Graphes : Parcours

Quentin Fortier

February 4, 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 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 seen = Array.make g.n false in
    let rec aux v =
        if not seen.(v) then (
            seen.(v) <- true;
            List.iter aux (g.adj v)
        ) in
    aux r
```

Parcours en profondeur (DFS)

```
let dfs g r =
    let seen = Array.make g.n false in
    let rec aux v =
        if not seen.(v) then (
            seen.(v) <- true;
            List.iter aux (g.adj 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.adj v` est en $O(1)$)

Parcours en profondeur (DFS)

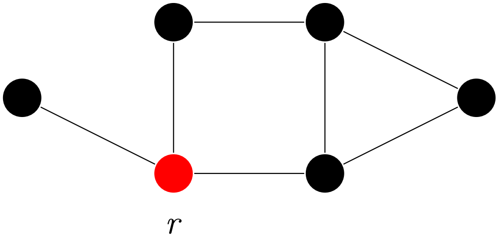
Le parcours en profondeur sur $G = (V, E)$ depuis une racine r consiste, **si r n'a pas déjà été visité**, à le traiter puis s'appeler récursivement sur ses voisins :

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

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

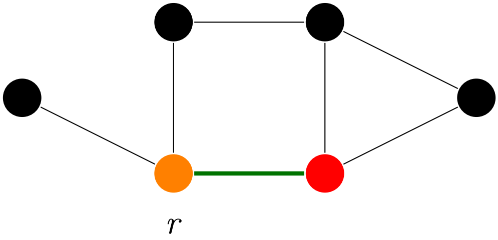
1. `Array.make` est en $O(|V|)$
2. on fait au plus $|V|$ appels à `g.adj` en $O(|V|)$

Parcours en profondeur (DFS) : Exemple



- `vu.(v) = false`
- sommet en cours de traitement
- appel récursif en pause
- visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



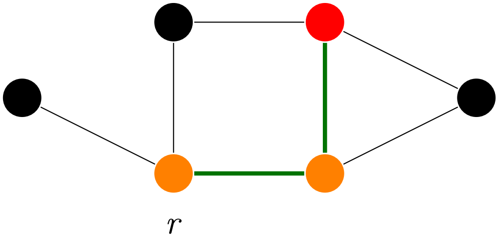
 `vu.(v) = false`

 sommet en cours de traitement

 appel récursif en pause

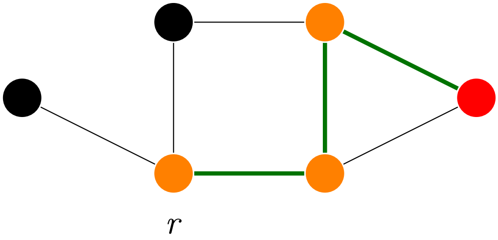
 visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



- `vu.(v) = false`
- sommet en cours de traitement
- appel récursif en pause
- visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



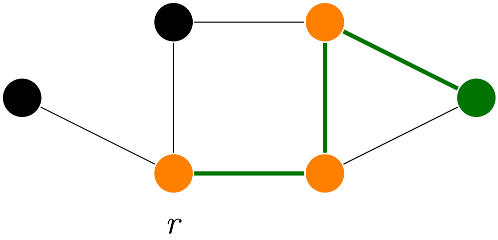
`vu.(v) = false`

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



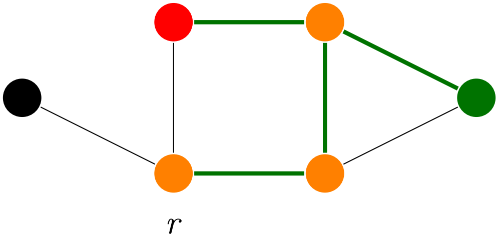
`vu.(v) = false`

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



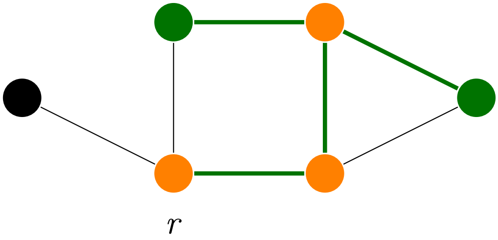
`vu.(v) = false`

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



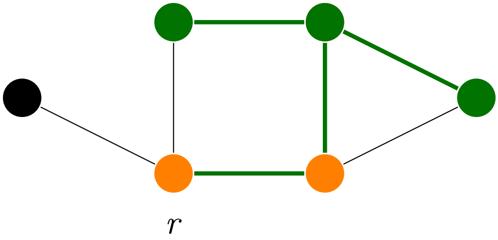
`vu.(v) = false`

sommet en cours de traitement

appel récursif en pause

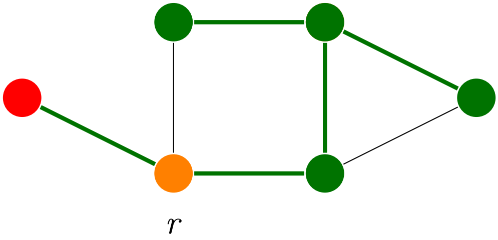
visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



