AED3 > Clase 5 > Search

Búsqueda / Recorrido en grafos

Objetivos:

- ➔ 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

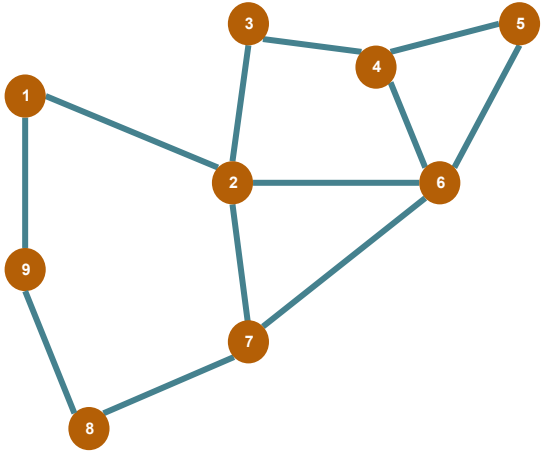
Repaso:

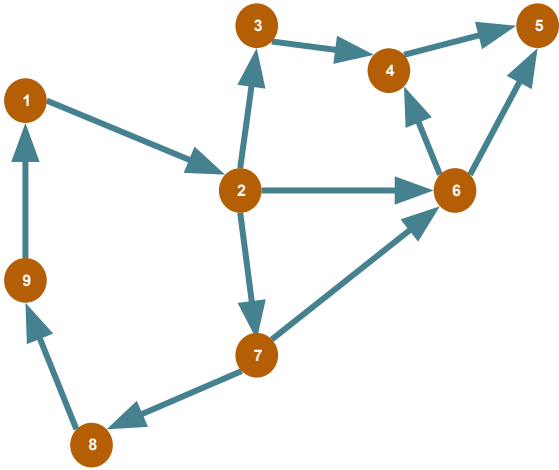
- ➔ G = (V, E)

Búsqueda / Recorrido en grafos

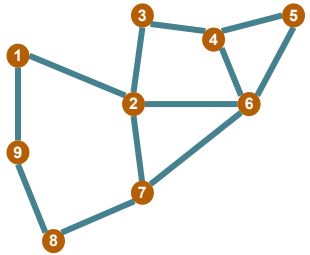
Repaso:

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





Búsqueda / Recorrido en grafos



Repaso:

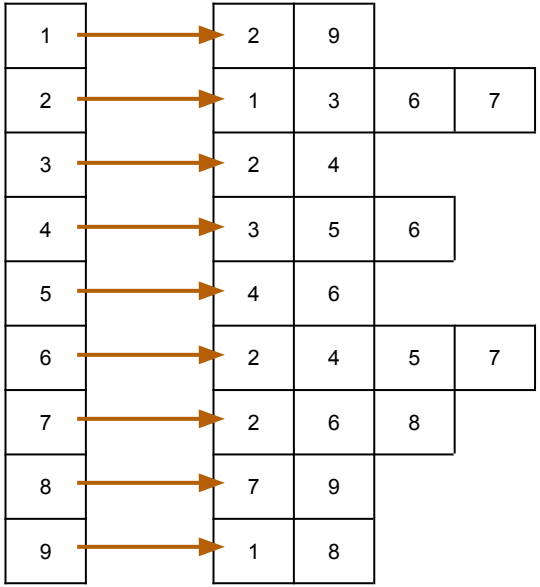
- ➔ G = (V, E)
- ➔ grafos y digrafos (lo que vamos a usar hoy)
- ➔ Listas de adyacencia

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

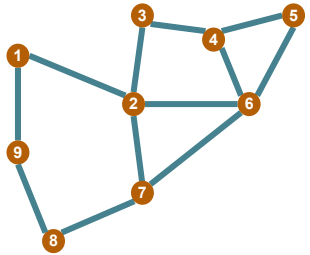
  ⟶ Representación rala (E ~ V)

  ⟶ Recorrerla lleva $\Theta(V+E)$

  ⟶ Es fácil definir muchos grafos con los mismos vértices



Búsqueda / Recorrido en grafos

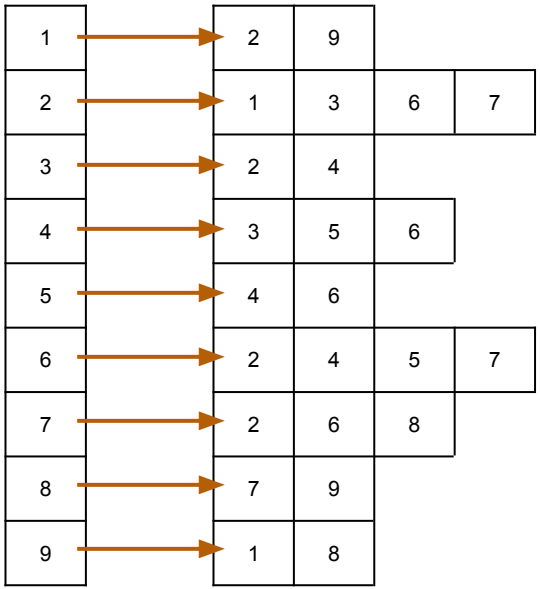


Repaso:

➔ Otras representaciones

⟶ Objeto: ***u.vecinos*** (CLRS)

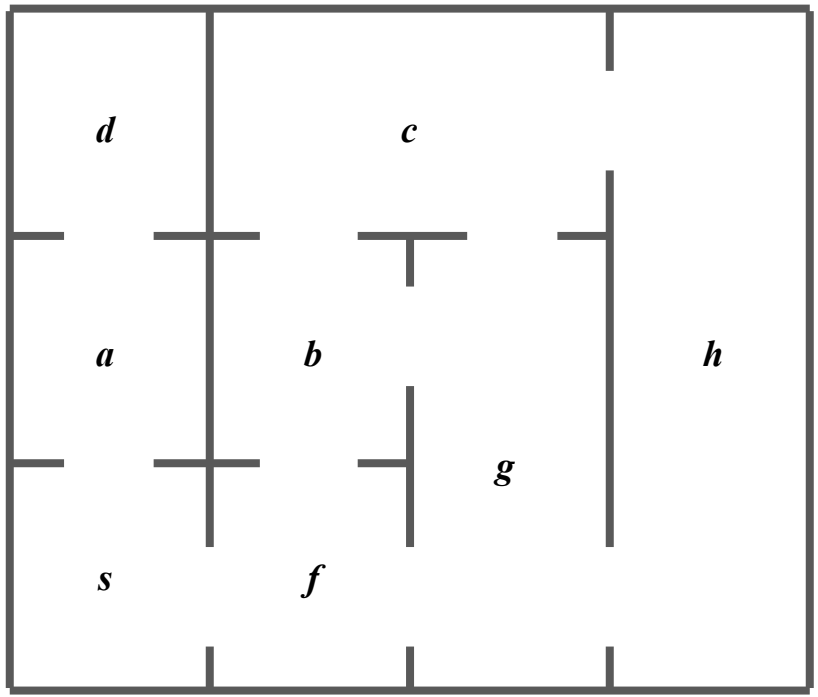
⟶ Representación implícita:

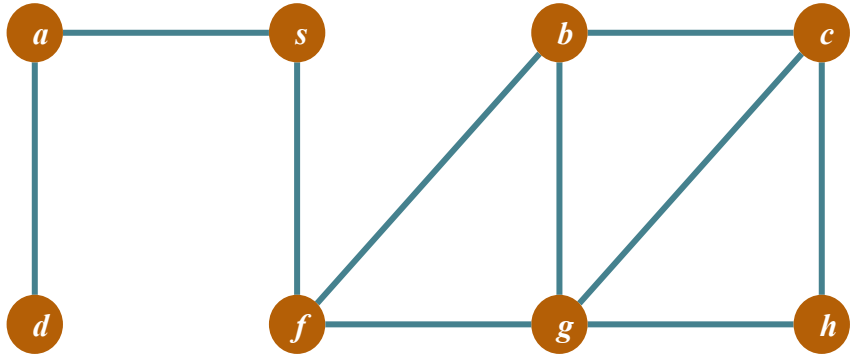


Defino una función Adj(u) o un método u.vecinos.

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!!

Breadth First Search (BFS)





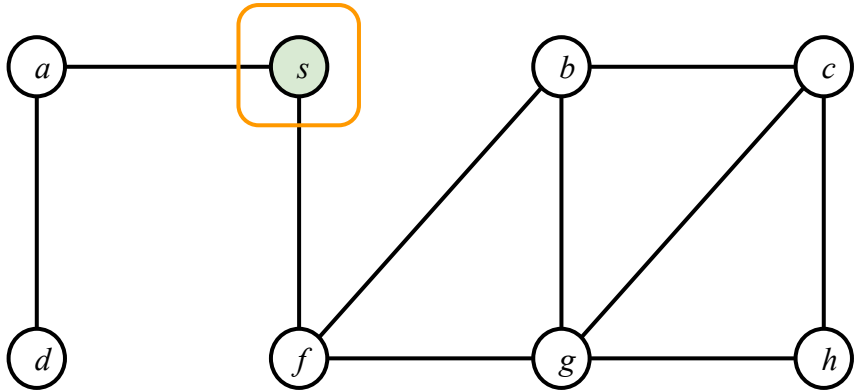
Moore (1959) "*The shortest path through a maze*" pensado para estimar el tráfico en las telecomunicaciones.

Breadth First Search (BFS)

- ➔ Objetivo:
  - 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.
  - $\Theta(V+E)$
- ➔ Idea:
  - 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

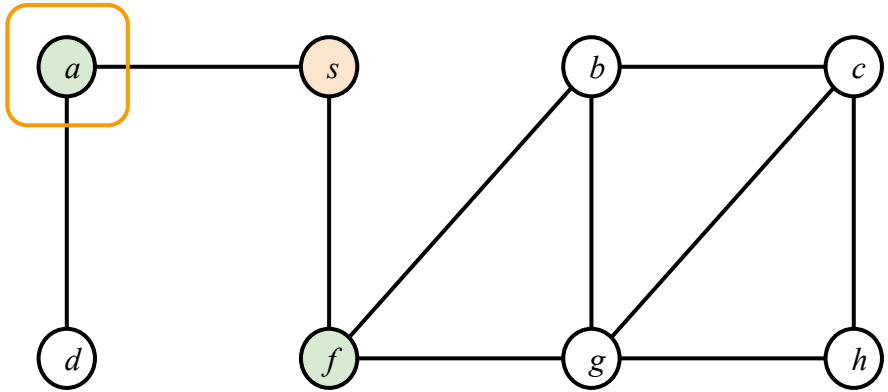
```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |  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

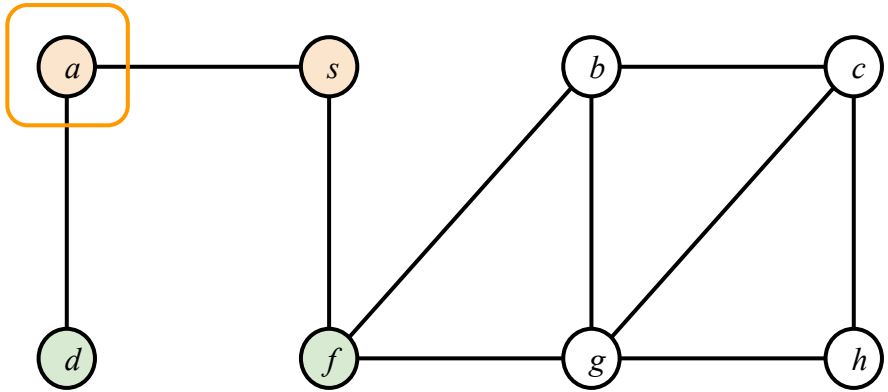
```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |  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

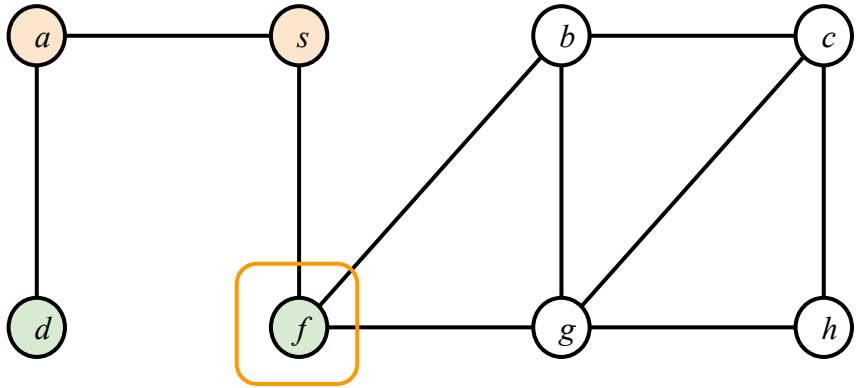
```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |  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

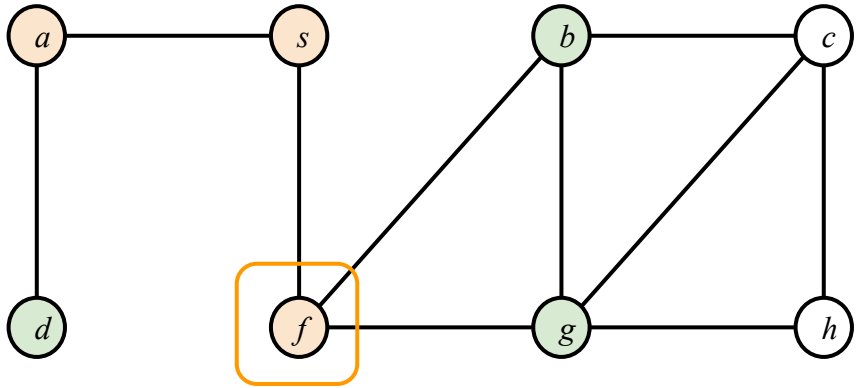
```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |  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

