Chapitre 2 : Les graphes

Option Informatique – MP

Lycée Chateaubriand

Table des matières

Plan

Définition

On a vu le parcours des arbres, dans le cas des graphes c'est un peu plus compliqué car il n'y a pas de sens évident et **surtout il faut éviter de se retrouver piéger dans un éventuel cycle du graphe.**

Définition

On a vu le parcours des arbres, dans le cas des graphes c'est un peu plus compliqué car il n'y a pas de sens évident et **surtout il faut éviter de se retrouver piéger dans un éventuel cycle du graphe.**

Définition 1 (Parcours d'un graphe)

Un parcours d'un graphe (S, A) *est une liste sans répétitions* $(x_1, \dots x_p)$ *de sommets du graphe telle que*

$$\forall i \in [\![1, p-1]\!], \exists j \in [\![1, i]\!], (x_j, x_{i+1}) \in A.$$

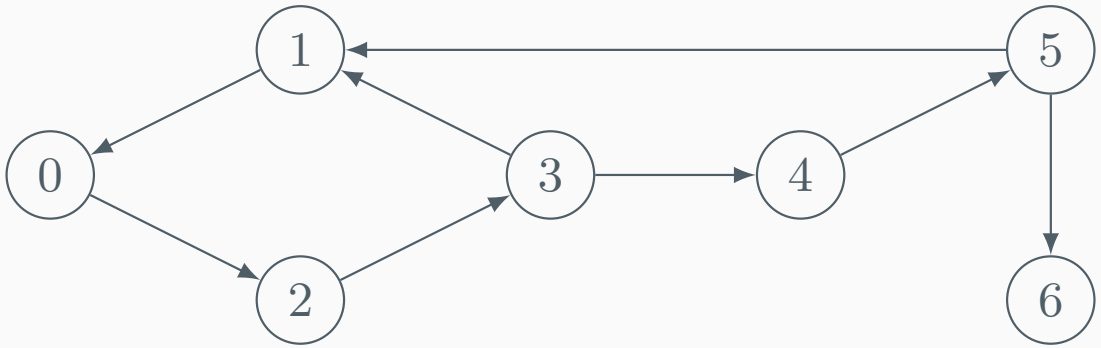
Définition

On a vu le parcours des arbres, dans le cas des graphes c'est un peu plus compliqué car il n'y a pas de sens évident et **surtout il faut éviter de se retrouver piéger dans un éventuel cycle du graphe.**

Définition 1 (Parcours d'un graphe)

Un parcours d'un graphe (S, A) *est une liste sans répétitions* $(x_1, \dots x_p)$ *de sommets du graphe telle que*

$$\forall i \in [\![1, p-1]\!], \exists j \in [\![1, i]\!], (x_j, x_{i+1}) \in A.$$



On pourra parcourir en faisant $0, 2, 3, 1, 4, 5, 6$ ou $0, 2, 3, 4, 5, 6, 1$.

On remarque que (x_i, x_{i+1}) n'est pas nécessairement une arête.

Méthodes de parcours

Il y a essentiellement deux manières de voir les parcours.

Méthodes de parcours

Il y a essentiellement deux manières de voir les parcours.

- On se donne un sommet x_1 qui sera l'origine et on parcourt le graphe tant que l'on peut ajouter des sommets.

Méthodes de parcours

Il y a essentiellement deux manières de voir les parcours.

- On se donne un sommet x_1 qui sera l'origine et on parcourt le graphe tant que l'on peut ajouter des sommets.
- On se donne un sommet x_1 qui sera l'origine et un sommet x_p qui sera l'arrivée et on parcourt le graphe tant que l'on n' est pas arrivé à x_p. Il se peut que l'on ne puisse pas y arriver. On dit alors que le parcours à échoué.

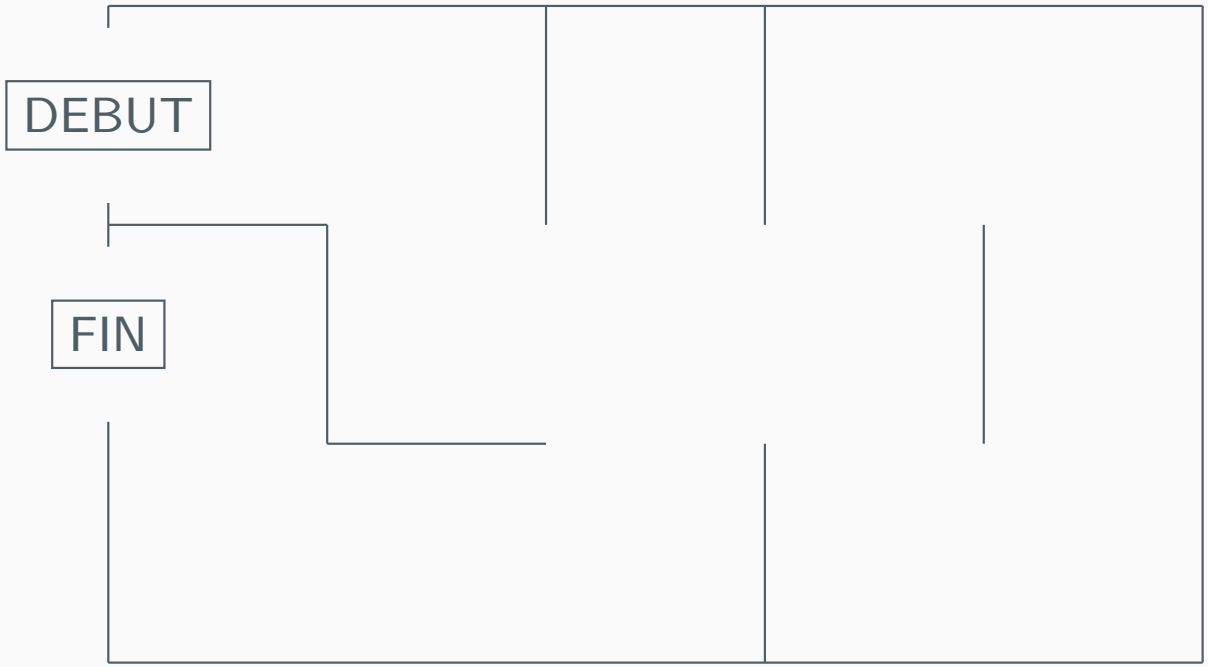
Méthodes de parcours

Il y a essentiellement deux manières de voir les parcours.

- On se donne un sommet x_1 qui sera l'origine et on parcourt le graphe tant que l'on peut ajouter des sommets.
- On se donne un sommet x_1 qui sera l'origine et un sommet x_p qui sera l'arrivée et on parcourt le graphe tant que l'on n' est pas arrivé à x_p. Il se peut que l'on ne puisse pas y arriver. On dit alors que le parcours à échoué.
- Nous allons avoir différents parcours qui se caractérisent par l'ordre dans lequel on parcourt les sommets.

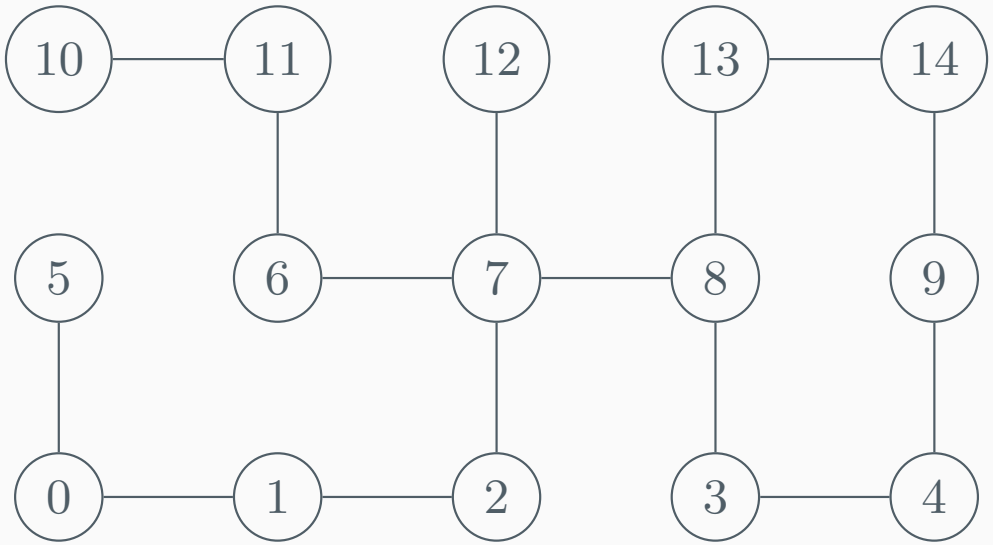
Exemple

Un exemple classique de parcours de graphe est le problème d'un labyrinthe. On considère le labyrinthe suivant :



Exemple

On peut associer à ce labyrinthe un graphe (non orienté car les chemins peuvent être parcourus dans les deux sens).



Trouver une solution à ce labyrinthe, revient à déterminer un parcours du graphe qui commence au sommet 10 et qui arrive au sommet 5.

Principe général

Attention

Le principe général des parcours est de partager les sommets du graphe en trois.

Principe général

Attention

Le principe général des parcours est de partager les sommets du graphe en trois.

- Les sommets qui ont été visités

Principe général

Attention

Le principe général des parcours est de partager les sommets du graphe en trois.

- Les sommets qui ont été visités
- Les sommets qui sont des voisins de sommets ayant été visités mais qui n'ont pas été visités eux même. On parle de la frontière

Principe général

Attention

Le principe général des parcours est de partager les sommets du graphe en trois.

- Les sommets qui ont été visités
- Les sommets qui sont des voisins de sommets ayant été visités mais qui n'ont pas été visités eux même. On parle de la frontière
- Les sommets qui n'ont pas encore été considérés.

Principe général

On applique alors un algorithme du type suivant :

Tant que la frontière n'est pas vide, on choisit un sommet de la frontière, s'il n'a pas déjà été visité, il passe dans les sommets visités et on ajoute à la frontière tous ses voisins.

Principe général

En notant F les
sommets de la frontière, Visit les sommets visités et, pour tout sommet x, V_x ses voi-

Algorithme 1 : parcours d'un graphe

Entrée Graphe $G = (S, A)$; sommet de départ s

:

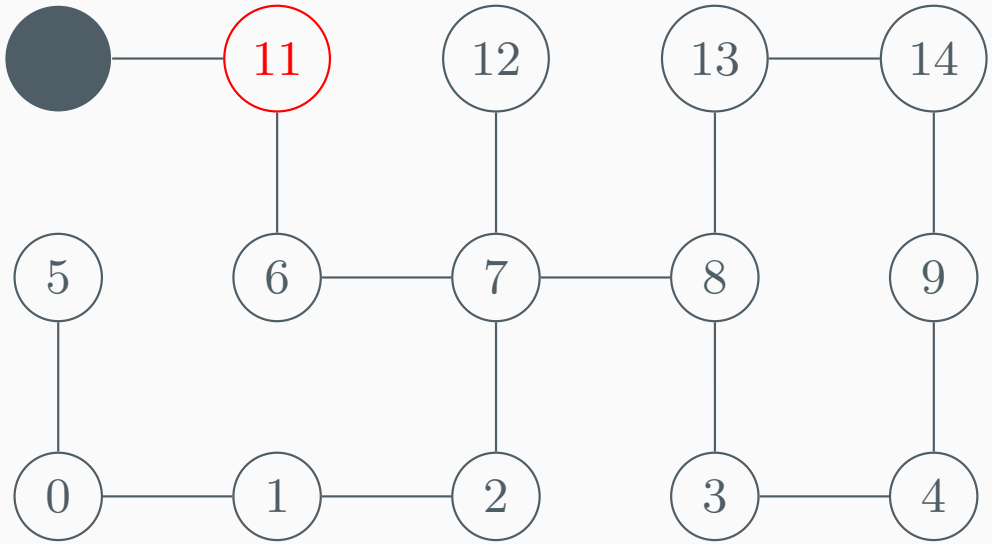
```
1 F ← {s}
2 Visit ← ∅
3 tant que F ≠ ∅ faire
4 |   Choisir un sommet x dans F (et l'enlever)
5 |   si x ∉ Visit alors
6 |   |   Ajouter les sommets de V_x à F
7 |   |   Ajouter x à Visit
8 |   fin
9 fin
```

sins :

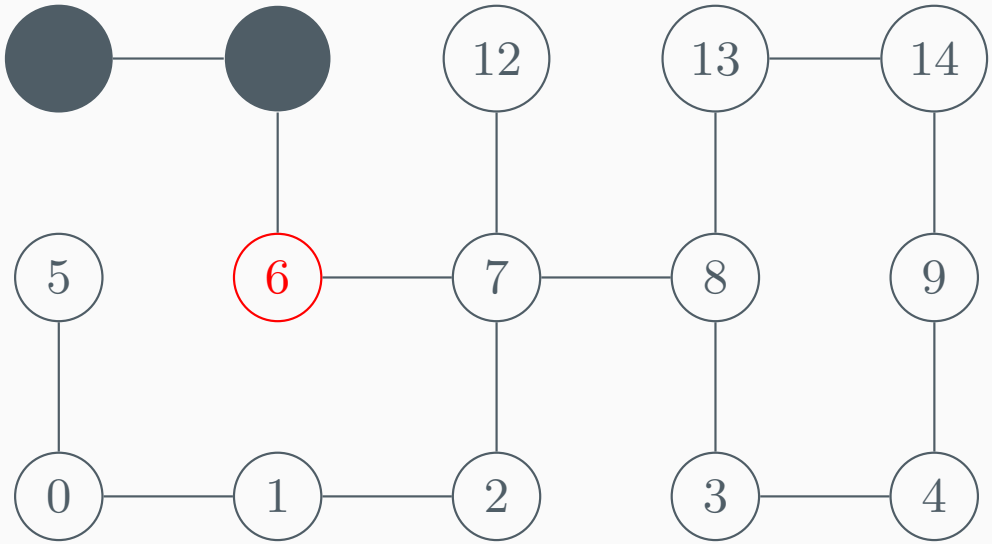
Remarque

On peut aussi se contenter d'ajouter à la frontière seulement les voisins qui n'ont pas été visités.

