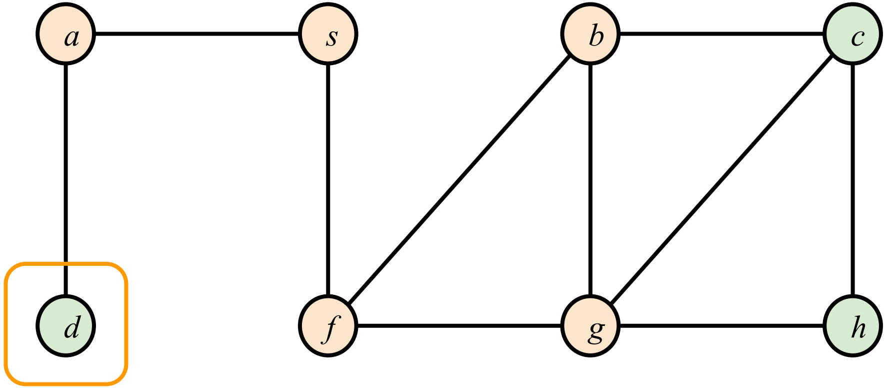
parent = {s: None, a: s, f: s, d: a, b: f, g: f, c: b, h: g}

frontier = [b, g, d]

Adj[g] = { b, c, f, h}

Breadth First Search (BFS) iterativo

```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |     |     |     |     level [ v ] = i
|     |     |     |     |     parent [ v ] = u
|     |     |     |     |     next.append( v )
|     |     frontier = next
|     |     i += 1
```



level = {*s*: 0, *a*: 1, *f*: 1, *d*: 2, *b*: 2, *g*: 2, *c*: 3, *h*: 3}

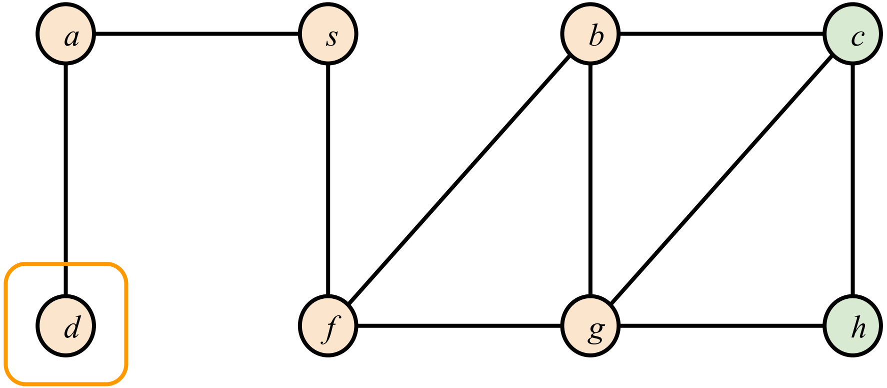
parent = {*s*: None, *a*: *s*, *f*: *s*, *d*: *a*, *b*: *f*, *g*: *f*, *c*: *b*, *h*: *g*}

frontier = [*b*, *g*, *d*]

Adj[*d*] = { *a* }

Breadth First Search (BFS) iterativo

```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |     |     |     |     level [ v ] = i
|     |     |     |     |     parent [ v ] = u
|     |     |     |     |     next.append( v )
|     |     frontier = next
|     |     i += 1
```



level $= \{s\text{: } 0, a\text{: } 1, f\text{: } 1, d\text{: } 2, b\text{: } 2, g\text{: } 2, c\text{: } 3, h\text{: } 3\}$

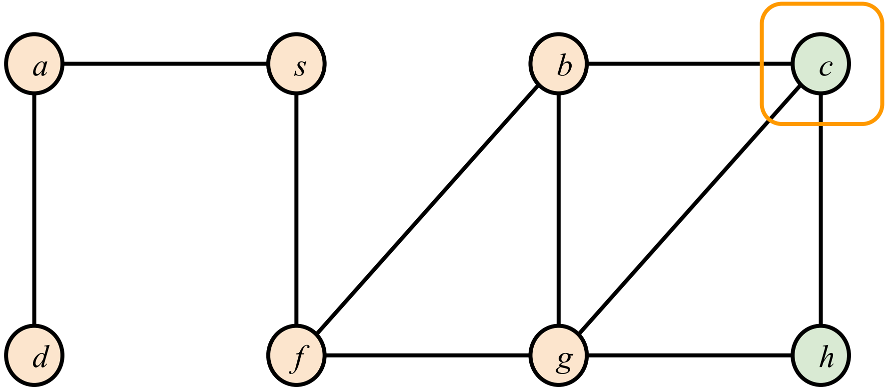
parent $= \{s\text{: None}, a\text{: } s, f\text{: } s, d\text{: } a, b\text{: } f, g\text{: } f, c\text{: } b, h\text{: } g\}$

frontier $= [\, b, g, d \,]$

Adj[*d*] $= \{\, a \,\}$

Breadth First Search (BFS) iterativo

```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |     |     |     |     level [ v ] = i
|     |     |     |     |     parent [ v ] = u
|     |     |     |     |     next.append( v )
|     |     frontier = next
|     |     i += 1
```



level = {s: 0, a: 1, f: 1, d: 2, b: 2, g: 2, c: 3, h: 3}

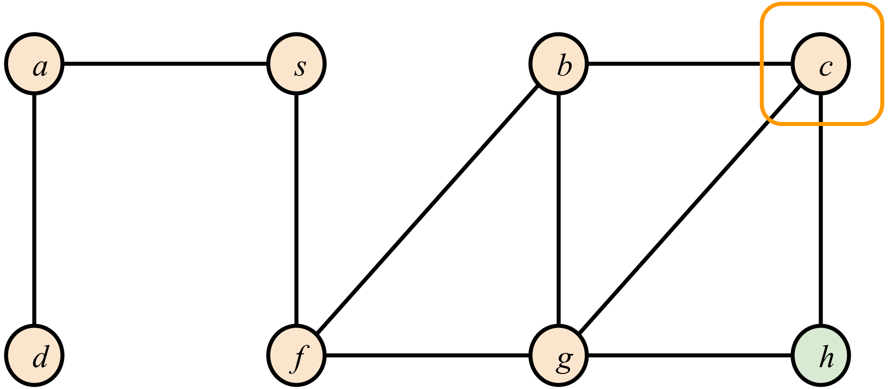
parent = {s: None, a: s, f: s, d: a, b: f, g: f, c: b, h: g}

frontier = [c, h]

Adj[c] = { b, g, h}

Breadth First Search (BFS) iterativo

```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |     |     |     |     level [ v ] = i
|     |     |     |     |     parent [ v ] = u
|     |     |     |     |     next.append( v )
|     |     frontier = next
|     |     i += 1
```



level = {*s*: 0, *a*: 1, *f*: 1, *d*: 2, *b*: 2, *g*: 2, *c*: 3, *h*: 3}

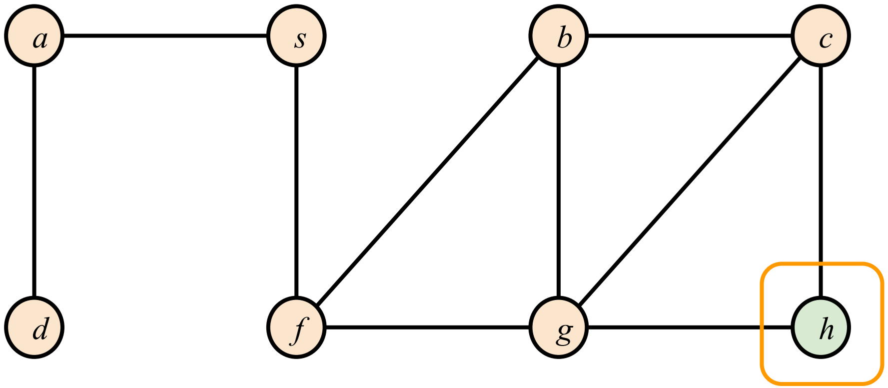
parent = {*s*: None, *a*: *s*, *f*: *s*, *d*: *a*, *b*: *f*, *g*: *f*, *c*: *b*, *h*: *g*}

frontier = [*c*, *h*]

Adj[*c*] = { *b*, *g*, *h*}

Breadth First Search (BFS) iterativo

```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |     |     |     |     level [ v ] = i
|     |     |     |     |     parent [ v ] = u
|     |     |     |     |     next.append( v )
|     |     frontier = next
|     |     i += 1
```



level = {*s*: 0, *a*: 1, *f*: 1, *d*: 2, *b*: 2, *g*: 2, *c*: 3, *h*: 3}

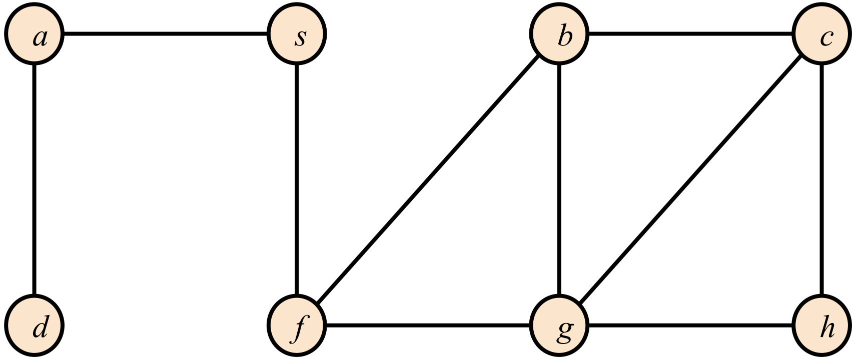
parent = {*s*: None, *a*: *s*, *f*: *s*, *d*: *a*, *b*: *f*, *g*: *f*, *c*: *b*, *h*: *g*}

frontier = [*c*, *h*]

Adj[*h*] = { *c*, *g* }

Breadth First Search (BFS) iterativo

```
BFS ( s , Adj ) :
|    level = { s : 0 }
|    parent = { s : None }
|    i = 1
|    frontier = [ s ] # level i-1
|    while frontier :
|    |    next = [ ] # level i
|    |    for u in frontier :
|    |    |    for v in Adj[ u ] :
|    |    |    |    if v not in level :
|    |    |    |    |    level [ v ] = i
|    |    |    |    |    parent [ v ] = u
|    |    |    |    |    next.append( v )
|    |    frontier = next
|    |    i += 1
```



level = {*s*: 0, *a*: 1, *f*: 1, *d*: 2, *b*: 2, *g*: 2, *c*: 3, *h*: 3}

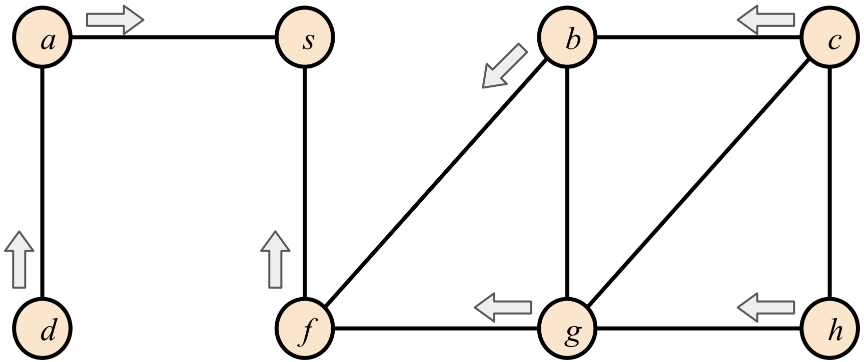
parent = {*s*: None, *a*: *s*, *f*: *s*, *d*: *a*, *b*: *f*, *g*: *f*, *c*: *b*, *h*: *g*}

frontier = [*c*, *h*]

Adj[*h*] = { *c*, *g* }

Breadth First Search (BFS) iterativo

```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |     |     |     |     level [ v ] = i
|     |     |     |     |     parent [ v ] = u
|     |     |     |     |     next.append( v )
|     |     frontier = next
|     |     i += 1
```



level = {*s*: 0, *a*: 1, *f*: 1, *d*: 2, *b*: 2, *g*: 2, *c*: 3, *h*: 3}

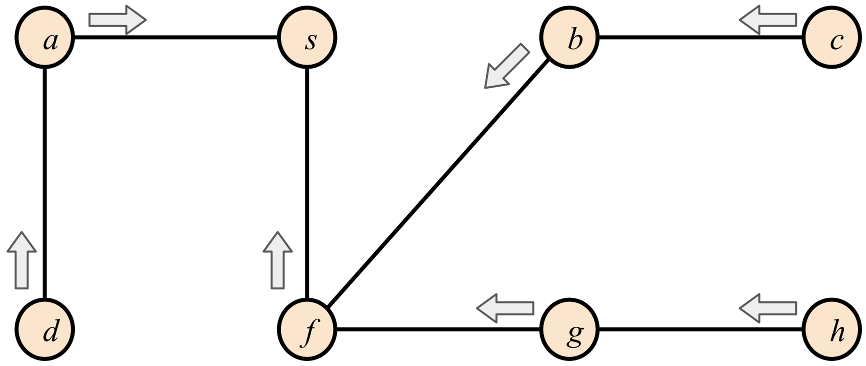
***parent* = {*s*: None, *a*: *s*, *f*: *s*, *d*: *a*, *b*: *f*, *g*: *f*, *c*: *b*, *h*: *g*}**

frontier = [*c*, *h*]

Adj[*h*] = { *c*, *g* }

BFS-tree

```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |     |     |     |     level [ v ] = i
|     |     |     |     |     parent [ v ] = u
|     |     |     |     |     next.append( v )
|     |     frontier = next
|     |     i += 1
```



level = {*s*: 0, *a*: 1, *f*: 1, *d*: 2, *b*: 2, *g*: 2, *c*: 3, *h*: 3}

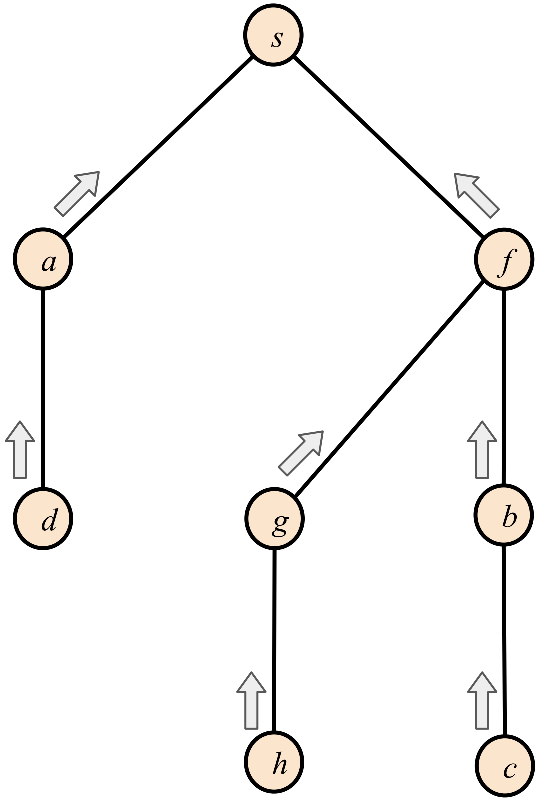
***parent* = {*s*: None, *a*: *s*, *f*: *s*, *d*: *a*, *b*: *f*, *g*: *f*, *c*: *b*, *h*: *g*}**

frontier = [*c*, *h*]