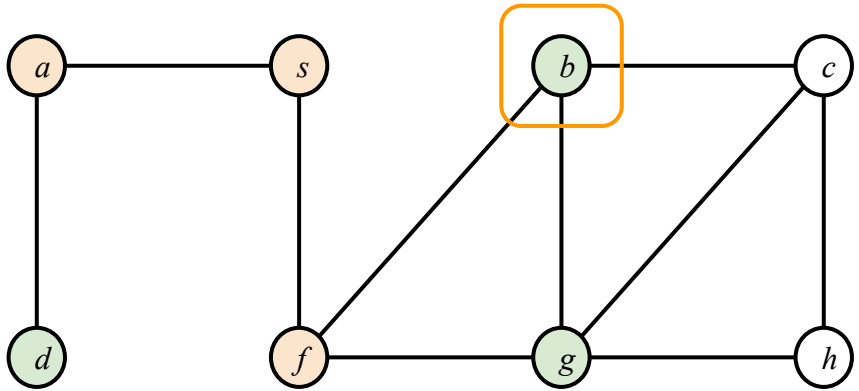
parent = *{s: None, a: s, f: s, d: a, b: f, g: f}*

frontier = *[a, f]*

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

Breadth First Search (BFS) iterativo

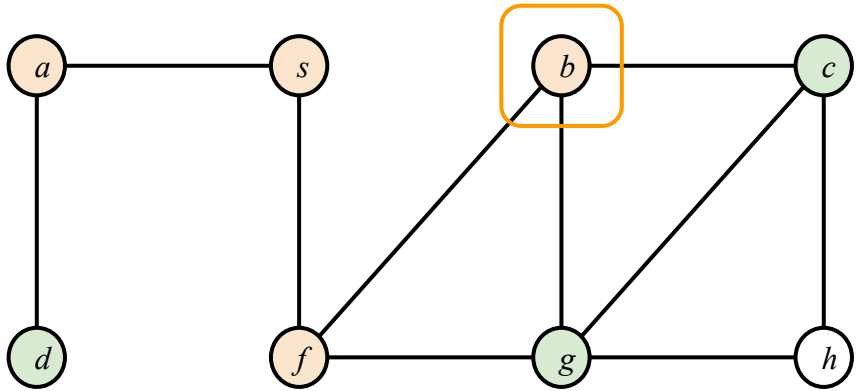
```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |  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

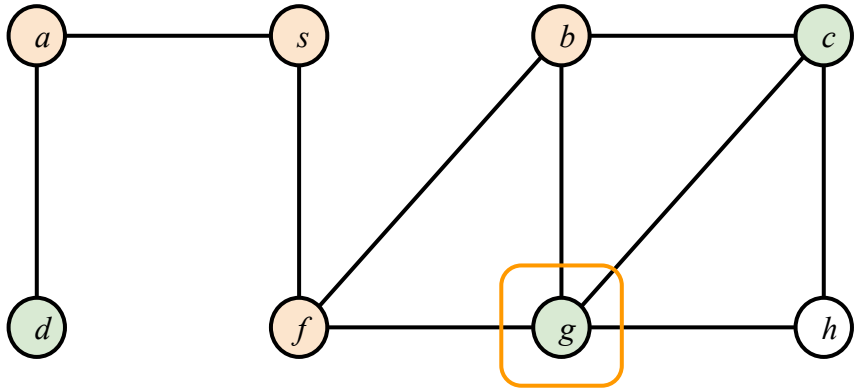
```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |  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

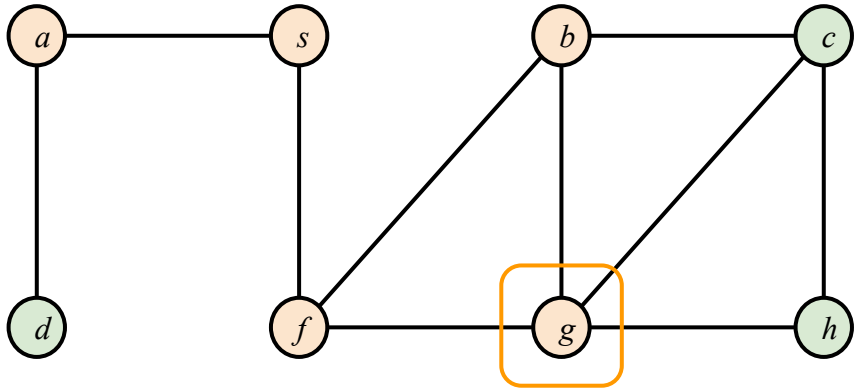
```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |  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

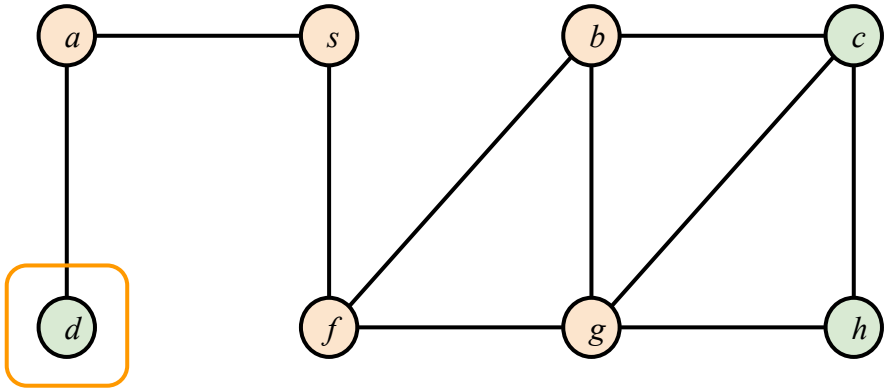
parent = *{s: None, a: s, f: s, d: a, b: f, g: f, c: b, h: g}*

frontier = *[b, g, d]*

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

Breadth First Search (BFS) iterativo

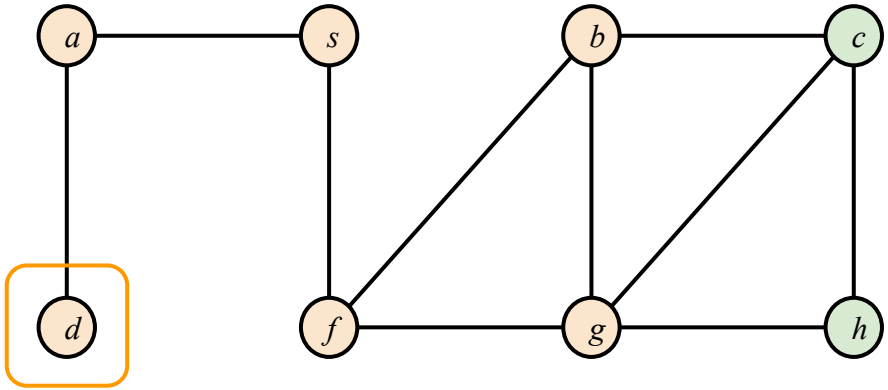
```
BFS ( s , Adj ) :
|     level = { s : 0 }
|     parent = { s : None }
|     i = 1
|     frontier = [ s ] # level i-1
|     while frontier :
|     |     next = [ ] # level i
|     |     for u in frontier :
|     |     |     for v in Adj[ u ] :
|     |     |     |     if v not in level :
|     |     |     |     |     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: 0, a: 1, f: 1, d: 2, b: 2, g: 2, c: 3, h: 3}*

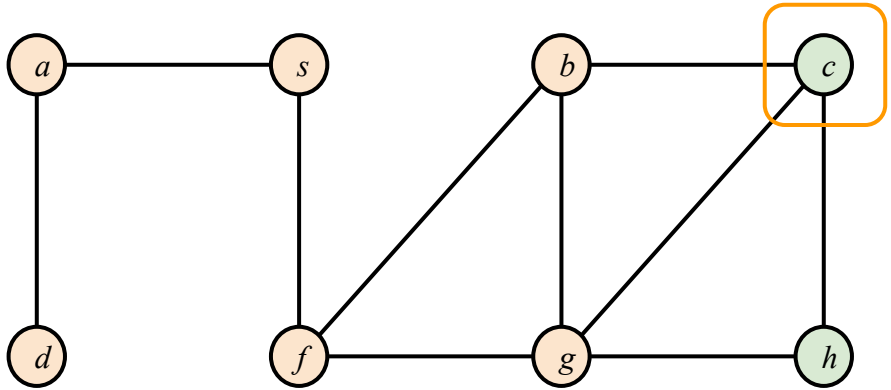
parent = *{s: None, a: s, f: s, d: a, b: f, g: f, c: b, h: g}*

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: 0, a: 1, f: 1, d: 2, b: 2, g: 2, c: 3, h: 3}*

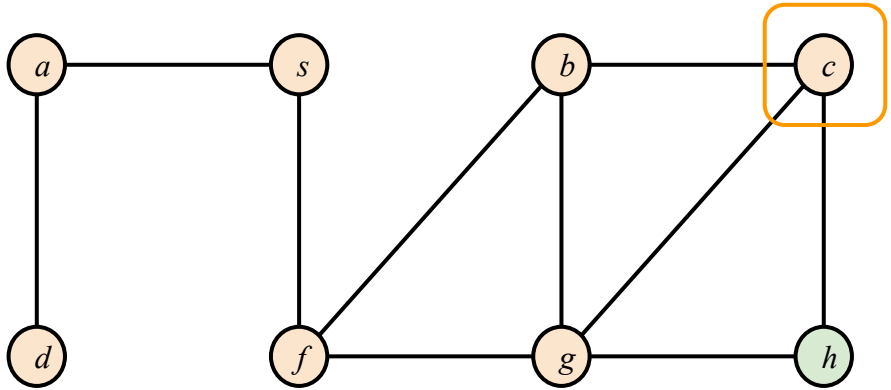
parent = *{s: None, a: s, f: s, d: a, b: f, g: f, c: b, h: g}*

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}*

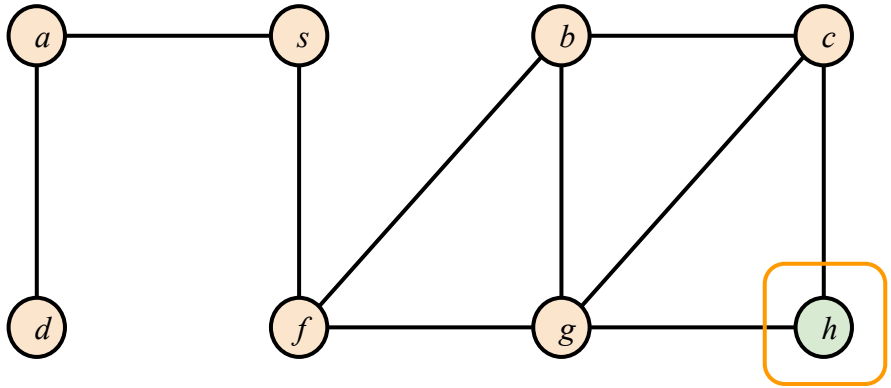
parent = *{s: None, a: s, f: s, d: a, b: f, g: f, c: b, h: g}*

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}*

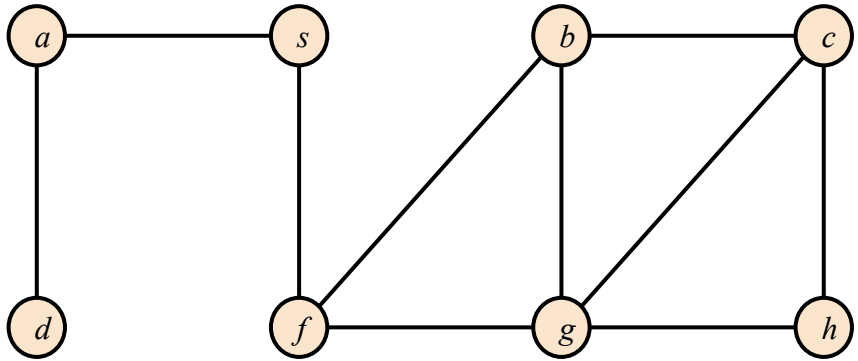
parent = *{s: None, a: s, f: s, d: a, b: f, g: f, c: b, h: g}*

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}*

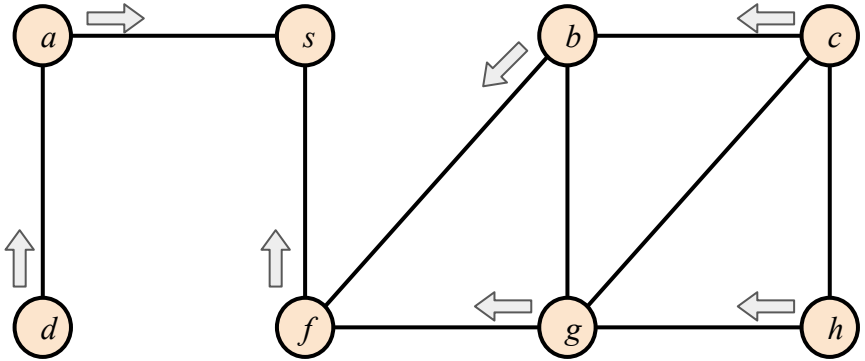
parent = *{s: None, a: s, f: s, d: a, b: f, g: f, c: b, h: g}*

