DFS y aplicaciones

Clase 21

IIC 2133 - Sección 2

Prof. Yadran Eterovic

Sumario

Detección de ciclos

```
isCyclic(G):
    for X ∈ V(G) :
        if X.color ≠ blanco :
            continue
        if CycleAfter(G, X) :
            return true
    return false
```

```
cycleAfter(G, X):
    if X.color = gris : return true
    if X.color = negro : return false
    X.color ← gris
    for Y tal que X → Y :
        if cycleAfter(G, Y) : return true
    X.color ← negro
    return false
```

Los colores representan el estado de los nodos

Detección de ciclos

```
isCyclic(G):
    for X ∈ V(G) :
        if X.color ≠ blanco :
            continue
        if CycleAfter(G, X) :
            return true
    return false
```

```
cycleAfter(G, X):
    if X.color = gris : return true
    if X.color = negro : return false
    X.color ← gris
    for Y tal que X → Y :
        if cycleAfter(G, Y) : return true
    X.color ← negro
    return false
```

Los colores representan el estado de los nodos

- blanco: no visitado

Detección de ciclos

```
isCyclic(G):
    for X ∈ V(G) :
        if X.color ≠ blanco :
            continue
        if CycleAfter(G, X) :
            return true
    return false
```

```
cycleAfter(G, X):
    if X.color = gris : return true
    if X.color = negro : return false
    X.color ← gris
    for Y tal que X → Y :
        if cycleAfter(G, Y) : return true
    X.color ← negro
    return false
```

Los colores representan el estado de los nodos

- blanco: no visitado
- gris: visitado y anterior al nodo actual visitado

Detección de ciclos

```
isCyclic(G):
    for X ∈ V(G) :
        if X.color ≠ blanco :
            continue
        if CycleAfter(G, X) :
            return true
    return false
```

```
cycleAfter(G, X):
    if X.color = gris : return true
    if X.color = negro : return false
    X.color ← gris
    for Y tal que X → Y :
        if cycleAfter(G, Y) : return true
    X.color ← negro
    return false
```

Los colores representan el estado de los nodos

- blanco: no visitado
- gris: visitado y anterior al nodo actual visitado
- negro: visitado y no anterior al nodo actual visitado

Detección de ciclos

```
isCyclic(G):
    for X ∈ V(G) :
        if X.color ≠ blanco :
            continue
        if CycleAfter(G, X) :
            return true
    return false
```

```
cycleAfter(G, X):
    if X.color = gris : return true
    if X.color = negro : return false
    X.color ← gris
    for Y tal que X → Y :
        if cycleAfter(G, Y) : return true
    X.color ← negro
    return false
```

Los colores representan el estado de los nodos

- blanco: no visitado
- gris: visitado y anterior al nodo actual visitado
- negro: visitado y no anterior al nodo actual visitado

Este algoritmo explora el grafo en profundidad

Búsqueda en profundidad

El algoritmo `isCyclic` sigue una estrategia de **búsqueda en profundidad**

- también abreviada **DFS** por las siglas de *Depth First Search*
- la estrategia es avanzar por aristas adyacentes hasta más no poder
- luego de agotar esa ruta, se retrocede y se exploran otras
- todas las rutas se exploran **en profundidad**

Búsqueda en profundidad

El algoritmo `isCyclic` sigue una estrategia de **búsqueda en profundidad**

- también abreviada **DFS** por las siglas de *Depth First Search*
- la estrategia es avanzar por aristas adyacentes hasta más no poder
- luego de agotar esa ruta, se retrocede y se exploran otras
- todas las rutas se exploran **en profundidad**

Podemos definir un algoritmo que recorre el grafo siguiendo esta idea

Algoritmo DFS básico

Algoritmo DFS básico

Queremos un algoritmo que solo recorre el grafo

Algoritmo DFS básico

Queremos un algoritmo que solo recorre el grafo

- la idea es no visitar nodos ya visitados

Algoritmo DFS básico

Queremos un algoritmo que solo recorre el grafo

- la idea es no visitar nodos ya visitados
- su único efecto en el grafo es cambiar colores

Algoritmo DFS básico

Queremos un algoritmo que solo recorre el grafo

- la idea es no visitar nodos ya visitados
- su único efecto en el grafo es cambiar colores
- lo usaremos como base para nuevos algoritmos

Algoritmo DFS básico

Queremos un algoritmo que solo recorre el grafo

- la idea es no visitar nodos ya visitados
- su único efecto en el grafo es cambiar colores
- lo usaremos como base para nuevos algoritmos

Usaremos el código de colores, dándoles una interpretación más general

Algoritmo DFS básico

Queremos un algoritmo que solo recorre el grafo

- la idea es no visitar nodos ya visitados
- su único efecto en el grafo es cambiar colores
- lo usaremos como base para nuevos algoritmos

Usaremos el código de colores, dándoles una interpretación más general

- blanco: no visitado

Algoritmo DFS básico

Queremos un algoritmo que solo recorre el grafo

- la idea es no visitar nodos ya visitados
- su único efecto en el grafo es cambiar colores
- lo usaremos como base para nuevos algoritmos

Usaremos el código de colores, dándoles una interpretación más general

- blanco: no visitado
- gris: visitado y tiene vecinos no visitados (*está en proceso*)

Algoritmo DFS básico

Queremos un algoritmo que solo recorre el grafo

- la idea es no visitar nodos ya visitados
- su único efecto en el grafo es cambiar colores
- lo usaremos como base para nuevos algoritmos

Usaremos el código de colores, dándoles una interpretación más general

- blanco: no visitado
- gris: visitado y tiene vecinos no visitados (*está en proceso*)
- negro: visitado y todos sus vecinos fueron visitados (*está terminado*)

Algoritmo DFS básico

```
input  : grafo G

Dfs(G):
    for u ∈ V(G) :
        u.color ← blanco
    for u ∈ V(G) :
        if u.color = blanco :
            DfsVisit(G, u)
```

Algoritmo DFS básico

input : grafo G

`Dfs`(G)**:**

  for $u \in V(G)$ **:**
    $u.color \leftarrow$ blanco
  for $u \in V(G)$ **:**
    if $u.color = blanco$ **:**
      `DfsVisit`(G, u)

input : grafo G, nodo $u \in V(G)$

`DfsVisit`(G, u)**:**

  $u.color \leftarrow$ gris
  for $v \in N_G(u)$ **:**
    if $v.color = blanco$ **:**
      `DfsVisit`(G, v)
  $u.color \leftarrow$ negro

donde $N_G(u)$ son los vecinos de u (nodos apuntados por aristas desde u)

Algoritmo DFS básico

input : grafo G

`Dfs`(G):

  for $u \in V(G)$ **:**

    $u.color \leftarrow$ blanco

  for $u \in V(G)$ **:**

    if $u.color = blanco$ **:**

      `DfsVisit`(G, u)

input : grafo G, nodo $u \in V(G)$

`DfsVisit`(G, u):

  $u.color \leftarrow$ gris

  for $v \in N_G(u)$ **:**

    if $v.color = blanco$ **:**

      `DfsVisit`(G, v)

  $u.color \leftarrow$ negro

donde $N_G(u)$ son los vecinos de u (nodos apuntados por aristas desde u)

