¿Se puede hacer el proyecto?

- Tenemos un proyecto complejo dividido en varias tareas
- Algunas tareas tienen como requisito otras tareas

¿Cómo sabemos si es posible realizar el proyecto completo?

Requisitos inconsistentes

Si la tarea B tiene como requisito la tarea A, escribimos

$$A \to B$$

Si existe alguna secuencia "circular" de requisitos:

$$X \to R \to \cdots \to K \to X,$$

… entonces no es posible realizar el proyecto

¿Es la única condición?

¿Cómo lo verificamos en el computador?

Si recibimos la lista de tareas y de requisitos

… ¿cómo hacemos un programa que revise esto?

¿Cuál será la forma más eficiente de hacerlo?

Grafos

Un **grafo *G*** es un conjunto de **nodos *V***

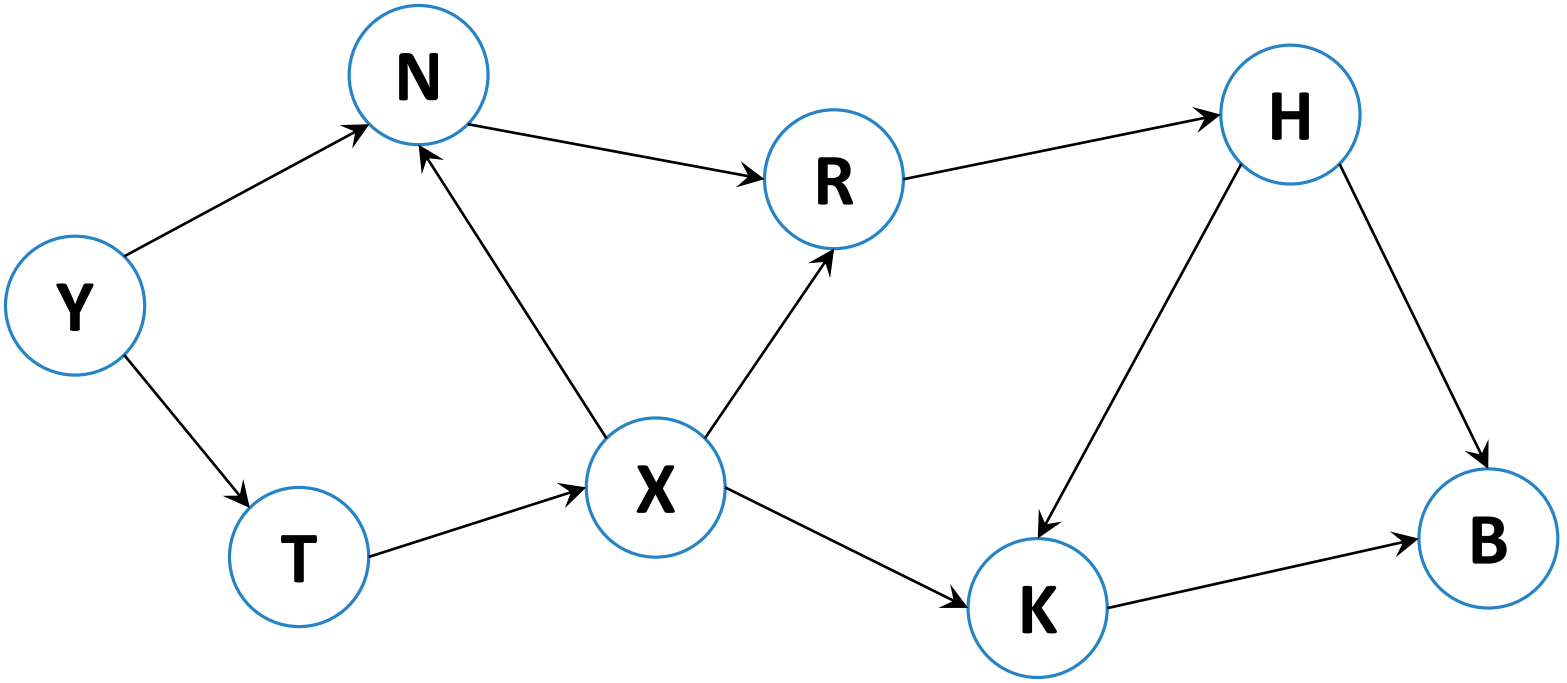
… y un conjunto de **aristas *E*** que unen pares de nodos

Es una forma de **representar** una situación de la vida real; p.ej.

- una red de computadores interconectados
- una red de tuberías para distribución de agua potable o gas

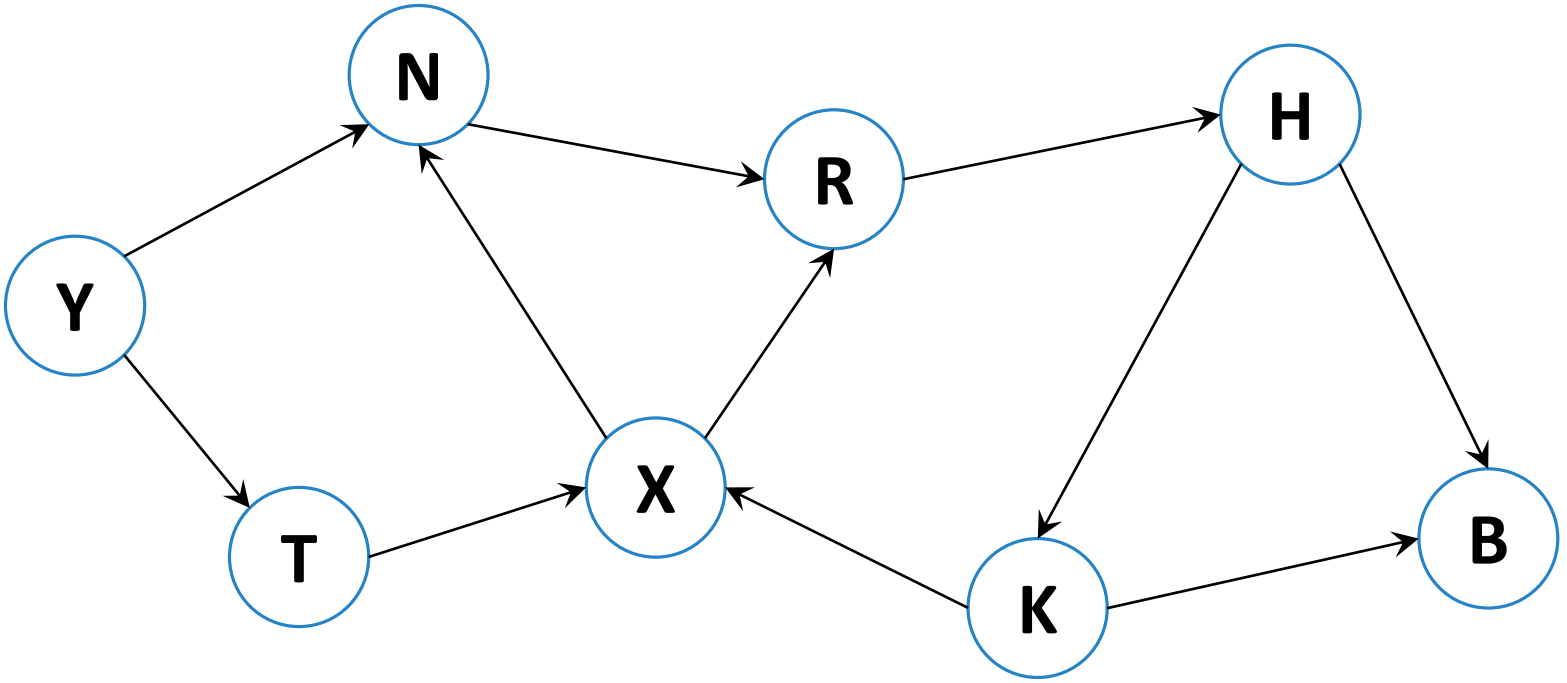
¿Cómo podríamos plantear el grafo de nuestro problema?

Dibujemos el grafo del proyecto



¿Algún problema con este proyecto?

¿Qué pasa en este caso?



¿Algún problema con este proyecto?

Ciclos

Si el grafo tiene un **ciclo**, entonces el proyecto (representado por el grafo) no puede llevarse a cabo

¿Cómo podemos buscar ciclos en un grafo de manera eficiente?

La relación *es posterior a*

Diremos que una tarea Y **es posterior a** una tarea X si:

$$X \rightarrow Y$$

o

Existe una tarea Z tal que $X \rightarrow Z$, y Y **es posterior a** Z

Significa que X debe realizarse antes que Y

Tareas posteriores a una tarea

Si una tarea **es posterior a** sí misma, entonces forma parte de un **ciclo**

¿Cómo podemos identificar las tareas posteriores a una tarea?

Pensemos en la definición de la propiedad

$\boldsymbol{posteriores}(\boldsymbol{X})$:

$\boldsymbol{P} \leftarrow \emptyset$

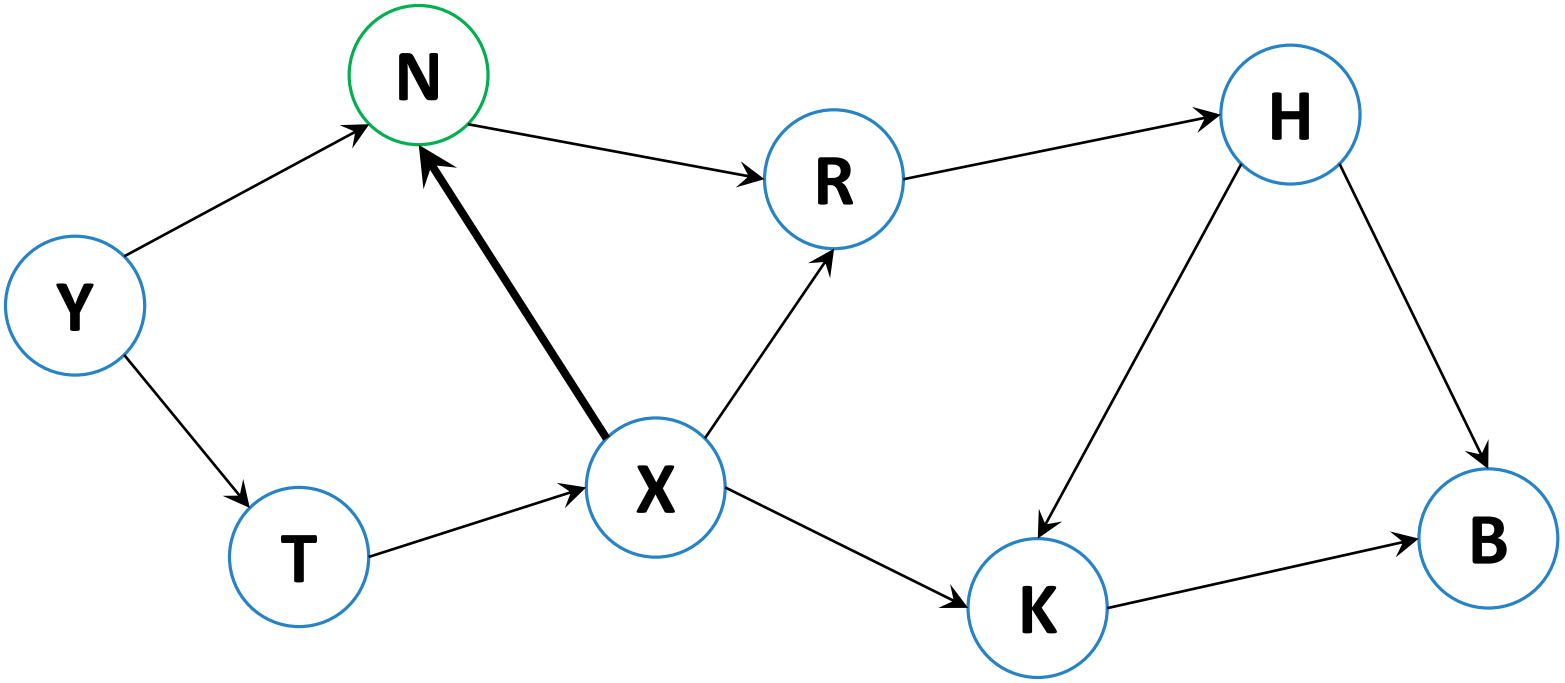
for $\boldsymbol{Y}$ tal que $\boldsymbol{X} \rightarrow \boldsymbol{Y}$:

$\boldsymbol{P} \leftarrow \boldsymbol{P} \cup \{\boldsymbol{Y}\}$

$\boldsymbol{P} \leftarrow \boldsymbol{P} \cup \boldsymbol{posteriores}(\boldsymbol{Y})$

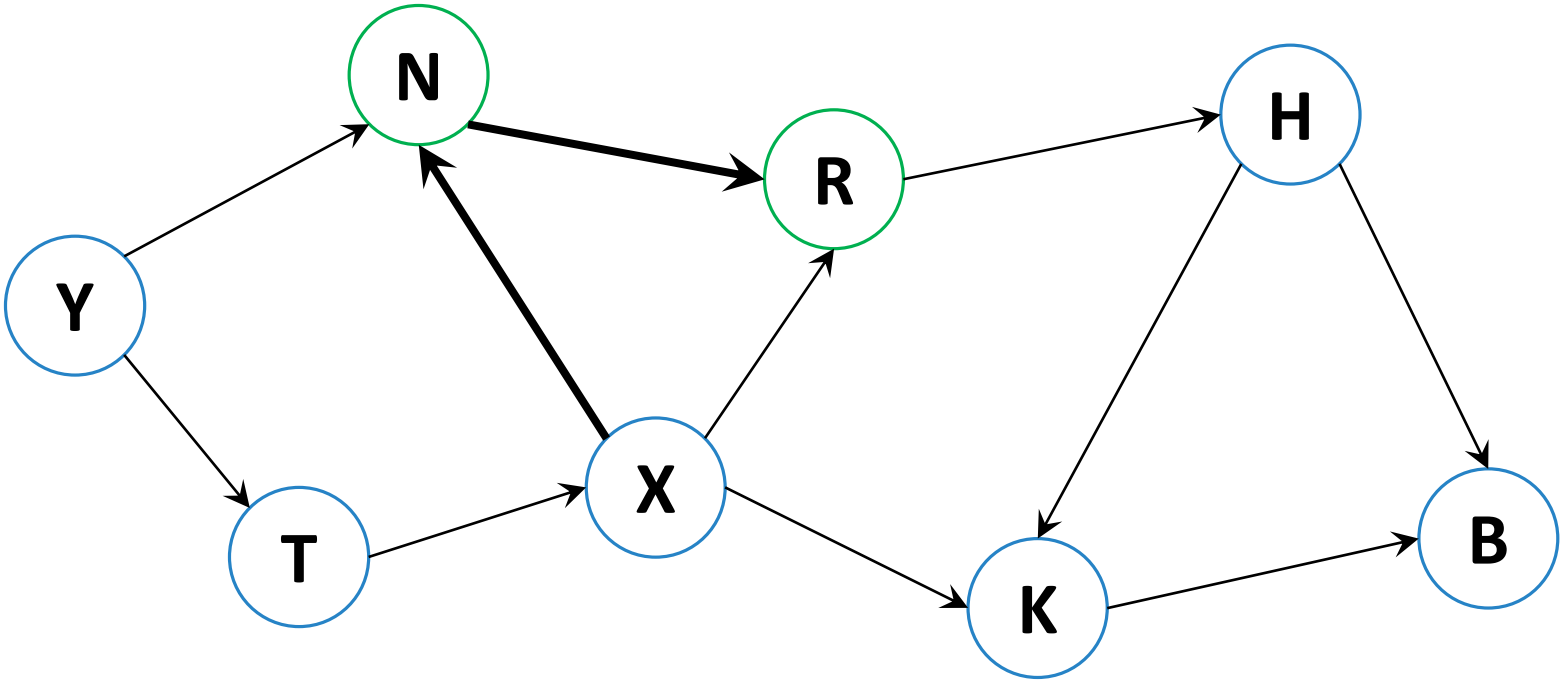
return $\boldsymbol{P}$

Tareas, o nodos, posteriores



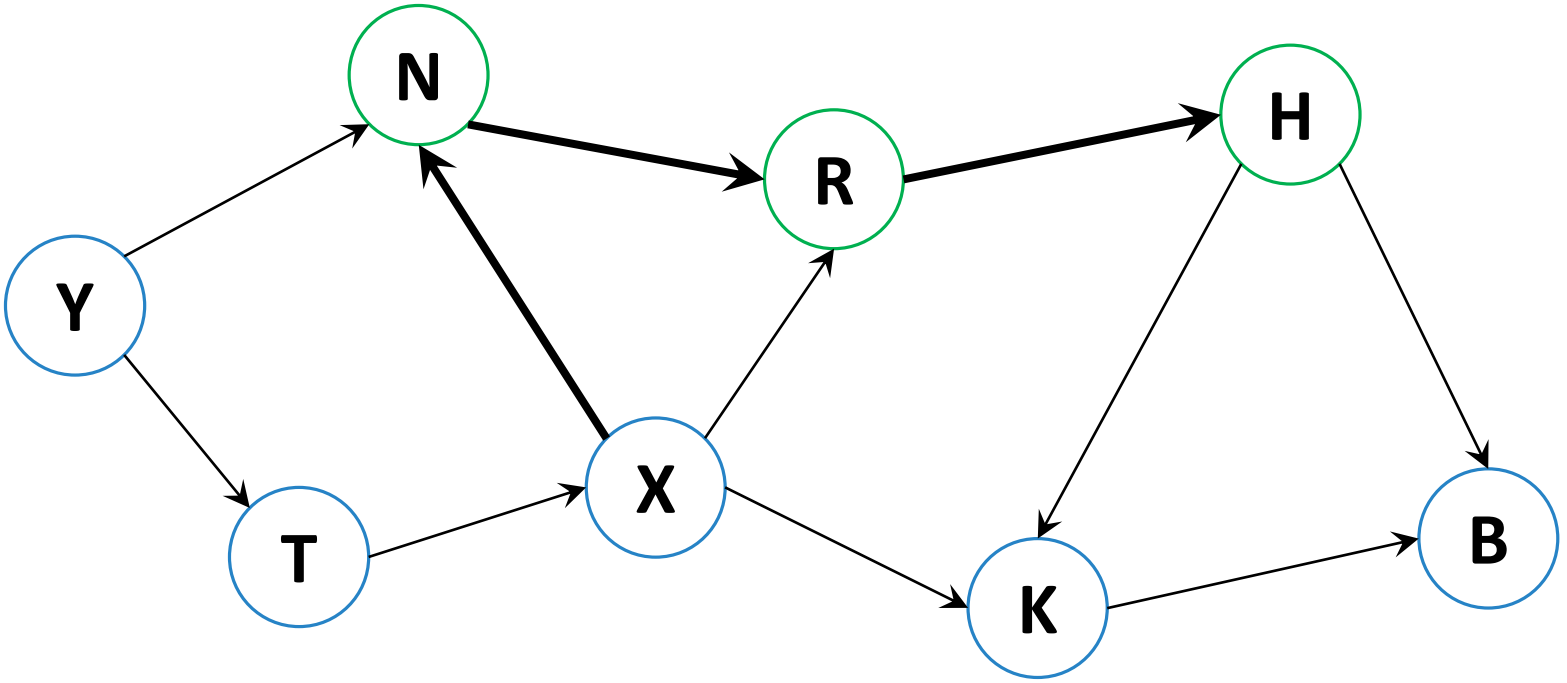
¿Cuáles nodos son posteriores a **X**?

Tareas, o nodos, posteriores



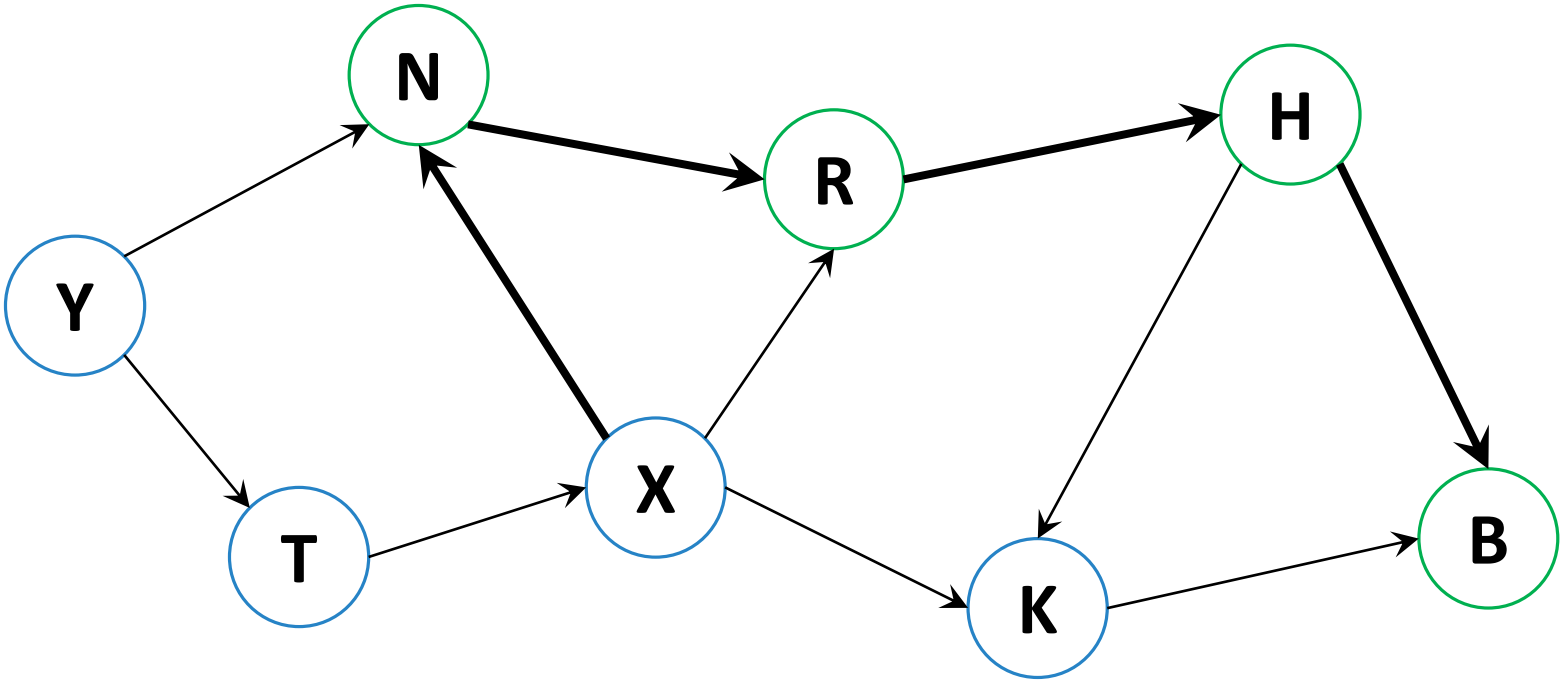
¿Cuáles nodos son posteriores a **N**?

Tareas, o nodos, posteriores



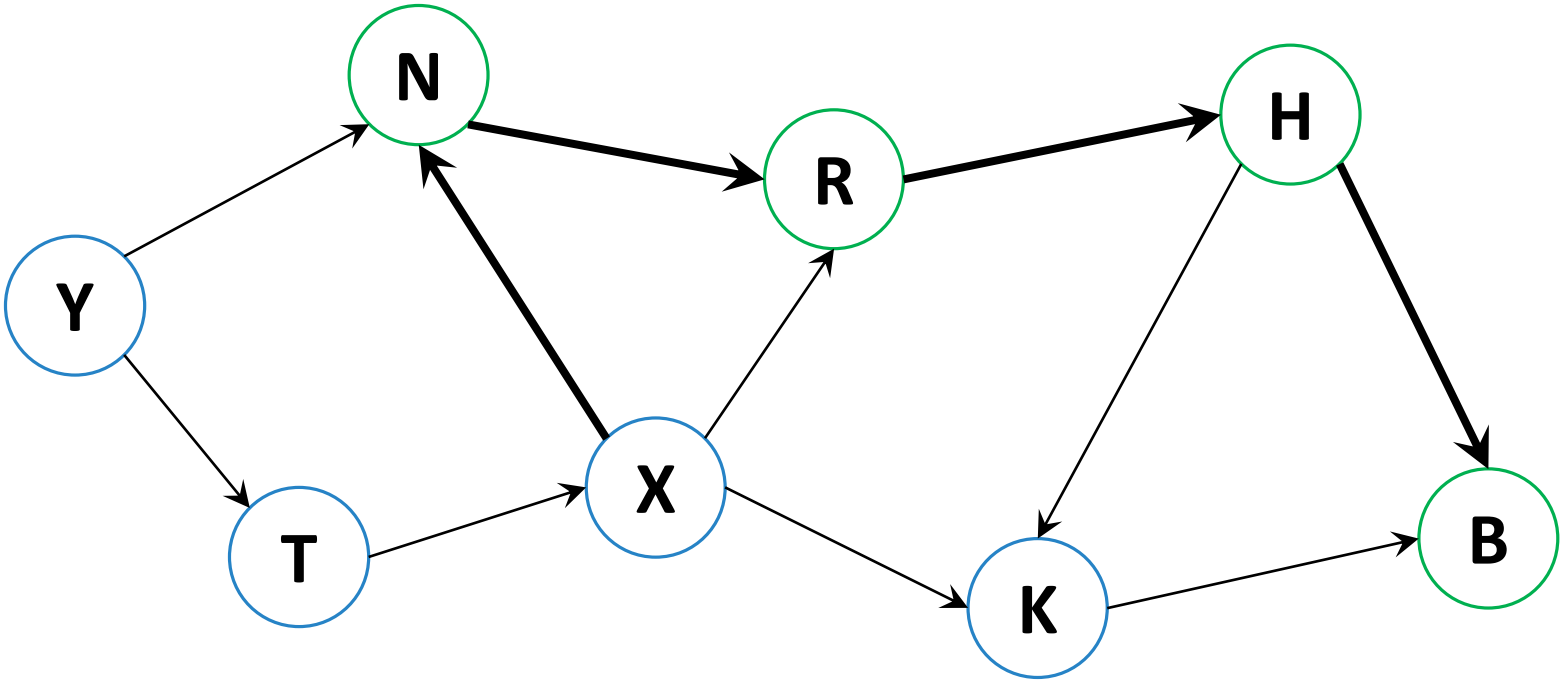
¿Cuáles nodos son posteriores a **R**?

