Graphes : Parcours en profondeur (DFS)

Quentin Fortier

April 5, 2023

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 :

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 : $\boxed{O(|V| + |E|)}$ 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 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 : $\boxed{\mathrm{O}(|V|^2)}$ si représenté par matrice d'adjacence car

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

Parcours en profondeur (DFS)

Théorème

`dfs g r` visite exactement les sommets accessibles depuis `r` dans le graphe `g`.

Preuve :

- Si v est un sommet visité alors les appels récursifs à `dfs` donnent un chemin de `r` à `v`.

Parcours en profondeur (DFS)

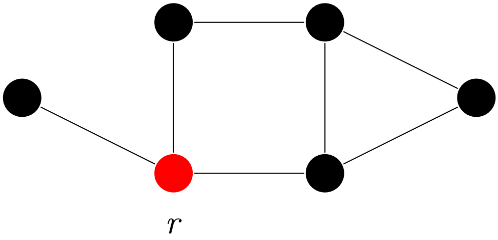
Théorème

`dfs g r` visite exactement les sommets accessibles depuis `r` dans le graphe `g`.

<u>Preuve</u> :

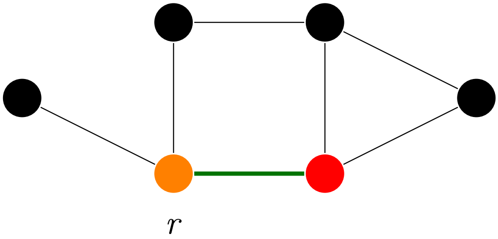
- Si v est un sommet visité alors les appels récursifs à `dfs` donnent un chemin de `r` à `v`.
- Supposons qu'il existe un chemin de r à un sommet v, noté $r = v_1 \to ... \to v_k = v$, mais que v ne soit jamais visité.
  Soit i le plus petit indice tel que v_i n'ai pas été visité.
  Alors $i > 1$ (car `r` est visité initialement) et l'appel de `dfs` sur v_{i-1} aurait dû visiter v_i : absurde.

Parcours en profondeur (DFS) : Exemple



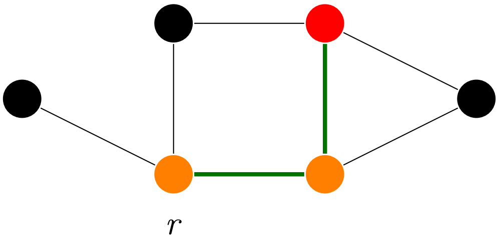
- sommet pas encore visité
- sommet en cours de traitement
- appel récursif en pause
- visite du sommet terminé

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



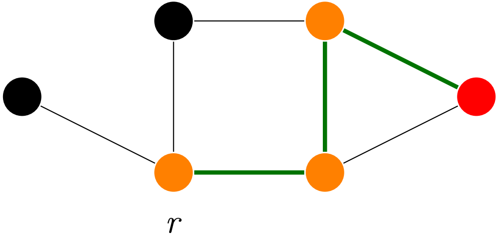
sommet pas encore visité

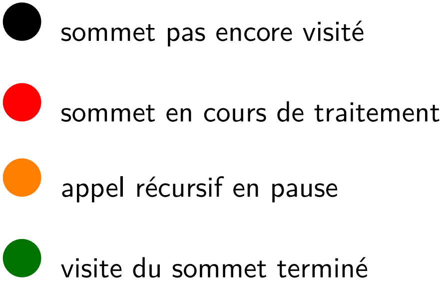
sommet en cours de traitement

appel récursif en pause

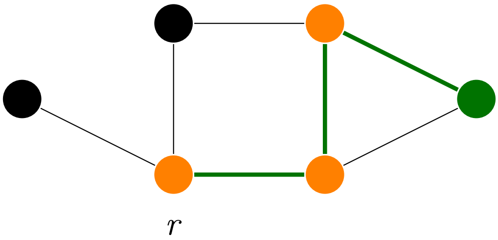
visite du sommet terminé

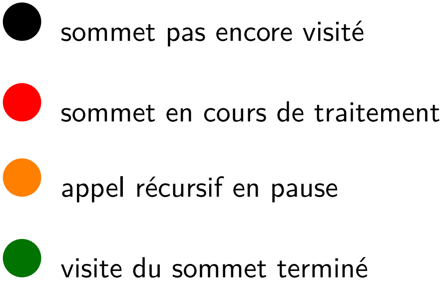
Parcours en profondeur (DFS) : Exemple



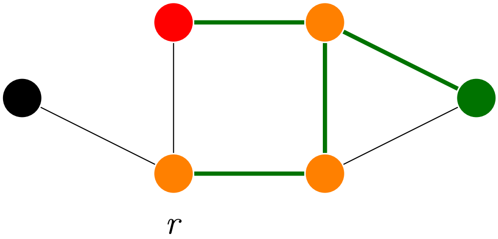


Parcours en profondeur (DFS) : Exemple





Parcours en profondeur (DFS) : Exemple



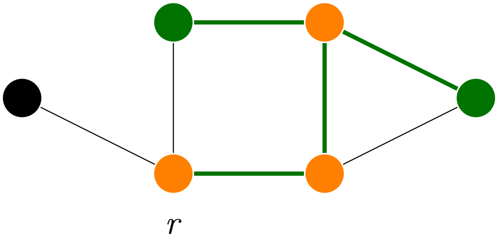
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



