Graphes : Parcours en largeur et profondeur

Quentin Fortier

May 15, 2022

Parcours de graphe

On a souvent besoin de parcourir les sommets d'un graphe un par un. Voici les deux principaux parcours, visitant les sommets de proche en proche :

Parcours de graphe

On a souvent besoin de parcourir les sommets d'un graphe un par un. Voici les deux principaux parcours, visitant les sommets de proche en proche :

1. **Parcours en profondeur** (ou DFS pour Depth-First Search) : on visite les sommets le plus profondément possible avant de revenir en arrière.

Parcours de graphe

On a souvent besoin de parcourir les sommets d'un graphe un par un. Voici les deux principaux parcours, visitant les sommets de proche en proche :

1. **Parcours en profondeur** (ou DFS pour Depth-First Search) : on visite les sommets le plus profondément possible avant de revenir en arrière.
2. **Parcours en largeur** (ou BFS pour Breadth-First Search) : on visite les sommets par distance croissante depuis un sommet de départ.

Parcours de graphe

On a souvent besoin de parcourir les sommets d'un graphe un par un. Voici les deux principaux parcours, visitant les sommets de proche en proche :

1. **Parcours en profondeur** (ou DFS pour Depth-First Search) : on visite les sommets le plus profondément possible avant de revenir en arrière.
2. **Parcours en largeur** (ou BFS pour Breadth-First Search) : on visite les sommets par distance croissante depuis un sommet de départ.

Si le graphe est connexe, tous les sommets sont visités.
Sinon, on applique un parcours sur chacune des composantes connexes.

Parcours en profondeur (DFS)

Un **parcours en profondeur** sur un graphe $G = (V, E)$ depuis un sommet u consiste à visiter u puis visiter récursivement chaque voisin de u qui n'a pas déjà été vu.

Parcours en profondeur (DFS)

Un **parcours en profondeur** sur un graphe $G = (V, E)$ depuis un sommet u consiste à visiter u puis visiter récursivement chaque voisin de u qui n'a pas déjà été vu.

Il est nécessaire de se souvenir des sommets déjà visités pour éviter de faire une infinité d'appels récursifs (en revisitant toujours les mêmes sommets).
D'où l'utilisation d'une liste `visited` et d'une fonction auxiliaire récursive (de façon à avoir accès à `visited` dans tous les appels récursifs).

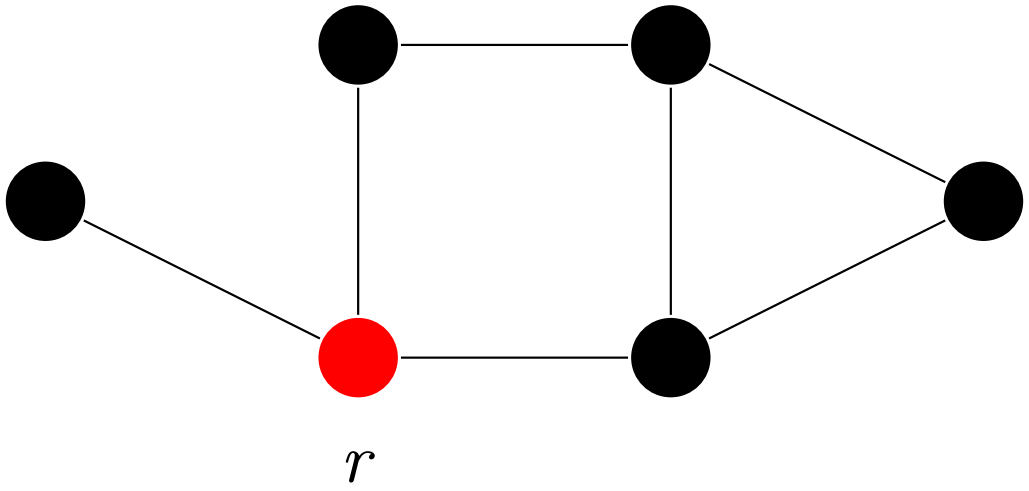
Parcours en profondeur (DFS)

Parcours en profondeur sur un graphe G représenté par liste d'adjacence, depuis un sommet s :

```
def dfs(G, s):
    visited = [False]*len(G)
    def aux(u):
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                aux(v)
    aux(s)
```