Tareas, o nodos, posteriores



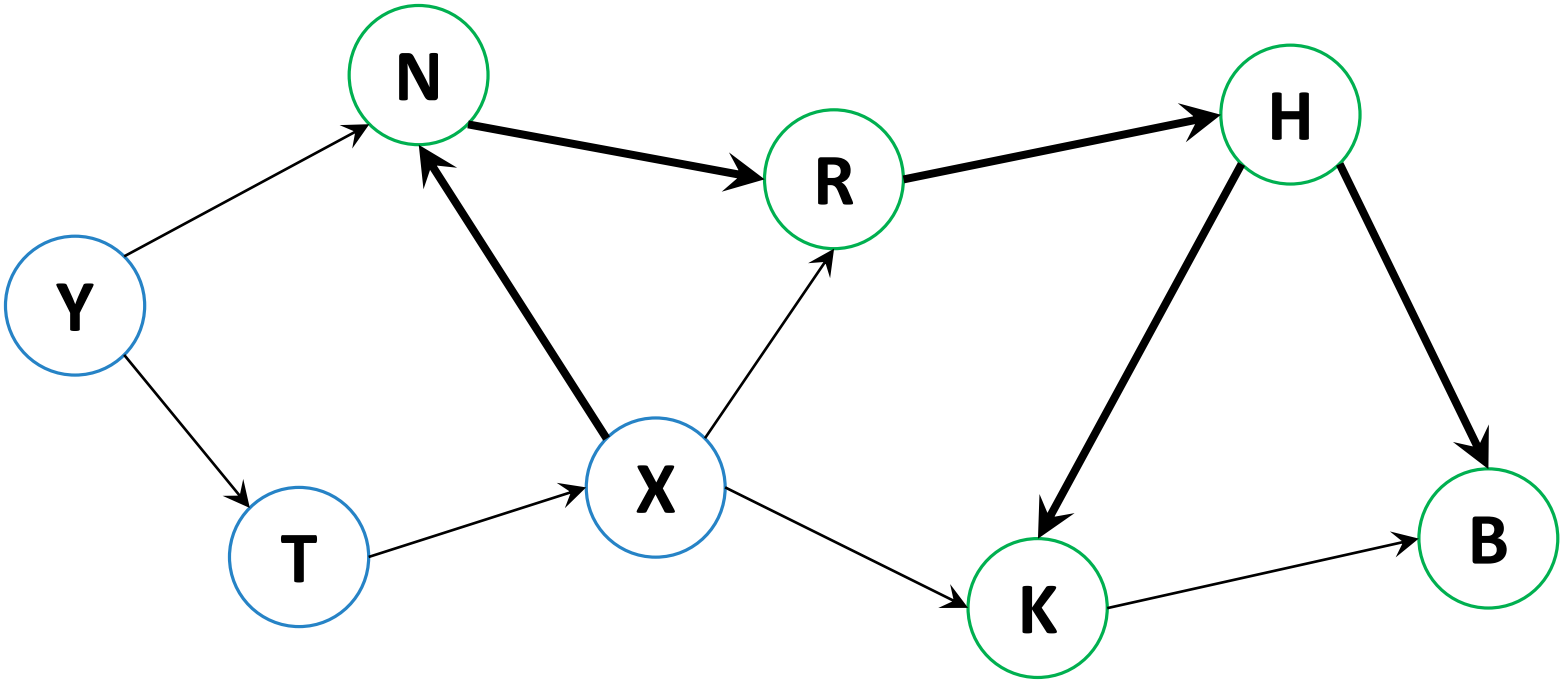
¿Cuáles nodos son posteriores a **H**?

Tareas, o nodos, posteriores



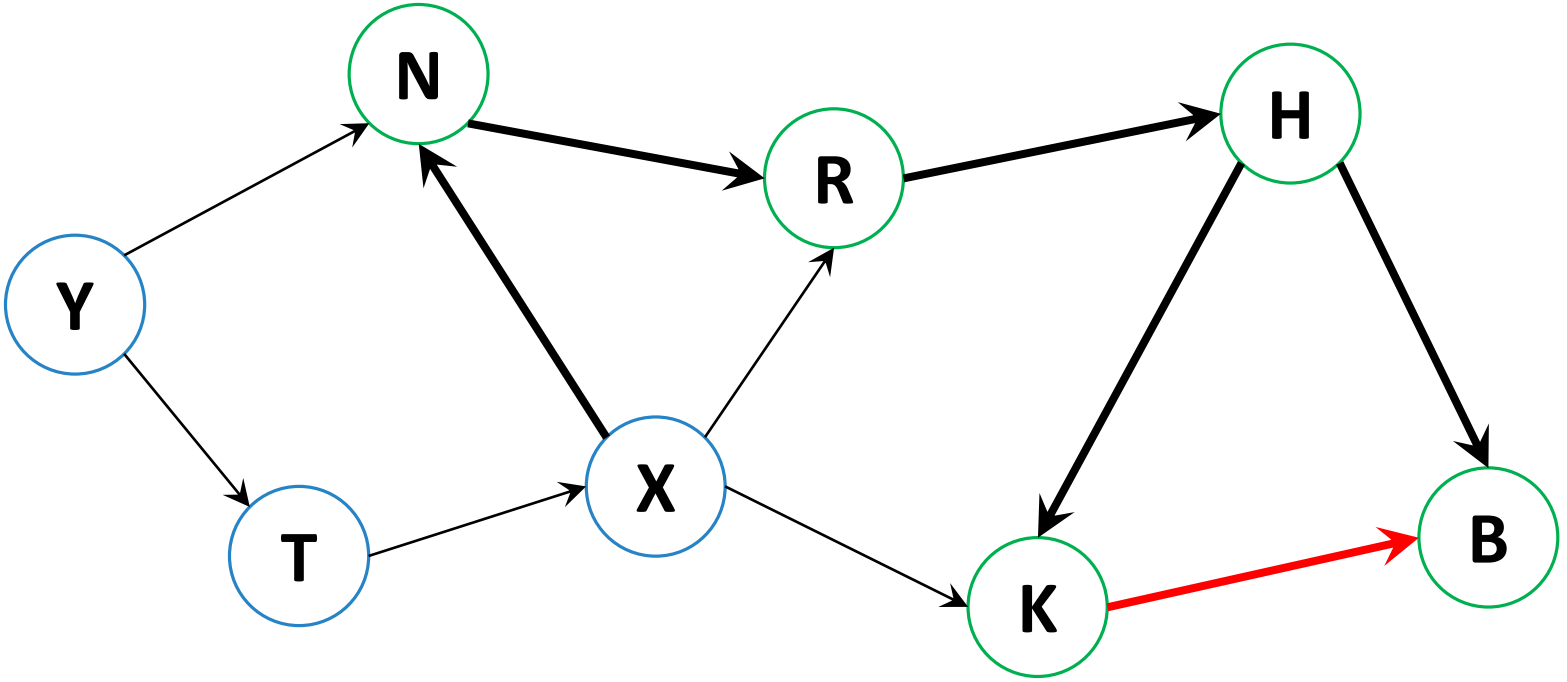
¿Cuáles nodos son posteriores a **B**?

Tareas, o nodos, posteriores



¿Cuáles nodos son posteriores a **K**?

Tareas, o nodos, posteriores



¿Cuáles nodos son posteriores a **B**? Espera…

Nodos por los que ya pasamos

Estamos haciendo llamadas repetidas

Es más, si pasamos por un ciclo, entonces el algoritmo no termina

¿Cómo se soluciona esto?

posteriores(X):

    if X está pintado: ***return*** $\emptyset$

    pintar X

    $P \leftarrow \emptyset$

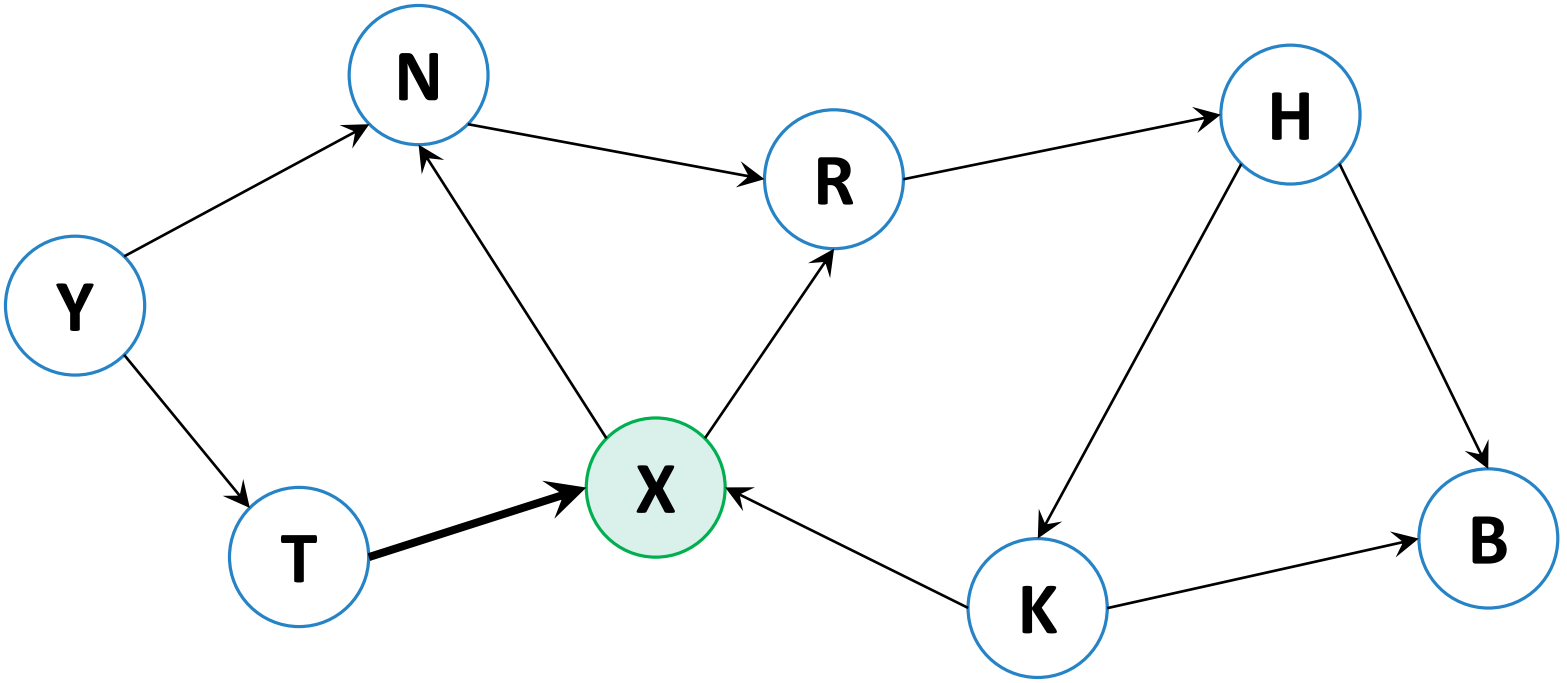
    for Y tal que $X \rightarrow Y$:

        $P \leftarrow P \cup \{Y\}$

        $P \leftarrow P \cup$ ***posteriores***(Y)

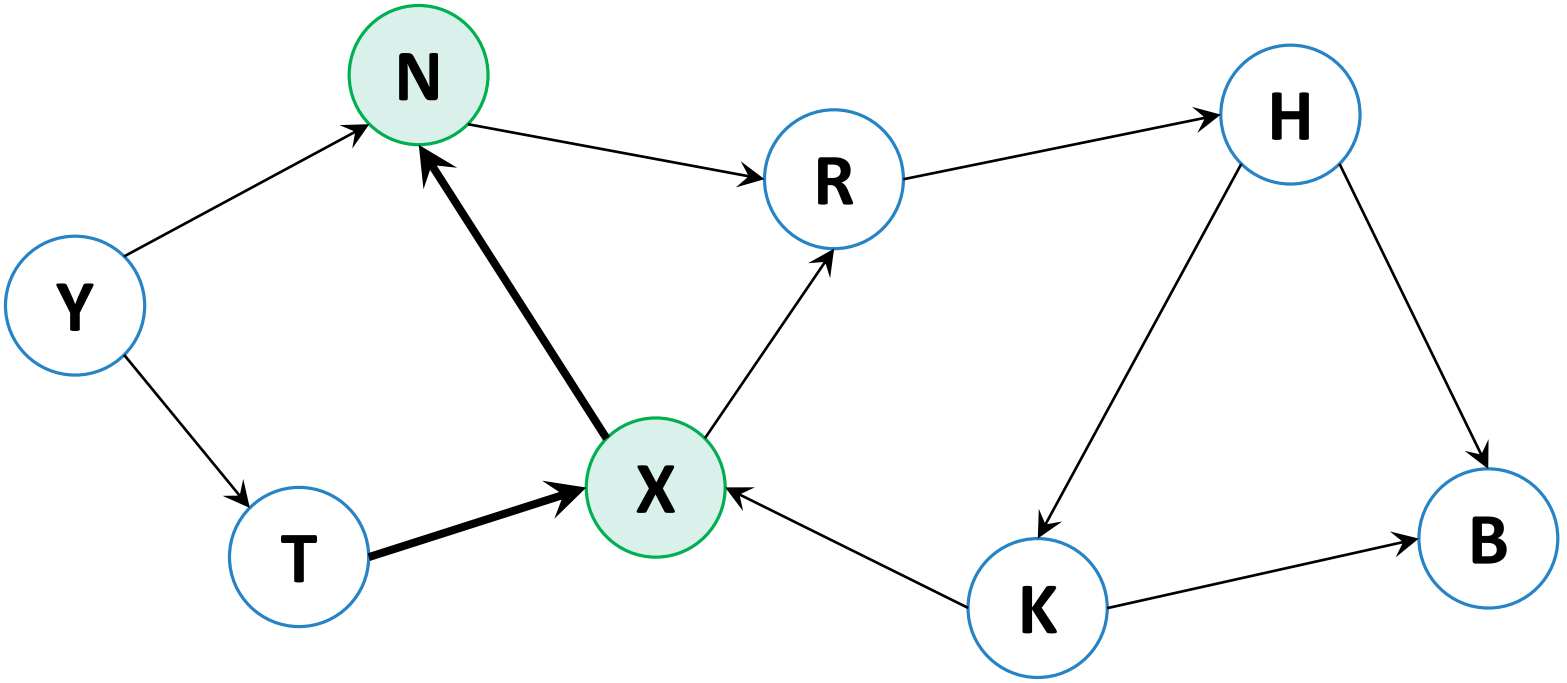
    return P

Nodos posteriores



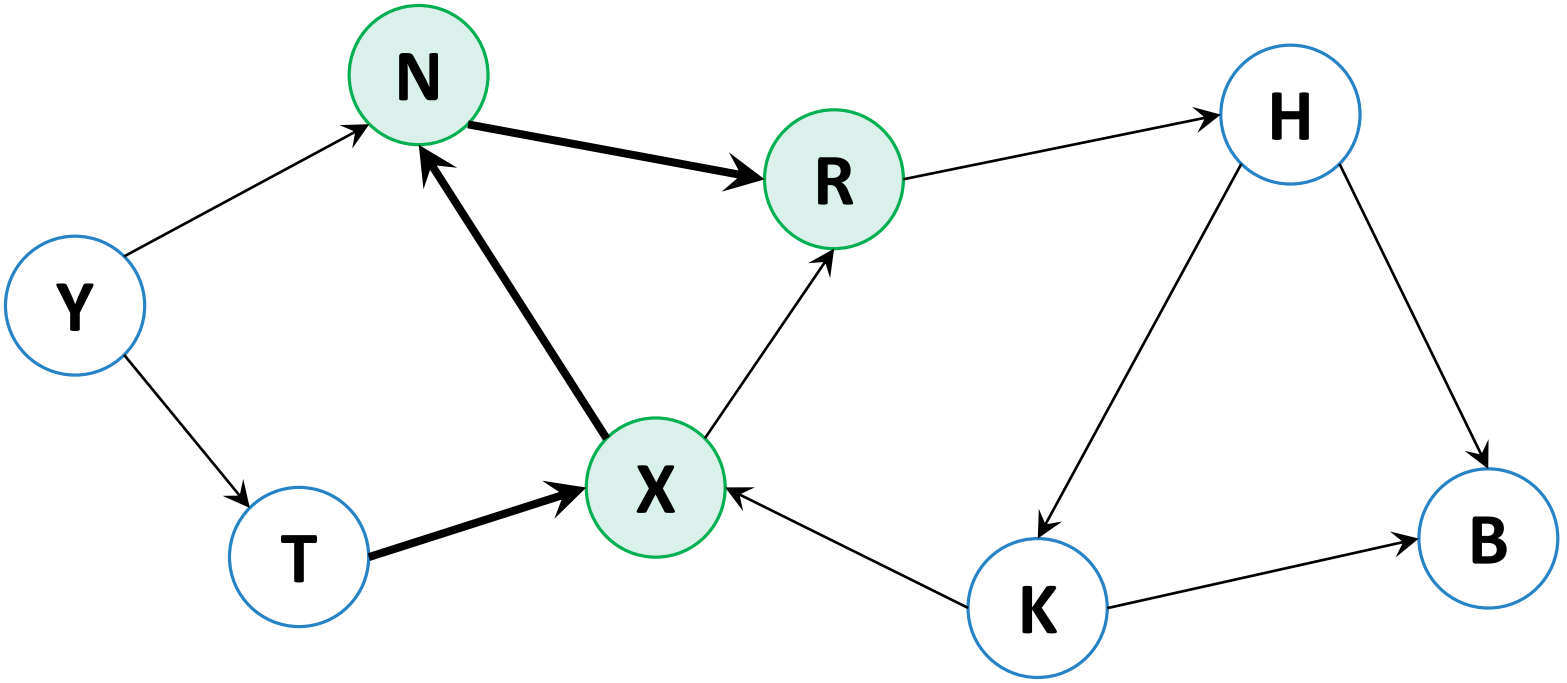
¿Cuáles nodos son posteriores a **T**?

Nodos posteriores



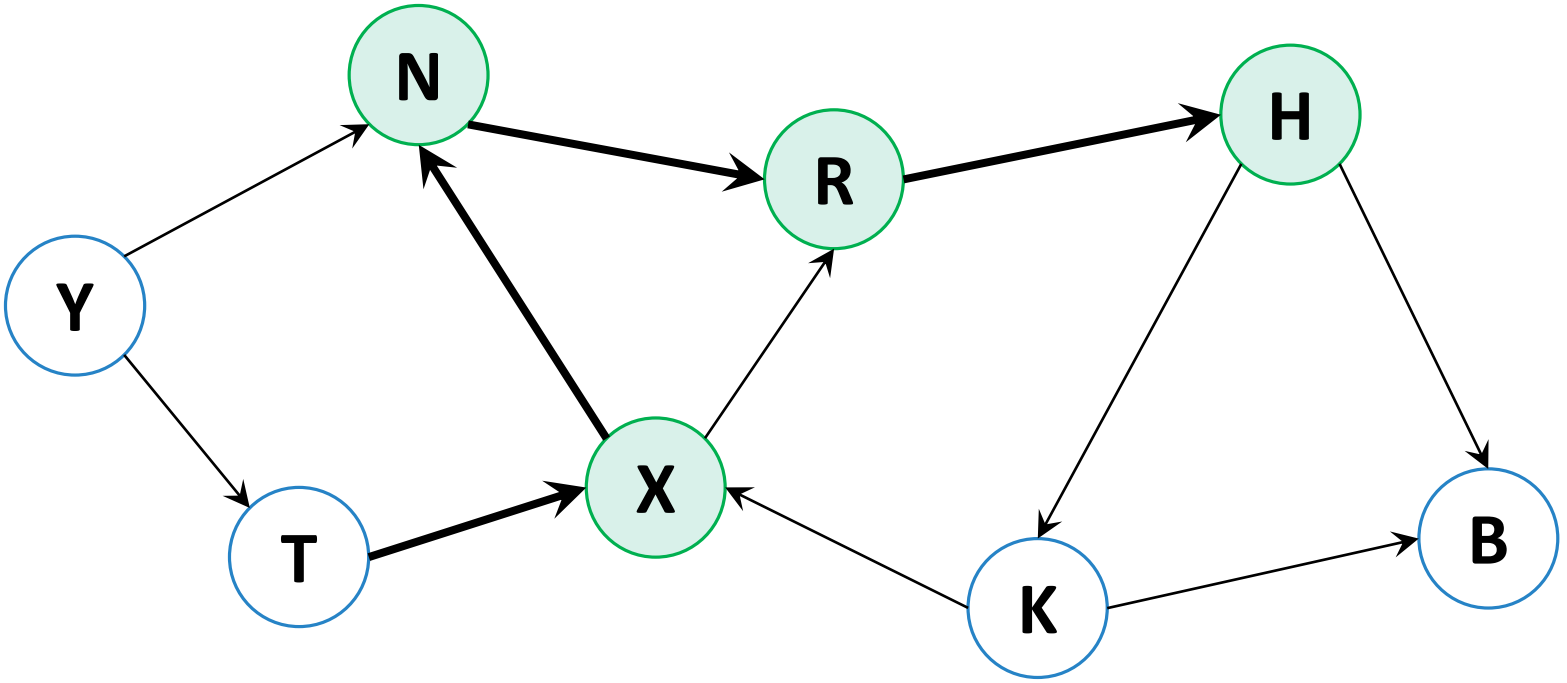
¿Cuáles nodos son posteriores a **X**?

