Représentation et parcours de graphes

MP/MP* Option info

Structure abstraite de graphe

On souhaite implémenter une structure de graphe possédant les opérations:

1. ajouter / supprimer une arête
2. (ajouter / supprimer un sommet)
3. savoir s'il existe une arête entre 2 sommets
4. connaître la liste des voisins d'un sommet
5. ...

Avec, si possible, une faible complexité en temps et espace.

Structure abstraite de graphe

On souhaite implémenter une structure de graphe possédant les opérations:

1. ajouter / supprimer une arête
2. (ajouter / supprimer un sommet)
3. savoir s'il existe une arête entre 2 sommets
4. connaître la liste des voisins d'un sommet
5. ...

Avec, si possible, une faible complexité en temps et espace.

Les sommets seront souvent des entiers consécutifs à partir de 0.

Matrice d'adjacence

On peut représenter un graphe non orienté $(V = \{0, ..., n-1\}, E)$ par une **matrice d'adjacence** `m : int array array` de taille $n \times n$:

- `m.(u).(v)` $= 1 \iff \{u, v\} \in E$
- `m.(u).(v)` $= 0 \iff \{u, v\} \notin E$

`m` est **symétrique**.

Matrice d'adjacence

On peut représenter un graphe non orienté $(V = \{0, ..., n-1\}, E)$ par une **matrice d'adjacence** `m : int array array` de taille $n \times n$:

- `m.(u).(v)` $= 1 \iff \{u, v\} \in E$
- `m.(u).(v)` $= 0 \iff \{u, v\} \notin E$

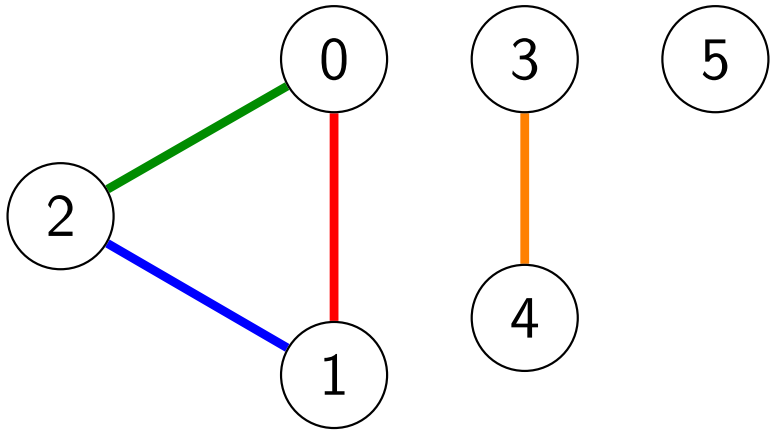
`m` est **symétrique**.

Pour un graphe orienté $(V, \vec{E})$:

- `m.(u).(v)` $= 1 \iff (u, v) \in \vec{E}$
- `m.(u).(v)` $= 0 \iff (u, v) \notin \vec{E}$

`m` **n'est pas symétrique** (a priori).

Exemple de matrice d'adjacence (non orienté)



$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Quitte à permuter lignes et colonnes (i.e renuméroter les sommets), les composantes connexes apparaissent par bloc.

Matrice d'adjacence

Si on représente un graphe orienté à n sommets et m arêtes par matrice d'adjacence `m`:

- complexité en espace:

Matrice d'adjacence

Si on représente un graphe orienté à n sommets et m arêtes par matrice d'adjacence `m`:

- complexité en espace: $\Theta(n^2)$
- ajouter arête (u, v): `m.(u).(v) <- 1`, $O(1)$
- supprimer arête (u, v): `m.(u).(v) <- 0`, $O(1)$
- ajouter / supprimer sommet:

Matrice d'adjacence

Si on représente un graphe orienté à n sommets et m arêtes par matrice d'adjacence `m`:

- complexité en espace: $\Theta(n^2)$
- ajouter arête (u, v): `m.(u).(v) <- 1`, $O(1)$
- supprimer arête (u, v): `m.(u).(v) <- 0`, $O(1)$
- ajouter / supprimer sommet: impossible avec `array`!
- test d'existence d'arête:

Matrice d'adjacence

Si on représente un graphe orienté à n sommets et m arêtes par matrice d'adjacence `m`:

- complexité en espace: $\Theta(n^2)$
- ajouter arête (u, v): `m.(u).(v) <- 1`, $O(1)$
- supprimer arête (u, v): `m.(u).(v) <- 0`, $O(1)$
- ajouter / supprimer sommet: impossible avec `array`!
- test d'existence d'arête: `m.(i).(j) = 1`, $O(1)$
- parcourir les voisins d'un sommet:

Matrice d'adjacence

Si on représente un graphe orienté à n sommets et m arêtes par matrice d'adjacence `m`:

- complexité en espace: $\Theta(n^2)$
- ajouter arête (u, v): `m.(u).(v) <- 1`, $O(1)$
- supprimer arête (u, v): `m.(u).(v) <- 0`, $O(1)$
- ajouter / supprimer sommet: impossible avec `array`!
- test d'existence d'arête: `m.(i).(j) = 1`, $O(1)$
- parcourir les voisins d'un sommet: parcourir `m.(i)`, $\Theta(n)$

⚠ modifier `m.(u).(v)` **et** `m.(v).(u)` pour un graphe non orienté.

Matrice d'adjacence

Exercices

1. Écrire une fonction `deg : int array array -> int -> int` telle que `deg m v` renvoie le degré du sommet `v` dans le graphe représenté par la matrice d'adjacence `m`.
2. Écrire une fonction `naretes : int array array -> int` renvoyant le nombre d'arêtes d'un graphe représenté par une matrice d'adjacence.
3. Écrire une fonction `voisins : int array array -> int -> int list` telle que `voisins m v` renvoie la liste des voisins de `v` dans le graphe représenté par `m`.

Puissance de la matrice d'adjacence

Question

Si $A = (a_{u,v})$ est une matrice d'adjacence d'un graphe à n sommets, que représente les coefficients de $A^k = (a_{u,v}^{(k)})$?

Puissance de la matrice d'adjacence

Question

Si $A = (a_{u,v})$ est une matrice d'adjacence d'un graphe à n sommets, que représente les coefficients de $A^k = (a_{u,v}^{(k)})$?