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}*

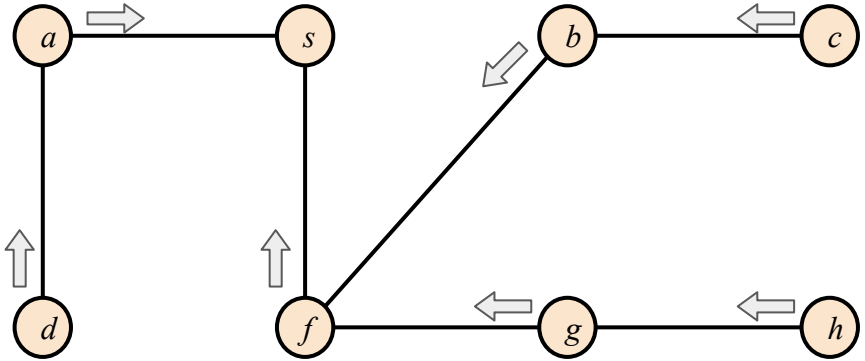
parent = ***{s: None, a: s, f: s, d: a, b: f, g: f, c: b, h: g}***

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}*

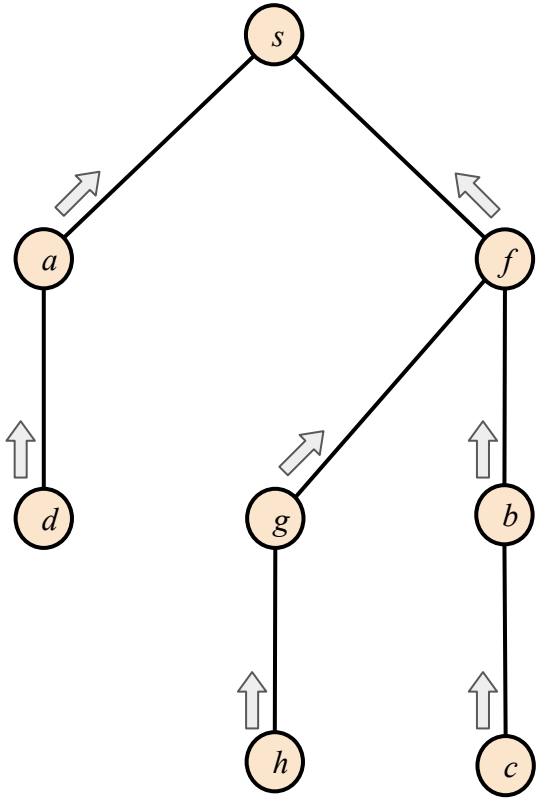
parent = ***{s: None, a: s, f: s, d: a, b: f, g: f, c: b, h: g}***

frontier = *[c, h]*

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

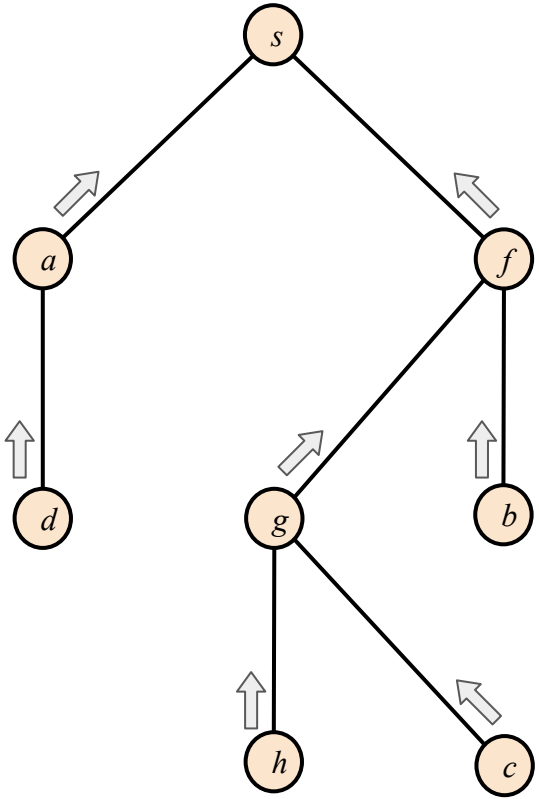
BFS-tree

$G\pi = (V\pi, E\pi)$
$V\pi = \{v \in V : parent[\, v \,] \neq None\} \cup \{s\}$
$E\pi = \{(parent[\, v \,], v): v \in V\pi - \{s\}\}$



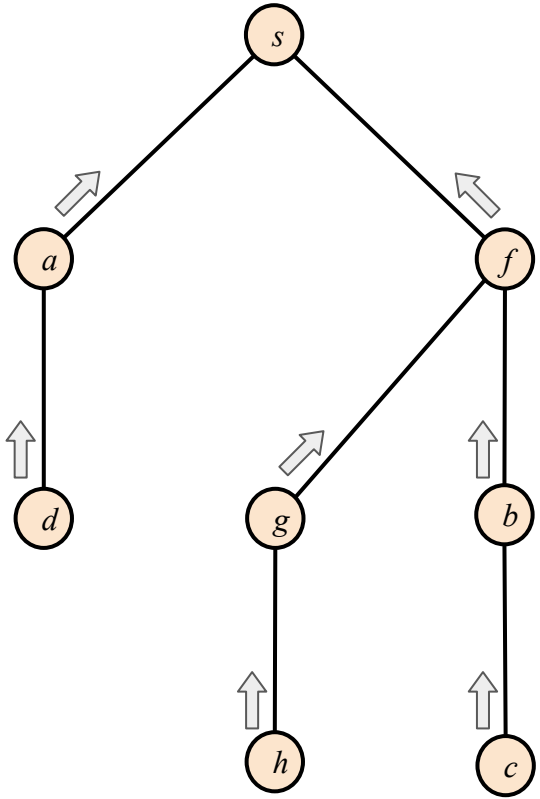
level = {s: 0, a: 1, f: 1, d: 2, b: 2, g: 2, c: 3, h: 3}
***parent* = {s: None, a: s, f: s, d: a, b: f, g: f, c: b, h: g}**
frontier = [c, h]
Adj[h] = { c, g }

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
$\Rightarrow$

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

$\Rightarrow O(E)$

$\Rightarrow O(V+E)$

Breadth First Search (BFS) iterativo (versión CLRS)

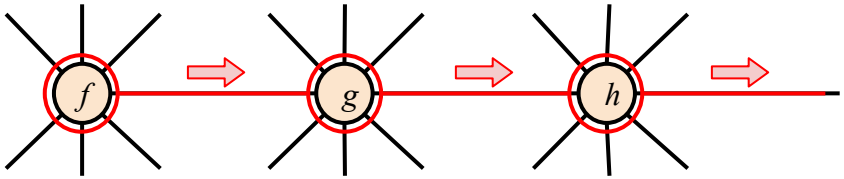
```
BFS ( G , s ) :
|     for cada nodo u ∈ G.V - { s }
|     |     u.color    = n        # w: nuevo, g: frontera descubierta, k: usado
|     |     u.d        = ∞        # distancia
|     |     u.π        = NIL      # parent / predecesor
|     s.color    = g
|     s.d        = 0
|     s.π        = NIL
|     Q          = Ø        # 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,s)    # guardo v para explorar después
|     |     u.color = k # termino de explorar y lo marco
```

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 )
```

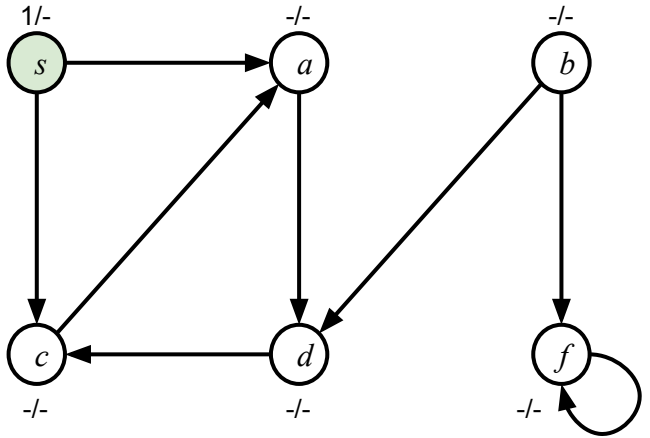
Deep First Search (DFS) recursivo

```
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 )
```

➔ *¡CUIDADO! Es importante guardar registro para no volver a explorar nodos ya visitados.*

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 )
```



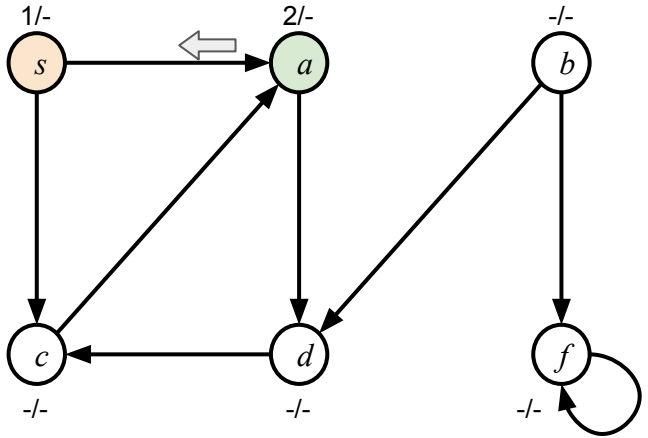
start = {}
finish = {}
parent = {}
k = 0

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 )
```



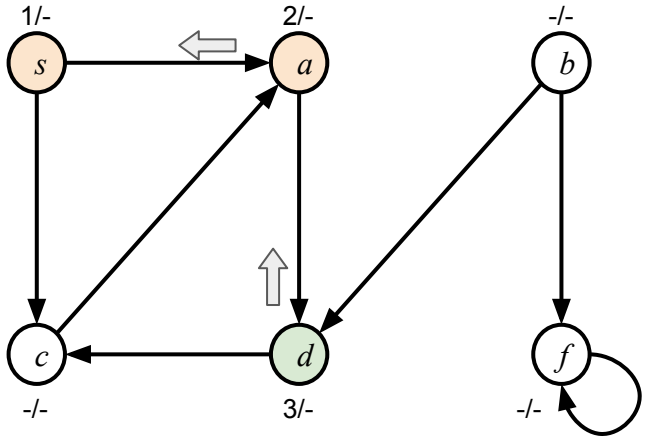
start = {s:1}
finish = {}
parent = {s:None}
k = 1

parent [a] = s
DFS-visit(Adj, a)

k = 2
start [a] = 2
Adj[a] = {d}
parent[d] = 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 )
```



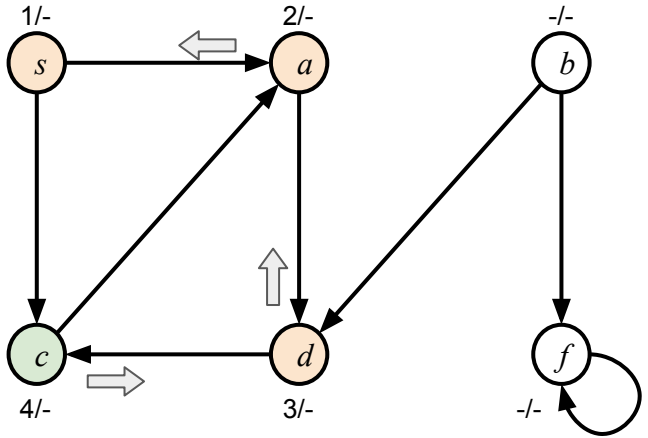
start = {s:1, a:2}
finish = {}
parent = {s:None, a:s}
k = 2

parent [d] = a
DFS-visit(Adj, d)

k = 3
start [d] = 3
Adj[d] = {c}
parent[c] = 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 )
```



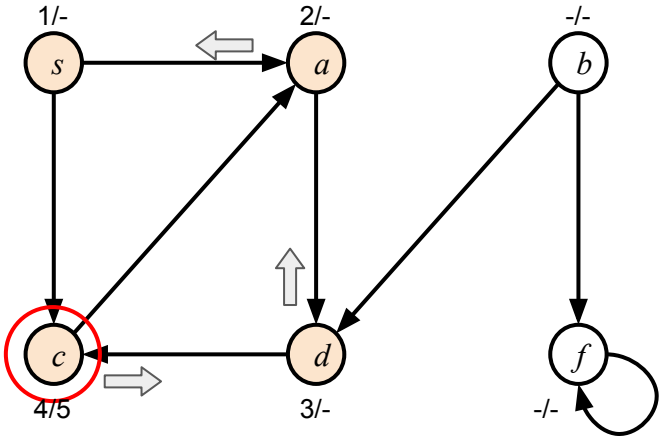
start = {s:1, a:2, d:3}
finish = {}
parent = {s:None, a:s, d:a}
k = 3

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 )
```



