Yadran Eterovic

Grafos

2 - 2017

Estructuras de Datos y Algoritmos - IIC2133

Ítemes y conexiones entre pares de ítemes

Mapas

El contenido de la Web

Circuitos eléctricos

Programación de tareas

Comercio

Postulaciones

Redes de computadores

Software

Redes sociales

Un **grafo** G es un par (V, E)

V es un **conjunto** (finito) **de vértices** (o *nodos*) de G

E es una relación binaria sobre V y es el **conjunto de aristas** (o *arcos*) de G —está formado por pares de vértices

Distinguimos cuatro tipos de grafos

En un **grafo direccional**, *E* está compuesto por pares ordenados de vértices:

- las aristas tienen dirección

En un **grafo no direccional**, *E* está compuesto por pares no ordenados de vértices:

- las aristas no tienen dirección

Ambos tipos de grafos pueden ser **sin costos** o **con costos**

… en este último caso, cada arista tiene asociado un número real

Hay dos formas típicas de representar un grafo $G = (V, E)$

Una colección de $|V|$ **listas de adyacencias**, que ocupa $\Theta(V+E)$ memoria

… preferida para representar grafos poco densos

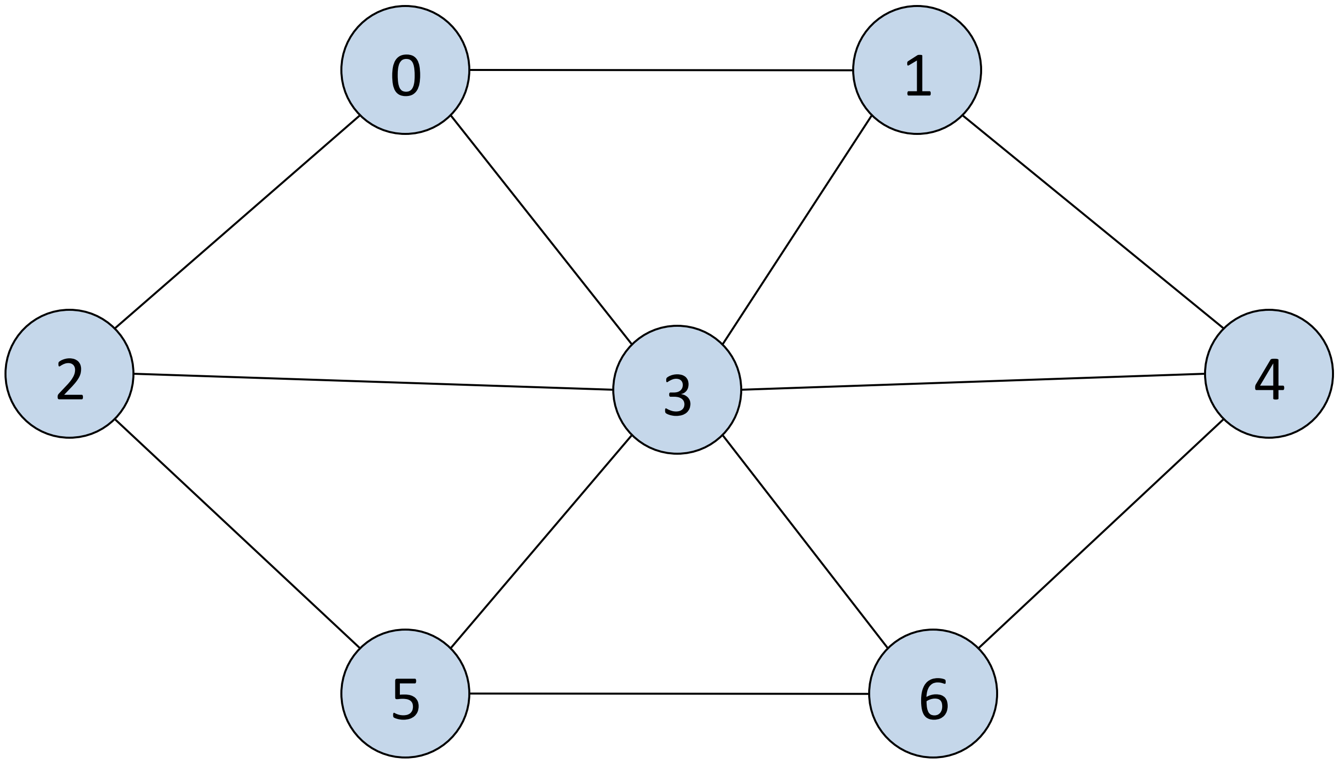
Una **matriz de adyacencias**, que ocupa $\Theta(V^2)$ memoria

… preferida para representar grafos densos

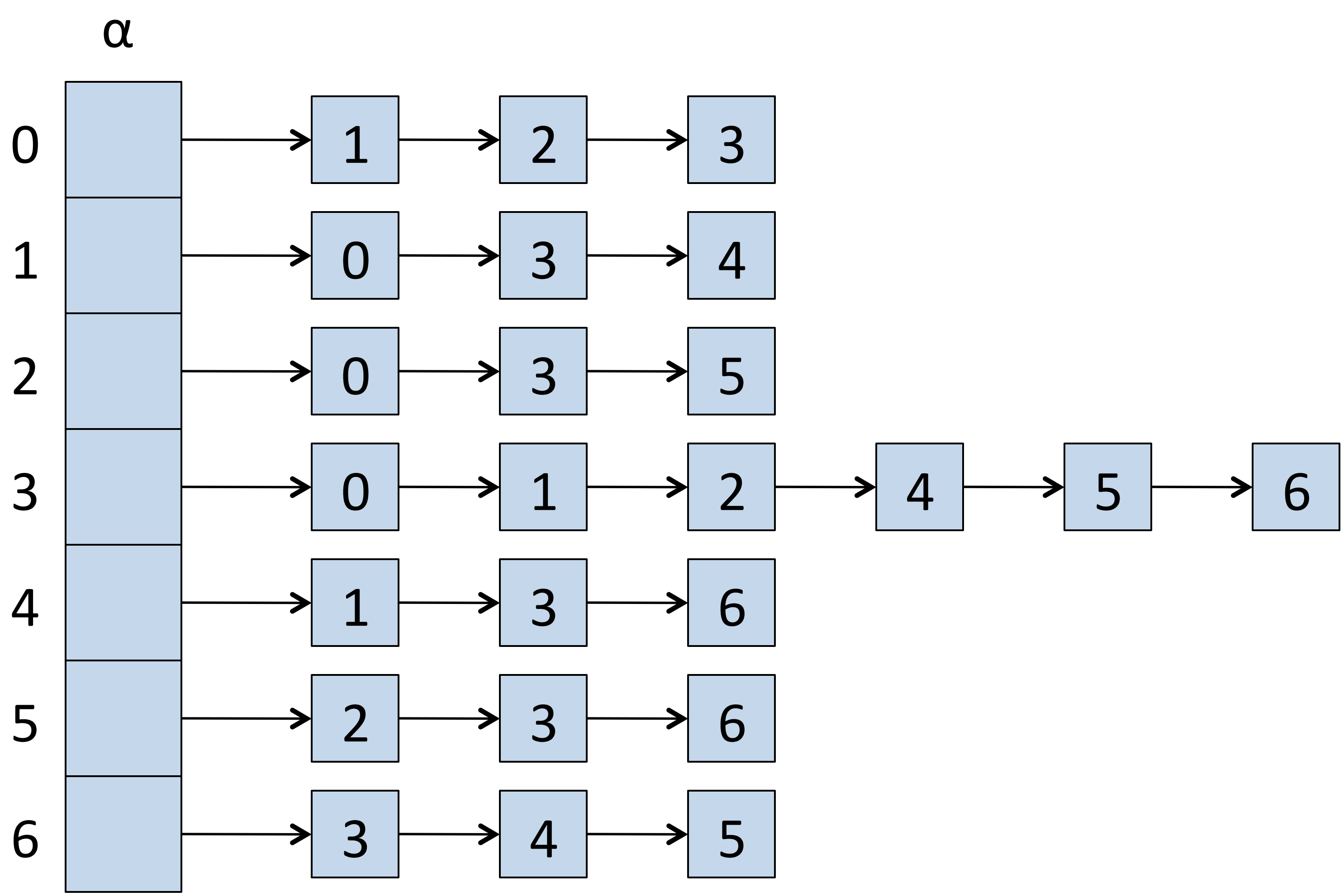
… o cuando queremos saber rápidamente si hay una arista entre dos vértices dados

Ambas son aplicables a grafos direccionales y no direccionales, con costos y sin costos

