Graphes : Parcours

Quentin Fortier

February 1, 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 (
            vu.(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 (
      vu.(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)

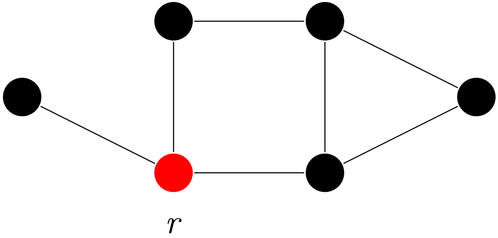
Le parcours en profondeur sur $G = (V, E)$ depuis une racine r consiste, **si r n'a pas déjà été visité**, à le traiter puis s'appeler récursivement sur ses voisins :

```
let dfs g r =
  let seen = Array.make g.n false in
  let rec aux v =
    if not seen.(v) then (
      vu.(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

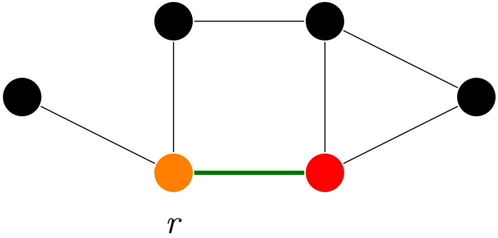


`vu.(v) = false`

`v` est en cours de traitement

appel récursif sur `v` en cours

Parcours en profondeur (DFS) : Exemple

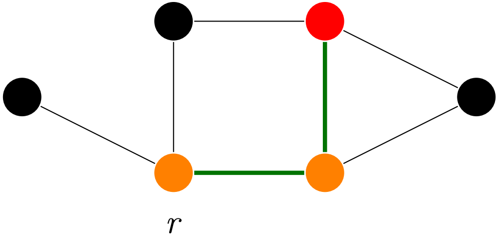


`vu.(v) = false`

`v` est en cours de traitement

appel récursif sur `v` en cours

Parcours en profondeur (DFS) : Exemple

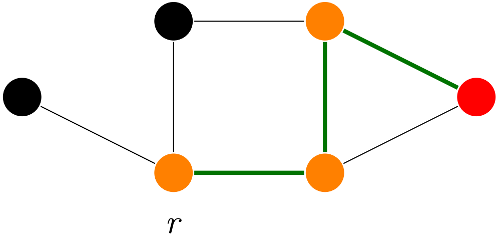


 `vu.(v) = false`

 `v` est en cours de traitement

 appel récursif sur `v` en cours

Parcours en profondeur (DFS) : Exemple

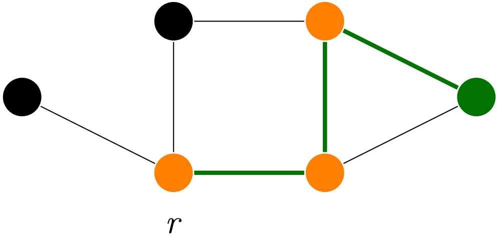


 `vu.(v) = false`

 `v` est en cours de traitement

 appel récursif sur `v` en cours

Parcours en profondeur (DFS) : Exemple

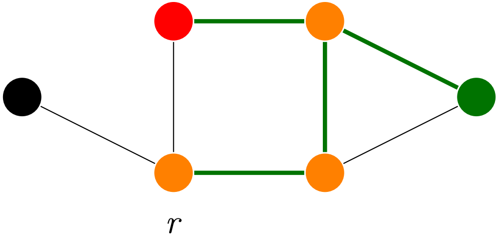


`vu.(v) = false`

`v` est en cours de traitement

appel récursif sur `v` en cours

Parcours en profondeur (DFS) : Exemple

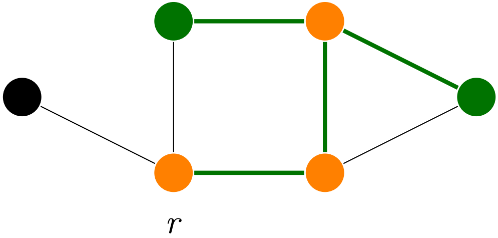


 `vu.(v) = false`

 `v` est en cours de traitement

 appel récursif sur `v` en cours

Parcours en profondeur (DFS) : Exemple

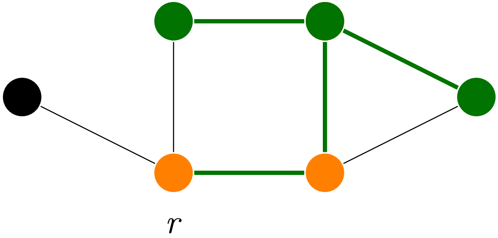



`vu.(v) = false`


`v` est en cours de traitement


appel récursif sur `v` en cours

Parcours en profondeur (DFS) : Exemple

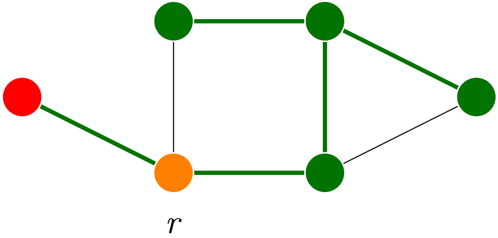


`vu.(v) = false`

`v` est en cours de traitement

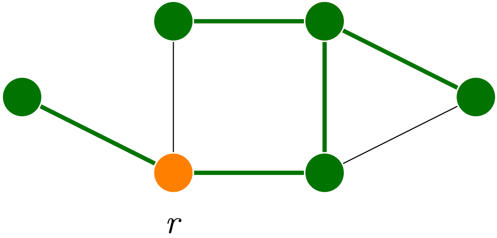
appel récursif sur `v` en cours

Parcours en profondeur (DFS) : Exemple



- `vu.(v) = false`
- `v` est en cours de traitement
- appel récursif sur `v` en cours

Parcours en profondeur (DFS) : Exemple

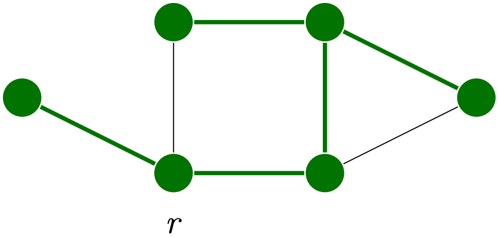


 `vu.(v) = false`

 `v` est en cours de traitement

 appel récursif sur `v` en cours

Parcours en profondeur (DFS) : Exemple



- `vu.(v) = false`
- `v` est en cours de traitement
- appel récursif sur `v` en cours

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 les tableaux des vus (`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 =
  let vu = make_vect g.n false in
  let rec aux v =
    if not vu.(v) then (vu.(v) <- true;
                        do_list aux (g.voisins v)) in
  for r = 0 to g.n - 1 do
    aux r;
  done;;
```