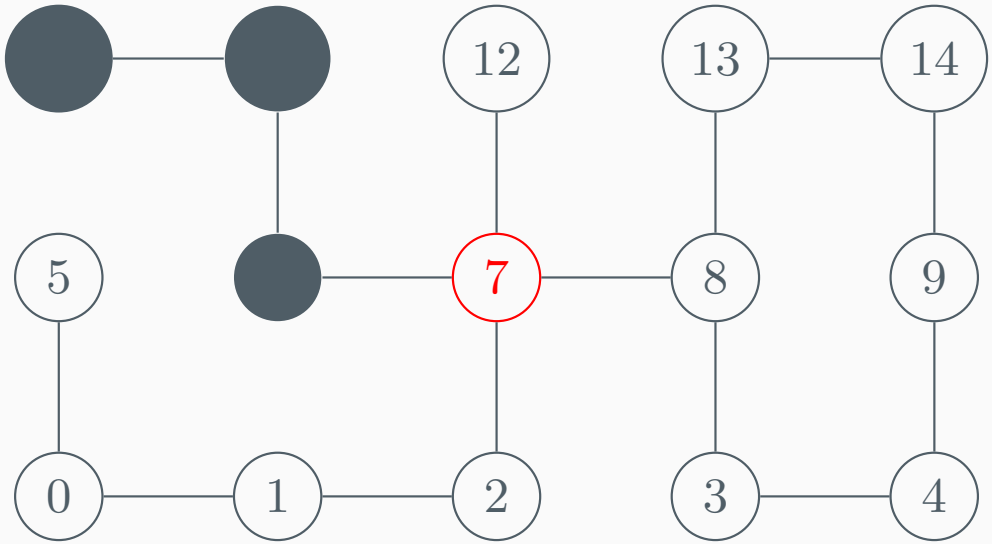
Exemple



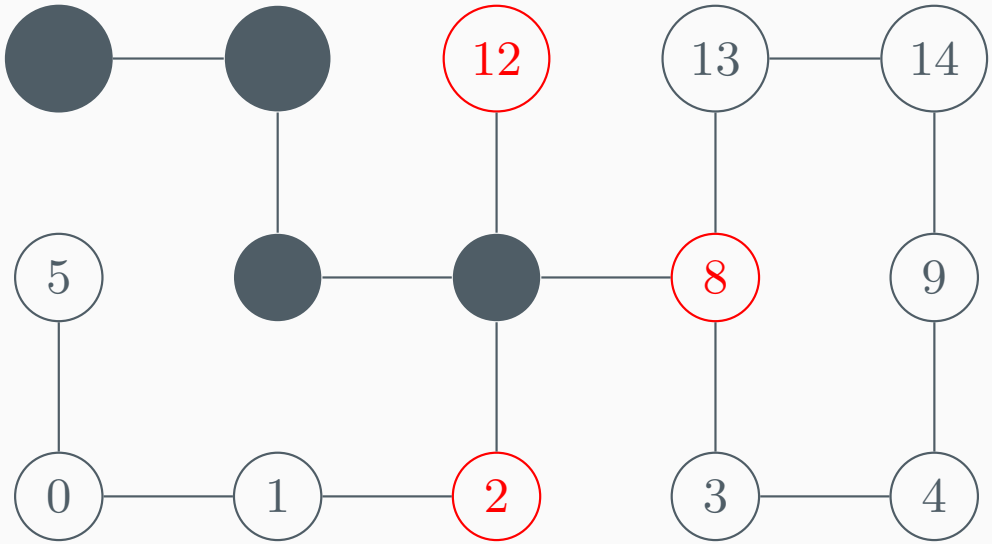
Exemple



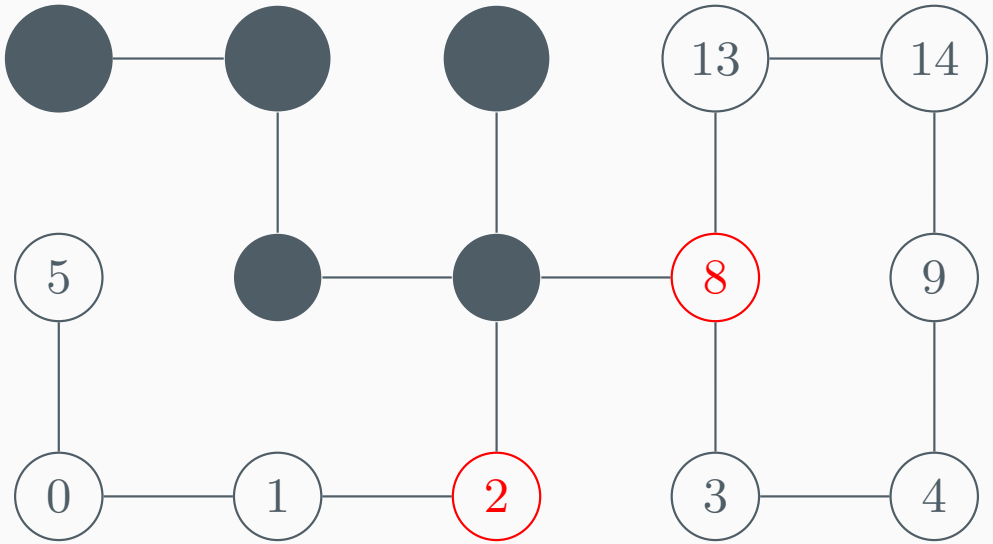
Exemple



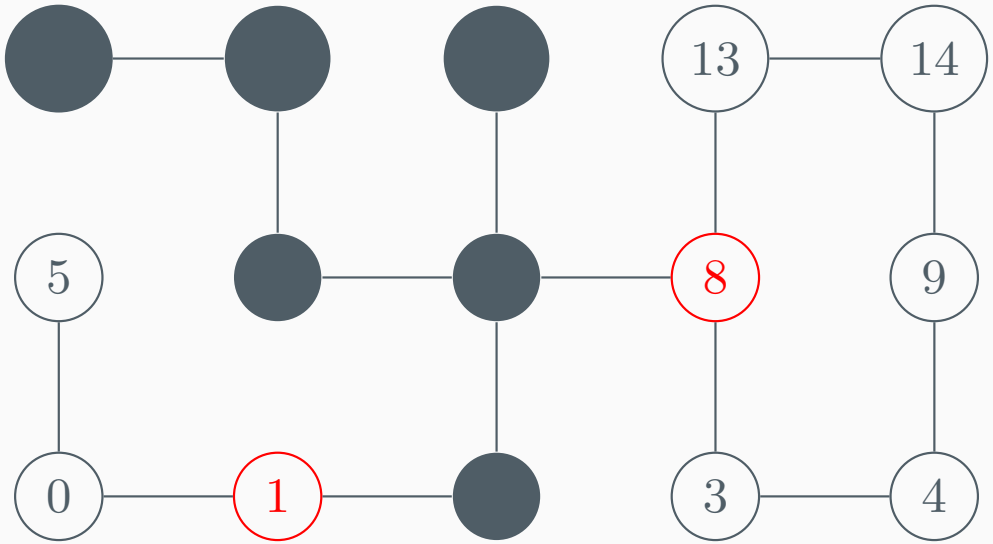
Exemple



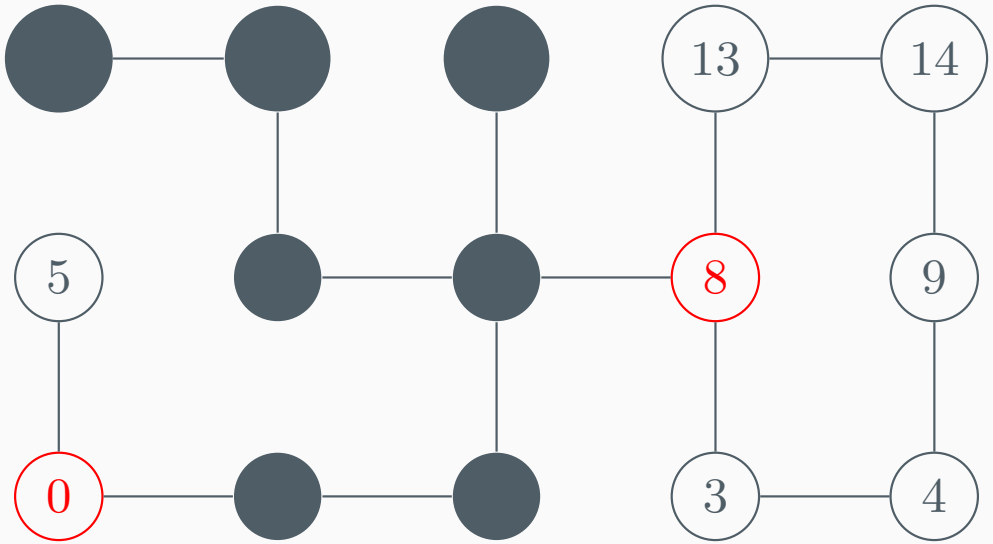
Exemple



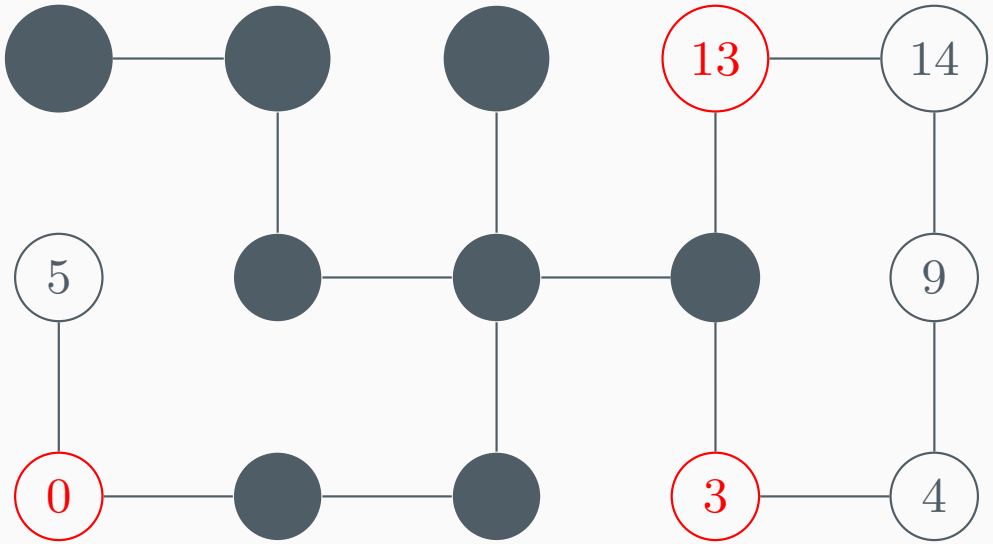
Exemple



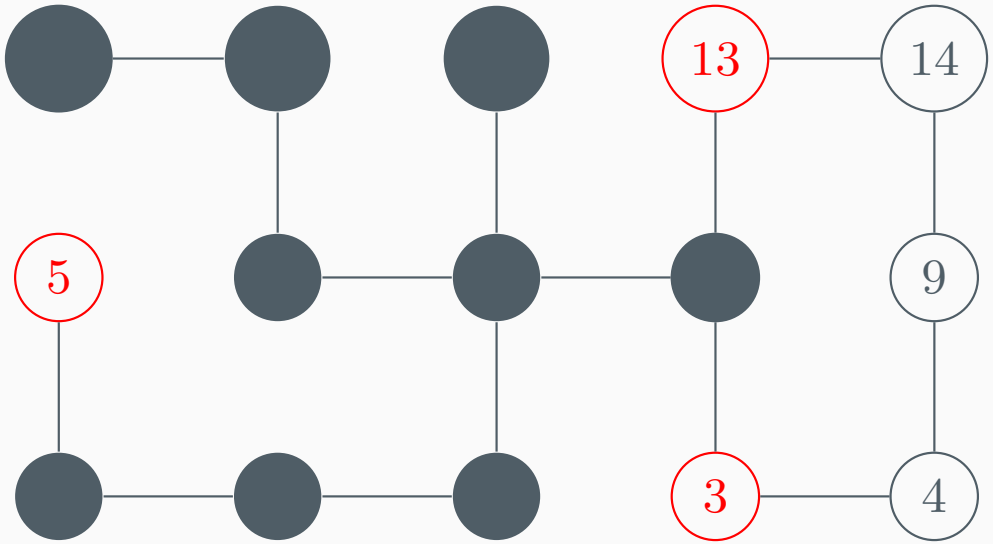
Exemple



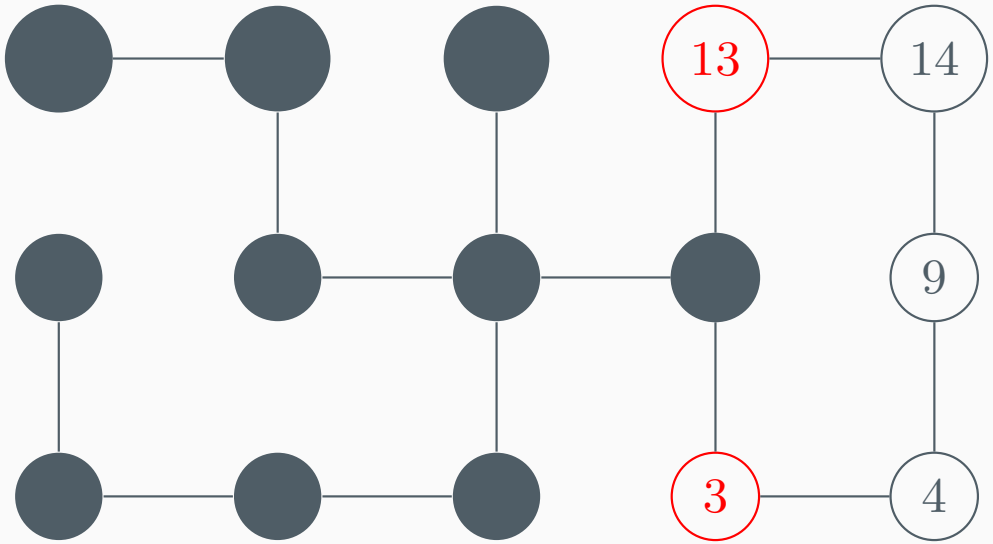
Exemple



Exemple



Exemple



Parcours en profondeur (DFS)

Parcours en profondeur

Le principe du parcours en profondeur que l'on l'appelle aussi DFS pour *Depth First Search* est de parcourir **en premier un voisin du sommet où l'on arrive.**

On peut l'implémenter de deux façons.

Parcours en profondeur

Le principe du parcours en profondeur que l'on l'appelle aussi DFS pour *Depth First Search* est de parcourir **en premier un voisin du sommet où l'on arrive.**

On peut l'implémenter de deux façons.

- On peut se contenter d'une simple fonction récursive qui se rappelle sur tous les voisins non visités du sommet où l'on se situe

Parcours en profondeur

Le principe du parcours en profondeur que l'on l'appelle aussi DFS pour *Depth First Search* est de parcourir **en premier un voisin du sommet où l'on arrive.**

On peut l'implémenter de deux façons.

- On peut se contenter d'une simple fonction récursive qui se rappelle sur tous les voisins non visités du sommet où l'on se situe
- On peut réaliser ce parcours à l'aide d'une pile (LIFO)

Parcours en profondeur

- On initialise

Parcours en profondeur

- On initialise
  - la liste des sommets déjà traités (Visit dans l'algorithme précédent) à la liste vide.

Parcours en profondeur

