

# Graphes : Parcours en profondeur (DFS)

Quentin Fortier

September 27, 2023

Pour parcourir les sommets d'un graphe :

- ➊ **Parcours en profondeur (Depth-First Search)** : on visite les sommets le plus profondément possible avant de revenir en arrière.
- ➋ **Parcours en largeur (Breadth-First Search)** : on visite les sommets par distance croissante depuis une racine.

Si le graphe est connexe, tous les sommets sont visités.

Sinon, on peut appliquer un parcours sur chacune des composantes connexes.

Pour simplifier, on va utiliser la fonction OCaml

```
List.iter : ('a -> unit) -> 'a list -> unit
```

qui applique une fonction à tous les éléments d'une liste.

# Parcours en profondeur (DFS)

Un DFS sur  $G = (V, E)$  depuis une racine  $r$  consiste, si  $r$  n'a pas déjà été visité, à le visiter puis s'appeler récursivement sur ses voisins :

# Parcours en profondeur (DFS)

Un DFS sur  $G = (V, E)$  depuis une racine  $r$  consiste, si  $r$  n'a pas déjà été visité, à le visiter puis s'appeler récursivement sur ses voisins :

---

```
let dfs g r =  
  let n = Array.length g in  
  let visited = Array.make n false in  
  let rec aux v =  
    if not visited.(v) then (  
      visited.(v) <- true;  
      List.iter aux g.(v)  
    ) in  
    aux r
```

---

$g$  est ici représenté par liste d'adjacence (de type `int list array`).

## Exercice

Adapter `dfs` si  $g$  est représenté par matrice d'adjacence.

# Parcours en profondeur (DFS)

---

```
let dfs g r =  
  let n = Array.length g in  
  let visited = Array.make n false in  
  let rec aux v =  
    if not visited.(v) then (  
      visited.(v) <- true;  
      List.iter aux g.(v)  
    ) in  
  aux r
```

---

Complexité :

# Parcours en profondeur (DFS)

---

```
let dfs g r =  
  let n = Array.length g in  
  let visited = Array.make n false in  
  let rec aux v =  
    if not visited.(v) then (  
      visited.(v) <- true;  
      List.iter aux g.(v)  
    ) in  
  aux r
```

---

Complexité :  $O(|V| + |E|)$  si représenté par liste d'adjacence car

- ① `Array.make` est en  $O(|V|)$
- ② chaque arête donne lieu à au plus 2 appels récurifs de `aux` (1 si orienté), d'où  $O(|E|)$  appels récurifs
- ③ chaque appel récurif est en  $O(1)$  (`g.(v)` est en  $O(1)$ )

# Parcours en profondeur (DFS)

Le parcours en profondeur sur  $G = (V, E)$  depuis une racine  $r$  consiste, si  $r$  n'a pas déjà été visité, à le traiter puis s'appeler récursivement sur ses voisins :

---

```
let dfs g r =  
  let n = Array.length g in  
  let visited = Array.make n false in  
  let rec aux v =  
    if not visited.(v) then (  
      visited.(v) <- true;  
      List.iter aux g.(v)  
    ) in  
  aux r
```

---

Complexité :  $O(|V|^2)$  si représenté par matrice d'adjacence car

- 1 `Array.make` est en  $O(|V|)$
- 2 on fait au plus  $|V|$  appels à `g.adj` en  $O(|V|)$



# Parcours en profondeur (DFS)

En Python (vu en MPSI) :

---

```
def dfs(G, s):  
    visited = [False]*len(G)  
    def aux(u):  
        if not visited[u]:  
            visited[u] = True  
            for v in G[u]:  
                aux(v)  
    aux(s)
```

---

# Parcours en profondeur (DFS) : Arbre binaire

L'ordre de visite des voisins est quelconque, *a priori*.

Dans le cas particulier d'un arbre binaire, on distingue plusieurs parcours en profondeur (depuis la racine), suivant l'ordre de parcours de  $N(r, g, d)$  :

- ➊ **Parcours préfixe** :  $r$ , puis  $g$ , puis  $d$
- ➋ **Parcours infixé** :  $g$ , puis  $r$ , puis  $d$
- ➌ **Parcours suffixe** :  $g$ , puis  $d$ , puis  $r$

# Parcours en profondeur (DFS) : Application à la connexité

## Question

Comment déterminer si un graphe **non orienté** est connexe?

# Parcours en profondeur (DFS) : Application à la connexité

## Question

Comment déterminer si un graphe **non orienté** est connexe?