Adj[*h*] = { *c*, *g* }

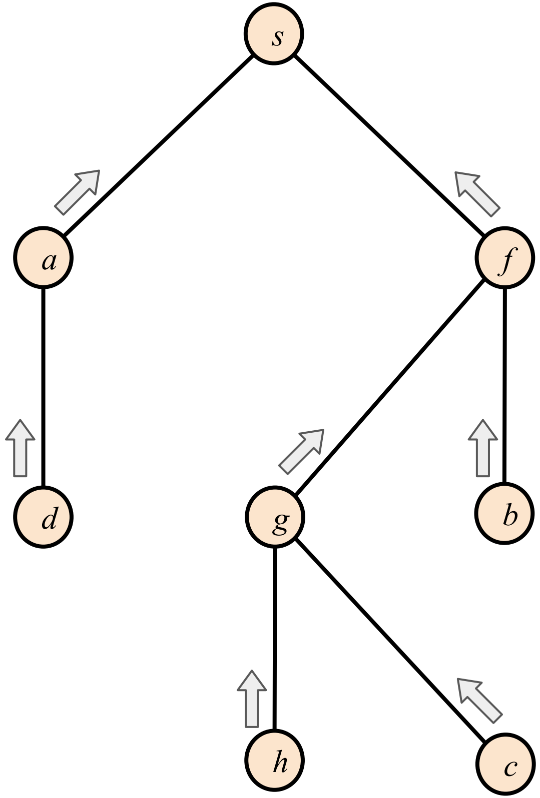
BFS-tree

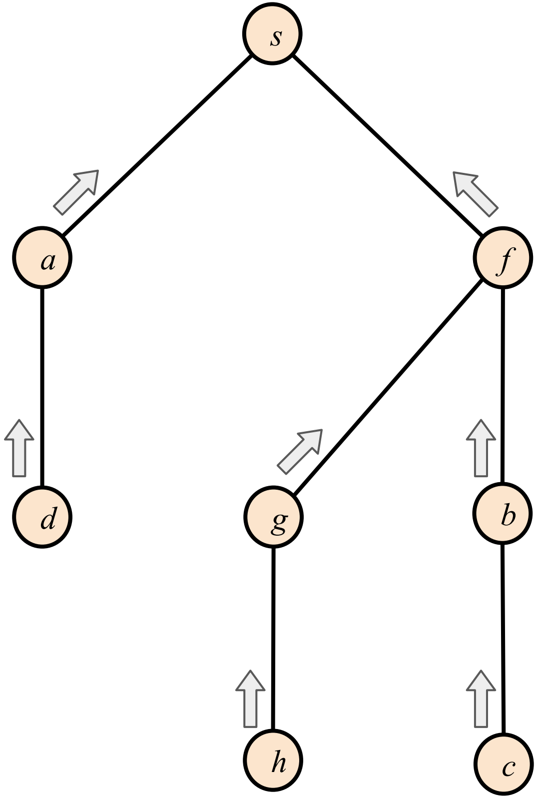
Gπ = (Vπ, Eπ)
Vπ = {v ∈ V : parent[v] ≠ None} ∪ {s}
Eπ = {(parent[v], v): v ∈ Vπ-{s} }



level = {s: 0, a: 1, f: 1, d: 2, b: 2, g: 2, c: 3, h: 3}
***parent* = {s: None, a: s, f: s, d: a, b: f, g: f, c: b, h: g}**
frontier = [c, h]
Adj[h] = { c, g }

BFS-tree





level = {s: 0, a: 1, f: 1, d: 2, b: 2, g: 2, c: 3, h: 3}

parent **= {s: None, a: s, f: s, d: a, b: f, g: f, c: b, h: g}**

level = {s: 0, a: 1, f: 1, d: 2, b: 2, g: 2, c: 3, h: 3}

parent **= {s: None, a: s, f: s, d: a, b: f, g: f, c: g, h: g}**

Breadth First Search (BFS) iterativo

```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |     |     |     |     level [ v ] = i
|     |     |     |     |     parent [ v *] = u
|     |     |     |     |     next.append( v )
|     |     frontier = next
|     |     i += 1
```

Cada vértice entra a la lista sólo una vez (y se explora sólo una vez)
$\Rightarrow$ *O(V)*

Cada vencindario (Adj[u]) se explora sólo una vez
$\Rightarrow$

$$\sum_{u \in V} |Adj[u]| = \begin{cases} |E| \ para\ grafos\ dirigidos \\ 2|E| \ para\ grafos\ no\ dirigidos \end{cases}$$

$\Rightarrow$ *O(E)*

$\boxed{\Rightarrow \boldsymbol{O(V+E)}}$

Breadth First Search (BFS) iterativo (versión CLRS)

```
BFS ( G , s ) :
|     for cada nodo u ∈ G.V - { s }
|     |     u.color    = n        # w: nuevo, g: frontera descubierta, k: usado
|     |     u.d        = ∞        # distancia
|     |     u.π        = NIL      # parent / predecesor
|     s.color    = g
|     s.d        = 0
|     s.π        = NIL
|     Q          = ∅         # Q: cola: Guardo los que tengo que explorar a continuación: frontera
|     ENQUEUE(Q,s)           # agrega s a la cola Q
|     while Q ≠ 0 :
|     |     u = DEQUEUE( Q )
|     |     for cada v ∈ G.Adj[ u ] :
|     |     |     if v.color == w :     # si no fue visita aún
|     |     |     |     v.color = g     # lo marco
|     |     |     |     v.d = u.d + 1   # actualizo la distancia
|     |     |     |     v.π = u         # u es el predecesor de v
|     |     |     |     ENQUEUE(Q,v)    # guardo v para explorar después
|     |     u.color = k # termino de explorar y lo marco
```

BFS demo:

Voy a usar la versión del CLRS pero es lo mismo (pueden pensarlo para la otra versión).

<u>Definición:</u> Un nodo u es ALCANZABLE desde s si existe un camino $P: s \rightarrow u$

<u>Lema 1:</u> Todo nodo u ALCANZABLE desde s es descubierto por BFS en algún momento. $\Rightarrow$ Tiene distancia finita.

Si el nodo NO es ALCANZABLE desde $s \Rightarrow$ NO es descubierto por BFS y tiene distancia infinita.

<u>Lema 2 (ORDEN):</u> Al procesar u (a una distancia d desde s) $\Rightarrow$ todos los nodos con distancia $\leq d$ ya fueron descubiertos.

<u>Lema 3 (COLA):</u> En todo momento, si el primer ítem de la cola está a distancia d de $s \Rightarrow$ todos los ítems de la cola están a distancia d ó $d+1$ de s. Y, además, todos los que están a distancia d están antes que los que tienen distancia $d+1$.

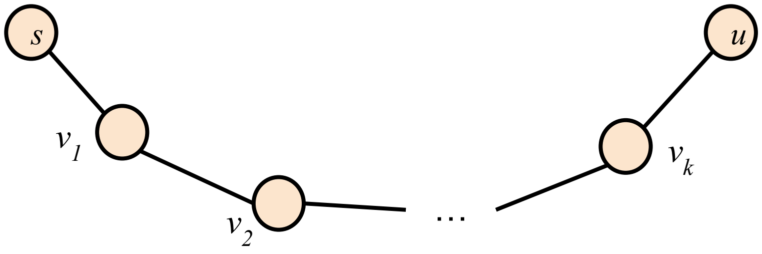
<u>Teo (Correctitud):</u> Cuando un nodo u es descubierto (agregado a la cola), su $dist[u]$ es exactamente $d(s,u)$.

BFS demo:

Lema 1: Todo nodo u ALCANZABLE desde s es descubierto por BFS en algún momento. $\Rightarrow$ Tiene distancia finita.

Si el nodo NO es ALCANZABLE desde $s \Rightarrow$ NO es descubierto por BFS y tiene distancia infinita.

Demo: (Absurdo) Hip.: Existen nodos que son ALCANZABLES y no son descubiertos por BFS.



* u es el nodo ALCANZABLE no descubierto más cercano a s.

* $\exists$ P: $s \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k \rightarrow u$.

* v_k es ALCANZABLE y $dist(s,v_k) < dist(s,u)$.

* $\Rightarrow$ u es descubierto al procesar v_k

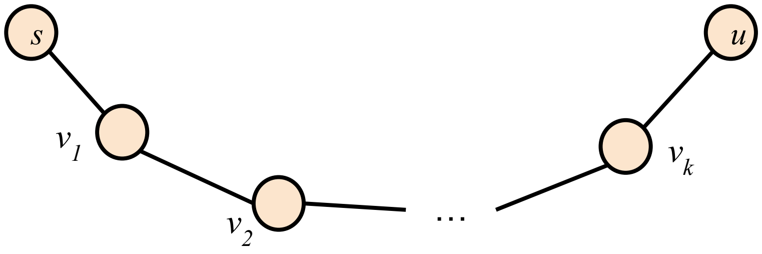
* $\Rightarrow$ ¡Absurdo! Todos los ALCANZABLES son descubiertos

BFS demo:

Lema 1: Todo nodo u ALCANZABLE desde s es descubierto por BFS en algún momento. $\Rightarrow$ Tiene distancia finita.

Si el nodo NO es ALCANZABLE desde s $\Rightarrow$ NO es descubierto por BFS y tiene distancia infinita.

Demo: (Absurdo) Hip.: Existen nodos que son NO son ALCANZABLES pero son descubiertos por BFS.



* u es descubierto al procesar v_k

* v_k es ALCANZABLE

* u es ALCANZABLE por P: $s \to v_1 \to v_2 \to \ldots \to v_k \to u$.

* $\Rightarrow$ ¡Absurdo! Los NO ALCANZABLES no son descubiertos (y quedan con distancia infinita)

BFS demo:

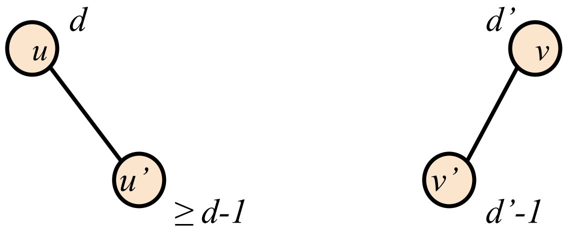
<u>Lema 2 (ORDEN):</u> Al procesar u (a una distancia d desde s) $\Rightarrow$ todos los nodos con distancia $\leq d$ ya fueron descubiertos.

<u>Demo:</u> (Absurdo) Hip.: Existe un primer nodo u a distancia d procesado y v a distancia d' no procesado, tal que $d' \leq d$ por primera vez.

BFS demo:

Lema 2 (ORDEN): Al procesar u (a una distancia d desde s) $\Rightarrow$ todos los nodos con distancia $\leq d$ ya fueron descubiertos.

Demo: (Absurdo) Hip.: Existe un primer nodo u a distancia d procesado y v a distancia d' no procesado, tal que $d' \leq d$ por primera vez.

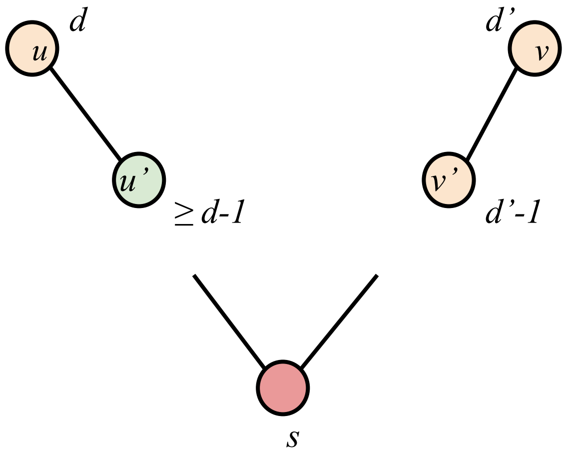


BFS demo:

Lema 2 (ORDEN): Al procesar u (a una distancia d desde s) $\Rightarrow$ todos los nodos con distancia $\leq d$ ya fueron descubiertos.

Demo: (Absurdo) Hip.: Existe un primer nodo u a distancia d procesado y v a distancia d' no procesado, tal que $d' \leq d$ por primera vez.

[u']



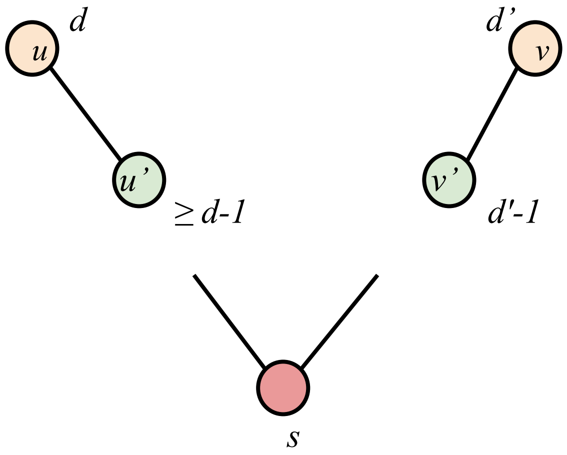
BFS demo:

Lema 2 (ORDEN): Al procesar u (a una distancia d desde s) $\Rightarrow$ todos los nodos con distancia $\leq d$ ya fueron descubiertos.

Demo: (Absurdo) Hip.: Existe un primer nodo u a distancia d procesado y v a distancia d' no procesado, tal que $d' \leq d$ por primera vez.

[u']

[u' v'] $\Rightarrow$ $d\text{-}1 \leq d'\text{-}1$.



BFS demo:

<u>Lema 2 (ORDEN):</u> Al procesar u (a una distancia d desde s) $\Rightarrow$ todos los nodos con distancia $\leq d$ ya fueron descubiertos.

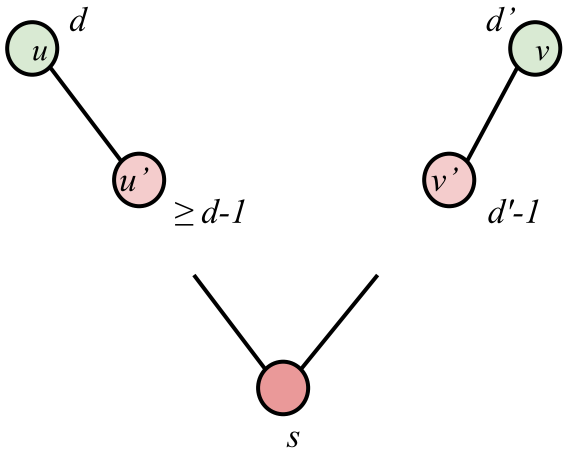
<u>Demo:</u> (Absurdo) Hip.: Existe un primer nodo u a distancia d procesado y v a distancia d' no procesado, tal que $d' \leq d$ por primera vez.

[u']

[u' v'] $\Rightarrow$ $d\text{-}1 \leq d'\text{-}1$.

[v' u] $\Rightarrow$ $d'\text{-}1 \leq d$.

[u v] $\Rightarrow$ $d \leq d'$ ¡Absurdo!



BFS demo:

<u>Lema 3 (COLA):</u> En todo momento, si el primer ítem de la cola está a distancia d de $s \Rightarrow$ todos los ítems de la cola están a distancia d ó $d+1$ de s. Y, además, todos los que están a distancia d están antes que los que tienen distancia $d+1$.

<u>Demo:</u>

* Todos los nodos a distancia d entran antes de que el primero de ellos salga.

* Los $d+1$ entran al ser procesados los d, que van saliendo.

* Todos los nodos a distancia $d-1$ ya salieron antes de que el primer $d+1$ entre. Similar con d y $> d+1$.