Complexité :

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

Complexité : $\boxed{\mathrm{O}(|V| + |E|)}$ 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 $\mathrm{O}(|E|)$ appels récursifs au total
2. chaque sommet donne lieu à un appel à `aux`, pour un total de $|V|$

Parcours en profondeur (DFS) : Application à la connexité

On peut ainsi déterminer les différentes composantes connexes d'un graphe non orienté...

Parcours en profondeur (DFS) : Application à la connexité

On peut ainsi déterminer les différentes composantes connexes d'un graphe non orienté...

On peut aussi utiliser un Union-Find, possédant les opérations :

- `uf_new n` : crée une partition de $\{0, ..., n-1\}$ où chaque élément est seul dans sa classe.
- `uf_find uf i` : renvoie un représentant de la classe de `i`.
- `uf_union uf i j` : réunie les classes de `i` et `j`.

Parcours en profondeur (DFS) : Application à la connexité

On peut ainsi déterminer les différentes composantes connexes d'un graphe non orienté...

On peut aussi utiliser un Union-Find, possédant les opérations :

- `uf_new n` : crée une partition de $\{0, ..., n-1\}$ où chaque élément est seul dans sa classe.
- `uf_find uf i` : renvoie un représentant de la classe de `i`.
- `uf_union uf i j` : réunie les classes de `i` et `j`.

```
let comp_connexes g =
  let uf = uf_new g.n in
  for u = 0 to g.n - 1 do
    do_list (uf_union uf u) (g.voisins u)
  done;
  uf;;
```

Parcours en profondeur (DFS) : Application à la connexité

```
let comp_connexes g =
  let uf = uf_new g.n in
  for u = 0 to g.n - 1 do
    do_list (uf_union uf u) (g.voisins u)
  done;
  uf;;
```

Deux sommets `u`, `v` sont alors dans la même composante connexe ssi `uf_find uf u` = `uf_find uf v`.