Pour $k = 2$:

$$a_{u,v}^{(2)} = \sum_{w=0}^{n-1} a_{u,w} a_{w,v}$$

Puissance de la matrice d'adjacence

Question

Si $A = (a_{u,v})$ est une matrice d'adjacence d'un graphe à n sommets, que représente les coefficients de $A^k = (a^{(k)}_{u,v})$?

Pour $k = 2$:

$$a^{(2)}_{u,v} = \sum_{w=0}^{n-1} a_{u,w} a_{w,v}$$

$$a_{u,w} a_{w,v} = 1 \iff u \to w \to v \text{ est un chemin}$$

Puissance de la matrice d'adjacence

Question

Si $A = (a_{u,v})$ est une matrice d'adjacence d'un graphe à n sommets, que représente les coefficients de $A^k = (a^{(k)}_{u,v})$?

Pour $k = 2$:

$$a^{(2)}_{u,v} = \sum_{w=0}^{n-1} a_{u,w} a_{w,v}$$

$$a_{u,w} a_{w,v} = 1 \iff u \to w \to v \text{ est un chemin}$$

$a^{(2)}_{u,v}$ est le nombre de chemins de longueur 2 de u à v!

Puissance de la matrice d'adjacence

Question

Si $A = (a_{u,v})$ est une matrice d'adjacence d'un graphe à n sommets, que représente les coefficients de $A^k = (a_{u,v}^{(k)})$?

$$a_{u,v}^{(k)} = \sum_{w=0}^{n-1} a_{u,w}^{(k-1)} a_{w,v}$$

Puissance de la matrice d'adjacence

Question

Si $A = (a_{u,v})$ est une matrice d'adjacence d'un graphe à n sommets, que représente les coefficients de $A^k = (a_{u,v}^{(k)})$?

$$a_{u,v}^{(k)} = \sum_{w=0}^{n-1} a_{u,w}^{(k-1)} a_{w,v}$$

Par récurrence sur k:

$$\boxed{a_{u,v}^{(k)} = \text{nombre de chemins de longueur } k \text{ de } u \text{ à } v}$$

Remarque: c'est vrai aussi bien pour les graphes orientés que non-orientés.

Puissance de la matrice d'adjacence

Soit $M(n)$ la complexité pour multiplier 2 matrices $n \times n$

Puissance de la matrice d'adjacence

Soit $M(n)$ la complexité pour multiplier 2 matrices $n \times n$ ($M(n) = \Theta(n^3)$ en naïf, $O(n^{2,8})$ avec la méthode de Strassen).

On peut calculer $A^k = A \times ... \times A$ en complexité $(k-1)M(n)$.

Puissance de la matrice d'adjacence

Soit $M(n)$ la complexité pour multiplier 2 matrices $n \times n$ ($M(n) = \Theta(n^3)$ en naïf, $O(n^{2,8})$ avec la méthode de Strassen).

On peut calculer $A^k = A \times ... \times A$ en complexité $(k-1)M(n)$.

Ou, mieux, par exponentiation rapide en utilisant:

$$\begin{cases} A^k = (A^{\frac{k}{2}})^2 & \text{si } k \text{ est pair} \\ A^k = A(A^{\frac{k-1}{2}})^2 & \text{sinon} \end{cases}$$

Complexité:

Puissance de la matrice d'adjacence

Soit $M(n)$ la complexité pour multiplier 2 matrices $n \times n$ ($M(n) = \Theta(n^3)$ en naïf, $O(n^{2,8})$ avec la méthode de Strassen).

On peut calculer $A^k = A \times ... \times A$ en complexité $(k-1)M(n)$.

Ou, mieux, par exponentiation rapide en utilisant:

$$\begin{cases} A^k = (A^{\frac{k}{2}})^2 & \text{si } k \text{ est pair} \\ A^k = A(A^{\frac{k-1}{2}})^2 & \text{sinon} \end{cases}$$

Complexité: $O(\log_2(k)M(n))$.

Liste d'adjacence

La représentation par **liste d'adjacence** consiste à stocker, pour chaque sommet, la liste de ses voisins.

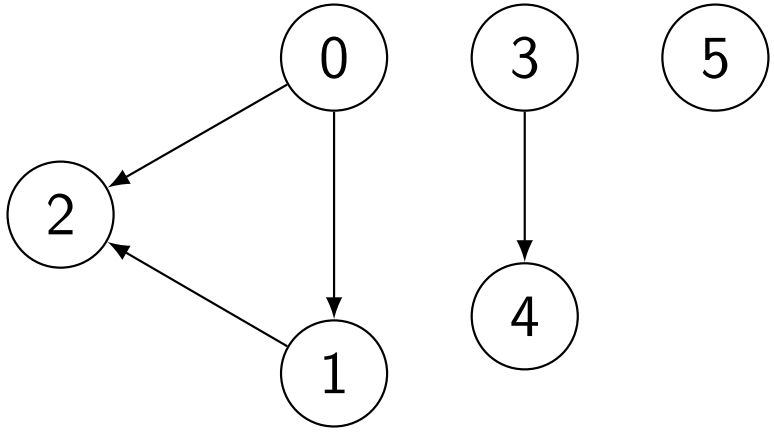
Liste d'adjacence

La représentation par **liste d'adjacence** consiste à stocker, pour chaque sommet, la liste de ses voisins.

Deux possibilités:

1. une liste de listes (`int list list`) `l0::l1::l2::...` où `l0` est la liste des voisins du sommet 0, `l1` est la liste des voisins du sommet 1...
2. un tableau de listes (`int list array`) `t` où `t.(i)` est la liste des voisins du sommet `i`.