- On initialise
  - la liste des sommets déjà traités (Visit dans l'algorithme précédent) à la liste vide.
  - la **pile** des sommets à traiter (F dans l'algorithme précédent) en mettant dedans le sommet d'origine.

Parcours en profondeur

- On initialise
  - la liste des sommets déjà traités (Visit dans l'algorithme précédent) à la liste vide.
  - la **pile** des sommets à traiter (F dans l'algorithme précédent) en mettant dedans le sommet d'origine.
- Tant que la pile des sommets à traiter n'est pas vide, on traite le sommet de la pile (s'il n'a pas été déjà visité). Pour cela :

Parcours en profondeur

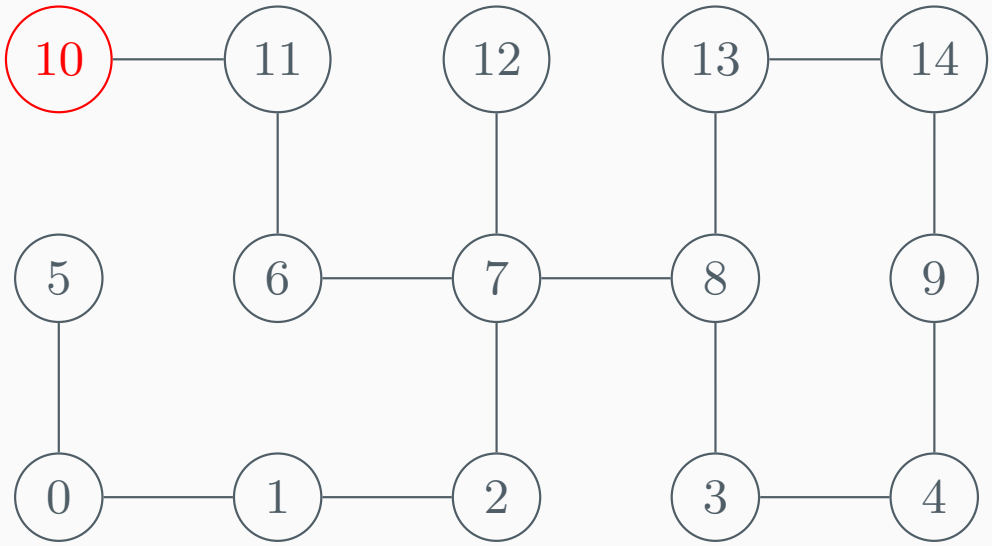
- On initialise
  - la liste des sommets déjà traités (Visit dans l'algorithme précédent) à la liste vide.
  - la **pile** des sommets à traiter (F dans l'algorithme précédent) en mettant dedans le sommet d'origine.
- Tant que la pile des sommets à traiter n'est pas vide, on traite le sommet de la pile (s'il n'a pas été déjà visité). Pour cela :
  - on l'enlève de la pile
  - on l'insère dans la liste des sommets traités
  - on ajoute tous ses voisins à la pile des sommets à traiter

Parcours en profondeur

Dans l'exemple ci-dessous :

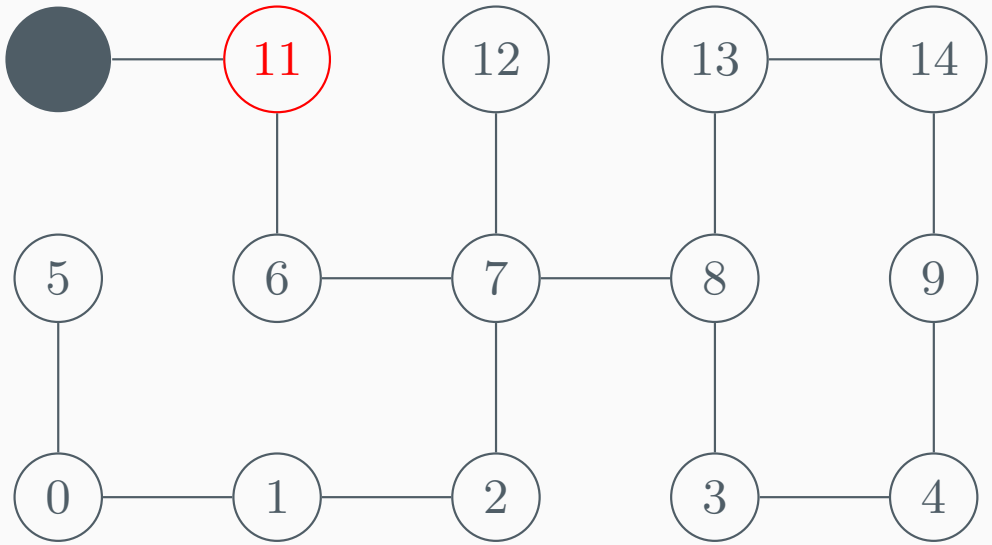
- Le sommet de la pile est à droite
- On insère les voisins dans la pile par ordre croissant

Parcours en profondeur



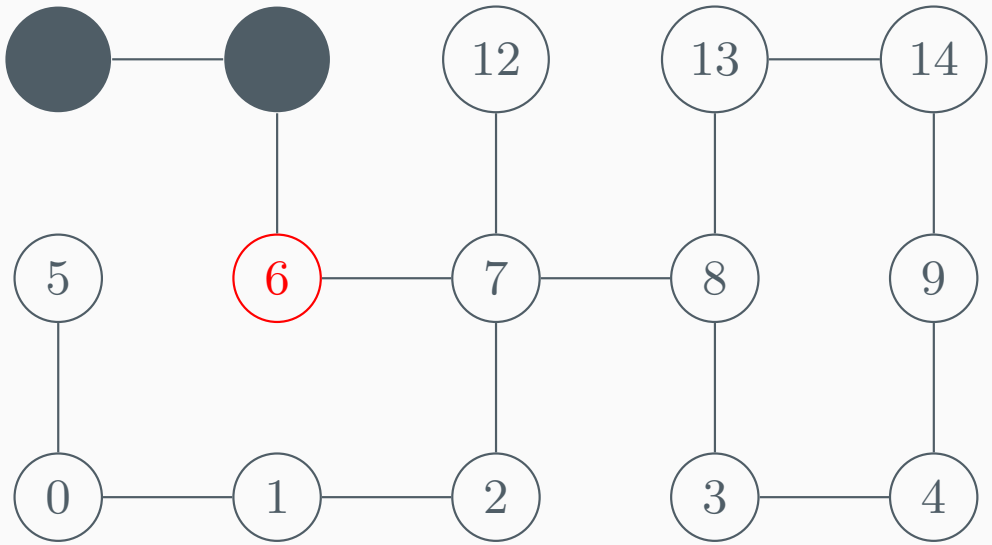
sommets traités	sommets à traiter
	10

Parcours en profondeur



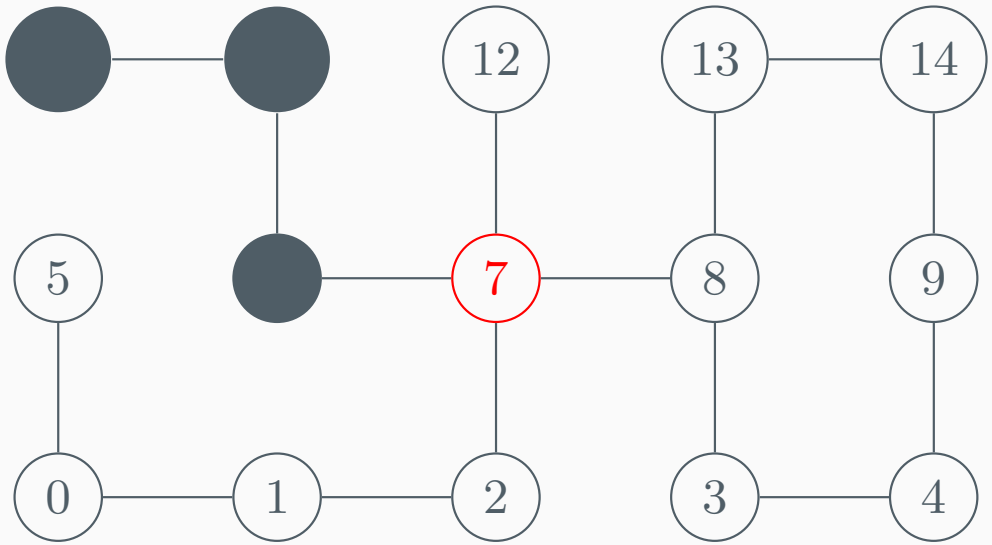
sommets traités	sommets à traiter
10	11

Parcours en profondeur



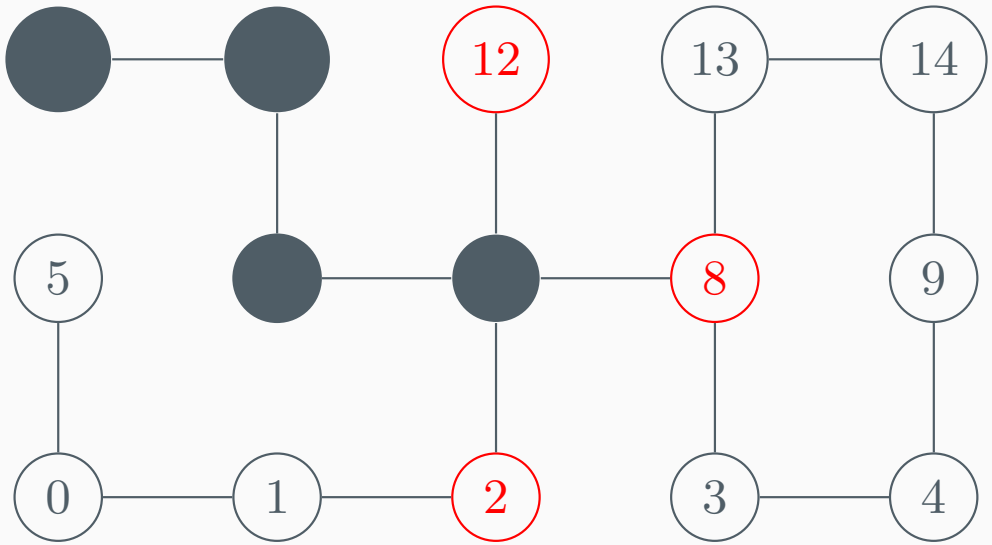
sommets traités	sommets à traiter
$10, 11$	6

