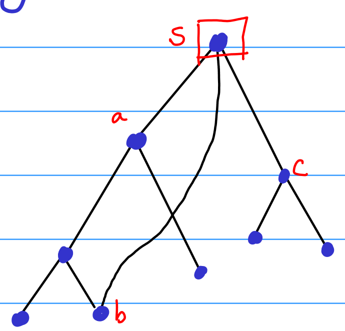
III Depth-first-search (DFS) o búsqueda por profundidad:

Dado un grafo finito G (dirigido o no dirigido) y un vértice inicial s, DFS recorre los vértices de G siguiendo todo camino de salida hasta no encontrar nuevos vértices y en ese momento regresa al último vértice anterior del que aún le quedan caminos por seguir.

Si quisiéramos explorar un laberinto desconocido buscando la salida DFS sería un buen método.

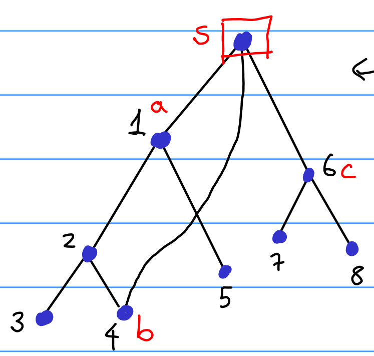
Obs: Si conoces el mapa del laberinto usa BFS para encontrar el camino más corto a la salida

Ejemplo:



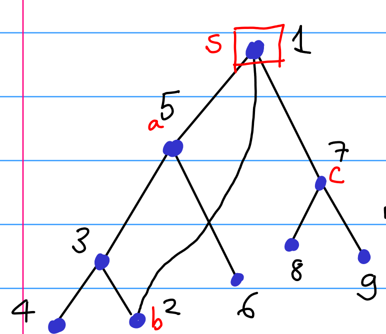
¿En qué orden recorre DFS los vértices del grafo iniciando en s?

Sol:



Nota: Asumimos que las listas Out(v) están ordenadas de derecha a izquiera según el dibujo (es decir $Out(s) = [a, b, c]$)

Ese orden CAMBIA EL ORDEN en que visitamos los vértices en DFS aunque es el mismo grafo.



Si $Out(S) = [b, a, c]$

Sol:

DFS tiene la siguiente implementación recursiva

INPUT: grafo G, vértice $v \in V(G)$, lista ya-vistos $\subseteq V(G) \setminus \{v\}$

OUTPUT: lista result $\subseteq V(G)$.

```
result = []
def DFS (G, v, ya_vistos):
        result.append (v)
        ya_vistos.append (v)
        for b in Out(v):
                if b not in ya_vistos :
                        DFS(G, b, ya_vistos)
```

[DFS Recursivo]

Teorema: [Propiedades de DFS]

(1) Al ejecutar DFS$(G, v, \text{ya-vistos})$ la lista result contiene "todos los vértices de G alcanzables desde v en el grafo $G \setminus$ ya-vistos"

$$\Lambda_G(v, \text{ya-vistos}) = \left\{ w \in V(G) : \begin{array}{l} \text{Existe un camino de } w \\ \text{a } v \text{ con vértices en} \\ V(G) \setminus \text{ya-vistos} \end{array} \right\}$$

(2) Ejecutar DFS$(G, v, \text{ya-vistos})$ requiere

$$O\left(\sum_{w \in \Lambda_G(v, \text{ya-vistos})} n_w \right) \quad \text{donde } n_w = |Out(w)|$$

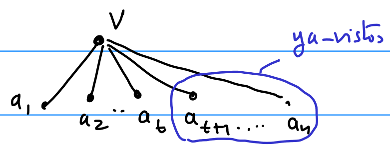
Dem: Por inducción en la longitud del árbol de recursión:

(BASE) No hay llamadas recursivas $\Leftrightarrow$ $Out(v) \subseteq$ ya-vistos

result = [v], no entramos en el if y no hay más llamadas.