Este algoritmo solo hace recursión cuando el nodo siguiente no ha sido visitado

Complejidad de DFS básico

Complejidad de DFS básico

input : grafo G

`Dfs`(G):

  for $u \in V(G)$ **:**

    $u.color \leftarrow$ blanco

  for $u \in V(G)$ **:**

    if $u.color = blanco$ **:**

      `DfsVisit`(G, u)

input : grafo G, nodo $u \in V(G)$

`DfsVisit`(G, u):

  $u.color \leftarrow$ gris

  for $v \in N_G(u)$ **:**

    if $v.color = blanco$ **:**

      `DfsVisit`(G, v)

  $u.color \leftarrow$ negro

Para ambas implementaciones de N_G (matriz o listas de adyacencias)

Complejidad de DFS básico

```
input : grafo G

Dfs(G):
    for u ∈ V(G) :
        u.color ← blanco
    for u ∈ V(G) :
        if u.color = blanco :
            DfsVisit(G, u)
```

```
input : grafo G, nodo u ∈ V(G)

DfsVisit(G, u):
    u.color ← gris
    for v ∈ N_G(u) :
        if v.color = blanco :
            DfsVisit(G, v)
    u.color ← negro
```

Para ambas implementaciones de N_G (matriz o listas de adyacencias)

- cada nodo solo inicializa blanco $\mathcal{O}(V)$

Complejidad de DFS básico

input : grafo G

```
Dfs(G):
    for u ∈ V(G) :
        u.color ← blanco
    for u ∈ V(G) :
        if u.color = blanco :
            DfsVisit(G, u)
```

input : grafo G, nodo $u \in V(G)$

```
DfsVisit(G, u):
    u.color ← gris
    for v ∈ N_G(u) :
        if v.color = blanco :
            DfsVisit(G, v)
    u.color ← negro
```

Para ambas implementaciones de N_G (matriz o listas de adyacencias)

- cada nodo solo inicializa blanco $\mathcal{O}(V)$
- cada nodo solo se visita (colorea gris) una vez $\mathcal{O}(V)$

Complejidad de DFS básico

input : grafo G

```
Dfs(G):
    for u ∈ V(G) :
        u.color ← blanco
    for u ∈ V(G) :
        if u.color = blanco :
            DfsVisit(G, u)
```

input : grafo G, nodo $u \in V(G)$

```
DfsVisit(G, u):
    u.color ← gris
    for v ∈ N_G(u) :
        if v.color = blanco :
            DfsVisit(G, v)
    u.color ← negro
```

Para ambas implementaciones de N_G (matriz o listas de adyacencias)

- cada nodo solo inicializa blanco $\mathcal{O}(V)$
- cada nodo solo se visita (colorea gris) una vez $\mathcal{O}(V)$
- cada arista se *recorre* una vez al chequear el vecino $\mathcal{O}(E)$

Complejidad de DFS básico

input : grafo G

$\texttt{Dfs}(G)$:

    for $u \in V(G)$ **:**
        $u.color \leftarrow$ blanco
    for $u \in V(G)$ **:**
        if $u.color = blanco$ **:**
            $\texttt{DfsVisit}(G, u)$

input : grafo G, nodo $u \in V(G)$

$\texttt{DfsVisit}(G, u)$:

    $u.color \leftarrow$ gris
    for $v \in N_G(u)$ **:**
        if $v.color = blanco$ **:**
            $\texttt{DfsVisit}(G, v)$
    $u.color \leftarrow$ negro

Para ambas implementaciones de N_G (matriz o listas de adyacencias)

- cada nodo solo inicializa blanco $\mathcal{O}(V)$
- cada nodo solo se visita (colorea gris) una vez $\mathcal{O}(V)$
- cada arista se *recorre* una vez al chequear el vecino $\mathcal{O}(E)$

El algoritmo DFS toma tiempo $\mathcal{O}(V + E)$, i.e. es lineal en $|G|$

Búsqueda en profundidad

Mencionamos que `Dfs` será la base de otros algoritmos

Búsqueda en profundidad

Mencionamos que `Dfs` será la base de otros algoritmos

- ya sabemos recorrer los nodos sin repetir

Búsqueda en profundidad

Mencionamos que `Dfs` será la base de otros algoritmos

- ya sabemos recorrer los nodos sin repetir
- además visitando/descubriendo todas las aristas

Búsqueda en profundidad

Mencionamos que `Dfs` será la base de otros algoritmos

- ya sabemos recorrer los nodos sin repetir
- además visitando/descubriendo todas las aristas

Agregaremos más información para poder extender el algoritmo base

Objetivos de la clase

Objetivos de la clase

- ☐ Comprender el recorrido DFS de grafos dirigidos

Objetivos de la clase

- ☐ Comprender el recorrido DFS de grafos dirigidos
- ☐ Comprender el algoritmo de orden topológico en grafos acíclicos

Objetivos de la clase

- ☐ Comprender el recorrido DFS de grafos dirigidos
- ☐ Comprender el algoritmo de orden topológico en grafos acíclicos
- ☐ Comprender el algoritmo de Kosaraju para componentes conexas

Sumario

Hacia el orden de los nodos

Hacia el orden de los nodos

Además de usar colores, *recordaremos* cuándo se hicieron esos cambios

Hacia el orden de los nodos

Además de usar colores, *recordaremos* cuándo se hicieron esos cambios

- cuando se visita un nodo, se pinta **gris**

Hacia el orden de los nodos

Además de usar colores, *recordaremos* cuándo se hicieron esos cambios

- cuando se visita un nodo, se pinta **gris**
- además definiremos un **tiempo de descubrimiento o inicio**

Hacia el orden de los nodos

Además de usar colores, *recordaremos* cuándo se hicieron esos cambios

- cuando se visita un nodo, se pinta **gris**
- además definiremos un **tiempo de descubrimiento o inicio**
- cuando se completa un nodo, se pinta **negro**

Hacia el orden de los nodos

Además de usar colores, *recordaremos* cuándo se hicieron esos cambios

- cuando se visita un nodo, se pinta **gris**
- además definiremos un **tiempo de descubrimiento o inicio**
- cuando se completa un nodo, se pinta **negro**
- además definiremos un **tiempo de término o finalización**

Hacia el orden de los nodos

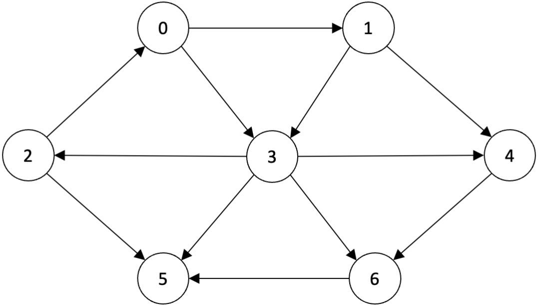
Además de usar colores, *recordaremos* cuándo se hicieron esos cambios

- cuando se visita un nodo, se pinta **gris**
- además definiremos un **tiempo de descubrimiento o inicio**
- cuando se completa un nodo, se pinta **negro**
- además definiremos un **tiempo de término o finalización**

Los tiempos de inicio y término serán números $1, 2, \ldots$ correlativos

DFS con tiempos de inicio y término

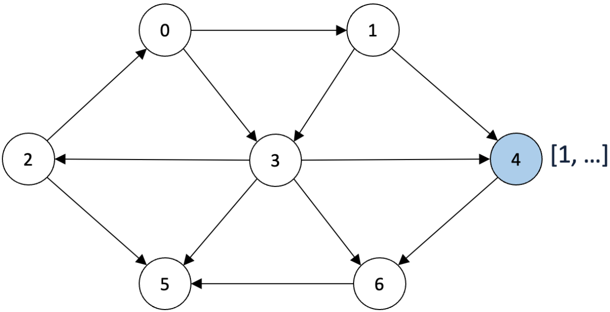
Comenzamos el recorrido usando $\texttt{DfsVisit}(G, 4)$