$\Rightarrow$ Sólo hay nodos con distancia d o $d+1$ cuando el primero es distancia d.

d	d	d	...	d	d	$d+1$	$d+1$	$d+1$	...	$d+1$	$d+1$

BFS demo:

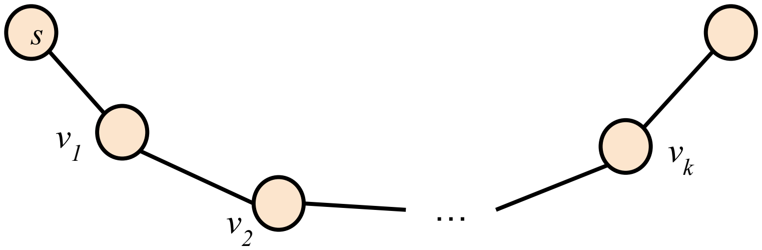
Teo (Correctitud): Cuando un nodo u es descubierto (agregado a la cola), su $dist[u]$ es exactamente $d(s,u)$.

Demo: (Inducción)

Caso base: $dist[s] = 0 = dist(s,s)$

Hip. Inductiva: $dist[u] = dist(s,u) \ \ \forall u \text{ tq } dist(s,u) \leq k$

Paso inductivo: Tomo v con $dist(s,v) = k+1$



* Se que v es ALCANZABLE $\Rightarrow$ va a ser descubierto (LEMA 1)

* v es descubierto a partir de un nodo $u \Rightarrow d(s,v) \leq d(s,u)+1$

* v es descubierto a partir de un nodo $u \Rightarrow d(s,v) \leq d(s,u)+1 \Rightarrow \mathrm{d(s,u)} \geq d(s,v) - 1 = k$

* v es descubierto después de que u salga de la cola $\Rightarrow d(s,u) < d(s,v)+1 = k+1$

* $\Rightarrow k \leq d(s,u) < k+1 \Rightarrow d(s,u) = k \Rightarrow$ (por H.I.) $dist[u] = d(s,u) = k$

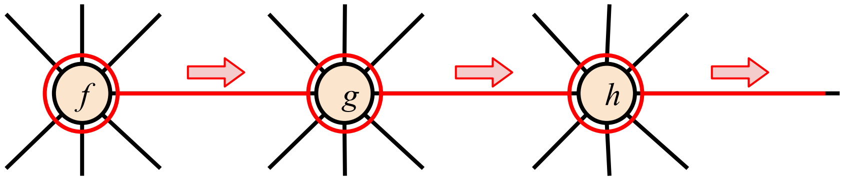
* $dist[v] \leftarrow dist[u] + 1 = k+1 \Rightarrow dist[v] = dist(s,v)$

Depth First Search (DFS)

- Objetivo:
  - Visitar todos los nodos.
    - generar un bosque a través de los caminos generados.
    - clasificar aristas
    - detectar ciclos
    - ordenar secuencias de estados (*Algoritmo topological sort*)
    - detectar componentes fuertemente conexas (*Algoritmo de Kosaraju*)
  - $\Theta(V+E)$

Depth First Search (DFS)

- Idea:
  - Empiezo por un nodo y voy visitando a un vecino, a un vecino de este vecino, etc… hasta agotar (en profundidad; luego empiezo por otro; y así siguiendo



- *Vamos a armarlo de forma recursiva y con backtracking hasta donde encuentre un nuevo camino para ir en profundidad.*
- *¡CUIDADO! Es importante guardar registro para no volver a explorar nodos ya visitados.*

Deep First Search (DFS) recursivo

```
DFS-visit ( Adj, u) :

|     for v in Adj[ u ] :
|     |     if v not in parent :
|     |     |     parent [ v ] = u
|     |     |     DFS-visit( Adj, v )



DFS ( V, Adj ) :

|     parent = {}

|     for u in V :
|     |     if u not in parent :
|     |     |     parent [ u ] = None
|     |     |     DFS-visit( Adj, u )
```

Deep First Search (DFS) recursivo

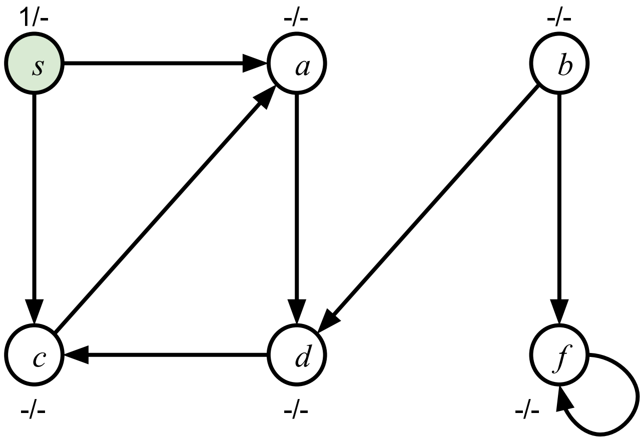
```
DFS-visit ( Adj, u) :
|     k += 1
|     start[u] = k
|     for v in Adj[ u ] :
|     |     if v not in parent :
|     |     |     parent [ v ] = u
|     |     |     DFS-visit( Adj, v )
|     k += 1
|     finish[ u ] = k

DFS ( V, Adj ) :
|     start = {}
|     finish = {}
|     parent = {}
|     k = 0
|     for u in V :
|     |     if u not in parent :
|     |     |     parent [ u ] = None
|     |     |     DFS-visit( Adj, u )
```

Deep First Search (DFS) recursivo

```
DFS-visit ( Adj, u) :
|     k += 1
|     start[u] = k
|     for v in Adj[ u ] :
|     |     if v not in parent :
|     |     |     parent [ v ] = u
|     |     |     DFS-visit( Adj, v )
|     k += 1
|     finish[ u ] = k

DFS ( V, Adj ) :
|     start = {}
|     finish = {}
|     parent = {}
|     k = 0
|     for u in V :
|     |     if u not in parent :
|     |     |     parent [ u ] = None
|     |     |     DFS-visit( Adj, u )
```



start = {}
finish = {}
parent = {}
k = 0

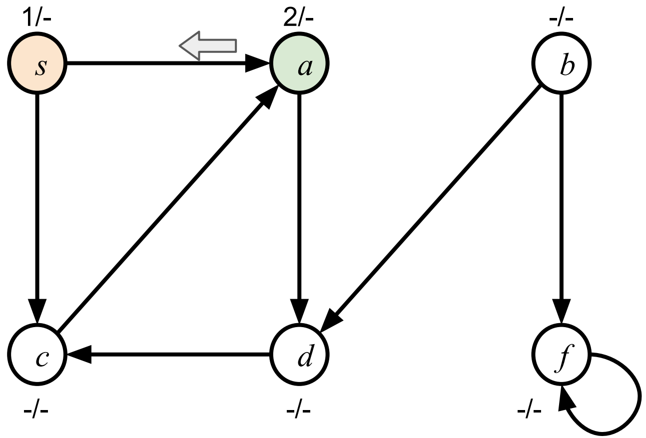
parent [s] = None
DFS-visit(Adj, s)

k = 1
start [s] = 1
Adj[s] = {a, c}
parent[a] = s

Deep First Search (DFS) recursivo

```
DFS-visit ( Adj, u) :
|     k += 1
|     start[u] = k
|     for v in Adj[ u ] :
|     |     if v not in parent :
|     |     |     parent [ v ] = u
|     |     |     DFS-visit( Adj, v )
|     k += 1
|     finish[ u ] = k

DFS ( V, Adj ) :
|     start = {}
|     finish = {}
|     parent = {}
|     k = 0
|     for u in V :
|     |     if u not in parent :
|     |     |     parent [ u ] = None
|     |     |     DFS-visit( Adj, u )
```



$start = \{s:1\}$

$finish = \{\}$

$parent = \{s:None\}$

$k = 1$

$k = 2$

$start\ [a] = 2$

$Adj[a] = \{d\}$

$parent[d] = a$

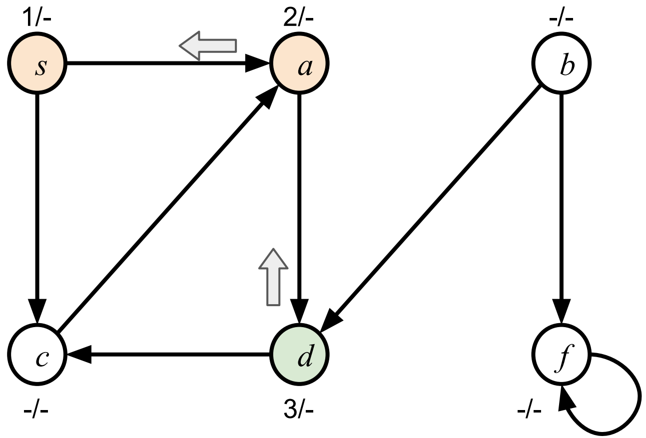
$parent\ [\ a\ ] = s$

$DFS\text{-}visit(\ Adj,\ a\ )$

Deep First Search (DFS) recursivo

```
DFS-visit ( Adj, u) :
|     k += 1
|     start[u] = k
|     for v in Adj[ u ] :
|     |     if v not in parent :
|     |     |     parent [ v ] = u
|     |     |     DFS-visit( Adj, v )
|     k += 1
|     finish[ u ] = k

DFS ( V, Adj ) :
|     start = {}
|     finish = {}
|     parent = {}
|     k = 0
|     for u in V :
|     |     if u not in parent :
|     |     |     parent [ u ] = None
|     |     |     DFS-visit( Adj, u )
```



start = {s:1, a:2}

finish = {}

parent = {s:None, a:s}

k = 2

k = 3

start [d] = 3

Adj[d] = {c}

parent[c] = d

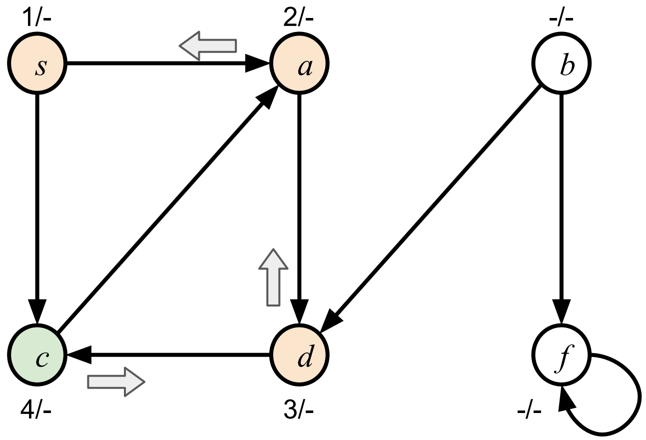
parent [d] = a

DFS-visit(Adj, d)

Deep First Search (DFS) recursivo

```
DFS-visit ( Adj, u) :
|     k += 1
|     start[u] = k
|     for v in Adj[ u ] :
|     |     if v not in parent :
|     |     |     parent [ v ] = u
|     |     |     DFS-visit( Adj, v )
|     k += 1
|     finish[ u ] = k

DFS ( V, Adj ) :
|     start = {}
|     finish = {}
|     parent = {}
|     k = 0
|     for u in V :
|     |     if u not in parent :
|     |     |     parent [ u ] = None
|     |     |     DFS-visit( Adj, u )
```



start = {s:1, a:2, d:3}

finish = {}

parent = {s:None, a:s, d:a}

k = 3

parent [c] = d

DFS-visit(Adj, c)

k = 4

start [c] = 4

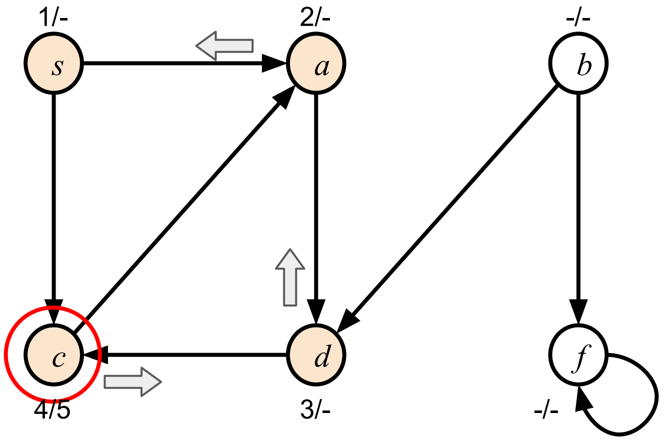
Adj[c] = {a}

 # NO SE CUMPLE EL IF()

Deep First Search (DFS) recursivo

```
DFS-visit ( Adj, u) :
|     k += 1
|     start[u] = k
|     for v in Adj[ u ] :
|     |     if v not in parent :
|     |     |     parent [ v ] = u
|     |     |     DFS-visit( Adj, v )
|     k += 1
|     finish[ u ] = k

DFS ( V, Adj ) :
|     start = {}
|     finish = {}
|     parent = {}
|     k = 0
|     for u in V :
|     |     if u not in parent :
|     |     |     parent [ u ] = None
|     |     |     DFS-visit( Adj, u )
```



start = {s:1, a:2, d:3, c:4}

finish = {c:5}

parent = {s:None, a:s, d:a, c:d}

k = 5

k = 4

start [c] = 4

Adj[c] = {a}

 # NO SE CUMPLE EL IF()

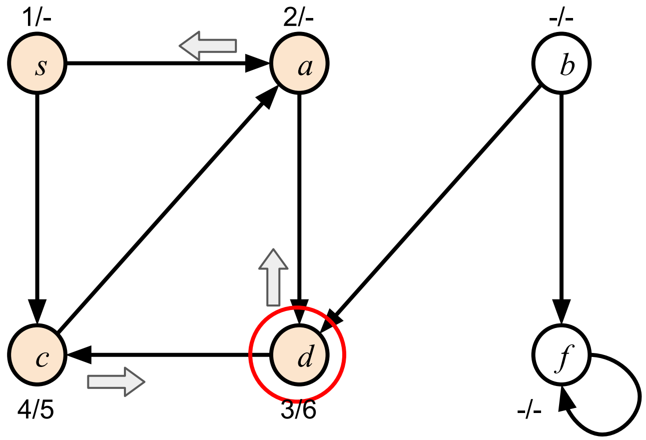
parent [c] = d

DFS-visit(Adj, c)