Nodos posteriores



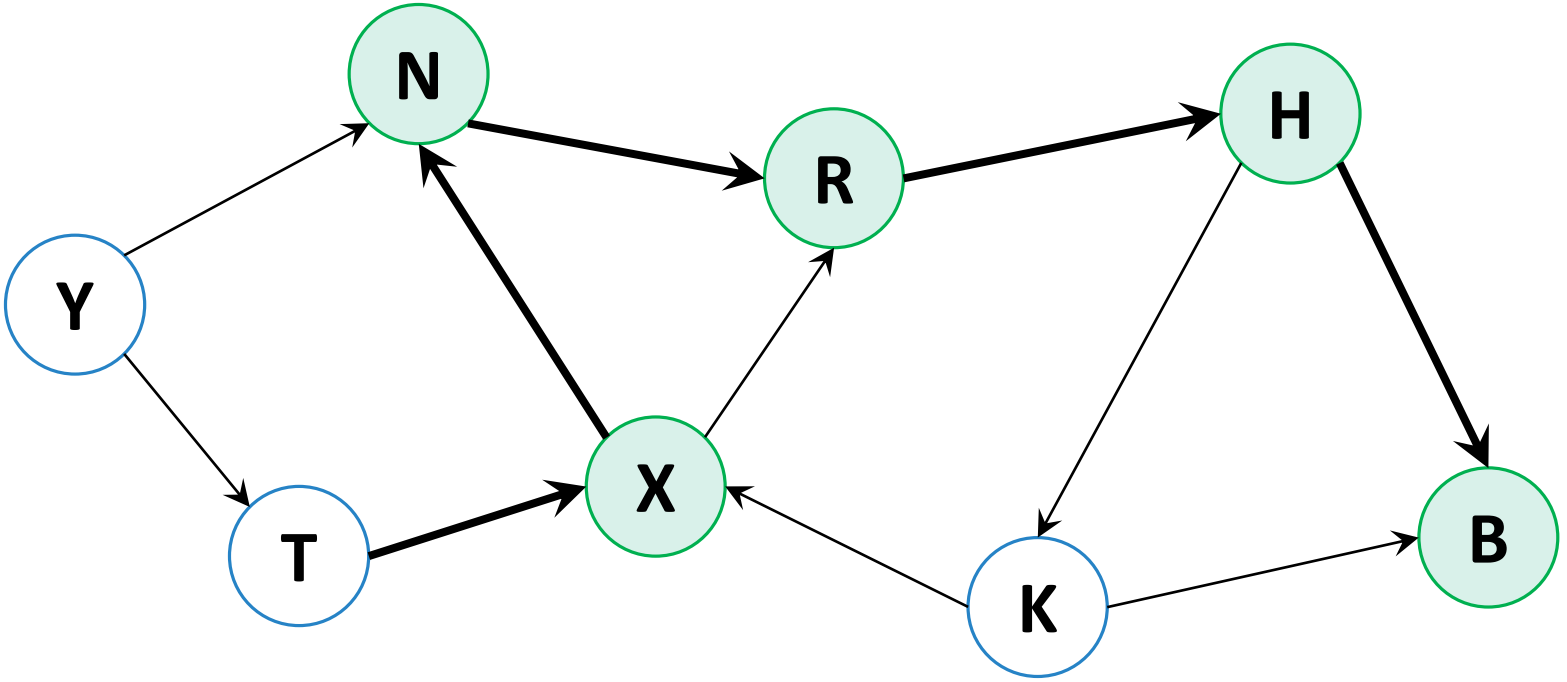
¿Cuáles nodos son posteriores a **N**?

Nodos posteriores



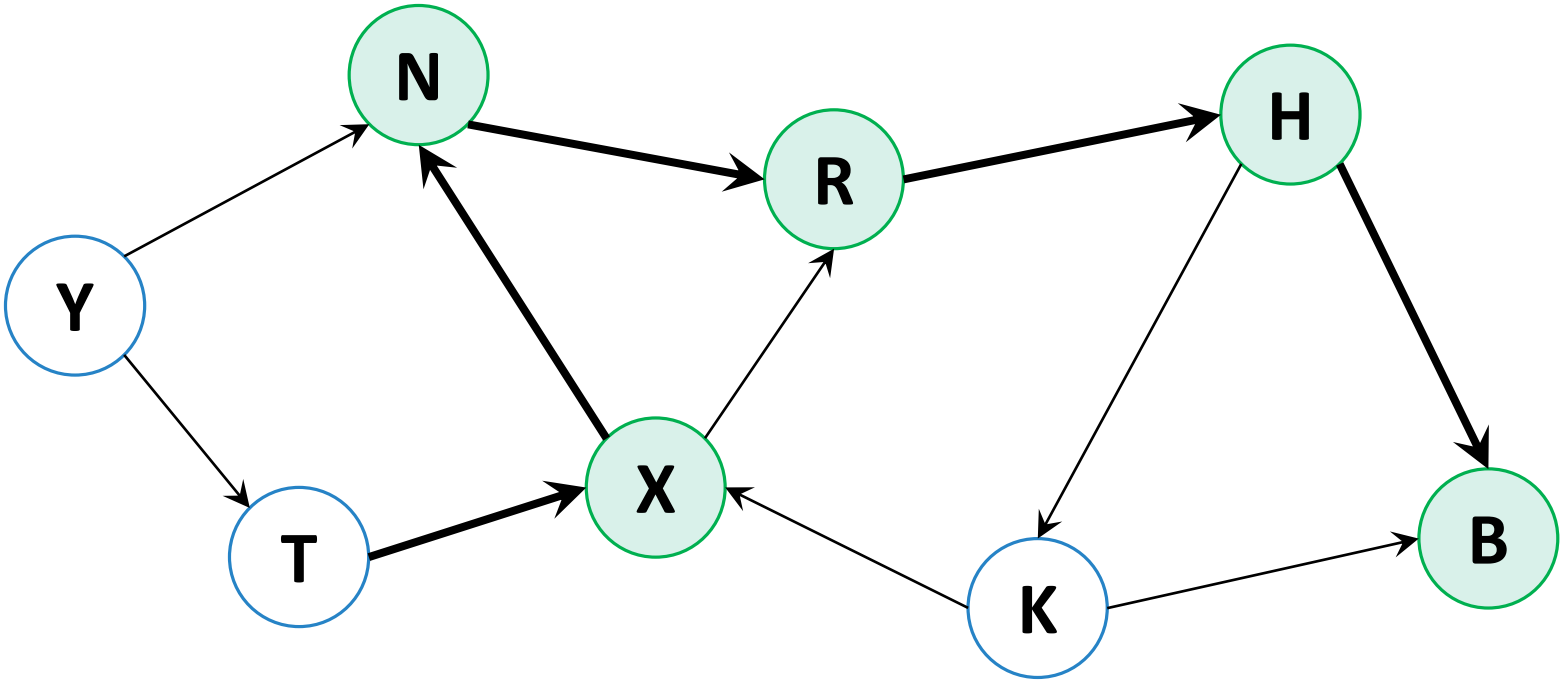
¿Cuáles nodos son posteriores a **R**?

Nodos posteriores



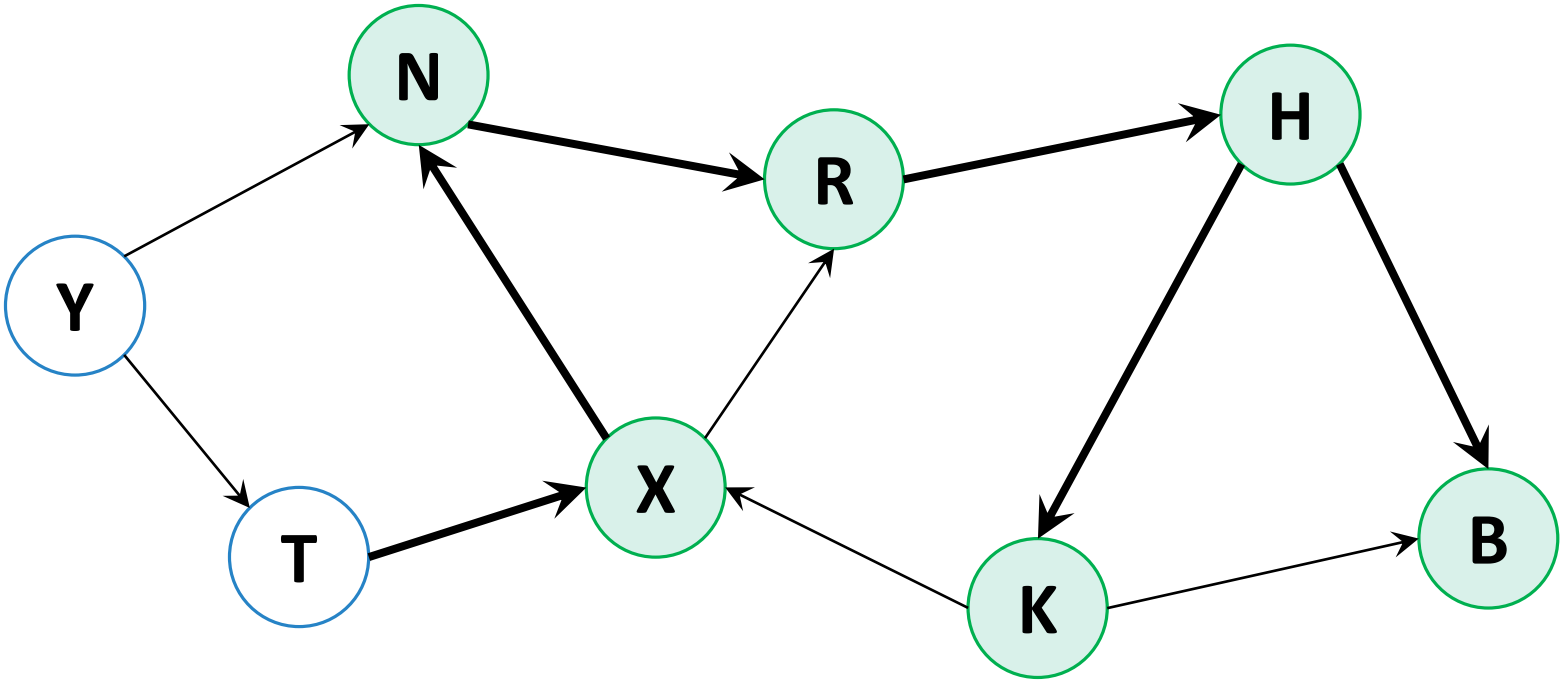
¿Cuáles nodos son posteriores a **H**?

Nodos posteriores



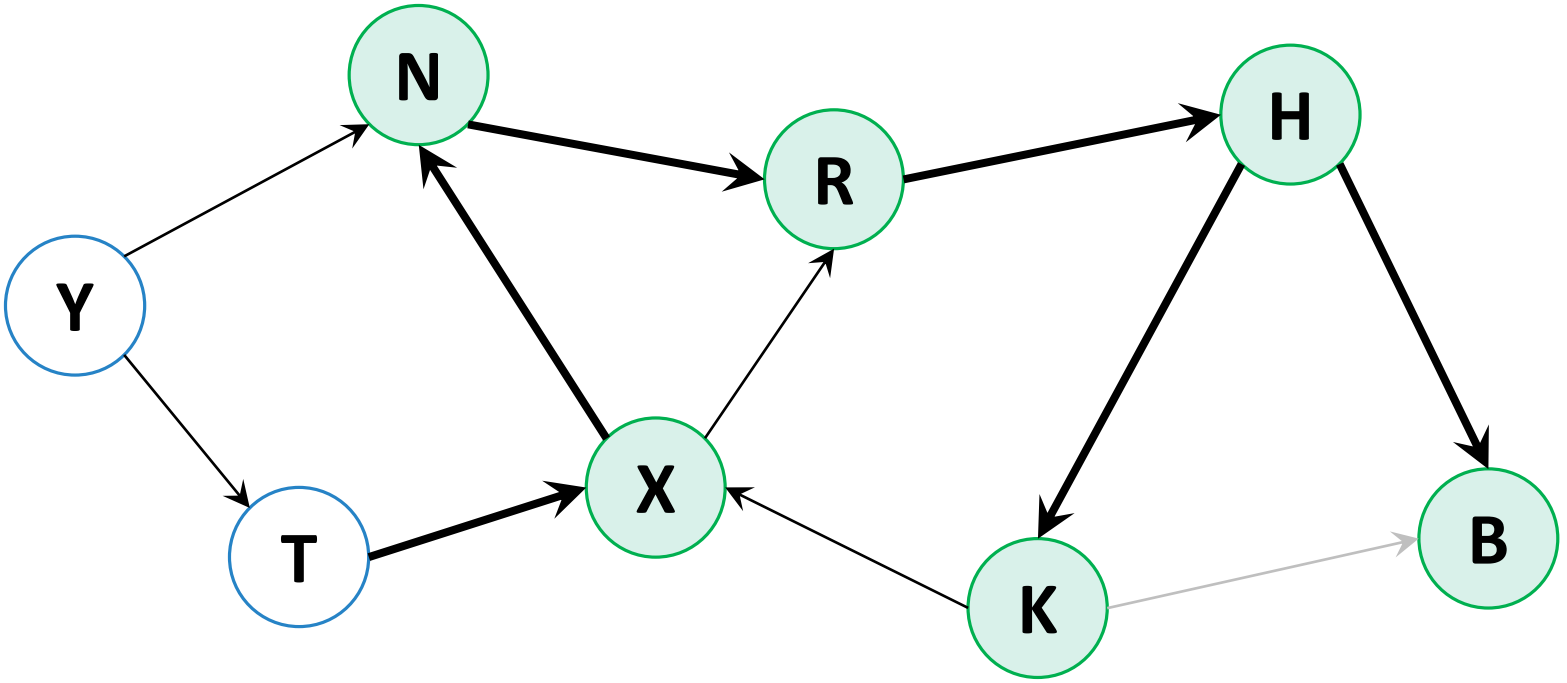
¿Cuáles nodos son posteriores a **B**?

Nodos posteriores



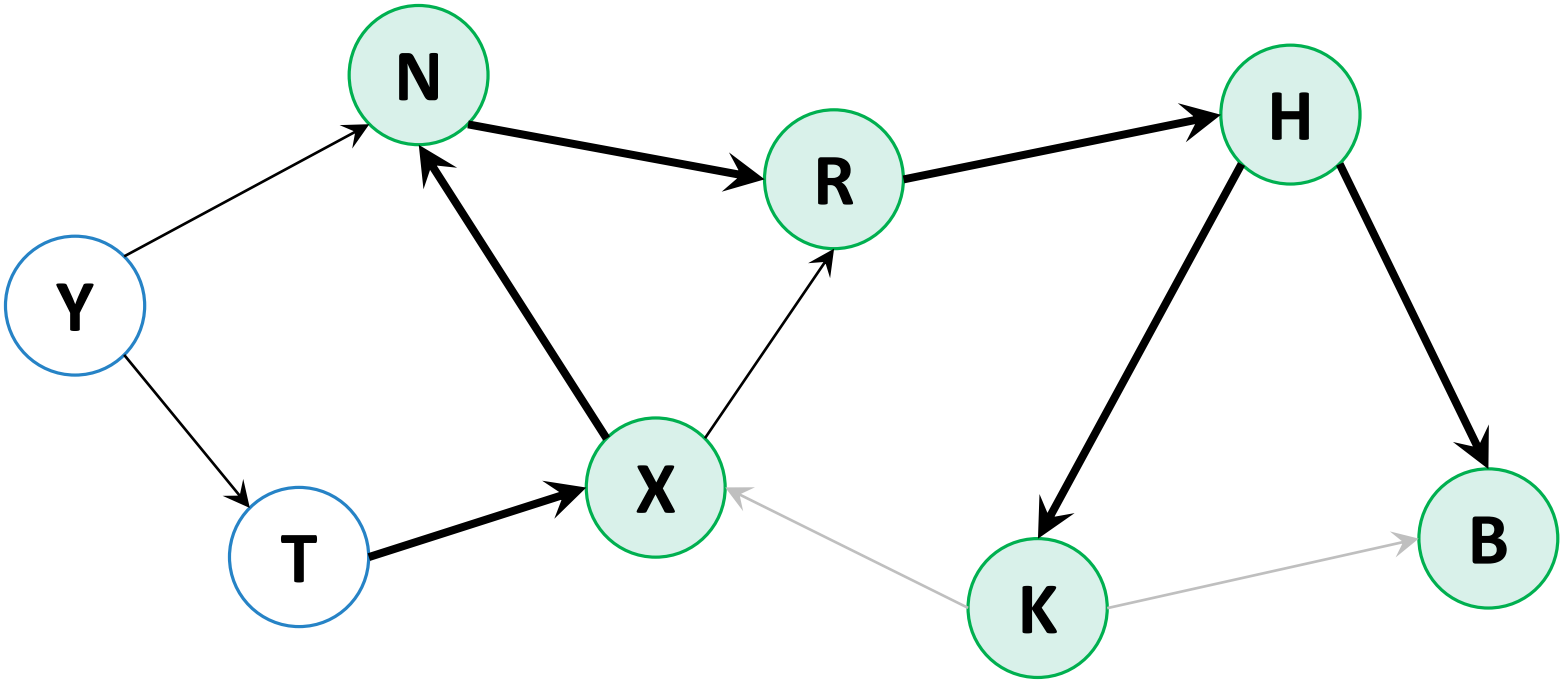
¿Cuáles nodos son posteriores a **H**?

Nodos posteriores



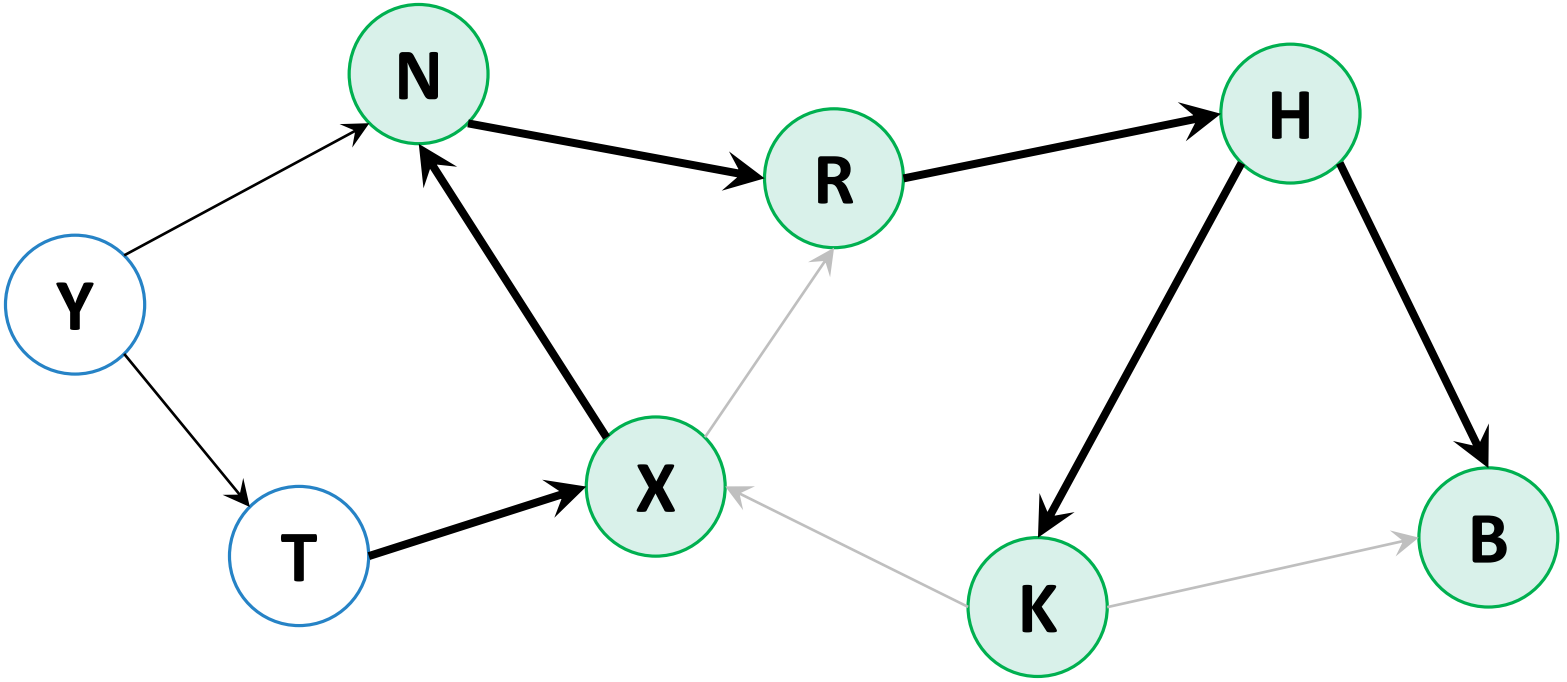
¿Cuáles nodos son posteriores a **K**?

Nodos posteriores



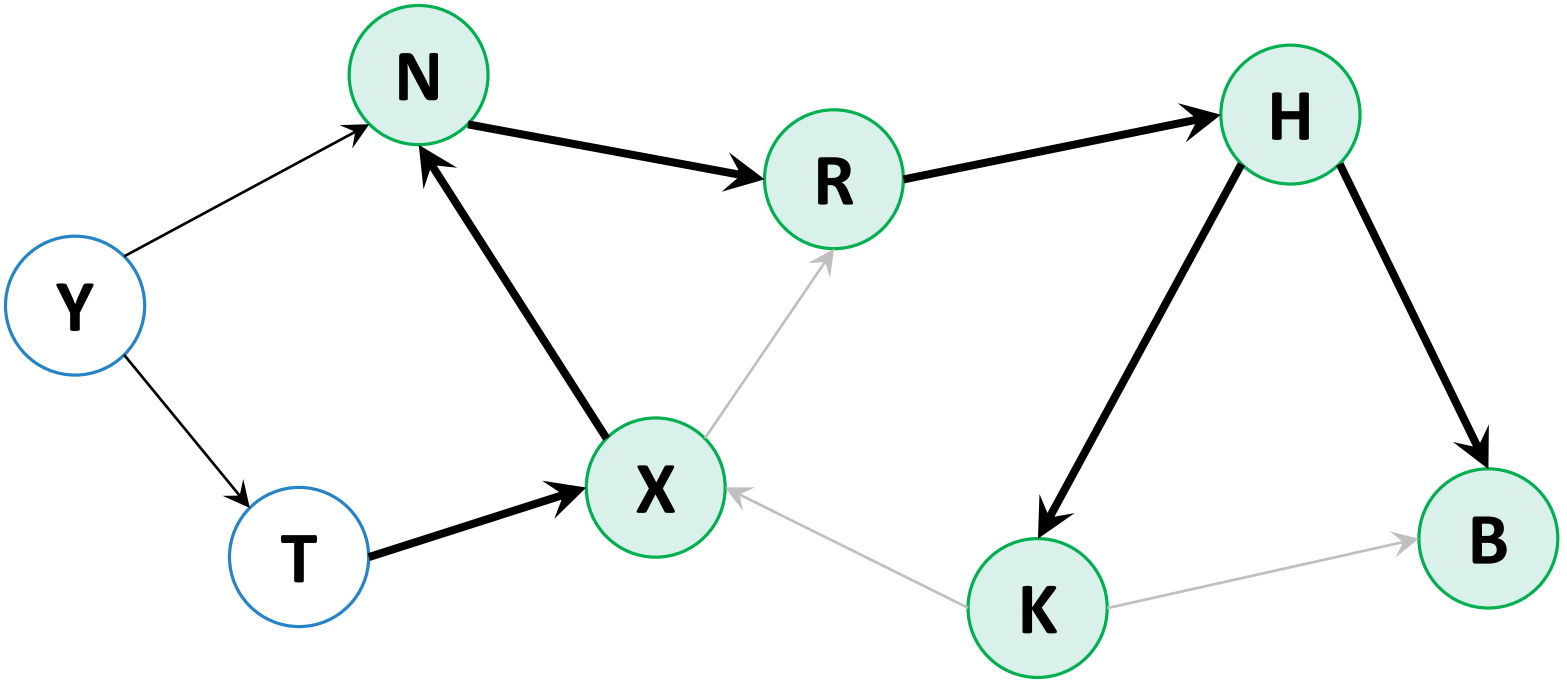
¿Cuáles nodos son posteriores a **K**?

Nodos posteriores



¿Cuáles nodos son posteriores a **X**?

Nodos posteriores



¡Listo!

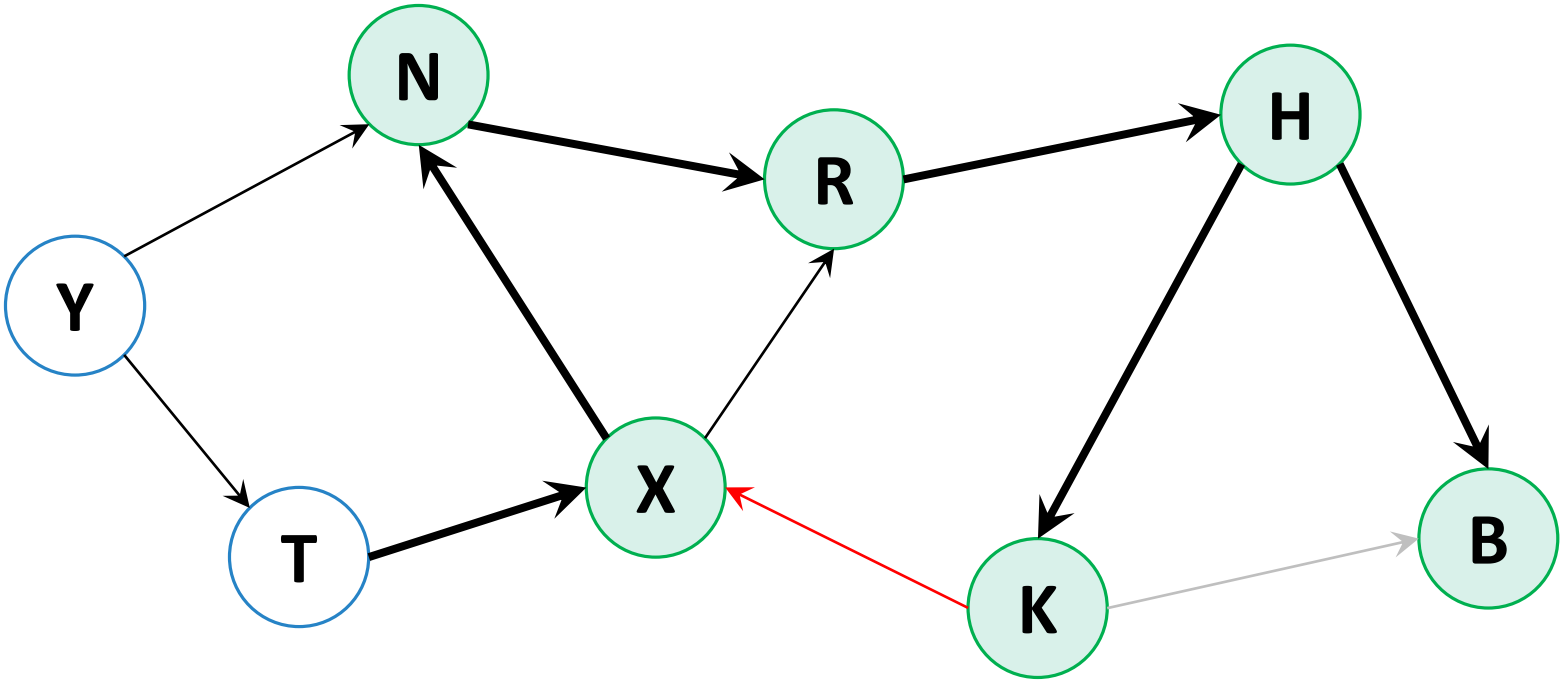
¿Cómo identificamos ciclos?