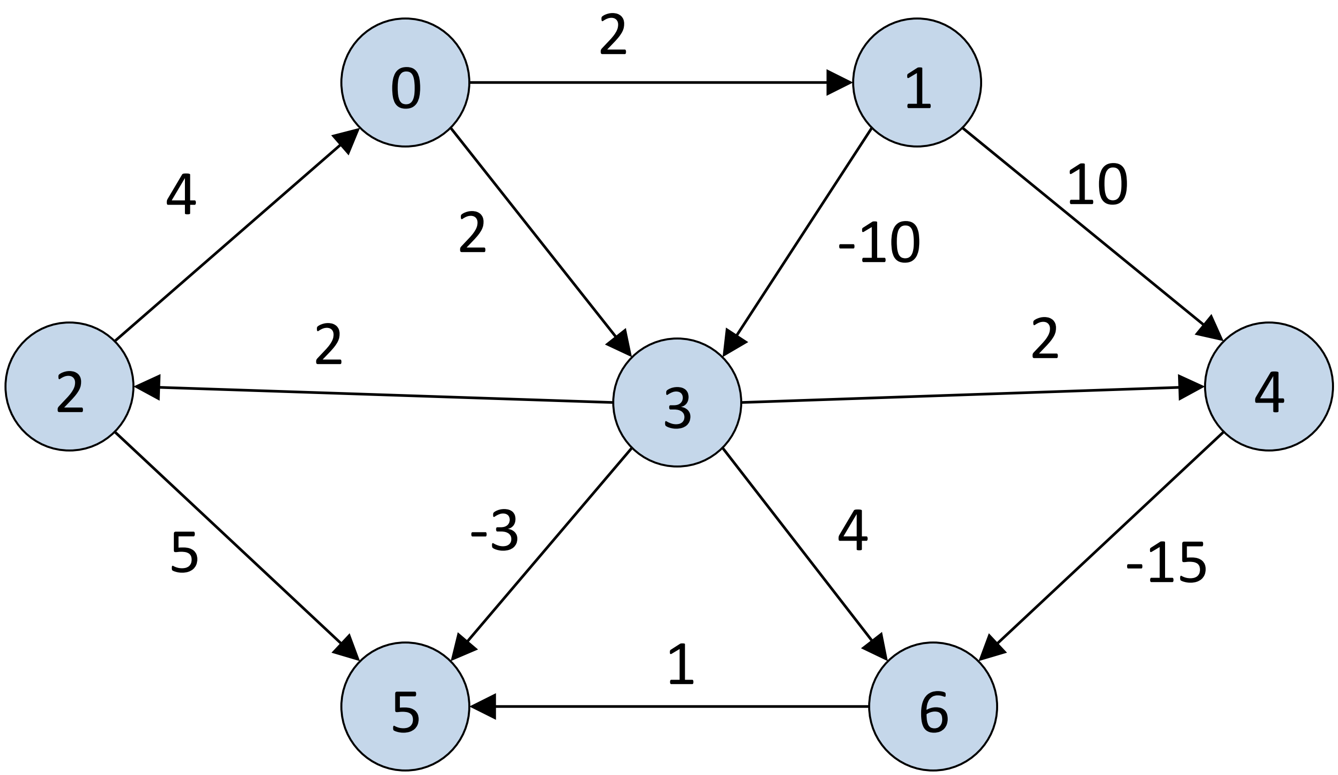
Un grafo *no direccional sin costos*



El grafo anterior representado por $|V|$ listas de adyacencias, α



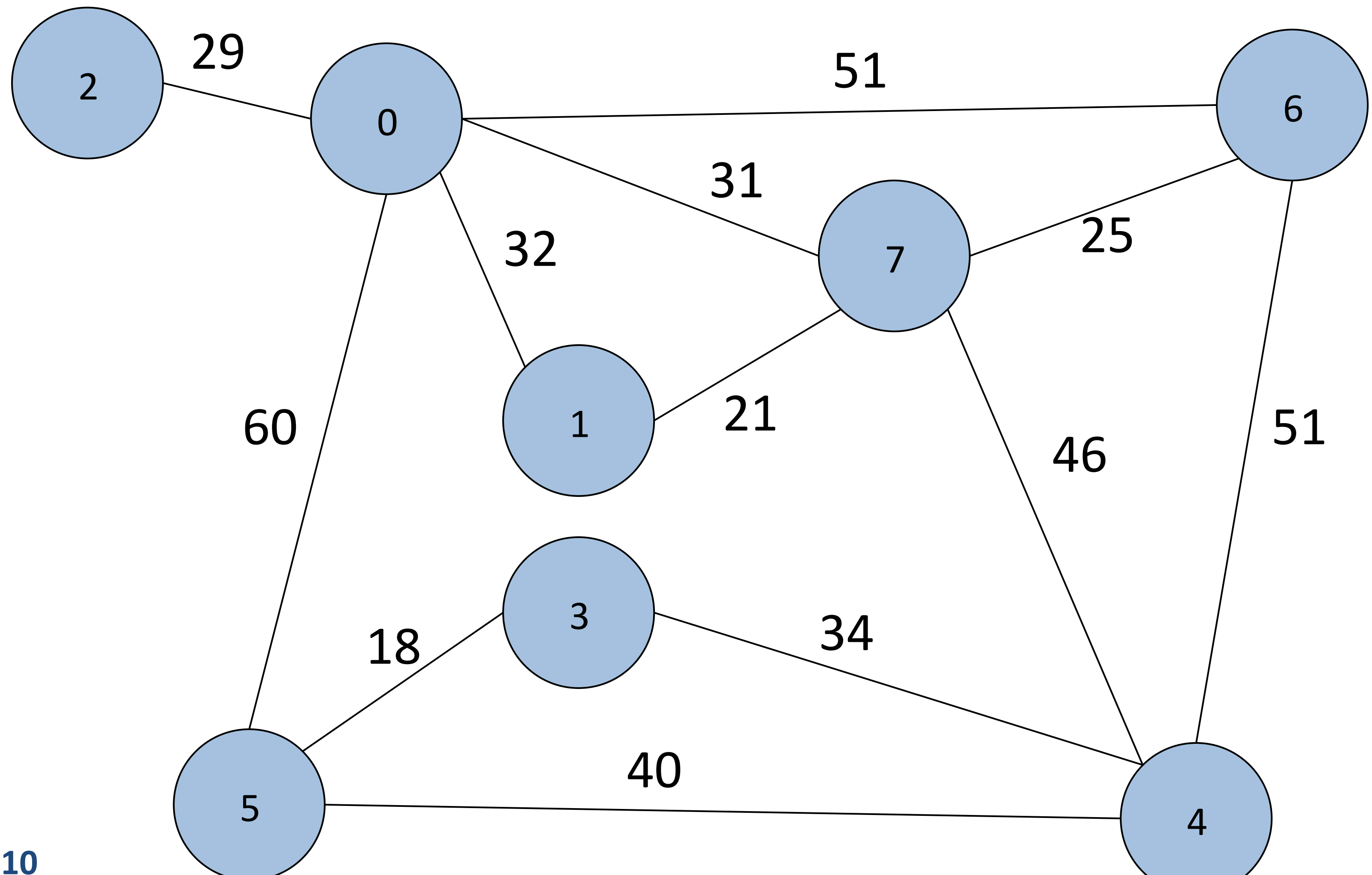
Un grafo *direccional con costos*



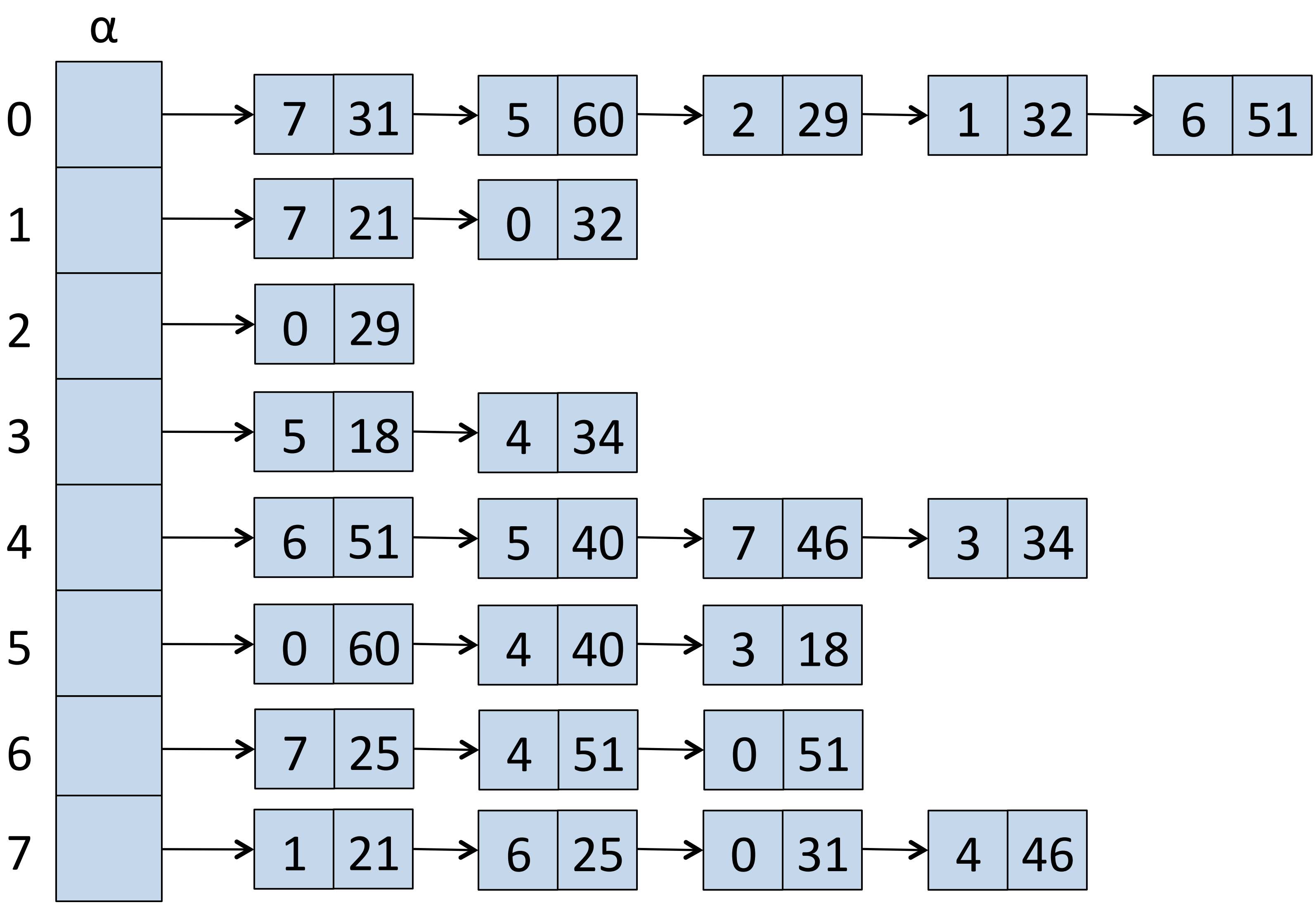
El grafo anterior representado por una matriz de adyacencias

	0	1	2	3	4	5	6
0		2		2			
1				-10	10		
2	4					5	
3			2		2	-3	4
4							-15
5							
6						1	

Un grafo *no direccional con costos*



El grafo anterior representado por $|V|$ listas de adyacencias, α



Al estudiar grafos, empleamos varias estructuras de datos y algoritmos ya conocidos

Listas ligadas y arreglos

Stacks y colas

(min-) *Heaps*

Tablas de *hash*

Ordenación

Recorrido en preorden

… y algunas estructuras de datos y estrategias algorítmicas nuevas

Estructuras de datos para conjuntos disjuntos

Algoritmos codiciosos

Programación dinámica

Dos algoritmos fundamentales que permiten determinar la estructura del grafo

Breadth first search (BFS), o **exploración en amplitud**

Depth first search (DFS), o **exploración en profundidad**

Ambos algoritmos exploran sistemáticamente el grafo a partir de algún vértice

… y descubren propiedades fundamentales del grafo

BFS: Exploración en amplitud

Dado $G = (V, E)$ y un vértice s, **BFS** explora las aristas E para descubrir todos los vértices que sean alcanzables desde s :

- calcula la distancia —en número de aristas— desde s a cada vértice **alcanzable**
- produce un **árbol** **BFS** con raíz s que contiene todos los vértices alcanzables desde s
- para todo vértice v alcanzable, la ruta en el árbol BFS desde s a v es una **ruta más corta** (menor número de aristas) de s a v en G
- sirve para grafos direccionales y no direccionales

BFS expande sistemáticamente la frontera entre vértices descubiertos y no descubiertos

Expande la frontera entre los vértices descubiertos y los no descubiertos, uniformemente en toda la amplitud de la frontera

Descubre todos los vértices a distancia k de s antes de descubrir algún vértice a distancia k+1

BFS pinta los vértices blancos, grises o negros

Todos los vértices empiezan *blancos*

Un vértice es **descubierto** la primera vez que es encontrado, y en ese momento se pinta de *gris*

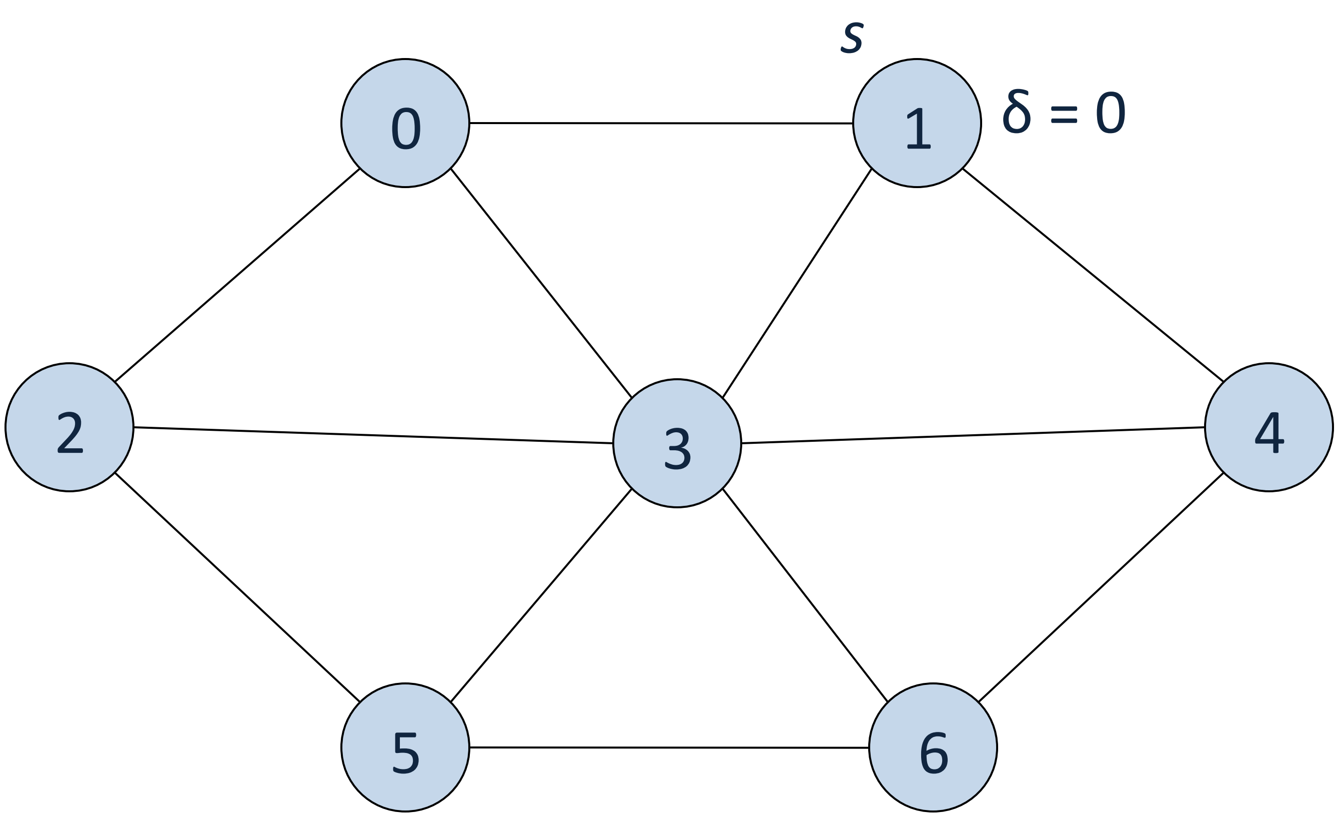
Los vértices grises representan la frontera entre los vértices descubiertos y los no descubiertos

Cuando todos los vértices adyacentes a un cierto vértice *u* han sido descubiertos, *u* se pinta de *negro*

```
bfs(s): —s es el vértice de partida
    for each u in V-{s}:
        u.color = white; u.δ = ∞; π[u] = null
    s.color = gray; s.δ = 0; π[s] = null
    q = Queue(); q.enqueue(s)
    while !q.empty():
        u = q.dequeue()
        for each v in α[u]:
            if v.color == white:
                v.color = gray; v.δ = u.δ+1
                π[v] = u; q.enqueue(v)
        u.color = black
```