Deep First Search (DFS) recursivo

```
DFS-visit ( Adj, u) :
|     k += 1
|     start[u] = k
|     for v in Adj[ u ] :
|     |     if v not in parent :
|     |     |     parent [ v ] = u
|     |     |     DFS-visit( Adj, v )
|     k += 1
|     finish[ u ] = k

DFS ( V, Adj ) :
|     start = {}
|     finish = {}
|     parent = {}
|     k = 0
|     for u in V :
|     |     if u not in parent :
|     |     |     parent [ u ] = None
|     |     |     DFS-visit( Adj, u )
```



start = {s:1, a:2, d:3, c:4}

finish = {c:5, d:6}

parent = {s:None, a:s, d:a, c:d}

k = 5

k += 1

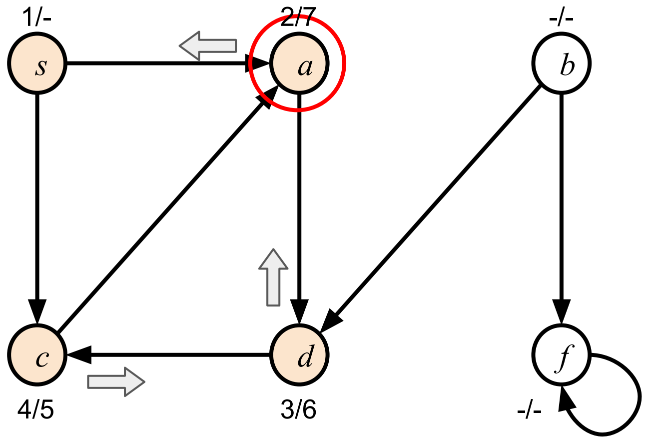
k = 6

 # NO SE CUMPLE EL IF()

Deep First Search (DFS) recursivo

```
DFS-visit ( Adj, u) :
|     k += 1
|     start[u] = k
|     for v in Adj[ u ] :
|     |     if v not in parent :
|     |     |     parent [ v ] = u
|     |     |     DFS-visit( Adj, v )
|     k += 1
|     finish[ u ] = k

DFS ( V, Adj ) :
|     start = {}
|     finish = {}
|     parent = {}
|     k = 0
|     for u in V :
|     |     if u not in parent :
|     |     |     parent [ u ] = None
|     |     |     DFS-visit( Adj, u )
```



start = {s:1, a:2, d:3, c:4}

finish = {c:5, d:6, a:7}

parent = {s:None, a:s, d:a, c:d}

k = 6

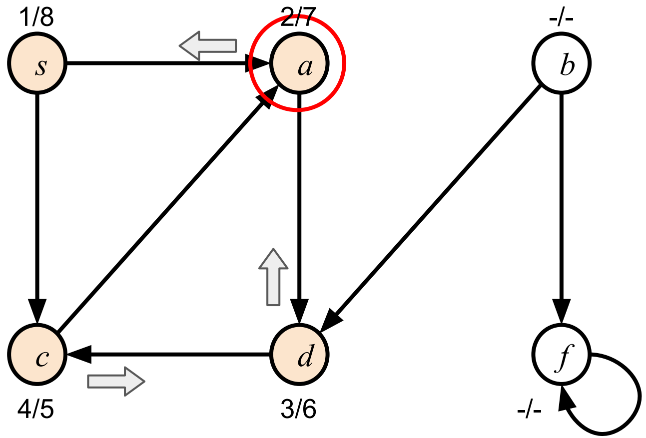
k = 7

 # NO SE CUMPLE EL IF()

Deep First Search (DFS) recursivo

```
DFS-visit ( Adj, u) :
|     k += 1
|     start[u] = k
|     for v in Adj[ u ] :
|     |     if v not in parent :
|     |     |     parent [ v ] = u
|     |     |     DFS-visit( Adj, v )
|     k += 1
|     finish[ u ] = k

DFS ( V, Adj ) :
|     start = {}
|     finish = {}
|     parent = {}
|     k = 0
|     for u in V :
|     |     if u not in parent :
|     |     |     parent [ u ] = None
|     |     |     DFS-visit( Adj, u )
```



start = {s:1, a:2, d:3, c:4}

finish = {c:5, d:6, a:7, s:8}

parent = {s:None, a:s, d:a, c:d}

k = 7

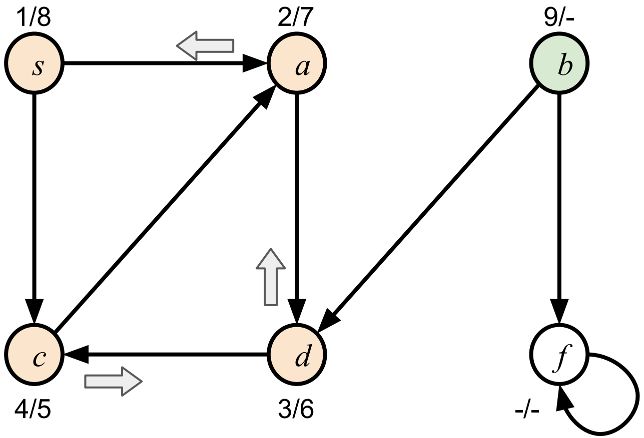
k = 8

 # NO SE CUMPLE EL IF()

Deep First Search (DFS) recursivo

```
DFS-visit ( Adj, u) :
|     k += 1
|     start[u] = k
|     for v in Adj[ u ] :
|     |     if v not in parent :
|     |     |     parent [ v ] = u
|     |     |     DFS-visit( Adj, v )
|     k += 1
|     finish[ u ] = k

DFS ( V, Adj ) :
|     start = {}
|     finish = {}
|     parent = {}
|     k = 0
|     for u in V :
|     |     if u not in parent :
|     |     |     parent [ u ] = None
|     |     |     DFS-visit( Adj, u )
```



start = {s:1, a:2, d:3, c:4}

finish = {c:5, d:6, a:7, s:8}

parent = {s:None, a:s, d:a, c:d}

k = 8

k = 9

start [b] = 9

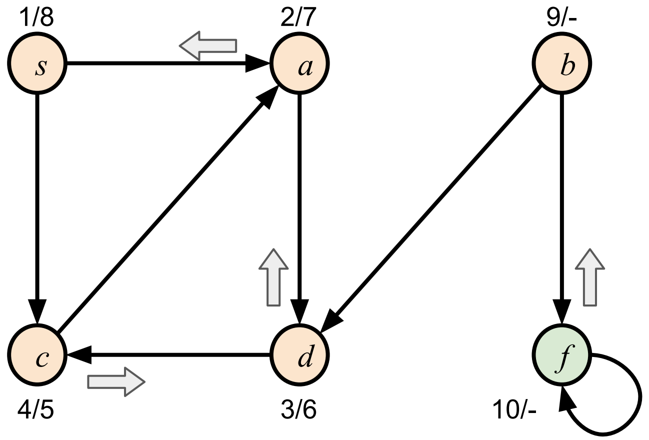
Adj[b] = {d, f}

parent[f] = b

Deep First Search (DFS) recursivo

```
DFS-visit ( Adj, u) :
|     k += 1
|     start[u] = k
|     for v in Adj[ u ] :
|     |     if v not in parent :
|     |     |     parent [ v ] = u
|     |     |     DFS-visit( Adj, v )
|     k += 1
|     finish[ u ] = k

DFS ( V, Adj ) :
|     start = {}
|     finish = {}
|     parent = {}
|     k = 0
|     for u in V :
|     |     if u not in parent :
|     |     |     parent [ u ] = None
|     |     |     DFS-visit( Adj, u )
```



start = {s:1, a:2, d:3, c:4, b:9}

finish = {c:5, d:6, a:7, s:8}

parent = {s:None, a:s, d:a, c:d, b:None, f:b}

k = 9

k = 10

start [b] = 10

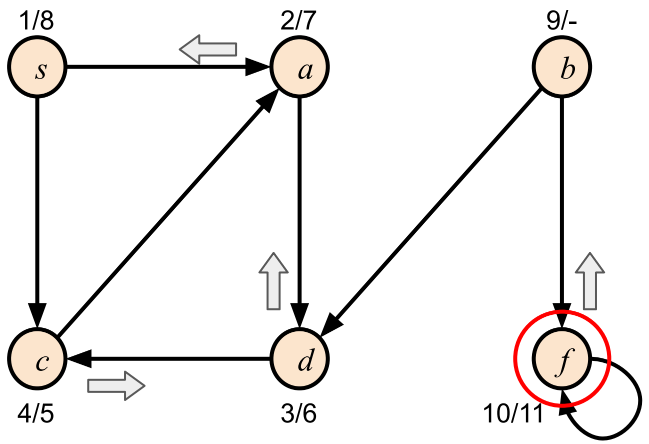
Adj[f] = {f}

 # NO SE CUMPLE EL IF()

Deep First Search (DFS) recursivo

```
DFS-visit ( Adj, u) :
|     k += 1
|     start[u] = k
|     for v in Adj[ u ] :
|     |     if v not in parent :
|     |     |     parent [ v ] = u
|     |     |     DFS-visit( Adj, v )
|     k += 1
|     finish[ u ] = k

DFS ( V, Adj ) :
|     start = {}
|     finish = {}
|     parent = {}
|     k = 0
|     for u in V :
|     |     if u not in parent :
|     |     |     parent [ u ] = None
|     |     |     DFS-visit( Adj, u )
```



start = {s:1, a:2, d:3, c:4, b:9, f:10}

finish = {c:5, d:6, a:7, s:8, f:11}

parent = {s:None, a:s, d:a, c:d, b:None, f:b}

k = 11

k = 10

start [b] = 10

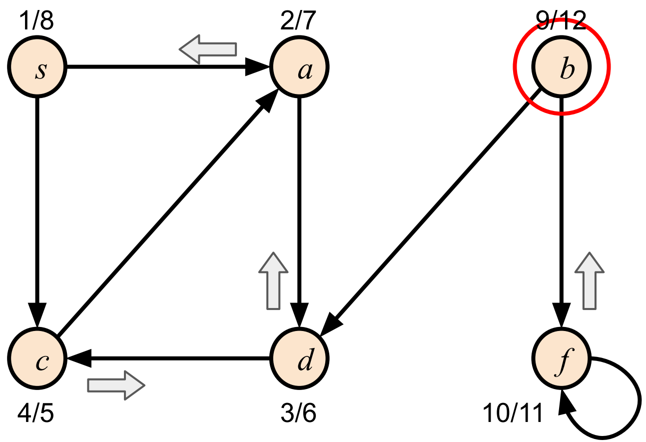
Adj[f] = {f}

 # NO SE CUMPLE EL IF()

Deep First Search (DFS) recursivo

```
DFS-visit ( Adj, u) :
|     k += 1
|     start[u] = k
|     for v in Adj[ u ] :
|     |     if v not in parent :
|     |     |     parent [ v ] = u
|     |     |     DFS-visit( Adj, v )
|     k += 1
|     finish[ u ] = k

DFS ( V, Adj ) :
|     start = {}
|     finish = {}
|     parent = {}
|     k = 0
|     for u in V :
|     |     if u not in parent :
|     |     |     parent [ u ] = None
|     |     |     DFS-visit( Adj, u )
```



start = {s:1, a:2, d:3, c:4, b:9, f:10}

finish = {c:5, d:6, a:7, s:8, f:11, b:12}

parent = {s:None, a:s, d:a, c:d, b:None, f:b}

k = 12

k = 12

 # NO SE CUMPLE EL IF()

Deep First Search (DFS) recursivo

```
DFS-visit ( Adj, u) :
|     k += 1
|     start[u] = k
|     for v in Adj[ u ] :
|     |     if v not in parent :
|     |     |     parent [ v ] = u
|     |     |     DFS-visit( Adj, v )
|     k += 1
|     finish[ u ] = k

DFS ( V, Adj ) :
|     start = {}
|     finish = {}
|     parent = {}
|     k = 0
|     for u in V :
|     |     if u not in parent :
|     |     |     parent [ u ] = None
|     |     |     DFS-visit( Adj, u )
```

$\Rightarrow$ ***O(V+E)***

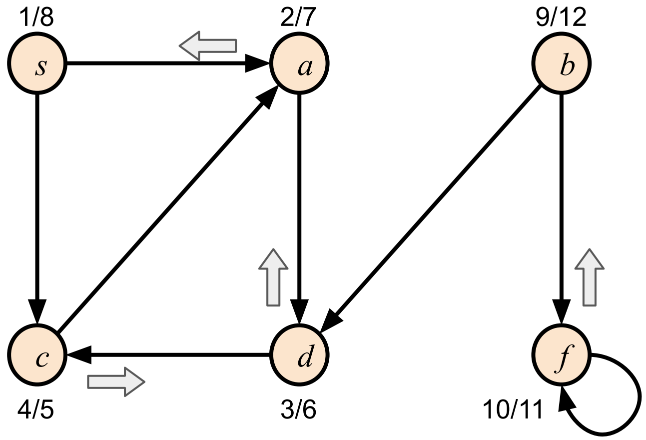
Cada vértice entra a la lista sólo una vez (y se explora sólo una vez)
$\Rightarrow$ *O(V)*

Cada vencindario (Adj[u]) se explora sólo una vez
$\Rightarrow$

$$\sum_{u \in V} |Adj[u]| = \begin{cases} |E| \; para \; grafos \; dirigidos \\ 2|E| \; para \; grafos \; no \; dirigidos \end{cases}$$

$\Rightarrow$ *O(E)*

Deep First Search (DFS) recursivo



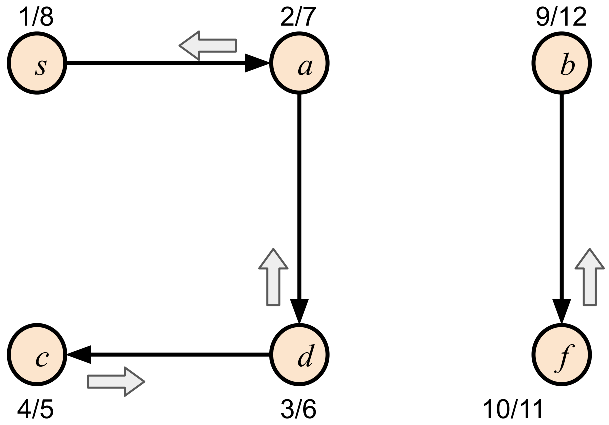
$start = \{s{:}1,\ a{:}2,\ d{:}3,\ c{:}4,\ b{:}9,\ f{:}10\}$

$finish = \{c{:}5,\ d{:}6,\ a{:}7,\ s{:}8,\ f{:}11,\ b{:}12\}$

$parent = \{s{:}None,\ a{:}s,\ d{:}a,\ c{:}d,\ b{:}None,\ f{:}b\}$

DFS: Clasificación de aristas

Tree edges (aristas): Formar el **bosque**



start = {s:1, a:2, d:3, c:4, b:9, f:10}

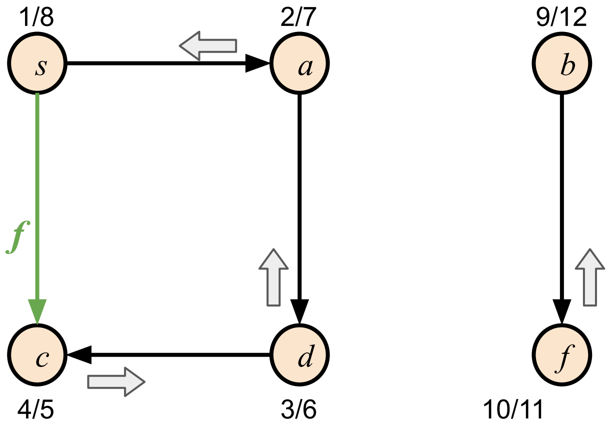
finish = {c:5, d:6, a:7, s:8, f:11, b:12}

parent = {s:None, a:s, d:a, c:d, b:None, f:b}

DFS: Clasificación de aristas

Tree edges (aristas): Formar el **bosque**

Forward edges (aristas) (*f*): Van hacia un descendiente.



start = {s:1, a:2, d:3, c:4, b:9, f:10}

finish = {c:5, d:6, a:7, s:8, f:11, b:12}

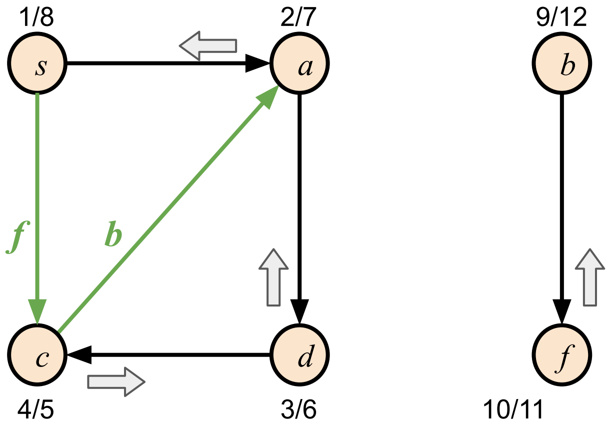
parent = {s:None, a:s, d:a, c:d, b:None, f:b}

DFS: Clasificación de aristas

Tree edges (aristas): Formar el **bosque**

Forward edges (aristas) (***f***): Van hacia un descendiente.