Tengamos presente cómo sacar conclusiones usando los tiempos

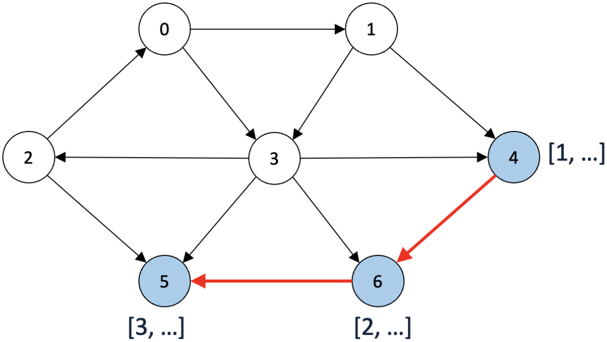
DFS con tiempos de inicio y término

Seteamos los tiempos para $u = 4$ al descubrirlo por primera vez



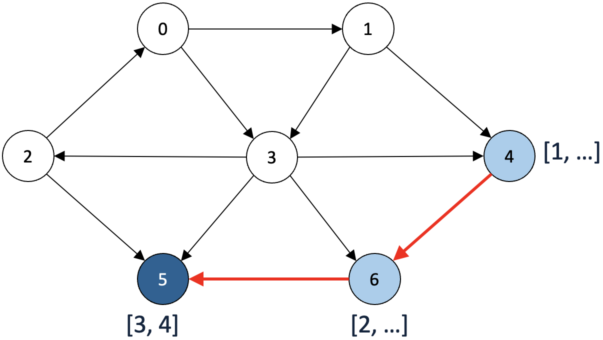
DFS con tiempos de inicio y término

Avanzamos por las aristas hasta agotarlas, llegando al nodo 5



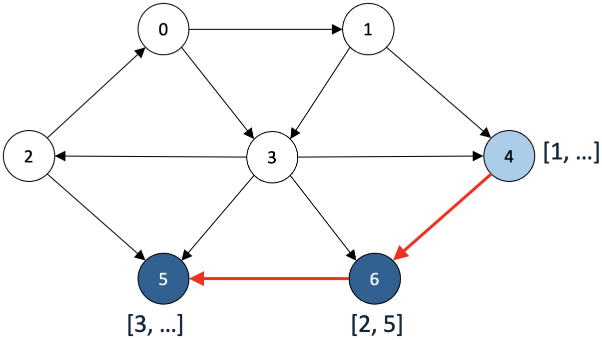
DFS con tiempos de inicio y término

Como $u = 5$ no tiene vecinos por visitar, lo terminamos y seteamos su tiempo de término



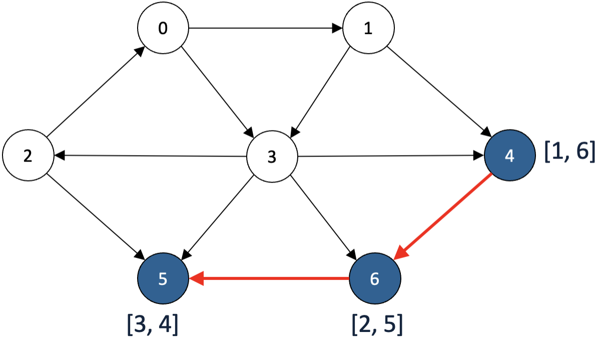
DFS con tiempos de inicio y término

El nodo $u = 6$ tampoco tiene vecinos por visitar y lo terminamos



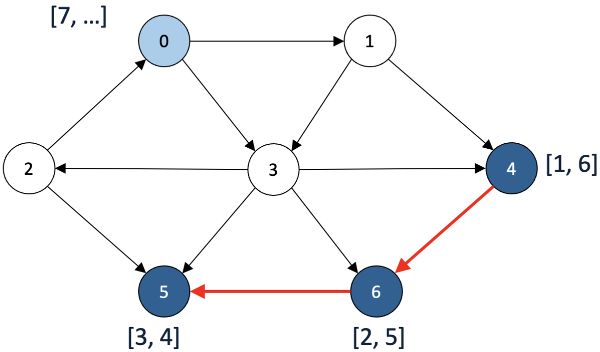
DFS con tiempos de inicio y término

Concluimos el llamado a $u = 4$ por no quedar vecinos por visitar



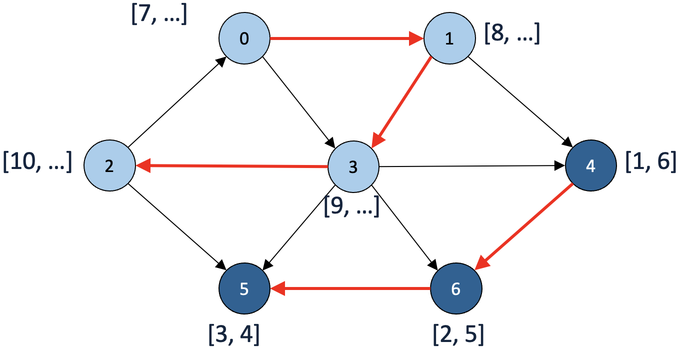
DFS con tiempos de inicio y término

Ahora hacemos llamado a `Dfs`$(G, 0)$ (los tiempos son correlativos, toca tiempo 7)