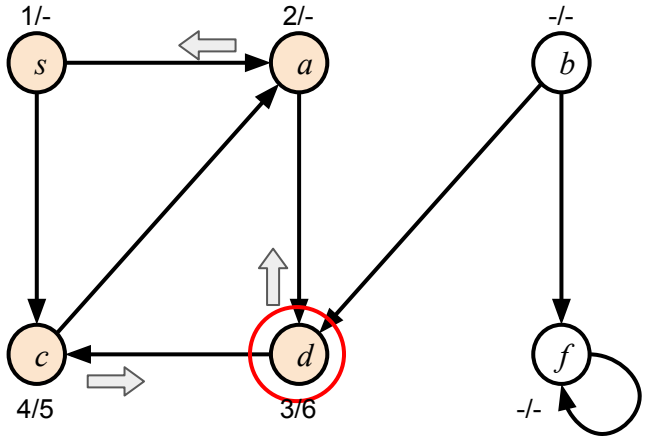
start = {s:1, a:2, d:3, c:4}
finish = {c:5}
parent = {s:None, a:s, d:a, c:d}
k = 5

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, d:6}

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

k = 5

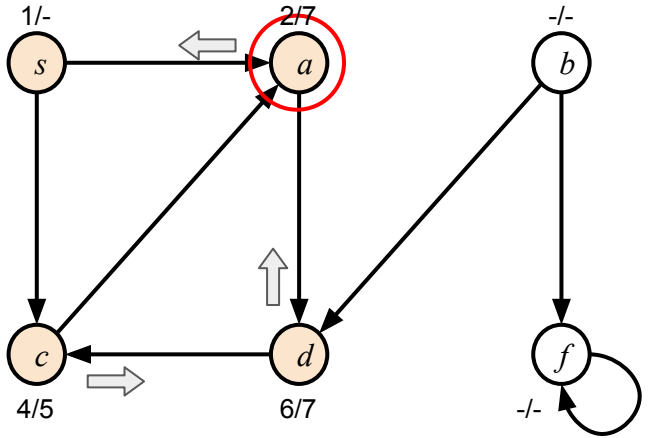
k += 1

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}

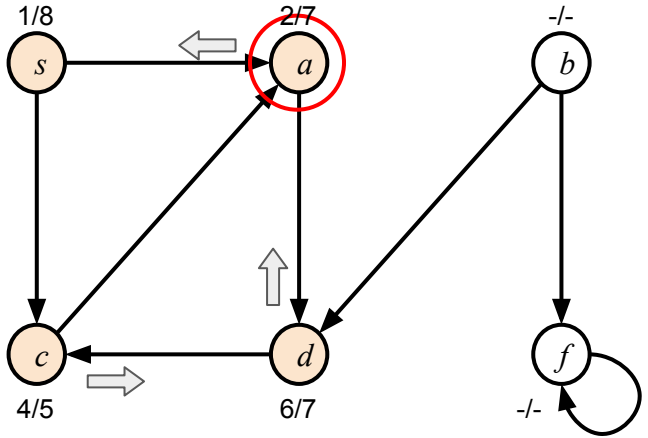
k = 6

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}

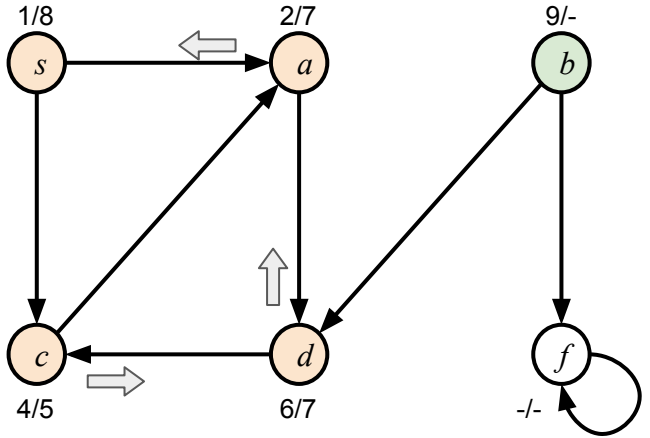
k = 7

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 )
```

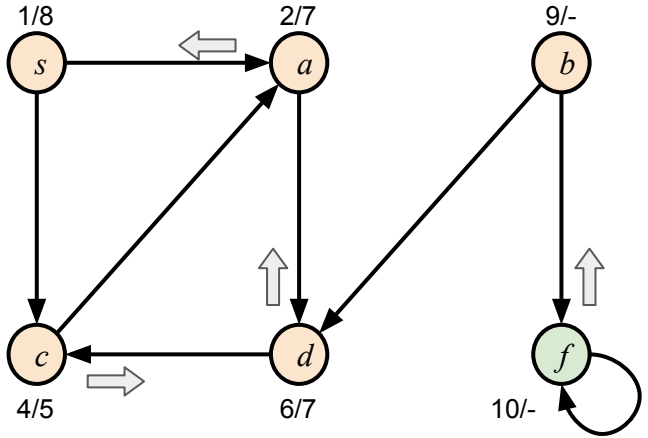


start = {s:1, a:2, d:3, c:4}
finish = {c:5, d:6, a:7, s:8}
parent = {s:None, a:s, d:a, c:d}
k = 8

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

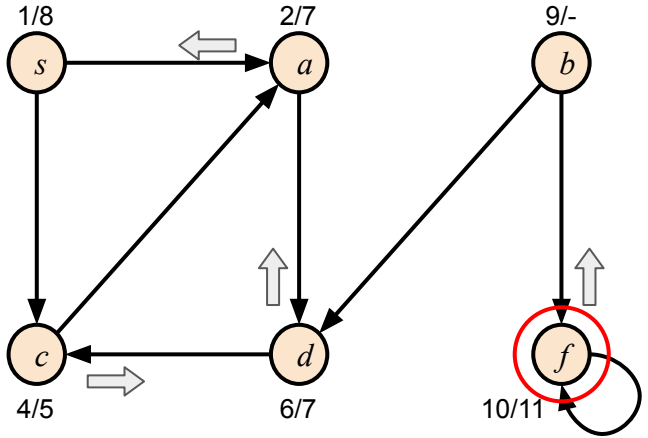
start [b] = 10

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

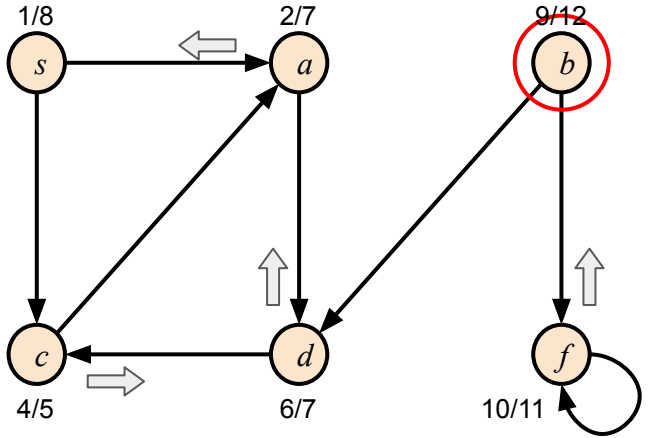
start [b] = 10

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 )
```

$\Rightarrow O(V+E)$

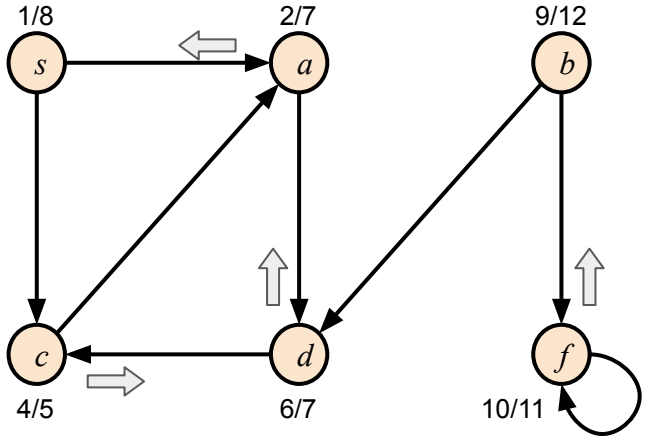
Cada vértice entra a la lista sólo una vez (y se explora sólo una vez)
$\Rightarrow O(V)$

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

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

$\Rightarrow O(E)$

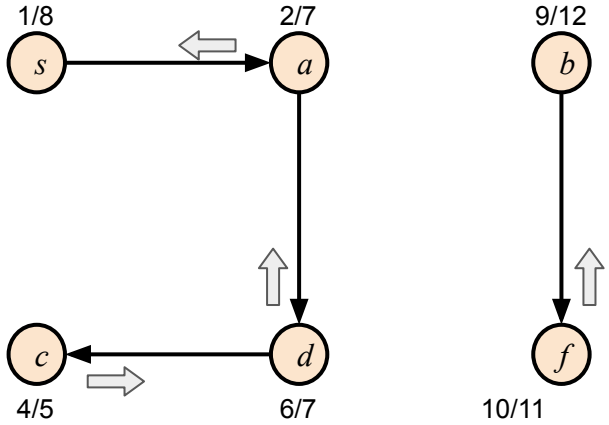
Deep First Search (DFS) recursivo



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**

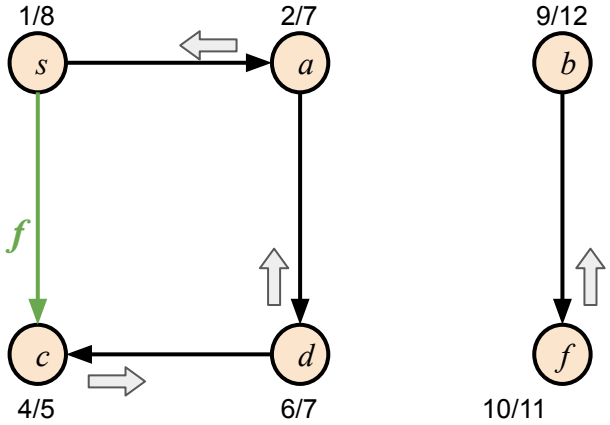


start = {s:1, a:2, d:3, c:4, b:9, f:10}
finish = {c:5, d:6, a:7, s:8, f:11, b:12}
parent = {s:None, a:s, d:a, c:d, b:None, f:b}

DFS: Clasificación de aristas

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

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



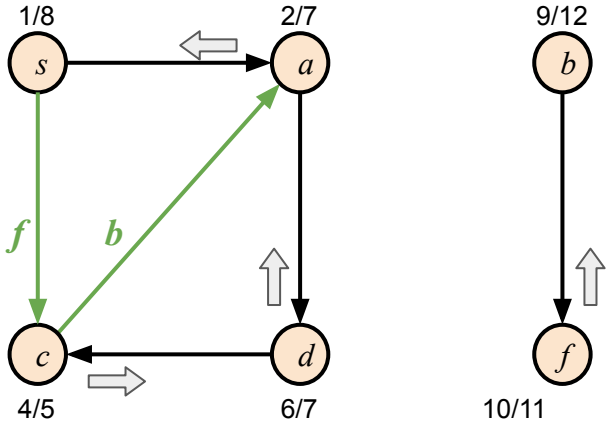
start = {s:1, a:2, d:3, c:4, b:9, f:10}
finish = {c:5, d:6, a:7, s:8, f:11, b:12}
parent = {s:None, a:s, d:a, c:d, b:None, f:b}

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}

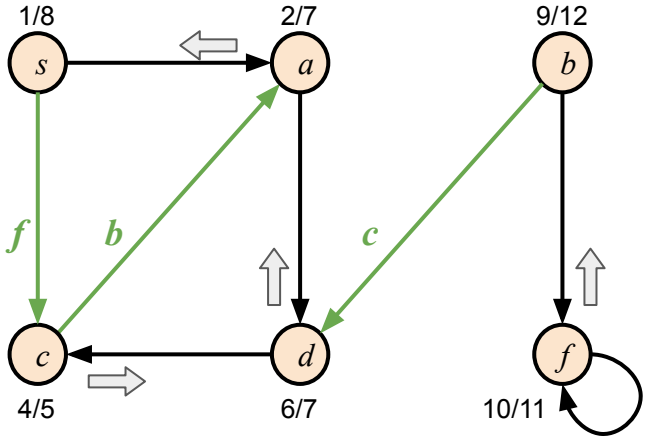
DFS: Clasificación de aristas

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}

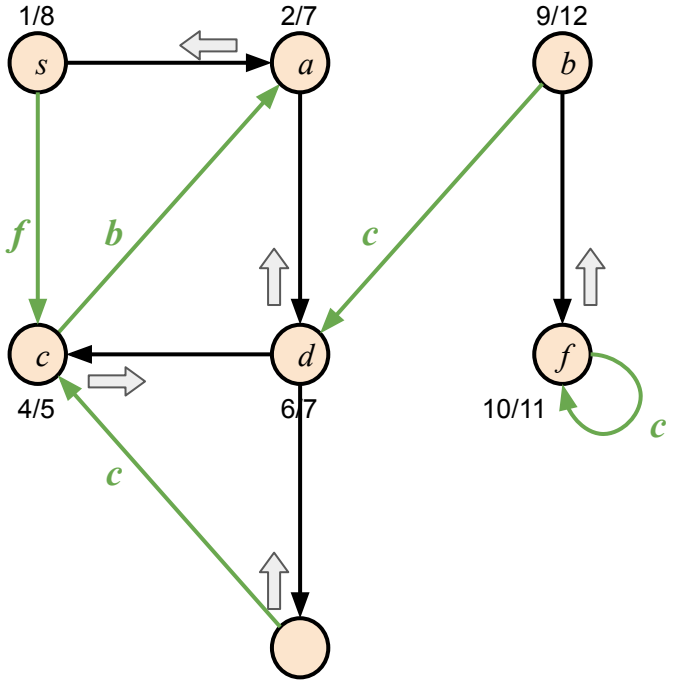
DFS: Clasificación de aristas

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

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

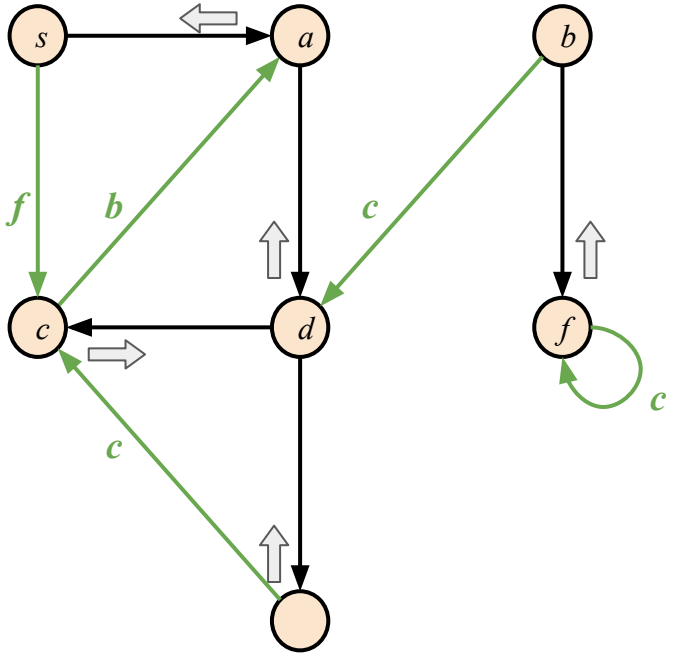
Backward edges (aristas) (***b***): Van hacia un ancestro (predecesor).

