Parcourir un graphe en largeur

1. Stratégie de parcours

L'idée du parcours est de visiter tous les sommets d'un graphe en partant d'un sommet quelconque.
Ces algorithmes de parcours d'un graphe sont à la base de nombreux algorithmes très utilisés : routage des paquets de données dans un réseau, découverte du chemin le plus court pour aller d'une ville à une autre...

Il existe deux méthodes pour parcourir un graphe :

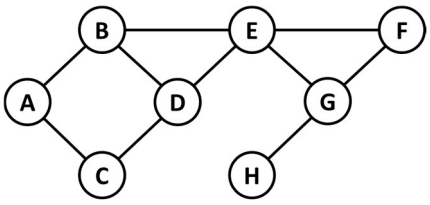
- Le **parcours en largeur** d'abord (**BFS**(Breadth First Search)) : l'idée est de partir d'un sommet, d'explorer tous ses voisins, puis d'explorer tous les voisins de ses voisins et ainsi de suite…
- Le **parcours en profondeur** d'abord (**DFS**(Depth First Search)) : l'idée est de partir d'un sommet, d'explorer un chemin à partir de ce sommet, lorsqu'on arrive sur un sommet qui n'a plus de voisins non visités, on remonte pour explorer les chemins non visités.

2. Parcours en largeur

2.1. Principe du parcours en largeur

Parcourir un graphe en largeur à partir d'un sommet, consiste à visiter le sommet puis ses enfants, puis les enfants de ses enfants...
Comme on l'a déjà vu avec les arbres, il faut utiliser une **file** et une **liste** pour marquer les sommets visités.

Prenons en exemple ce graphe :

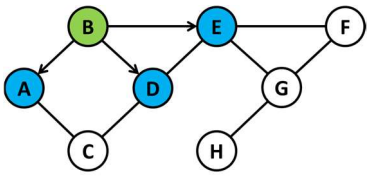


On dispose d'un graphe G, d'une liste des **`sommets_visités`** et d'une file des **`sommets_a_visiter`**.
Le sommet de départ est par exemple B, **on l'enfile**.
Puis tant que la file **`sommets_a_visiter`** n'est pas vide :

- On défile **`sommets_a_visiter`** dans une variable **`t`**.
- Si **`t`** n'est pas dans **`sommets_visités`**
  - On l'ajoute à **`sommets_visités`**
- Pour chaque voisin de **`t`**
  - S'il n'est ni dans **`sommets_visités`** ni dans la file **`sommets_a_visiter`**
    - On l'enfile
- On renvoie **`sommets_visités`**

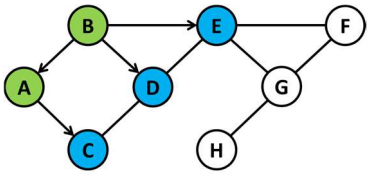
Au premier tour de la boucle **`tant que`**, les contenus des variables sont :

```
t = B
sommets_visités = [B]
sommets_a_visiter = [A, D, E]
```



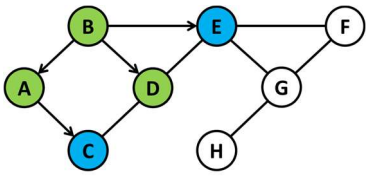
Au second tour :

```
t = A
sommets_visités = [B, A]
sommets_a_visiter = [D, E, C]
```

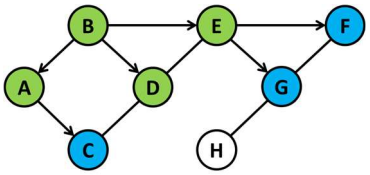


✍ Compléter les contenus des variables **t**, **sommets_visités** et **sommets_a_visiter** aux tours suivants.

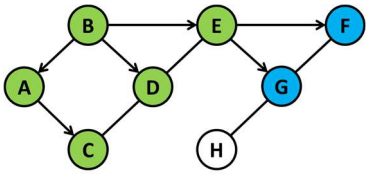
Au 3ème tour :
t =
sommets_visités =
sommets_a_visiter =



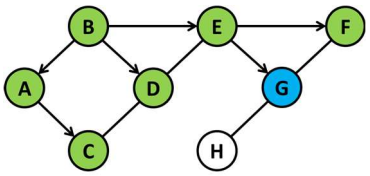
Au 4ème tour :
t =
sommets_visités =
sommets_a_visiter =



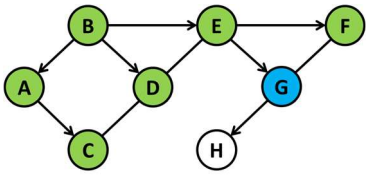
Au 5ème tour :
t =
sommets_visités =
sommets_a_visiter =



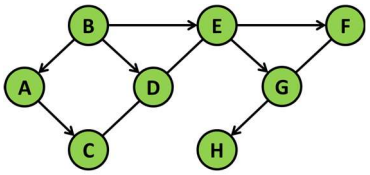
Au 6ème tour :
t =
sommets_visités =
sommets_a_visiter =



Au 7ème tour :
t =
sommets_visités =
sommets_a_visiter =

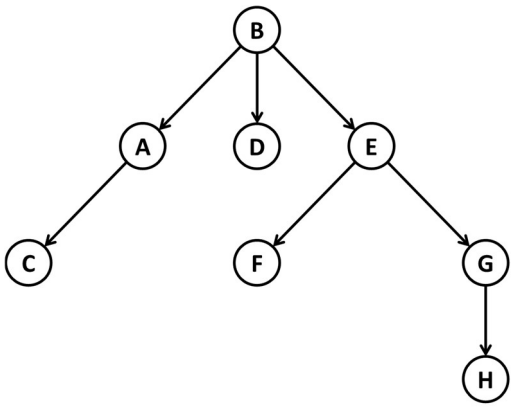


Au 8ème tour :
t =
sommets_visités =
sommets_a_visiter =



Au final l'arborescence associée au parcours peut donc être modélisée de la façon suivante :