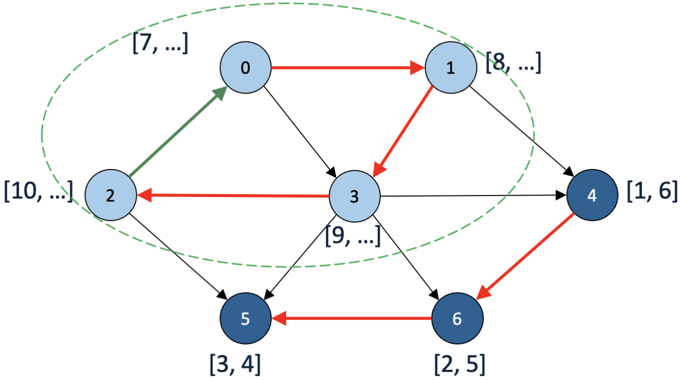
DFS con tiempos de inicio y término

Avanzamos por las aristas $(0, 1)$, $(1, 3)$ y $(3, 2)$



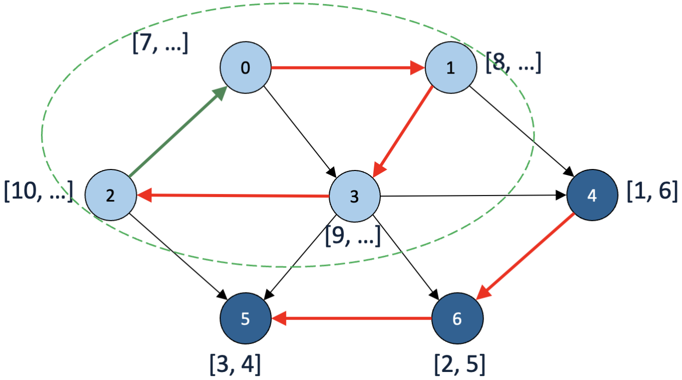
DFS con tiempos de inicio y término

La arista $(2, 0)$ apunta a un nodo **gris**: detectamos un ciclo



DFS con tiempos de inicio y término

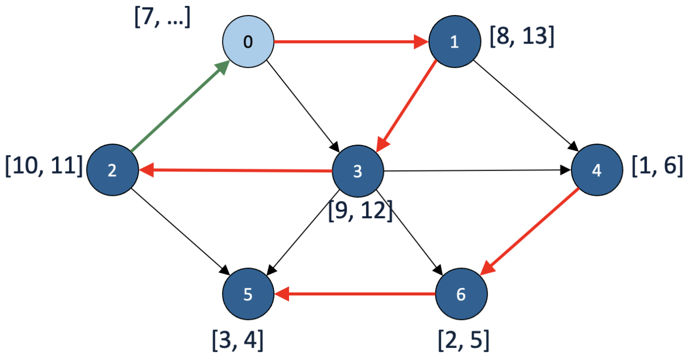
La arista $(2, 0)$ apunta a un nodo **gris**: detectamos un ciclo



¿Necesitamos realmente el color?

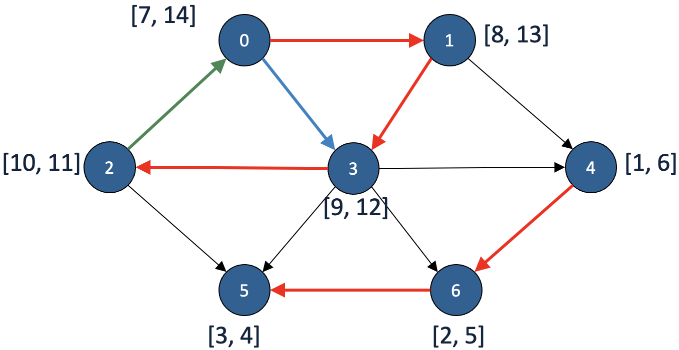
DFS con tiempos de inicio y término

Las otras aristas apuntan a nodos ya terminados y retrocedemos hasta $u = 0$



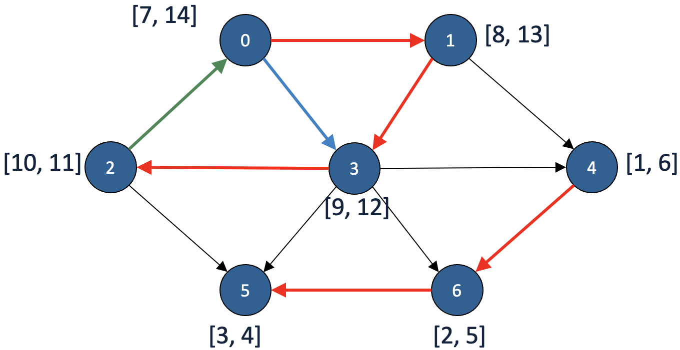
DFS con tiempos de inicio y término

Desde $u = 0$ no quedan más nodos por descubrir y terminamos



DFS con tiempos de inicio y término

Desde $u = 0$ no quedan más nodos por descubrir y terminamos



¿Qué diferencia la arista $(0, 3)$ de la $(3, 6)$?

DFS con tiempos de inicio y término

input : grafo G

```
Dfs(G):
    t ← 1
    for u ∈ V(G) :
        u.start ← 0
        u.end ← 0
    for u ∈ V(G) :
        if u.start = 0 :
            t ← DfsVisit(G, u, t)
```

input : grafo G, nodo $u \in V(G)$, tiempo t

output: tiempo $t \geq 1$

```
DfsVisit(G, u, t):
    u.start ← t
    t ← t + 1
    for v ∈ N_G(u) :
        if v.start = 0 :
            t ← DfsVisit(G, v, t)
    u.end ← t
    t ← t + 1
    return t
```

DFS con tiempos de inicio y término

input : grafo G

$\texttt{Dfs}(G)$**:**

$t \leftarrow 1$

for $u \in V(G)$ **:**

  $u.start \leftarrow 0$

  $u.end \leftarrow 0$

for $u \in V(G)$ **:**

  if $u.start = 0$ **:**

    $t \leftarrow \texttt{DfsVisit}(G, u, t)$

input : grafo G, nodo $u \in V(G)$, tiempo t

output: tiempo $t \geq 1$

$\texttt{DfsVisit}(G, u, t)$**:**

$u.start \leftarrow t$

$t \leftarrow t + 1$

for $v \in N_G(u)$ **:**

  if $v.start = 0$ **:**

    $t \leftarrow \texttt{DfsVisit}(G, v, t)$

$u.end \leftarrow t$

$t \leftarrow t + 1$

return t

Importante: el recorrido DFS puede generar un **bosque** de árboles independientes

Propiedades de los tiempos

Propiedades de los tiempos

Definición

Dado G y $u, v \in G$, diremos que u es **descendiente** de v al ejecutar $\texttt{Dfs}(G)$ si u es visitado por primera vez luego de v y antes de que v sea terminado. En tal caso diremos que ambos pertenecen al mismo **árbol DFS**

Propiedades de los tiempos

Definición

Dado G y $u, v \in G$, diremos que u es **descendiente** de v al ejecutar $\texttt{Dfs}(G)$ si u es visitado por primera vez luego de v y antes de que v sea terminado. En tal caso diremos que ambos pertenecen al mismo **árbol DFS**

Proposición

Dado G y $u, v \in V(G)$, luego de ejecutar $\texttt{Dfs}(G)$ se definen los intervalos

$$I_u = \{u.start, \ldots, u.end\} \qquad I_v = \{v.start, \ldots, v.end\}$$

Propiedades de los tiempos

Definición

Dado G y $u, v \in G$, diremos que u es **descendiente** de v al ejecutar $\texttt{Dfs}(G)$ si u es visitado por primera vez luego de v y antes de que v sea terminado. En tal caso diremos que ambos pertenecen al mismo **árbol DFS**

Proposición

Dado G y $u, v \in V(G)$, luego de ejecutar $\texttt{Dfs}(G)$ se definen los intervalos

$$I_u = \{u.start, \dots, u.end\} \qquad I_v = \{v.start, \dots, v.end\}$$

Estos intervalos cumplen una de las siguientes propiedades

Propiedades de los tiempos

Definición

Dado G y $u, v \in G$, diremos que u es **descendiente** de v al ejecutar $\texttt{Dfs}(G)$ si u es visitado por primera vez luego de v y antes de que v sea terminado. En tal caso diremos que ambos pertenecen al mismo **árbol DFS**

Proposición

Dado G y $u, v \in V(G)$, luego de ejecutar $\texttt{Dfs}(G)$ se definen los intervalos

$$I_u = \{u.start, \dots, u.end\} \qquad I_v = \{v.start, \dots, v.end\}$$

Estos intervalos cumplen una de las siguientes propiedades

- $I_u \cap I_v = \varnothing$ (no están en el mismo árbol DFS)

Propiedades de los tiempos

Definición

Dado G y $u, v \in G$, diremos que u es **descendiente** de v al ejecutar $\texttt{Dfs}(G)$ si u es visitado por primera vez luego de v y antes de que v sea terminado. En tal caso diremos que ambos pertenecen al mismo **árbol DFS**