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



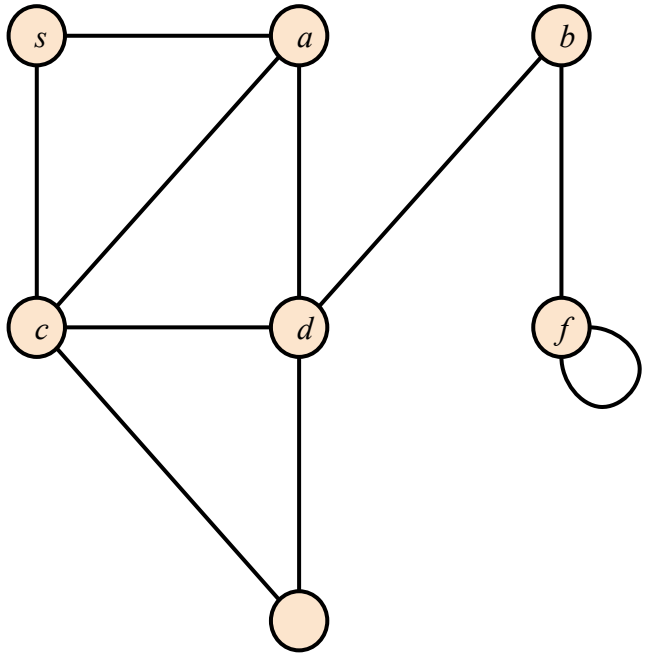
DFS: Clasificación de aristas

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



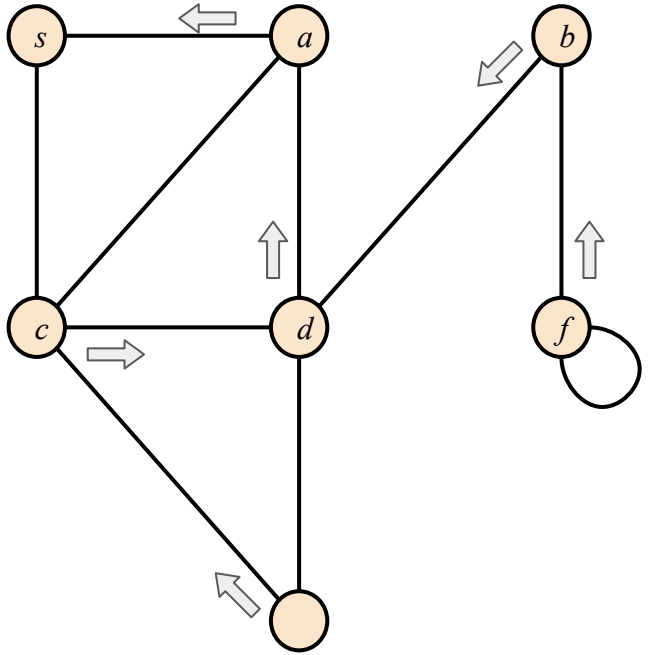
DFS: Clasificación de aristas

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



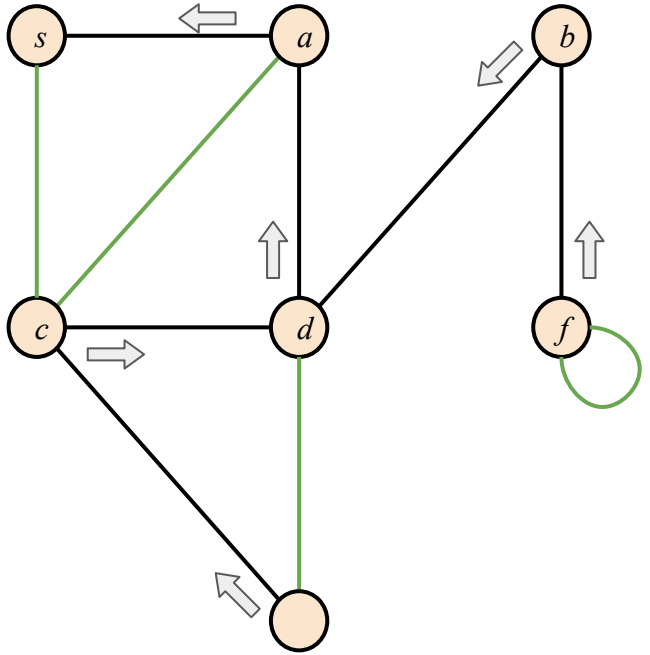
DFS: Clasificación de aristas

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



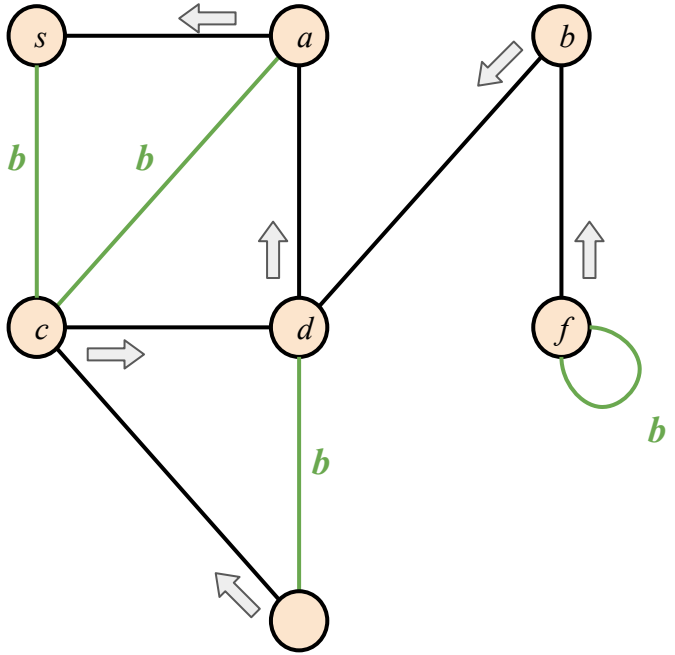
DFS: Clasificación de aristas

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

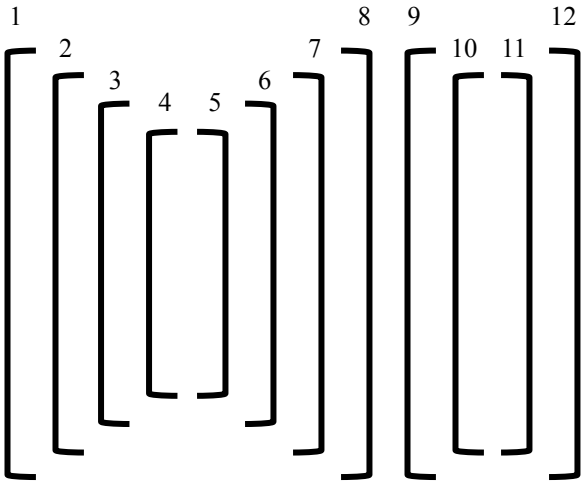


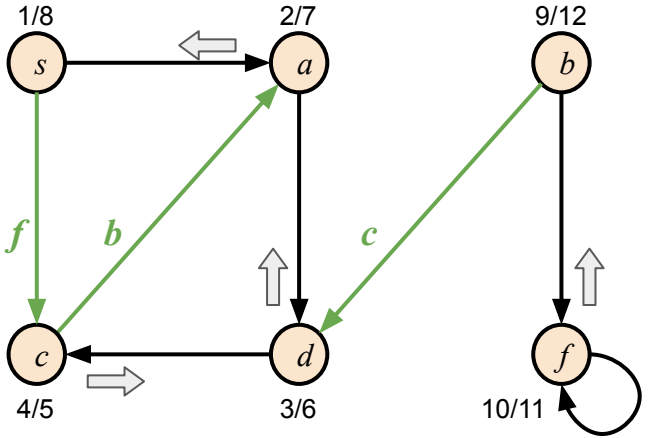
DFS: Clasificación de aristas

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



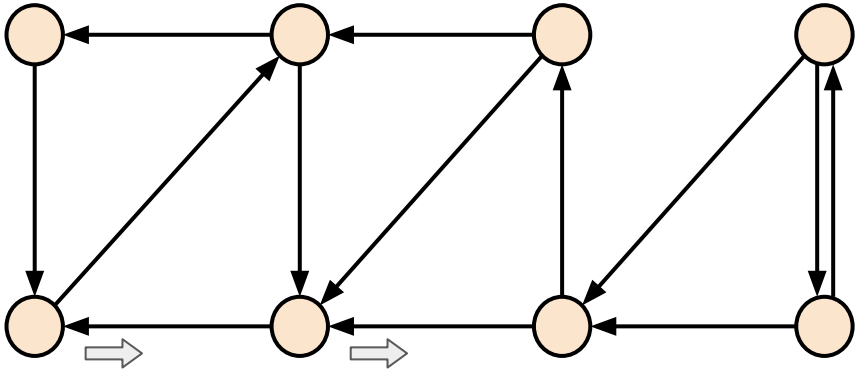
Otra manera de verlo…



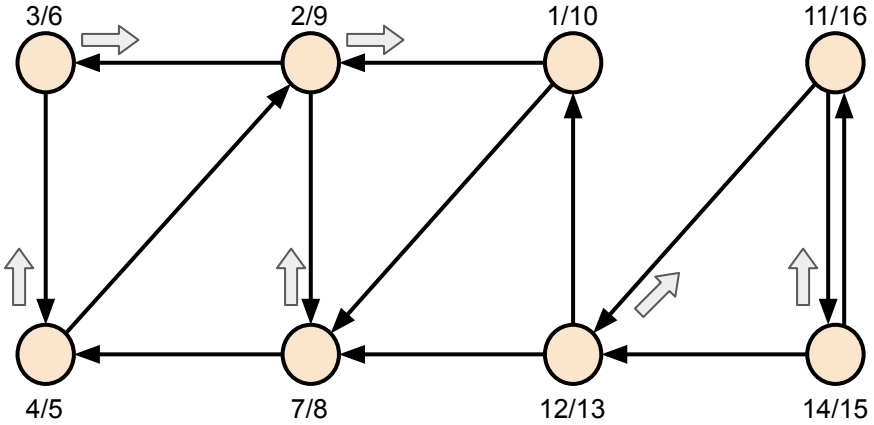


start = {s:1, a:2, d:3, c:4, b:9, f:10}
finish = {c:5, d:6, a:7, s:8, f:11, b:12}
parent = {s:None, a:s, d:a, c:d, b:None, f:b}