- `vu.(v) = false`
- sommet en cours de traitement
- appel récursif en pause
- visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



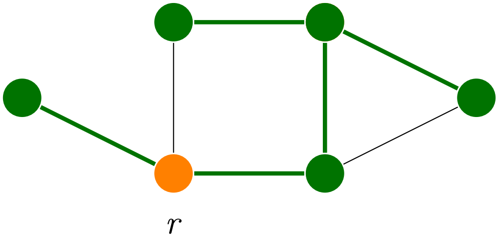
`vu.(v) = false`

sommet en cours de traitement

appel récursif en pause

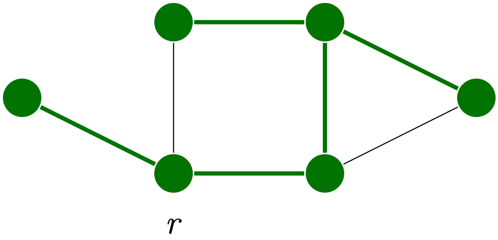
visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



- `vu.(v) = false`
- sommet en cours de traitement
- appel récursif en pause
- visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



`vu.(v) = false`

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Parcours en profondeur (DFS) : Exemple

En C :

```c
// m : matrice d'adjacence d'un graphe
// n : nombre de sommets (= taille de m)
// s : sommet de départ
// seen : seen[v] indique si v a été visité
// (initialement seen est rempli de false)
void dfs(int** m, int n, int s, bool* seen) {
    if(seen[s])
        return;
    seen[s] = true;
    // traiter s (l'afficher, par ex.)
    for(int v = 0; v < n; v++)
        if(m[s][v] == 1)
            dfs(m, n, v, seen);
}
```

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

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.

```
exception Cycle;;

let has_cycle g r =
    let seen = Array.make g.n false in
    let rec aux p u = (* p a permis de découvrir u *)
        if seen.(u) then raise Cycle;
        seen.(u) <- true;
        List.filter ((<>) p) (g.adj 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?

<u>2ème solution</u> : stocker le prédécesseur de chaque sommet dans un tableau.

```
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.adj 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

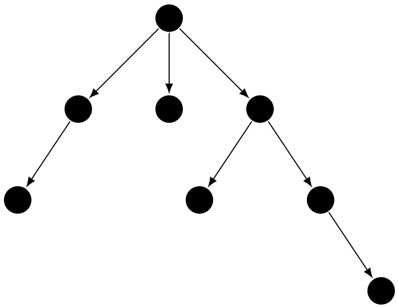
Comment déterminer si un graphe **orienté** $\vec{G} = (V, \vec{E})$ contient un cycle?

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 $\overrightarrow{G}$.

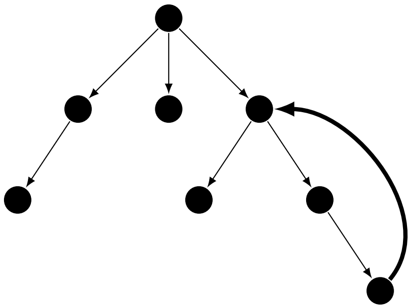


Parcours en profondeur (DFS) : Détection de cycle

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 $\overrightarrow{G}$ depuis r :

Théorème

$$\overrightarrow{G} \text{ a un cycle } \overrightarrow{C} \text{ atteignable depuis } r$$
$$\iff$$
$$A \text{ possède un arc arrière}$$

Parcours en profondeur (DFS) : Détection de cycle

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

Théorème

$$\vec{G} \text{ a un cycle } \vec{\mathcal{C}} \text{ atteignable depuis } r$$
$$\Longleftrightarrow$$
$$A \text{ possède un arc arrière}$$

<u>Preuve</u>:
$\Longleftarrow$: évident.
$\Longrightarrow$: Soit v_0 le **premier** sommet de $\vec{\mathcal{C}}$ atteint par A. Notons $\vec{\mathcal{C}} = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k \rightarrow v_0$.

Parcours en profondeur (DFS) : Détection de cycle

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

Théorème

$\vec{G}$ 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 $\vec{\mathcal{C}} = v_0 \to v_1 \to ... \to v_k \to v_0$.

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

$$\Longleftrightarrow$$

A possède un arc arrière

Preuve:

$\Longleftarrow$: évident.