Parcours en profondeur



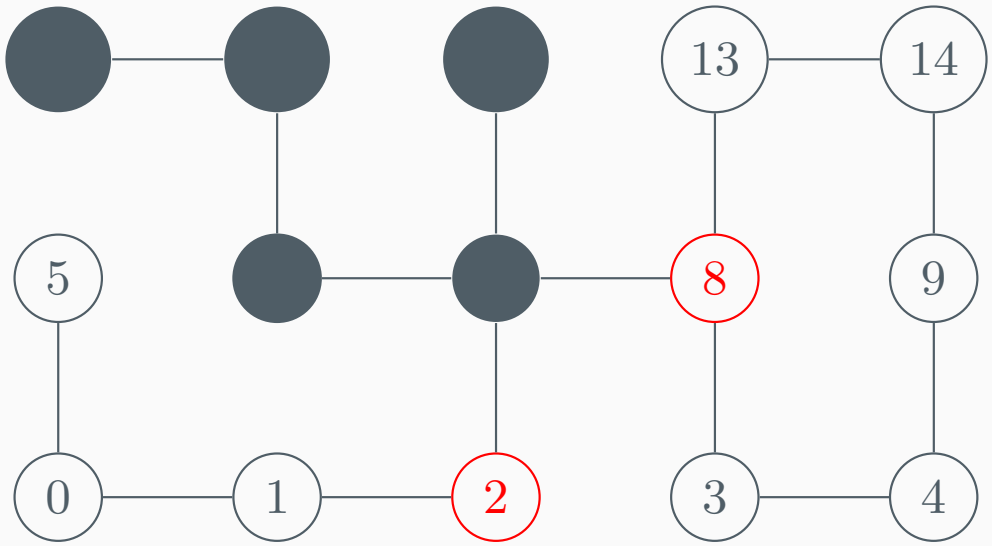
sommets traités	sommets à traiter
$10, 11, 6$	7

Parcours en profondeur



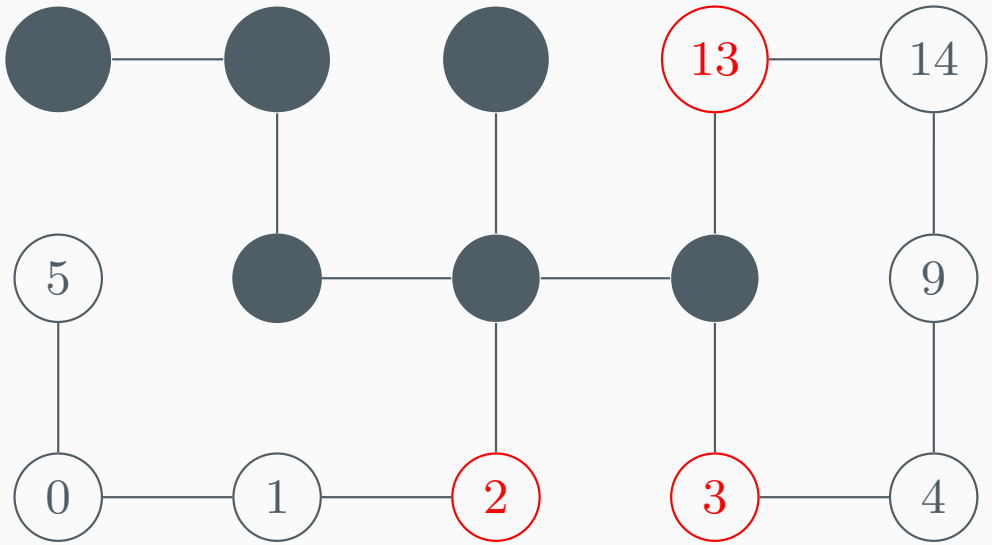
sommets traités	sommets à traiter
$10, 11, 6, 7$	$12, 8, 2$

Parcours en profondeur



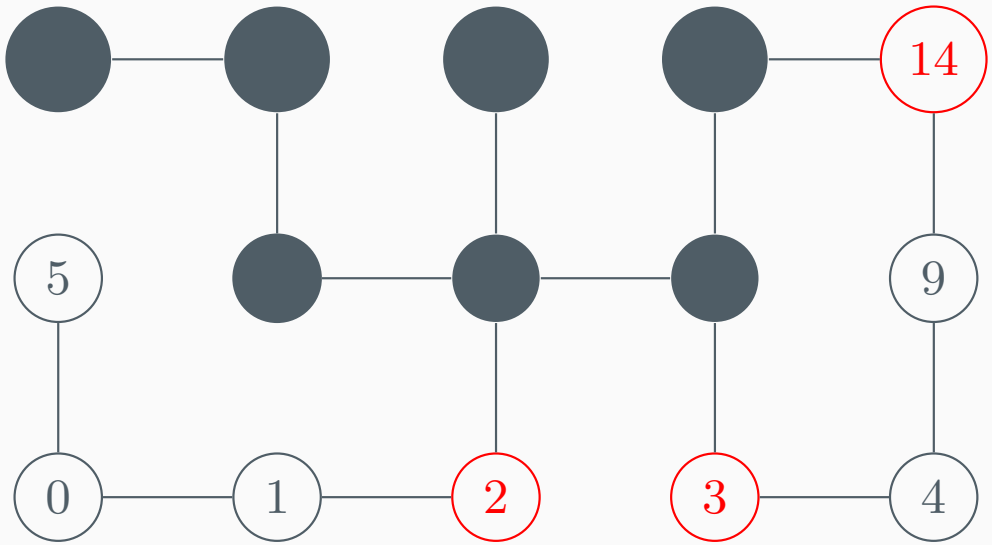
sommets traités	sommets à traiter
$10, 11, 6, 7, 12$	$8, 2$

Parcours en profondeur



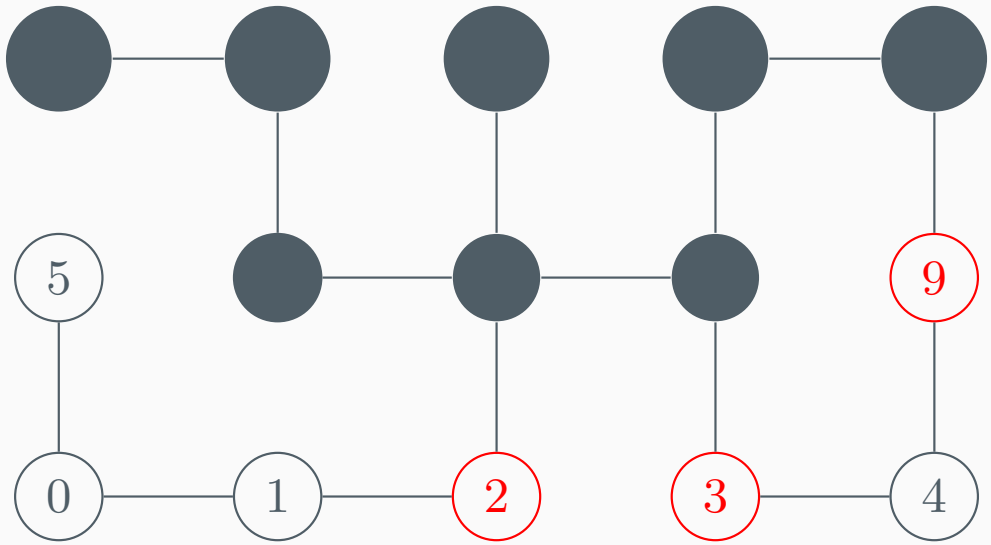
sommets traités	sommets à traiter
$10, 11, 6, 7, 12, 8$	$13, 3, 2$

Parcours en profondeur



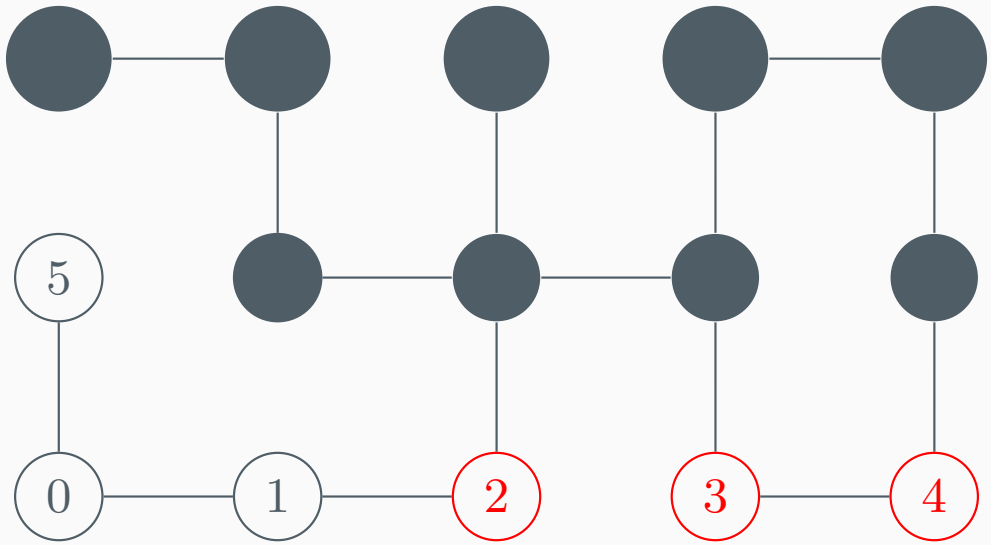
sommets traités	sommets à traiter
$10, 11, 6, 7, 12, 8, 13$	$14, 3, 2$

Parcours en profondeur



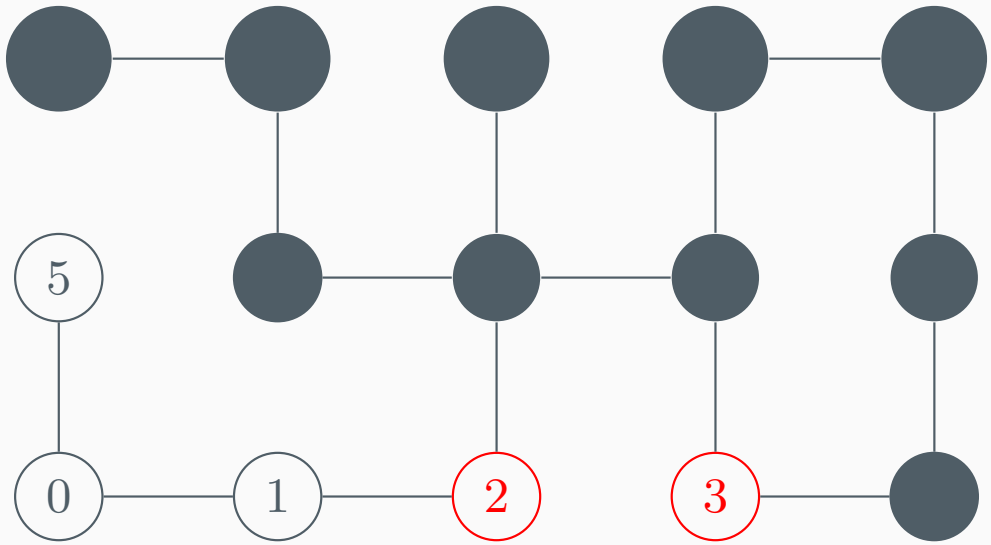
sommets traités	sommets à traiter
$10, 11, 6, 7, 12, 8, 13, 14$	$9, 3, 2$