`visited[u]` vaut `True` si `u` a déjà été visité

Parcours en profondeur (DFS) : Exemple



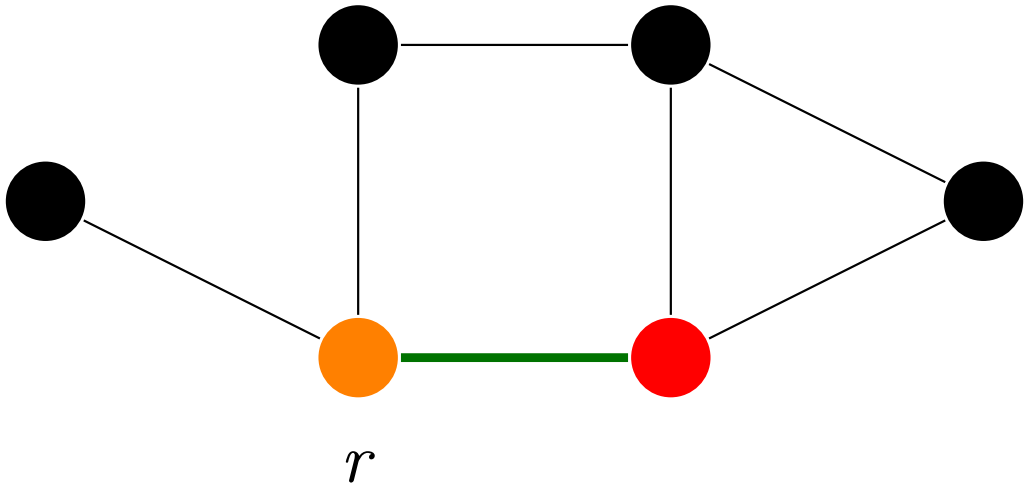
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



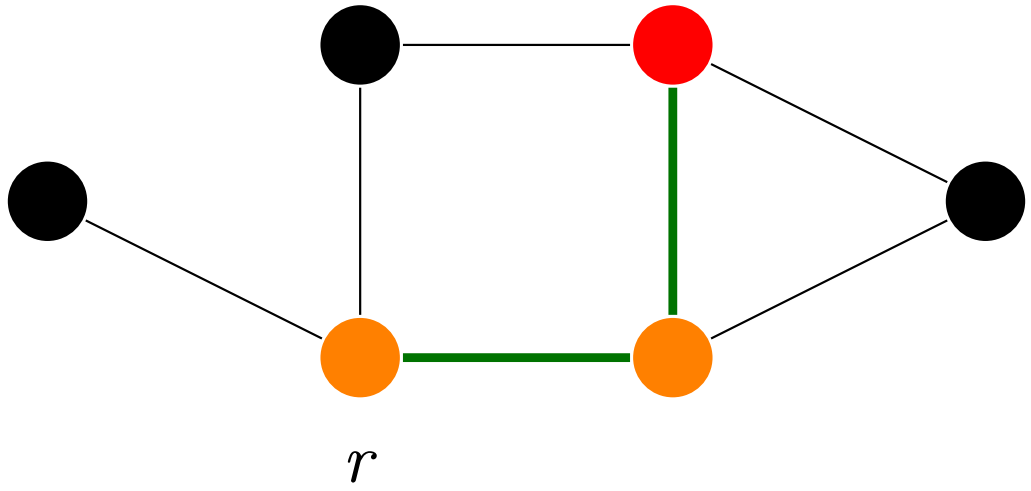
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



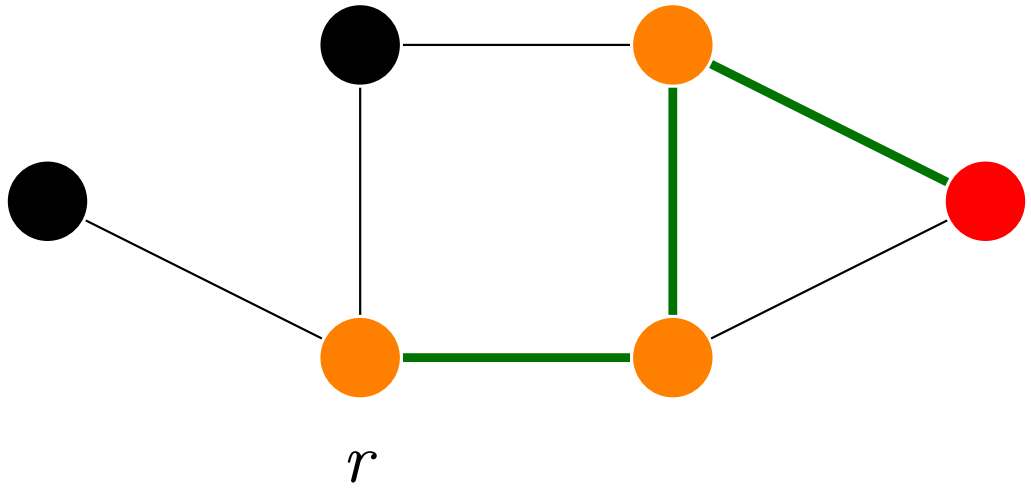
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