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

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

`seen.(v) = 0` $\iff$ v non visité

`seen.(v) = 1` $\iff$ appel récursif sur v en cours

`seen.(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 seen = 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 seen.(u) then (
            seen.(u) <- true;
            List.iter (fun v -> Stack.push v p) (g.adj u)
        )
    done;;
```

Un sommet est marqué comme seen quand il est traité, pas au moment de l'ajouter dans la pile.
$\Longrightarrow$ Le même sommet peut apparaître plusieurs fois dans la pile.

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 seen = 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 seen.(u) then (
            seen.(u) <- true;
            List.iter (fun v -> Stack.push v p) (g.adj u)
        )
    done;;
```

Un sommet est marqué comme seen quand il est traité, pas au moment de l'ajouter dans la pile.
$\Longrightarrow$ Le même sommet peut apparaître plusieurs fois dans la pile.

Qu'obtient-on avec une file au lieu d'une pile?

Parcours en largeur (BFS) : Avec file

```
let bfs g r =
    let seen = Array.make g.n false in
    let q = Queue.create () in
    let add u =
        if not seen.(u) then (seen.(v) <- true; Queue.add v q) in
    add r;
    while not Queue.is_empty q do
        let u = Queue.pop q in
        (* traiter u *)
        List.iter add (g.adj u)
    done;;
```

Parcours en largeur (BFS) : Avec file

```
let bfs g r =
    let seen = Array.make g.n false in
    let q = Queue.create () in
    let add u =
        if not seen.(u) then (seen.(v) <- true; Queue.add v q) in
    add r;
    while not Queue.is_empty q do
        let u = Queue.pop q in
        (* traiter u *)
        List.iter add (g.adj u)
    done;;
```

La file `f` est toujours de la forme :

$\longleftarrow$ | sommets à distance d | sommets à distance $d+1$ | $\longleftarrow$

Parcours en largeur (BFS) : Avec file

```
let bfs g r =
    let seen = Array.make g.n false in
    let q = Queue.create () in
    let add u =
        if not seen.(u) then (seen.(v) <- true; Queue.add v q) in
    add r;
    while not Queue.is_empty q do
        let u = Queue.pop q in
        (* traiter u *)
        List.iter add (g.adj u)
    done;;
```

La file `f` est toujours de la forme :

sommets à distance d	sommets à distance $d+1$

Les sommets sont donc **traités par distance croissante** à `r` : d'abord `r`, puis les voisins de `r`, puis ceux à distance 2...

Parcours en largeur (BFS) : Avec file

En C, en supposant avoir un type queue avec des fonctions `queue* create(void)`, `bool is_empty(queue*)`, `void push(queue*, int)` et `int pop(queue*)` :

```c
void bfs(int** m, int n, int s) {
    bool *seen = malloc(n*sizeof(bool));
    for(int i = 0; i < n; i++)
        seen[i] = false;
    queue* q = create();
    while(!is_empty(q)) {
        int u = pop(q);
        // traiter u
        for(int v = 0; v < n; v++)
            if(m[u][v] == 1 && !seen[v])
                push(q, v);
    }
}
```

Parcours en largeur (BFS) : Avec 2 couches

sommets à distance d	sommets à distance $d+1$

On peut aussi utiliser deux listes : `cur` pour la couche courante, `next` pour la couche suivante.

```
let bfs g r =
    let seen = Array.make g.n false in
    let rec aux cur next = match cur with
        | [] -> if next <> [] then aux next []
        | u::q when seen.(u) -> aux q next
        | u::q -> seen.(u) <- true;
                  aux q (next @ (g.adj u)) in
    aux [r] []
```

Parcours en largeur (BFS) : Application au calcul de distance

Question

Comment connaître la distance d'un sommet `r` à un autre?

Parcours en largeur (BFS) : Application au calcul de distance

Question

Comment connaître la distance d'un sommet `r` à un autre?

Conserver la distance en argument puis la stocker dans un tableau `dist` (`dist.(v)` va contenir la distance de `r` à `v`) :

Parcours en largeur (BFS) : Application au calcul de distance

Question

Comment connaître la distance d'un sommet `r` à un autre?

Conserver la distance en argument puis la stocker dans un tableau `dist` (`dist.(v)` va contenir la distance de `r` à `v`) :

```
let bfs g r =
    let dist = Array.make g.n (-1) in
    let rec aux d cur next = match cur with
        | [] -> if next <> [] then aux (d + 1) next []
        | u::q when dist.(u) <> -1 -> aux d q next
        | u::q -> dist.(u) <- d;
                  aux d q ((g.adj u) @ next) in
    aux 0 [r] []; dist
```

Complexité :

Parcours en largeur (BFS) : Application au calcul de distance

Question

Comment connaître la distance d'un sommet `r` à un autre?