Parcours en profondeur



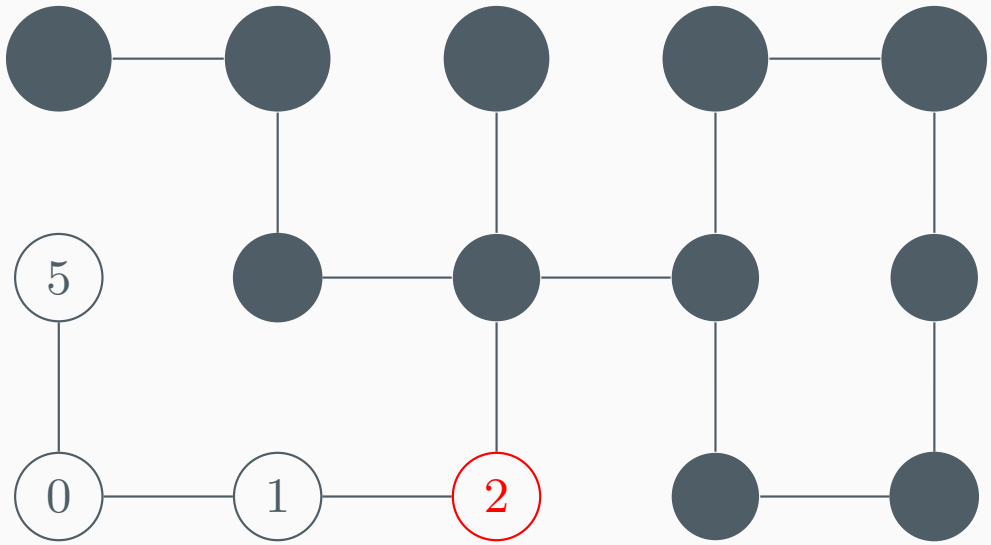
sommets traités	sommets à traiter
$10, 11, 6, 7, 12, 8, 13, 14, 9$	$4, 3, 2$

Parcours en profondeur



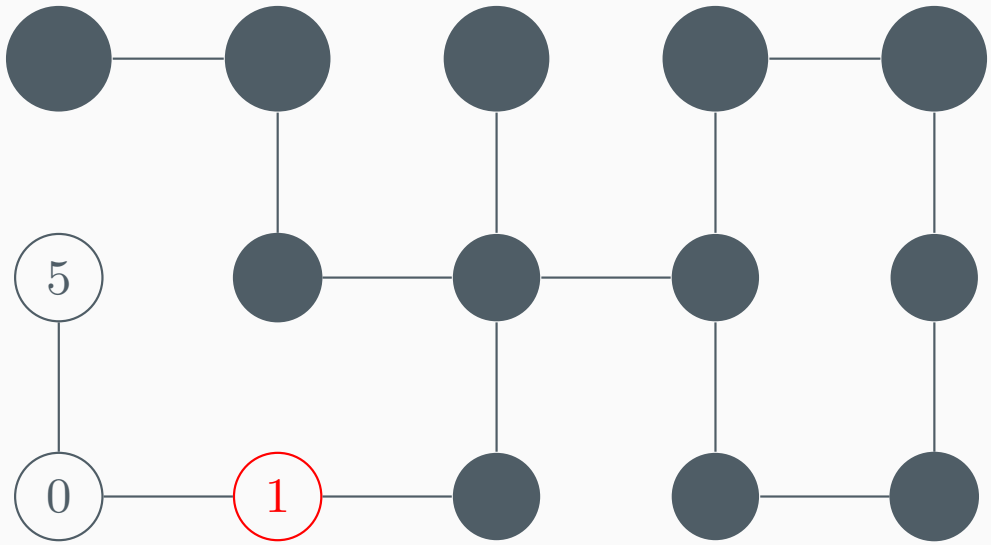
sommets traités	sommets à traiter
$10, 11, 6, 7, 12, 8, 13, 14, 9, 4$	$3, 3, 2$

Parcours en profondeur



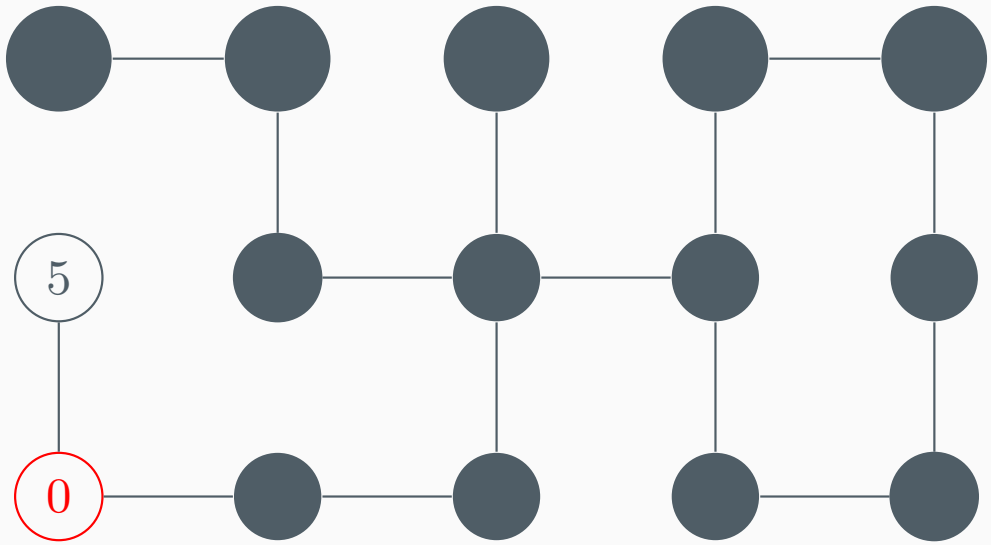
sommets traités	sommets à traiter
$10, 11, 6, 7, 12, 8, 13, 14, 9, 4, 3$	2

Parcours en profondeur



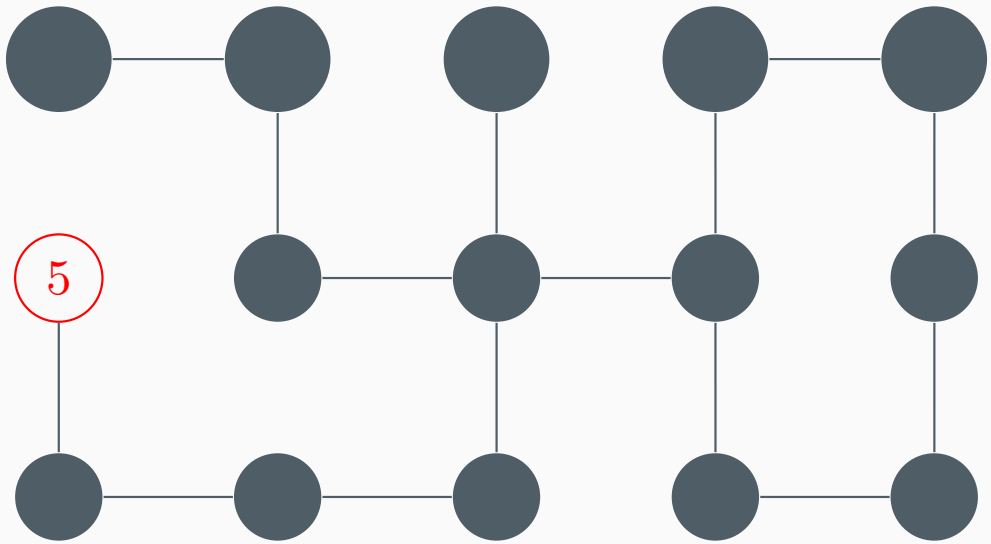
sommets traités	sommets à traiter
$10, 11, 6, 7, 12, 8, 13, 14, 9, 4, 3, 2$	1

Parcours en profondeur



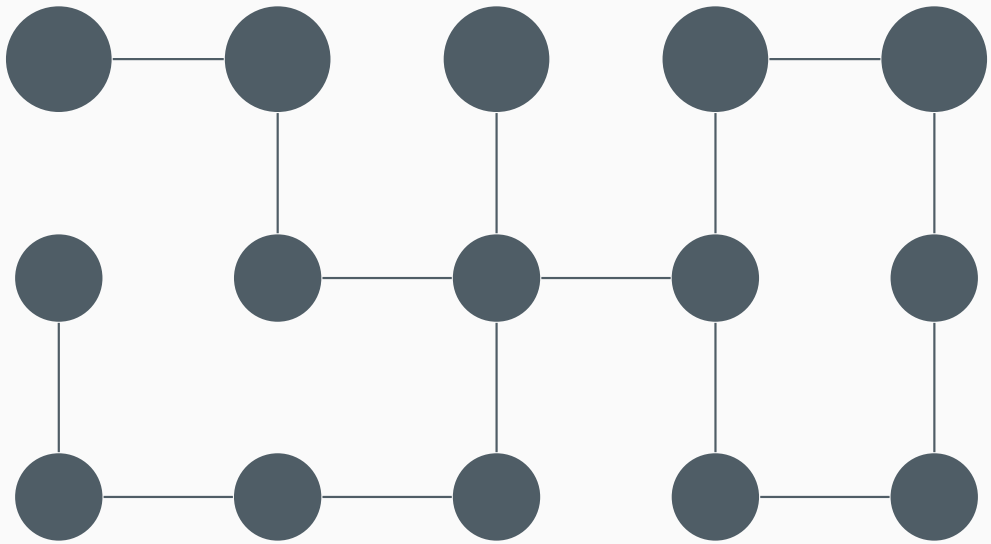
sommets traités	sommets à traiter
$10, 11, 6, 7, 12, 8, 13, 14, 9, 4, 3, 2, 1$	0

Parcours en profondeur



sommets traités	sommets à traiter
$10, 11, 6, 7, 12, 8, 13, 14, 9, 4, 3, 2, 1, 0$	5

Parcours en profondeur



sommets traités	sommets à traiter
$10, 11, 6, 7, 12, 8, 13, 14, 9, 4, 3, 2, 1, 0, 5$	$\emptyset$

Parcours en profondeur

- Dans un parcours en profondeur on traite en priorité les voisins du dernier sommet visité. Cela correspond au parcours en profondeur d'un arbre

Parcours en profondeur

- Dans un parcours en profondeur on traite en priorité les voisins du dernier sommet visité. Cela correspond au parcours en profondeur d'un arbre
- Dans l'exemple ci-dessus on cherche à parcourir tout le graphe, comme dit précédemment, on peut s'arrêter dès qu' un noeud est atteint.

Parcours en profondeur

- Dans un parcours en profondeur on traite en priorité les voisins du dernier sommet visité. Cela correspond au parcours en profondeur d'un arbre
- Dans l'exemple ci-dessus on cherche à parcourir tout le graphe, comme dit précédemment, on peut s'arrêter dès qu' un noeud est atteint.
- C'est le parcours en profondeur qui modélise une navigation internet

Implémentation

On veut implémenter le parcours en profondeur.

- On prendra l'implémentation des graphes par les listes d'adjacentes.
- On va garder en mémoire les sommets visités dans un tableau `visit` detype `bool array`.

Implémentons l'algorithme récursif. Il n'y a pas lieu de s'occuper précisément la frontière.

On va réaliser un parcours en profondeur, sans rien faire de particulier et en s'arrêtant uniquement quand tous les sommets auront été visités.

Implémentation

```
let dfs graphe i =
  let n = Array.length graphe in
    let visit = Array.make n false in
      let rec parcours l = match l with
          | [] -> ()
          | t :: q ->  if not visit.(t)
              then (visit.(t) <- true;
                  parcours graphe.(t)
                );
              parcours q
      in visit.(i) <- true; parcours graphe.(i);;
```

Complexité du parcours en profondeur

On suppose que les opérations pour enpiler et dépiler se font en temps constant.

Complexité du parcours en profondeur

On suppose que les opérations pour enpiler et dépiler se font en temps constant.

- On voit que lors du parcours en profondeur tous les sommets sont visités une fois et chaque arête est considérée une fois (ou deux fois dans le cas d'un graphe non orienté).

  On a donc une complexité temporelle :

Complexité du parcours en profondeur

On suppose que les opérations pour enpiler et dépiler se font en temps constant.

- On voit que lors du parcours en profondeur tous les sommets sont visités une fois et chaque arête est considérée une fois (ou deux fois dans le cas d'un graphe non orienté).

  On a donc une complexité temporelle :

  $$O(|S| + |A|)$$

- La complexité en espace est :

Complexité du parcours en profondeur

On suppose que les opérations pour enpiler et dépiler se font en temps constant.

- On voit que lors du parcours en profondeur tous les sommets sont visités une fois et chaque arête est considérée une fois (ou deux fois dans le cas d'un graphe non orienté).

  On a donc une complexité temporelle :

  $$O(|S| + |A|)$$

- La complexité en espace est :

  $$O(|S| + |A|)$$

  car le tableau des sommets visités est de taille $|S|$ et la frontière est au maximum de taille $|A|$ (un sommet peut être ajouté autant de fois que son degré entrant).

Parcours en largeur (BFS)

Parcours en largeur

On veut maintenant étudier un parcours où l'on explore **tous les voisins** d'un sommet avant de passer aux voisins de ses voisins.

Le principe est alors d'utiliser une file (de type FIFO) pour stocker la frontière. On l'appelle aussi BFS pour *Breadth First Search*

Parcours en largeur

- On initialise

Parcours en largeur

- On initialise
  - la liste des sommets déjà traités (Visit dans l'algorithme précédent) à la liste vide.

Parcours en largeur

- On initialise
  - la liste des sommets déjà traités (Visit dans l'algorithme précédent) à la liste vide.
  - la **file** des sommets à traiter (F dans l'algorithme précédent) en mettant dedans le sommet d'origine.