Backward edges (aristas) (***b***): Van hacia un ancestro (predecesor).



start = {s:1, a:2, d:3, c:4, b:9, f:10}

finish = {c:5, d:6, a:7, s:8, f:11, b:12}

parent = {s:None, a:s, d:a, c:d, b:None, f:b}

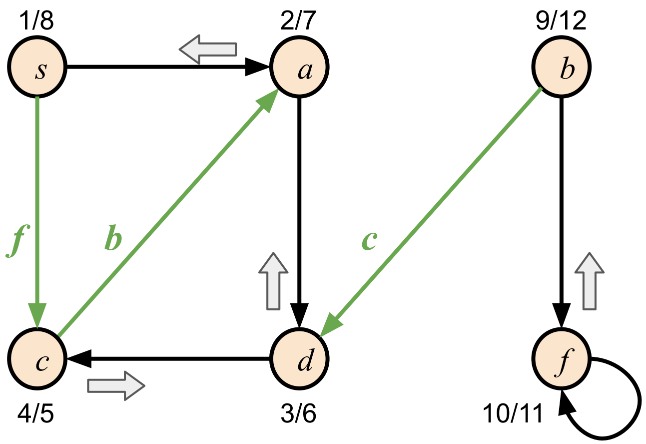
DFS: Clasificación de aristas

Tree edges (aristas): Formar el **bosque**

Forward edges (aristas) (***f***): Van hacia un descendiente.

Backward edges (aristas) (***b***): Van hacia un ancestro (predecesor).

Cross-edges (aristas) (***c***): Van a otro árbol del bosque.



start = {s:1, a:2, d:3, c:4, b:9, f:10}

finish = {c:5, d:6, a:7, s:8, f:11, b:12}

parent = {s:None, a:s, d:a, c:d, b:None, f:b}

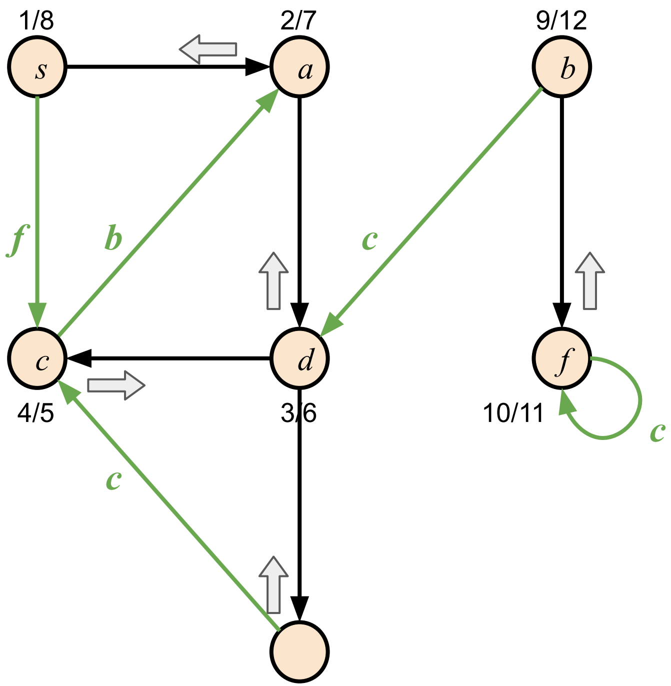
DFS: Clasificación de aristas

Tree edges (aristas): Formar el **bosque**

Forward edges (aristas) (***f***): Van hacia un descendiente.

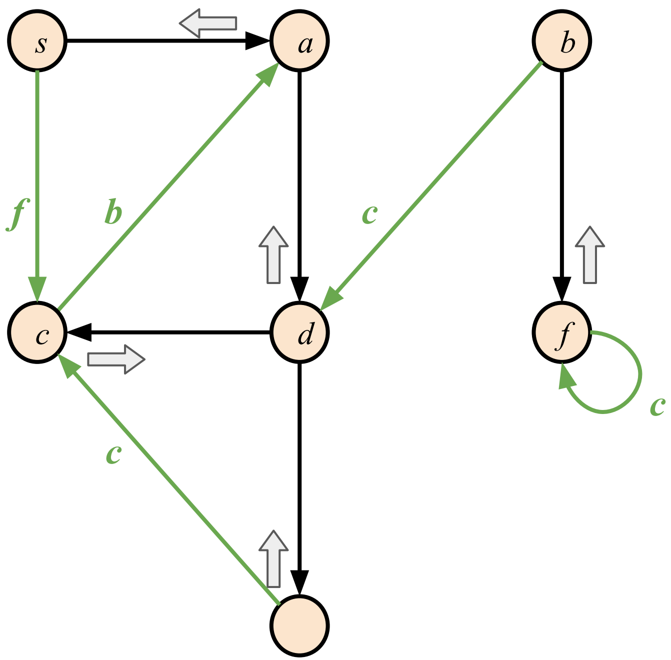
Backward edges (aristas) (***b***): Van hacia un ancestro (predecesor).

Cross-edges (aristas) (***c***): Van a otro árbol del bosque, ó entre ramas (sin relación de parentesco).



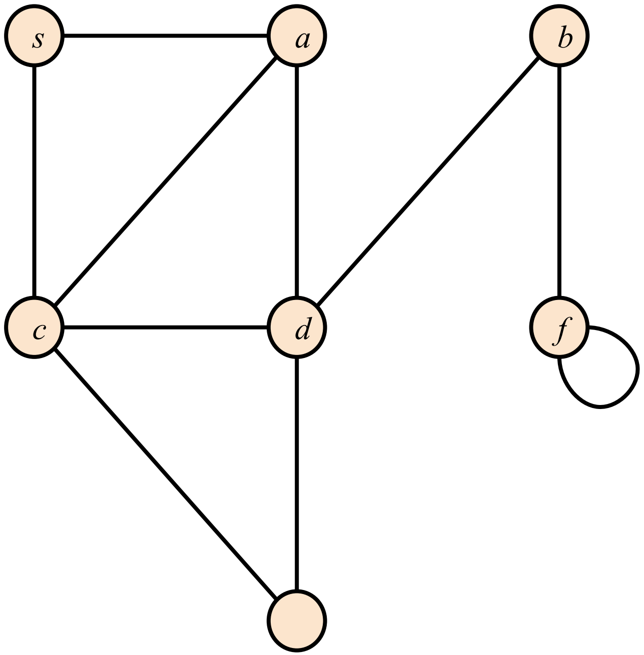
DFS: Clasificación de aristas

	Directed	*Undirected*
Tree	*X*	
Forward	*X*	
Backward	*X*	
Cross	*X*	



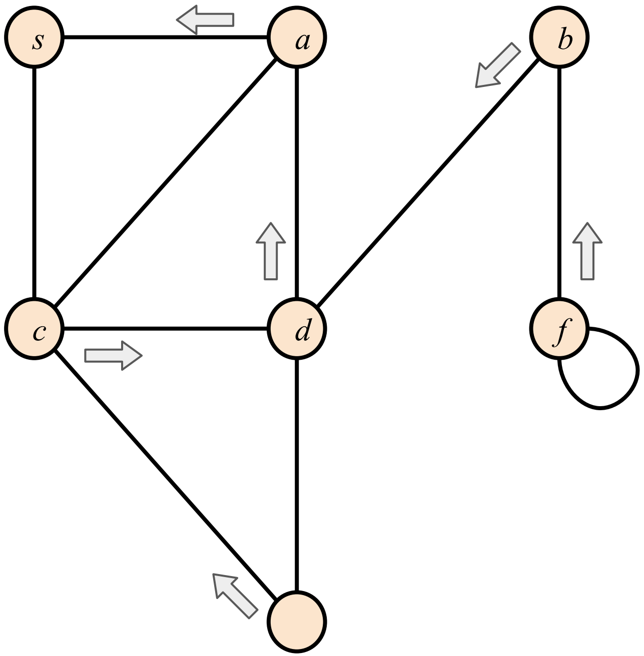
DFS: Clasificación de aristas

	Directed	*Undirected*
Tree	*X*	
Forward	*X*	
Backward	*X*	
Cross	*X*	



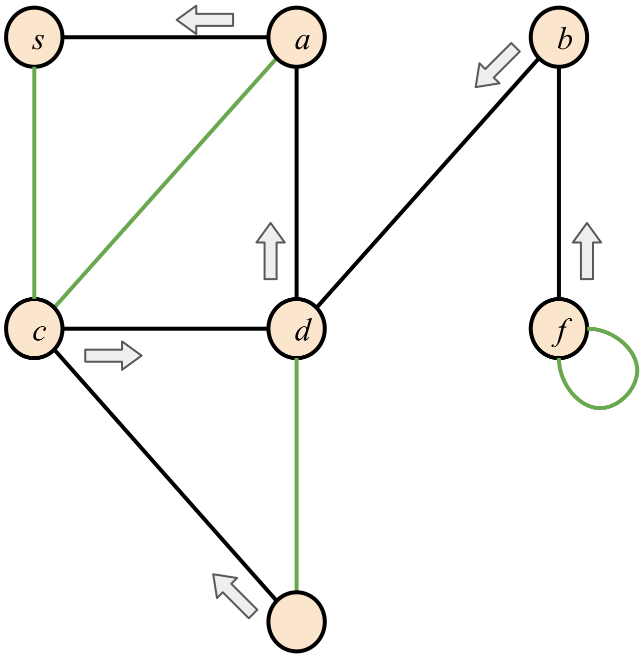
DFS: Clasificación de aristas

	Directed	*Undirected*
Tree	*X*	
Forward	*X*	
Backward	*X*	
Cross	*X*	



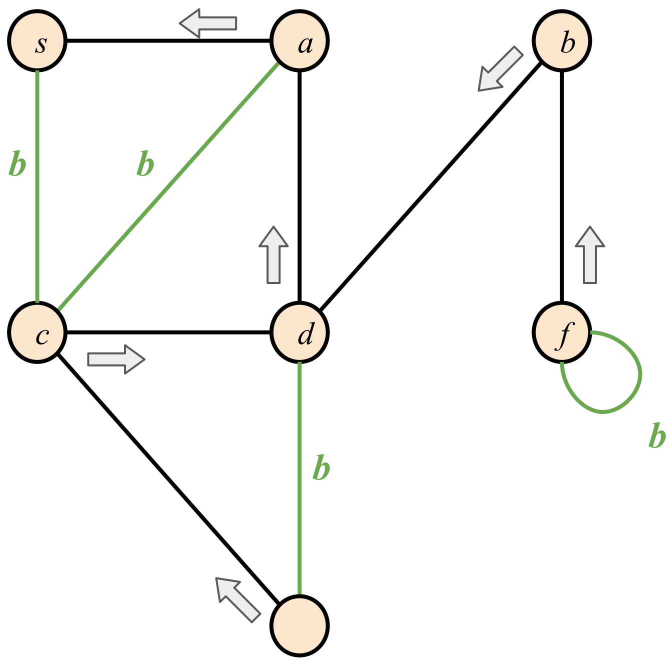
DFS: Clasificación de aristas

	Directed	*Undirected*
Tree	*X*	
Forward	*X*	
Backward	*X*	
Cross	*X*	

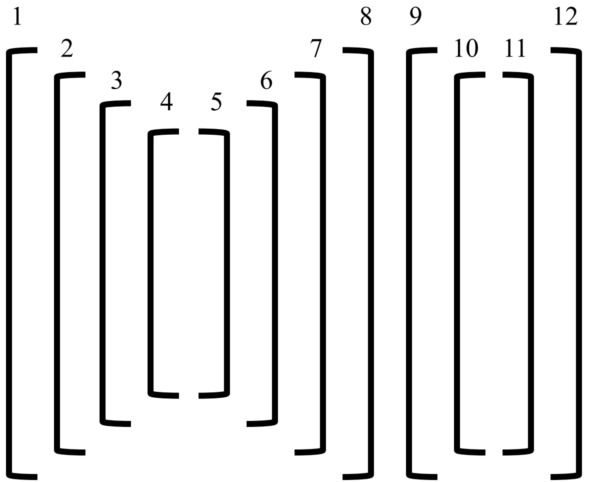


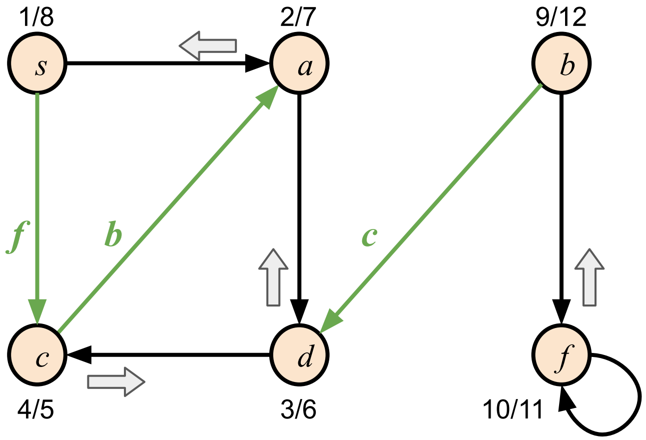
DFS: Clasificación de aristas

	Directed	*Undirected*
Tree	*X*	*X*
Forward	*X*	
Backward	*X*	*X*
Cross	*X*	



Otra manera de verlo…



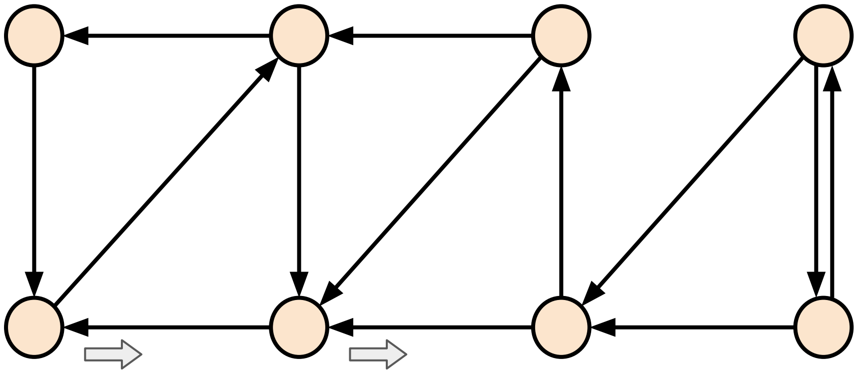


start = {*s*:1, *a*:2, *d*:3, *c*:4, *b*:9, *f*:10}

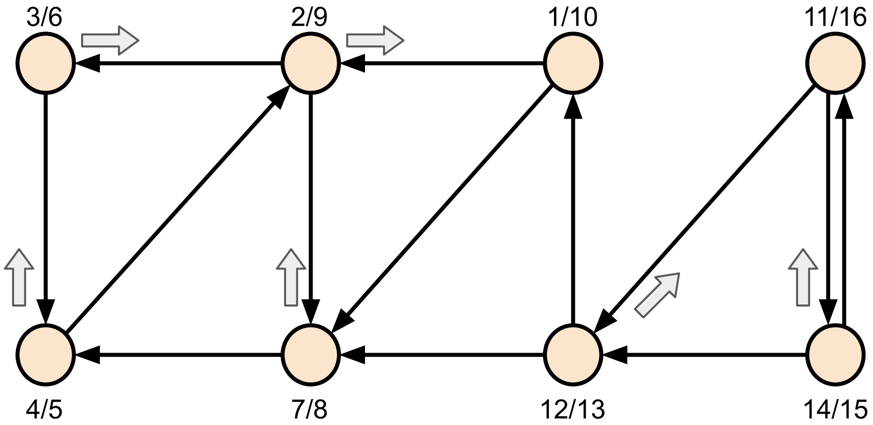
finish = {*c*:5, *d*:6, *a*:7, *s*:8, *f*:11, *b*:12}