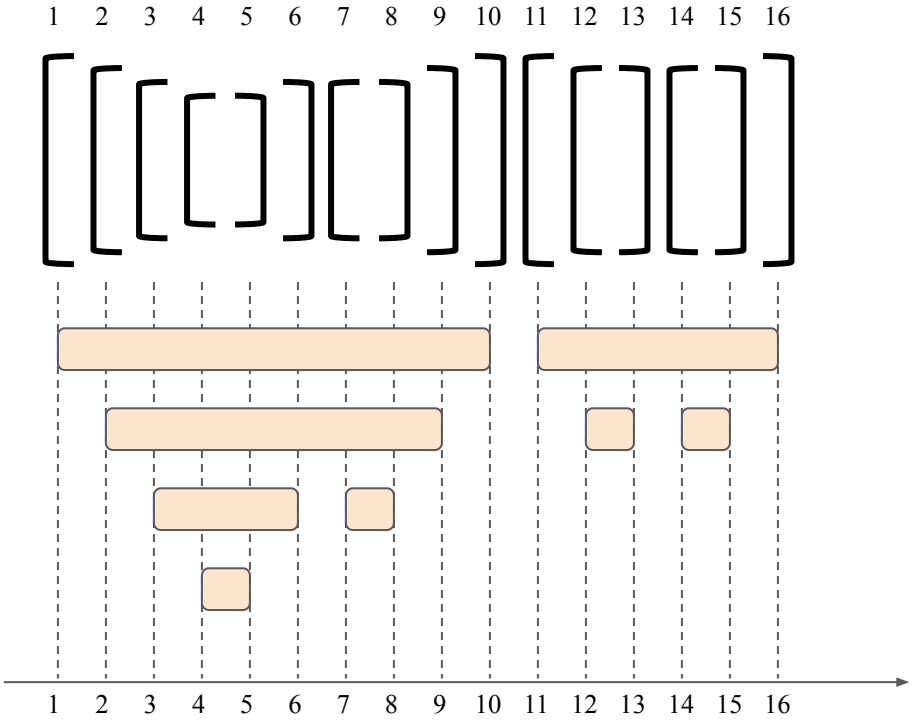
Otra manera de verlo…

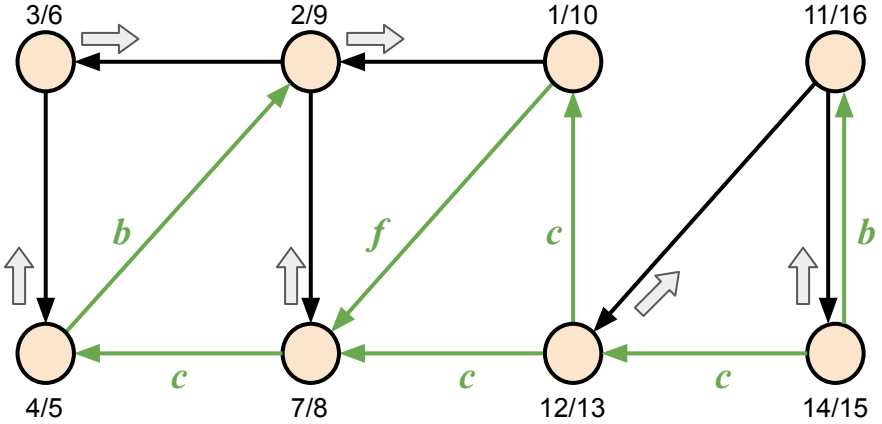


Otra manera de verlo…



Otra manera de verlo…





Detección de ciclos (con DFS)

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

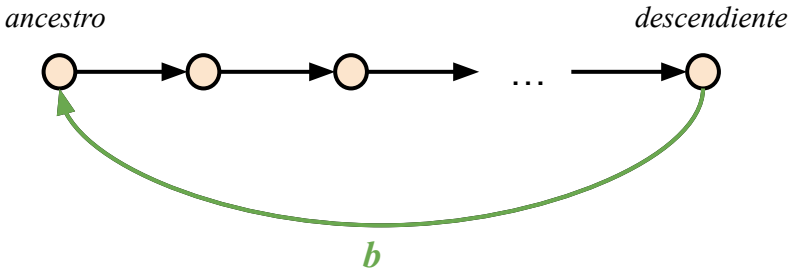
G tiene un ciclo $\Leftrightarrow$ el bosque DFS tiene una arista backward

Detección de ciclos (con DFS)

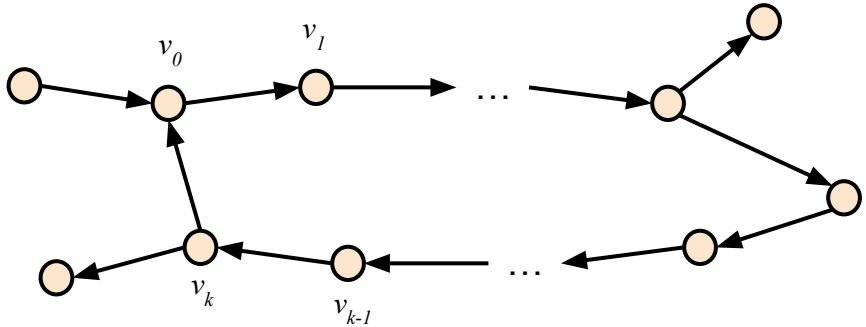
Teorema 1: Dado un digrafo G = (V, E),

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

Demostración (⇐):



Demostración (⇒):



Lo mismo vale para un no dirigido.

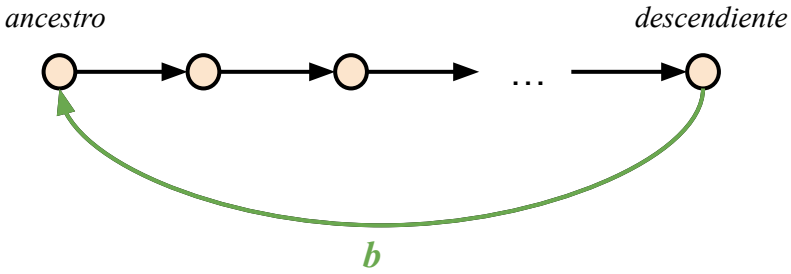
❑

Detección de ciclos (con DFS)

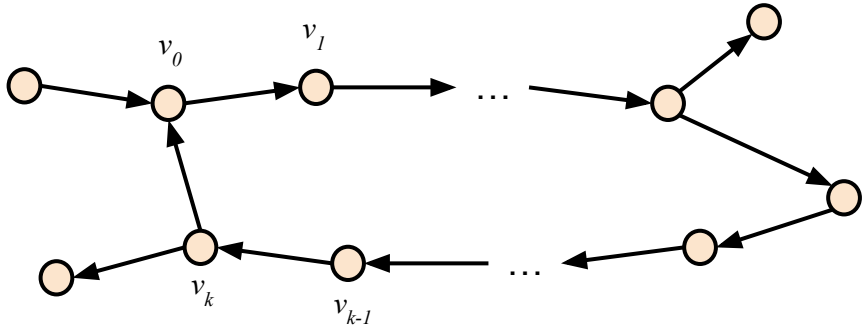
Teorema 1: Dado un digrafo G = (V, E),

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

Demostración (⇐):



Demostración (⇒):



Lo mismo vale para un no dirigido.

❑

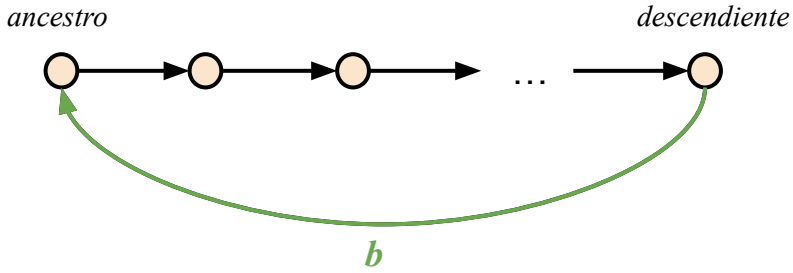
Detección de ciclos (con DFS)

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

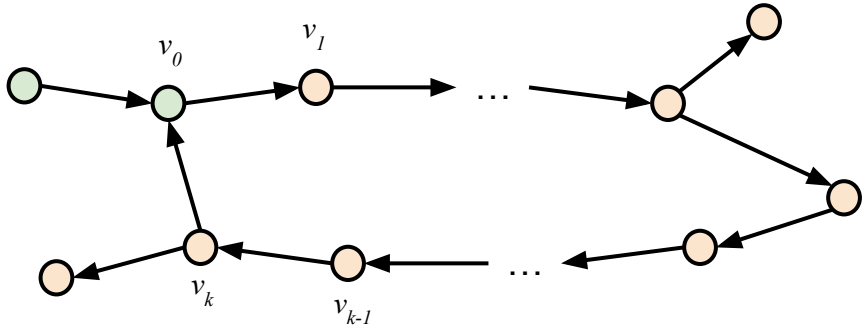
Teorema 1: Dado un digrafo G = (V, E),

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

Demostración (⇐):



Demostración (⇒):



Lo mismo vale para un no dirigido.

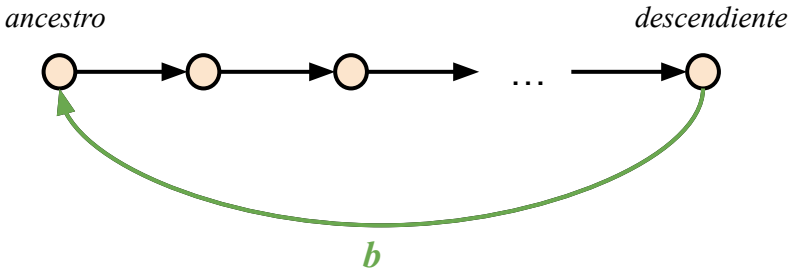
❑

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 (⇐):

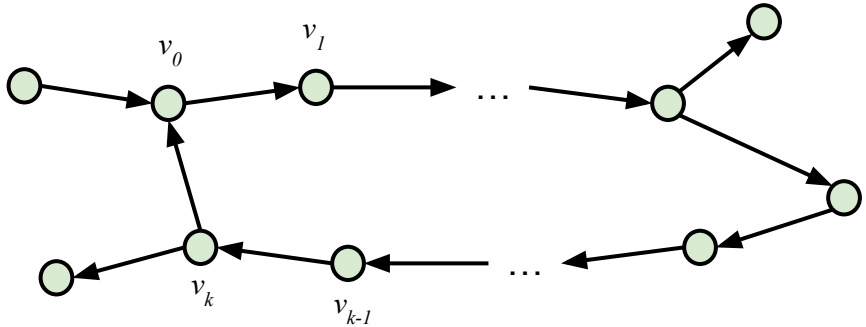


Lo mismo vale para un no dirigido.

❑

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

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***.

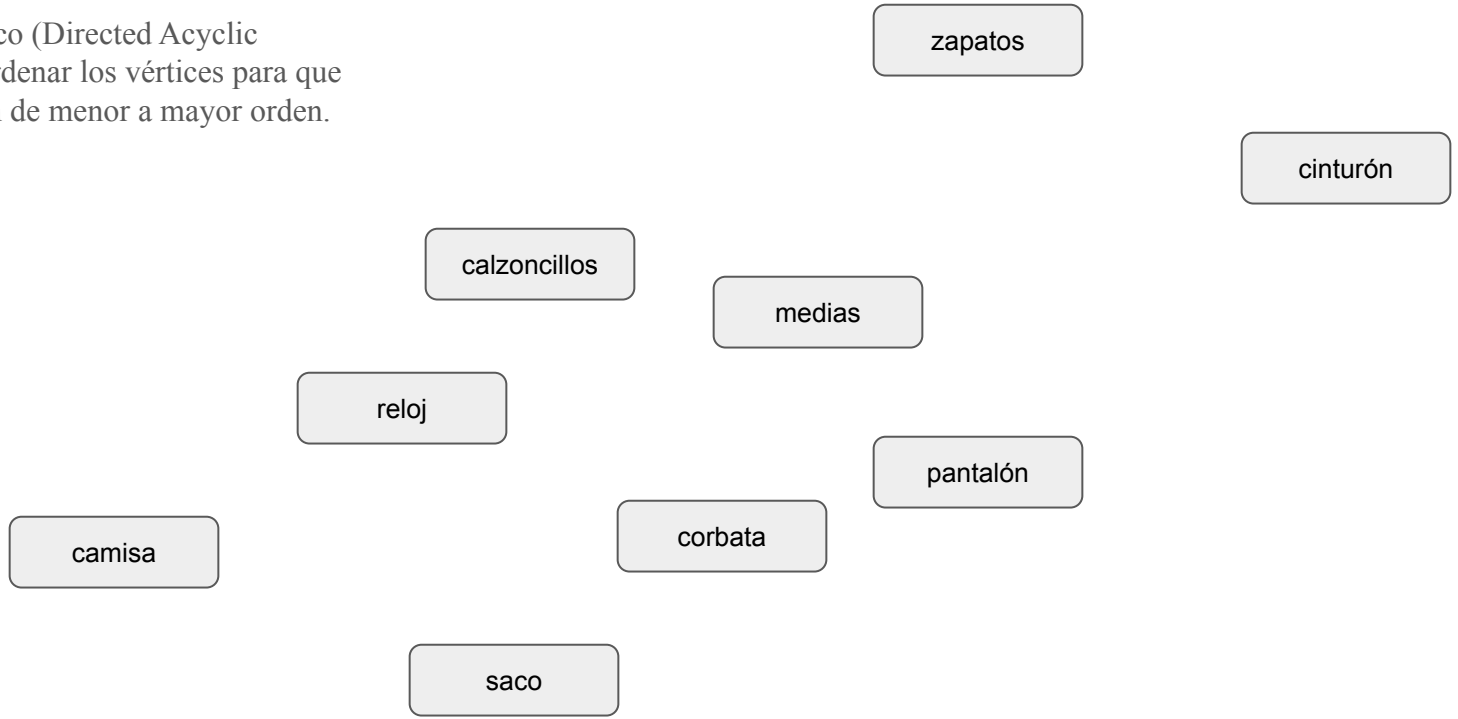
❑

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.