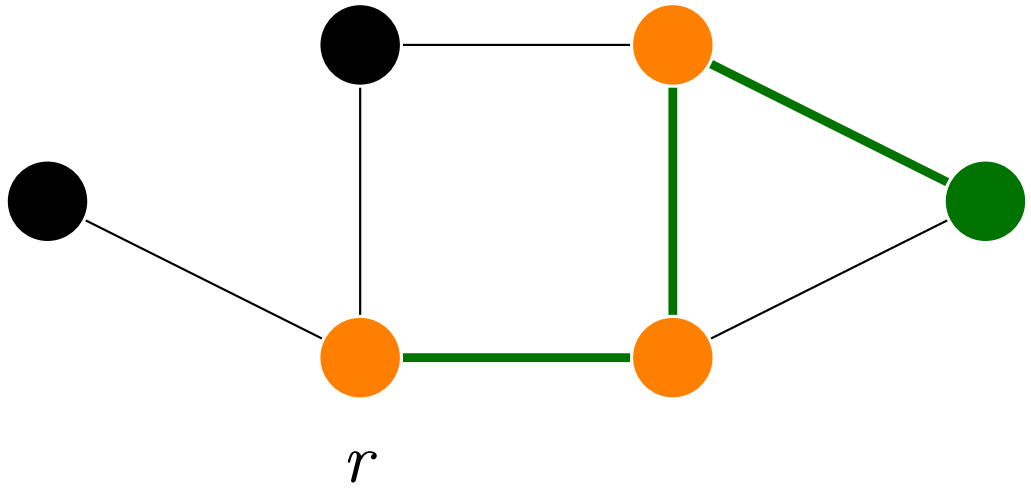
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



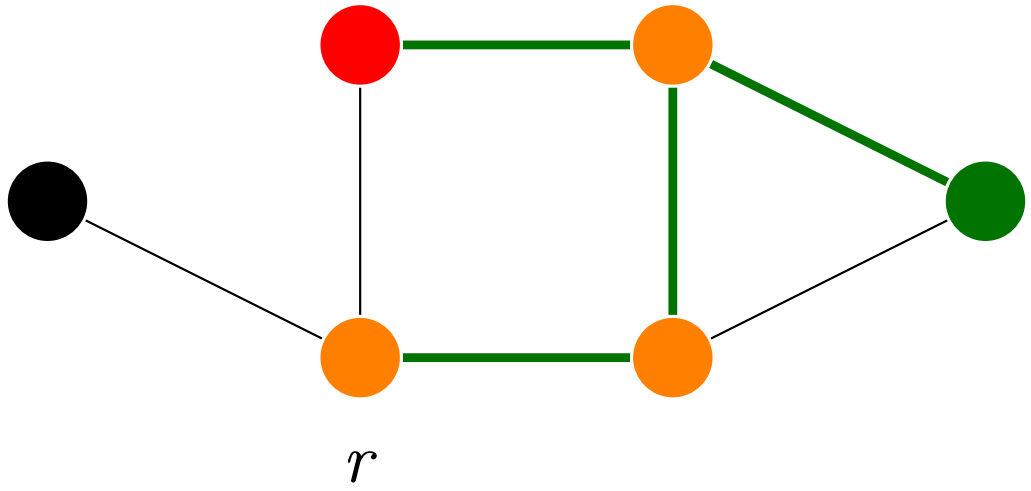
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