Il suffit de vérifier que le tableau `visited` ne contient que des `true`.

# Parcours en profondeur (DFS) : Application à la connexité

Si le graphe n'est pas connexe, on peut effectuer un parcours sur chacune des composantes connexes :

---

```
let dfs g r =  
  let n = Array.length g in  
  let visited = Array.make n false in  
  let rec aux v =  
    if not visited.(v) then (  
      visited.(v) <- true;  
      List.iter aux g.(v)  
    ) in  
  for r = 0 to g.n - 1 do  
    aux r  
  done
```

---

# Parcours en profondeur (DFS) : Application à la connexité

## Exercice

Écrire une fonction

`chemin : int int array -> int -> int -> bool` telle que  
`chemin g u v` détermine s'il existe un chemin de `u` vers `v` dans le  
graphe `g`.

## Parcours en profondeur (DFS) : Détection de cycle

### Question

Comment déterminer si un graphe **non orienté** contient un cycle?

# Parcours en profondeur (DFS) : Détection de cycle

## Question

Comment déterminer si un graphe **non orienté** contient un cycle?

On regarde si on revient sur un sommet déjà visité...

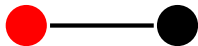


# Parcours en profondeur (DFS) : Détection de cycle

## Question

Comment déterminer si un graphe **non orienté** contient un cycle?

On regarde si on revient sur un sommet déjà visité...et que ce n'est pas un fils qui revient sur son père!

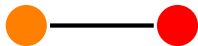


# Parcours en profondeur (DFS) : Détection de cycle

## Question

Comment déterminer si un graphe **non orienté** contient un cycle?

On regarde si on revient sur un sommet déjà visité...et que ce n'est pas un fils qui revient sur son père!



# Parcours en profondeur (DFS) : Détection de cycle

## Question

Comment déterminer si un graphe **non orienté** contient un cycle?

On regarde si on revient sur un sommet déjà visité...et que ce n'est pas un fils qui revient sur son père!



# Parcours en profondeur (DFS) : Détection de cycle

## Question

Comment déterminer si un graphe **non orienté** contient un cycle?

1ère solution : ne pas s'appeler récursivement sur son père.

```
let has_cycle g = (* g : graphe non orienté représenté par liste d'adjacence *)
  let n = Array.length g in
  let pred = Array.make n (-1) in
  let ans = ref false in
  let rec aux p u = (* p a permis de découvrir u *)
    if pred.(u) = -1 then (
      pred.(u) <- p;
      List.iter (aux p) g.(u)
    )
    else if pred.(p) <> u then ans := true in
  for i = 0 to n - 1 do
    aux i i (* cherche un cycle depuis le sommet i *)
  done;
```

# Parcours en profondeur (DFS) : Détection de cycle

## Question

Comment déterminer si un graphe **non orienté** contient un cycle?

2ème solution : stocker le prédécesseur de chaque sommet dans un tableau.

---

```
let has_cycle g =
  let n = Array.length g in
  let pred = Array.make n (-1) in
  let ans = ref false in
  let rec aux p u = (* p a permis de découvrir u *)
    if pred.(u) = -1 then (
      pred.(u) <- p;
      List.iter (aux p) g.(u)
    )
    else if pred.(p) <> u then ans := true in
  for i = 0 to n - 1 do
    aux i i (* cherche un cycle depuis le sommet i *)
  done;
  !ans
```

## Parcours en profondeur (DFS) : Détection de cycle

### Question

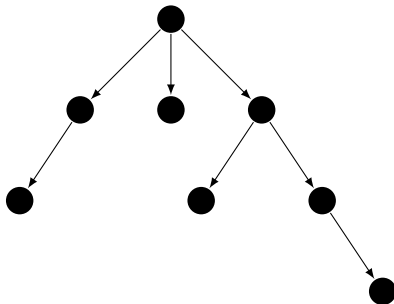
Comment déterminer si un graphe **orienté**  $\vec{G} = (V, \vec{E})$  contient un cycle?

# Parcours en profondeur (DFS) : Détection de cycle

## Question

Comment déterminer si un graphe **orienté**  $\vec{G} = (V, \vec{E})$  contient un cycle?

Soit  $A$  un arbre de parcours en profondeur de  $\vec{G}$ .