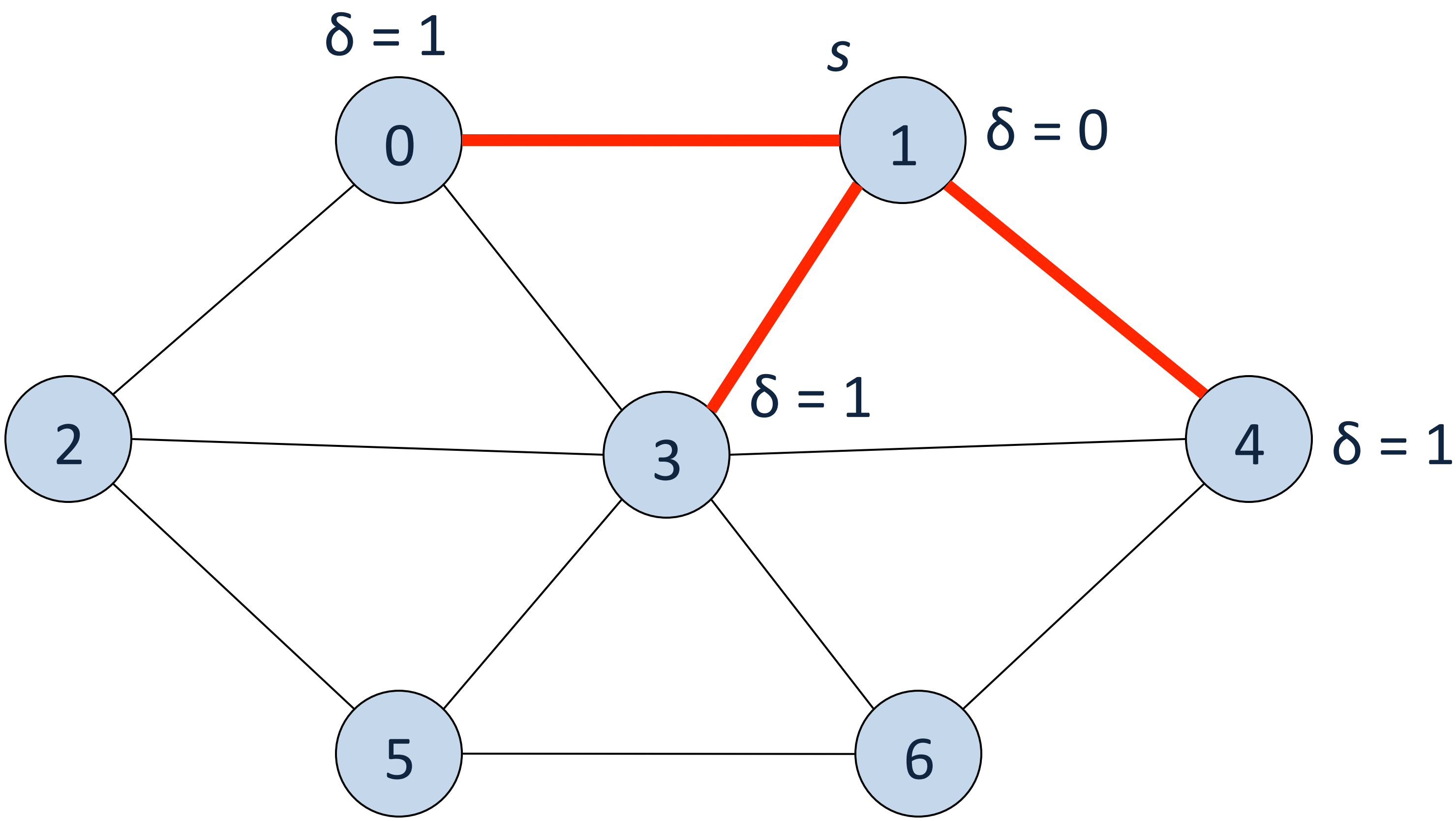
BFS del grafo de la diap. #5, a partir del vértice s = 1:

Vértices a distancia 0 de s



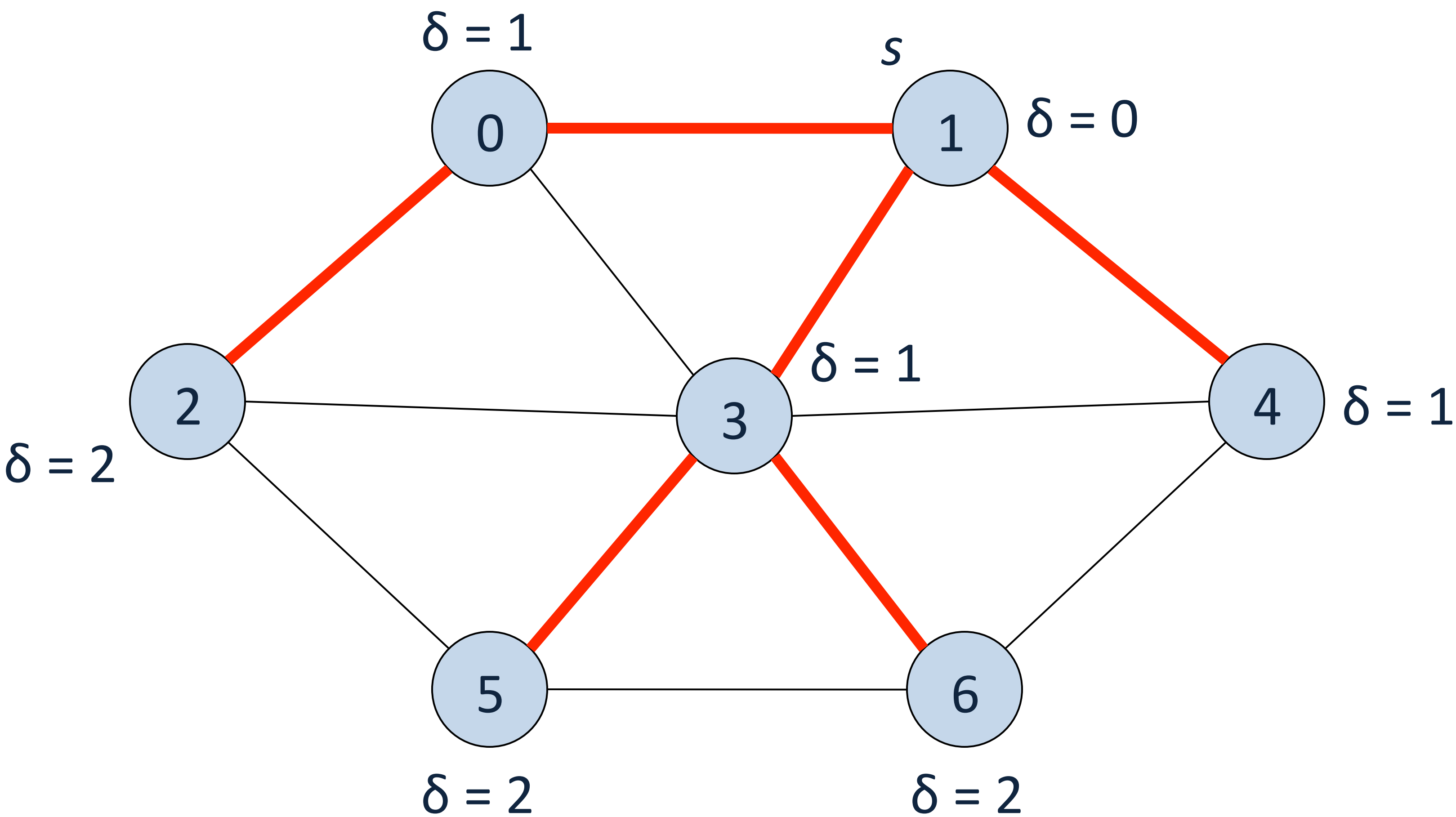
BFS del grafo de la diap. #5, a partir del vértice *s* = 1:

Vértices a distancias 0 y 1 de *s*



BFS del grafo de la diap. #5, a partir del vértice s = 1:

Vértices a distancias 0, 1 y 2 de s



DFS: Exploración en profundidad

Las aristas son exploradas a partir del vértice *v* descubierto más recientemente

… que aún tiene aristas no exploradas que salen de él:

- cuando todas las aristas de *v* han sido exploradas, la exploración retrocede para explorar aristas que salen del vértice a partir del cual *v* fue descubierto
- busca más profundamente en *G* mientras sea posible

DFS pinta los vértices blancos, grises o negros

Todos los vértices son inicialmente *blancos*

Un vértice se pinta de *gris* cuando es descubierto

Un vértice se pinta de *negro* cuando su lista de adyacencias ha sido examinada exhaustivamente

DFS asigna dos *tiempos* a cada vértice *v*

v.d registra el tiempo (la hora) cuando *v* es descubierto (por primera vez)

… y, consecuentemente, pintado de gris

v.f registra el tiempo cuando la lista de adyacencias de *v* ha sido examinada exhaustivamente

… y, consecuentemente, *v* ha sido pintado de negro

```
dfs():
   for each u in V:
      u.color = white
      π[u] = null
   time = 0
   for each u in V:
      if u.color == white:
         time = dfsVisit(u, time)
```

```
dfsVisit(u, time):
   u.color = gray
   time = time+1
   u.d = time
   for each v in α[u]:
      if v.color == white:
         π[v] = u
         time = dfsVisit(v, time)
   u.color = black
   time = time+1
   u.f = time
   return time
```

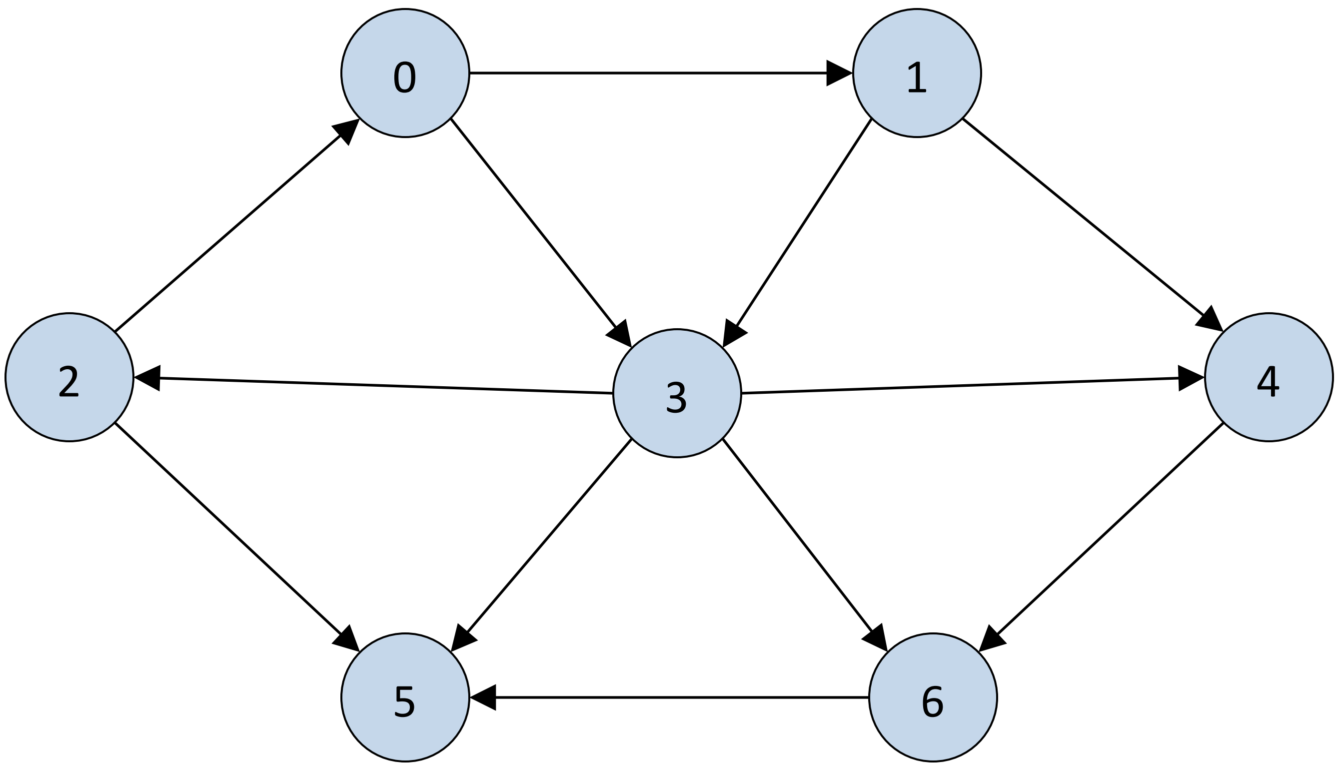
Para dos vértices cualquiera, *u* y *v*, sólo una de las siguientes tres condiciones es verdadera