El for requiere $O(n_v)$ pasos. ✓

(IND) Hay llamadas adicionales ssi $Out(v) \not\subseteq$ ya-vistos



Sucesivamente llamamos

DFS$(G, a_1, \text{ya-vistos})$

DFS$(G, a_2, \text{ya-vistos})$

⋮

DFS$(G, a_t, \text{ya-vistos})$

ya-vistos cambia en cada uno de estos.

Todo camino que sale de v pasa por algún a_i
Podemos partir los vértices de llegada en conjuntos disjuntos M_i
así:

Γ_i = Vértices de G a los que se puede llegar desde v iniciando por a_i.

$M_1 = \Gamma_1$
$M_2 = \Gamma_2 \setminus \Gamma_1$
$M_3 = \Gamma_3 \setminus (\Gamma_1 \cup \Gamma_2)$
$M_4 = \Gamma_4 \setminus (\Gamma_1 \cup \Gamma_2 \cup \Gamma_3)$
⋮

y por hipótesis de inducción concluimos que

$$DFS(G, a_1, ya\text{-}vistos) \longrightarrow M_1$$
$$DFS(G, a_2, ya\text{-}vistos) \longrightarrow M_2$$
⋮
$$DFS(G, a_t, ya\text{-}vistos) \longrightarrow M_t$$

result ← $v, M_1, M_2, \ldots, M_t$

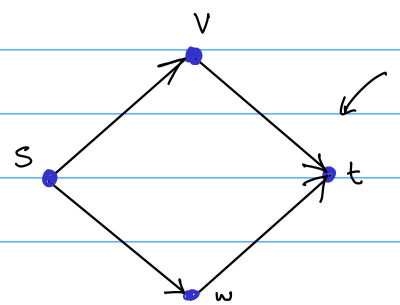
y tambien que el tiempo total es

$$O\Big(\underbrace{n_v}_{} + \sum_{w \in M_1 \sqcup \ldots \sqcup M_t} n_w\Big) = O\Big(\sum_{w \in \Lambda(v, ya\text{-}vistos)} n_w\Big)$$

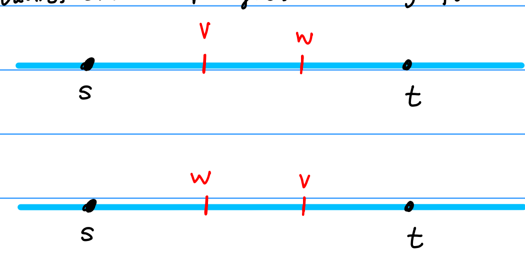
chequea si cada $b \in Out(v)$ $\in$ ya-vistos

III.2 Aplicación: Topological orderings.

Def: Sea G un grafo dirigido. Un orden topológico de G es una función $f: V(G) \longrightarrow \mathbb{R}$ tal que
(i) f asume valores distintos en $V(G)$ y
(ii) $(a,b) \in E(G) \Rightarrow f(a) < f(b)$

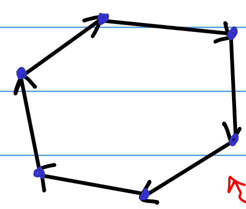


Cuántos órdenes topológicos tiene el grafo?



Si G es un grafo de precedencias (imagine los prerrequisitos de los cursos de una carrera universitaria) entonces un orden topológico es un orden en el que se pueden completar los cursos respetando los prerrequisitos.

Ejemplo: No todo grafo dirigido permite un orden topológico, por qué?



Def: Un grafo dirigido es acíclico [DAG = Directed Acyclic Graph] si no contiene ciclos orientados como

Teorema: Un grafo dirigido G admite un orden topológico ssi G es un DAG.