Exemple de liste d'adjacence int list array (orienté)




```
# let g = [| [1; 2]; [2]; []; [4]; []; [] |];;
val g : int list array = [|[1; 2]; [2]; []; [4]; []; []|]
```

Comparaison

Pour un graphe orienté à n sommets et m arêtes:

	`array array`	`list array`	`list list`
espace mémoire	$\Theta(n^2)$		
ajouter (u, v)	$O(1)$		
supprimer (u, v)	$O(1)$		
existence (u, v)	$O(1)$		
voisins de u	$\Theta(n)$		
ajouter sommet	X		
supprimer sommet	X		

Comparaison

Pour un graphe orienté à n sommets et m arêtes:

	`array array`	`list array`	`list list`
espace	$\Theta(n^2)$	$\Theta(n+m)$	$\Theta(n+m)$
ajouter (u, v)	$O(1)$	$O(1)$	$O(n)$
supprimer (u, v)	$O(1)$	$O(\deg^+(u))$	$O(n)$
existence (u, v)	$O(1)$	$O(\deg^+(u))$	$O(n)$
voisins de u	$\Theta(n)$	$\Theta(\deg^+(u))$	$O(n)$
ajouter sommet	X	X	$O(n)$
supprimer sommet	X	X	$O(n)$

Comparaison

Pour un graphe orienté à n sommets et m arêtes:

	`array array`	`list array`	`list list`
espace	$\Theta(n^2)$	$\Theta(n+m)$	$\Theta(n+m)$
ajouter (u, v)	$O(1)$	$O(1)$	$O(n)$
supprimer (u, v)	$O(1)$	$O(\deg^+(u))$	$O(n)$
existence (u, v)	$O(1)$	$O(\deg^+(u))$	$O(n)$
voisins de u	$\Theta(n)$	$\Theta(\deg^+(u))$	$O(n)$
ajouter sommet	X	X	$O(n)$
supprimer sommet	X	X	$O(n)$

Si $m = \Theta(n^2)$ (graphe dense): matrice d'adjacence conseillée.
Si $m = O(n)$ (graphe creux, ex: arbre): liste d'adjacence conseillée.

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.

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.

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 faire un parcours sur chacune des composantes connexes.

do_list

Pour simplifier la présentation, on va utiliser une fonction utilitaire

```
do_list : ('a -> unit) -> 'a list -> unit
```

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

do_list

Pour simplifier la présentation, on va utiliser une fonction utilitaire

```
do_list : ('a -> unit) -> 'a list -> unit
```

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

```
let rec do_list f l = match l with
  | [] -> ()
  | e::q -> (f e; do_list f q);;
```

do_list

Pour simplifier la présentation, on va utiliser une fonction utilitaire

```
do_list : ('a -> unit) -> 'a list -> unit
```

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

```
let rec do_list f l = match l with
  | [] -> ()
  | e::q -> (f e; do_list f q);;
```

(Cette fonction existe déjà en OCaml sous le nom `List.iter`)

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:

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 vu = Array.make g.n false in
  let rec aux v =
    if not vu.(v) then (vu.(v) <- true;
                        (* traiter v *)
                        do_list aux (g.voisins v)) in
  aux r;;
```

Complexité:

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 vu = Array.make g.n false in
  let rec aux v =
    if not vu.(v) then (vu.(v) <- true;
                        (* traiter v *)
                        do_list aux (g.voisins 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 récursifs de `aux` (1 si orienté), d'où $O(|E|)$ appels récursifs
3. chaque appel récursif est en $O(1)$ (`g.voisins v` est en $O(1)$)

Parcours en profondeur (DFS)

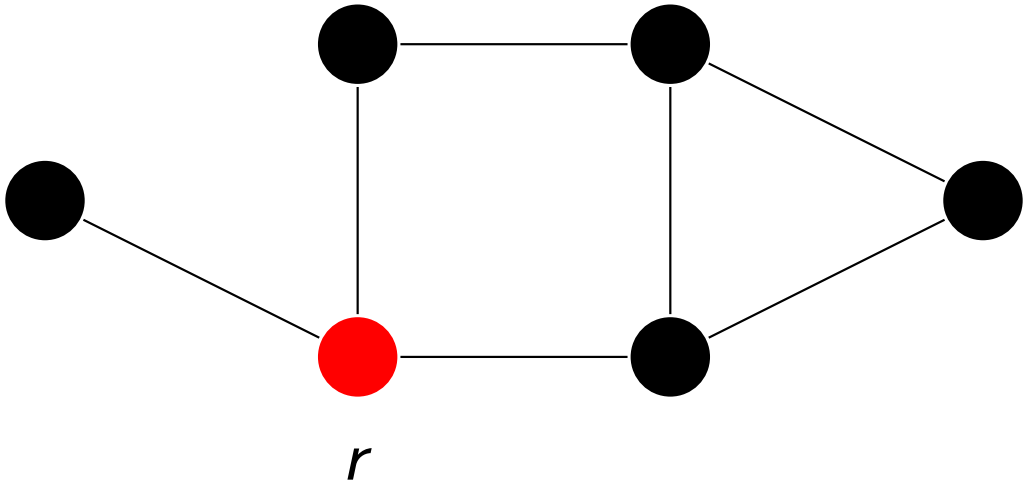
Le parcours en profondeur sur $G = (V, E)$ depuis une racine r consiste, **si r n'a pas déjà été visité**, à le traiter puis s'appeler récursivement sur ses voisins:

```
let dfs g r =
  let vu = Array.make g.n false in
  let rec aux v =
    if not vu.(v) then (vu.(v) <- true;
                        (* traiter v *)
                        do_list aux (g.voisins 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.voisins` en $O(|V|)$