Los intervalos [*u.d*, *u.f*] y [*v.d*, *v.f*] son *disjuntos*, y ni *u* ni *v* es descendiente del otro en el bosque DFS

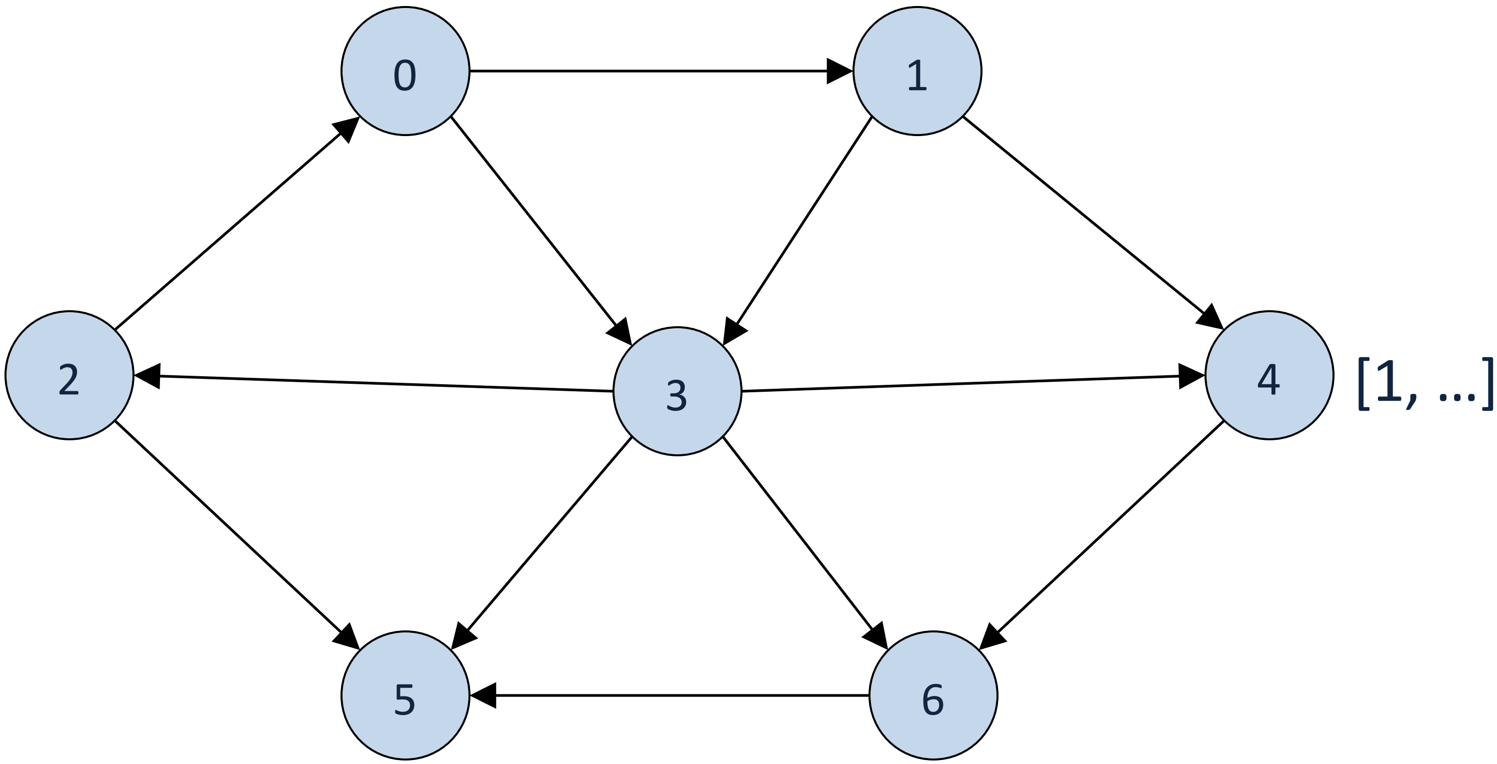
El intervalo [*u.d*, *u.f*] *está contenido* en el intervalo [*v.d*, *v.f*], y *u es descendiente de v* en un árbol DFS

El intervalo [*v.d*, *v.f*] *está contenido* en el intervalo [*u.d*, *u.f*], y *v es descendiente de u* en un árbol DFS

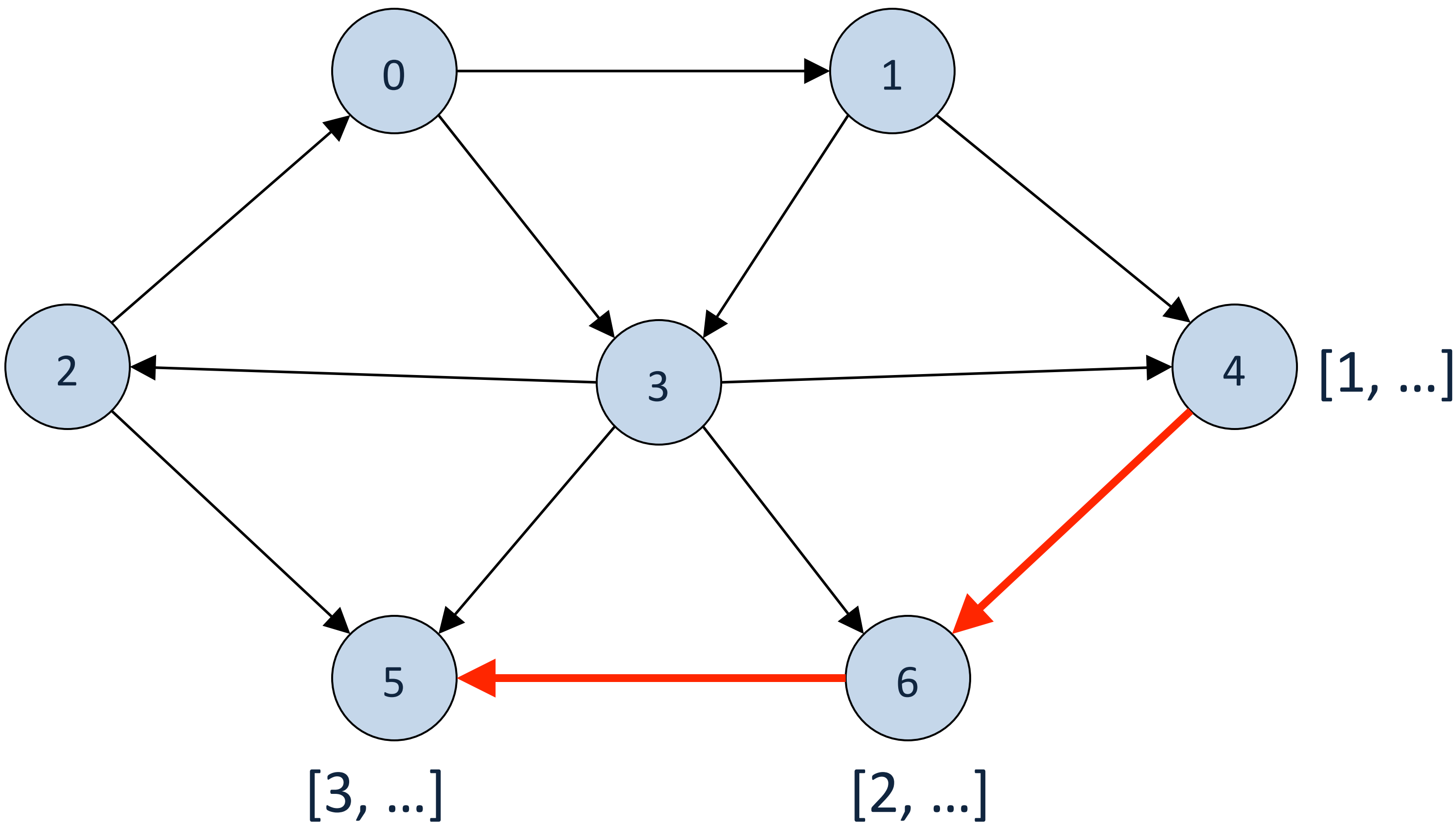
Un grafo, *G, direccional sin costos*



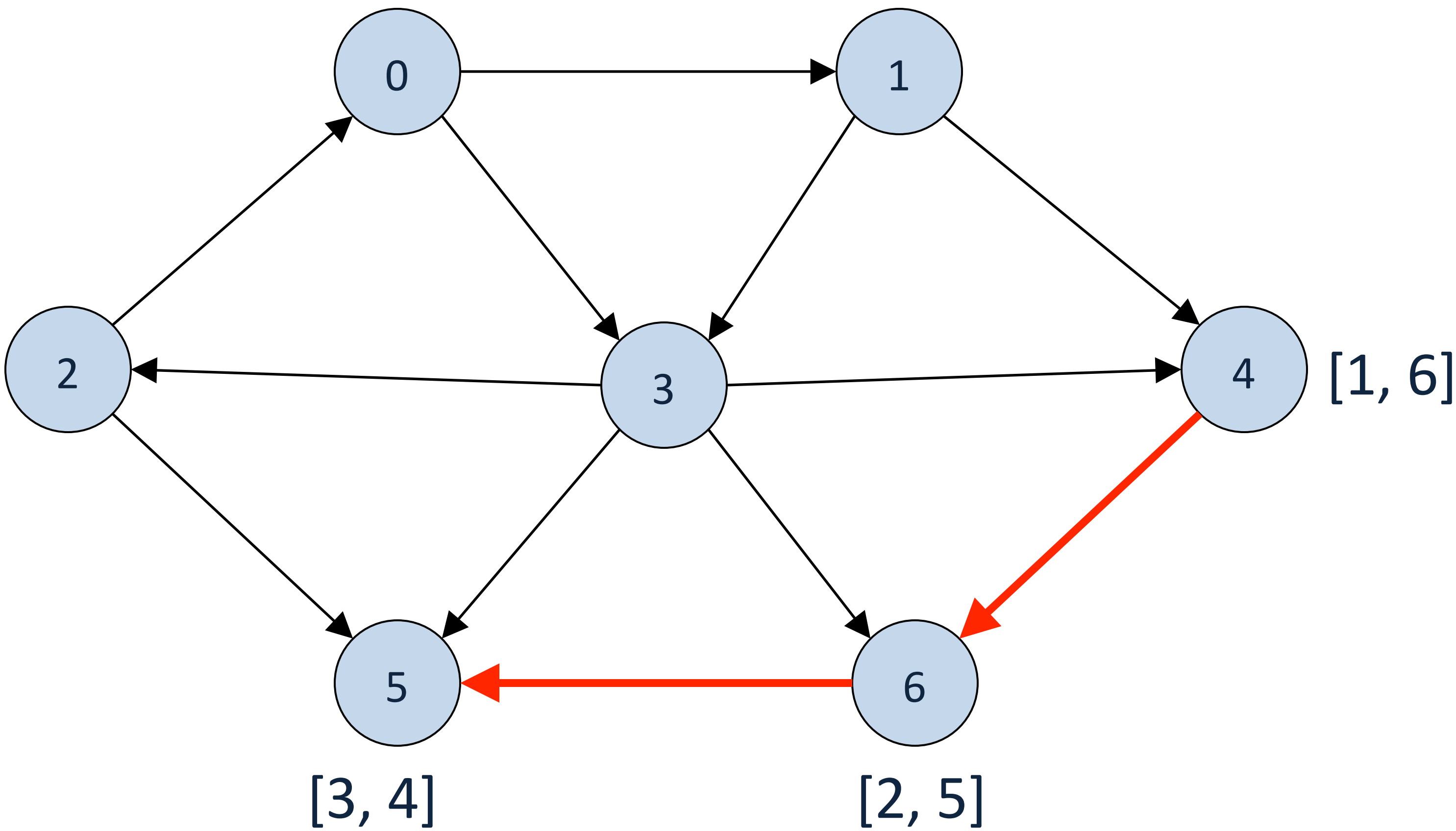
DFS de *G*, a partir del vértice 4 ...