Ok, podemos identificar las tareas posteriores a una tarea

Ahora, viendo este algoritmo, ¿se nos ocurre algo?

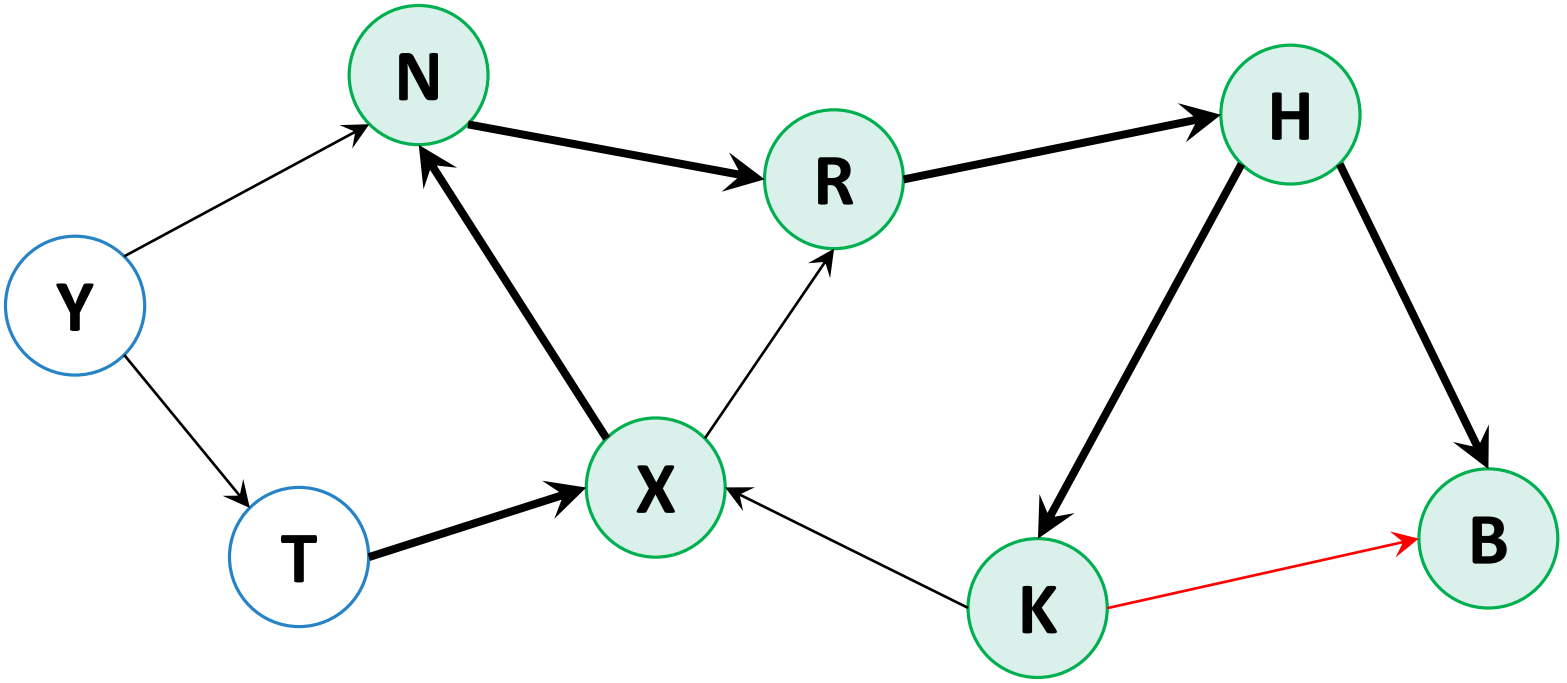
¿Podemos usar este enfoque para identificar ciclos?

Algo así como esto



¡Algoritmo, date cuenta de que esto es un ciclo!

… ¿y como esto?



¡Y que esto otro no!

Observación

Si el nodo recién descubierto, Y, está pintado, hay dos posibilidades:

- Si lo descubrió un nodo posterior a Y, hay ciclo
- Si lo descubrió un nodo anterior a Y, no hay ciclo (aún)

Hasta que $\boldsymbol{posteriores}(X)$ retorne, todos los nodos explorados son posteriores a X

hay ciclo luego de (X):

    if X está pintado de gris: ***return true***

    if X está pintado de negro: ***return false***

    Pintar X de gris

    for Y tal que $X \to Y$:

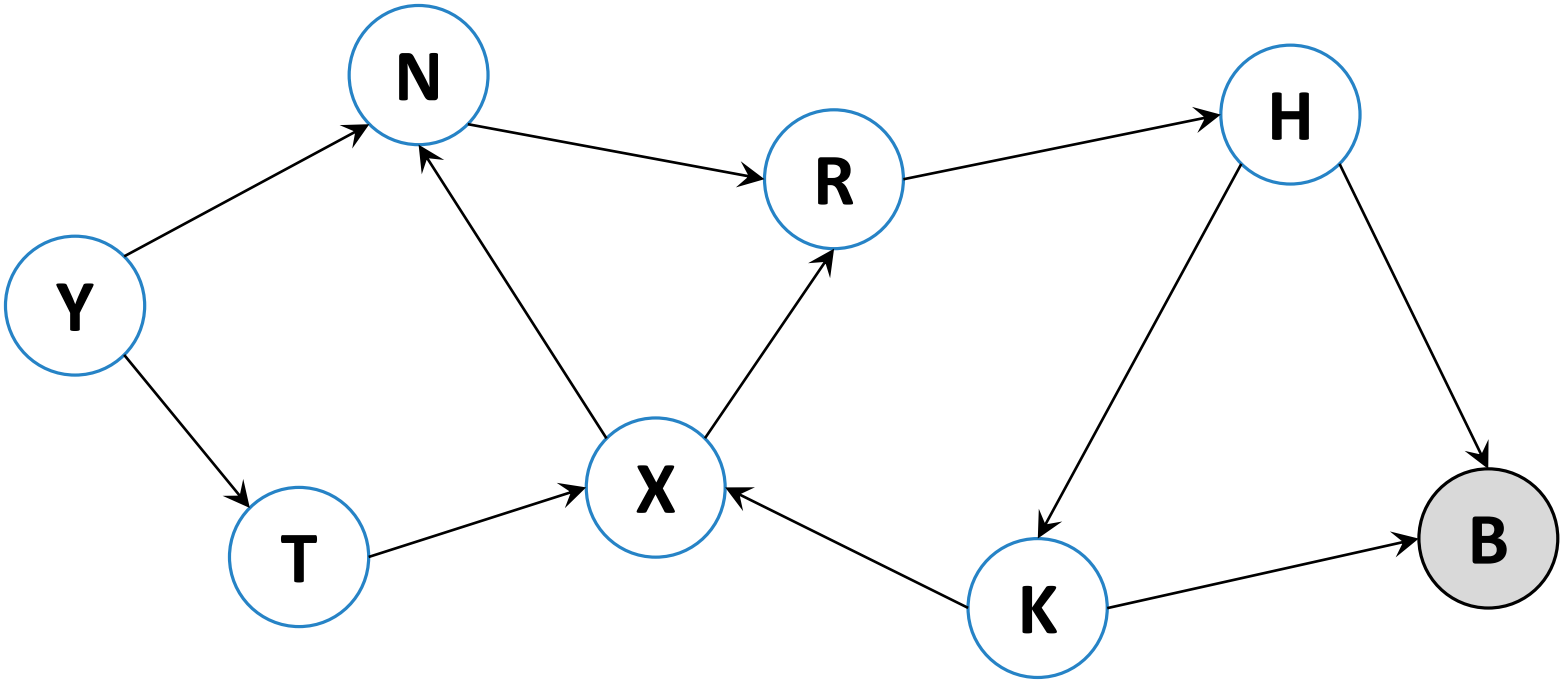
        if hay ciclo luego de (Y), ***return true***

    Pintar X de negro

    return false

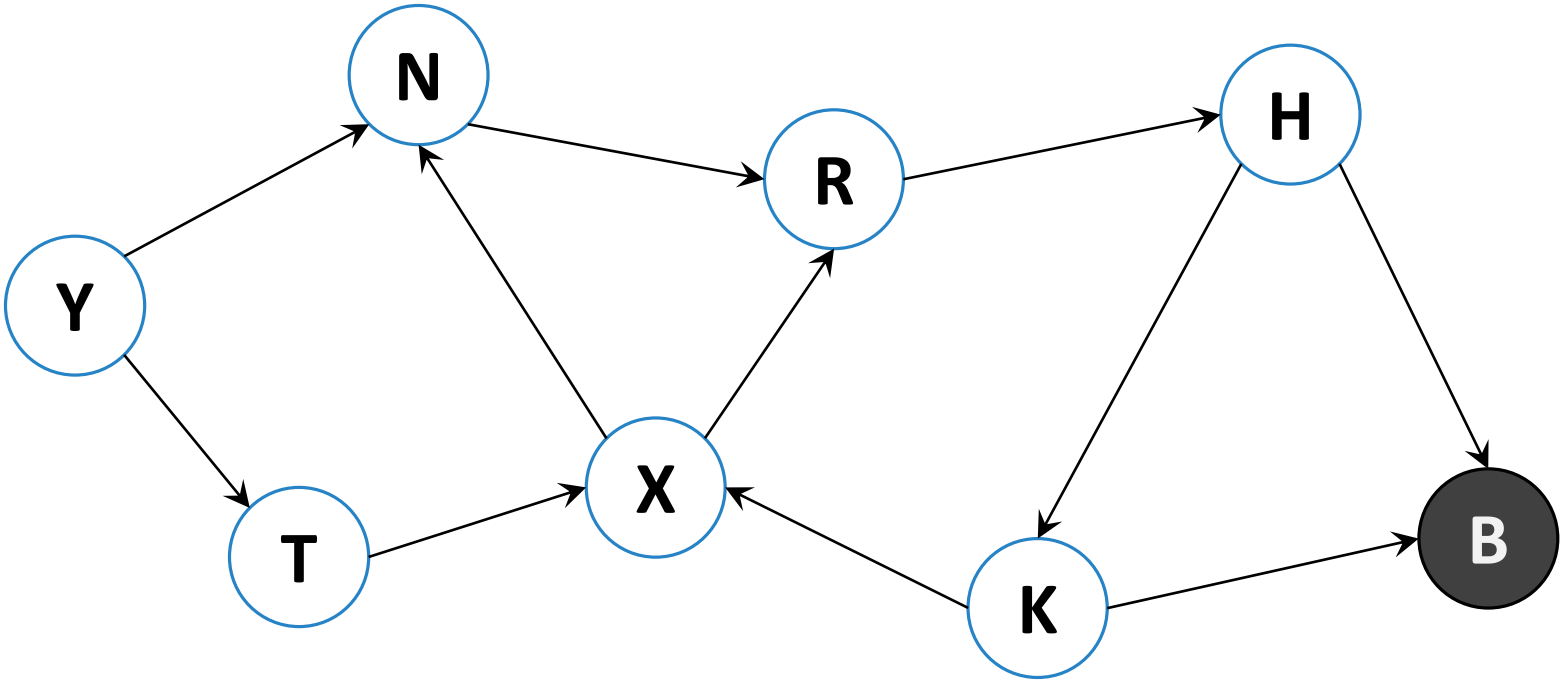
```
hay ciclo en(G(V, E)):
    for X ∈ V:
        if X está pintado, continue
        if hay ciclo luego de (X):
            return true
    return false
```

El algoritmo *hay ciclo en* en acción



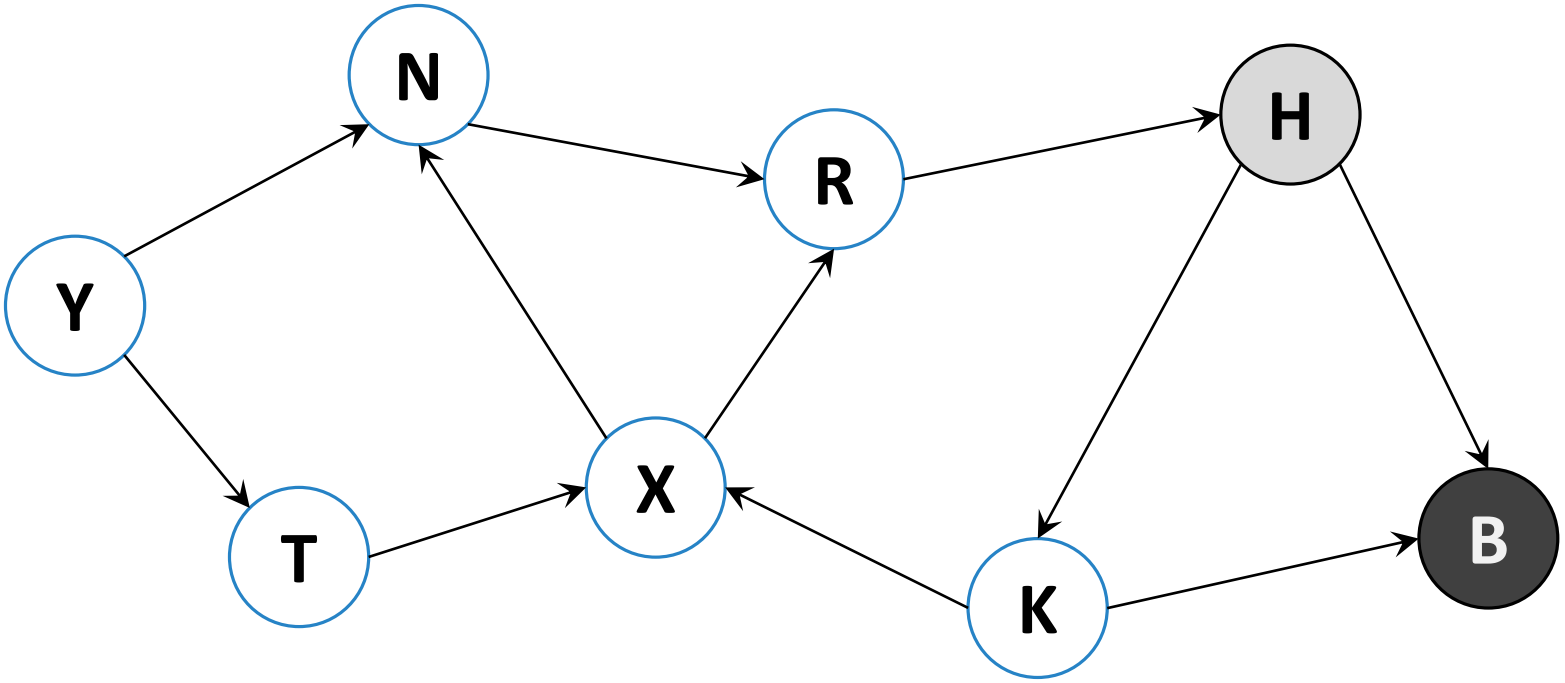
¿Hay un ciclo luego de **B**?

El algoritmo en acción



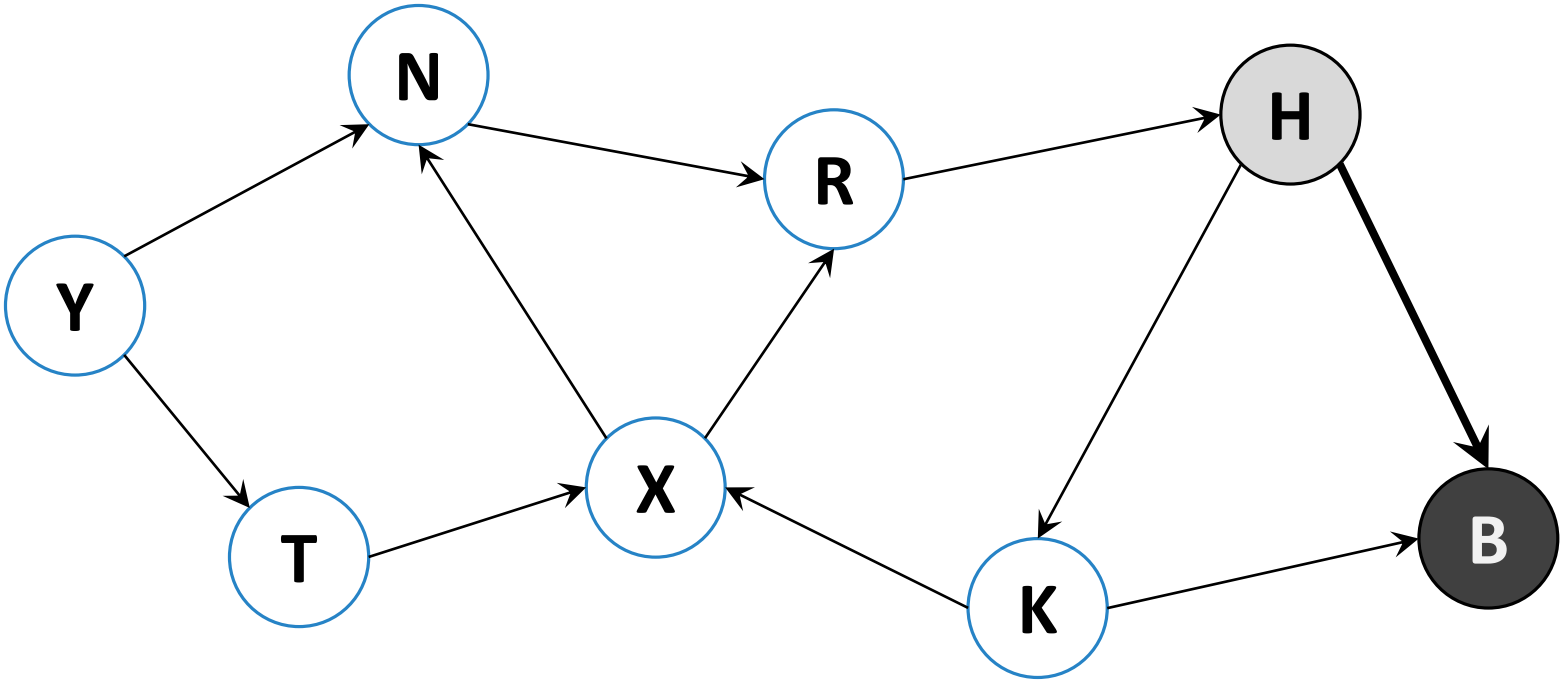
No

El algoritmo en acción



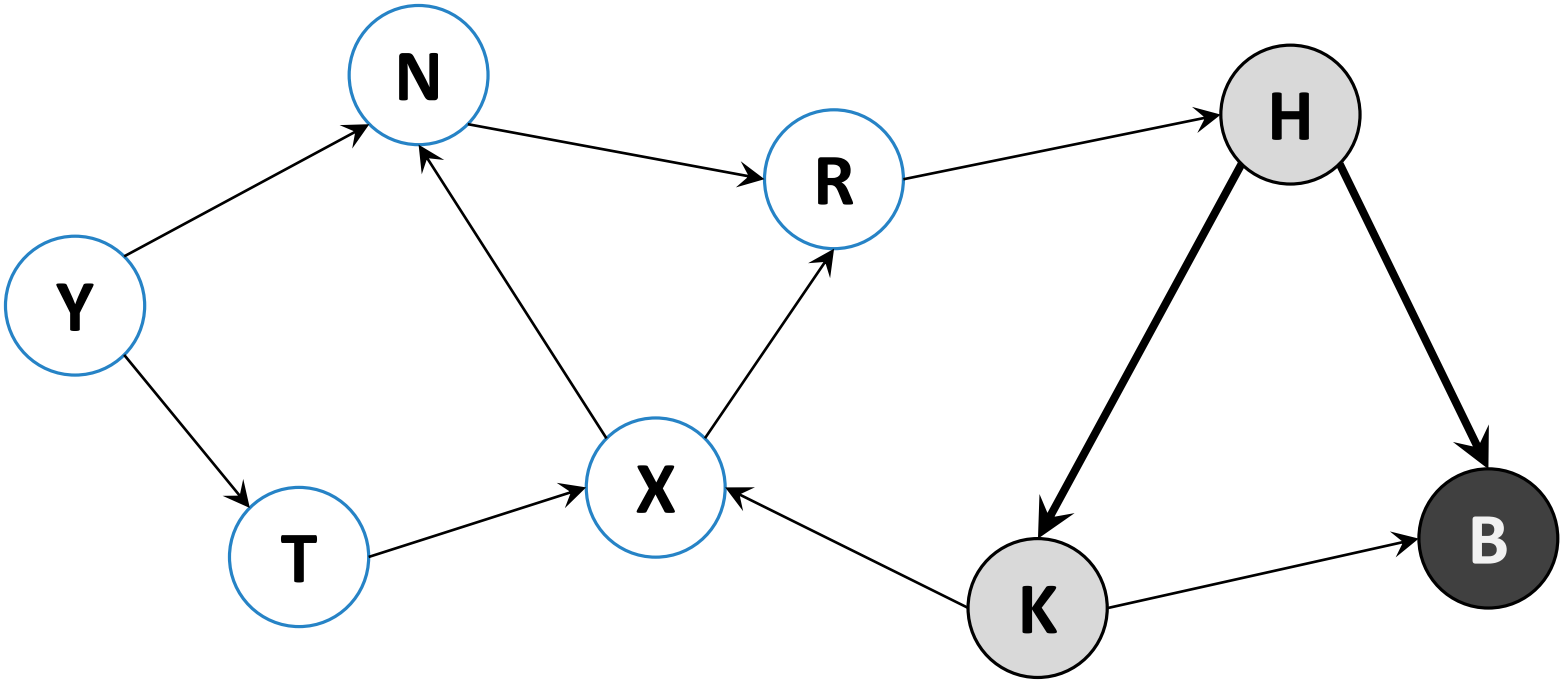
OK, ¿hay un ciclo luego de **H**?