Parcours en profondeur: exemple

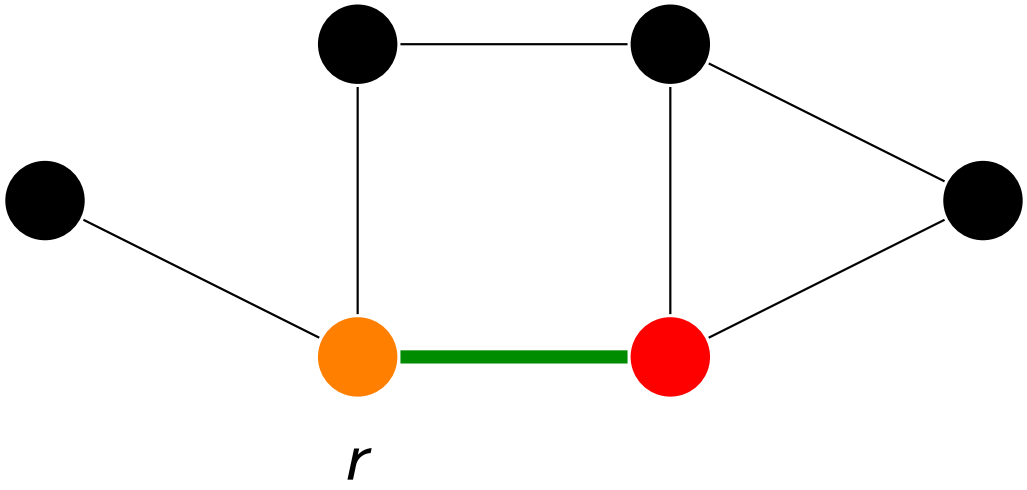


`vu.(v) = false`

`v` est en cours de traitement

appel récursif sur `v` en cours

Parcours en profondeur: exemple

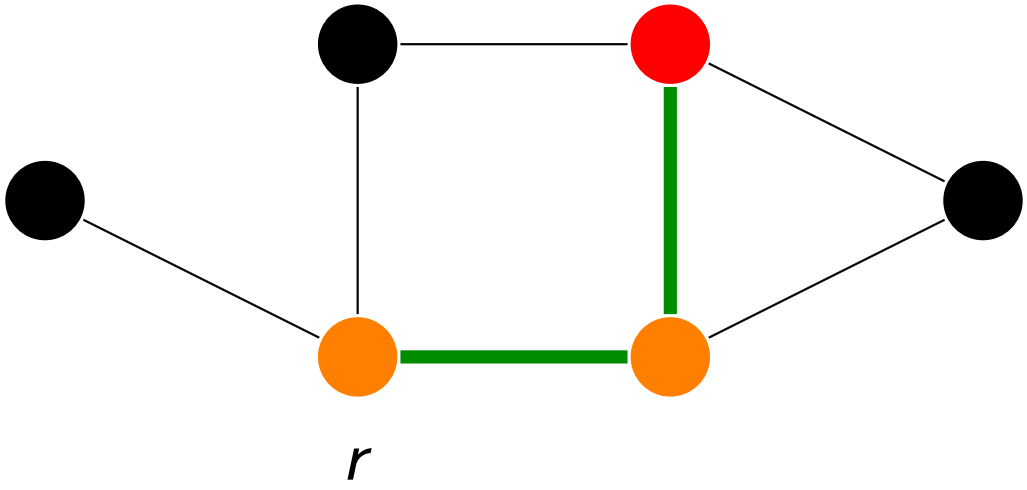


`vu.(v) = false`

`v` est en cours de traitement

appel récursif sur `v` en cours

Parcours en profondeur: exemple

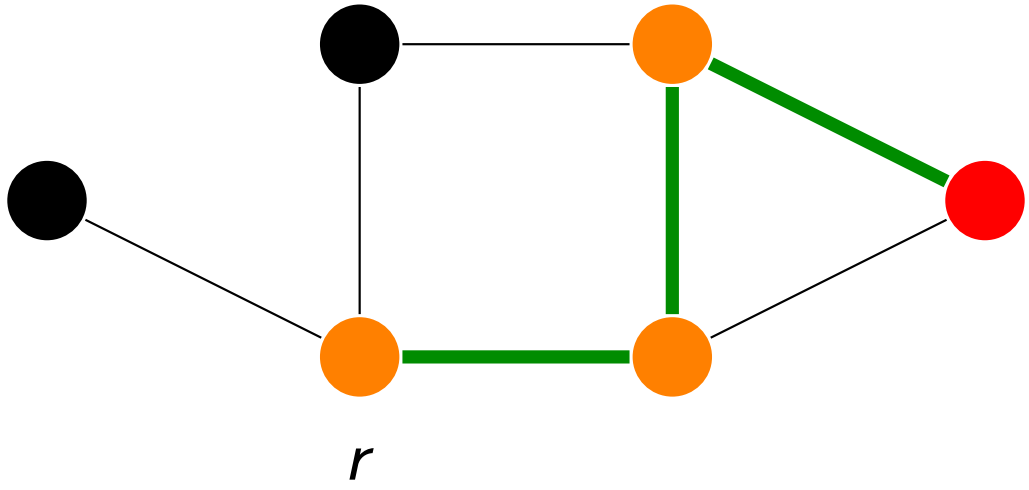


`vu.(v) = false`

`v` est en cours de traitement

appel récursif sur `v` en cours

Parcours en profondeur: exemple

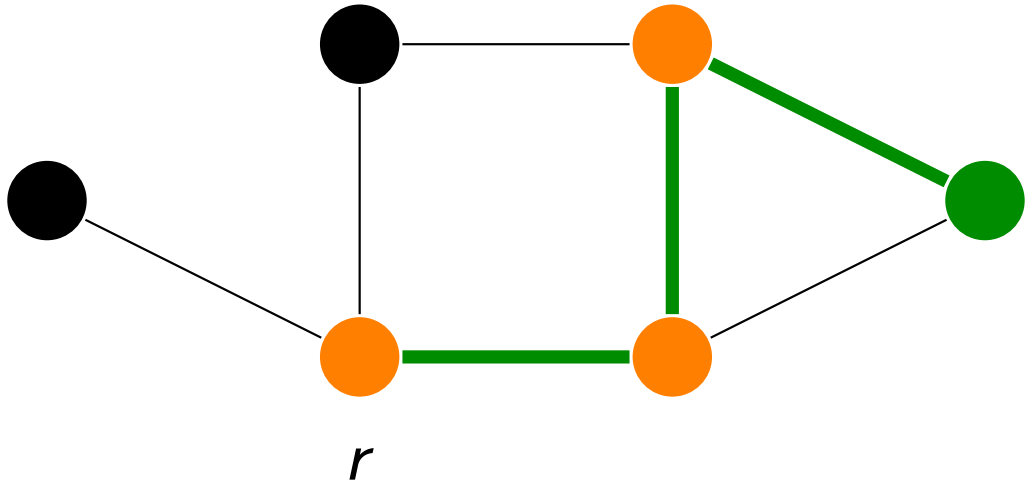


`vu.(v) = false`

`v` est en cours de traitement

appel récursif sur `v` en cours

Parcours en profondeur: exemple

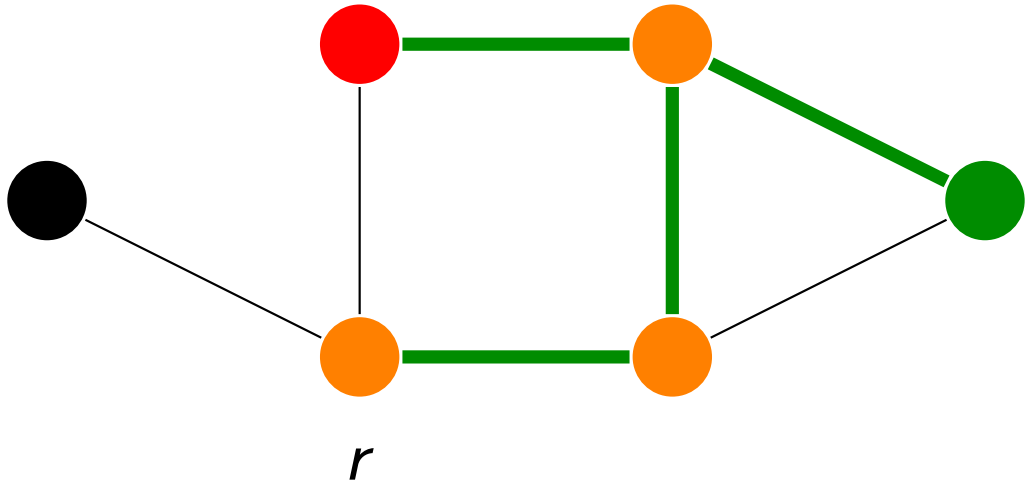


`vu.(v) = false`

`v` est en cours de traitement

appel récursif sur `v` en cours

Parcours en profondeur: exemple

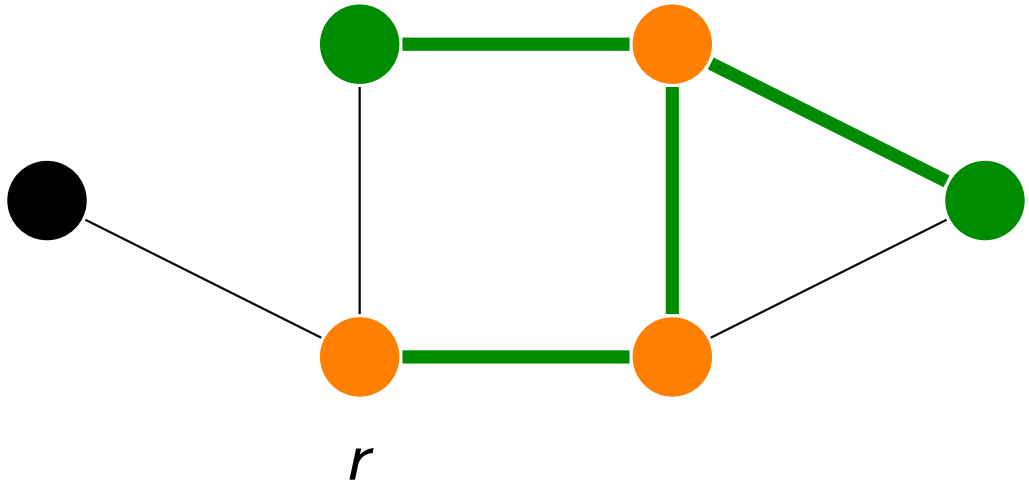


`vu.(v) = false`

`v` est en cours de traitement

appel récursif sur `v` en cours

Parcours en profondeur: exemple

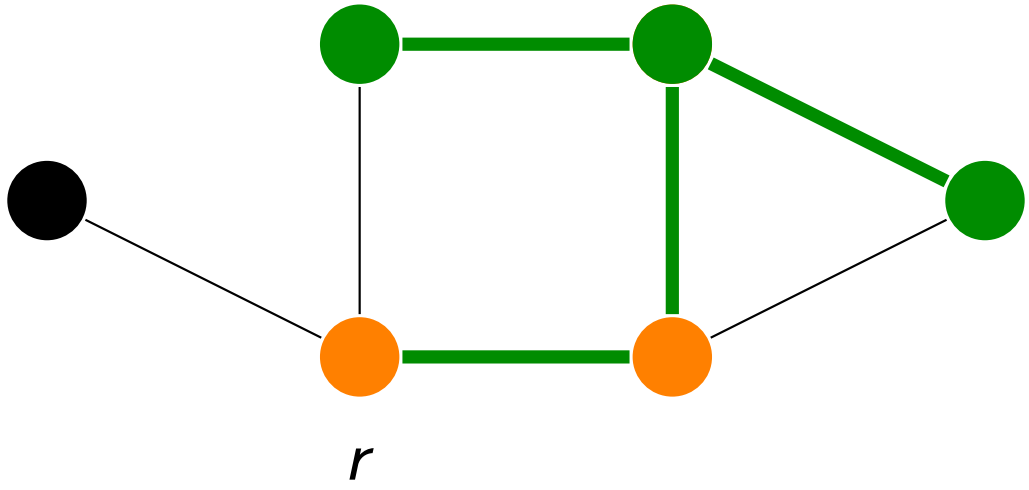


`vu.(v) = false`

`v` est en cours de traitement

appel récursif sur `v` en cours

Parcours en profondeur: exemple

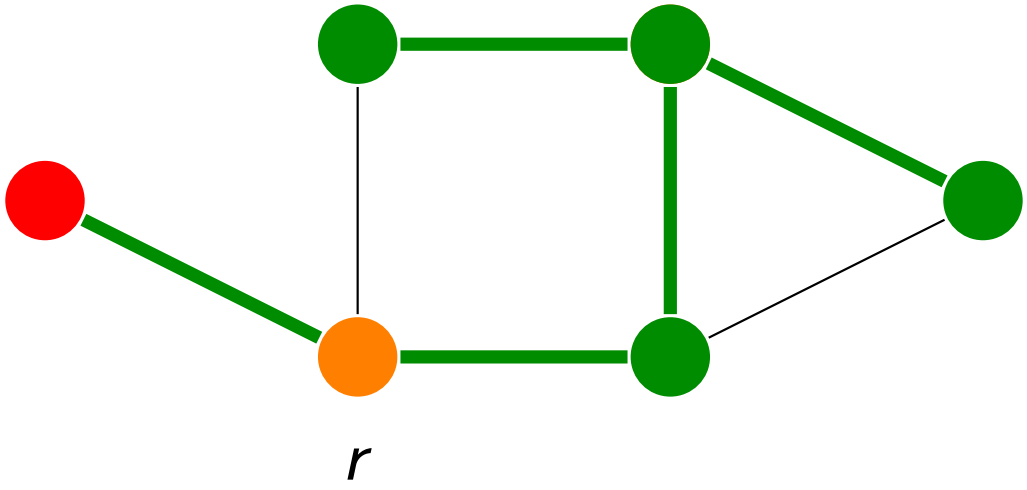


`vu.(v) = false`

`v` est en cours de traitement

appel récursif sur `v` en cours

Parcours en profondeur: exemple

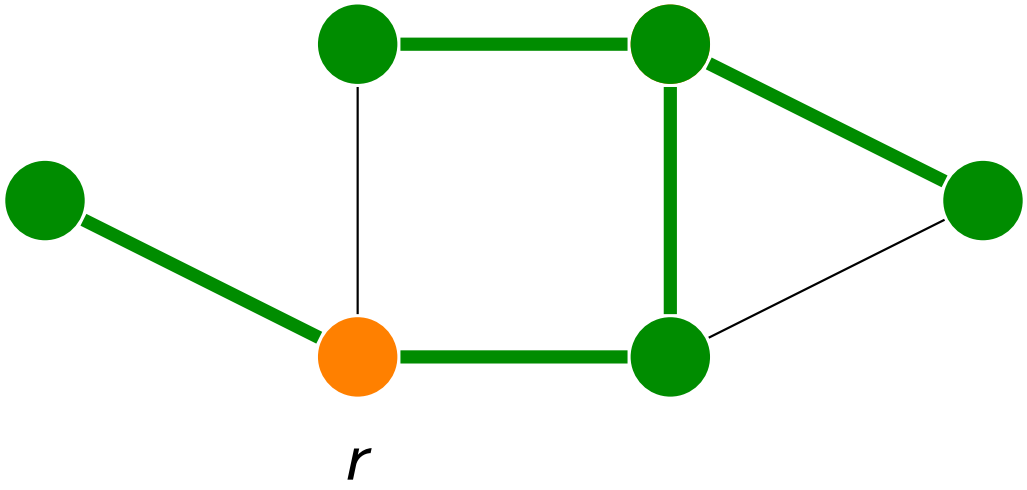


`vu.(v) = false`

`v` est en cours de traitement

appel récursif sur `v` en cours

Parcours en profondeur: exemple

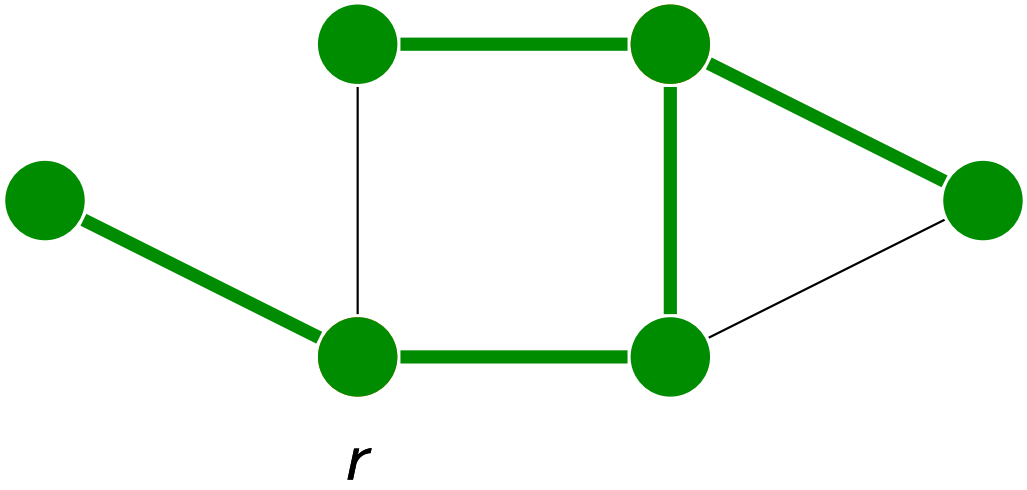


`vu.(v) = false`

`v` est en cours de traitement

appel récursif sur `v` en cours

Parcours en profondeur: exemple

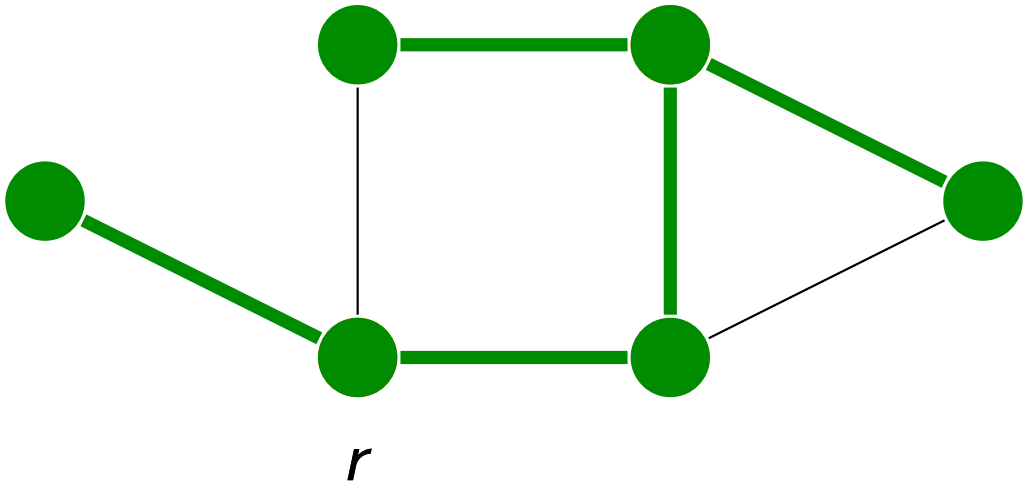


`vu.(v) = false`

`v` est en cours de traitement