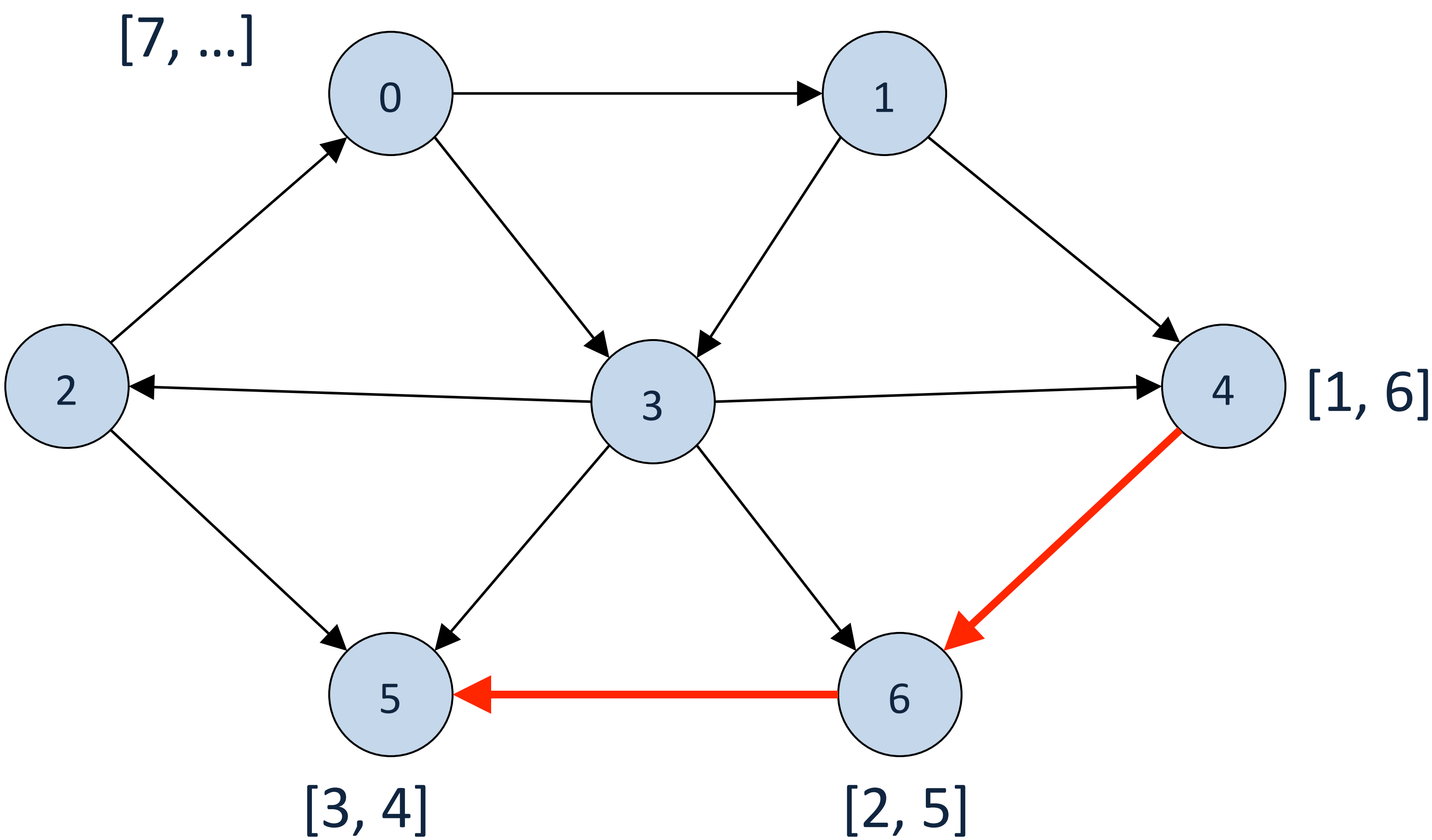
DFS de *G*, a partir del vértice 4 …



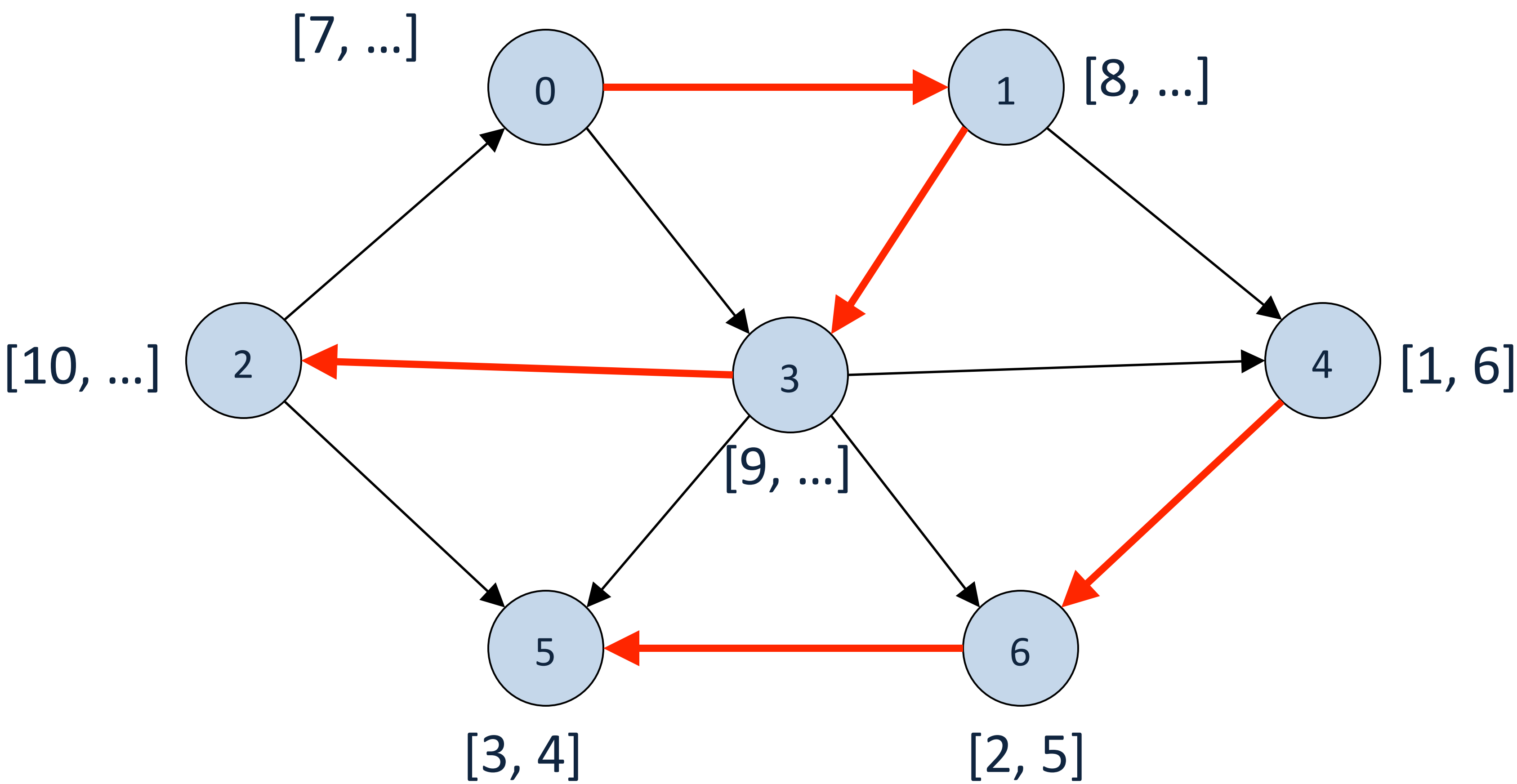
DFS de *G*, a partir del vértice 4 …



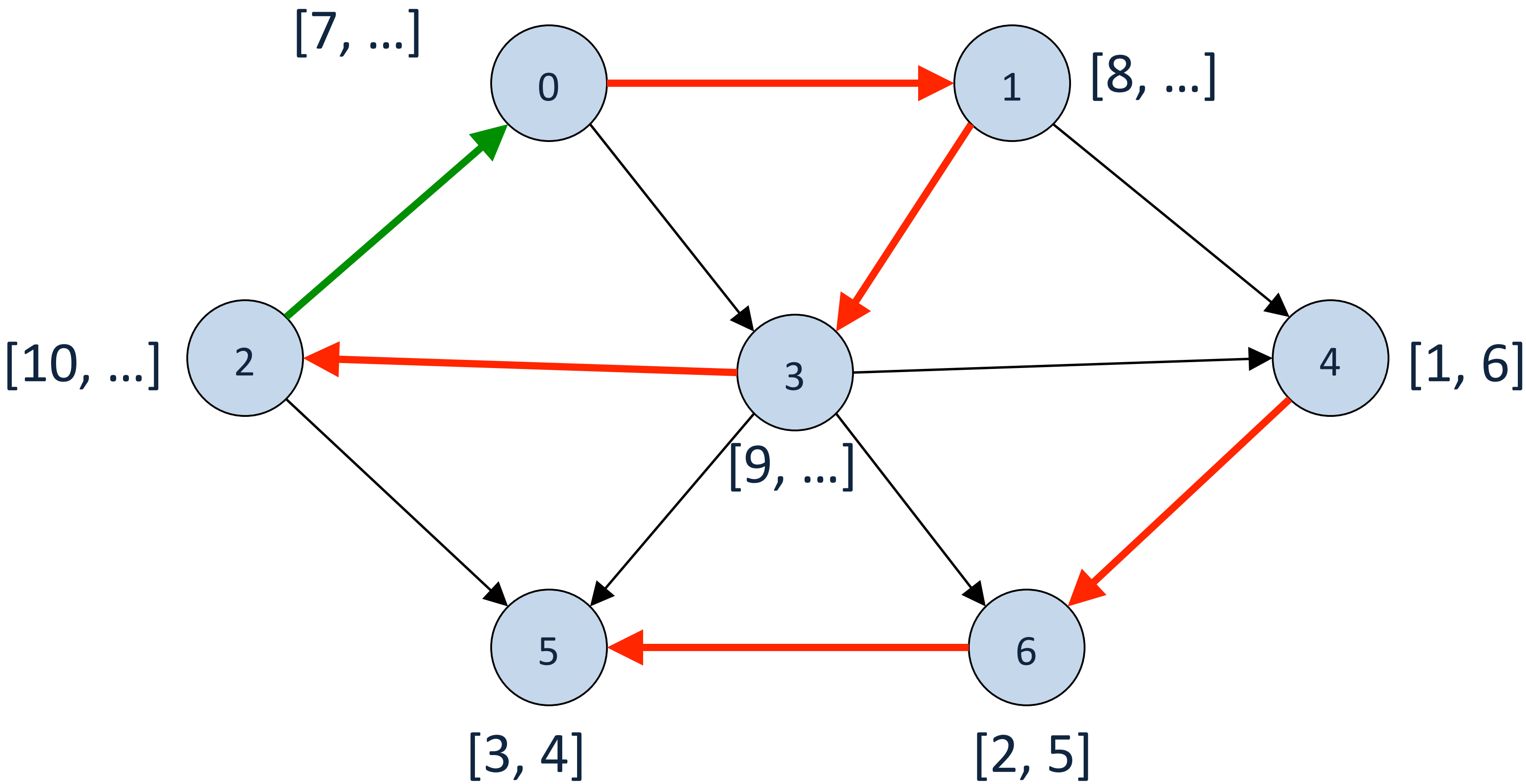
DFS de *G*, a partir del vértice 4 y, luego, del vértice 0



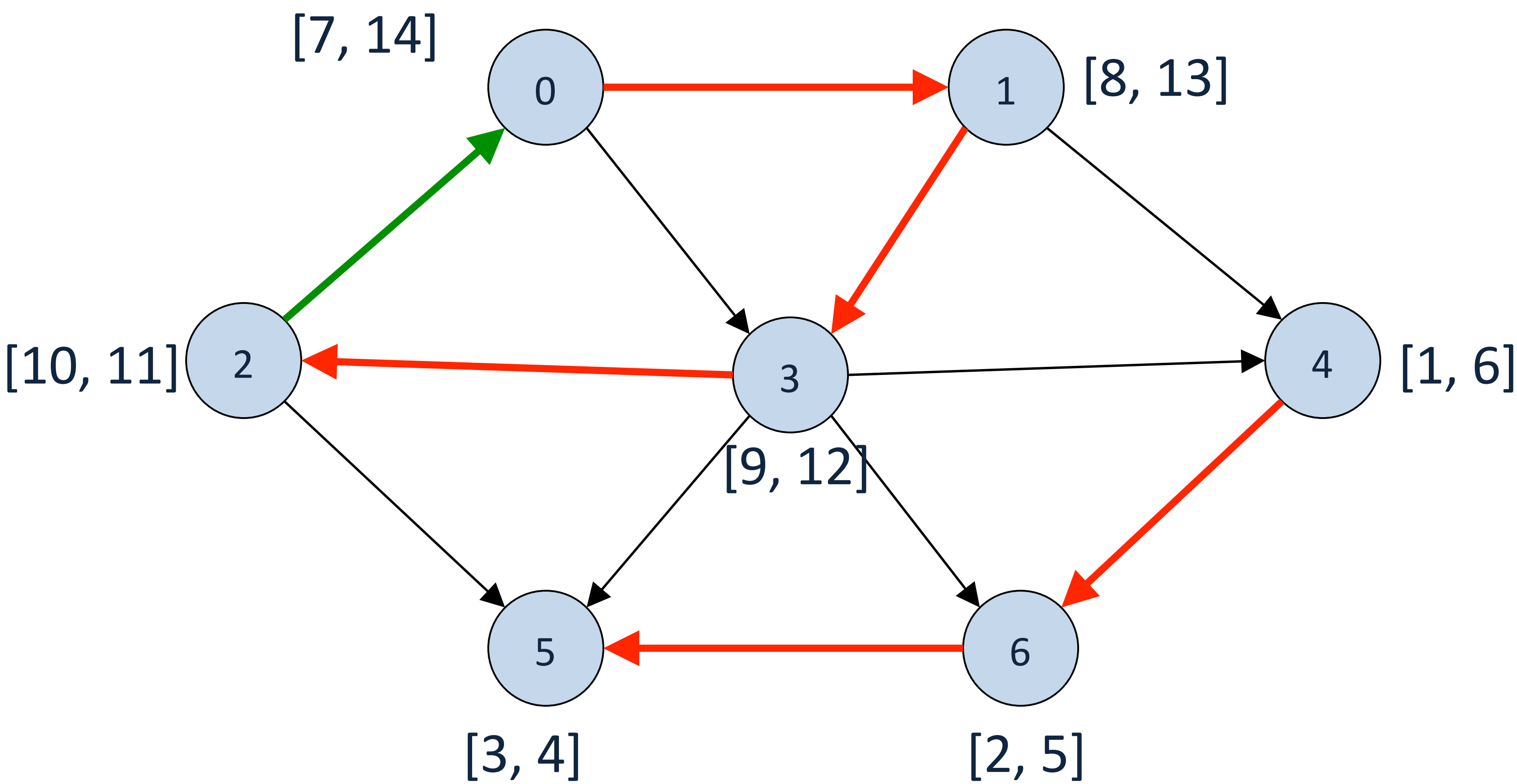
DFS de *G*, a partir del vértice 4
y, luego, del vértice 0