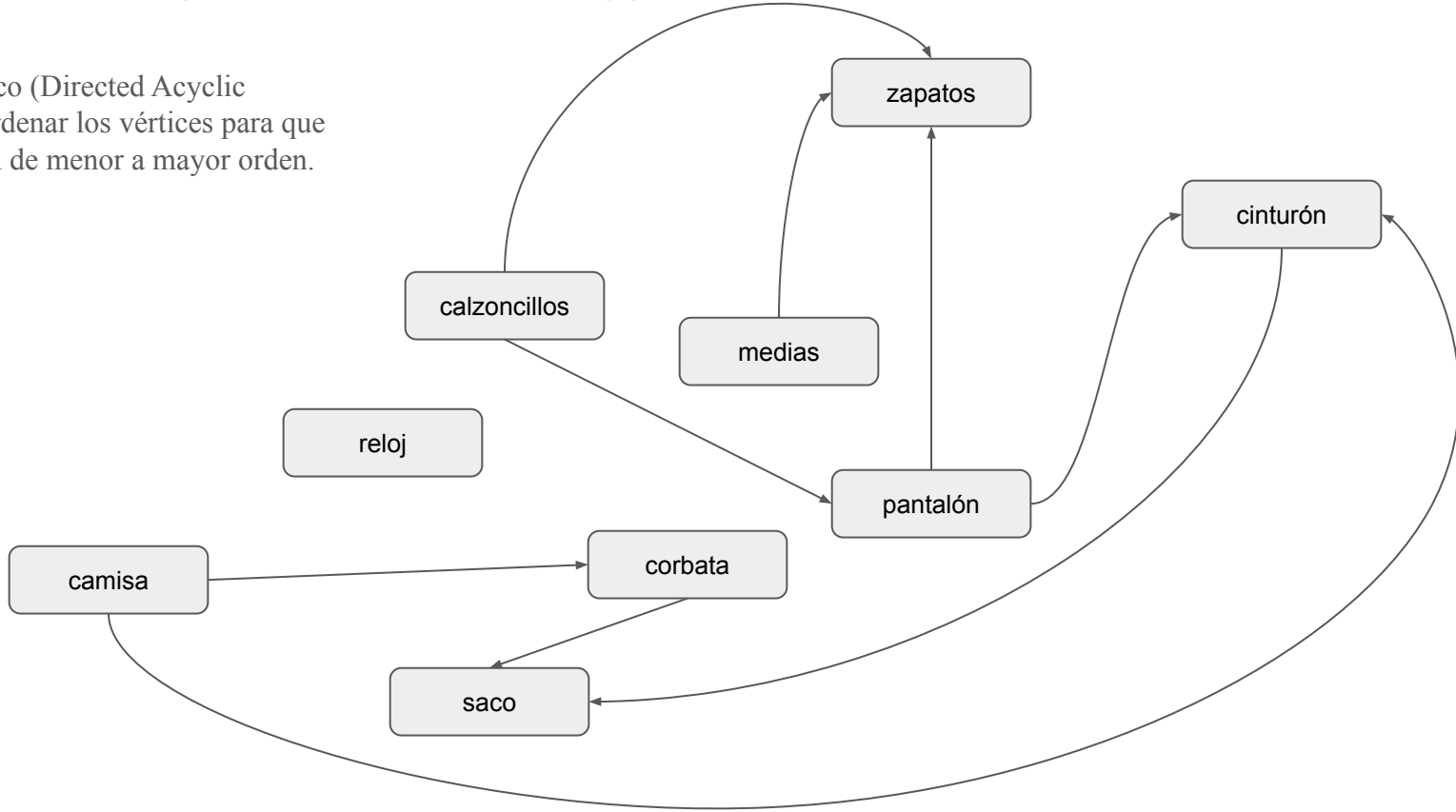
Topological sort (Job Scheduling)

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



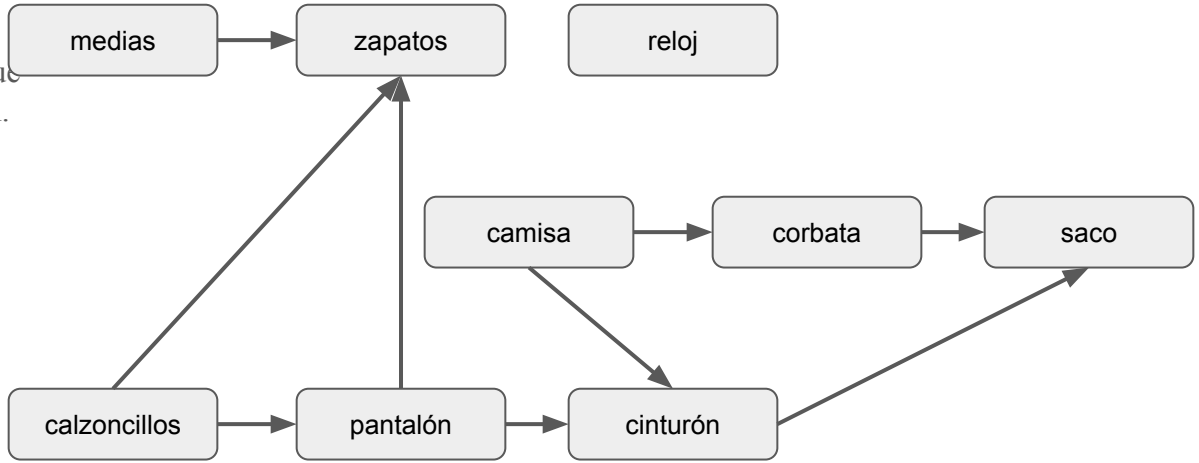
Topological sort (Job Scheduling)

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



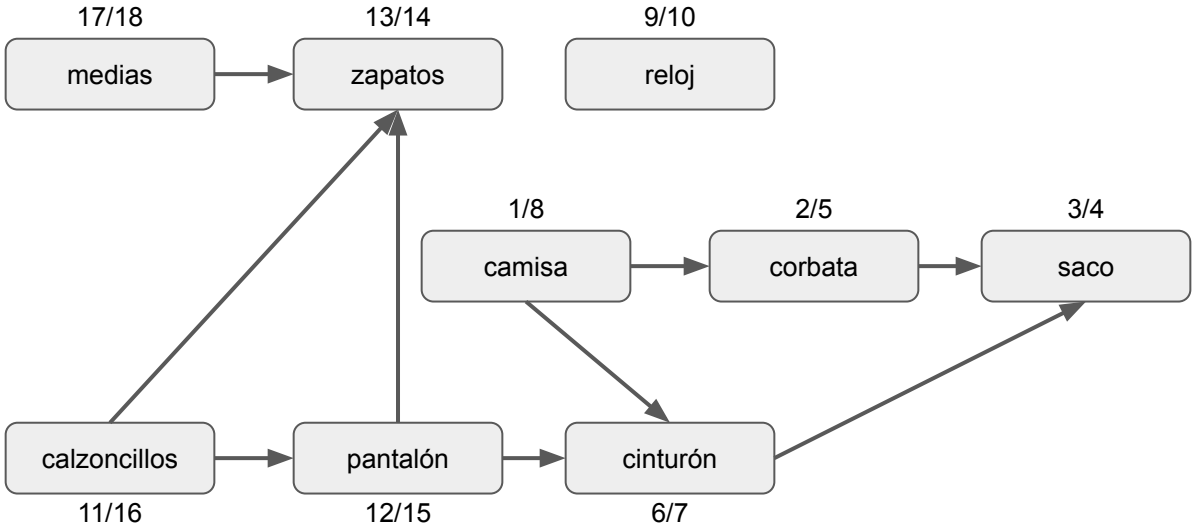
Topological sort (Job Scheduling)

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



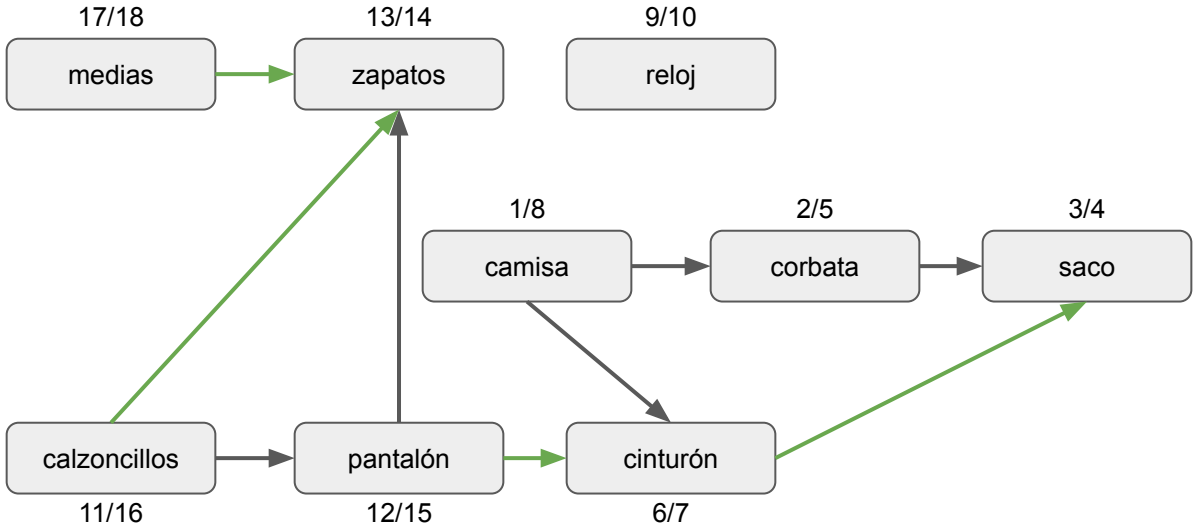
Topological sort (Job Scheduling)

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



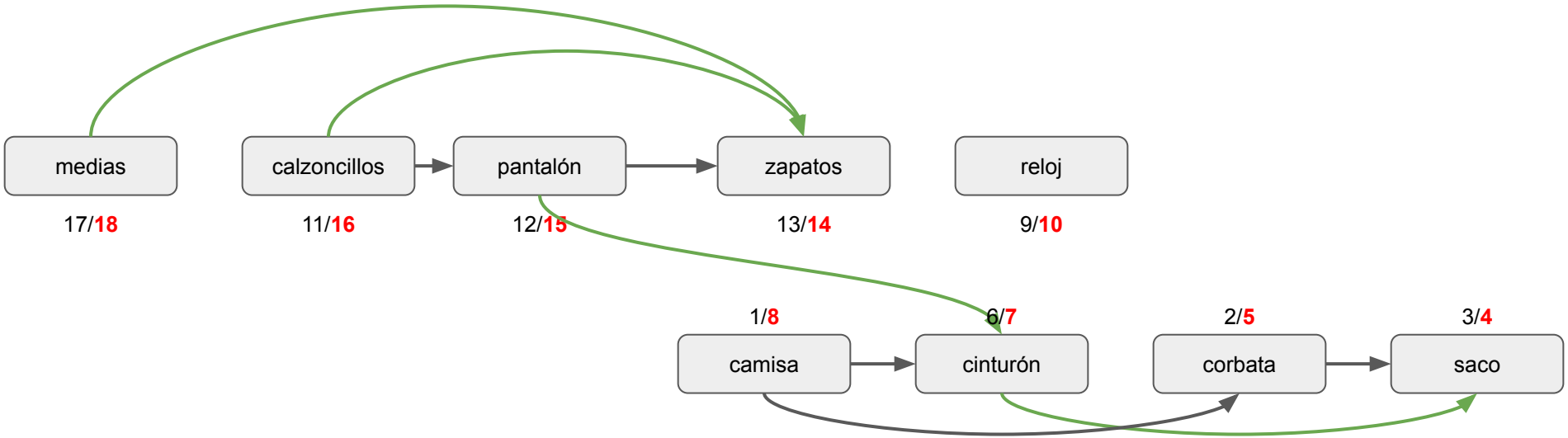
Topological sort (Job Scheduling)

```
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
```

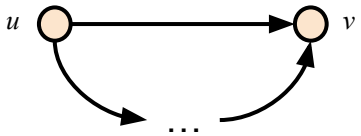
Teorema 1: Dado un digrafo G = (V, E) sin ciclos (DAG),

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

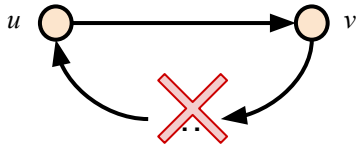
Caso 1: start[u] < start[v]



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

❑

Caso 2: 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 $G = (V, E)$,

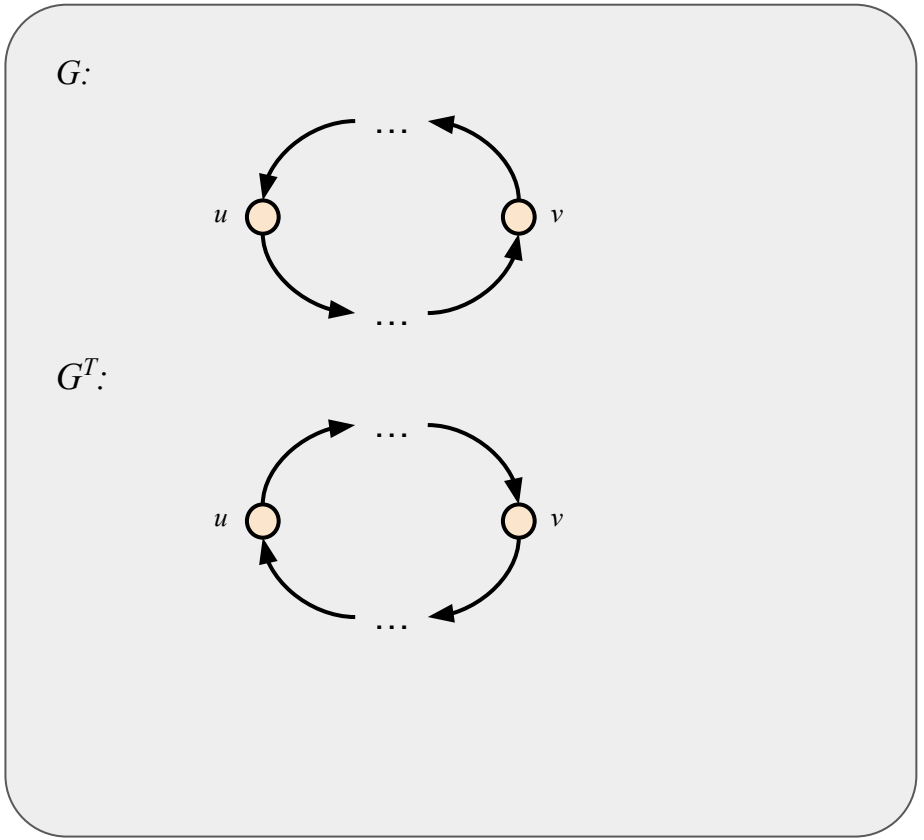
$G^T = (V, E^T)$ con $E^T = \{(u,v): (v,u) \in E\}$

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

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!