appel récursif sur `v` en cours

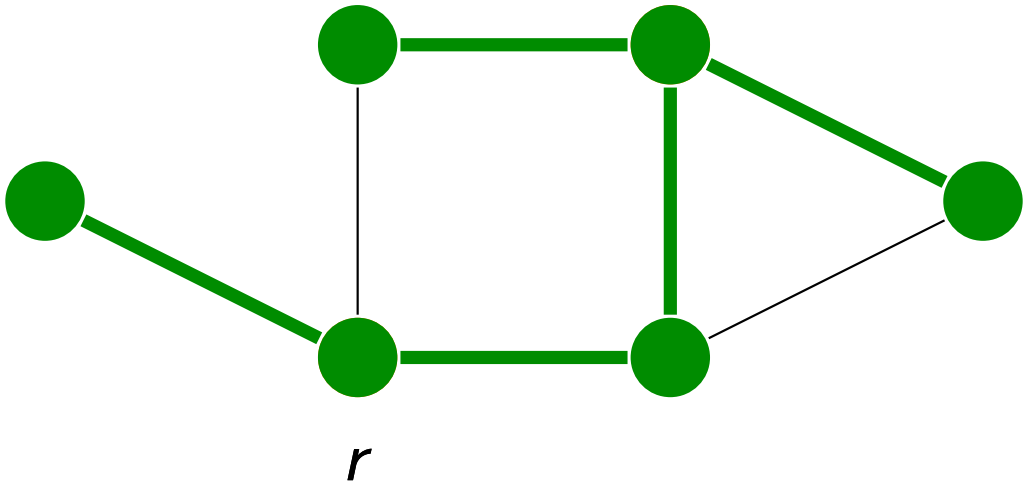
Parcours en profondeur: exemple



Question

Que peut-on dire de l'ensemble des sommets et arêtes visités lors d'un parcours?

Parcours en profondeur: exemple



Question

Que peut-on dire de l'ensemble des sommets et arêtes visités lors d'un parcours?

C'est un **arbre**.

Connexité

```
let dfs g r =
  let vu = Array.make g.n false in
  let rec aux v =
    if not vu.(v) then (vu.(v) <- true;
                        (* traiter v *)
                        do_list aux (g.voisins v)) in
  aux r;;
```

Question

Comment déterminer si un graphe est connexe?

Connexité

```
let dfs g r =
  let vu = Array.make g.n false in
  let rec aux v =
    if not vu.(v) then (vu.(v) <- true;
                        (* traiter v *)
                        do_list aux (g.voisins v)) in
  aux r;;
```

Question

Comment déterminer si un graphe est connexe?

il suffit de vérifier que `vu` ne contient que des `true` après parcours.

Composantes connexes

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

```
let dfs g =
  let vu = Array.make g.n false in
  let rec aux v =
    if not vu.(v) then (vu.(v) <- true;
                        (* traiter v *)
                        do_list aux (g.voisins v)) in
  for r = 0 to g.n - 1 do
    if not vu.(r) then aux r
  done;;
```

Complexité:

Composantes connexes

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

```
let dfs g =
  let vu = Array.make g.n false in
  let rec aux v =
    if not vu.(v) then (vu.(v) <- true;
                        (* traiter v *)
                        do_list aux (g.voisins v)) in
  for r = 0 to g.n - 1 do
    if not vu.(r) then aux r
  done;;
```