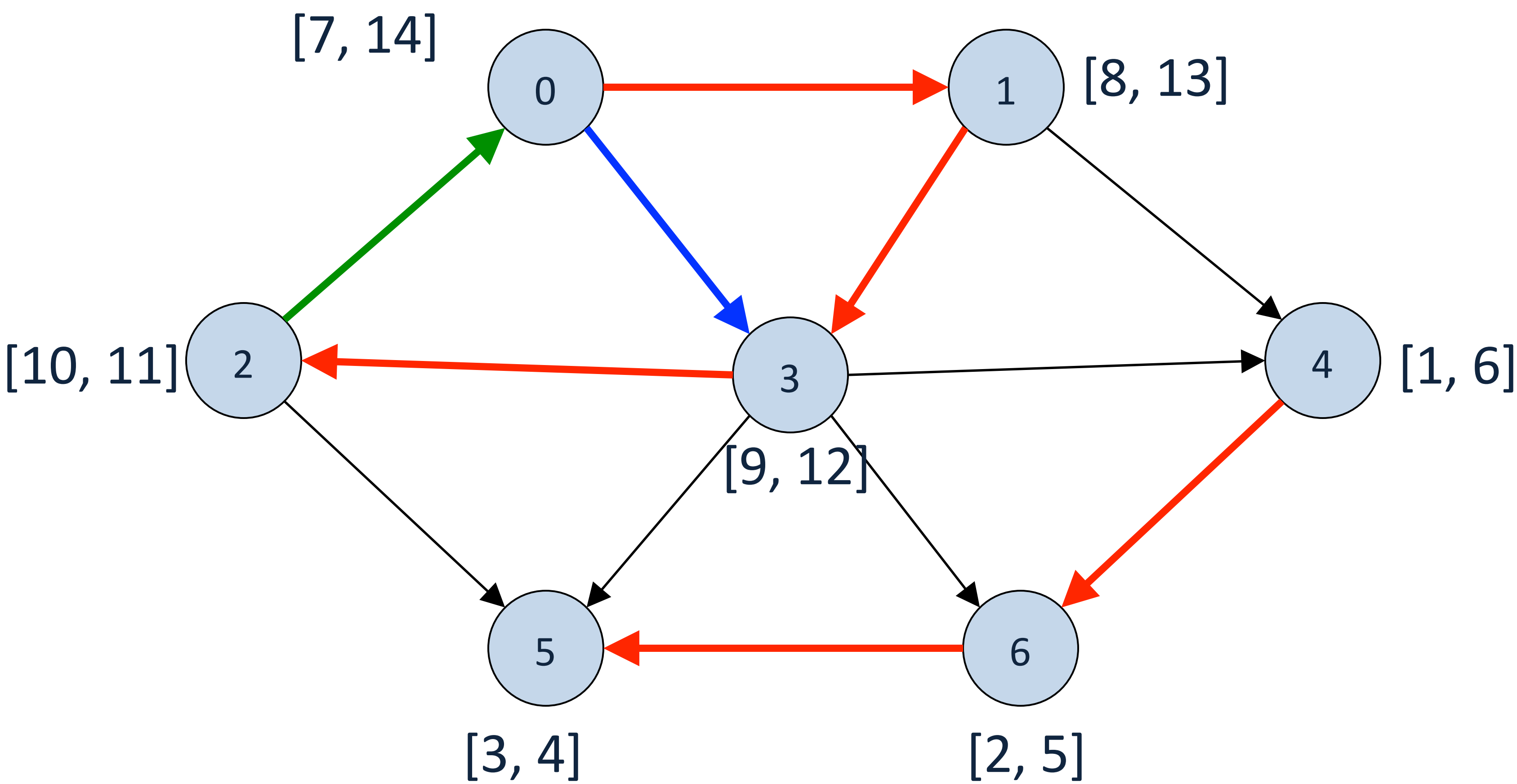
DFS de *G*, a partir del vértice 4 y, luego, del vértice 0



DFS de *G*, a partir del vértice 4 y, luego, del vértice 0



DFS de *G*, a partir del vértice 4
y, luego, del vértice 0



Definimos cuatro tipos de aristas en términos del bosque DFS G_π producido al explorar G

Aristas de árbol: aristas en el bosque G_π; la arista (u, v) es una arista de árbol si v fue descubierto por primera vez al explorar (u, v) —las aristas rojas

Aristas hacia atrás: aristas (u, v) que conectan un vértice u a un ancestro v en un árbol DFS —la arista verde

Aristas hacia adelante: aristas (u, v) que no son de árbol y conectan un vértice u a un descendiente v en un árbol DFS; *no aparecen en grafos no direccionales* —la arista azul

Aristas cruzadas: todas las otras aristas; *no aparecen en grafos no direccionales*

Tres problemas en grafos sin costos

a) Grafos no direccionales:

- encontrar las *componentes conectadas*

b) Grafos direccionales acíclicos:

- *ordenar* el grafo *topológicamente*

c) Grafos direccionales:

- encontrar las *componentes fuertemente conectadas*

¿Cómo determinamos las *componentes conectadas* de un grafo no direccional?

Un grafo no direccional se dice **conectado** si todo par de vértices está conectado por una ruta

La **componentes conectadas** de un grafo son las clases de equivalencia de vértices bajo la relación "*es alcanzable desde*"

Si G es estático, las componentes conectadas de G pueden ser determinadas usando DFS

El bosque G_π contiene tantos árboles como el número de componentes conectadas de G

Durante `dfs()`, asignamos a cada vértice u un número $c(u)$ entre 1 y k —en que k es el número de componentes conectadas de G— tal que $c(u) = c(v)$ si y sólo si u y v están en la misma componente conectada

(Para G dinámico, ver diap. #96)

Un grafo direccional acíclico *G* se puede *ordenar topológicamente*

La **ordenación topológica** de *G* es una ordenación lineal de todos los vértices

… tal que si *G* contiene la arista (*u*, *v*), entonces *u* aparece antes que *v* en la ordenación

El algoritmo de ordenación topológica

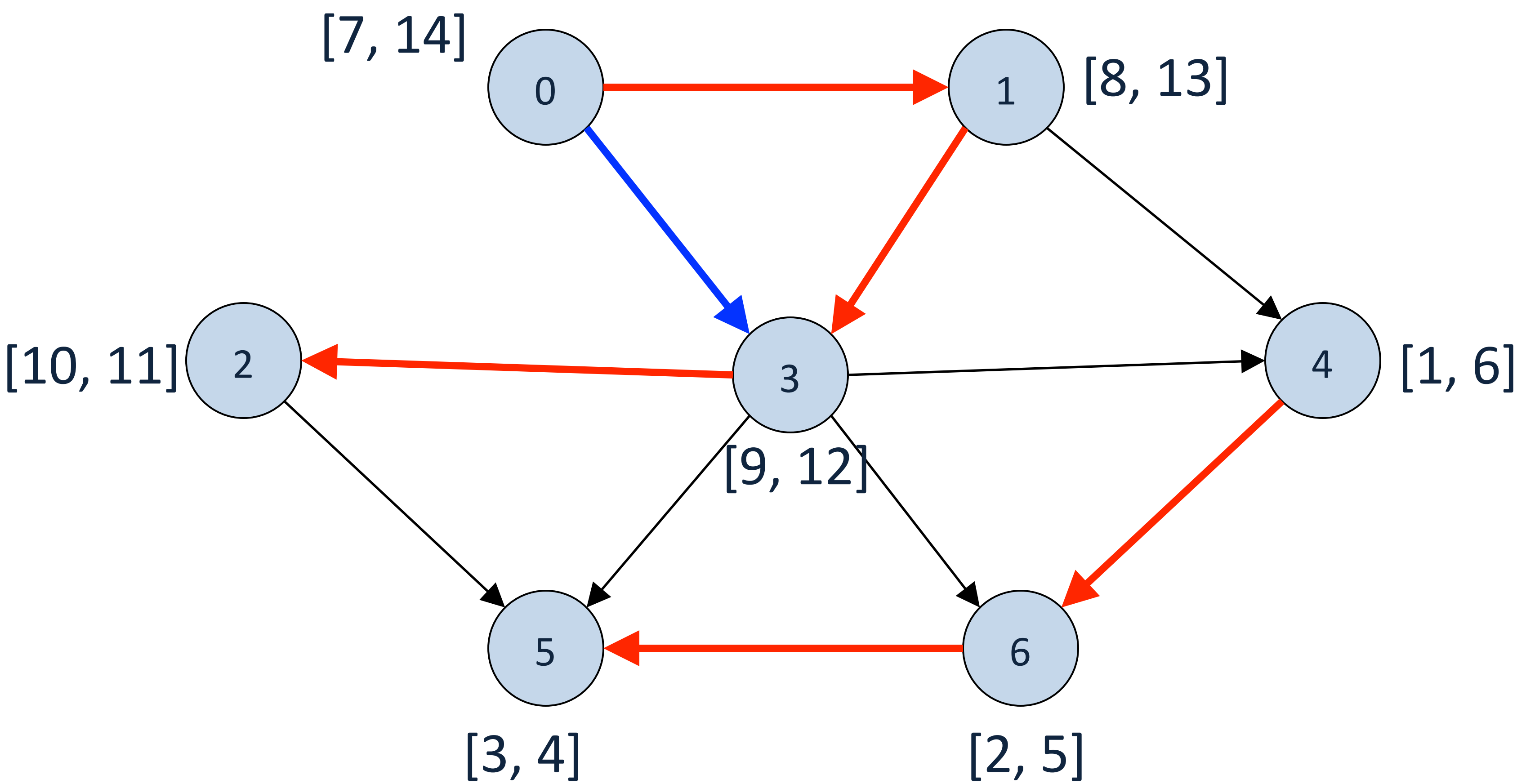
`topSort()`

1) Ejecute `dfs()` para calcular los tiempos f para cada vértice

2) Cada vez que calcule el tiempo f para un vértice, inserte ese vértice al frente de una lista ligada

3) `return` la lista ligada de vértices

DFS de un grafo acíclico, a partir del vértice 4 y, luego, del vértice 0



En un grafo que tiene ciclos es *imposible* producir un orden lineal de sus vértices:

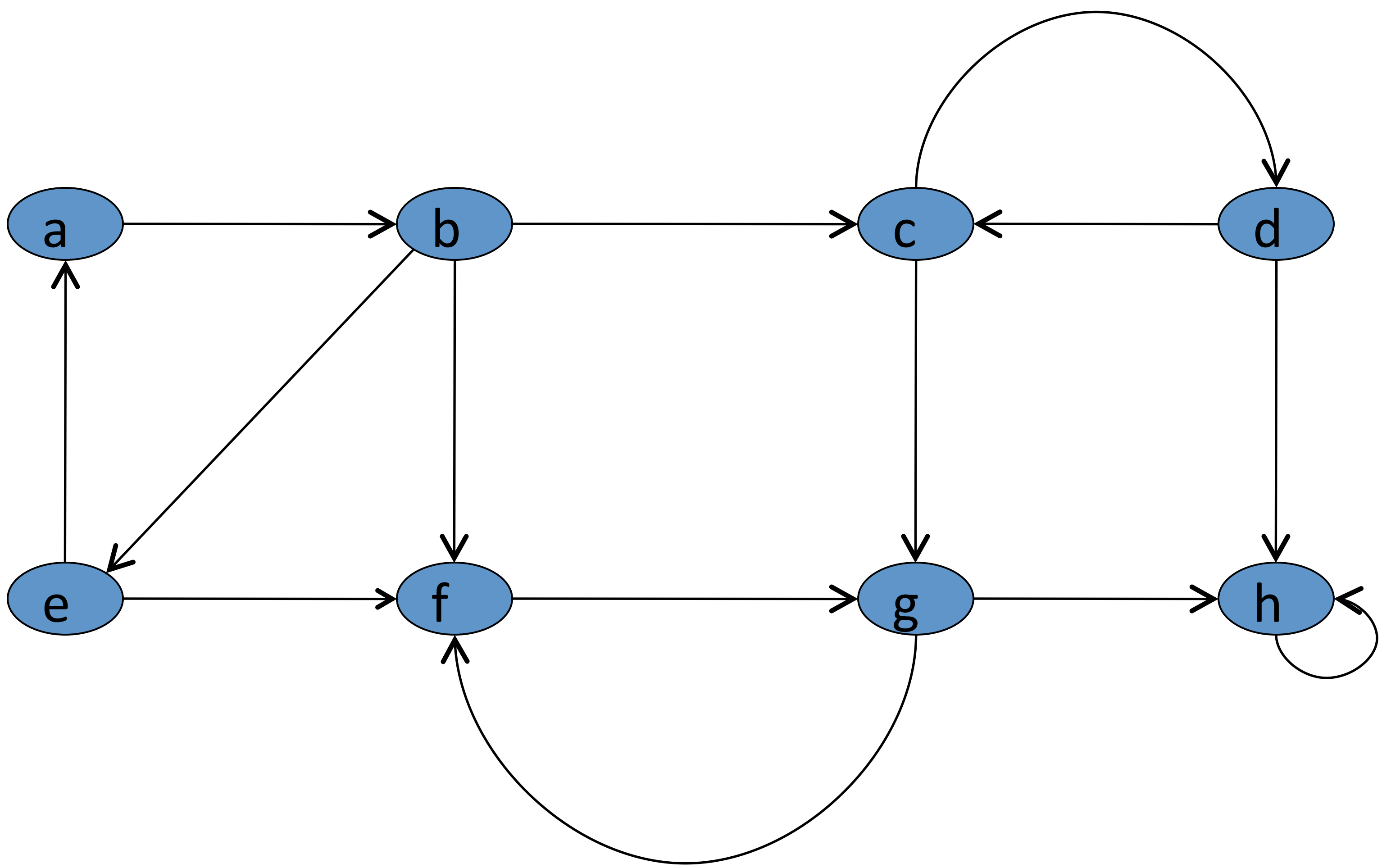
- p.ej., el grafo de la diap. #26

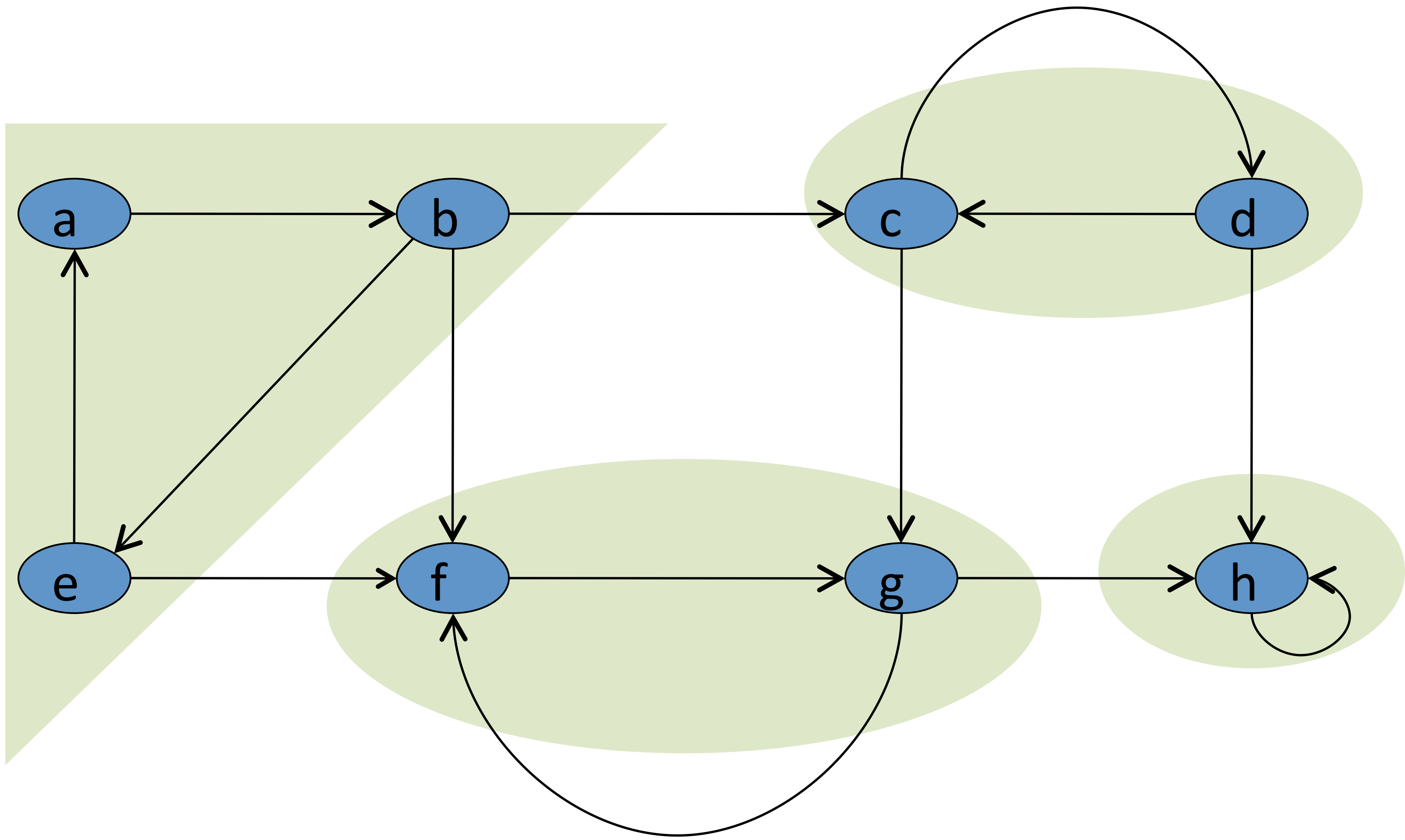
Un grafo direccional *G* es acíclico si y sólo si DFS de *G* no produce aristas hacia atrás

Para demostrar la corrección de `topSort()`, basta demostrar que para cualquier par de vértices u, v, si hay una arista de u a v, entonces $v.f < u.f$

¿Qué son las componentes *fuertemente* conectadas de un grafo direccional?

Las **componentes fuertemente conectadas** (scc's) de un grafo direccional $G = (V, E)$ son conjuntos máximos de vértices $C \subseteq V$ tales que para todo par de vértices u y v en C, u y v son mutuamente alcanzables —se puede llegar a v desde u, y se puede llegar a u desde v


a
b
c
d
e
f
g
h


a
b
c
d
e
f
g
h

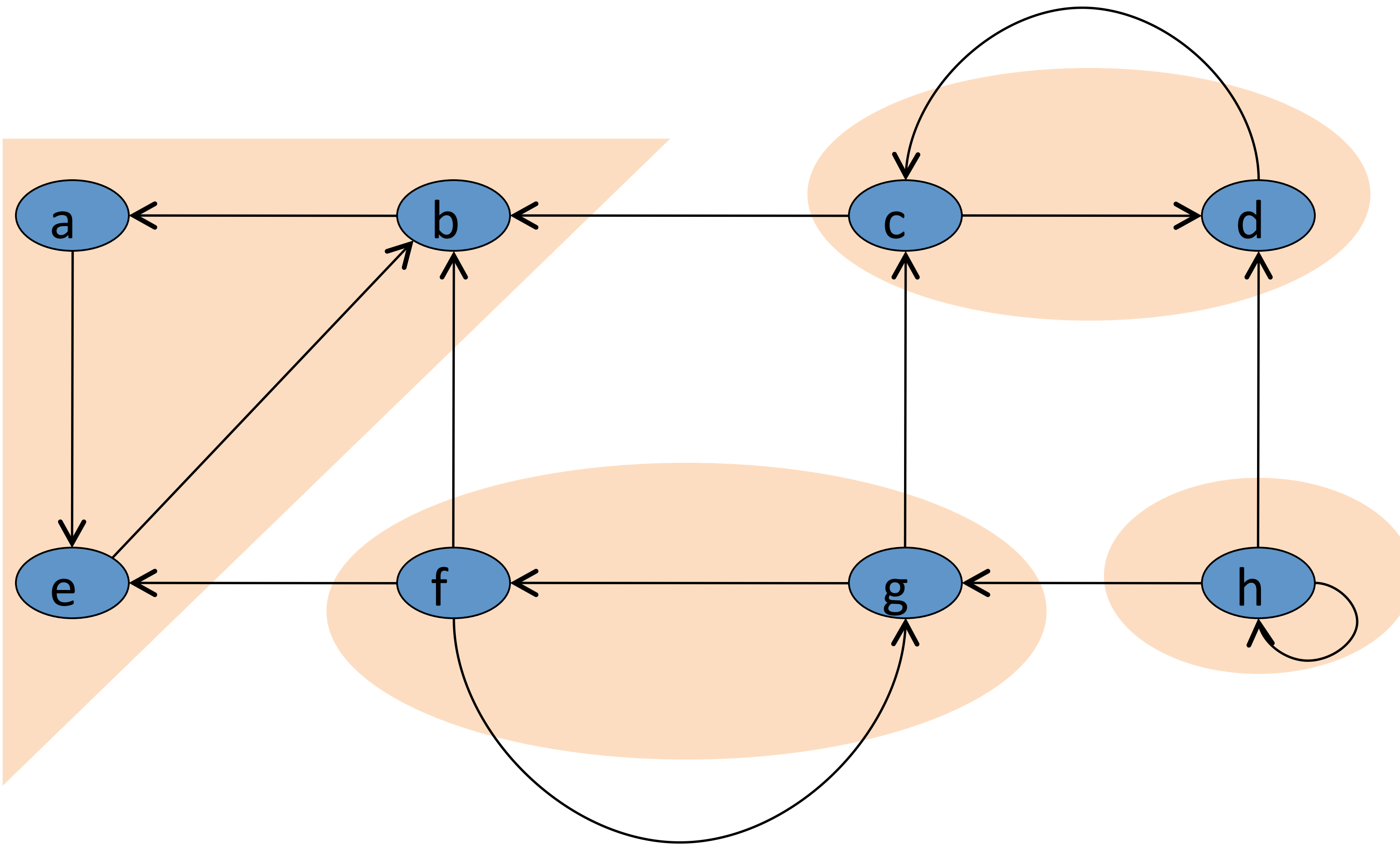
El algoritmo para determinar las SCC's de G usa el *grafo transpuesto* de G

