

Graphes : Parcours en largeur (BFS)

Quentin Fortier

September 17, 2023

Une **file** est une structure de donnée possédant les opérations :

- Ajout d'un élément à la fin de la file.
- Extraction (suppression et renvoi) de l'élément au début de la file.
Ainsi, c'est toujours l'élément le plus ancien qui est extrait.



Une **file** est une structure de donnée possédant les opérations :

- Ajout d'un élément à la fin de la file.
- Extraction (suppression et renvoi) de l'élément au début de la file.
Ainsi, c'est toujours l'élément le plus ancien qui est extrait.



Une file est aussi appelée structure FIFO (*First In, First Out*) alors qu'une pile est LIFO (*Last In, First Out*).

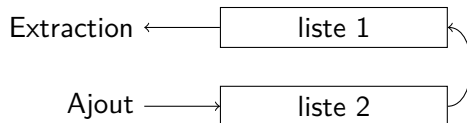
Exercice

Donner des implémentations possibles de file.

Exercice

Donner des implémentations possibles de file.

- Avec 2 listes :



Parcours en largeur (BFS) : Avec file

Parcours en largeur avec file q :

```
let bfs (g : int list array) (r : int) =  
  let seen = Array.make (Array.length g) false in  
  let q = Queue.create () in  
  let add v =  
    if not seen.(v) then (  
      seen.(v) <- true; Queue.add v q  
    ) in  
  add r;  
  while not (Queue.is_empty q) do  
    let u = Queue.pop q in  
    (* traitez u *)  
    List.iter add g.(u)  
  done
```

Parcours en largeur (BFS) : Avec file

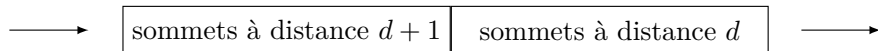
Parcours en largeur avec file q :

```
let bfs (g : int list array) (r : int) =  
  let seen = Array.make (Array.length g) false in  
  let q = Queue.create () in  
  let add v =  
    if not seen.(v) then (  
      seen.(v) <- true; Queue.add v q  
    ) in  
  add r;  
  while not (Queue.is_empty q) do  
    let u = Queue.pop q in  
    (* traitez u *)  
    List.iter add g.(u)  
  done
```

Ressemble beaucoup au DFS avec pile.

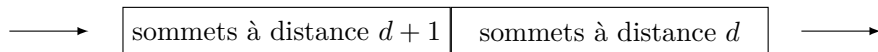
Parcours en largeur (BFS) : Avec file

La file q est toujours de la forme :



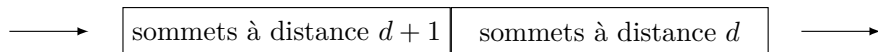
Parcours en largeur (BFS) : Avec file

La file q est toujours de la forme :



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

Parcours en largeur (BFS) : Avec 2 couches



Une variante du BFS utilise deux listes : `cur` pour la couche courante, `next` pour la couche suivante.

```
let bfs g r =  
  let seen = Array.make (Array.length g) 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.(u)) in  
  aux [r] []
```

Application au calcul de distance

Question

Comment connaître la distance d'un sommet s aux autres?

Application au calcul de distance

Question

Comment connaître la distance d'un sommet s aux autres?

Stocker des couples (sommet, distance) dans la file :

Application au calcul de distance

Question

Comment connaître la distance d'un sommet s aux autres?

Stocker des couples (sommet, distance) dans la file :

```
let bfs g r =  
  let dist = Array.make (Array.length g) (-1) in  
  let q = Queue.create () in  
  let add d v =  
    if dist.(v) = -1 then (  
      dist.(v) <- d;  
      Queue.add (v, d) q  
    ) in  
  add 0 r;  
  while not (Queue.is_empty q) do  
    let u, d = Queue.pop q in  
    List.iter (add (d + 1)) g.(u)  
  done;  
  dist (* dist.(u) est la distance de r à u *)
```

Théorème

bfs g r renvoie un tableau `dist` tel que, pour tout sommet v accessible depuis r , $\text{dist}.(v) = d(r, v)$

Preuve : Par invariant de boucle : voir le poly de Jean-Baptiste Bianquis.

Application au calcul de distance

Question

Comment connaître la distance d'un sommet x aux autres?

En utilisant un BFS avec deux couches :

Application au calcul de distance

Question

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

En utilisant un BFS avec deux couches :

```
let bfs g r =  
  let dist = Array.make (Array.length g) (-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.(u) @ next) in  
  aux 0 [r] [];  
  dist (* dist.(u) est la distance de r à u *)
```

Application au calcul de distance

Remarque : le parcours en largeur ne sert à trouver les distances que si le graphe est non pondérée, et où la longueur d'un chemin est son nombre d'arêtes. Sur les graphes pondérés, il faut utiliser Dijkstra/Floyd-Warshall/Bellman-Ford...

Application au calcul de distance : Plus courts chemins

Question

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

Application au calcul de distance : 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 (Array.length g) (-1) in  
  let q = Queue.create () in  
  let add p v = (* p est le père de v *)  
    if pred.(v) = -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  
    List.iter (add u) g.(u)  
  done;  
  pred (* pred.(v) est le prédécesseur de v dans le parcours en
```

Application au calcul de distance : Plus courts chemins

Question

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

Application au calcul de distance : Plus courts chemins

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)
```