Dem: Si G es un DAG entonces existe un vértice fuente s al que no entra NINGUNA arista (para encontrarlo empiece de un vértice w cualquiera y camine hacia atrás por cualquier arista que entre a w, este procedimiento nunca repite vértices pues G es DAG y como el grafo es finito debe terminar en alguna fuente). Asigne a $f(s)$ el primer valor y repita el procedimiento recursivamente sobre el grafo $G \setminus \{s\}$ asignando a las fuentes sucesivas valores de f cada vez más grandes.

Recíprocamente, si f es un orden topológico de G y $v_0, v_1, v_2, \ldots v_k = v_0$ son un ciclo entonces

$f(v_0) < f(v_1) < \ldots < f(v_{k-1}) < f(v_k)$

y como $v_k = v_0$ contradicción! luego G debe ser acíclico.

El algoritmo recursivo requiere una búsqueda lineal en cada paso $O(n) + O(n-1) + \ldots + O(1) = O(n^2)$ en tiempo.

Usando DFS podemos encontrar órdenes topológicos mucho más rápidamente así:

$\boxed{\text{TOPOLOGICAL - SORT DFS:}}$

```
def  DFS-Topo (G, v)
        ya-vistos.append(v)
        for b in Out(v):
                if b not in ya-vistos:
                        DFS-Topo(G, b)
        f(s) = curLabel
        curLabel = curLabel - 1
```

Globales:
Lista ya-vistos
entero curLabel
(que va decreciendo)

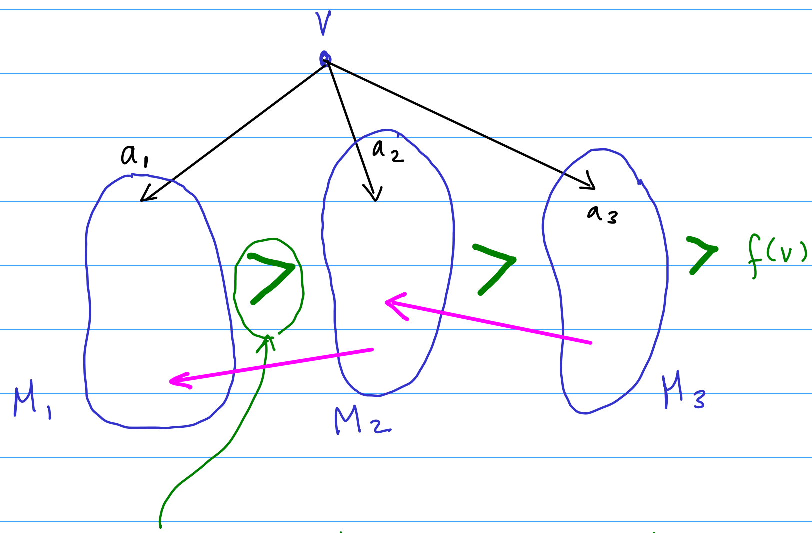
Input: Grafo G
vértice v

Postcondición: Todo vértice alcanzable desde v es marcado como visitado y se le asigna f válido.

```
curLabel = |V(G)|
ya-vistos = [ ]
f = [0, 0, ..., 0]  (del tamaño |V(G)|)
for a in G.vertices:
        if a not in ya-vistos:
                DFS-Topo(G, a)
```

Teorema: [DFS-topo] El algoritmo DFS-Topo produce un topological ordering de un DAG G en tiempo $O(m+n)$.

Dem: La esencia del algoritmo es el paso recursivo, que podemos entender con la descomposición M_i de la demostración de DFS



Al terminar la llamada recursiva los valores de f son correctos en cada M_i y satisfacen la desigualdad ($f(m_1) > f(m_2)$ $\forall\, m_1 \in M_1,\ m_2 \in M_2$)

luego las aristas moradas de arriba no pueden crear contradicción y f es un topological ordering de todo el grafo.

Son el mismo número de llamadas recursivas y tienen la misma complejidad que DFS.