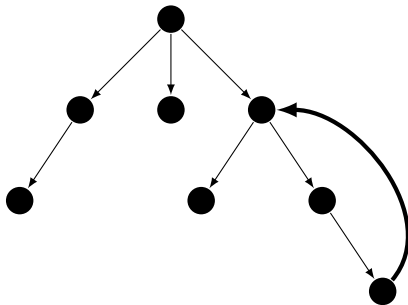


# Parcours en profondeur (DFS) : Détection de cycle

## Question

Comment déterminer si un graphe **orienté**  $\vec{G} = (V, \vec{E})$  contient un cycle?

Soit  $A$  un arbre de parcours en profondeur de  $\vec{G}$ .



Un **arc arrière** de  $A$  est un arc  $\vec{e} \in \vec{E}$  d'un sommet de  $A$  vers un de ses ancêtres.



## Parcours en profondeur (DFS) : Détection de cycle

Soit  $A$  un arbre de parcours en profondeur de  $\vec{G}$  depuis  $r$  :

### Théorème

$\vec{G}$  a un cycle  $\vec{C}$  atteignable depuis  $r$



$A$  possède un arc arrière

# Parcours en profondeur (DFS) : Détection de cycle

Soit  $A$  un arbre de parcours en profondeur de  $\vec{G}$  depuis  $r$  :

## Théorème

$$\begin{array}{c} \vec{G} \text{ a un cycle } \vec{C} \text{ atteignable depuis } r \\ \iff \\ A \text{ possède un arc arrière} \end{array}$$

Preuve :

$\Leftarrow$  : évident.

$\Rightarrow$  : Soit  $v_0$  le **premier** sommet de  $\vec{C}$  atteint par  $A$ . Notons  $\vec{C} = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k \rightarrow v_0$ .

# Parcours en profondeur (DFS) : Détection de cycle

Soit  $A$  un arbre de parcours en profondeur de  $\vec{G}$  depuis  $r$  :

## Théorème

$$\begin{array}{c} \vec{G} \text{ a un cycle } \vec{C} \text{ atteignable depuis } r \\ \iff \\ A \text{ possède un arc arrière} \end{array}$$

Preuve :

$\Leftarrow$  : évident.

$\Rightarrow$  : Soit  $v_0$  le **premier** sommet de  $\vec{C}$  atteint par  $A$ . Notons  $\vec{C} = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k \rightarrow v_0$ .

Alors l'appel de `dfs` sur  $v_0$  va visiter  $v_k$  :

# Parcours en profondeur (DFS) : Détection de cycle

Soit  $A$  un arbre de parcours en profondeur de  $\vec{G}$  depuis  $r$  :

## Théorème

$\vec{G}$  a un cycle  $\vec{C}$  atteignable depuis  $r$

$\iff$

$A$  possède un arc arrière

Preuve :

$\Leftarrow$  : évident.

$\Rightarrow$  : Soit  $v_0$  le **premier** sommet de  $\vec{C}$  atteint par  $A$ . Notons  $\vec{C} = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k \rightarrow v_0$ .

Alors l'appel de `dfs` sur  $v_0$  va visiter  $v_k$  :  $(v_k, v_0)$  est un **arc arrière**.

# Parcours en profondeur (DFS) : Détection de cycle

On teste l'existence d'un arc arrière (qui revient sur un sommet en cours d'appel récursif) :

---

```
let has_cycle g =  
  (* g : graphe orienté représenté par liste d'adjacence *)  
  let n = Array.length g in  
  let visited = Array.make n 0 in  
  let ans = ref false in  
  let rec aux v = match visited.(v) with  
    | 0 -> visited.(v) <- 1;  
              List.iter aux g.(v);  
              visited.(v) <- 2  
    | 1 -> ans := true  
    | _ -> () in  
  for i = 0 to n - 1 do  
    aux i (* cherche un cycle depuis le sommet i *)  
  done;  
  !ans
```

---

# Parcours en profondeur (DFS) : Avec pile

Les appels récurifs d'un DFS peuvent être simulés avec une pile p :

---

```
let rec dfs g r =  
  let n = Array.length g in  
  let visited = Array.make n false in  
  let p = Stack.create () in  
  Stack.push r p;  
  while not Stack.is_empty p do  
    let u = Stack.pop p in  
    if not visited.(u) then (  
      visited.(u) <- true;  
      List.iter (fun v -> Stack.push v p) g.(u)  
    )  
  done
```

---

Un sommet est marqué comme vu quand il est traité, pas au moment de l'ajouter dans la pile.

⇒ Le même sommet peut apparaître plusieurs fois dans la pile.