parent = {*s*:None, *a*:*s*, *d*:*a*, *c*:*d*, *b*:None, *f*:*b*}

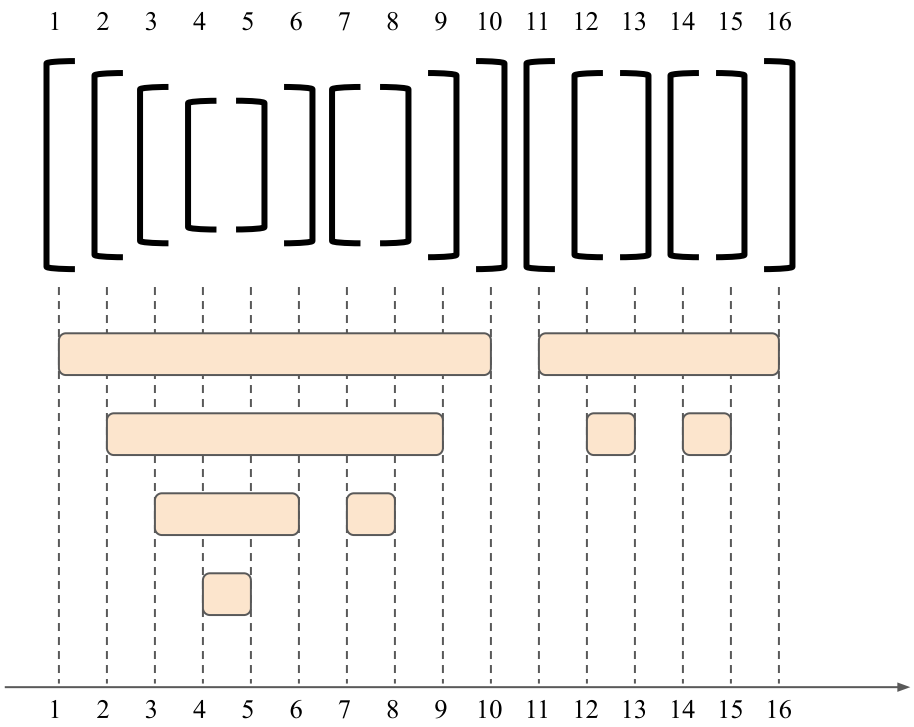
Otra manera de verlo…

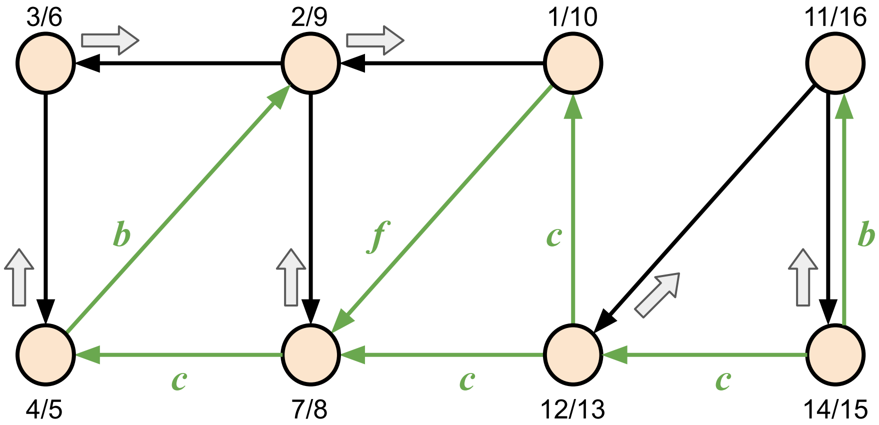


Otra manera de verlo…



Otra manera de verlo…





Detección de ciclos (con DFS)

Teorema 1: Dado un digrafo G = (V, E),

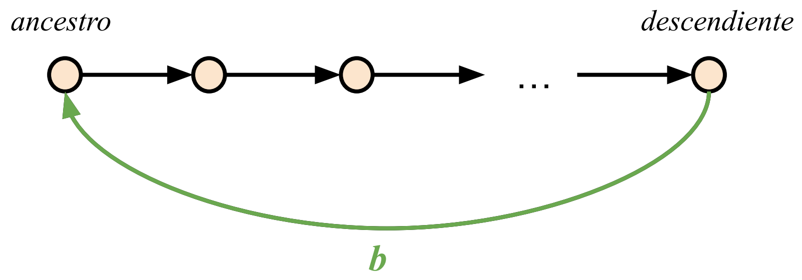
G tiene un ciclo $\Leftrightarrow$ el bosque DFS tiene una arista backward

Detección de ciclos (con DFS)

Teorema 1: Dado un digrafo G = (V, E),

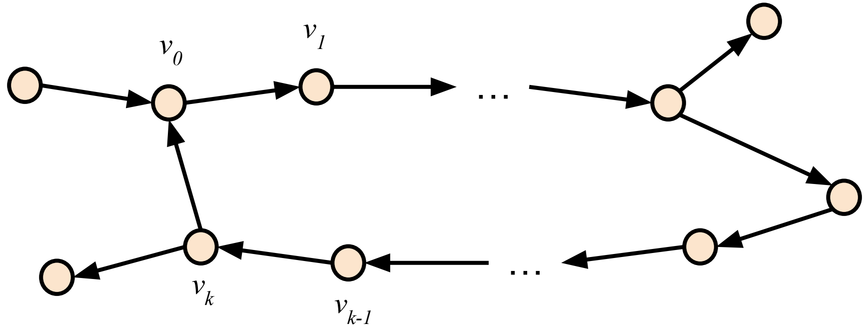
G tiene un ciclo ⇔ el bosque DFS tiene una arista backward

Demostración (⇐):



Lo mismo vale para un no dirigido.

Demostración (⇒):



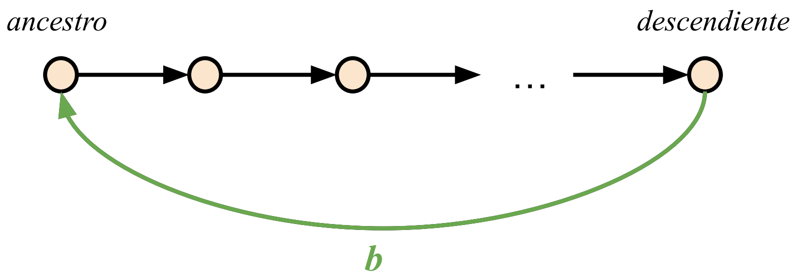
❏

Detección de ciclos (con DFS)

Teorema 1: Dado un digrafo G = (V, E),

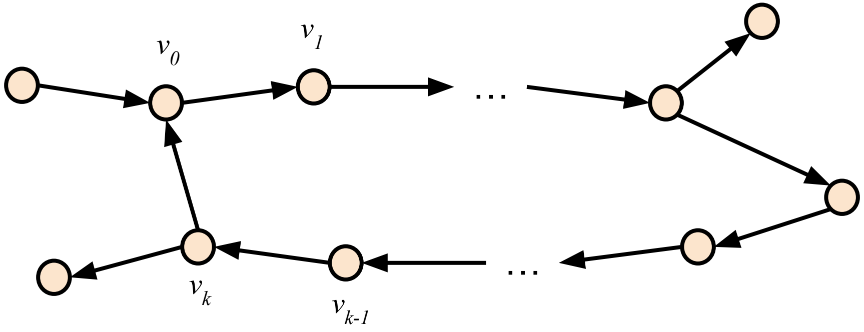
G tiene un ciclo ⇔ el bosque DFS tiene una arista backward

Demostración (⇐):



Lo mismo vale para un no dirigido.

Demostración (⇒):



❑

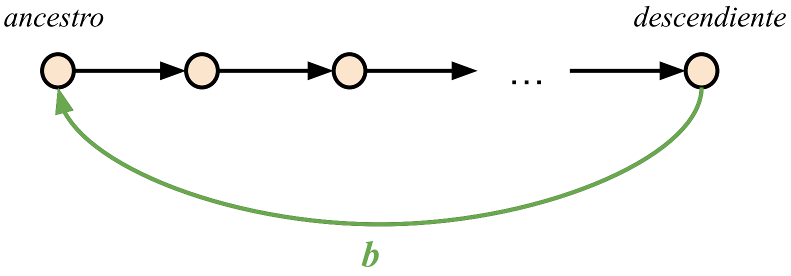
Detección de ciclos (con DFS)

v_0 es el primer vértice visitado por DFS dentro del ciclo (puede no ser el primero de todos.

Teorema 1: Dado un digrafo G = (V, E),

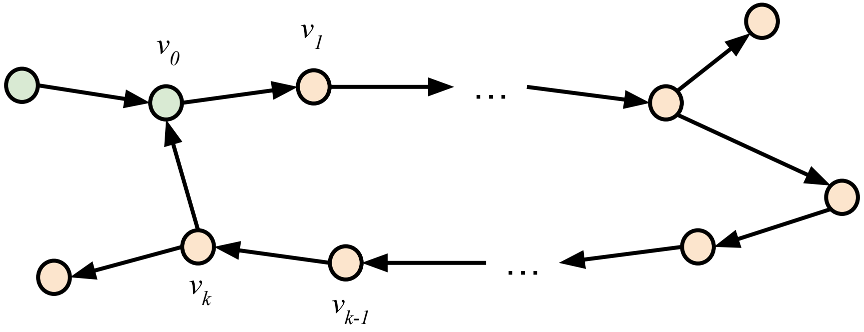
G tiene un ciclo ⇔ el bosque DFS tiene una arista backward

Demostración (⇐):



Lo mismo vale para un no dirigido.

Demostración (⇒):



❑

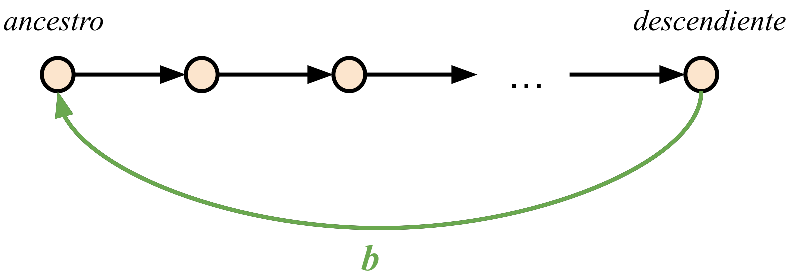
Detección de ciclos (con DFS)

v_0 es el primer vértice visitado por DFS dentro del ciclo (puede no ser el primero de todos.

Teorema 1: Dado un digrafo G = (V, E),

G tiene un ciclo ⇔ el bosque DFS tiene una arista backward

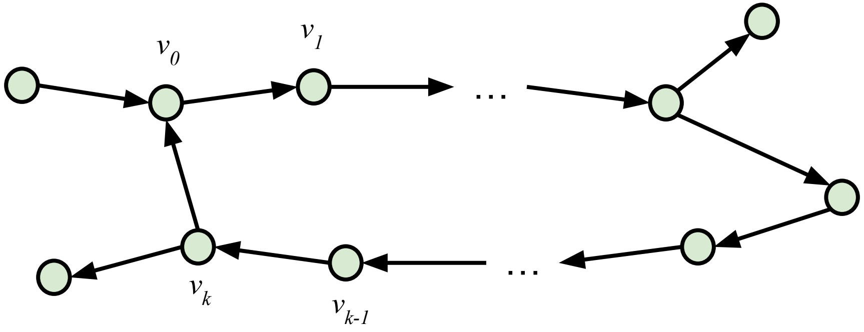
Demostración (⇐):



Lo mismo vale para un no dirigido.

❑

Demostración (⇒):



Como existe un camino v_0 , v_1 , … v_{k-1} , v_k (es un ciclo), entonces v_0 va a ser ancestro de v_k .

Y desde v_k no se vuelve a visitar v_0 , entonces

(v_k , v_0) tiene que es una arista ***backward***.

❑

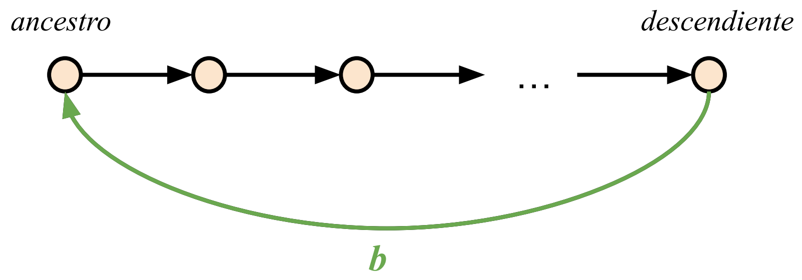
Detección de ciclos (con DFS)

v_0 es el primer vértice visitado por DFS dentro del ciclo (puede no ser el primero de todos.

Teorema 1: Dado un digrafo G = (V, E),

G tiene un ciclo ⇔ el bosque DFS tiene una arista backward

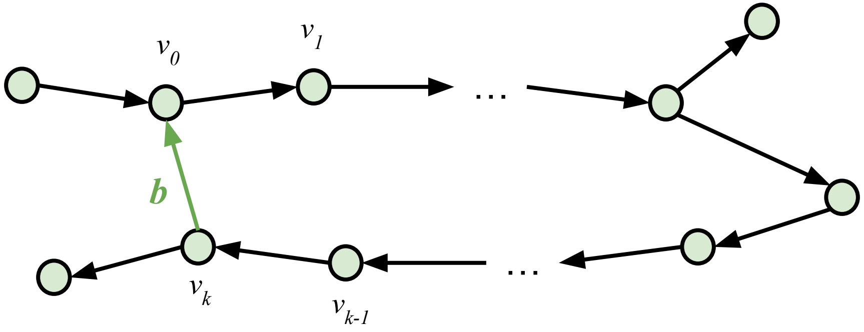
Demostración (⇐):



Lo mismo vale para un no dirigido.

❏

Demostración (⇒):



Como existe un camino v_0 , v_1 , … v_{k-1} , v_k (es un ciclo), entonces v_0 va a ser ancestro de v_k .

Y desde v_k no se vuelve a visitar v_0 , entonces

(v_k , v_0) tiene que es una arista ***backward***.