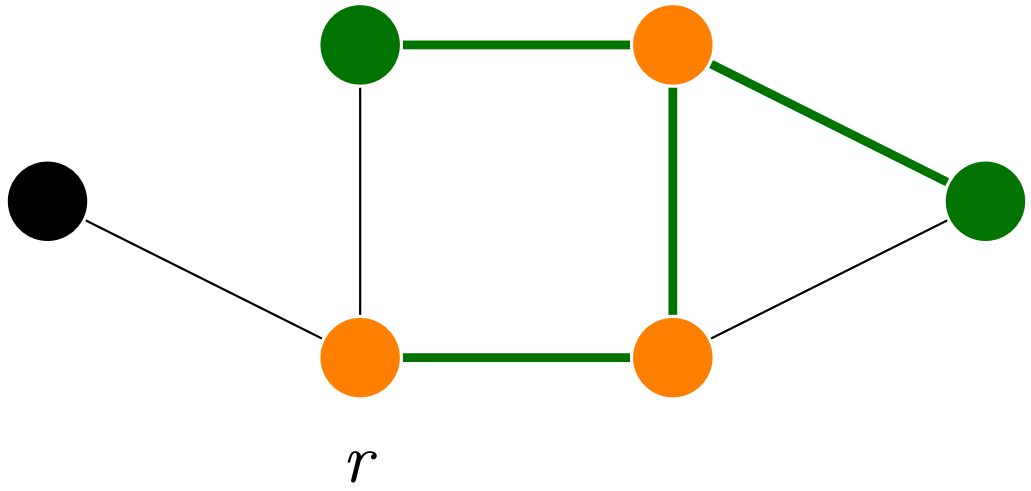
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



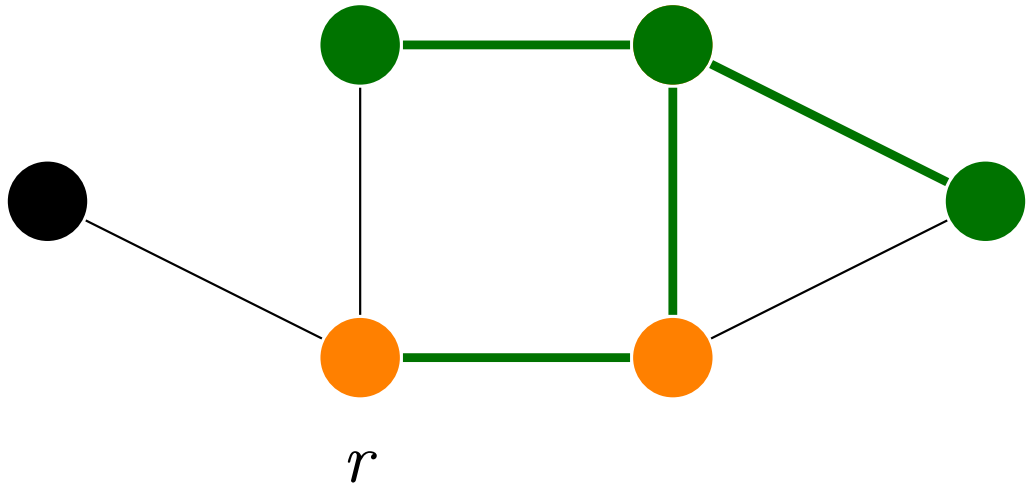
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



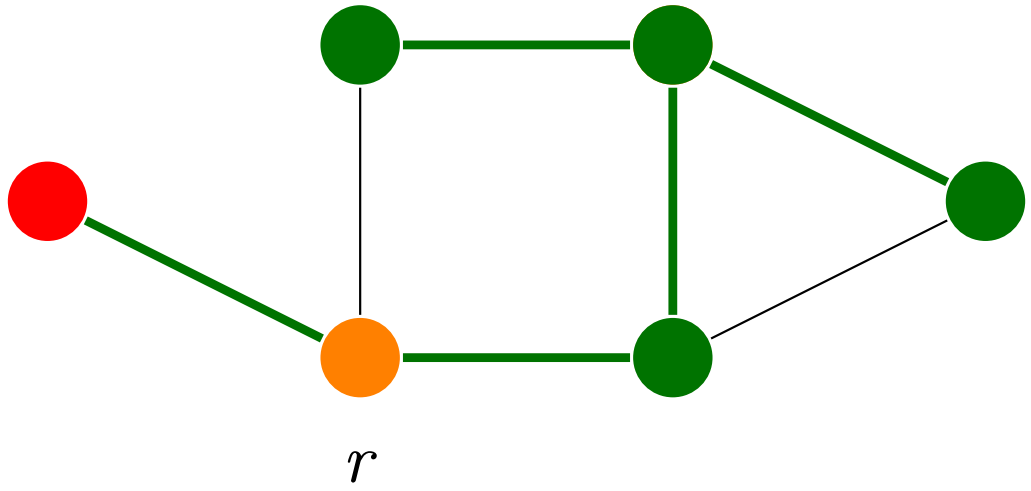
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