Conserver la distance en argument puis la stocker dans un tableau `dist` (`dist.(v)` va contenir la distance de `r` à `v`) :

```ocaml
let bfs g r =
    let dist = Array.make g.n (-1) in
    let rec aux d cur next = match cur with
        | [] -> if next <> [] then aux (d + 1) next []
        | u::q when dist.(u) <> -1 -> aux d q next
        | u::q -> dist.(u) <- d;
                  aux d q ((g.adj u) @ next) in
    aux 0 [r] []; dist
```

Complexité : $O(|V| + |E|)$ avec liste d'adjacence.

Parcours en largeur (BFS) : Plus courts chemins

Question

Comment connaître un plus court chemin d'un sommet `r` à un autre?

Parcours en largeur (BFS) : Plus courts chemins

Question

Comment connaître un plus court chemin d'un sommet `r` à un autre?

On stocke dans `pred.(v)` le sommet qui a permis de découvrir `v` :

Parcours en largeur (BFS) : Plus courts chemins

Question

Comment connaître un plus court chemin d'un sommet `r` à un autre?

On stocke dans `pred.(v)` le sommet qui a permis de découvrir `v` :

```
let bfs g r =
    let pred = Array.make g.n (-1) in
    let q = Queue.create () in
    let add p u = (* p est le père de u *)
        if pred.(u) = -1 then (pred.(v) <- p; Queue.add v q) in
    add r r;
    while not Queue.is_empty q do
        let u = Queue.pop q in
        (* traiter u *)
        List.iter (add u) (g.adj u)
    done; pred
```

Complexité :

Parcours en largeur (BFS) : Plus courts chemins

Question

Comment connaître un plus court chemin d'un sommet `r` à un autre?

On stocke dans `pred.(v)` le sommet qui a permis de découvrir `v` :

```
let bfs g r =
    let pred = Array.make g.n (-1) in
    let q = Queue.create () in
    let add p u = (* p est le père de u *)
        if pred.(u) = -1 then (pred.(v) <- p; Queue.add v q) in
    add r r;
    while not Queue.is_empty q do
        let u = Queue.pop q in
        (* traiter u *)
        List.iter (add u) (g.adj u)
    done; pred
```

Complexité : $O(|V| + |E|)$ avec liste d'adjacence.

Parcours en largeur (BFS) : Plus courts chemins

```
let bfs g r =
    let pred = Array.make g.n (-1) in
    let q = Queue.create () in
    let add p u = (* p est le père de u *)
        if pred.(u) = -1 then (pred.(v) <- p; Queue.add v q) in
    add r r;
    while not Queue.is_empty q do
        let u = Queue.pop q in
        (* traiter u *)
        List.iter (add u) (g.adj u)
    done; pred
```

Question

Comment en déduire un plus court chemin de r à v?

Parcours en largeur (BFS) : Plus courts chemins

```
let bfs g r =
    let pred = Array.make g.n (-1) in
    let q = Queue.create () in
    let add p u = (* p est le père de u *)
        if pred.(u) = -1 then (pred.(v) <- p; Queue.add v q) in
    add r r;
    while not Queue.is_empty q do
        let u = Queue.pop q in
        (* traiter u *)
        List.iter (add u) (g.adj u)
    done; pred
```

Question

Comment en déduire un plus court chemin de r à v?

```
let rec path pred v =
    if pred.(v) = v then [v]
    else v::path pred pred.(v)
```

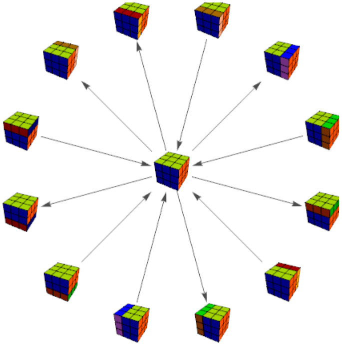
Parcours en largeur (BFS) : Plus courts chemins

Application : résoudre un Rubik's Cube avec le nombre minimum de coups.

Parcours en largeur (BFS) : Plus courts chemins

Application : résoudre un Rubik's Cube avec le nombre minimum de coups.

1. Sommets = configurations possibles du Rubik's Cube.
2. Arêtes = mouvements élémentaires.



Parcours en largeur (BFS) : Plus courts chemins

Application : résoudre un Rubik's Cube avec le nombre minimum de coups.

1. Sommets = configurations possibles du Rubik's Cube.
2. Arêtes = mouvements élémentaires.

Théorème (2010)

Le **diamètre** (distance max entre deux sommets) du graphe des configurations du Rubik's Cube est 20.

$\Longrightarrow$ on peut résoudre n'importe quel Rubik's Cube en au plus 20 mouvements.