❏

Topological sort (Job Scheduling)

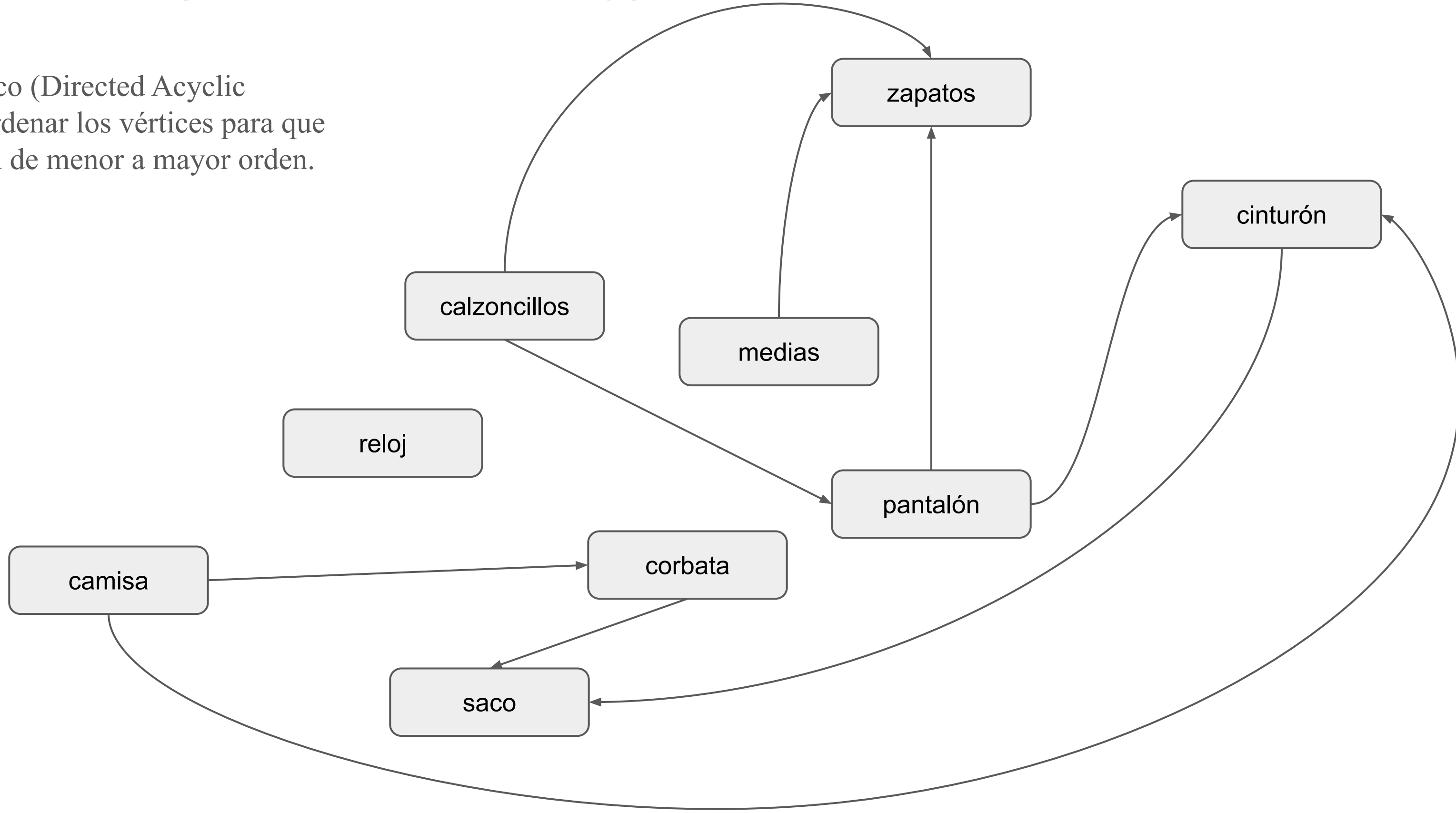
Dado un Digrafo Acíclico (Directed Acyclic Graph, DAG), quiero ordenar los vértices para que todas las aristas apunten de menor a mayor orden.

Topological sort (Job Scheduling)

Dado un Digrafo Acíclico (Directed Acyclic Graph, DAG), quiero ordenar los vértices para que todas las aristas apunten de menor a mayor orden.

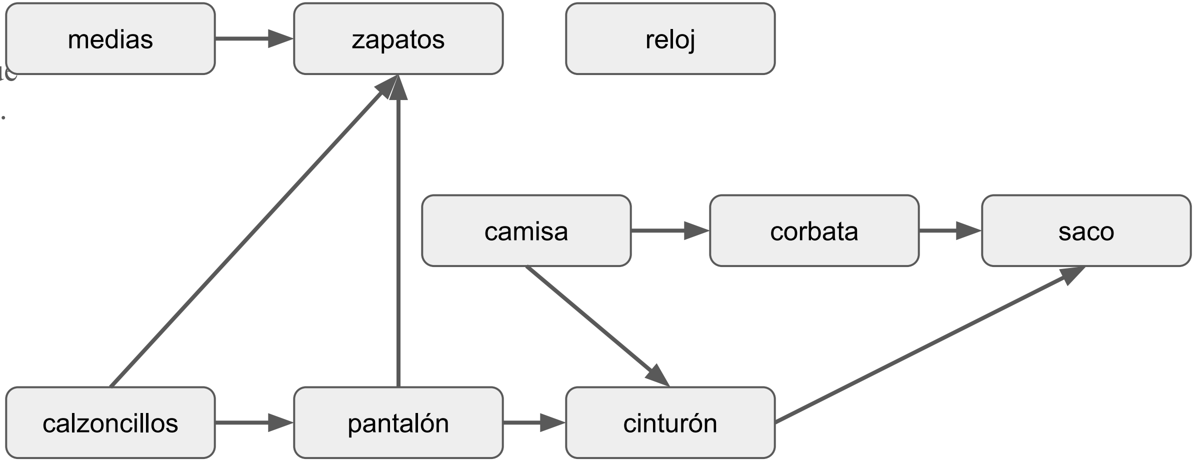
Topological sort (Job Scheduling)

Dado un Digrafo Acíclico (Directed Acyclic Graph, DAG), quiero ordenar los vértices para que todas las aristas apunten de menor a mayor orden.



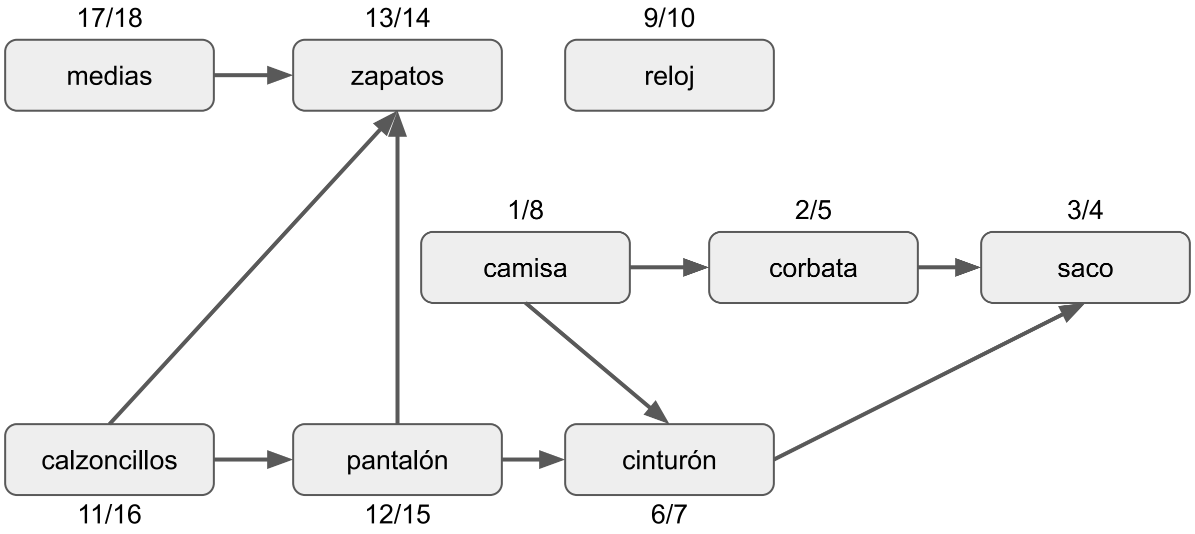
Topological sort (Job Scheduling)

Dado un Digrafo Acíclico (Directed Acyclic Graph, DAG), quiero ordenar los vértices para que todas las aristas apunten de menor a mayor orden.



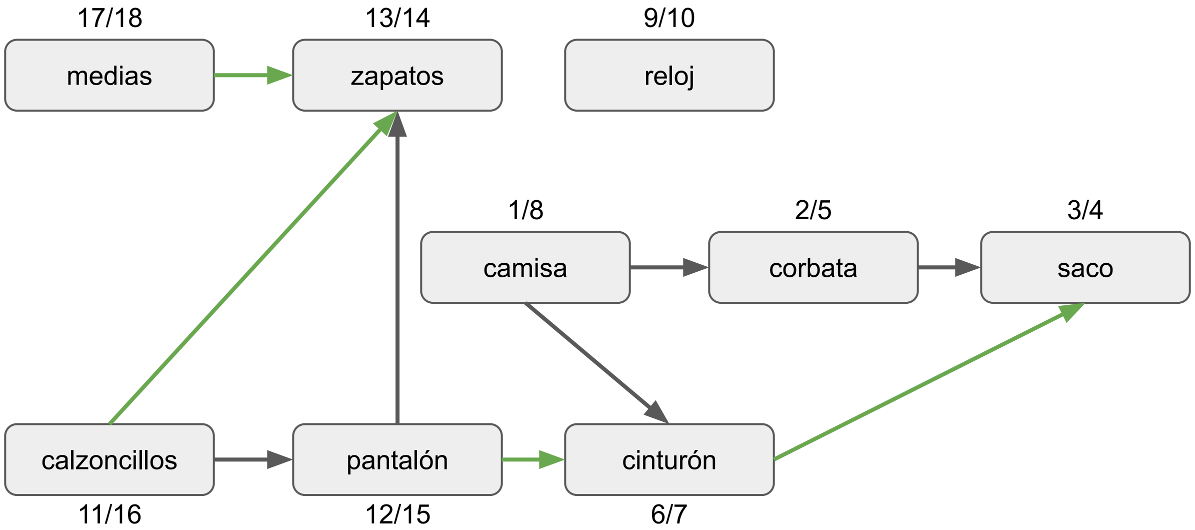
Topological sort (Job Scheduling)

```
Topological_sort ( G ) :
|     DFS ( G )
|     return invertir finish
```



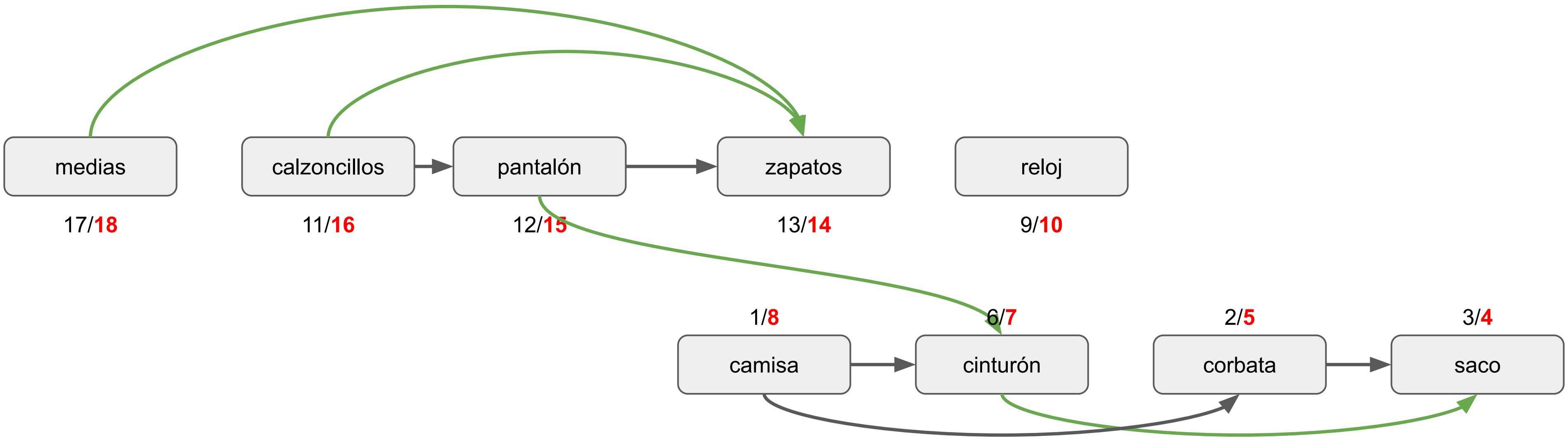
Topological sort (Job Scheduling)

```
Topological_sort ( G ) :
|    DFS ( G )
|    return invertir finish
```



Topological sort (Job Scheduling)

```
Topological_sort ( G ) :
|     DFS ( G )
|     return invertir finish
```



Topological sort (Job Scheduling)

```
Topological_sort ( G ) :
|    DFS ( G )
|    return invertir finish
```

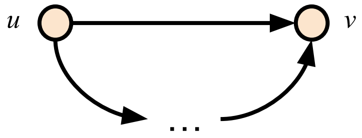
Teorema 1: Dado un digrafo G = (V, E) sin ciclos (DAG),

todas las aristas van de menor a mayor $\Leftrightarrow$ $\forall u,v \,/\, (u,v) \in$: $finish[u] > finish[v]$

Demostración:

Podría existir otro camino que conecte u y v, entonces

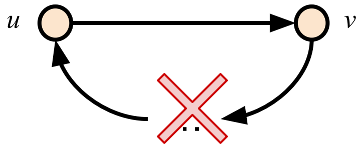
<u>Caso 1</u>: start[u] < start[v]



Por cualquier camino va a valer que $finish[u] > finish[v]$

❑

<u>Caso 2</u>: start[u] > start[v]



Esto no puede ocurrir porque pedí que no tenga ciclos.

❑

Componentes Fuertemente Conexas (c.f.c.) (Kosaraju)

- ➔ Aplicaciones: Descomponer en c.f.c. es el punto de partida (requisito) de muchos algoritmos.
- ➔ Idea:
  - ◆ G y G^T tienen las mismas c.f.c.

G^T?

Dado $G = (V, E)$,

$G^T = (V, E^T)$ con $E^T = \{(u,v) : (v,u) \in E\}$

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \qquad A^T = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

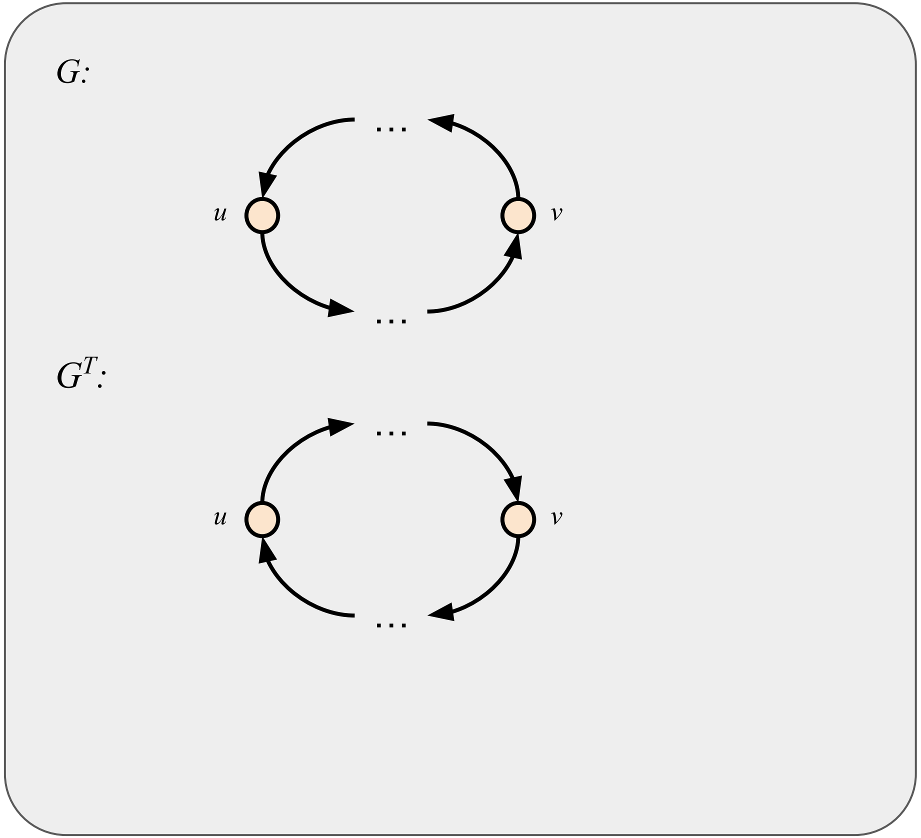
$(1, 2) : (2, 1)$

Con listas de adyacencia es $O(V+E)$ trasponer.

Componentes Fuertemente Conexas (c.f.c.) (Kosaraju)

➔ Aplicaciones: Descomponer en c.f.c. es el punto de partida (requisito) de muchos algoritmos.

