# Graphes : Parcours en profondeur (DFS)

Quentin Fortier

September 21, 2022

# Parcours de graphe

Comme pour les arbres, on a souvent besoin de parcourir les sommets/arêtes d'un graphe. Les deux principaux :

1. **Parcours en profondeur** (**D**epth-**F**irst **S**earch) : on visite les sommets le plus profondément possible avant de revenir en arrière.
2. **Parcours en largeur** (**B**readth-**F**irst **S**earch) : 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.

# Parcours de graphe

Pour simplifier la présentation, 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 :

Array.length g

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

voisins de g.

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)

Array.length g

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

O(n) O(p)

n = |V|, p = |E|

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

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

# Parcours en profondeur (DFS)

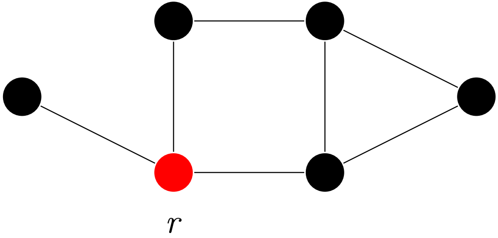
Le parcours en profondeur sur $G = (V, E)$ depuis une racine $r$ consiste, si $r$ n'a pas déjà été visité, à le traiter puis s'appeler récursivement sur ses voisins :

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

Complexité : $\boxed{O(|V|^2)}$ 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) : Exemple



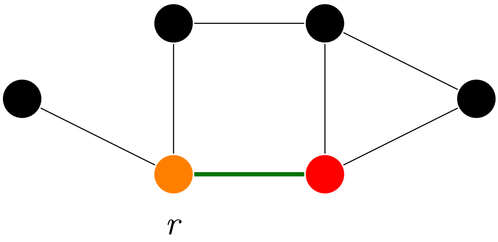

sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

# Parcours en profondeur (DFS) : Exemple



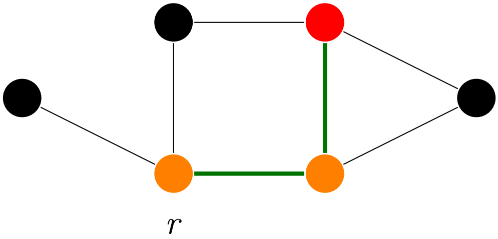

sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

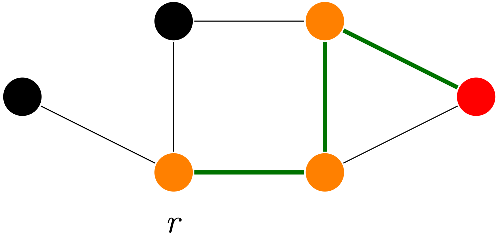
visite du sommet terminé

# Parcours en profondeur (DFS) : Exemple




- sommet pas encore visité
- sommet en cours de traitement
- appel récursif en pause
- visite du sommet terminé

# Parcours en profondeur (DFS) : Exemple



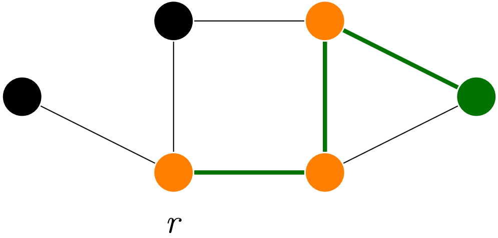

sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

# Parcours en profondeur (DFS) : Exemple



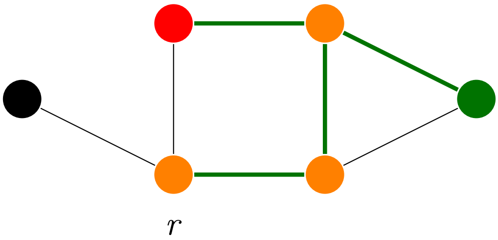

sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

# Parcours en profondeur (DFS) : Exemple



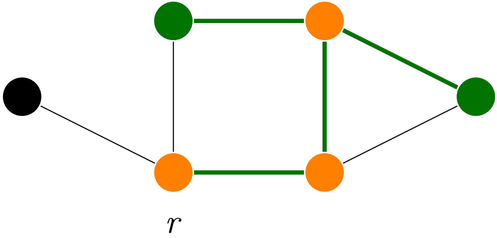

sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

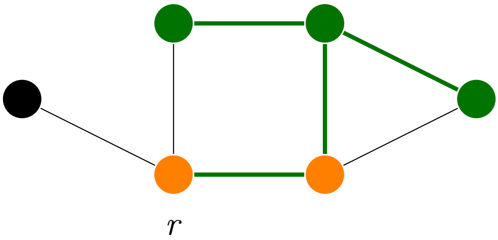
visite du sommet terminé

# Parcours en profondeur (DFS) : Exemple




- sommet pas encore visité
- sommet en cours de traitement
- appel récursif en pause
- visite du sommet terminé