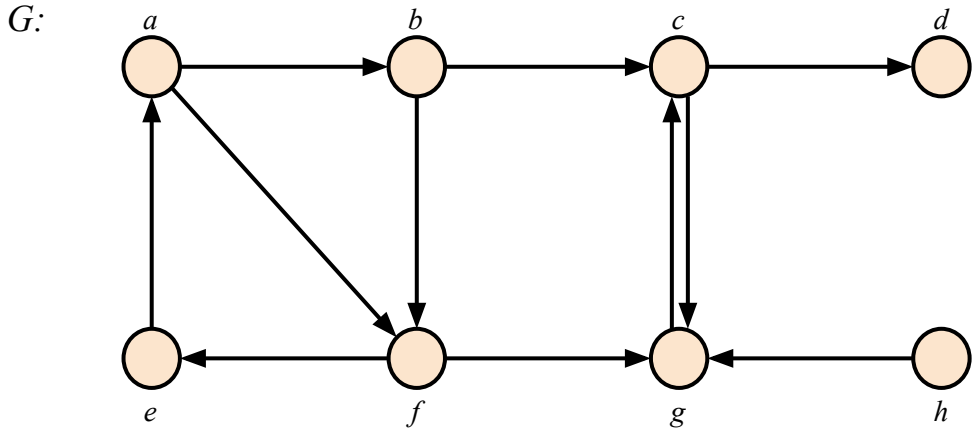
Kosaraju (componentes fuertemente conexas)

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

Los árboles resultantes son las componentes conexas.

Kosaraju (componentes fuertemente conexas)

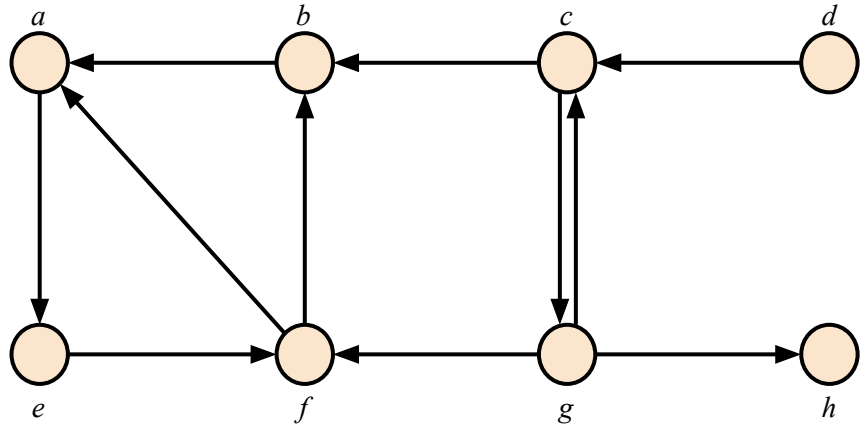
```
KOSARAJU ( G ) :
|     computo G^T
|     DFS ( G^T )
|     DFS ( G )  # usando finish de G^T
                 en sentido inverso
```



Kosaraju (componentes fuertemente conexas)

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

G^T:



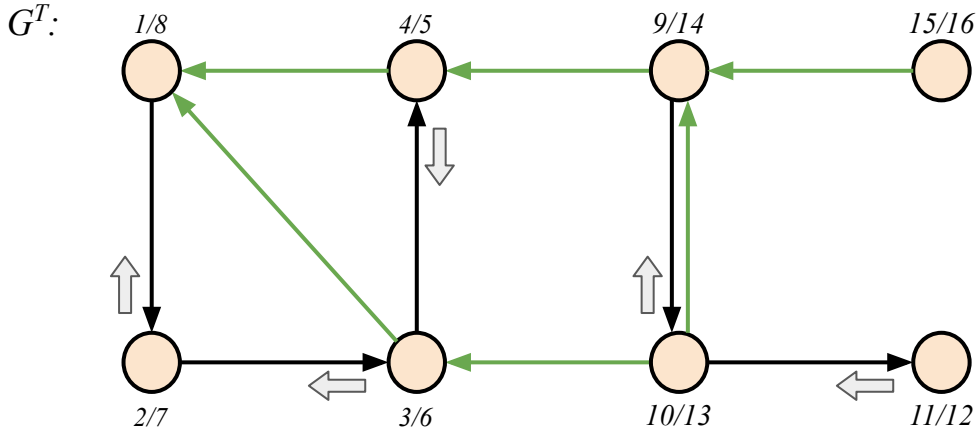
Kosaraju (componentes fuertemente conexas)

```
KOSARAJU ( G ) :
|     computo G^T
|     DFS ( G^T )
|     DFS ( G )   # usando finish de G^T
                  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}

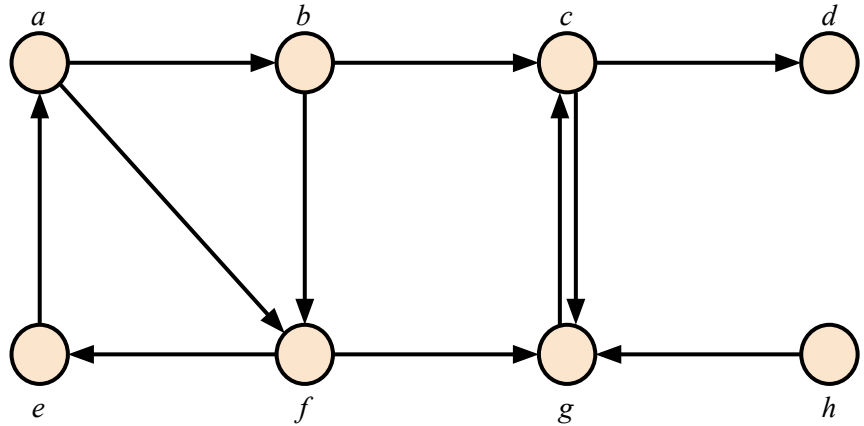


Kosaraju (componentes fuertemente conexas)

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

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

G:

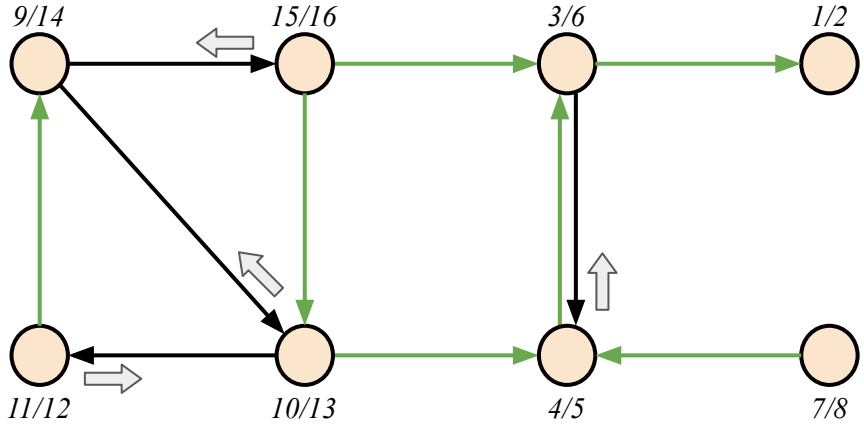


Kosaraju (componentes fuertemente conexas)

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

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

G:

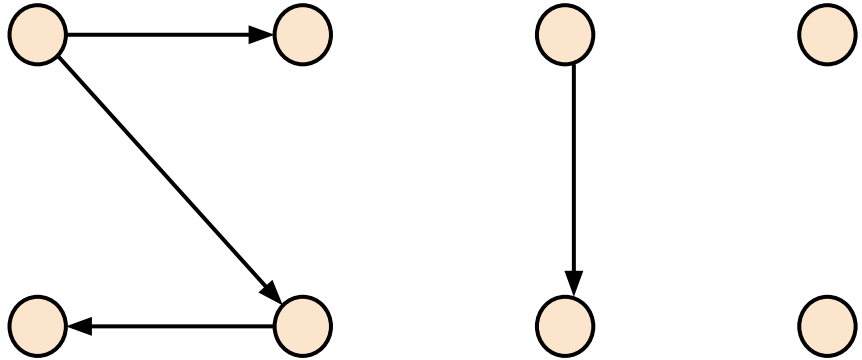


Kosaraju (componentes fuertemente conexas)

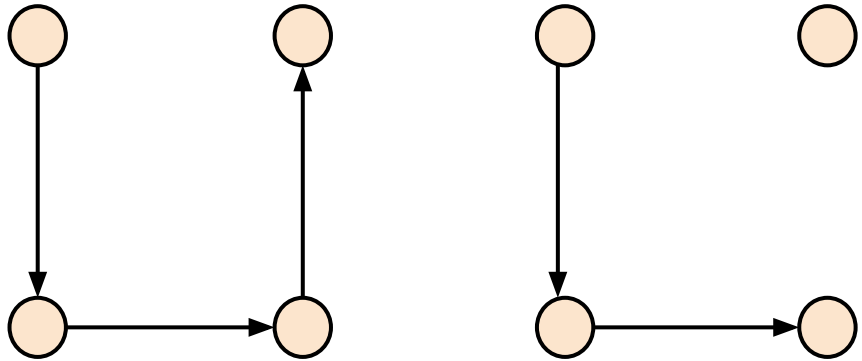
```
KOSARAJU ( G ) :
|     computo G^T
|     DFS ( G^T )
|     DFS ( G )  # usando finish de G^T
                 en sentido inverso
```


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

G:

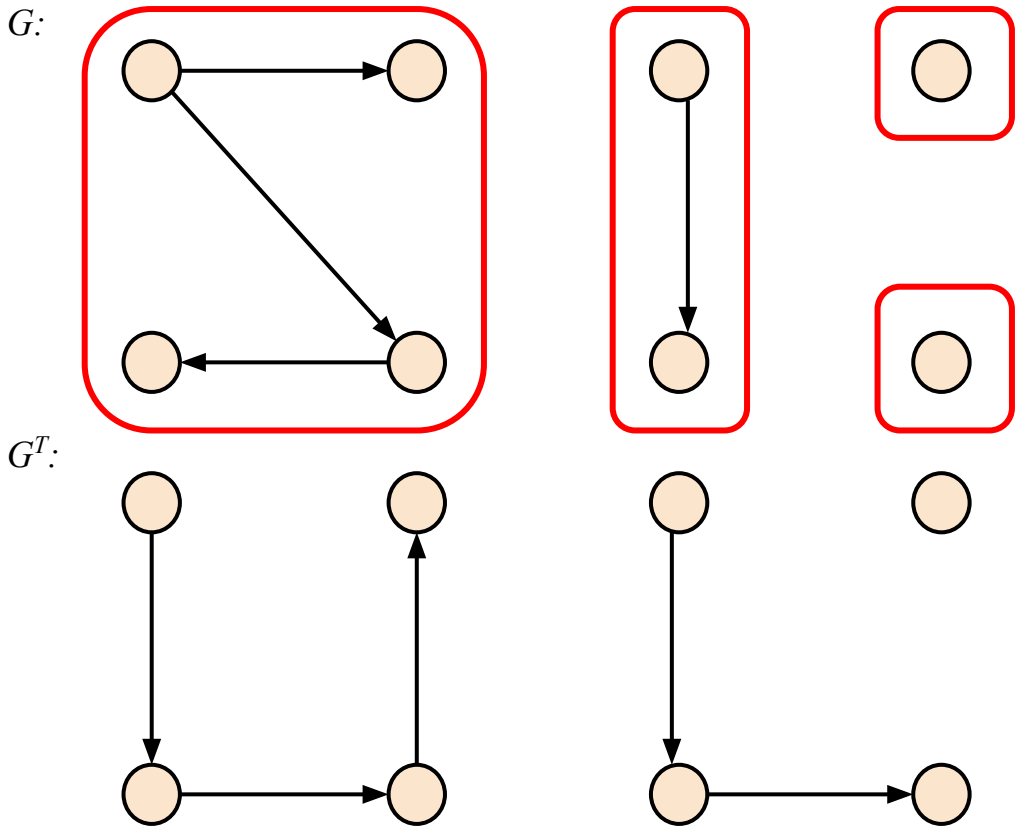


G^T:



Kosaraju (componentes fuertemente conexas)

```
KOSARAJU ( G ) :
|     computo G^T
|     DFS ( G^T )
|     DFS ( G )  # usando finish de G^T
                 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
```