➔ Idea:
  - G y G^T tienen las mismas c.f.c.



Componentes Fuertemente Conexas (c.f.c.) (Kosaraju)

- Aplicaciones: Descomponer en c.f.c. es el punto de partida (requisito) de muchos algoritmos.

- Idea:
  - G y G^T tienen las mismas c.f.c.
  - lo recorro en una dirección y después en la inversa, y si es posible entonces conexo!

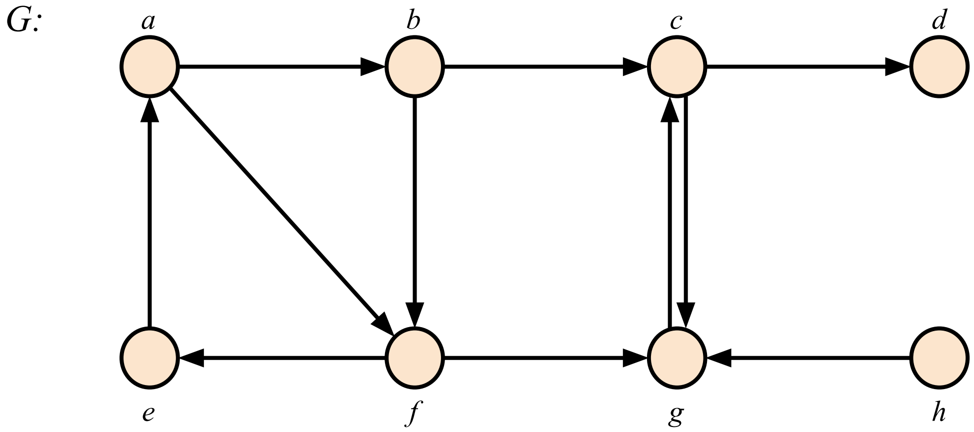
Kosaraju (componentes fuertemente conexas)

```
KOSARAJU ( G ) :
|     computo Gᵀ
|     DFS ( Gᵀ )
|     DFS ( G )  # usando finish de Gᵀ
                 en sentido inverso
```

Los árboles resultantes son las componentes conexas.

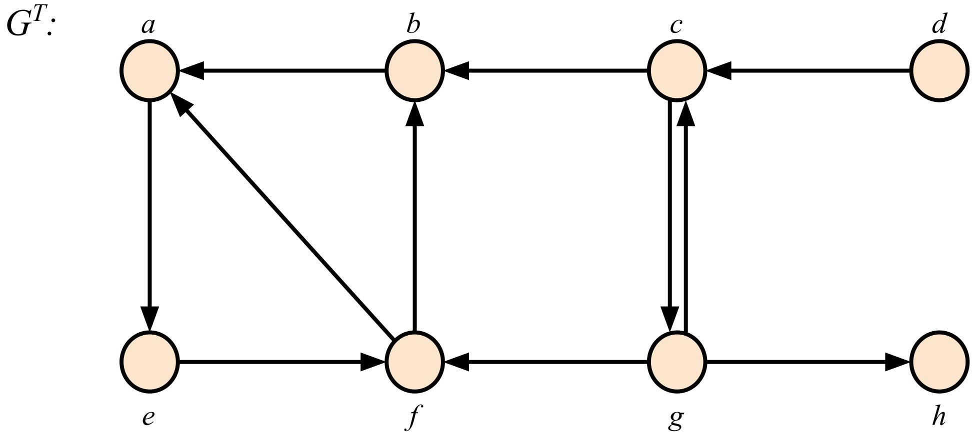
Kosaraju (componentes fuertemente conexas)

```
KOSARAJU ( G ) :
|     computo Gᵀ
|     DFS ( Gᵀ )
|     DFS ( G )  # usando finish de Gᵀ
                 en sentido inverso
```



Kosaraju (componentes fuertemente conexas)

```
KOSARAJU ( G ) :
|     computo Gᵀ
|     DFS ( Gᵀ )
|     DFS ( G )  # usando finish de Gᵀ
                 en sentido inverso
```



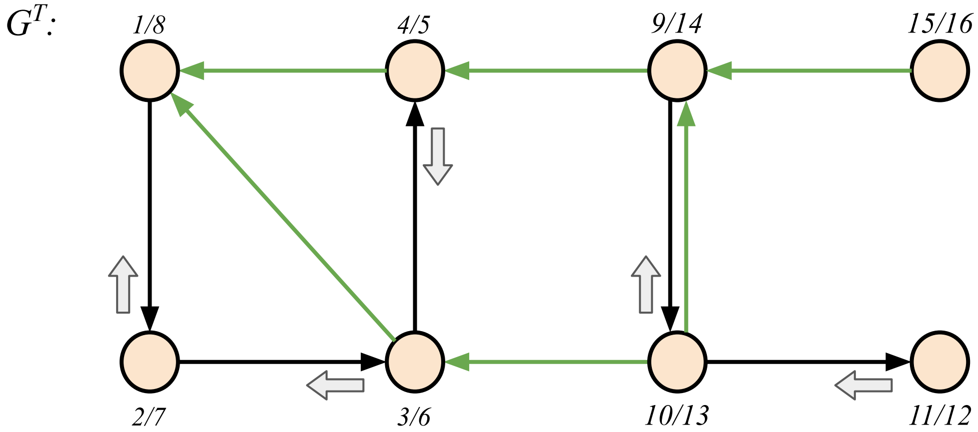
Kosaraju (componentes fuertemente conexas)

```
KOSARAJU ( G ) :
|     computo Gᵀ
|     DFS ( Gᵀ )
|     DFS ( G )  # usando finish de Gᵀ
                 en sentido inverso
```

G^T: *finish* = {b, f, e, a, h, g, c, d}

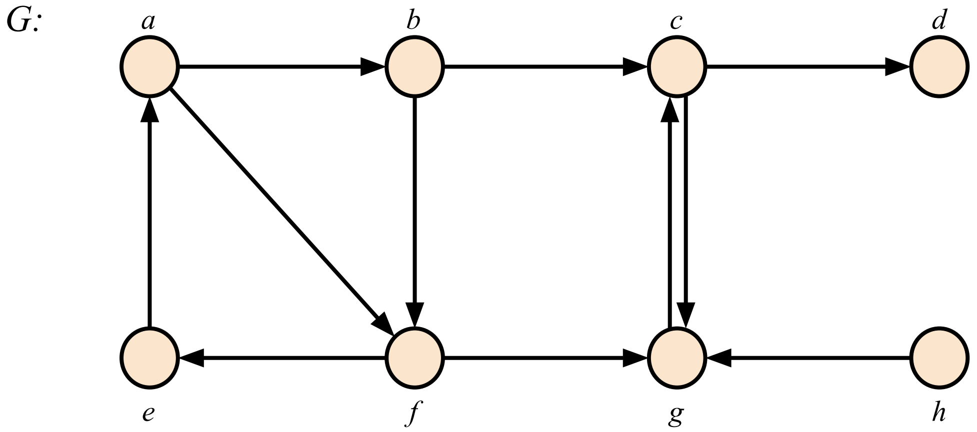
INVERTIR

G^T: *finish* = {d, c, g, h, a, e, f, b}



Kosaraju (componentes fuertemente conexas)

```
KOSARAJU ( G ) :
|     computo Gᵀ
|     DFS ( Gᵀ )
|     DFS ( G )  # usando finish de Gᵀ
                 en sentido inverso
```

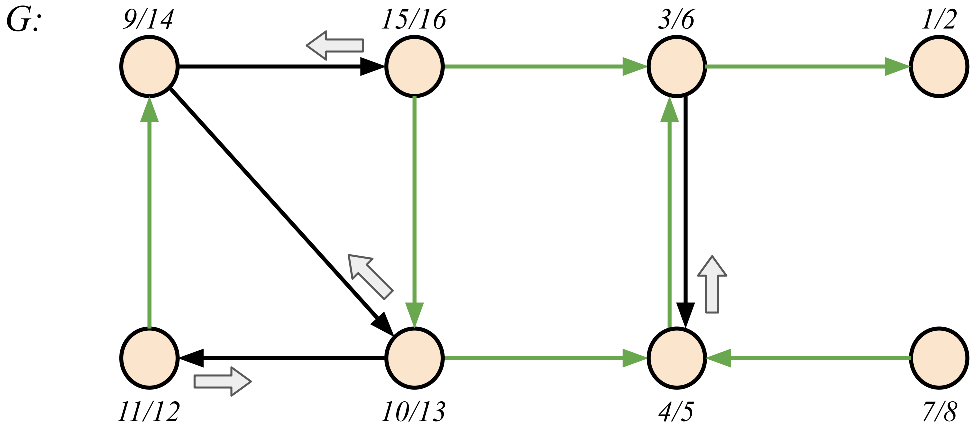


G^T: *finish* = {*d*, *c*, *g*, *h*, *a*, *e*, *f*, *b*}

Kosaraju (componentes fuertemente conexas)

```
KOSARAJU ( G ) :
|     computo Gᵀ
|     DFS ( Gᵀ )
|     DFS ( G )  # usando finish de Gᵀ
                 en sentido inverso
```

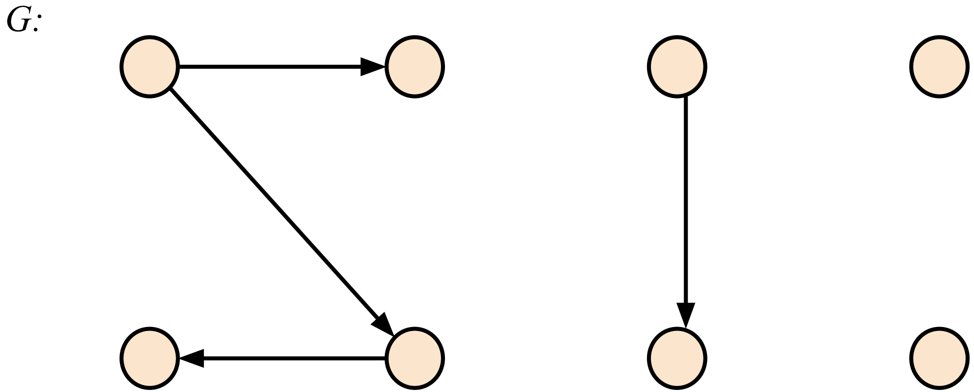
G^T: *finish* = {d, c, g, h, a, e, f, b}

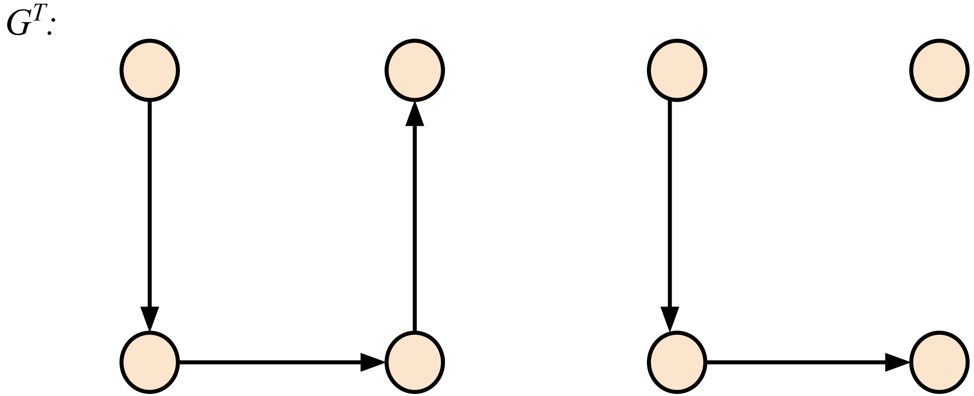


Kosaraju (componentes fuertemente conexas)

```
KOSARAJU ( G ) :
|     computo Gᵀ
|     DFS ( Gᵀ )
|     DFS ( G )  # usando finish de Gᵀ
                 en sentido inverso
```

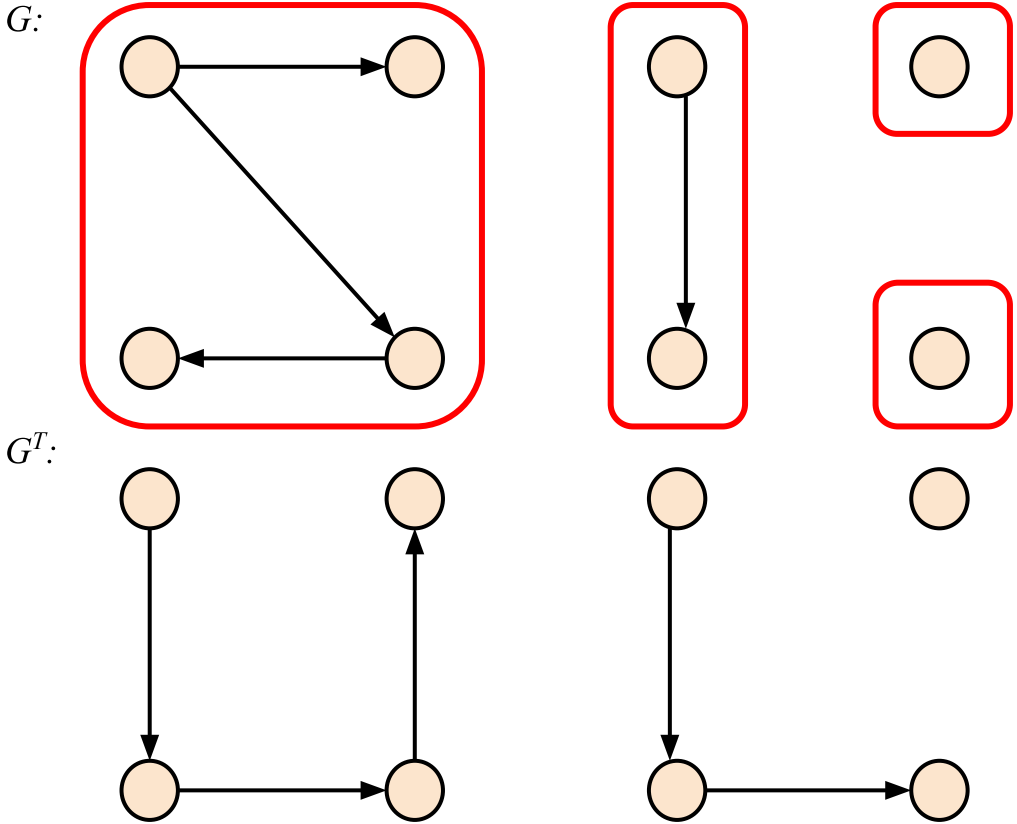
G^T: *finish* = {*d, c, g, h, a, e, f, b*}





Kosaraju (componentes fuertemente conexas)

```
KOSARAJU ( G ) :
|     computo Gᵀ
|     DFS ( Gᵀ )
|     DFS ( G )  # usando finish de Gᵀ
                   en sentido inverso
```



BFS / DFS iterativos

```
RECORRER ( s , Adj ) :
|     i = 0
|     level      = { s : i }
|     parent     = { s : None }   # start / finish para DFS
|     LISTA      = [ s ]          # similar a frontera
|     while LISTA :
|     |     v = elegir un nodo de LISTA       # BFS: Como COLA: tomo el primero
|     |                                       # DFS: Como PILA: tomo el último
|     |     if u in Adj[ v ] and u not in LISTA :
|     |     |     i += 1
|     |     |     level [ u ] = i
|     |     |     parent [ u ] = v
|     |     |     LISTA.append( u )
|     |     else
|     |     |     LISTA = LISTA\{ v }         # BFS: Como COLA: saco el primero
|     |     |                                 # DFS: Como PILA: saco el último
```