Proposición

Dado G y $u, v \in V(G)$, luego de ejecutar $\texttt{Dfs}(G)$ se definen los intervalos

$$I_u = \{u.start, \dots, u.end\} \qquad I_v = \{v.start, \dots, v.end\}$$

Estos intervalos cumplen una de las siguientes propiedades

- $I_u \cap I_v = \varnothing$ (no están en el mismo árbol DFS)
- $I_u \subset I_v$ (u es descendiente de v)

Propiedades de los tiempos

Definición

Dado G y $u, v \in G$, diremos que u es **descendiente** de v al ejecutar $\texttt{Dfs}(G)$ si u es visitado por primera vez luego de v y antes de que v sea terminado. En tal caso diremos que ambos pertenecen al mismo **árbol DFS**

Proposición

Dado G y $u, v \in V(G)$, luego de ejecutar $\texttt{Dfs}(G)$ se definen los intervalos

$$I_u = \{u.start, \ldots, u.end\} \qquad I_v = \{v.start, \ldots, v.end\}$$

Estos intervalos cumplen una de las siguientes propiedades

- $I_u \cap I_v = \varnothing$ (no están en el mismo árbol DFS)
- $I_u \subset I_v$ (u es descendiente de v)
- $I_v \subset I_u$ (v es descendiente de u)

Tipos de aristas en DFS

Tipos de aristas en DFS

Estas convenciones permiten distinguir aristas al recorrer con DFS

Tipos de aristas en DFS

Estas convenciones permiten distinguir aristas al recorrer con DFS

Definición

Dado G y $u, v \in G$, luego de ejecutar $\texttt{Dfs}(G)$ la arista (u, v) puede ser

Tipos de aristas en DFS

Estas convenciones permiten distinguir aristas al recorrer con DFS

Definición

Dado G y $u, v \in G$, luego de ejecutar $\texttt{Dfs}(G)$ la arista (u, v) puede ser

- **Arista de árbol** si v fue descubierto al transitar (u, v)

Tipos de aristas en DFS

Estas convenciones permiten distinguir aristas al recorrer con DFS

Definición

Dado G y $u, v \in G$, luego de ejecutar $\texttt{Dfs}(G)$ la arista (u, v) puede ser

- **Arista de árbol** si v fue descubierto al transitar (u, v)
- **Arista hacia atrás** si u es descendiente de v en un árbol DFS

Tipos de aristas en DFS

Estas convenciones permiten distinguir aristas al recorrer con DFS

Definición

Dado G y $u, v \in G$, luego de ejecutar $\texttt{Dfs}(G)$ la arista (u, v) puede ser

- **Arista de árbol** si v fue descubierto al transitar (u, v)
- **Arista hacia atrás** si u es descendiente de v en un árbol DFS
- **Arista hacia adelante** si (u, v) no es de árbol y v es descendiente de u

Tipos de aristas en DFS

Estas convenciones permiten distinguir aristas al recorrer con DFS

Definición

Dado G y $u, v \in G$, luego de ejecutar $\texttt{Dfs}(G)$ la arista (u, v) puede ser

- **Arista de árbol** si v fue descubierto al transitar (u, v)
- **Arista hacia atrás** si u es descendiente de v en un árbol DFS
- **Arista hacia adelante** si (u, v) no es de árbol y v es descendiente de u
- **Arista cruzada** en otro caso

Tipos de aristas en DFS

Estas convenciones permiten distinguir aristas al recorrer con DFS

Definición

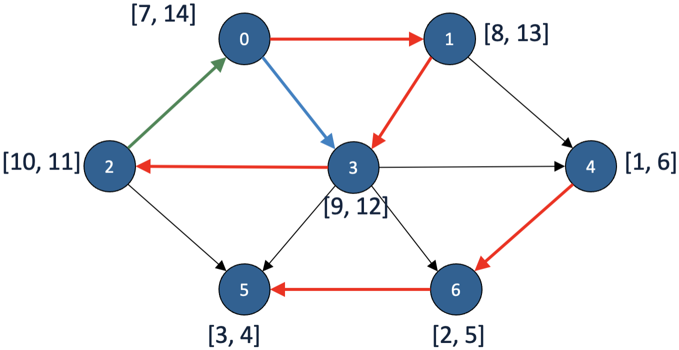
Dado G y $u, v \in G$, luego de ejecutar $\texttt{Dfs}(G)$ la arista (u, v) puede ser

- **Arista de árbol** si v fue descubierto al transitar (u, v)
- **Arista hacia atrás** si u es descendiente de v en un árbol DFS
- **Arista hacia adelante** si (u, v) no es de árbol y v es descendiente de u
- **Arista cruzada** en otro caso

Proposición

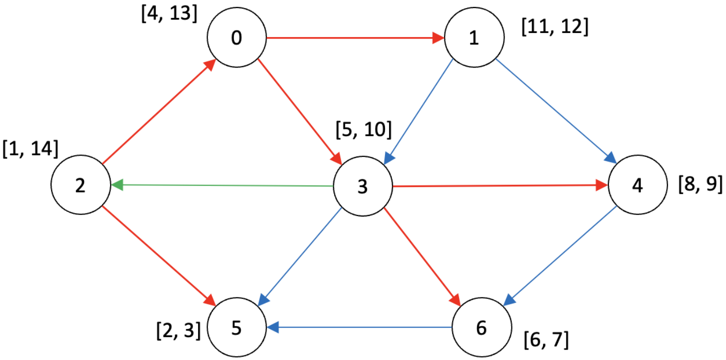
Un grafo G es acíclico si, y solo si, $\texttt{Dfs}(G)$ no produce aristas hacia atrás

Tipos de aristas en DFS



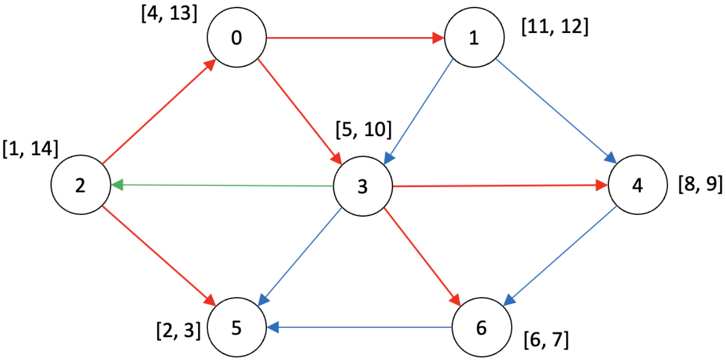
Tipos de aristas en DFS

Notemos que la proposición no depende del orden que se escogen los nodos para llamar `DfsVisit` en los llamados recursivos



Tipos de aristas en DFS

Notemos que la proposición no depende del orden que se escogen los nodos para llamar `DfsVisit` en los llamados recursivos



Comenzando el recorrido en $u = 2$ también detecta el ciclo, formando un solo árbol DFS

Sumario

Orden topológico

Orden topológico

Definición

Sea G un grafo dirigido. Un **orden topológico** de G es una secuencia de sus nodos

$$v_0, v_1, \ldots, v_n \qquad v_i \in V(G)$$

tal que

Orden topológico

Definición

Sea G un grafo dirigido. Un **orden topológico** de G es una secuencia de sus nodos

$$v_0, v_1, \ldots, v_n \qquad v_i \in V(G)$$

tal que

- todo nodo $u \in V(G)$ aparece en la secuencia

Orden topológico

Definición

Sea G un grafo dirigido. Un **orden topológico** de G es una secuencia de sus nodos

$$v_0, v_1, \ldots, v_n \qquad v_i \in V(G)$$

tal que

- todo nodo $u \in V(G)$ aparece en la secuencia
- no hay elementos repetidos en la secuencia