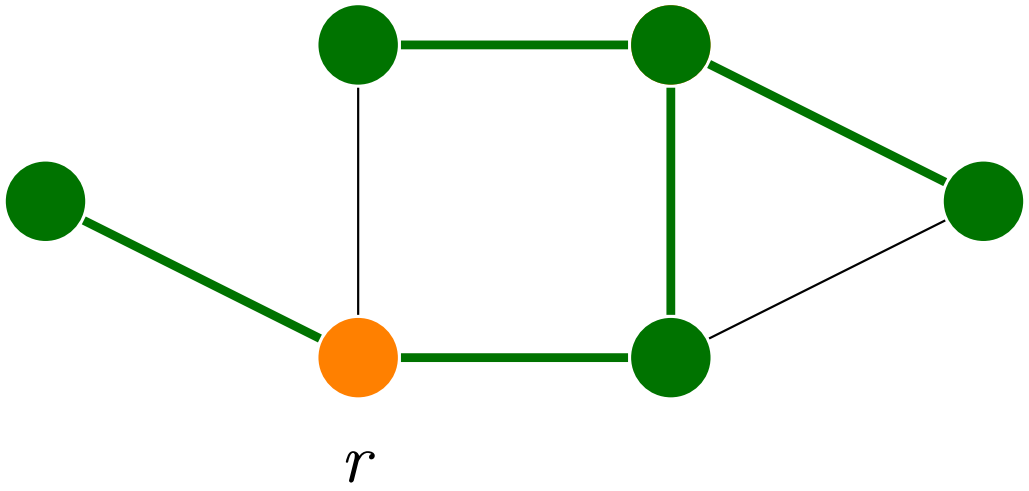
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



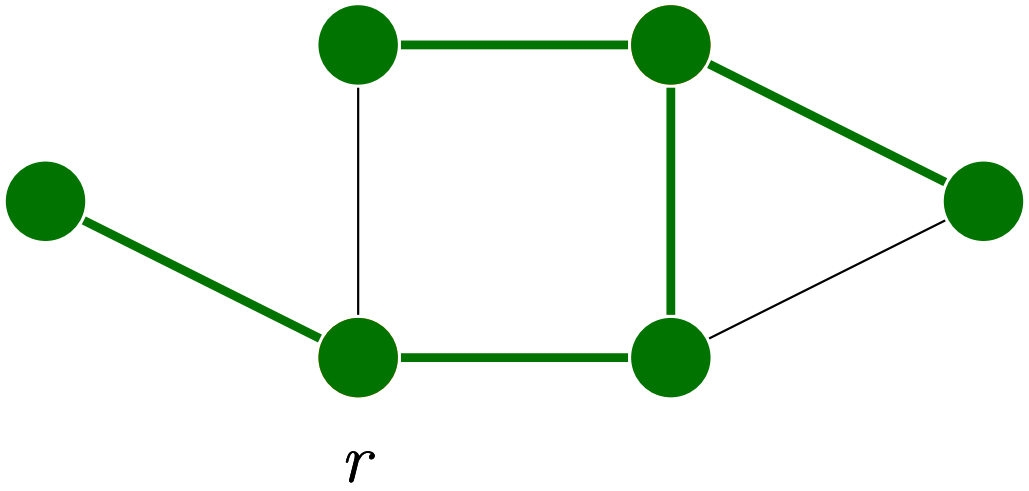
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Parcours en profondeur (DFS) : Exemple



sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Parcours en profondeur (DFS) : Exemple

Exercice

Adapter le code de la fonction ci-dessous pour afficher (avec `print`) les sommets dans l'ordre de leur visite dans le parcours en profondeur.

```
def dfs(G, s):
    visited = [False]*len(G)
    def aux(u):
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                aux(v)
    aux(s)
```

Parcours en profondeur (DFS) : Exemple

Exercice

Écrire un parcours en profondeur pour un graphe représenté par matrice d'adjacence.

Parcours en profondeur (DFS) : Application à la connexité

Question

Comment déterminer si un graphe **non orienté** est connexe?

Parcours en profondeur (DFS) : Application à la connexité

Question

Comment déterminer si un graphe **non orienté** est connexe?