Parcours en largeur

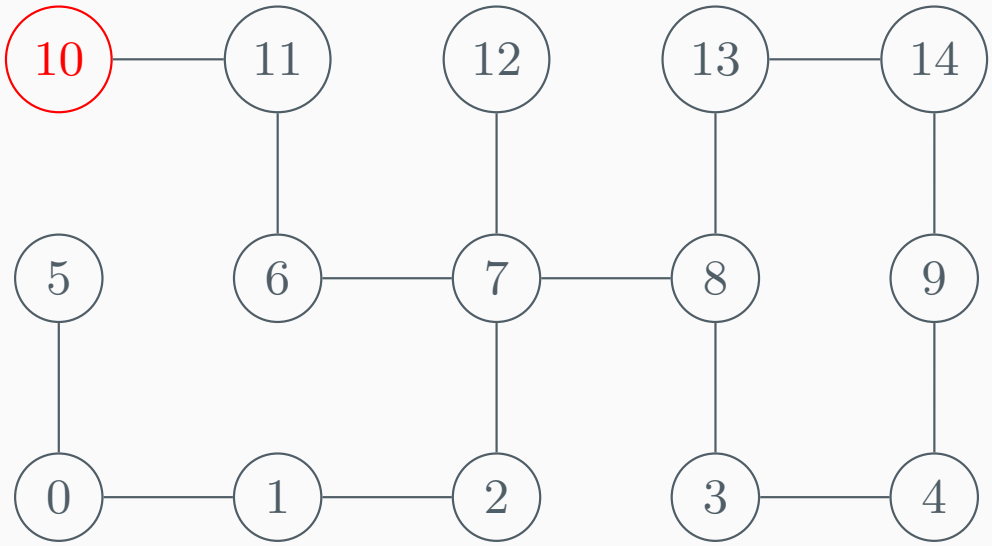
- On initialise
  - la liste des sommets déjà traités (Visit dans l'algorithme précédent) à la liste vide.
  - la **file** des sommets à traiter (F dans l'algorithme précédent) en mettant dedans le sommet d'origine.
- Tant que la file des sommets à traiter n'est pas vide, on traite la tête de la file (s'il n'a pas été déjà visité). Pour cela :

Parcours en largeur

- On initialise
  - la liste des sommets déjà traités (Visit dans l'algorithme précédent) à la liste vide.
  - la **file** des sommets à traiter (F dans l'algorithme précédent) en mettant dedans le sommet d'origine.
- Tant que la file des sommets à traiter n'est pas vide, on traite la tête de la file (s'il n'a pas été déjà visité). Pour cela :
  - on l'enlève de la file
  - on l'insère dans la liste des sommets traités
  - on ajoute tous ses voisins à la queue de la file des sommets à traiter

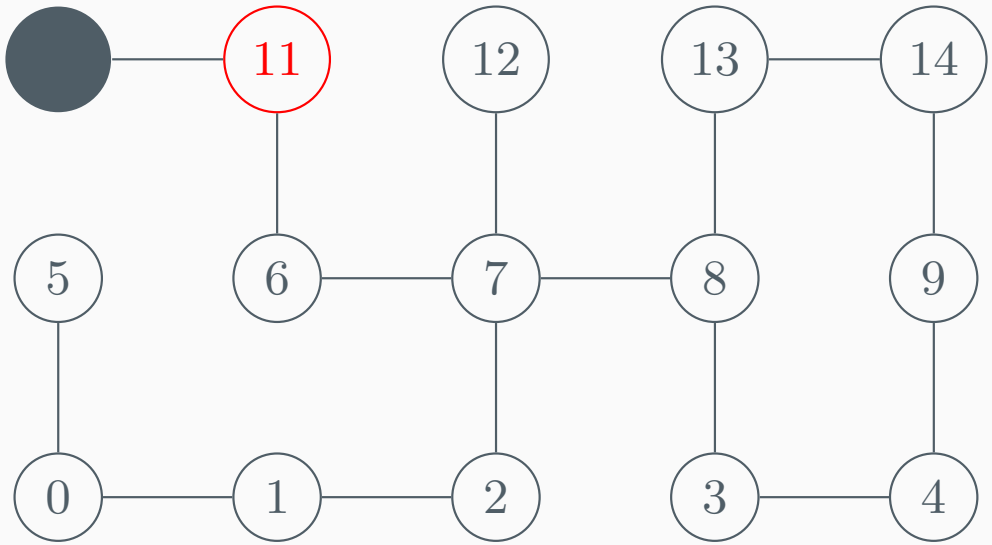
Voici, dans le cas de notre exemple comment évolue la liste F et la file Visit. Dans la colonne « sommets à traiter », on indique la file des sommets à traiter où on insère dans la file à droite et on extrait à gauche. Par commodité on insérera à chaque étape les voisins par ordre croissant.

Parcours en largeur



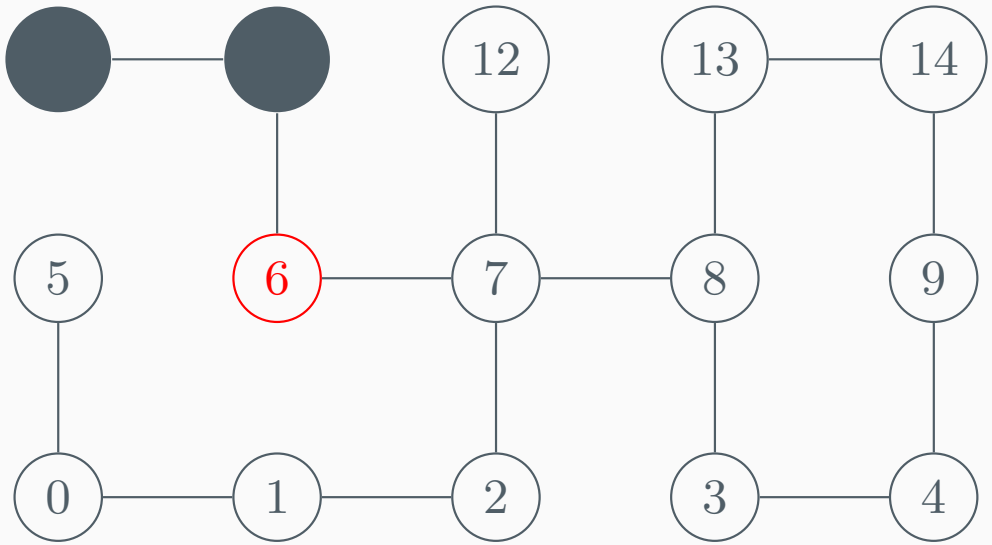
sommets traités	sommets à traiter
	10

Parcours en largeur



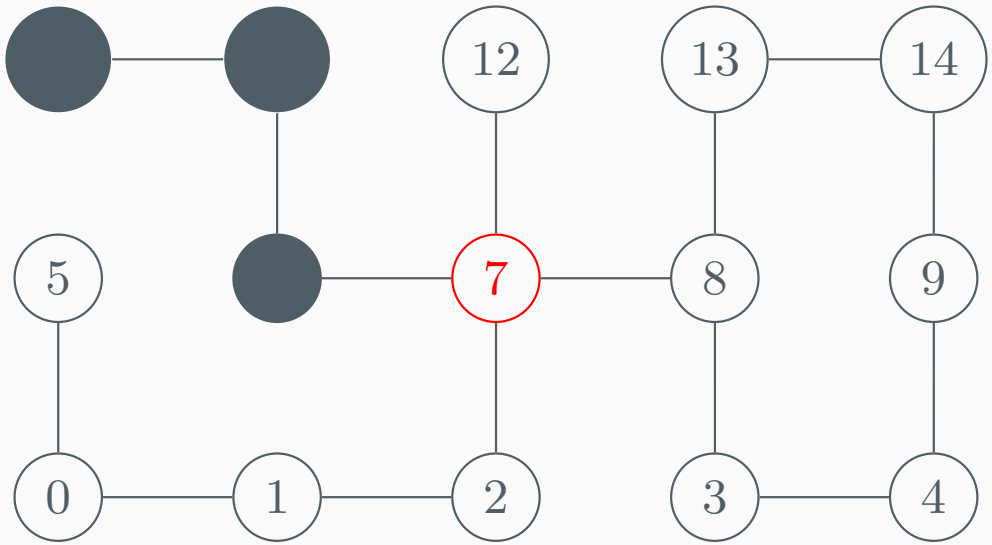
sommets traités	sommets à traiter
10	11

Parcours en largeur



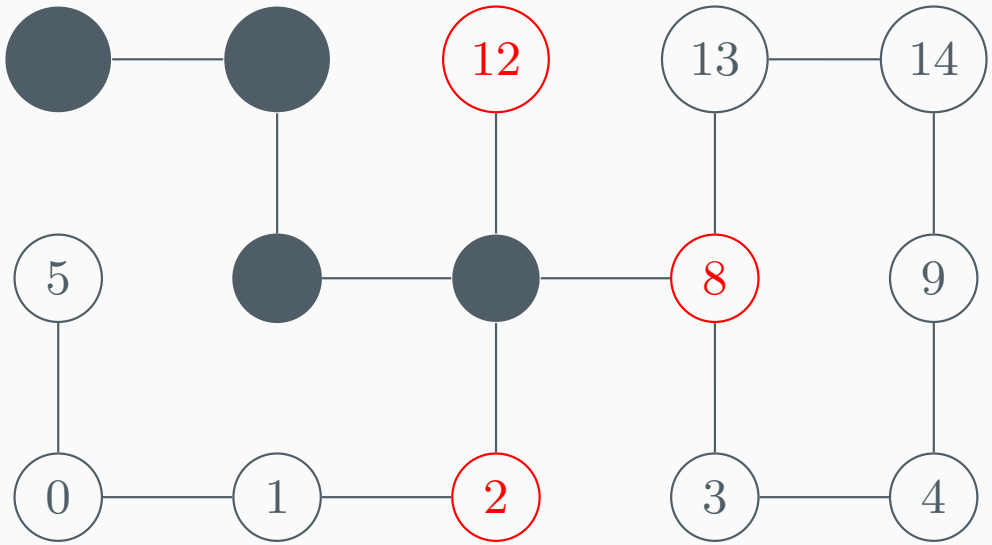
sommets traités	sommets à traiter
$10, 11$	6

Parcours en largeur



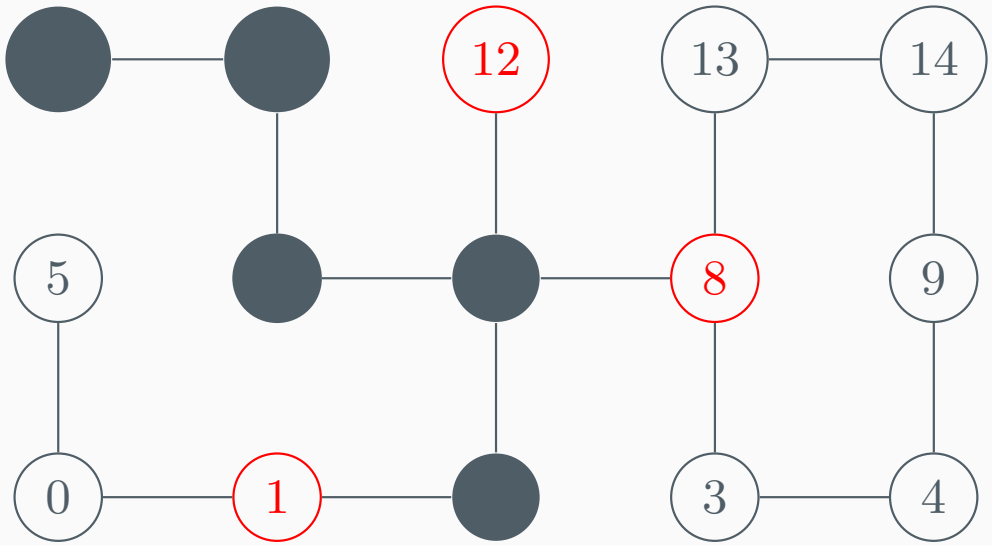
sommets traités	sommets à traiter
$10, 11, 6$	7

Parcours en largeur



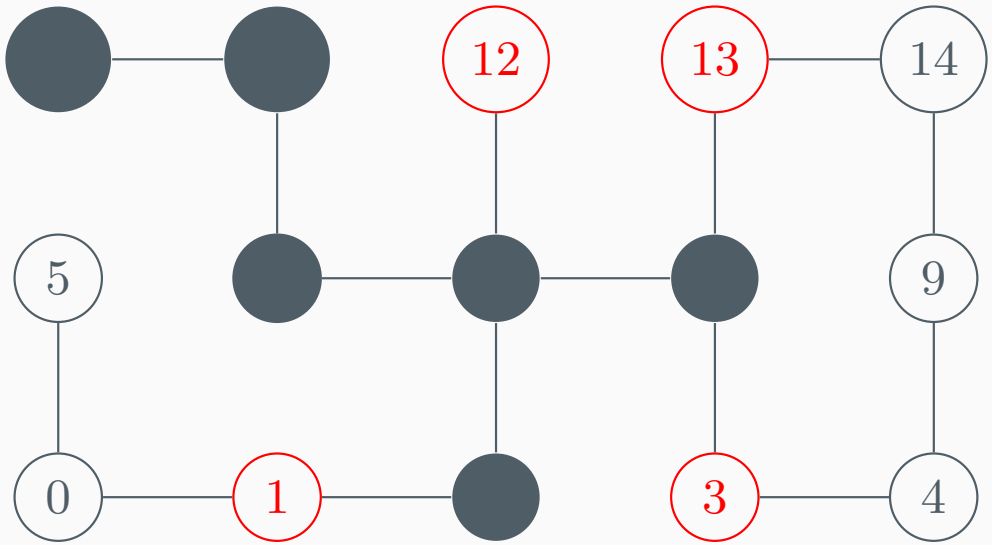
sommets traités	sommets à traiter
$10, 11, 6, 7$	$2, 8, 12$