Orden topológico

Definición

Sea G un grafo dirigido. Un **orden topológico** de G es una secuencia de sus nodos

$$v_0, v_1, \ldots, v_n \qquad v_i \in V(G)$$

tal que

- todo nodo $u \in V(G)$ aparece en la secuencia
- no hay elementos repetidos en la secuencia
- si $(v_i, v_j) \in E(G)$, entonces v_i aparece antes que v_j en la secuencia

Orden topológico

Definición

Sea G un grafo dirigido. Un **orden topológico** de G es una secuencia de sus nodos

$$v_0, v_1, \ldots, v_n \qquad v_i \in V(G)$$

tal que

- todo nodo $u \in V(G)$ aparece en la secuencia
- no hay elementos repetidos en la secuencia
- si $(v_i, v_j) \in E(G)$, entonces v_i aparece antes que v_j en la secuencia

Proposición

Si G es cíclico, entonces no existe un orden topológico

Orden topológico

Definición

Sea G un grafo dirigido. Un **orden topológico** de G es una secuencia de sus nodos

$$v_0, v_1, \ldots, v_n \qquad v_i \in V(G)$$

tal que

- todo nodo $u \in V(G)$ aparece en la secuencia
- no hay elementos repetidos en la secuencia
- si $(v_i, v_j) \in E(G)$, entonces v_i aparece antes que v_j en la secuencia

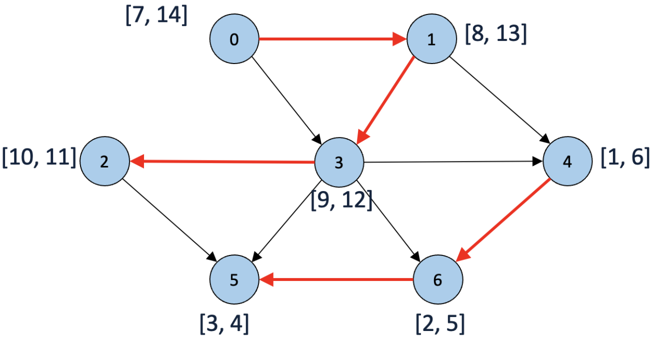
Proposición

Si G es cíclico, entonces no existe un orden topológico

Podemos usar `Dfs` para construir un orden topológico de un grafo acíclico

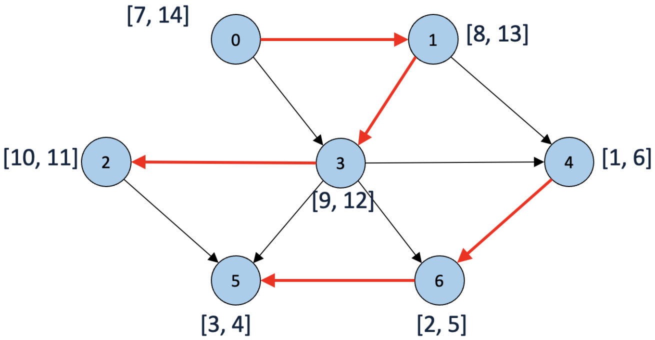
Orden topológico

Consideremos el siguiente bosque de árboles DFS sobre un grafo acíclico



¿Qué orden topológico sugiere este bosque?

Orden topológico



Deducimos el orden topológico de G dado por

0 1 3 2 4 6 5

Orden topológico

input : grafo G
output: lista L de nodos en orden topológico

```
TopSort(G):
    L ← lista vacía
    Dfs(G)
    Insertar en L nodos en orden decreciente según end
    return L
```

Orden topológico

input : grafo G
output: lista L de nodos en orden topológico

```
TopSort(G):
1     L ← lista vacía
2     Dfs(G)
3     Insertar en L nodos en orden decreciente según end
4     return L
```

¿Podemos modificar `Dfs` para no tener que ordenar en la línea 3?

Orden topológico

input : grafo G
output: lista de nodos L

`TopSort`(G)**:**

$L \leftarrow$ lista vacía
$t \leftarrow 1$
for $u \in V(G)$ **:**
  $u.start \leftarrow 0$
  $u.end \leftarrow 0$
for $u \in V(G)$ **:**
  if $u.start = 0$ **:**
    $t \leftarrow$ `TopDfsVisit`(G, L, u, t)
return L

input : grafo G, lista de nodos L, nodo $u \in V(G)$, tiempo t
output: tiempo $t \geq 1$

`TopDfsVisit`(G, L, u, t)**:**

$u.start \leftarrow t$
$t \leftarrow t + 1$
for $v \in N_G(u)$ **:**
  if $v.start = 0$ **:**
    $t \leftarrow$ `TopDfsVisit`(G, L, v, t)
$u.end \leftarrow t$
Insertar u como cabeza de L
$t \leftarrow t + 1$
return t

Orden topológico

input : grafo G
output: lista de nodos L

`TopSort`(G)**:**

- $L \leftarrow$ lista vacía
- $t \leftarrow 1$
- **for** $u \in V(G)$ **:**
  - $u.start \leftarrow 0$
  - $u.end \leftarrow 0$
- **for** $u \in V(G)$ **:**
  - **if** $u.start = 0$ **:**
    - $t \leftarrow$ `TopDfsVisit`(G, L, u, t)
- **return** L

input : grafo G, lista de nodos L, nodo $u \in V(G)$, tiempo t
output: tiempo $t \geq 1$

`TopDfsVisit`(G, L, u, t)**:**

- $u.start \leftarrow t$
- $t \leftarrow t + 1$
- **for** $v \in N_G(u)$ **:**
  - **if** $v.start = 0$ **:**
    - $t \leftarrow$ `TopDfsVisit`(G, L, v, t)
- $u.end \leftarrow t$
- Insertar u como cabeza de L
- $t \leftarrow t + 1$
- **return** t

Al igual que `Dfs`, este algoritmo es $\mathcal{O}(V + E)$

Sumario

Componentes fuertemente conectadas

Componentes fuertemente conectadas

Definición

Sea G un grafo dirigido. Una **componente fuertemente conectada (CFC)** es un conjunto maximal de nodos $C \subseteq V(G)$ tal que dados $u, v \in C$, existe un camino dirigido desde u hasta v

Componentes fuertemente conectadas

Definición

Sea G un grafo dirigido. Una **componente fuertemente conectada (CFC)** es un conjunto maximal de nodos $C \subseteq V(G)$ tal que dados $u, v \in C$, existe un camino dirigido desde u hasta v

Proposición

Si G es cíclico y los nodos de $B \subseteq V(G)$ forman un ciclo, entonces existe una componente fuertemente conectada C tal que $B \subseteq C$

Componentes fuertemente conectadas

Definición

Sea G un grafo dirigido. Una **componente fuertemente conectada (CFC)** es un conjunto maximal de nodos $C \subseteq V(G)$ tal que dados $u, v \in C$, existe un camino dirigido desde u hasta v

Proposición

Si G es cíclico y los nodos de $B \subseteq V(G)$ forman un ciclo, entonces existe una componente fuertemente conectada C tal que $B \subseteq C$