Avantage de l'Union-Find : on peut mettre à jour facilement les composantes connexes si on rajoute des arêtes.

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?

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

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

```
let has_cycle g r =
  let vu = make_vect g.n false in
  let res = ref false in
  let rec aux p u = (* p a permis de découvrir v *)
    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;;
```

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!

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

```
let has_cycle g r =
  let pere = make_vect 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 (DFS) : Détection de cycle

Question

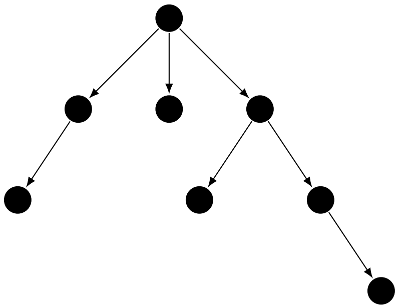
Comment déterminer si un graphe **orienté** $\vec{G} = (V, \vec{E})$ contient un cycle?

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 $\vec{G}$.

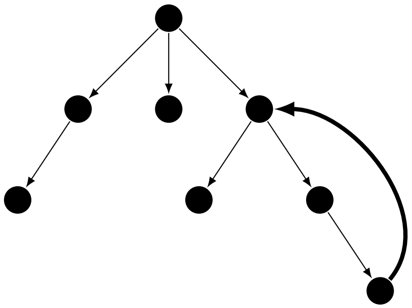


Parcours en profondeur (DFS) : Détection de cycle

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 $\overrightarrow{G}$ depuis r :

Théorème

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

$$\iff$$

A possède un arc arrière

Parcours en profondeur (DFS) : Détection de cycle

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

Théorème

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

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

Parcours en profondeur (DFS) : Détection de cycle

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

Théorème

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

<u>Preuve</u>:

$\Longleftarrow$: évident.

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

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 $\vec{\mathcal{C}}$ 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 $\vec{\mathcal{C}} = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_k \rightarrow v_0$.

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 vu = make_vect g.n 0 in
  let res = ref false in
  let rec aux v =
    if vu.(v) = 0 then (vu.(v) <- 1;
                        do_list aux (g.voisins v);
                        vu.(v) <- 2)
    else if vu.(v) = 1 then res := true in
  aux r;
  !res;;
```

`vu.(v) = 0` $\iff$ `v` non visité

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

`vu.(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 dfs g r =
  let vu = make_vect 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;
       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 (DFS) : Avec pile

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

```
let dfs g r =
  let vu = make_vect 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;
       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 file

```
let bfs g r =
  let vu = make_vect 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 file

```
let bfs g r =
  let vu = make_vect 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:

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

Parcours en largeur (BFS) : Avec file

```
let bfs g r =
  let vu = make_vect 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:

←	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 vu = make_vect 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;
                 aux q (g.voisins v @ next))
  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 = make_vect 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 :

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 = make_vect 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.

Parcours en largeur (BFS) : Application au calcul de plus court chemin

Question

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

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

Parcours en largeur (BFS) : Application au calcul de 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 = make_vect 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 :

Parcours en largeur (BFS) : Application au calcul de 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 = make_vect 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.

Parcours en largeur (BFS) : Application au calcul de plus court chemin

```
let bfs g r =
  let pere = make_vect 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?

Parcours en largeur (BFS) : Application au calcul de plus court chemin

```
let bfs g r =
  let pere = make_vect 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));;
```

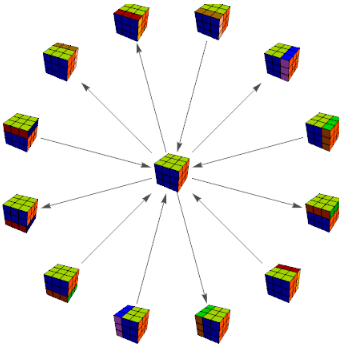
Parcours en largeur (BFS) : Application au calcul de plus court chemin

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

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



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