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

1. chaque arête donne lieu à au plus 2 appels récursifs de `aux` (1 si orienté), d'où $O(|E|)$ appels récursifs au total
2. chaque sommet donne lieu à un appel à `aux`, pour un total de $|V|$

Déterminer si un graphe non orienté contient un cycle

Question

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

Déterminer si un graphe non orienté contient un cycle

Question

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

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

Déterminer si un graphe non orienté contient un cycle

Question

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

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

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!

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

```
let has_cycle g r =
  let vu = Array.length g.n false in
  let res = ref false in
  let rec aux p u = (* p a permis de découvrir u *)
    if vu.(u) then res := true
    else (vu.(u) <- true;
          do_list (fun v -> if v <> p then aux u v) (g.voisins u)) in
  aux (-1) r;
  !res;;
```

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!

2ème solution: stocker le prédécesseur de chaque sommet dans `pere`.

```
let has_cycle g r =
  let pere = Array.make g.n (-1) in
  let res = ref false in
  let rec aux p v = (* p a permis de découvrir v *)
    if pere.(v) = -1 then (pere.(v) <- p;
                           do_list (aux v) (g.voisins v))
    else if pere.(p) <> v then res := true in
  aux (-1) r;
  !res;;
```

Parcours en profondeur avec une pile

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

```
let dfs g r =
  let vu = Array.make g.n false in