Los nodos de un ciclo pertenecen a la misma CFC

Componentes fuertemente conectadas

Definición

Sea G un grafo dirigido. Una **componente fuertemente conectada (CFC)** es un conjunto maximal de nodos $C \subseteq V(G)$ tal que dados $u, v \in C$, existe un camino dirigido desde u hasta v

Proposición

Si G es cíclico y los nodos de $B \subseteq V(G)$ forman un ciclo, entonces existe una componente fuertemente conectada C tal que $B \subseteq C$

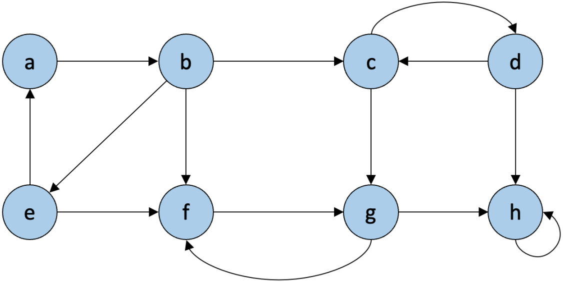
Los nodos de un ciclo pertenecen a la misma CFC

Proposición

Un grafo G acíclico tiene 0 componentes fuertemente conectadas

Componentes fuertemente conectadas

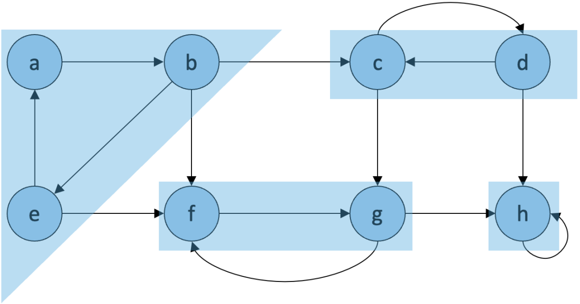
Consideremos el siguiente grafo dirigido cíclico



¿Cuáles son las componentes fuertemente conectadas de G?

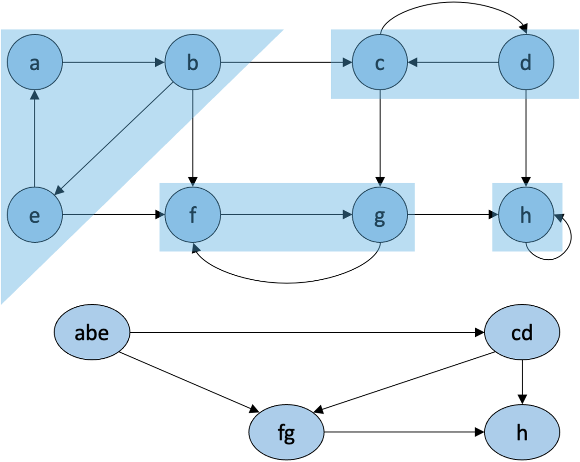
Componentes fuertemente conectadas

Existen 4 CFC's en el grafo anterior



Notemos que es necesario poder *ir y volver* dentro de una CFC

Componentes fuertemente conectadas



Cada componente tiene un **representante** que combina sus nodos

Grafo transpuesto

Para proponer un algoritmo, necesitamos un grafo nuevo

Grafo transpuesto

Para proponer un algoritmo, necesitamos un grafo nuevo

Definición

Sea G un grafo dirigido. Decimos que G^T es el grafo **transpuesto** de G si

Grafo transpuesto

Para proponer un algoritmo, necesitamos un grafo nuevo

Definición

Sea G un grafo dirigido. Decimos que G^T es el grafo **transpuesto** de G si

- $V(G) = V(G^T)$

Grafo transpuesto

Para proponer un algoritmo, necesitamos un grafo nuevo

Definición

Sea G un grafo dirigido. Decimos que G^T es el grafo **transpuesto** de G si

- $V(G) = V(G^T)$
- $\forall u, v \in V(G).\ (u, v) \in E(G) \to (v, u) \in E(G^T)$

Grafo transpuesto

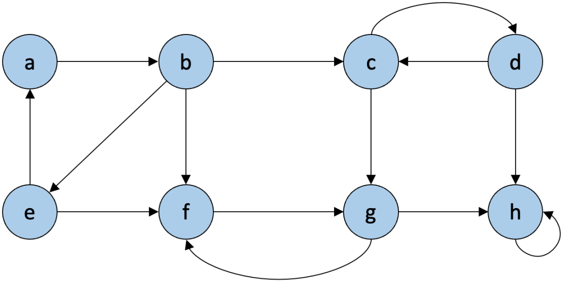
Para proponer un algoritmo, necesitamos un grafo nuevo

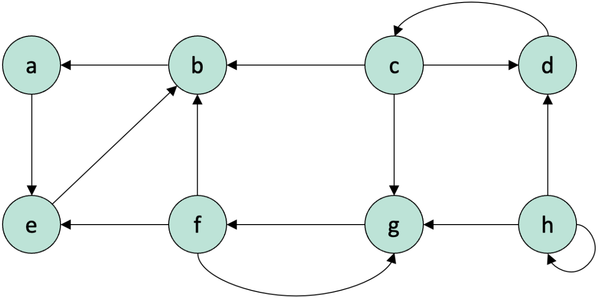
Definición

Sea G un grafo dirigido. Decimos que G^T es el grafo **transpuesto** de G si

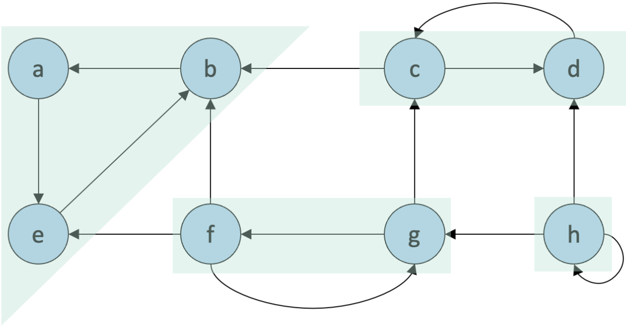
- $V(G) = V(G^T)$
- $\forall u, v \in V(G).\ (u, v) \in E(G) \rightarrow (v, u) \in E(G^T)$

El transpuesto se obtiene invirtiendo todas las aristas de G


a
b
c
d
e
f
g
h


a
b
c
d
e
f
g
h

Grafo transpuesto

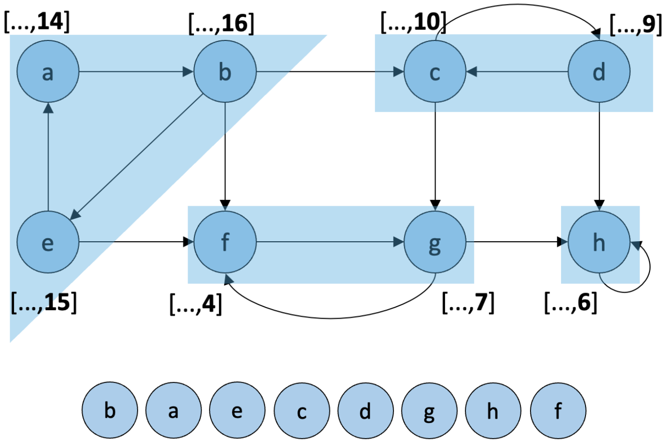


Proposición

Los grafos G y G^T tienen las mismas componentes fuertemente conectadas

Hacia un algoritmo para determinar las CFC

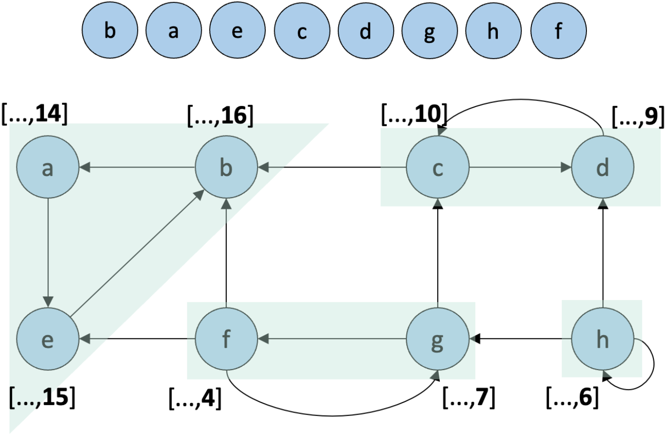
Construimos un orden de los nodos de G según tiempos de término