Il suffit de vérifier que le tableau `visited` 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 :

```
def dfs(G, s):
    visited = [False]*len(G)
    def aux(u):
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                aux(v)
    for u in range(n):
        aux(u)
```

Parcours en profondeur (DFS) : Détection de cycle

Question

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

Parcours en profondeur (DFS) : Détection de cycle

Question

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

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

Parcours en profondeur (DFS) : Détection de cycle

Question

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

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

Parcours en profondeur (DFS) : Détection de cycle

Question

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

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

Parcours en profondeur (DFS) : Détection de cycle

Question

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

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

déjà visité

Parcours en profondeur (DFS) : Détection de cycle

Question

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

Il faut donc éviter de s'appeler récursivement sur son père :

```
def has_cycle(G, s):
    # renvoie True si G a un cycle atteignable depuis s
    visited = [False]*len(G)
    def aux(u, p): # p : sommet ayant permis de découvrir u
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                if v != p and aux(v, u):
                    return True
        return False
    return aux(s, -1)
```

Parcours en profondeur (DFS) : Avec pile

Au lieu d'utiliser la récursivité, on peut aussi stocker les prochains sommets à visiter dans une liste, utilisée comme une **pile** :

```
def dfs(G, s): # G représenté par liste d'adjacence
    visited = [False]*len(G)
    pile = [s]
    while len(pile) > 0:
        u = pile.pop()
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                pile.append(v)
```

Pile

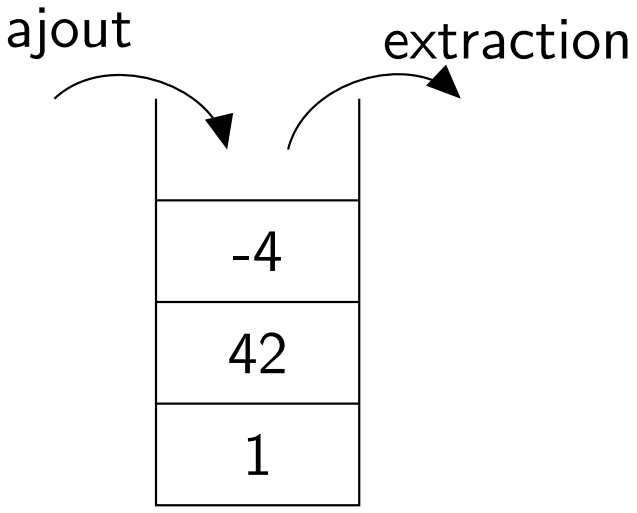
Une **pile** est une structure de donnée possédant les opérations :

- Ajout d'un élément au dessus de la pile.
- Extraction (suppression et renvoi) de l'élément au dessus de la pile. Ainsi, c'est toujours le dernier élément rajouté qui est extrait.

Pile

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

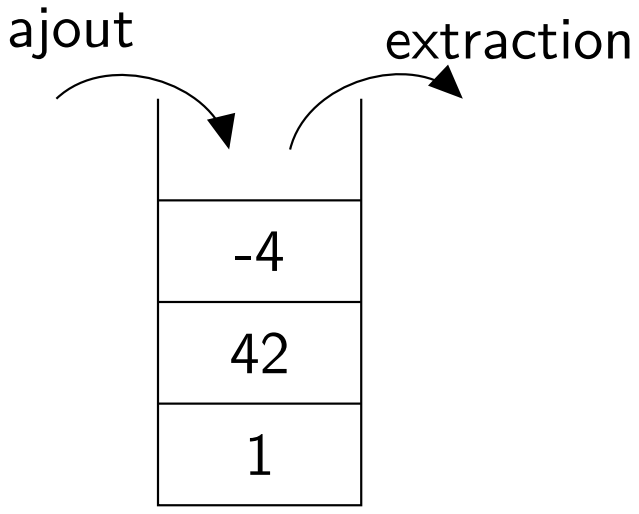
- Ajout d'un élément au dessus de la pile.
- Extraction (suppression et renvoi) de l'élément au dessus de la pile. Ainsi, c'est toujours le dernier élément rajouté qui est extrait.



Pile

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

- Ajout d'un élément au dessus de la pile.
- Extraction (suppression et renvoi) de l'élément au dessus de la pile. Ainsi, c'est toujours le dernier élément rajouté qui est extrait.



On peut implémenter une pile avec une liste `L` en Python, en utilisant `L.append(e)` pour ajouter un élément `e` et `L.pop()` pour l'extraction.

Pile

- La *call stack* (pile d'appel) est une pile utilisée par le processeur pour stocker les appels de fonctions en cours d'exécution ainsi que leur arguments. L'erreur *stack overflow* signifie qu'il n'y a plus assez de mémoire dans la *call stack*.