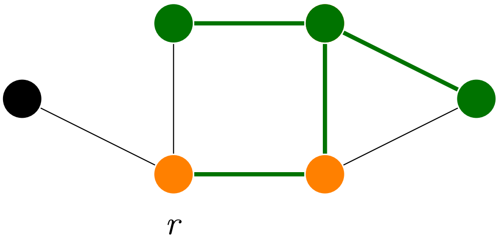

sommet pas encore visité


sommet en cours de traitement


appel récursif en pause


visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



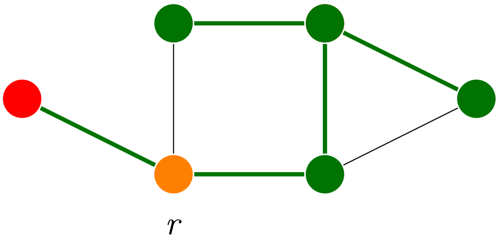
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



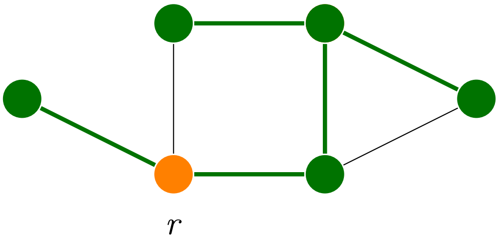
sommet pas encore visité

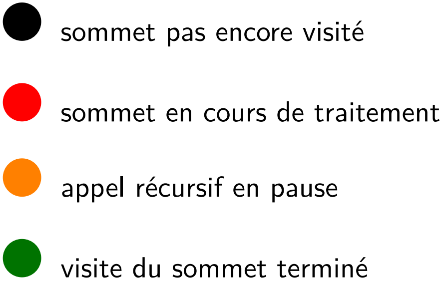
sommet en cours de traitement

appel récursif en pause

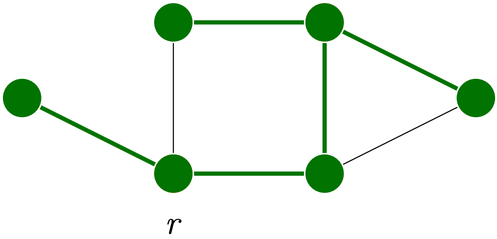
visite du sommet terminé

Parcours en profondeur (DFS) : Exemple





Parcours en profondeur (DFS) : Exemple



sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

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?

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!

déjà visité

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.

```
let has_cycle g = (* g : graphe non orienté représenté par liste d
    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

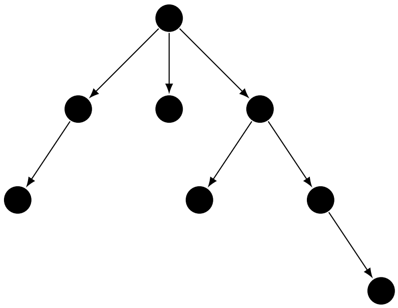
Comment déterminer si un graphe **orienté** $\vec{G} = (V, \vec{E})$ contient un cycle?

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 $\vec{G}$.

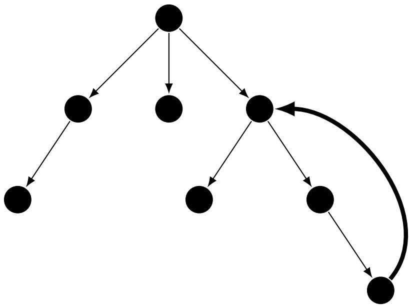


Parcours en profondeur (DFS) : Détection de cycle

Question

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

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



Un **arc arrière** de A est un arc $\overrightarrow{e} \in \overrightarrow{E}$ 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 $\overrightarrow{G}$ depuis r :

Théorème

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

$$\Longleftrightarrow$$

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

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

$$\Longleftrightarrow$$

A possède un arc arrière

Preuve :

$\Longleftarrow$: évident.

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

Parcours en profondeur (DFS) : Détection de cycle

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

Théorème

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

$$\Longleftrightarrow$$

A possède un arc arrière

<u>Preuve</u> :
$\Longleftarrow$: évident.
$\Longrightarrow$: Soit v_0 le **premier** sommet de $\overrightarrow{\mathcal{C}}$ atteint par A. Notons $\overrightarrow{\mathcal{C}} = v_0 \to v_1 \to ... \to v_k \to v_0$.
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 $\overrightarrow{G}$ depuis r :

Théorème

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

$$\iff$$

A possède un arc arrière

Preuve :

$\Longleftarrow$: évident.

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

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 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 récursifs 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.
$\Longrightarrow$ Le même sommet peut apparaître plusieurs fois dans la pile.