El algoritmo en acción



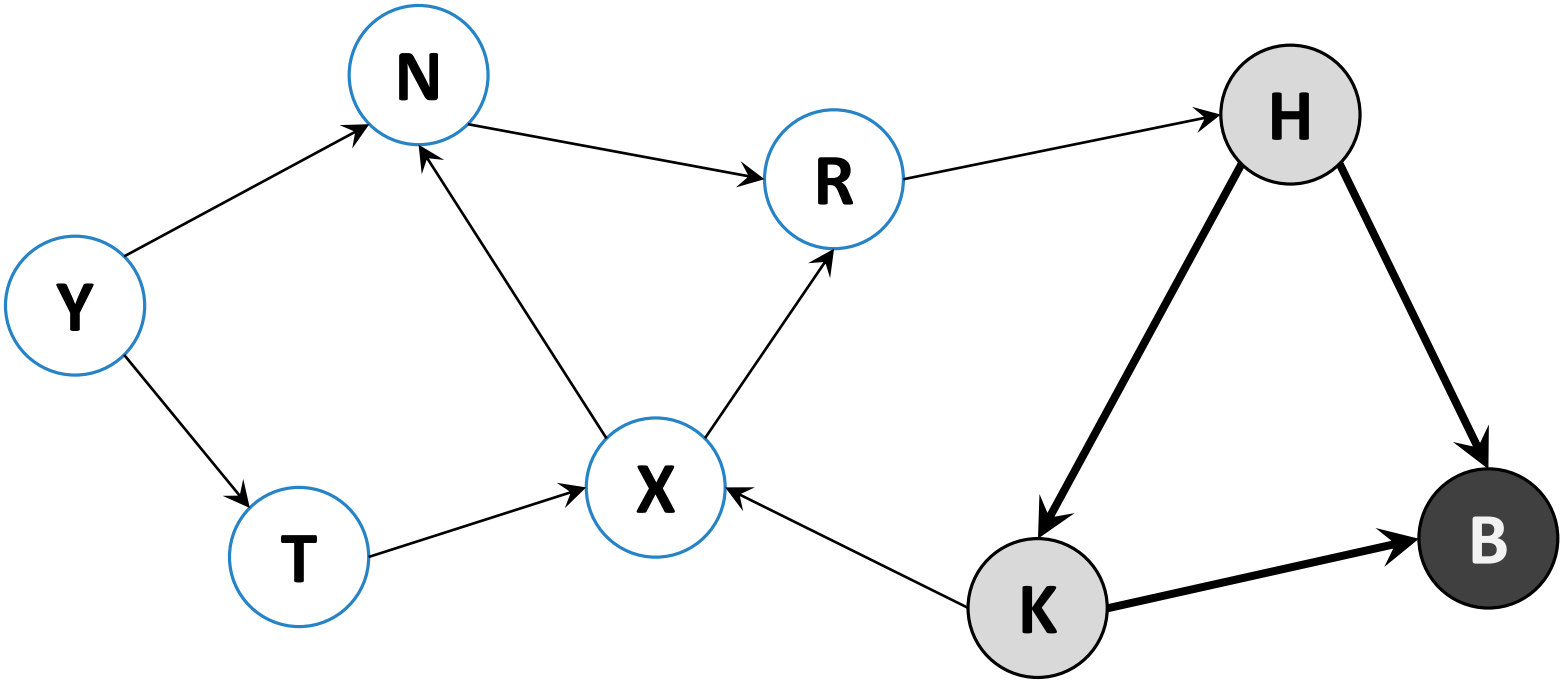
¿Hay un ciclo luego de **H**?

El algoritmo en acción



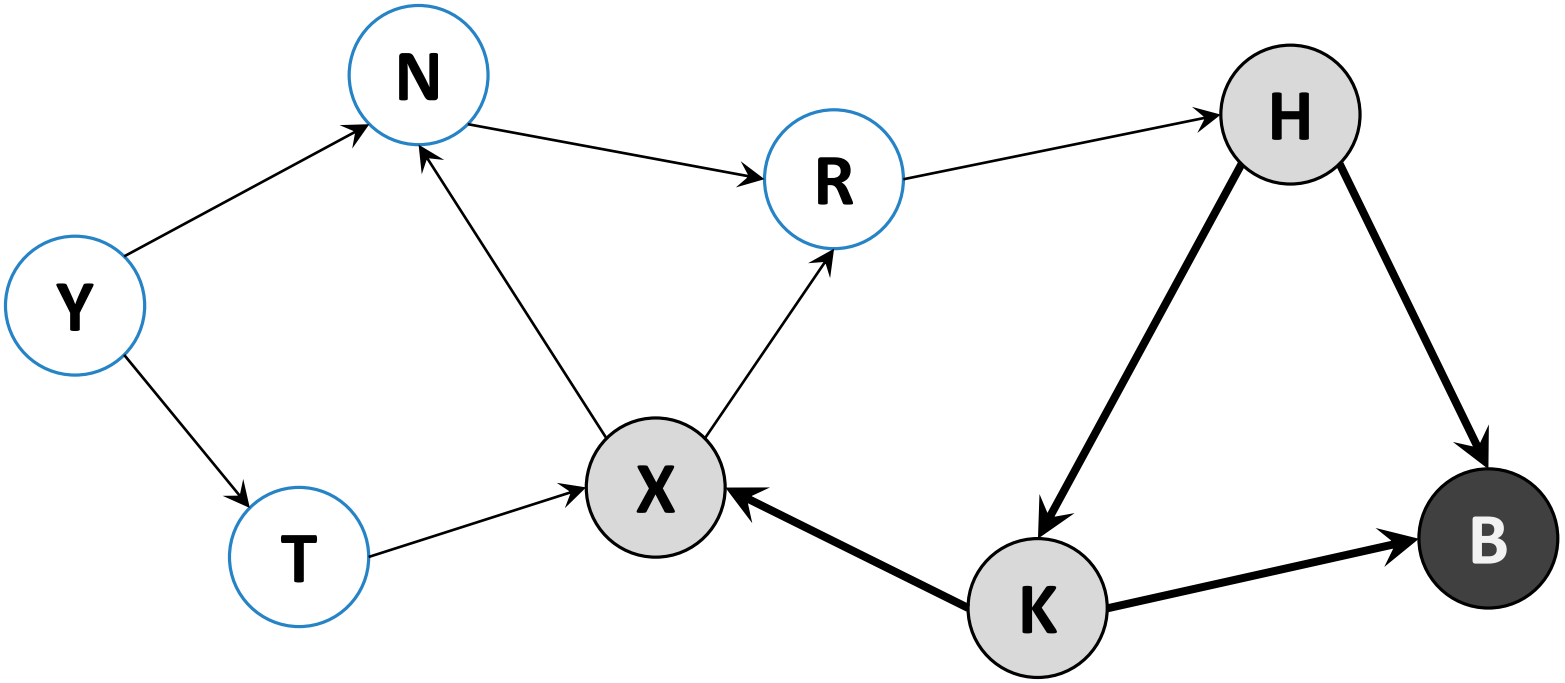
¿Hay un ciclo luego de **K**?

El algoritmo en acción



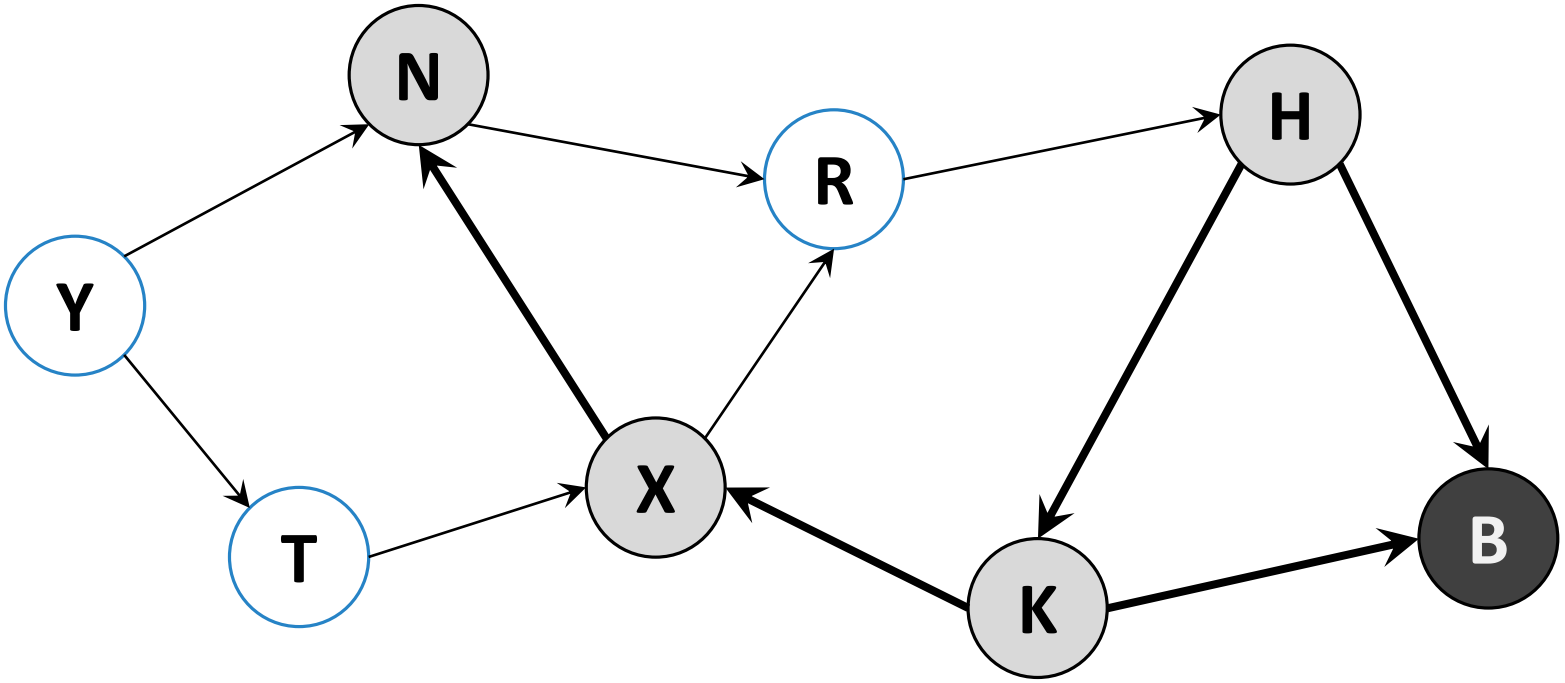
¿Hay un ciclo luego de **K**?

El algoritmo en acción



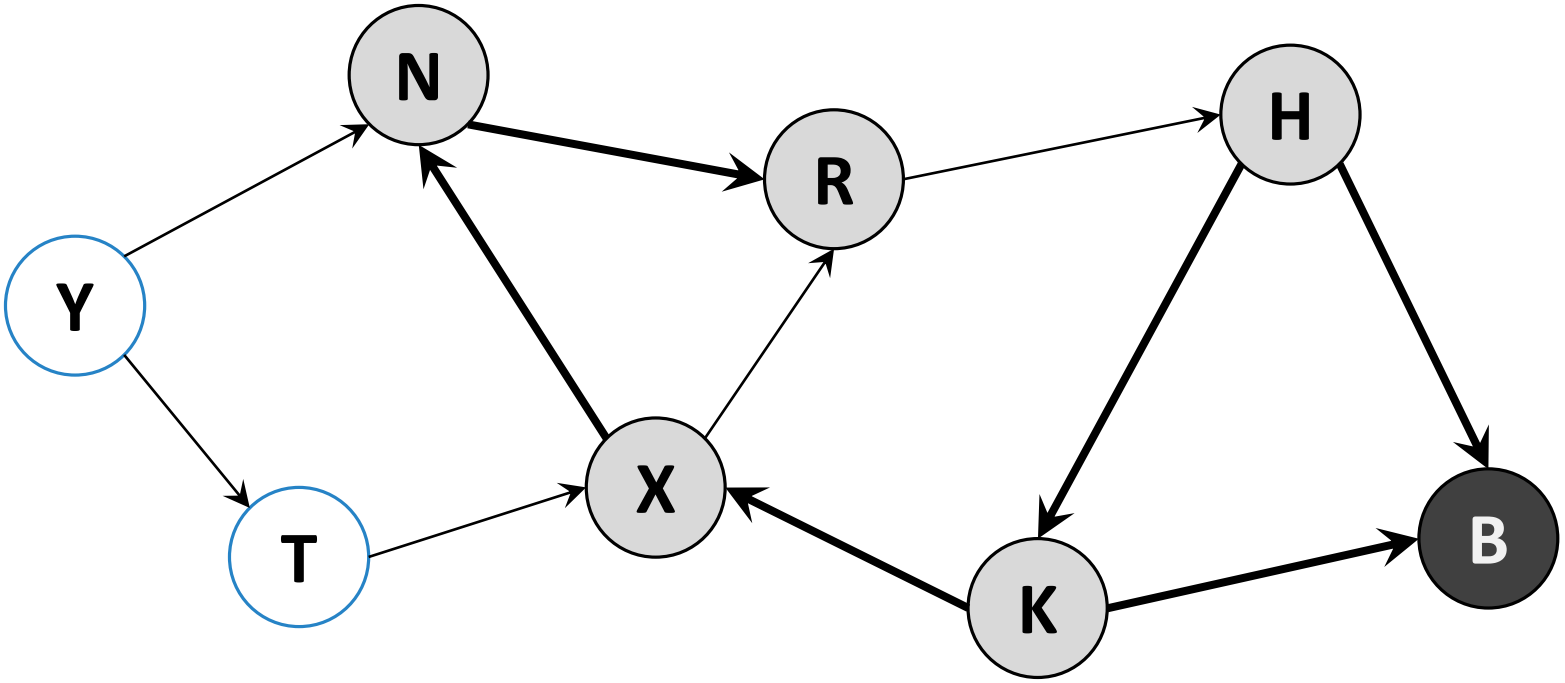
¿Hay un ciclo luego de **X**?

El algoritmo en acción



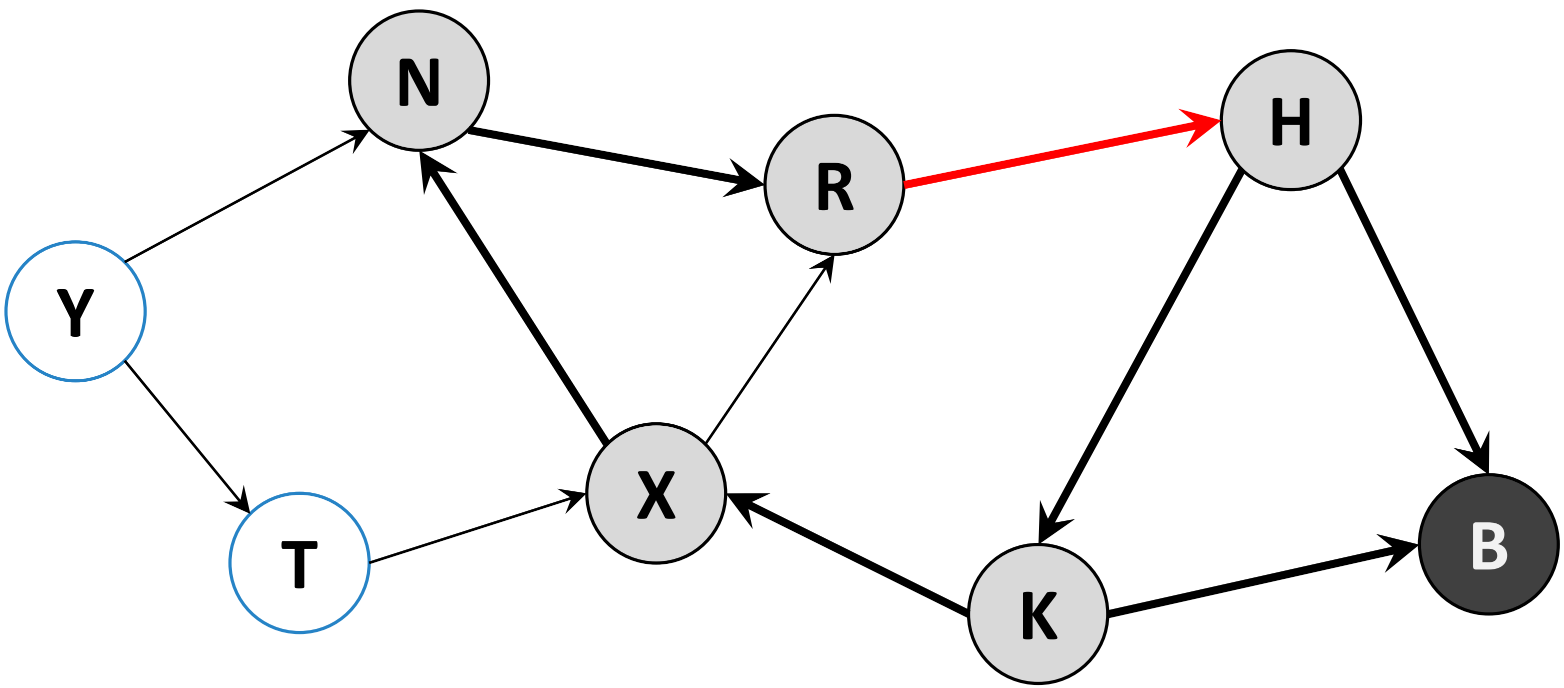
¿Hay un ciclo luego de **N**?

El algoritmo en acción



¿Hay un ciclo luego de **R**?

El algoritmo en acción



!!!

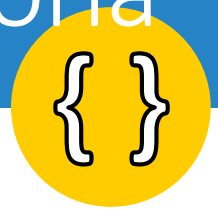
Depth First Search (DFS)

Algoritmos como estos se llaman de **búsqueda en profundidad**

Llegan hasta el final de una rama antes de empezar a explorar otra

¿Cuál es la complejidad de estos algoritmos?

Representación de grafos en memoria



Hay dos principales maneras de representar un grafo:

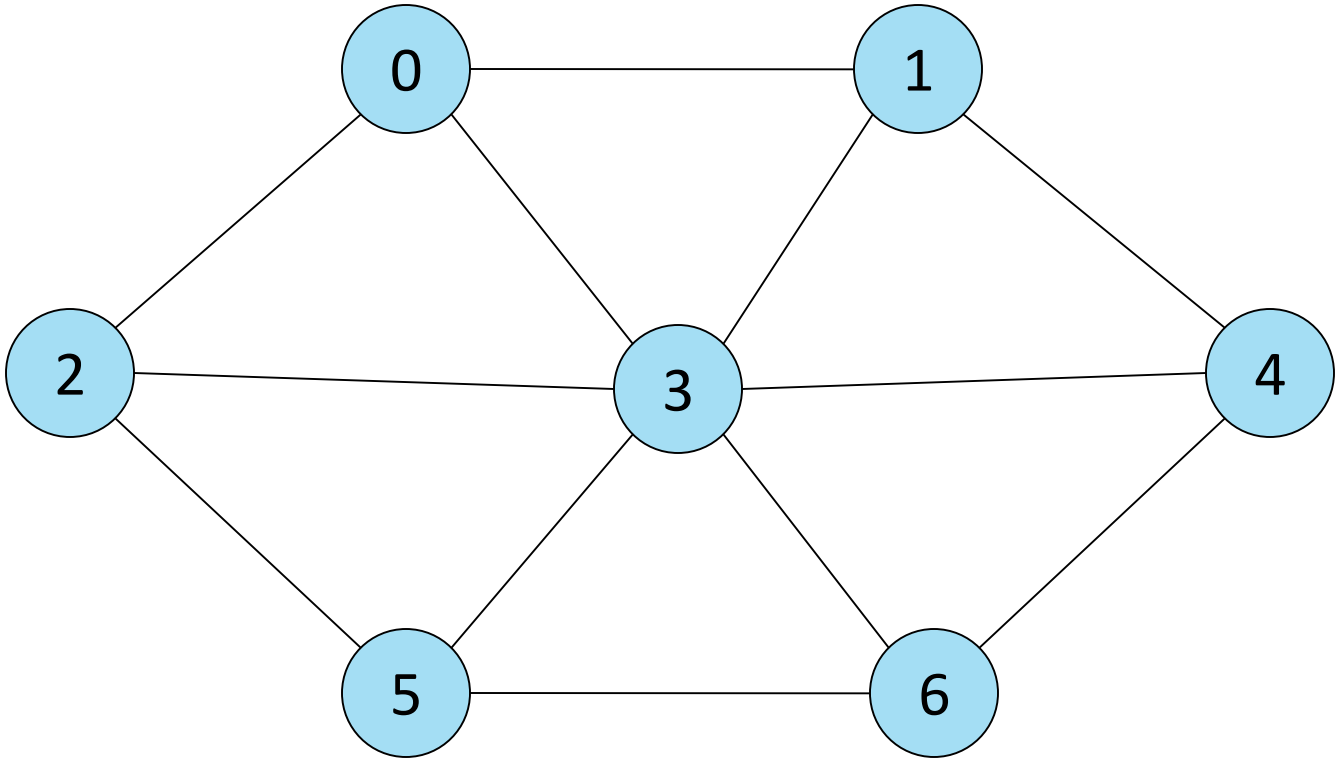
1. Listas de adyacencias

   Cada nodo tiene una lista de los nodos a los que tiene una arista

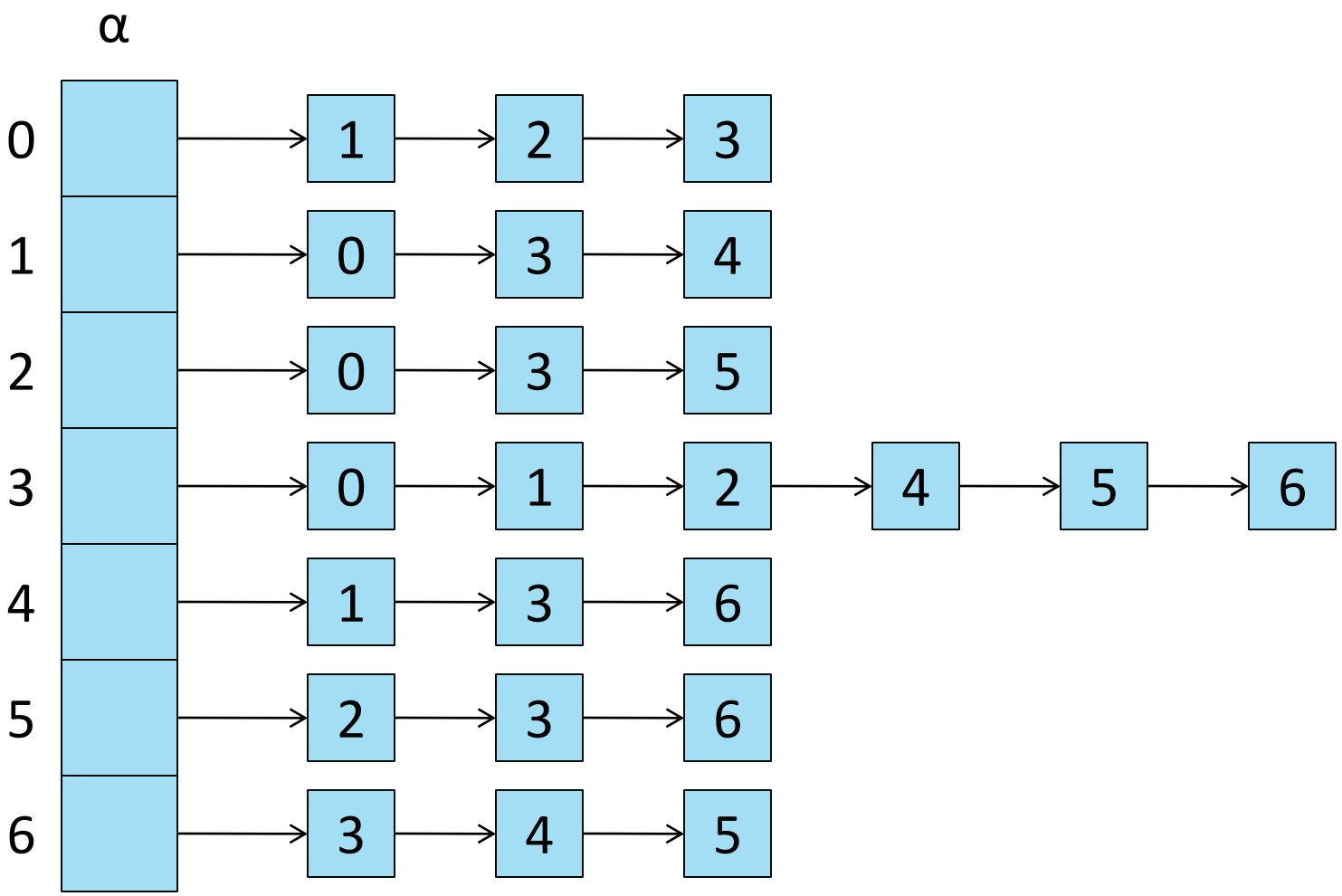
2. Matriz de adyacencias

   La coordenada x, y de la matriz indica si la arista (x, y) está en el grafo

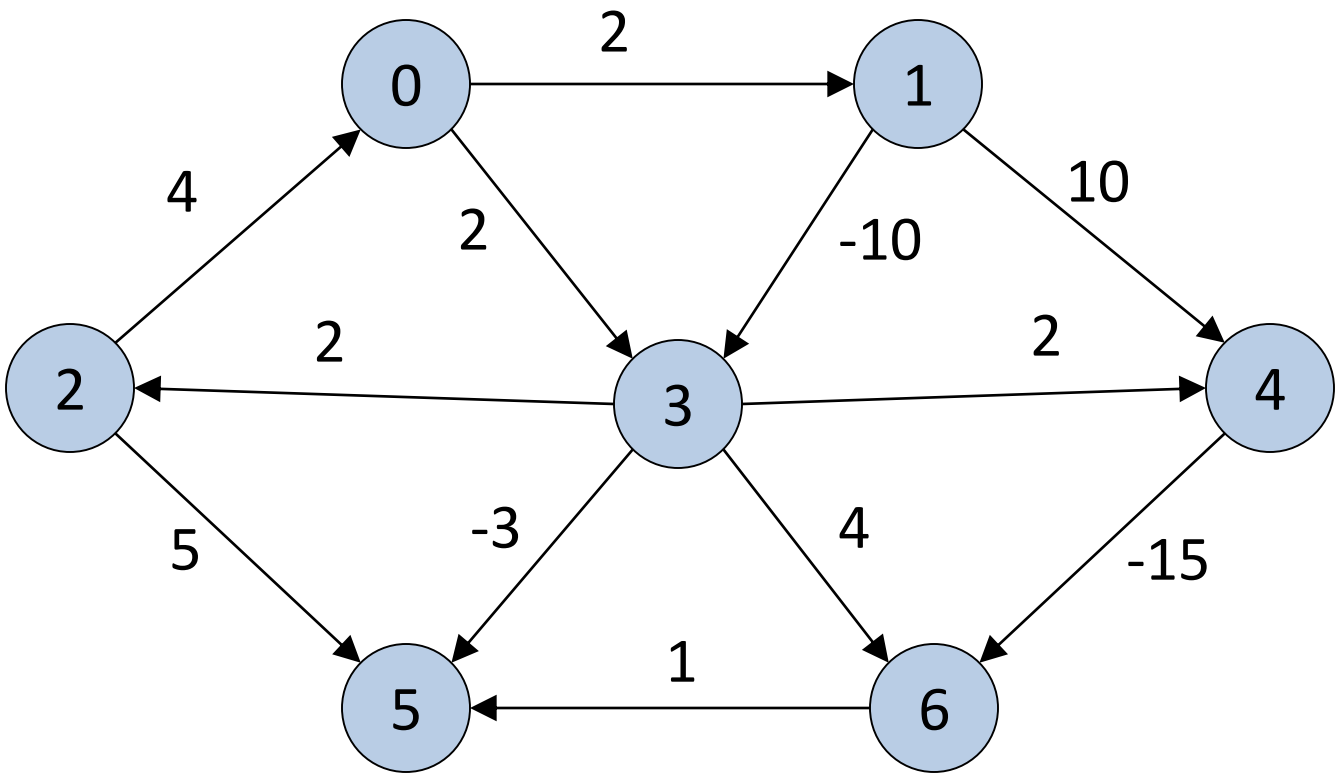
Un grafo *no direccional sin costos*



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



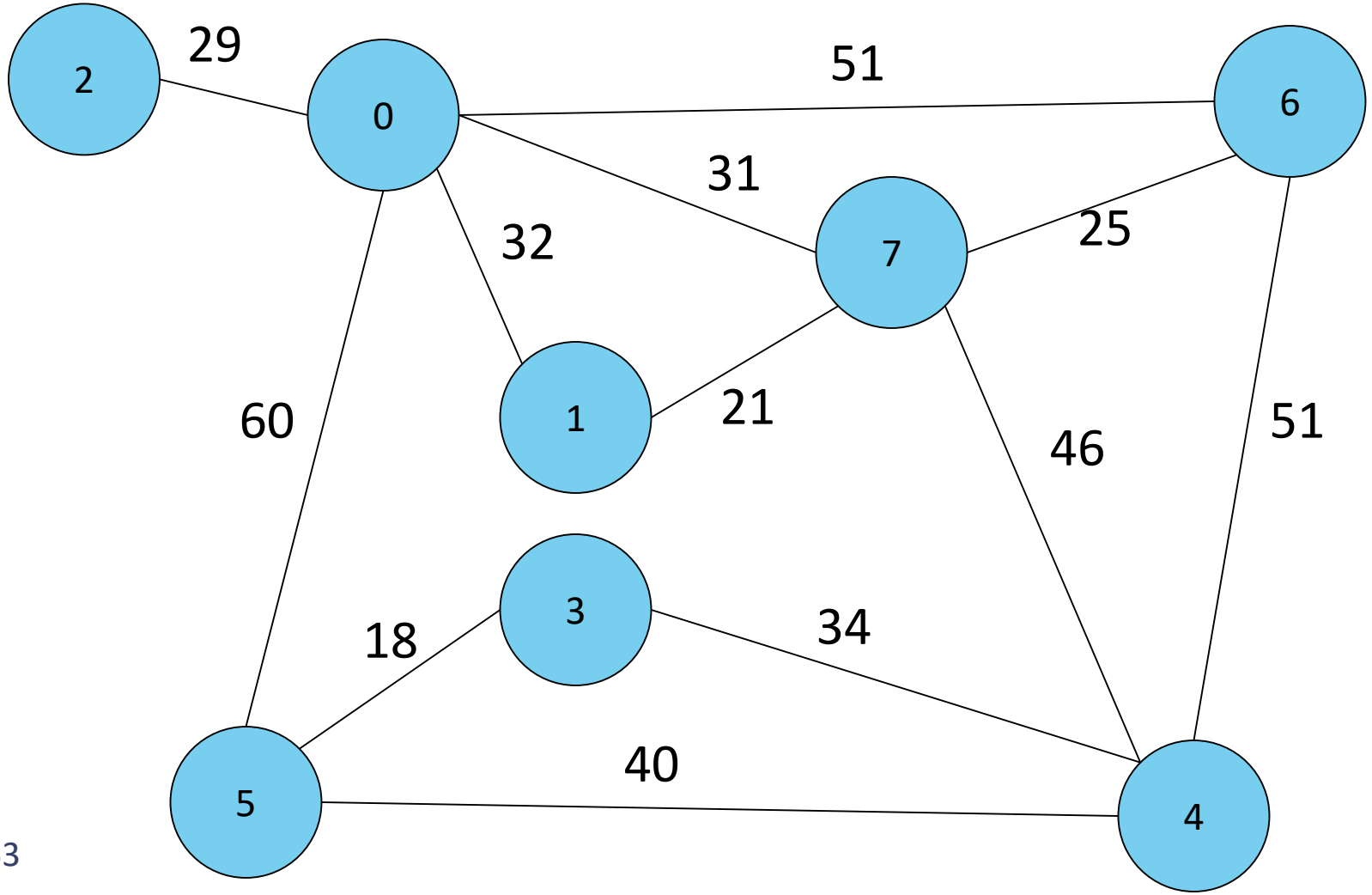
Un grafo *direccional con costos*



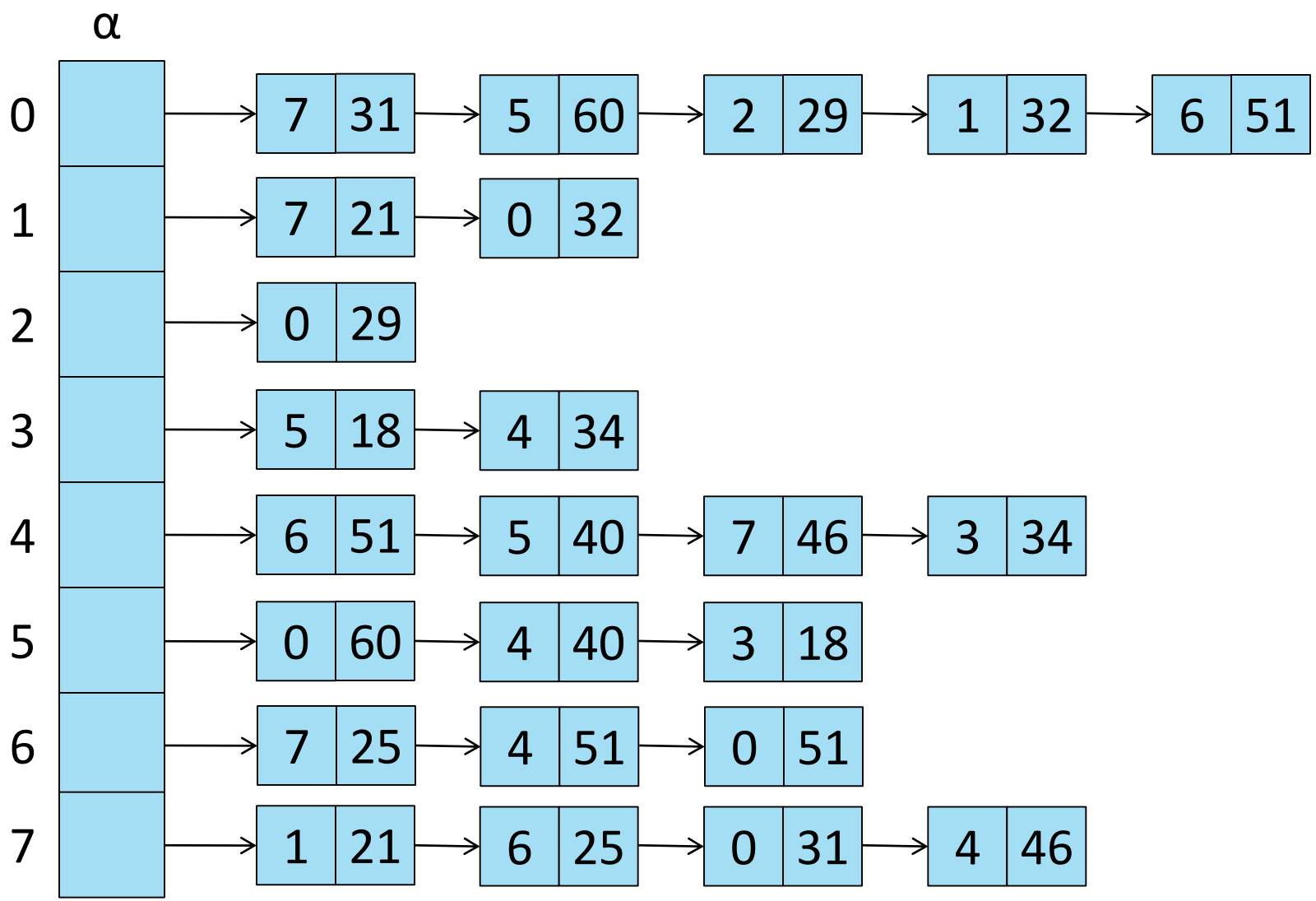
El grafo anterior representado por una matriz de adyacencias