Pile

- La *call stack* (pile d'appel) est une pile utilisée par le processeur pour stocker les appels de fonctions en cours d'exécution ainsi que leur arguments. L'erreur *stack overflow* signifie qu'il n'y a plus assez de mémoire dans la *call stack*.
- Passer de récursif à itératif en simulant des appels récursifs. Par exemple, pour utiliser une pile au lieu d'une fonction récursive pour le parcours en profondeur.

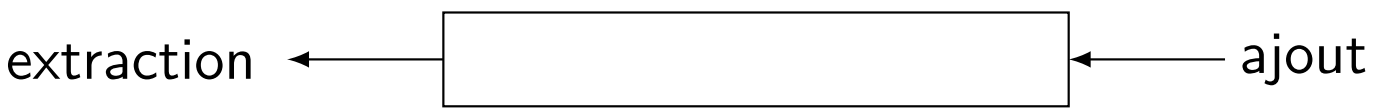
Pile

- La *call stack* (pile d'appel) est une pile utilisée par le processeur pour stocker les appels de fonctions en cours d'exécution ainsi que leur arguments. L'erreur *stack overflow* signifie qu'il n'y a plus assez de mémoire dans la *call stack*.
- Passer de récursif à itératif en simulant des appels récursifs. Par exemple, pour utiliser une pile au lieu d'une fonction récursive pour le parcours en profondeur.
- Dans un éditeur de texte, chaque action est ajoutée à une pile. Lorsque vous revenez en arrière (Ctrl + Z), l'éditeur extrait le dessus de la pile pour revenir à l'état précédent.

File

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.



File

On pourrait utiliser une liste `L` Python comme une file avec `L.append(e)` pour ajouter à la fin et `L.pop(0)` pour supprimer le 1er élément.
Mais `L.pop(0)` serait en O(n) (quand on supprime le 1er élément il faut décaler tous les autres), ce qui n'est pas satisfaisant.

File

On pourrait utiliser une liste `L` Python comme une file avec `L.append(e)` pour ajouter à la fin et `L.pop(0)` pour supprimer le 1er élément.
Mais `L.pop(0)` serait en $O(n)$ (quand on supprime le 1er élément il faut décaler tous les autres), ce qui n'est pas satisfaisant.

À la place, on peut utiliser la classe `deque` (pour *doubly ended queue*) du module `collections`, qui permet d'ajouter à la fin avec `append` et de supprimer au début avec `popleft` :

```
from collections import deque

q = deque() # file vide
q.append(4)
q.append(7)
q.popleft() # renvoie 4
q.append(-5)
q.popleft() # renvoie 7
```

Parcours en largeur (BFS) : Avec file

Si on utilise une file au lieu d'une pile dans le DFS, on obtient :

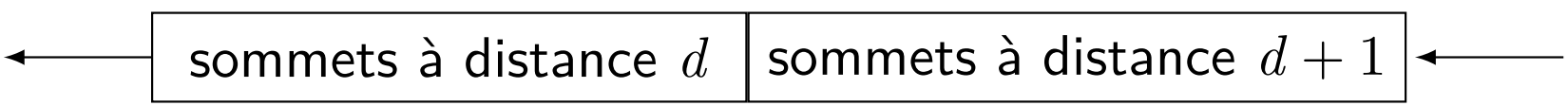
```
def bfs(G, s):
    visited = [False]*len(G)
    file = deque([s])
    while len(file) > 0:
        u = file.popleft()
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                file.append(v)
```

Parcours en largeur (BFS) : Avec file

Si on utilise une file au lieu d'une pile dans le DFS, on obtient :

```
def bfs(G, s):
    visited = [False]*len(G)
    file = deque([s])
    while len(file) > 0:
        u = file.popleft()
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                file.append(v)
```

La file est toujours de la forme :

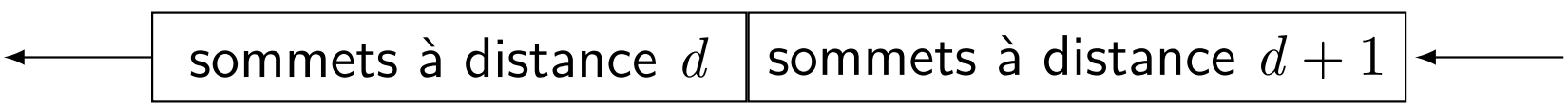


Parcours en largeur (BFS) : Avec file

Si on utilise une file au lieu d'une pile dans le DFS, on obtient :