  let p = new_pile () in
  p.push r;
  while not p.is_empty () do
    let u = p.pop () in
    if not vu.(u) then
      (vu.(u) <- true;
       (* traiter u *)
       do_list p.push (g.voisins u))
  done;;
```

Parcours en profondeur avec une pile

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

```
let dfs g r =
  let vu = Array.make g.n false in
  let p = new_pile () in
  p.push r;
  while not p.is_empty () do
    let u = p.pop () in
    if not vu.(u) then
      (vu.(u) <- true;
       (* traiter u *)
       do_list p.push (g.voisins 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.

Parcours en profondeur avec une pile

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

```
let dfs g r =
  let vu = Array.make g.n false in
  let p = new_pile () in
  p.push r;
  while not p.is_empty () do
    let u = p.pop () in
    if not vu.(u) then
      (vu.(u) <- true;
       (* traiter u *)
       do_list p.push (g.voisins 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.

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

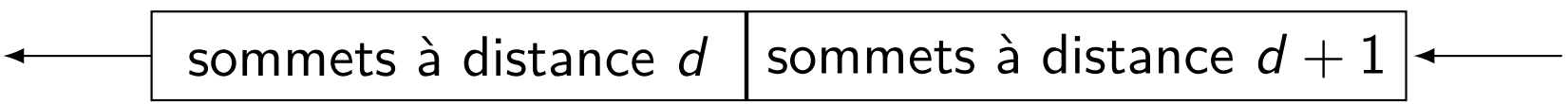
Parcours en largeur (BFS) avec une file

```
let bfs g r =
  let vu = Array.make g.n false in
  let f = file_new () in
  let add v =
    if not vu.(v) then (vu.(v) <- true; f.add v) in
  add r;
  while not f.is_empty () do
    let u = f.take () in
    (* traiter u *)
    do_list add (g.voisins u)
  done;;
```

Parcours en largeur (BFS) avec une file

```
let bfs g r =
  let vu = Array.make g.n false in
  let f = file_new () in
  let add v =
    if not vu.(v) then (vu.(v) <- true; f.add v) in
  add r;
  while not f.is_empty () do
    let u = f.take () in
    (* traiter u *)
    do_list add (g.voisins u)
  done;;
```