[B, A, D, E, C, F, G, H]



2.2. Implémentation du parcours en largeur

L'algorithme de parcours en largeur qui permet de retourner la liste des sommets visités est le suivant :

```
Enfiler le sommet de départ
Tant que la file des sommets à visiter n'est pas vide faire
     Défiler la file dans une variable t
     Si t n'a pas été visité
          Alors l'ajouter aux sommets déjà visités
     Fin si
     Pour chaque voisin de t faire
          S'il ne fait pas parti des sommets visités et à visiter
               Alors l'enfiler
          Fin si
     Fin pour
Fin tant que
Renvoyer les sommets visités
```

Pour l'implémentation de l'algorithme de parcours en largeur, il est nécessaire d'utiliser une classe File. La classe suivante peut être utilisée. La méthode **`present(self, x)`** à été rajoutée. Elle renvoie vrai si x est dans la file.

```
class File:
    ''' classe File : création d'une instance File avec une liste '''

    def __init__(self):
        self.L = []

    def vide(self):
        return self.L == []

    def defiler(self):
        assert not self.vide(), "file vide"
        return self.L.pop(0)

    def enfiler(self,x):
        self.L.append(x)

    def taille(self):
        return len(self.L)

    def sommet(self):
        return self.L[0]

    def present(self,x):
        return x in self.L
```

Le code ci-dessous permet de créer le graphe **g** à partir d'un dictionnaire.

```
g = dict()
g['A'] = ['B','C']
g['B'] = ['A','D','E']
g['C'] = ['A','D']
g['D'] = ['B','C','E']
g['E'] = ['B','D','F','G']
g['F'] = ['E','G']
g['G'] = ['E','F','H']
g['H'] = ['G']
```

La fonction **voisins(graphe, sommet)** renvoie les voisins d'un sommet.

```
def voisin(graphe, sommet):
    """renvoie les voisins d'un sommet"""
    return graphe[sommet]
```

✍ Implémenter l'algorithme de parcours en largeur sous la forme de la fonction **bfs(graphe, sommet)** prenant en paramètre le graphe **graphe** (saisi sous forme d'une liste d'adjacence) et le sommet **sommet** de départ et retournant la liste des sommets visités.

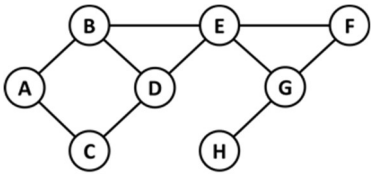
✍ Vérifier l'algorithme avec les tests ci-dessous et le graphe **g**.

- `bfs({'S': []},'S') == ['S']`
- `bfs({'A': ['B', 'C'],'B': [],'C': []},'A') == ['A', 'B', 'C']`

3. Recherche d'un chemin entre deux sommets d'un graphe

Sachant parcourir un graphe, le but maintenant est d'afficher un chemin entre deux sommets d'un graphe.

Par exemple A – B – E – G – H est un chemin possible entre les sommets A et H.



La méthode consiste à mémoriser les sommets voisins du sommet visité comme clés d'un dictionnaire et ayant pour valeur son parent (le sommet visité).
Le sommet de départ n'aura bien entendu pas de parent (**None**).
À la fin, le dictionnaire parents sera :

```
{'A': None, 'B': 'A', 'C': 'A', 'D': 'B', 'E': 'B', 'F': 'E', 'G': 'E', 'H': 'G'}
```

Il nous faudra lire ce dictionnaire pour pouvoir établir le chemin entre A et H.
H a pour parent G qui a pour parent E qui a pour parent B qui a pour parent A.
D'où le chemin : A – B – E – G – H.

✍ Modifier la fonction **bfs(graphe, sommet)** afin d'obtenir la fonction **bfs_chemin(graphe, sommet)**.
La fonction **bfs_chemin(graphe, sommet)** doit retourner un dictionnaire, nommé parents avec tous les sommets parcourus et leur parent.
Le départ n'ayant pas de parents, le dictionnaire sera initialisé comme ceci :

```
parents[sommet] = None
```

Le dictionnaire parents contient les sommets visités (clés) et leurs parents (valeurs).
Il faut maintenant exploiter ce dictionnaire pour faire afficher un chemin entre deux sommets.
L'idée est de lire le parcours du graphe à partir d'**arrivee** à l'envers en remontant les parents.

```
Initialiser une liste solution à vide
Stocker le parcours du graphe à partir de depart avec les parents dans un dictionnaire parent
Initialiser le sommet visité à arrivee
Tant que le sommet visité à un parent faire
     solution ← sommet visité
     le sommet visité devient le parent
Fin tant que
Renvoyer la liste solution (de depart à arrivee)
```

✍ Implémenter une fonction **chemin(graphe, depart, arrivee)** prenant en paramètre un graphe, le sommet de départ et le sommet d'arrivé et retournant la route à suivre sous forme d'une liste de sommets.

```
>>> chemin(g, 'A', 'H')
['A', 'B', 'E', 'G', 'H']
```

Source : https://isn-icn-ljm.pagesperso-orange.fr/NSI-TLE/co/section_chapitre4.html