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

… es decir, E^T consiste en las aristas de G con sus direcciones invertidas

G y G^{T} tienen exactamente las mismas componentes fuertemente conectadas

u y v son mutuamente alcanzables en G si y sólo si lo son en G^{T}


a
b
c
d
e
f
g
h

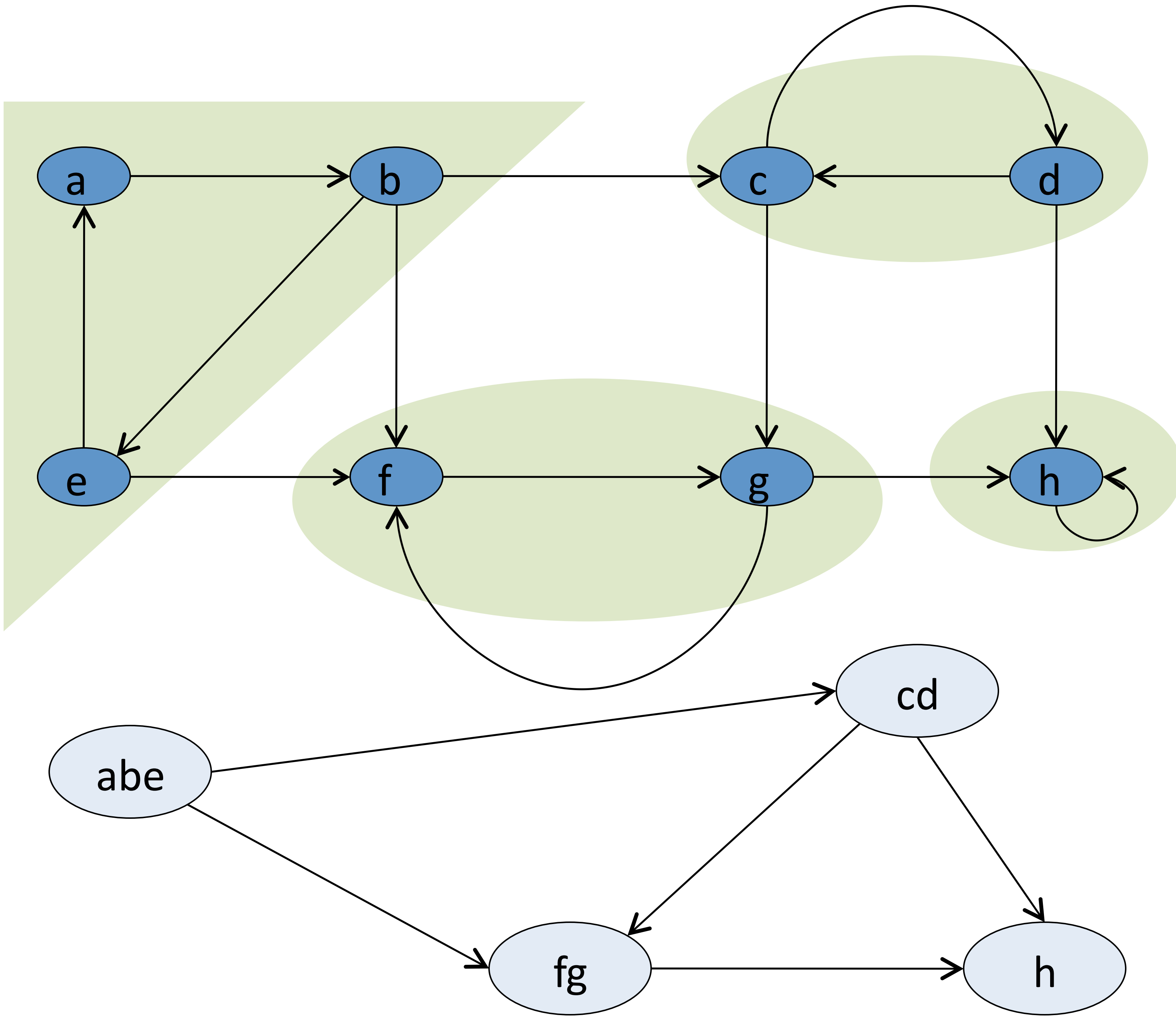
Definamos el *grafo de componentes* de G, $G^{\text{SCC}} = (V^{\text{SCC}}, E^{\text{SCC}})$

Supongamos que G tiene las componentes fuertemente conectadas $C_1, C_2, \ldots, C_k$

V^{SCC} es $\{v_1, v_2, \ldots, v_k\}$ y contiene un vértice v_i por cada componente fuertemente conectada C_i de G

Hay una arista $(v_i , v_j) \in E^{\text{SCC}}$ si G tiene una arista direccional (x, y) para algún $x \in C_i$ y algún $y \in C_j$

La propiedad clave es que G^{SCC} es un **grafo direccional acíclico** (DAG)

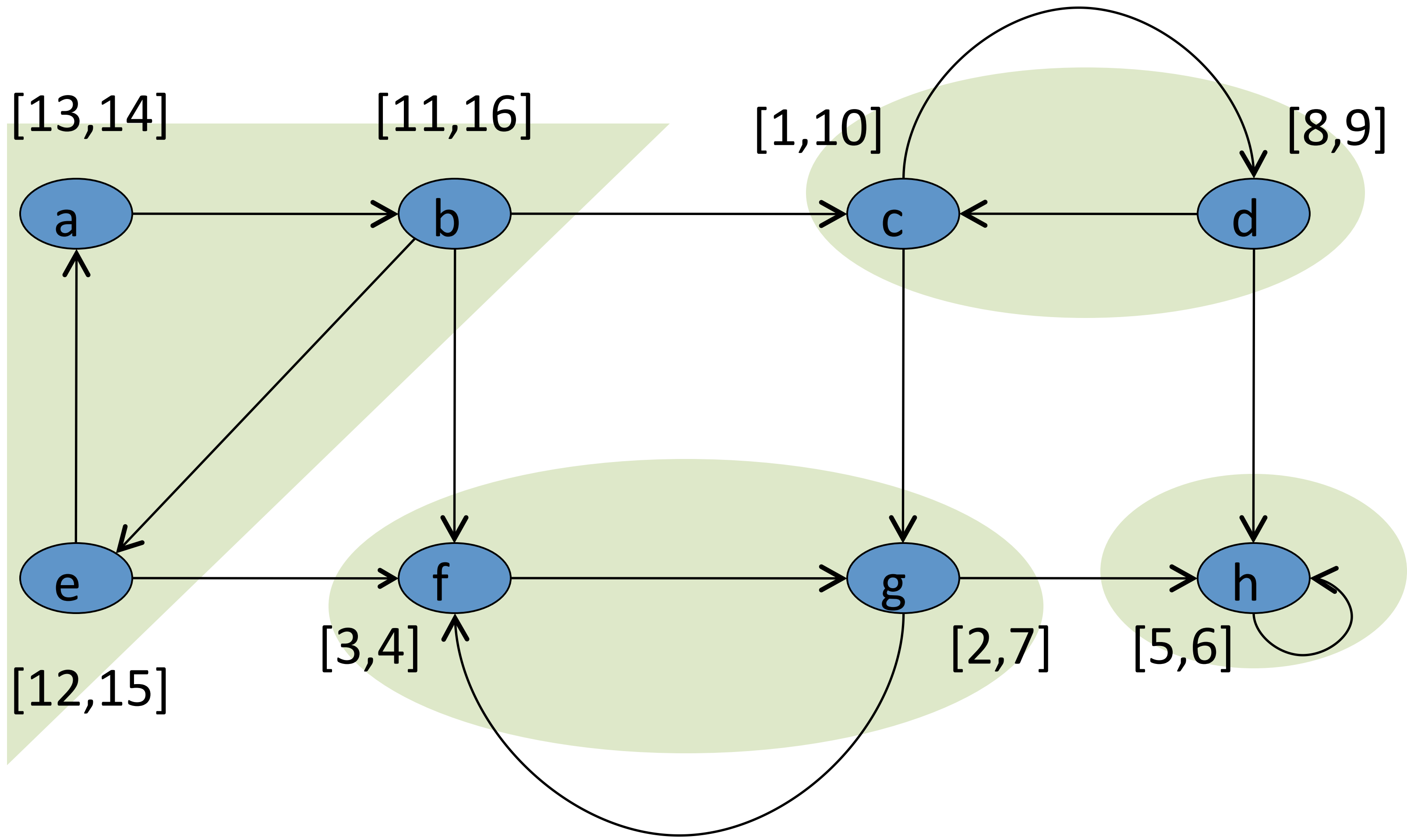

a
b
c
d
e
f
g
h
cd
abe
fg
h

Hagamos una exploración DFS de G

Sea $U \subseteq V$

Definimos $d(U) = \min_{u \in U}\{ u.d \}$ —el tiempo de descubrimiento más temprano de cualquier vértice en U

Definimos $f(U) = \max_{u \in U} \{ u.f \}$ —el tiempo de finalización más tardío de cualquier vértice en U

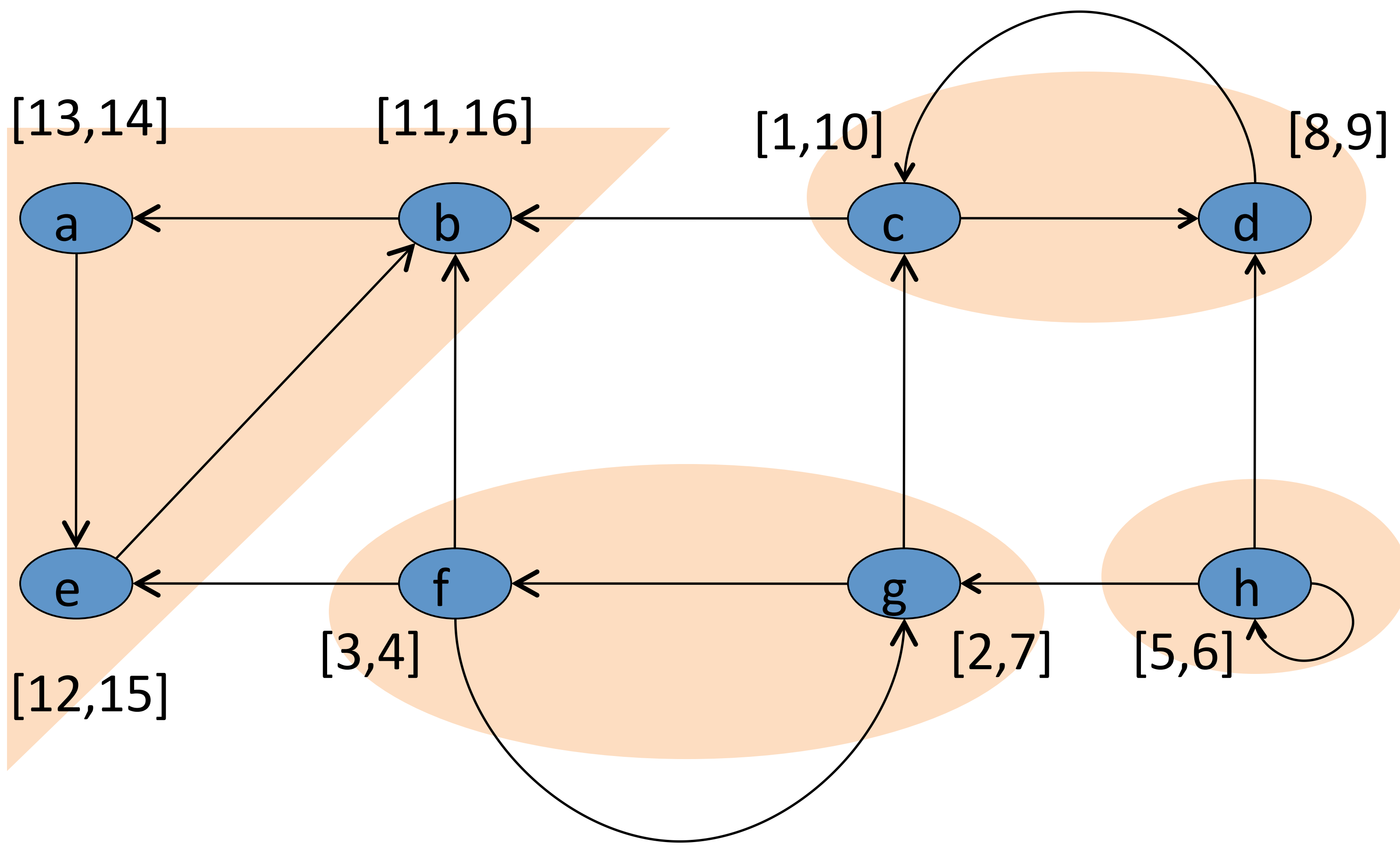

[13,14]
[11,16]
[1,10]
[8,9]
a
b
c
d
e
f
g
h
[3,4]
[2,7]
[5,6]
[12,15]

Una propiedad clave entre SCC's y tiempos de finalización

Sean C y D componentes fuertemente conectadas distintas de $G = (V, E)$:

- si hay una arista $(u, v) \in E$, en que $u \in C$ y $v \in D$, entonces $f(C) > f(D)$
- si hay una arista $(u, v) \in E^T$, en que $u \in C$ y $v \in D$, entonces $f(C) < f(D)$

Cada arista en G^T que va entre SCC's distintas va de una con un tiempo de finalización más temprano a otra con un tiempo de finalización más tardío


[13,14]
[11,16]
[1,10]
[8,9]
a
b
c
d
e
f
g
h
[3,4]
[2,7]
[5,6]
[12,15]

Hagamos ahora una exploración DFS de G^T

En el ciclo principal de `dfs()`, consideremos los vértices en orden decreciente de los *u.f* determinados en la exploración DFS de *G*:

- empezamos con la SCC *C* cuyo tiempo de finalización es máximo
- la exploración empieza en un vértice *x* de *C* y visita todos los vértices de *C*
- no hay aristas en G^T de *C* a ninguna otra SCC —el árbol con raíz *x* contiene exactamente los vértices de *C*

En resumen, el algoritmo para encontrar SCC's de un grafo *G* es el siguiente

realizamos DFS de *G*, para calcular los tiempos de finalización de cada vértice

determinamos G^T

realizamos DFS de G^T, pero en el ciclo principal consideramos los vértices en orden decreciente de *u.f*, calculado antes

los vértices de cada árbol en el bosque primero-en-profundidad recién formado son una SCC diferente