Parcours en largeur



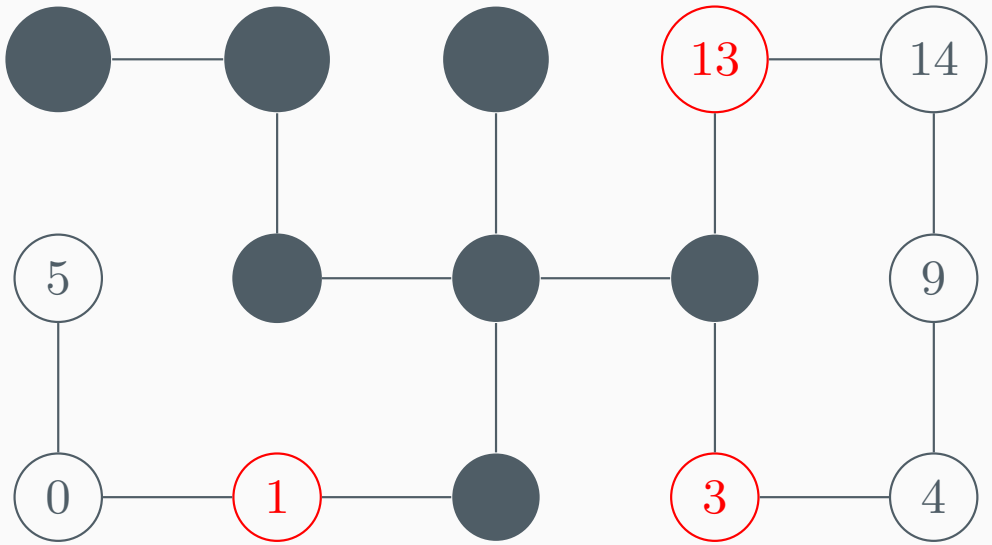
sommets traités	sommets à traiter
$10, 11, 6, 7, 2$	$8, 12, 1$

Parcours en largeur



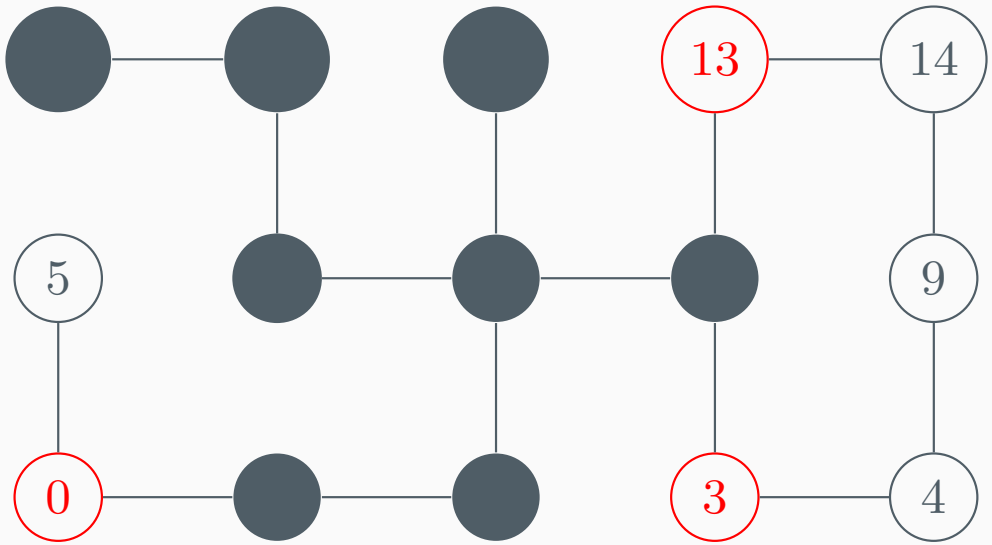
sommets traités	sommets à traiter
$10, 11, 6, 7, 2, 8$	$12, 1, 3, 13$

Parcours en largeur



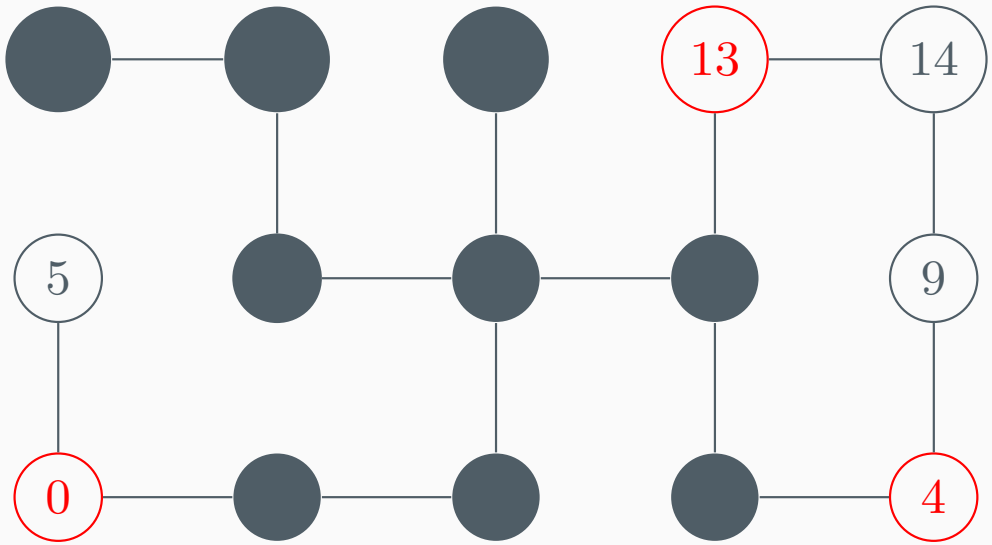
sommets traités	sommets à traiter
$10, 11, 6, 7, 2, 8, 12$	$1, 3, 13$

Parcours en largeur



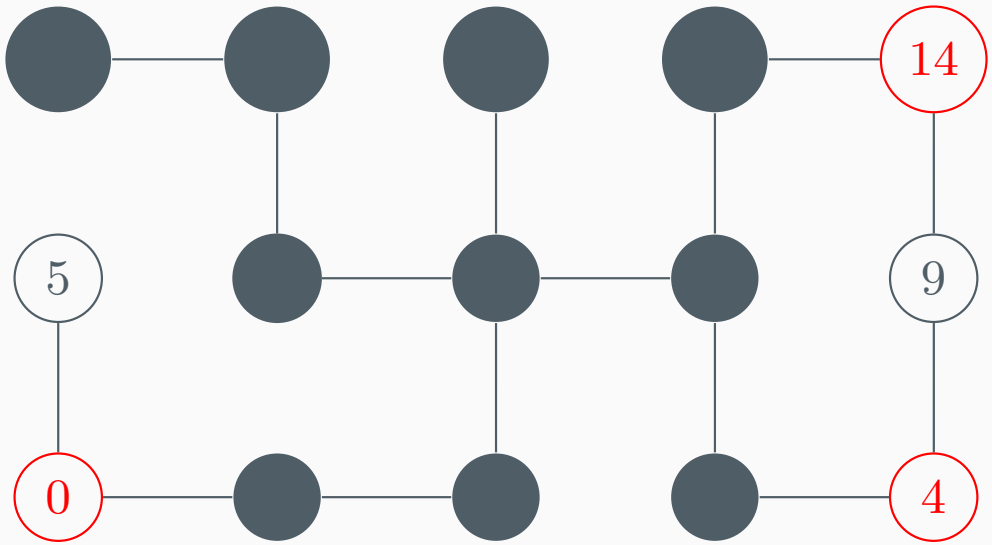
sommets traités	sommets à traiter
$10, 11, 6, 7, 2, 8, 12, 1$	$3, 13, 0$

Parcours en largeur



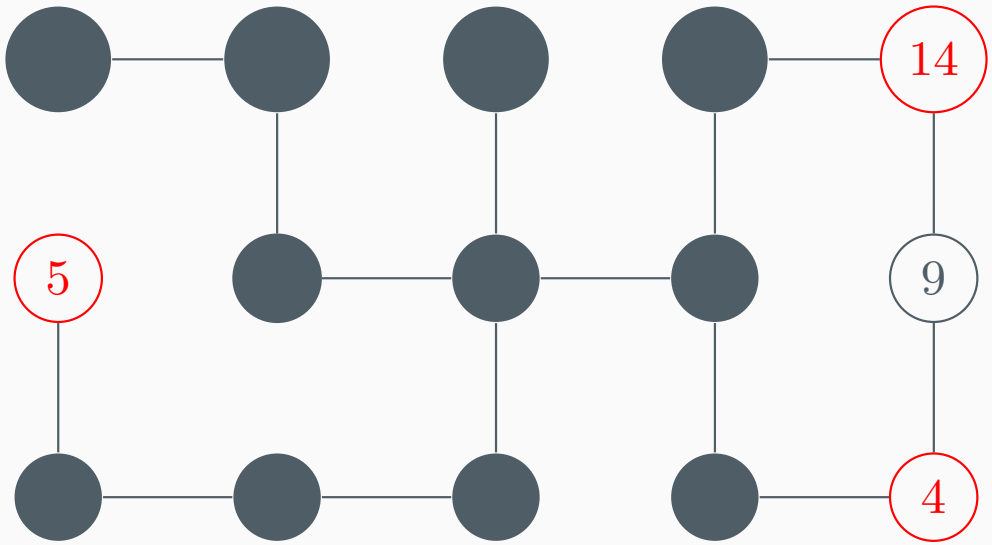
sommets traités	sommets à traiter
$10, 11, 6, 7, 2, 8, 12, 1, 3$	$13, 0, 4$

Parcours en largeur



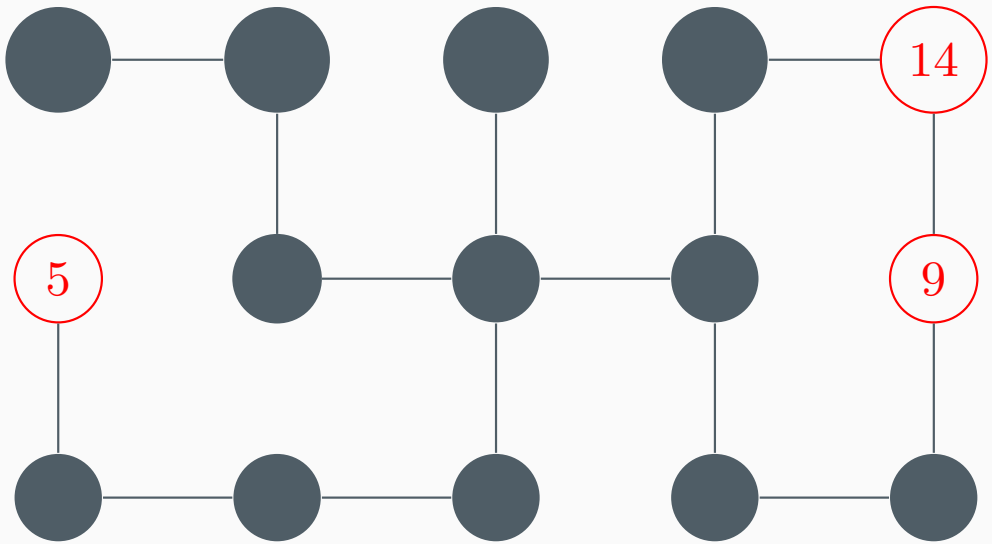
sommets traités	sommets à traiter
$10, 11, 6, 7, 2, 8, 12, 1, 3, 13$	$0, 4, 14$