	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



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
```

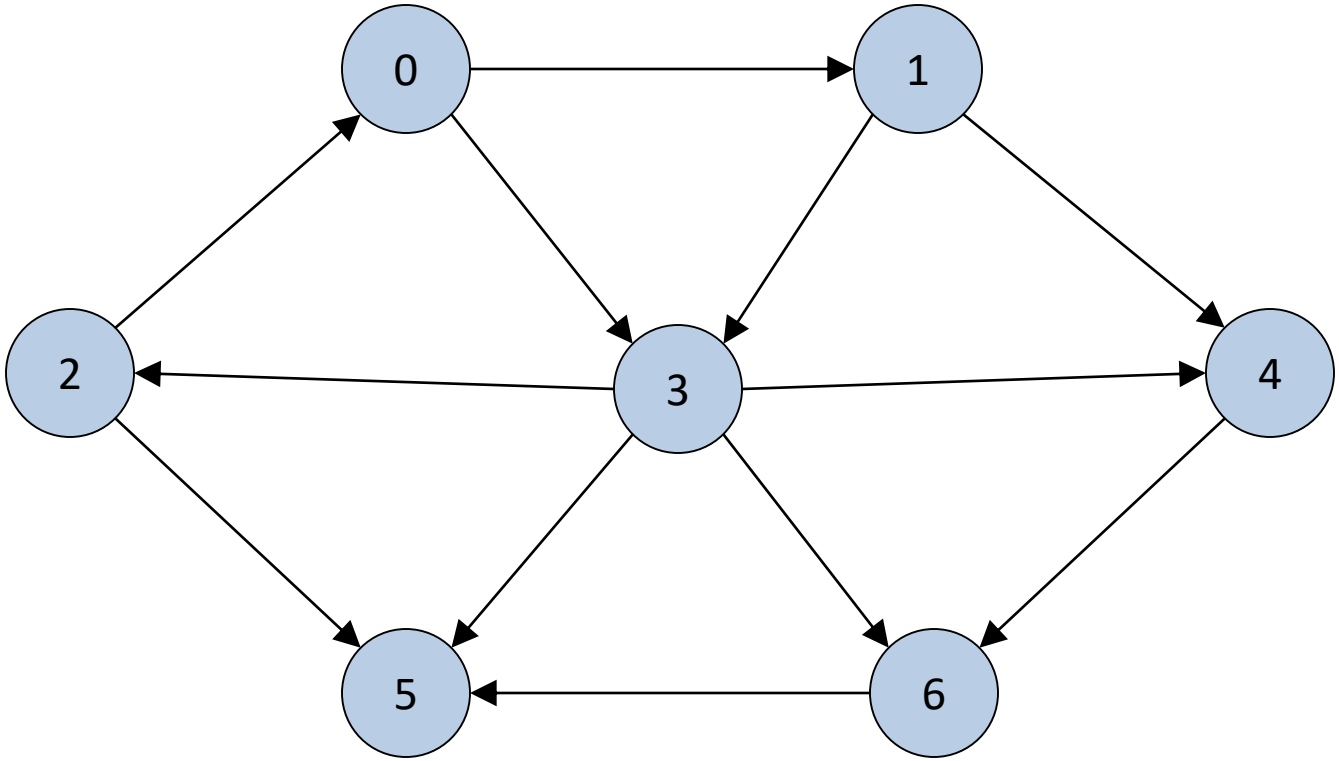
Para dos vértices cualquiera, *u* y *v*, sólo una de las siguientes tres condiciones es verdadera

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

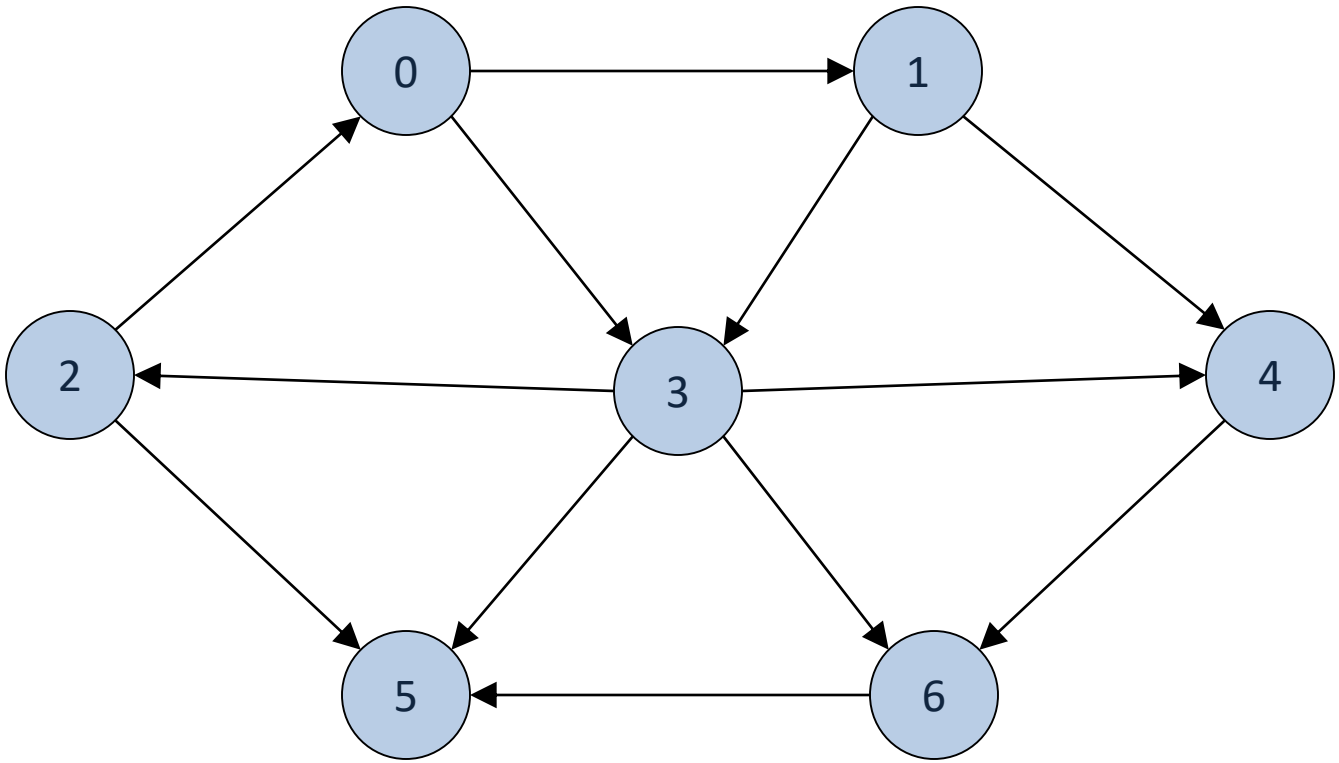
El intervalo [*u.d*, *u.f*] *está contenido* en el intervalo [*v.d*, *v.f*], y *u es descendiente de v* en un árbol DFS

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*

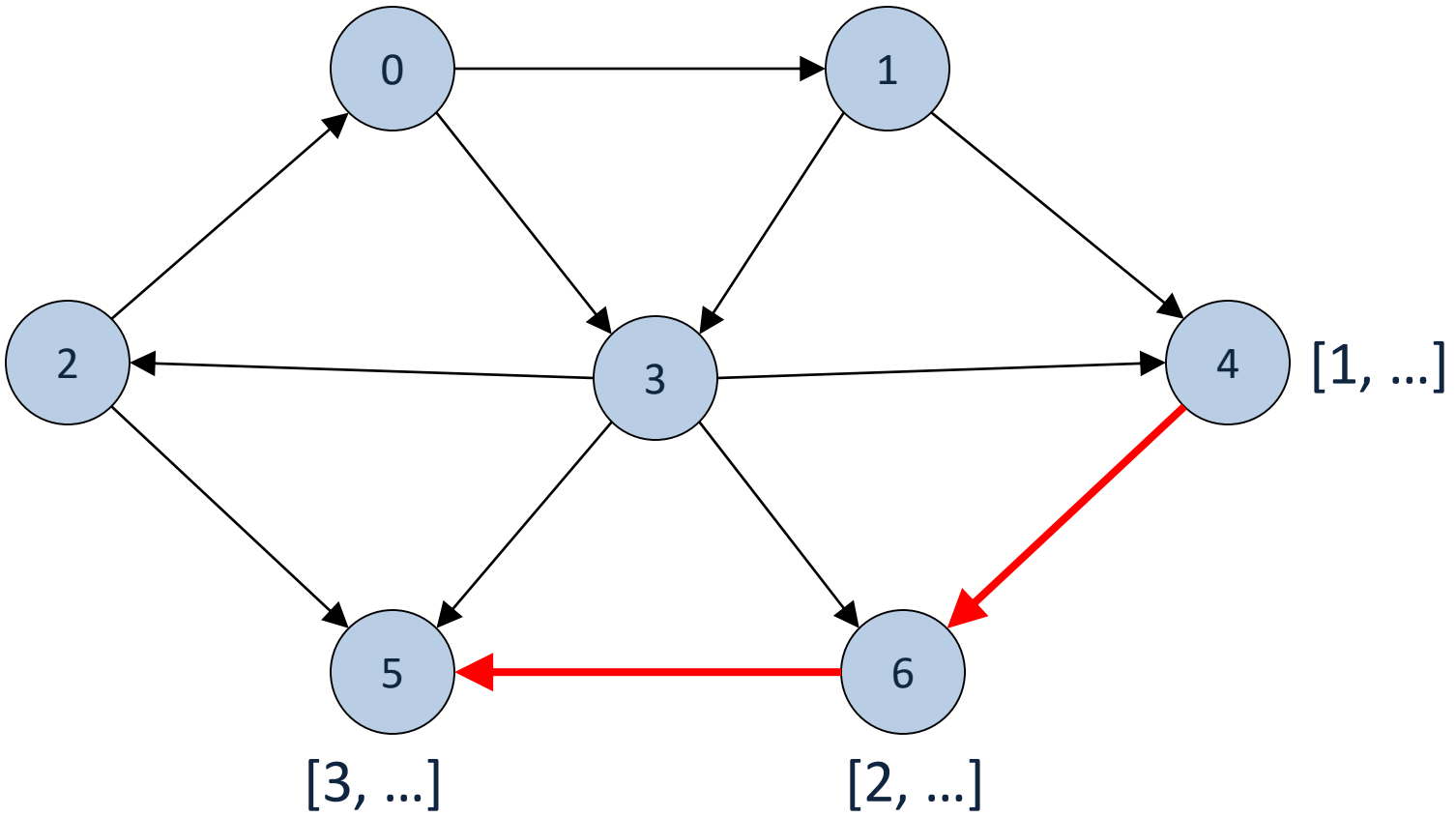


DFS de *G*, a partir del vértice 4 …

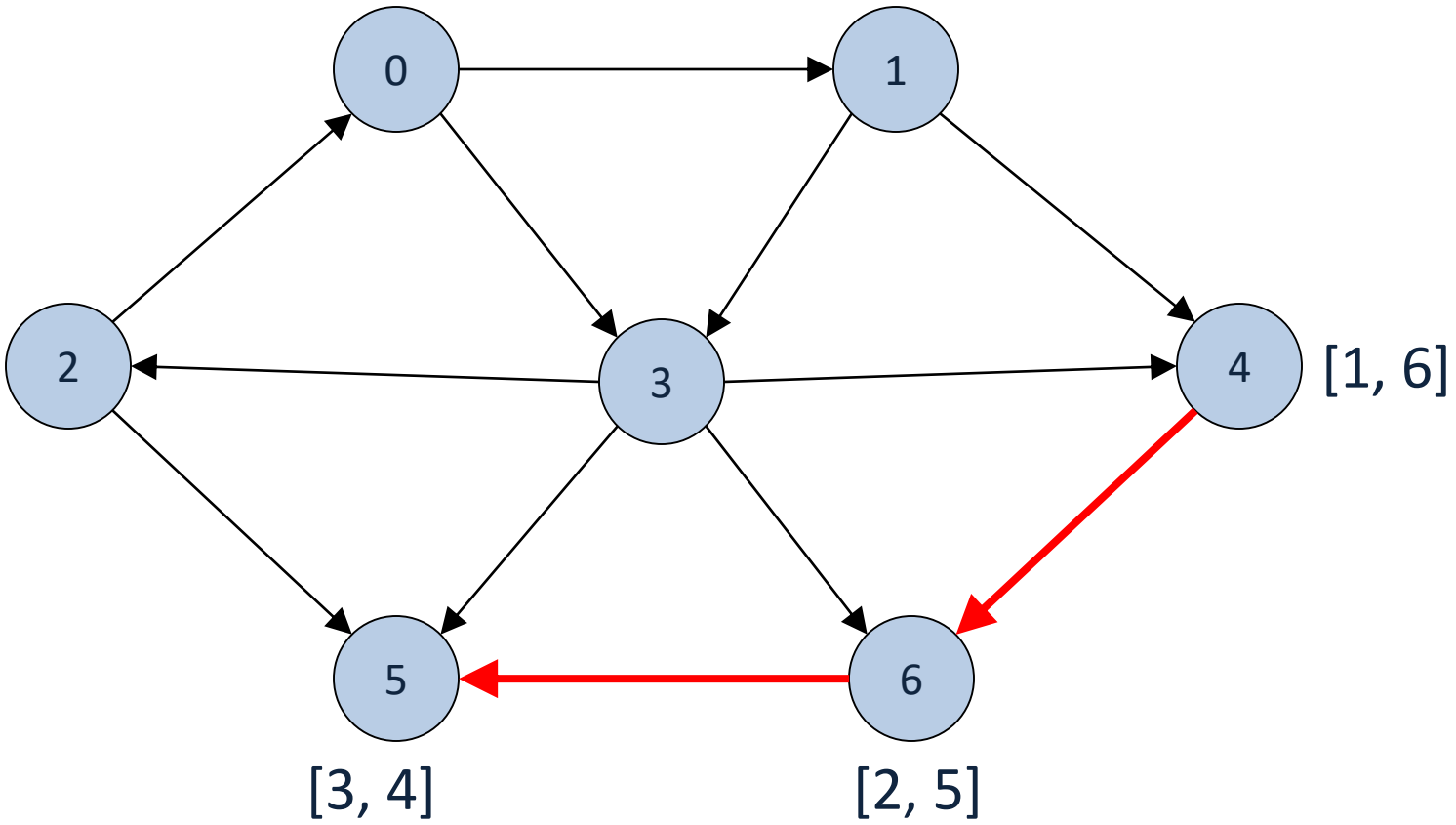


[1, …]

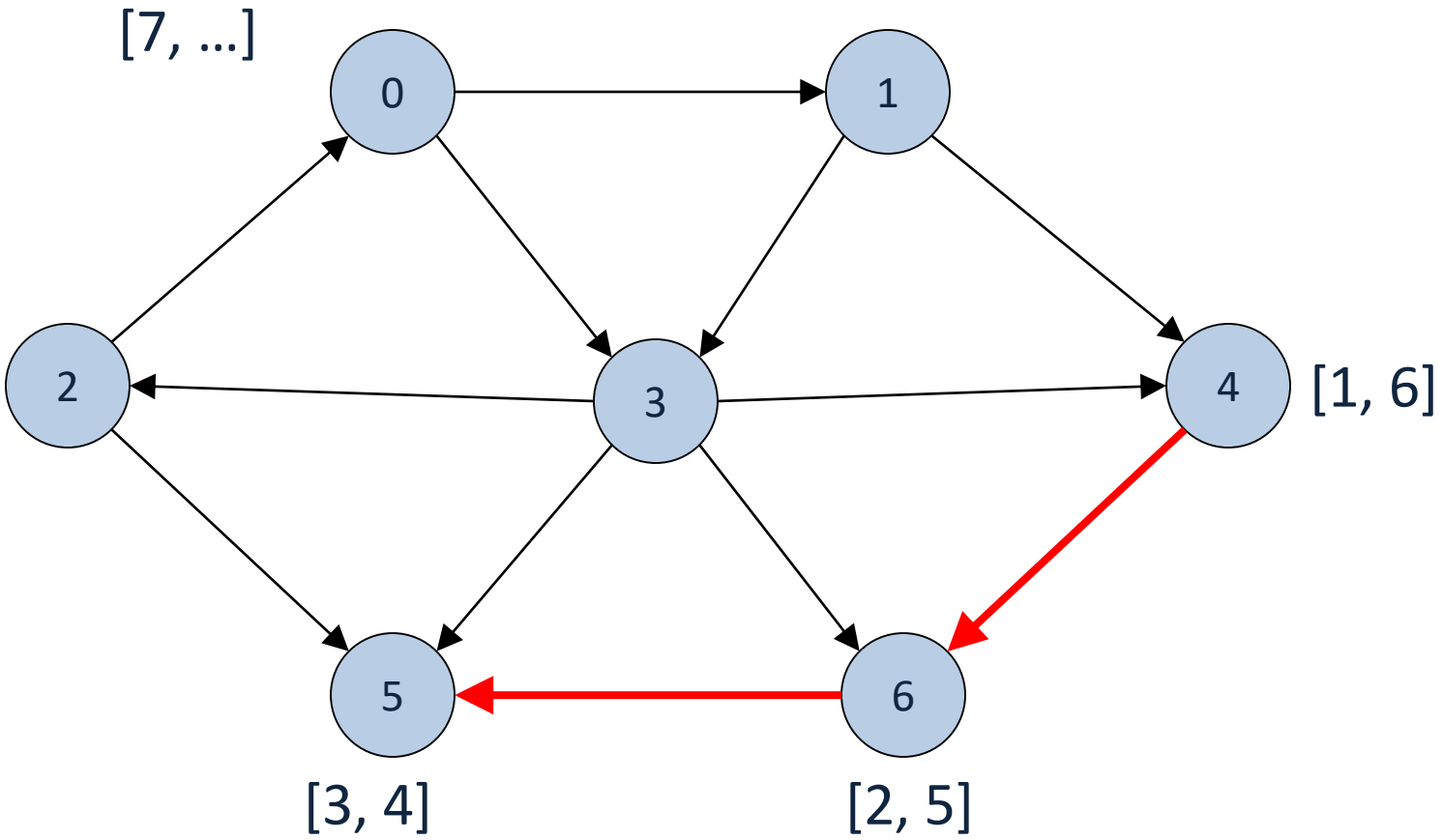
DFS de *G*, a partir del vértice 4 …



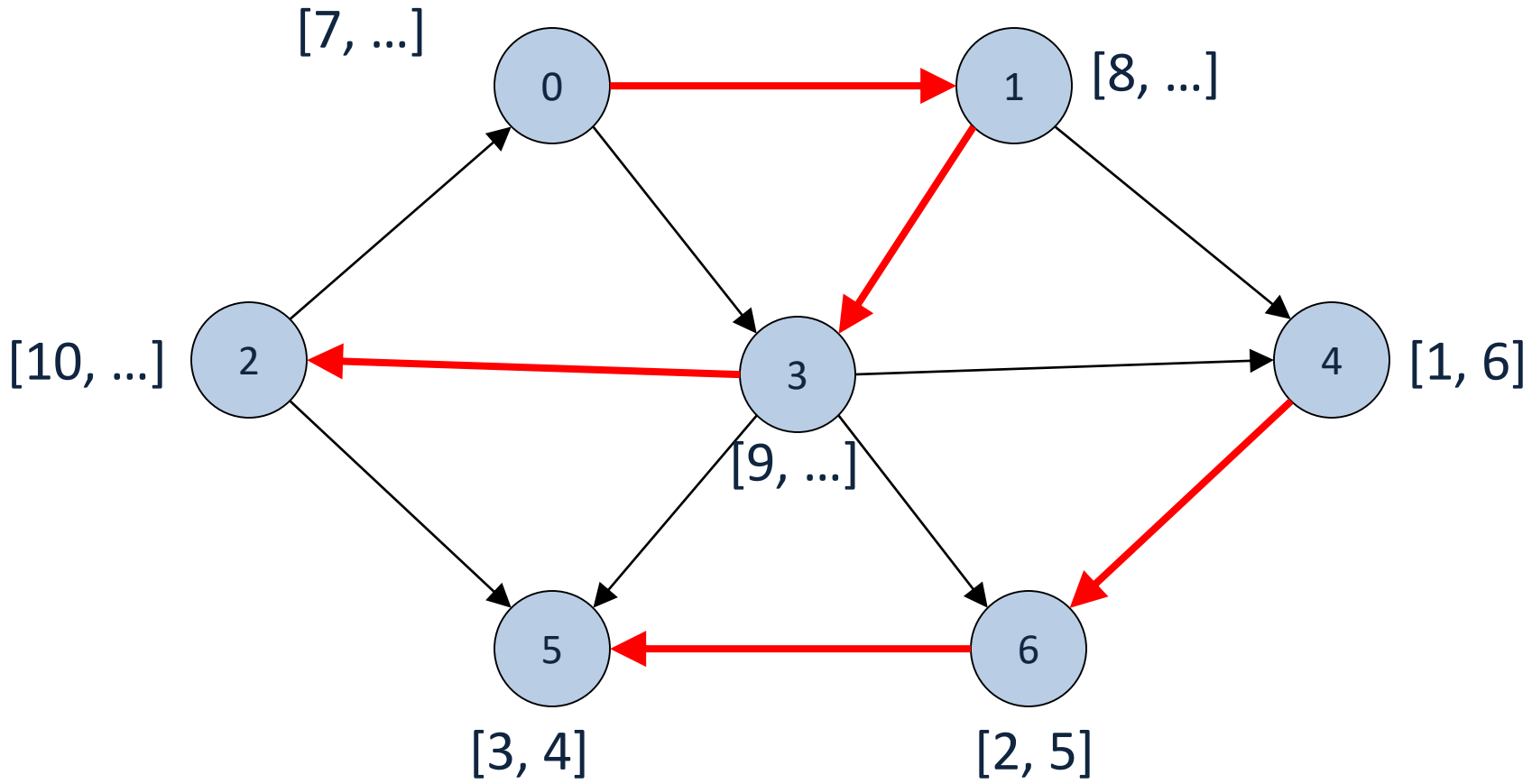
DFS de *G*, a partir del vértice 4 …



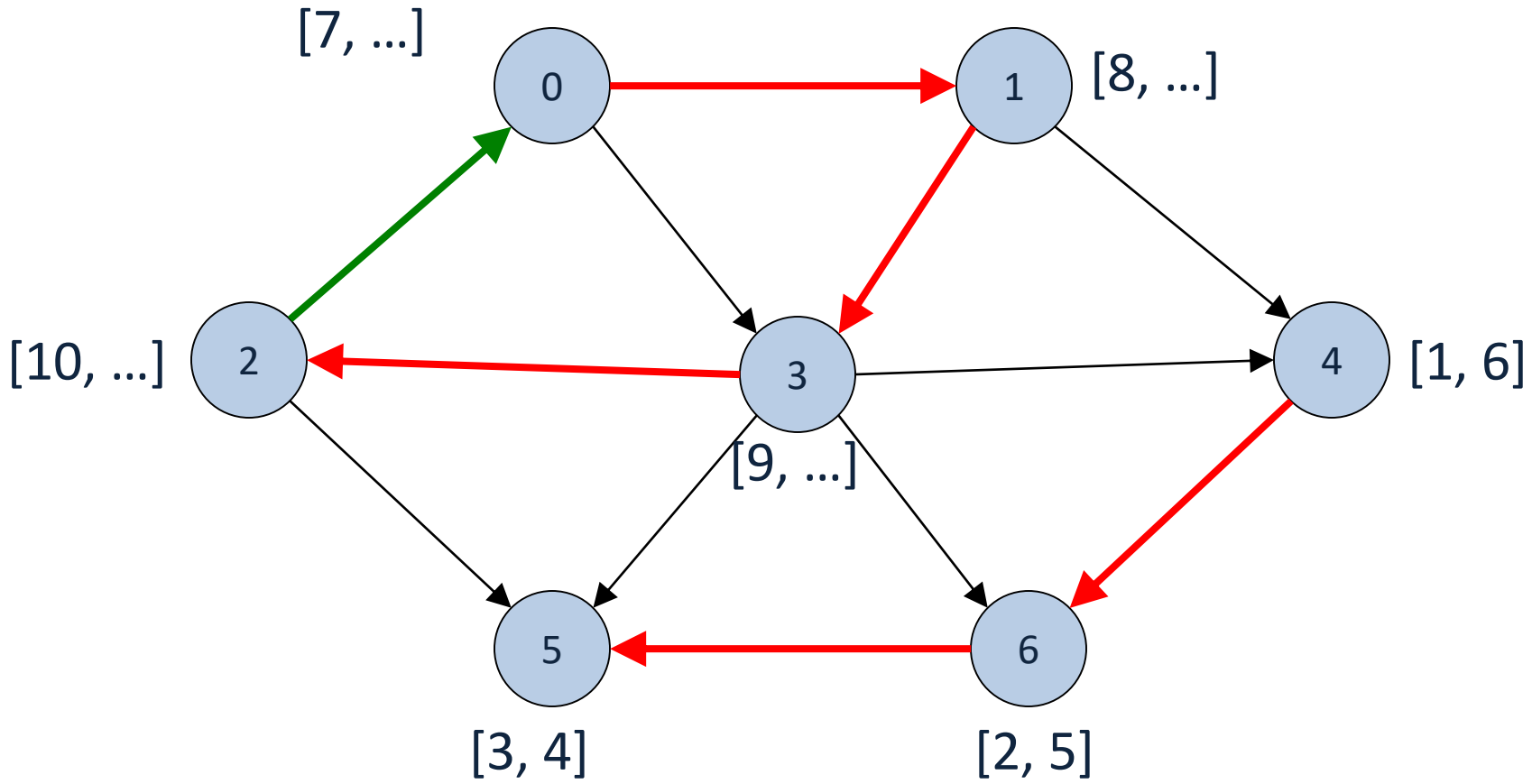
DFS de *G*, a partir del vértice 4 y, luego, del vértice 0



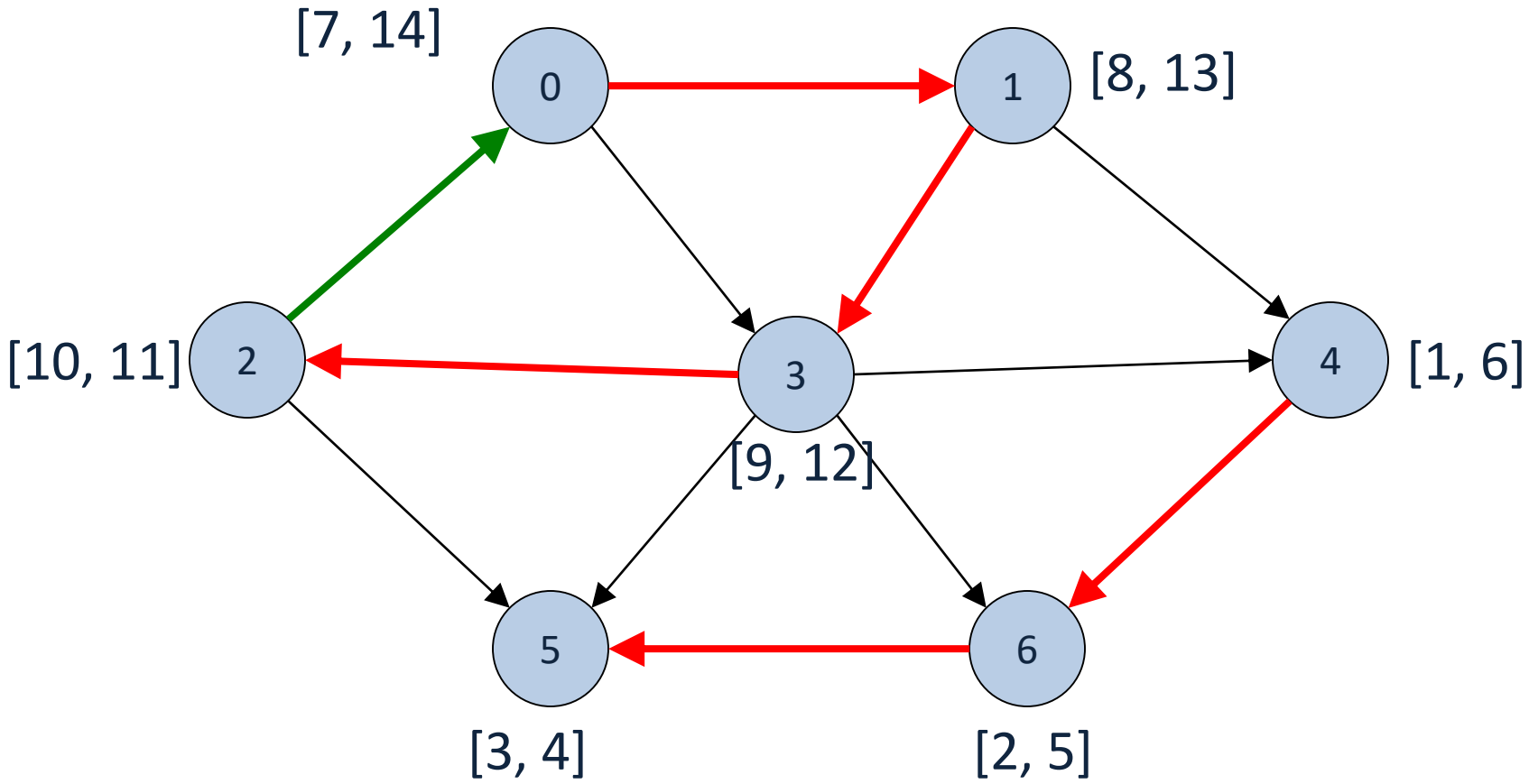
DFS de *G*, a partir del vértice 4 y, luego, del vértice 0



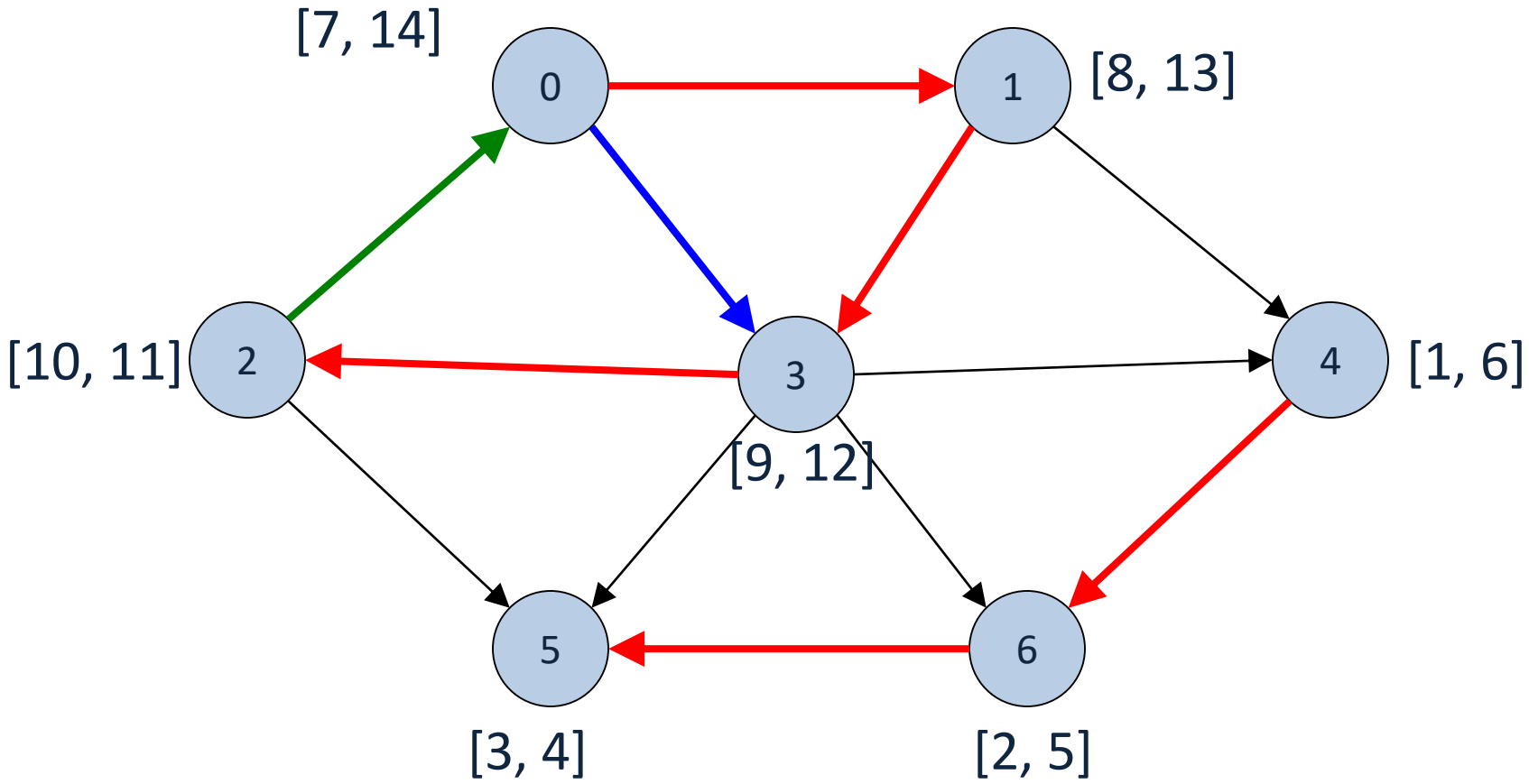
DFS de *G*, a partir del vértice 4 y, luego, del vértice 0



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*

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

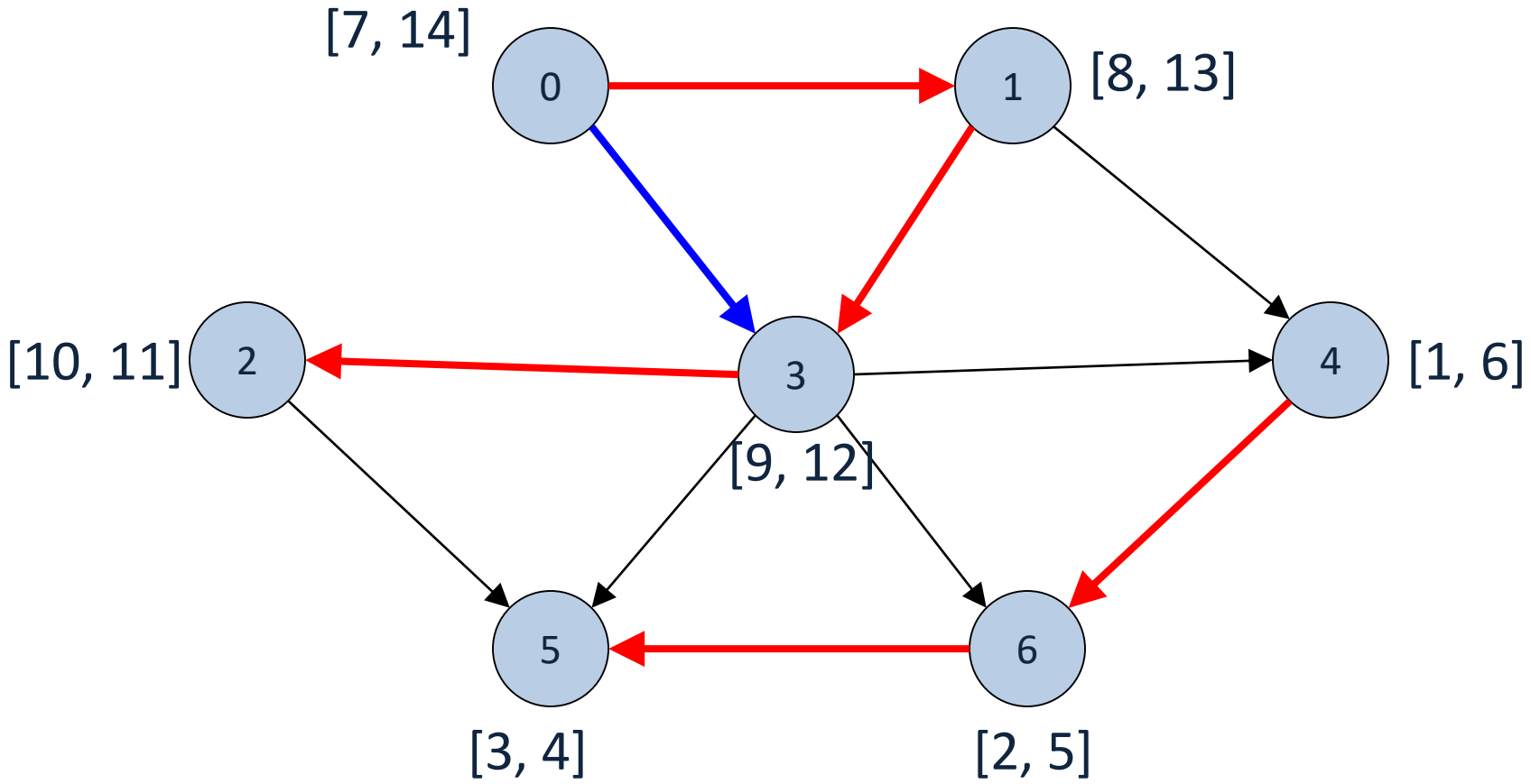
`topSort()`

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

2) Cada vez que calculamos el tiempo *f* para un vértice, insertamos 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



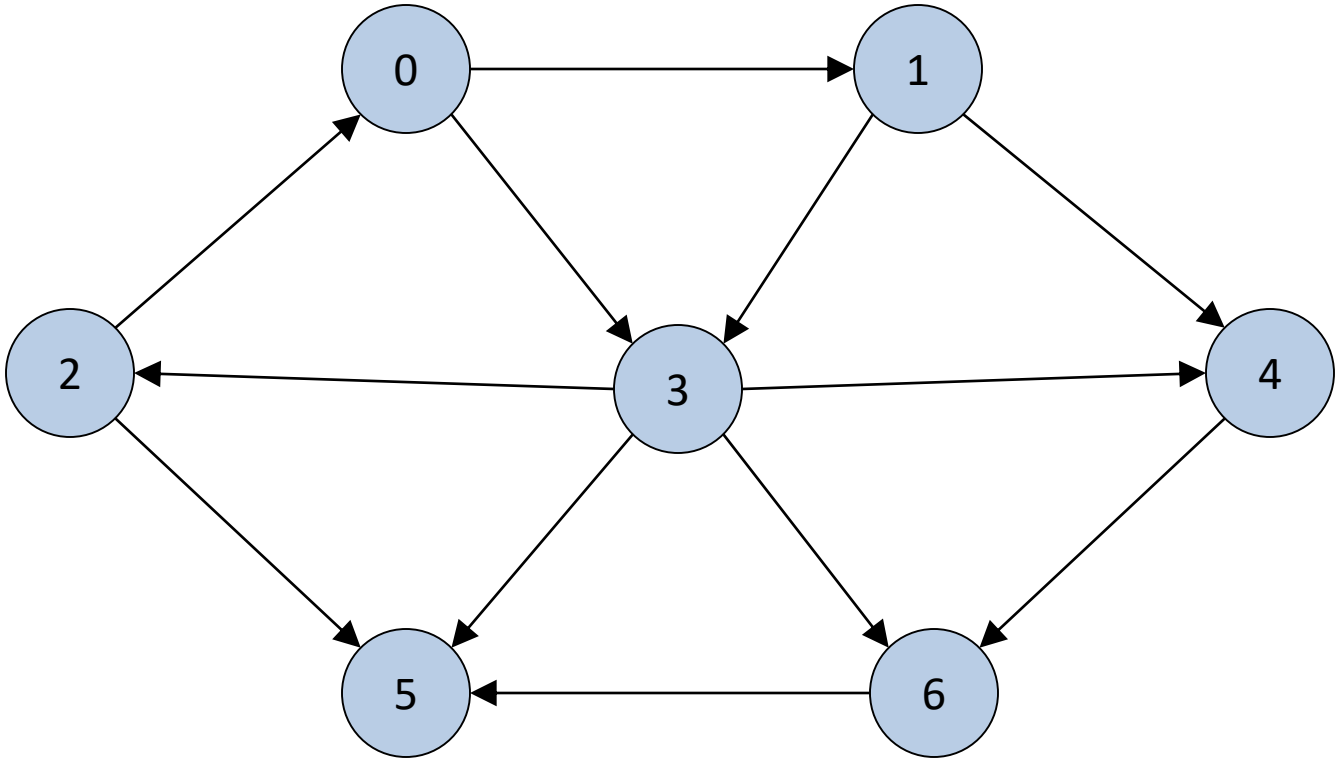
En un grafo que tiene ciclos es *imposible* producir un orden lineal de sus vértices:

- p.ej., el grafo de la próxima diap.

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

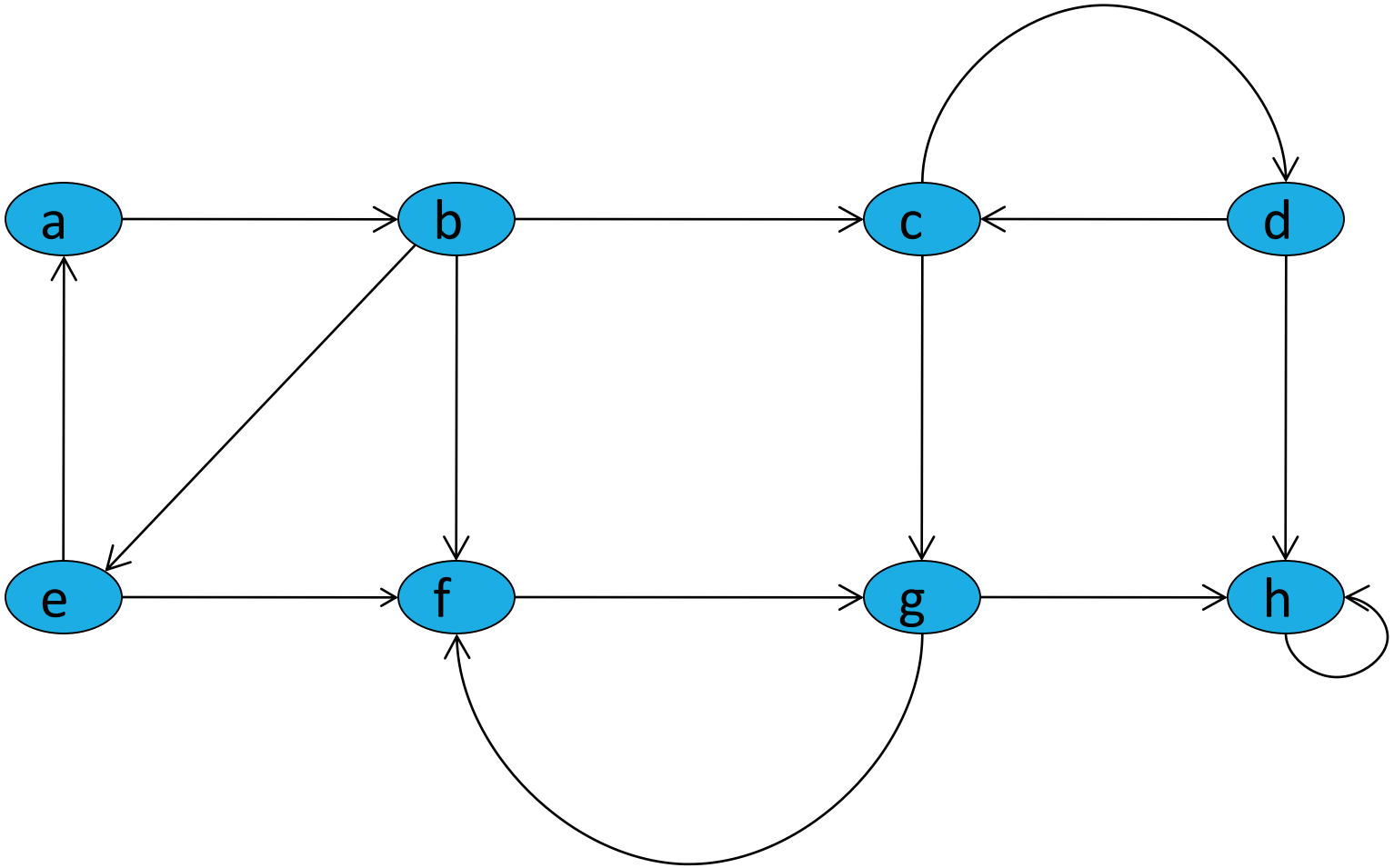
Para demostrar la corrección de `topSort()`, basta demostrar que para cualquier par de vértices *u*, *v*, si hay una arista de *u* a *v*, entonces $v.f < u.f$

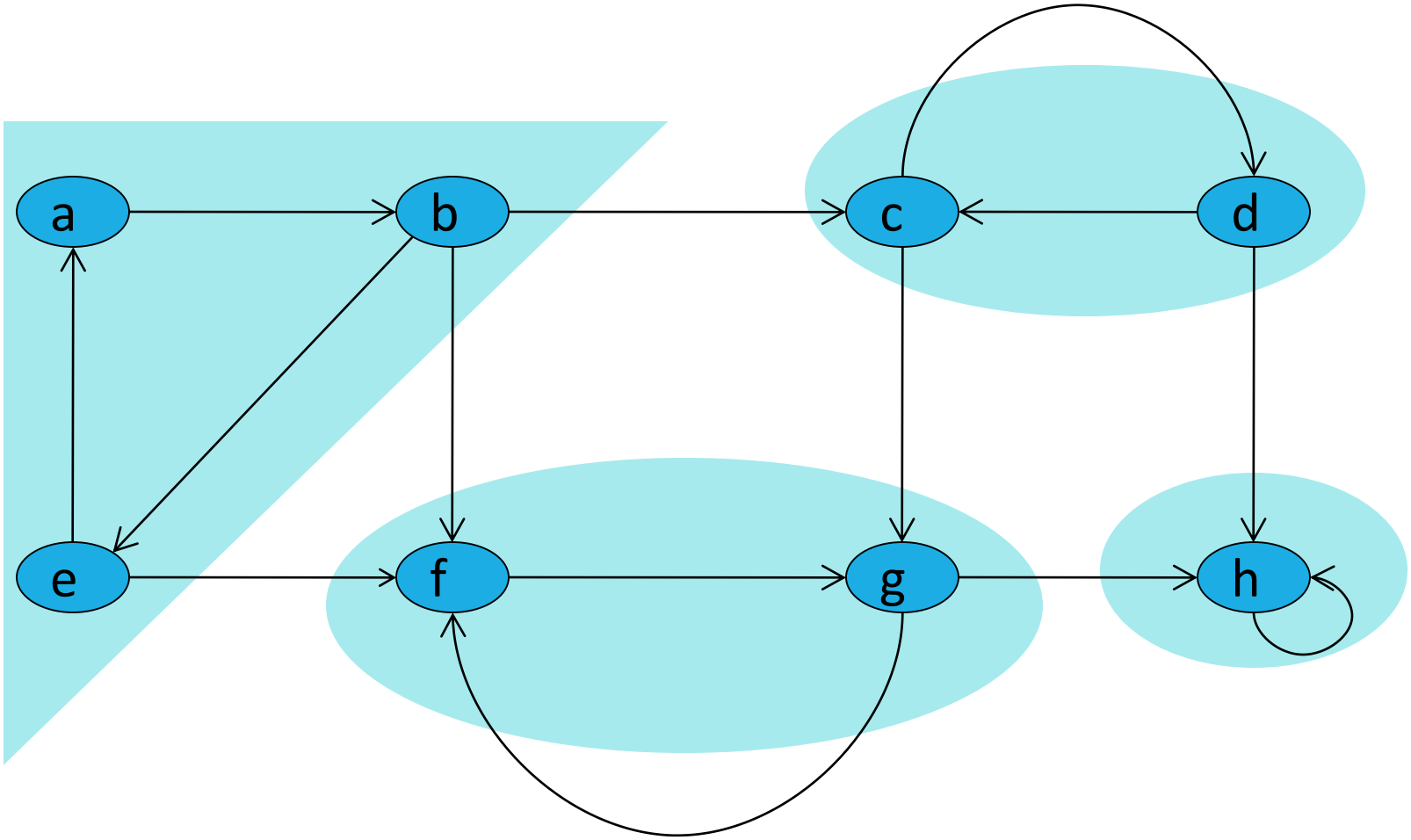
Un grafo *direccional cíclico*



¿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

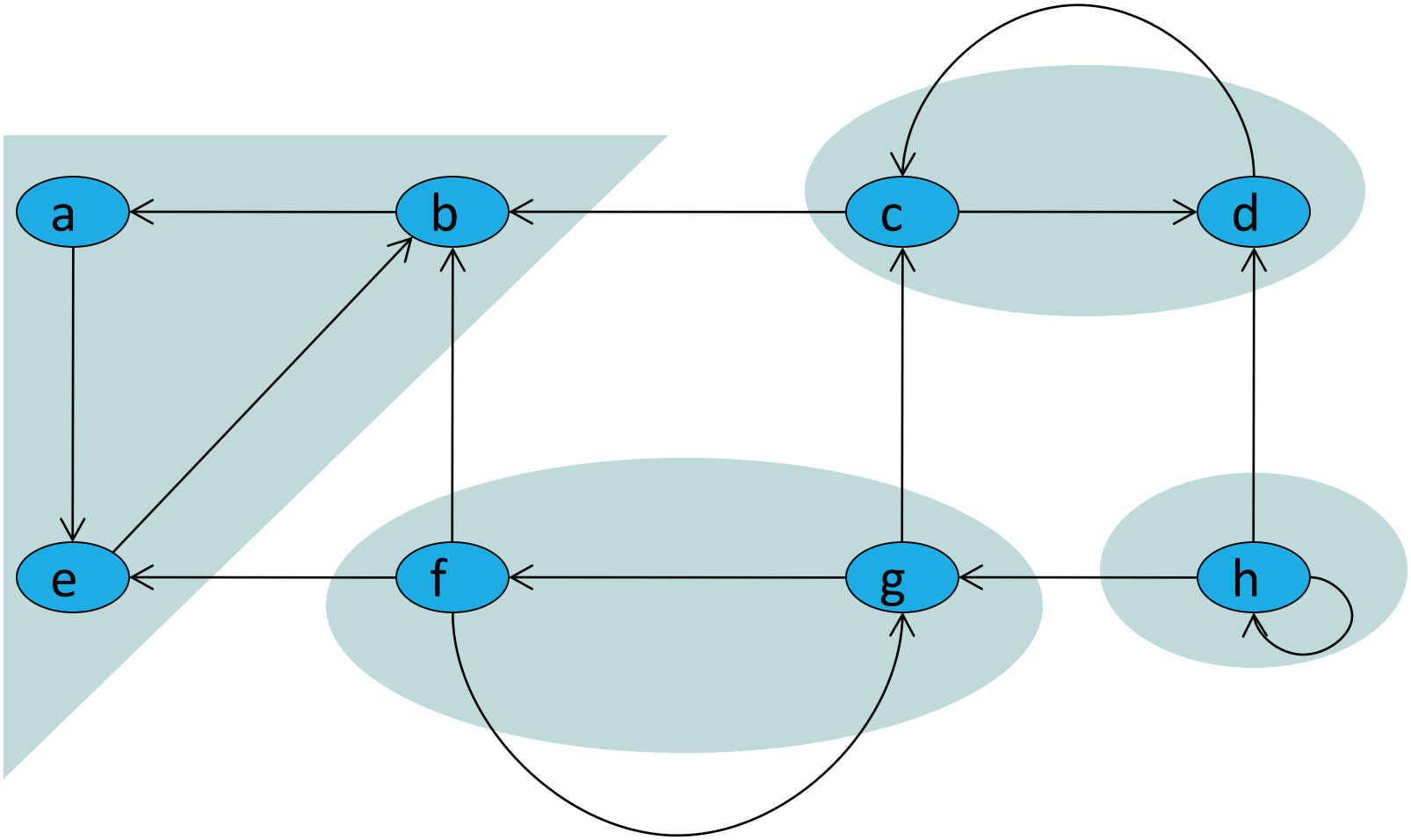
El algoritmo para determinar las scc's de G usa el *grafo transpuesto* de G

$G^T = (V, E^T)$, 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

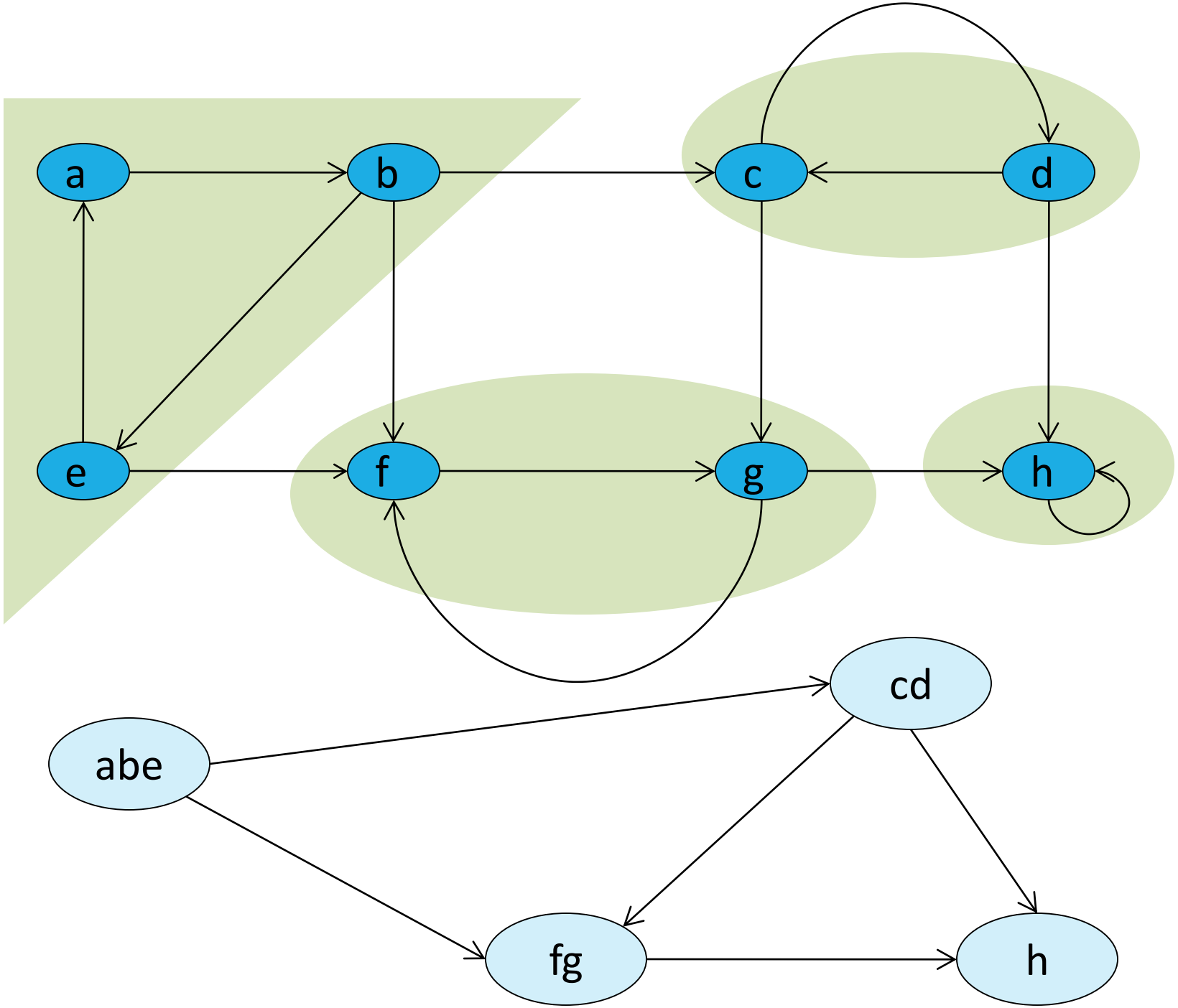
Definamos el *grafo de componentes* de G, $G^{SCC} = (V^{SCC}, E^{SCC})$

Supongamos que G tiene las componentes fuertemente conectadas C_1, C_2, …, 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 $(v_i , v_j) \in E^{SCC}$ 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)

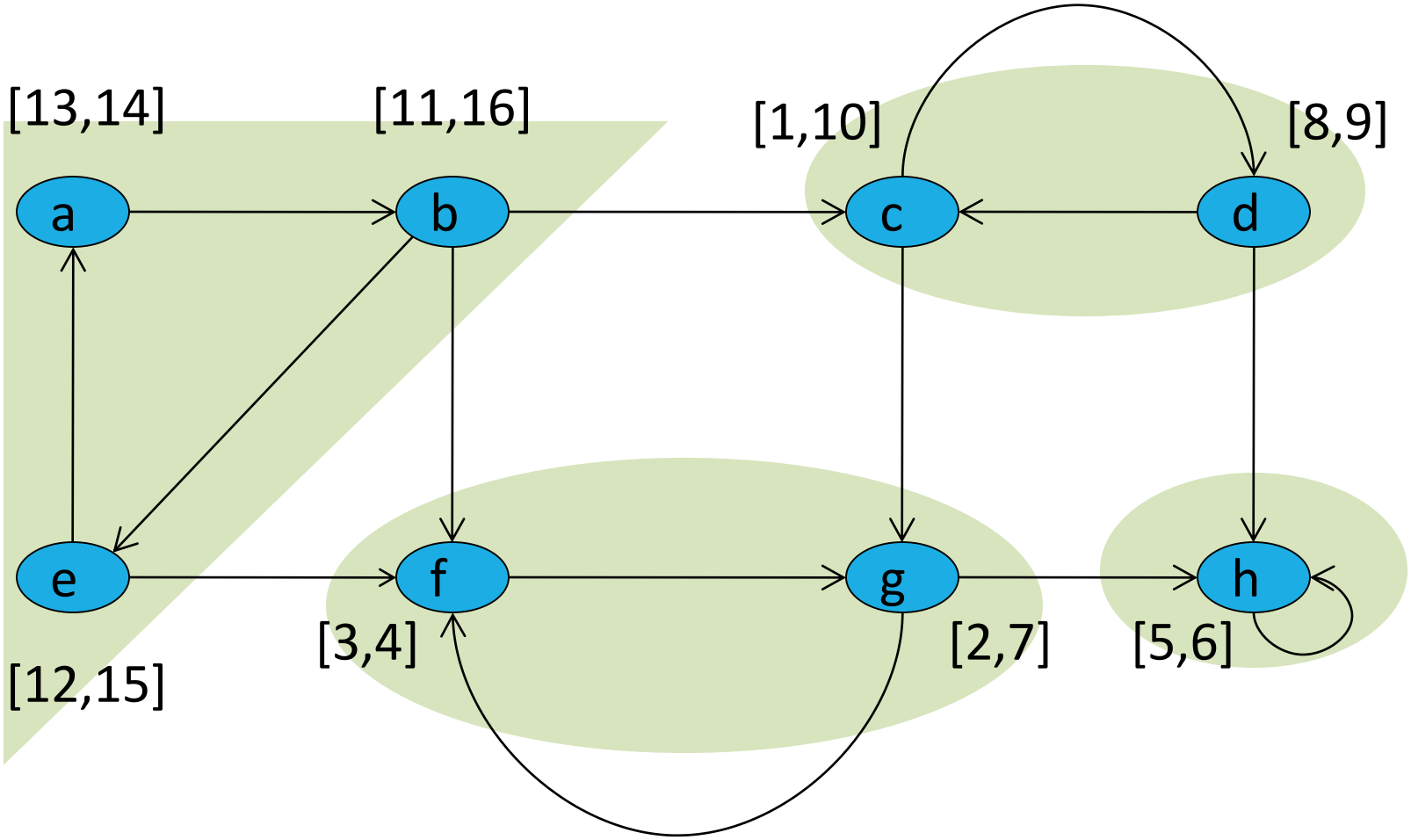

a
b
c
d
e
f
g
h
cd
abe
fg
h

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

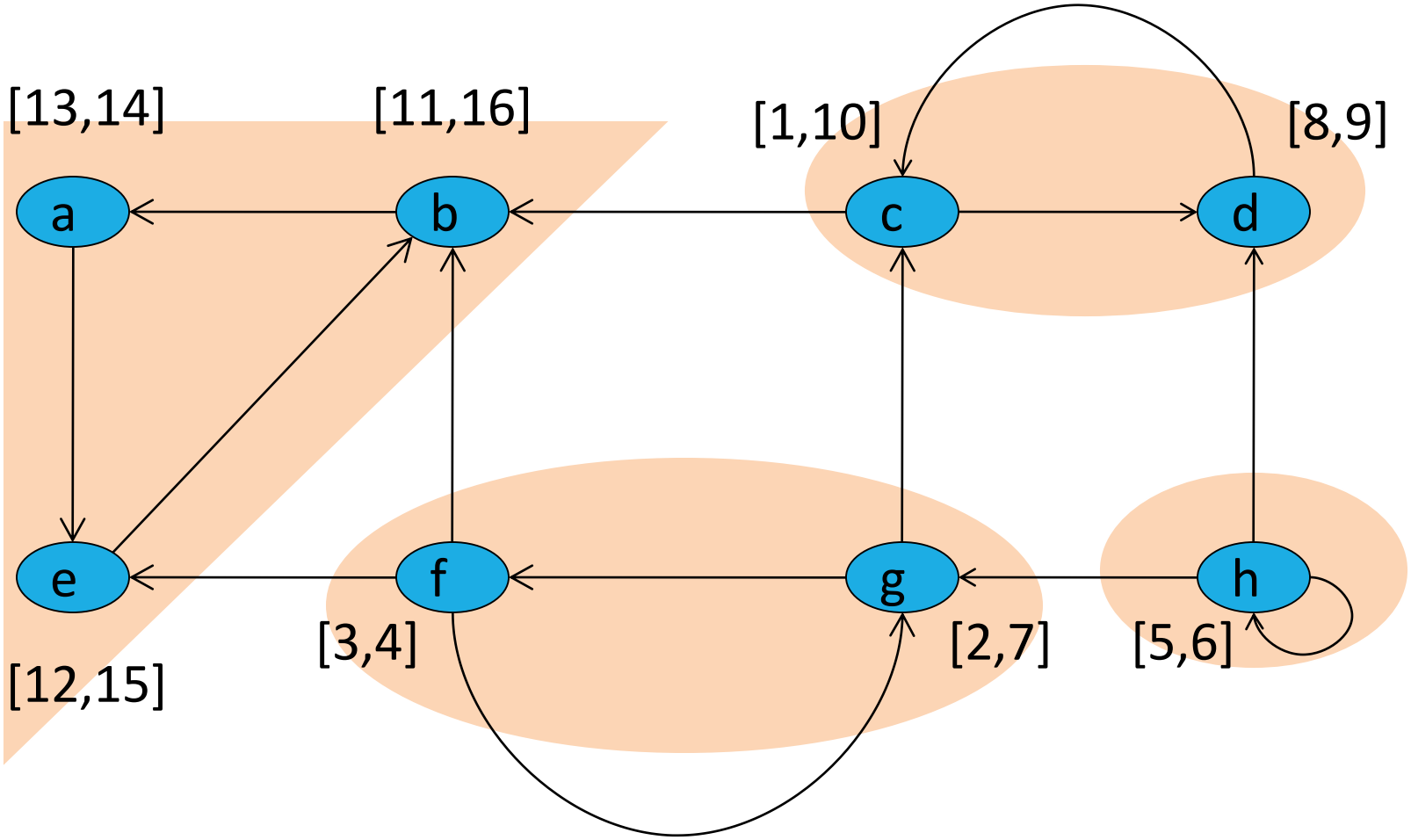

[13,14]
[11,16]
[1,10]
[8,9]
a
b
c
d
e
f
g
h
[12,15]
[3,4]
[2,7]
[5,6]

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 decrecien-te de *u.f*, calculado antes

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

Tres problemas en grafos *con costos*

a) Grafos no direccionales:

- encontrar el *árbol de cobertura de costo mínimo*

b) Grafos direccionales:

- encontrar la *ruta más corta desde un vértice a todos los otros*

c) Grafos direccionales:

- encontrar las *rutas más cortas entre todos los pares de vértices*