¡Ojo! Esto no es un orden topológico porque G es cíclico

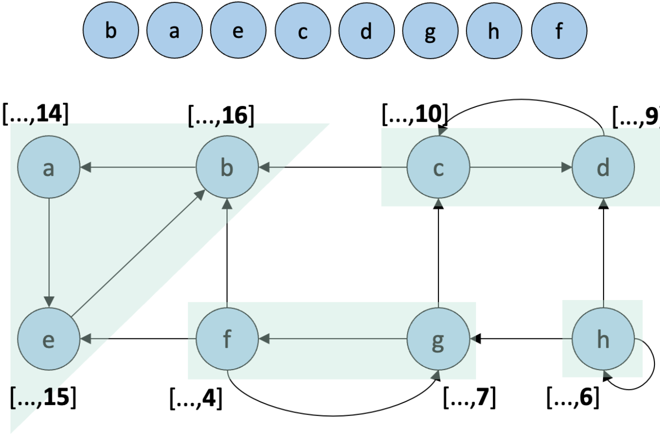
Hacia un algoritmo para determinar las CFC

Recorremos el grafo transpuesto partiendo según el orden anterior



Hacia un algoritmo para determinar las CFC

Recorremos el grafo transpuesto partiendo según el orden anterior



Al transponer, no es posible ir de *b* a *c*
porque están en componentes diferentes

Algoritmo de Kosaraju

input : grafo G

```
Kosaraju(G):
    L ← TopSort(G)
    for u ∈ L :
        Assign(u, u)
```

input : grafo G, nodo $u \in V(G)$, nodo representante r

```
Assign(G, u, r):
    if u.rep = ∅ :
        u.rep ← r
        for v ∈ N_{G^T}(u) :
            Assign(G, v, r)
```

Algoritmo de Kosaraju

input : grafo G

```
Kosaraju(G):
    L ← TopSort(G)
    for u ∈ L :
        Assign(u, u)
```

input : grafo G, nodo $u \in V(G)$, nodo representante r

```
Assign(G, u, r):
    if u.rep = ∅ :
        u.rep ← r
        for v ∈ N_{G^T}(u) :
            Assign(G, v, r)
```

No olvidar: no podemos interpretar L como orden topológico.
Es un orden que se construye de la misma forma

Algoritmo de Kosaraju

El algoritmo de Kosaraju se basa en las propiedades del siguiente grafo

Algoritmo de Kosaraju

El algoritmo de Kosaraju se basa en las propiedades del siguiente grafo

Definición

Dado un grafo G dirigido, sean $C_1, \ldots, C_k$ sus componentes fuertemente conectadas. Se define el **grafo de componentes** G^{CFC} según

Algoritmo de Kosaraju

El algoritmo de Kosaraju se basa en las propiedades del siguiente grafo

Definición

Dado un grafo G dirigido, sean $C_1, \ldots, C_k$ sus componentes fuertemente conectadas. Se define el **grafo de componentes** G^{CFC} según

- $V(G^{CFC}) = \{C_1, \ldots, C_k\}$

Algoritmo de Kosaraju

El algoritmo de Kosaraju se basa en las propiedades del siguiente grafo

Definición

Dado un grafo G dirigido, sean $C_1, \ldots, C_k$ sus componentes fuertemente conectadas. Se define el **grafo de componentes** G^{CFC} según

- $V(G^{CFC}) = \{C_1, \ldots, C_k\}$
- Si $(u, v) \in E(G)$ y $u \in C_i, v \in C_j$, entonces $(C_i, C_j) \in E(G^{CFC})$

Algoritmo de Kosaraju

El algoritmo de Kosaraju se basa en las propiedades del siguiente grafo

Definición

Dado un grafo G dirigido, sean $C_1, \ldots, C_k$ sus componentes fuertemente conectadas. Se define el **grafo de componentes** G^{CFC} según

- $V(G^{CFC}) = \{C_1, \ldots, C_k\}$
- Si $(u, v) \in E(G)$ y $u \in C_i, v \in C_j$, entonces $(C_i, C_j) \in E(G^{CFC})$

Teorema

El grafo de componentes G^{CFC} es un grafo dirigido acíclico

Algoritmo de Kosaraju

El algoritmo de Kosaraju se basa en las propiedades del siguiente grafo

Definición

Dado un grafo G dirigido, sean $C_1, \ldots, C_k$ sus componentes fuertemente conectadas. Se define el **grafo de componentes** G^{CFC} según

- $V(G^{CFC}) = \{C_1, \ldots, C_k\}$
- Si $(u, v) \in E(G)$ y $u \in C_i, v \in C_j$, entonces $(C_i, C_j) \in E(G^{CFC})$

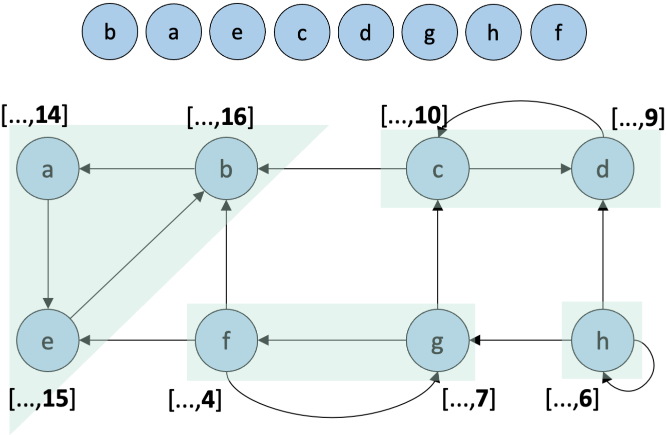
Teorema

El grafo de componentes G^{CFC} es un grafo dirigido acíclico

Corolario

El grafo de componentes G^{CFC} tiene un orden topológico

Hacia un algoritmo para determinar las CFC



La forma en que recorremos las componentes nos da su orden topológico
$(bae)(cd)(gf)(h)$

Sumario

Objetivos de la clase

Objetivos de la clase

- ☐ Comprender el recorrido DFS de grafos dirigidos

Objetivos de la clase

- ☐ Comprender el recorrido DFS de grafos dirigidos
- ☐ Comprender el algoritmo de orden topológico en grafos acíclicos

Objetivos de la clase

- ☐ Comprender el recorrido DFS de grafos dirigidos
- ☐ Comprender el algoritmo de orden topológico en grafos acíclicos
- ☐ Comprender el algoritmo de Kosaraju para componentes conexas