Parcours en largeur



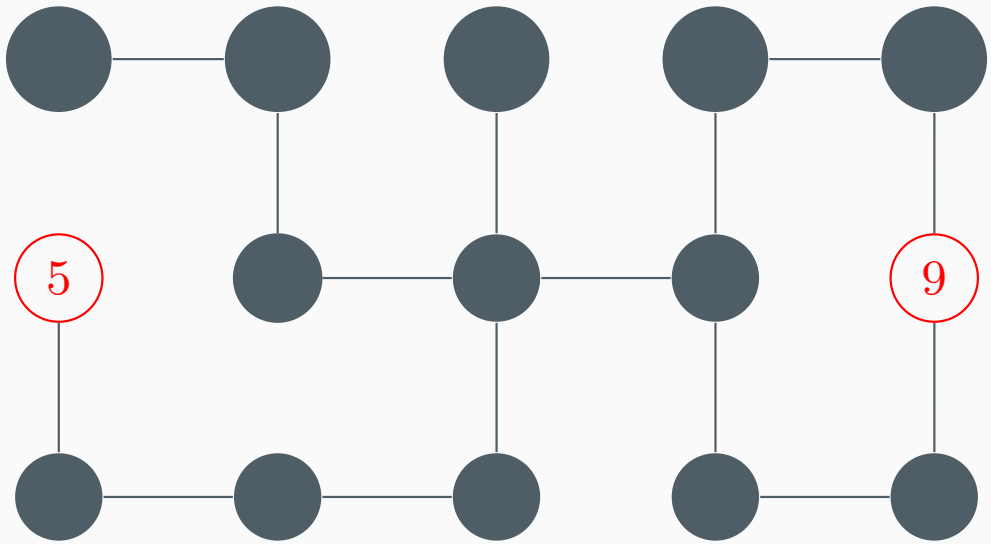
sommets traités	sommets à traiter
$10, 11, 6, 7, 2, 8, 12, 1, 3, 13, 0$	$4, 14, 5$

Parcours en largeur



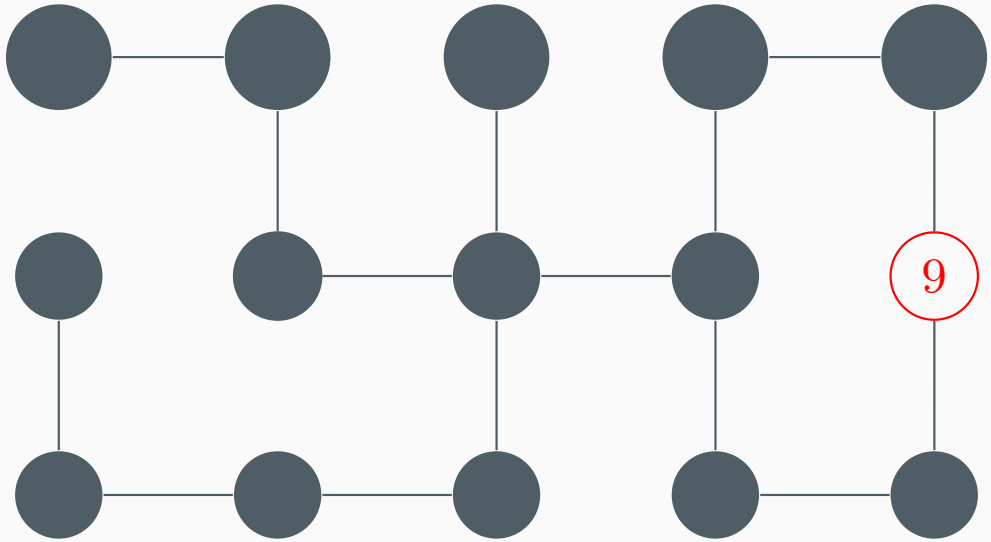
sommets traités	sommets à traiter
$10, 11, 6, 7, 2, 8, 12, 1, 3, 13, 0, 4$	$14, 5, 9$

Parcours en largeur



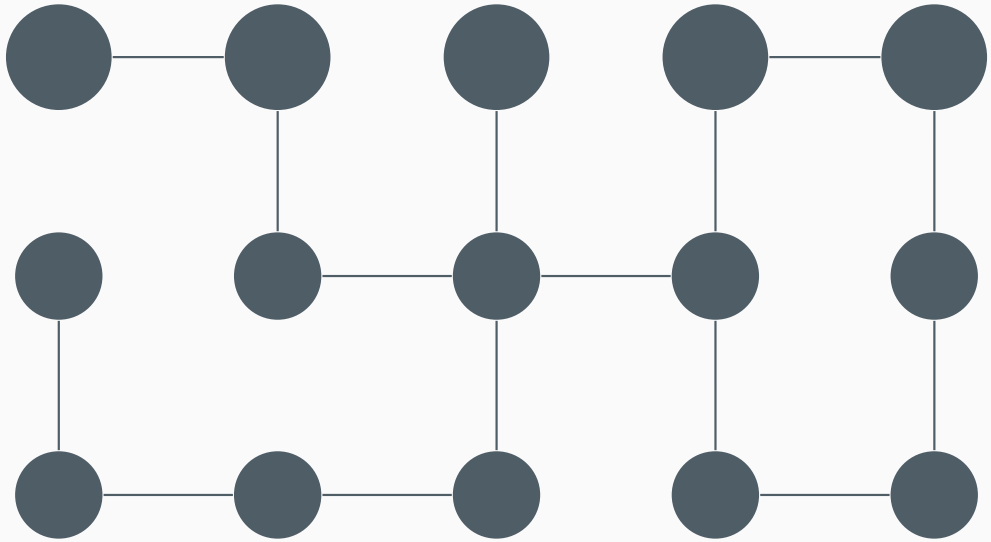
sommets traités	sommets à traiter
$10, 11, 6, 7, 2, 8, 12, 1, 3, 13, 0, 4, 14$	$5, 9$

Parcours en largeur



sommets traités	sommets à traiter
$10, 11, 6, 7, 2, 8, 12, 1, 3, 13, 0, 4, 14$	9

Parcours en largeur



sommets traités	sommets à traiter
$10, 11, 6, 7, 2, 8, 12, 1, 3, 13, 0, 4, 14, 9$	$\emptyset$

Parcours en largeur

- Dans un parcours en largeur on traite **tous les voisins** d'un sommet avant de passer aux voisins d'un autre sommet. Cela correspond au parcours en largeur d'un arbre

Parcours en largeur

- Dans un parcours en largeur on traite **tous les voisins** d'un sommet avant de passer aux voisins d'un autre sommet. Cela correspond au parcours en largeur d'un arbre
- Dans l'exemple ci-dessus on cherche à parcourir tout le graphe, comme dit précédemment, on peut s'arrêter dès que un noeud est atteint.

Parcours en largeur

- Dans un parcours en largeur on traite **tous les voisins** d'un sommet avant de passer aux voisins d'un autre sommet. Cela correspond au parcours en largeur d'un arbre
- Dans l'exemple ci-dessus on cherche à parcourir tout le graphe, comme dit précédemment, on peut s'arrêter dès que un noeud est atteint.
- C'est ce genre de parcours que l'on utilise si on veut regarder en priorité les nœuds « proches » du sommet de départ.

Implémentation

On veut implémenter le parcours en largueur.

- On prendra encore l'implémentation des graphes par les listes d'adjacentes.
- On garde en mémoire les sommets visités dans un tableau `visit` de type `bool array`.
- On implémentera les files par des **références de listes**.

Les files

- `let creeFile () =`
- `let enfile f a =` qui permet d'enfiler un entier
- `let estVide f =`
- La fonction defile est

```
let defile f =
```

Les files

- `let creeFile () = ref []`
- `let enfile f a = f := !f @ [a]` qui permet d'enfiler un entier
- `let estVide f = (!f = [])`
- La fonction defile est

```
let defile f = match !f with
|[] -> failwith "la file est vide"
|t :: q -> f:= q; t
```

On va avoir aussi besoin d'une fonction permettant de rajouter dans la file tous les éléments d'une liste.

```
let rec enfileliste f l = match l with
  |[] -> ()
  |t :: q ->  enfile f t;  enfileliste  f q;;
```

Parcours en largeur

Le parcours en largeur est alors

Parcours en largeur

Le parcours en largeur est alors

```
let parcours_larg g i =
  let visit =  Array.make (Array.length g) false in
  let f = creefile() in enfile f i;
  while not(estVide f) do
     let x = defile f in
          if not(visit.(x)) then
             (visit.(x) <- true;
              enfileliste f g.(u));
  done;;
```

Complexité du parcours en largeur

On suppose que les opérations pour enfiler et défiler se font en temps constant. On voit que lors du parcours en profondeur tous les sommets sont visités une fois et chaque arête est aussi visitée une fois.

- On a donc une complexité temporelle

Complexité du parcours en largeur

On suppose que les opérations pour enfiler et défiler se font en temps constant. On voit que lors du parcours en profondeur tous les sommets sont visités une fois et chaque arête est aussi visitée une fois.

- On a donc une complexité temporelle

$$O(|S| + |A|)$$

- La complexité en espace est de

Complexité du parcours en largeur

On suppose que les opérations pour enfiler et défiler se font en temps constant. On voit que lors du parcours en profondeur tous les sommets sont visités une fois et chaque arête est aussi visitée une fois.

- On a donc une complexité temporelle
$$O(|S| + |A|)$$
- La complexité en espace est de
$$O(|S| + |A|)$$
Comme pour le parcours en profondeur

Exemples

Nous allons traiter quelques exemples qui se resolvent avec des parcours de graphe.

Composantes connexes

On se donne un graphe G non orienté (il n'y a donc pas de différences entre connexe et fortement connexe) et on veut savoir s'il est connexe et si ce n'est pas le cas faire la liste de ses composantes connexes.

Avant cela on va s'intéresser à la recherche de l'existence d'un chemin entre deux sommets

Composantes connexes

On se donne deux sommets i et j et on veut savoir s'il existe un chemin qui va de i à j. L'idée est de parcourir le graphe en partant de i jusqu'à atteindre (ou pas) j. On réalise un parcours en profondeur (recursif).

Composantes connexes

```
let trouve graphe i j =
  let n = Array.length graphe in
    let visit = Array.make n false in
        let rec parcours l = if not(visit.(j)) then
        match l with
            | [] -> ()
            | t :: q -> if t = j
                then visit.(j) <- true
                else (if not visit.(t)
                    then (visit.(t) <- true;
                        parcours graphe.(t));
                    parcours q);
        in visit.(i) <- true; parcours graphe.(i);
        visit.(j);;
```

Composantes connexes

Ecrivons maintenant une fonction qui renvoie la composante connexe d'un sommet i. Il suffit juste d'ajouter un accumulateur qui garde la liste de sommets visités. Faire attention à n'ajouter les sommets que la première fois.

Composantes connexes

```
let composanteConnexe graphe i  =
  let n = Array.length graphe in
    let visit = Array.make n false in
      let list = ref [i] in
        let rec parcours l = match l with
            | [] -> ()
            | t :: q ->  if not visit.(t)
                    then (visit.(t) <- true;
                    list := t::!list;
                        parcours graphe.(t));
                    parcours q;
        in visit.(i) <- true; parcours graphe.(i);
        !list;;
```

Composantes connexes

Pour finir on peut obtenir la liste de toutes les composantes connexes en relançant la fonction précédente sur un sommet qui n'est pas encore dans les composantes connexes déjà établies.

Recherche du plus court chemin

On se donne un graphe et deux sommets et on cherche le plus court chemin reliant les deux sommets.

Cette fois, le parcours en profondeur est moins utile. On veut visiter notre but avec le moins d'étapes possibles.

On voit que le parcours en largeur parcourt le graphe en visitant d'abord les nœuds de distance 1, puis ceux de distance 2 et ainsi de suite. L'idée est donc de réaliser un parcourt en largeur.

Recherche du plus court chemin

Une fois que l'on aura atteint le sommet cible, on voudra reconstruire le chemin.

Pour cela, on va garder en mémoire, pour chaque nœud atteint, le noeud précédent dans le chemin le plus court (c'est-à-dire le nœud d'où l'on vient lors que l'on considère ce nœud **pour la première fois**).

De ce fait, la file contient en permanence les nœuds à visiter ainsi que les nœuds qui les précédent.

On utilise aussi un tableau `prec` tel que `prec.(i)` contient le numéro du sommet qui précède i. Il nous permet de ne pas avoir un tableau `visit` car un noeud est visité si et seulement `prec.(i)` a une valeur différente de la valeur initiale.

Recherche du plus court chemin

Ecrivons la fonction de parcours qui renvoie le tableau `prec`

Recherche du plus court chemin

Ecrivons la fonction de parcours qui renvoie le tableau prec

```
let pluscourtchemin g i =
  let n = Array.length g  in
  let prec = Array.make n (-1) in
  let f = creefile() in enfile f (i,i);
  while not(estVide f) do
     let (xp,x) = defile f in
     if (prec.(x) = -1) then
         (prec(x) <- xp;
          enfileliste2 f  x g.(x));
  done;;
  prec;;
```

où

```
let rec enfliste2 f s l = match l with
   | [] -> ()
   | x :: q -> enfile f (s,x) ; enfliste2 f s q
```

Recherche du plus court chemin

On peut maintenant reconstruire le chemin :

Recherche du plus court chemin

On peut maintenant reconstruire le chemin :

```
let chemin g i j =
let prec = pluscourtchemin g i in
    let rec reconstruit acc k =
    if k = i then (acc)
    else reconstruit (k::acc) prec.(k)
 in reconstruit [] j;;
```

4 Chemin de plus faible poids

- Les graphes pondérés
- Algorithme de Floyd Warshall
- Algorithme de Dijkstra