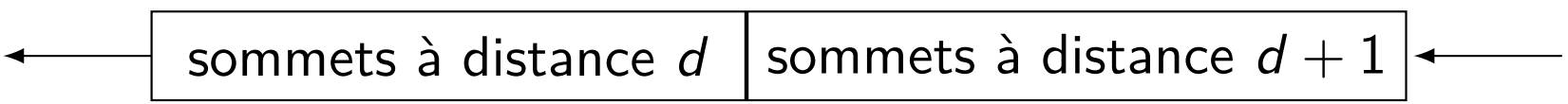
La file `f` est toujours de la forme:



Parcours en largeur (BFS) avec une file

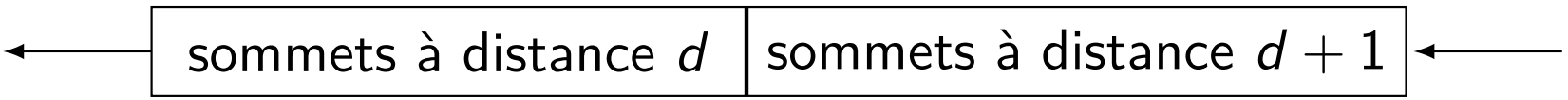
```
let bfs g r =
  let vu = Array.make g.n false in
  let f = file_new () in
  let add v =
    if not vu.(v) then (vu.(v) <- true; f.add v) in
  add r;
  while not f.is_empty () do
    let u = f.take () in
    (* traiter u *)
    do_list add (g.voisins u)
  done;;
```

La file `f` est toujours de la forme:



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 deux couches



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

```
let bfs g r =
  let vu = Array.make g.n false in
  let rec aux cur next = match cur with
    | [] -> if next <> [] then aux next [] (* couche suivante *)
    | v::q when vu.(v) -> aux q next
    | v::q -> (vu.(v) <- true;
               (* traiter v *)
               aux q (g.voisins v @ next)) in
  aux [r] [];;
```

Distance

Question

Comment connaître la distance d'un sommet `r` aux autres?

Distance

Question

Comment connaître la distance d'un sommet `r` aux autres?

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

Distance

Question

Comment connaître la distance d'un sommet `r` aux autres?

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 dist
            else aux (d+1) next []
    | v::q when dist.(v) <> -1 -> aux d q next
    | v::q -> (dist.(v) <- d;
               aux d q (g.voisins v @ next)) in
  aux 0 [r] [];;
```

Complexité:

Distance

Question

Comment connaître la distance d'un sommet `r` aux autres?

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 dist
            else aux (d+1) next []
    | v::q when dist.(v) <> -1 -> aux d q next
    | v::q -> (dist.(v) <- d;
               aux d q (g.voisins v @ next)) in
  aux 0 [r] [];;
```

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

Plus court chemin

Question

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

Plus court chemin

Question

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

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

Plus court chemin

Question

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

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

```
let bfs g r =
  let pere = Array.make g.n (-1) in
  let f = file_new () in
  let add p v = (* p est le pere de v *)
    if pere.(v) = -1 then (pere.(v) <- p; f.add v) in
  add r r;
  while not f.is_empty () do
    let u = f.take () in
    do_list (add u) (g.voisins u)
  done;
  pere;;
```

Complexité:

Plus court chemin

Question

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

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

```
let bfs g r =
  let pere = Array.make g.n (-1) in
  let f = file_new () in
  let add p v = (* p est le pere de v *)
    if pere.(v) = -1 then (pere.(v) <- p; f.add v) in
  add r r;
  while not f.is_empty () do
    let u = f.take () in
    do_list (add u) (g.voisins u)
  done;
  pere;;
```

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

Plus court chemin

```
let bfs g r =
  let pere = Array.make g.n (-1) in
  let f = file_new () in
  let add p v = (* p est le pere de v *)
    if pere.(v) = -1 then (pere.(v) <- p; f.add v) in
  add r r;
  while not f.is_empty () do
    let u = f.take () in
    do_list (add u) (g.voisins u)
  done;
  pere;;
```

Question

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

Plus court chemin

```
let bfs g r =
  let pere = Array.make g.n (-1) in
  let f = file_new () in
  let add p v = (* p est le pere de v *)
    if pere.(v) = -1 then (pere.(v) <- p; f.add v) in
  add r r;
  while not f.is_empty () do
    let u = f.take () in
    do_list (add u) (g.voisins u)
  done;
  pere;;
```

Question

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

```
let rec chemin pere v =
  if pere.(v) = v then [v]
  else v::(chemin pere pere.(v));;
```

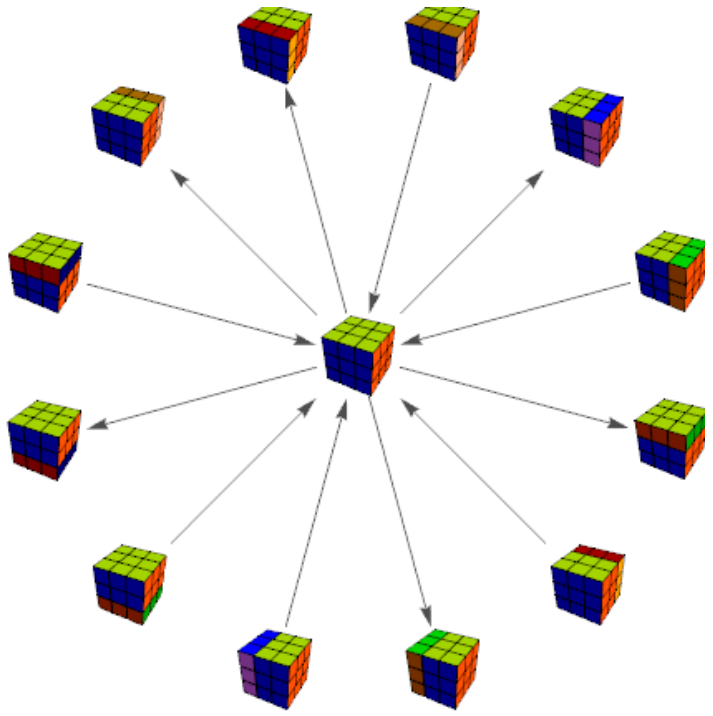
Plus court chemin

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

Plus court chemin

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

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



Plus court chemin

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 maximum 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.