# Parcours en profondeur (DFS) : Exemple



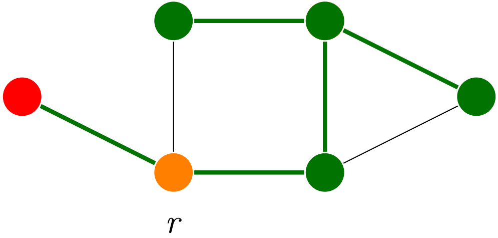

sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

# Parcours en profondeur (DFS) : Exemple



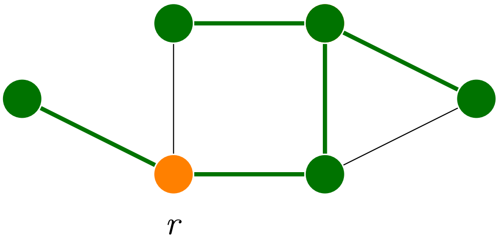

sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

# Parcours en profondeur (DFS) : Exemple



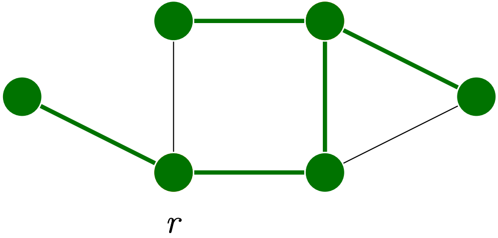

sommet pas encore visité

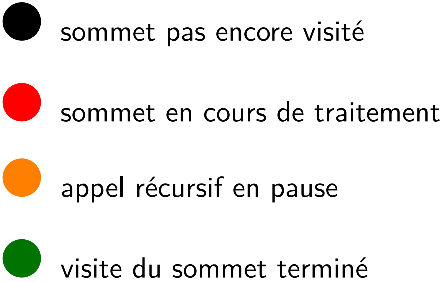
sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

# Parcours en profondeur (DFS) : Exemple



# Parcours en profondeur (DFS) : Exemple

En Python (vu en MPSI) :

```python
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)` :

1. **Parcours préfixe** : `r`, puis `g`, puis `d`
2. **Parcours infixe** : `g`, puis `r`, puis `d`
3. **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?

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 visited = Array.make g.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?

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?

<u>1ère solution</u> : ne pas s'appeler récursivement sur son père.

```
exception Cycle;;

let has_cycle g r =
    let visited = Array.make g.n false in
    let rec aux p u = (* p a permis de découvrir u *)
        if visited.(u) then raise Cycle;
        visited.(u) <- true;
        List.filter ((<>) p) (g.(u))
        |> List.map (aux p) in
    try (aux r r; false)
    with Cycle -> true
```

# 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.

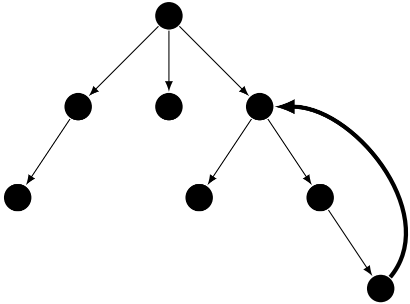
```ocaml
let has_cycle g r =
    let pred = Array.make g.n (-1) in
    let rec aux p u = (* p a permis de découvrir u *)
        if pred.(u) = -1 then (
            pred.(u) <- p;
            List.iter (aux p) (g.(u))
        )
        else if pred.(p) <> u then raise Cycle in
    try (aux r r; false)
    with Cycle -> true
```

# 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 $r$

$\iff$

$A$ possède un arc arrière

Preuve:

$\Longleftarrow$ : évident.

$\Longrightarrow$ : Soit $v_0$ le **premier** sommet de $\vec{C}$ atteint par $A$. Notons $\vec{C} = v_0 \to v_1 \to \ldots \to v_k \to v_0$.

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_dir g r =
    let visited = Array.make g.n false in
    let rec aux v = match visited.(v) with
        | 0 -> visited.(v) <- 1;
               List.iter aux g.(v);
               visited.(v) <- 2
        | 1 -> raise Cycle
        | _ -> () in
    try (aux r; false)
    with Cycle -> true
```

`visited.(v) = 0` $\iff$ v non visité
`visited.(v) = 1` $\iff$ appel récursif sur v en cours
`visited.(v) = 2` $\iff$ visite de v terminé

# Parcours en profondeur (DFS) : Avec pile

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

```
let rec dfs g r =
    let visited = Array.make g.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.
$\Longrightarrow$ Le même sommet peut apparaître plusieurs fois dans la pile.