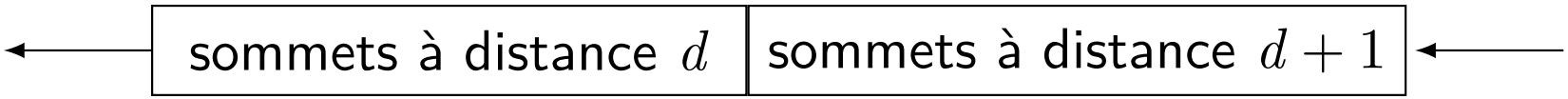
```
def bfs(G, s):
    visited = [False]*len(G)
    file = deque([s])
    while len(file) > 0:
        u = file.popleft()
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                file.append(v)
```

La file 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.

```
def bfs(G, s):
    visited = [False]*len(G)
    cur, next = [s], []
    while len(cur) + len(next) > 0:
        if len(cur) == 0:
            cur, next = next, []
        u = cur.pop()
        if not visited[u]:
            for v in G[u]:
                next.append(v)
```

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

Question

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

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

Question

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

Stocker des couples (sommet, distance) dans la file et stocker les distances dans un tableau `dist` (`dist[v]` = distance de `s` à `v`) :

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

Question

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

Stocker des couples (sommet, distance) dans la file et stocker les distances dans un tableau `dist` (`dist[v]` = distance de `s` à `v`) :

```
def distances(G, s):
    dist = [-1]*len(G)
    q = deque([(s, 0)])
    while len(q) > 0:
        u, d = q.popleft()
        if dist[u] == -1:
            dist[u] = d
            for v in G[u]:
                q.append((v, d + 1))
    return dist
```

Parcours en largeur (BFS) : Plus courts chemins

Question

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

Parcours en largeur (BFS) : Plus courts chemins

Question

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

Stocker `pred[v]` = prédécesseur de `v` dans le parcours :

```
def bfs(G, s):
    pred = [-1]*len(G)
    q = deque([(s, s)])
    while len(q) > 0:
        u, p = q.popleft()
        if pred[u] == -1:
            pred[u] = p
            for v in G[u]:
                q.append((v, u))
    return pred
```

Parcours en largeur (BFS) : Plus courts chemins

Question

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

On peut ensuite remonter les prédécesseurs de `v` à `s` :

```
def path(pred, s, v):
    L = []
    while v != s:
        L.append(v)
        v = pred[v]
    L.append(s)
    return L[::-1] # inverse le chemin
```

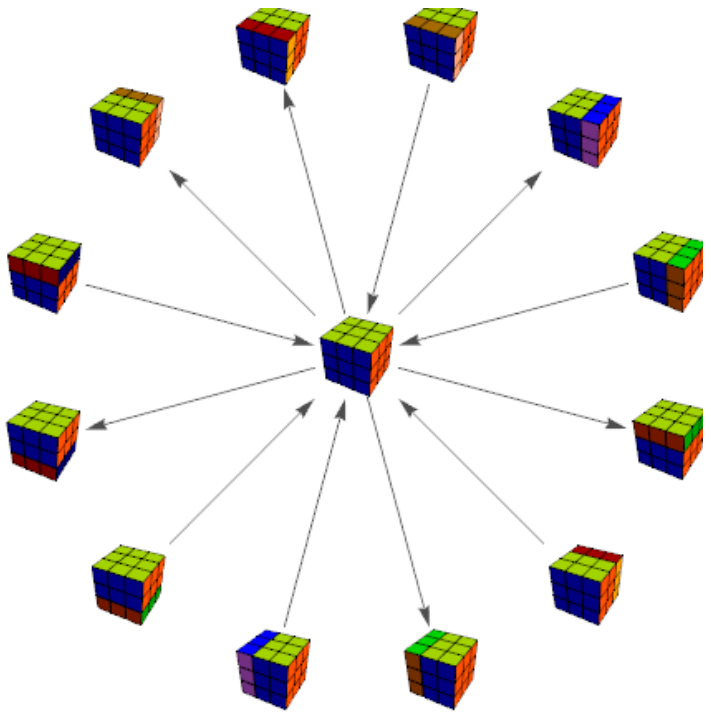
Parcours en largeur (BFS) : Plus courts chemins

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

Parcours en largeur (BFS) : Plus courts chemins

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

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



Parcours en largeur (BFS) : Plus courts chemins

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

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

Théorème (2010)

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

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