Chapitre # (ALGO) 11 Parcours en largeur d'un graphe

Rien ne se passe dans l'Univers sans qu'un minimum ou un maximum apparaisse.
— **Leonhard Euler**

Résumé & Plan

Le parcours en largeur consiste, à partir d'un sommet donné, de visiter tous les sommets successeurs. On répète l'opération tant qu'il existe des sommets non visités. Ce type de parcours est d'une manière imagée, une sorte de propagation.

1. PARCOURS EN LARGEUR

Dans le TP précédent, nous avons étudié les graphes et nous avons vu quelques implémentation possibles en Python, notamment par dictionnaire des voisins que nous allons largement utiliser ici. Un des premiers algorithmes qu'on doit savoir utiliser sur un graphe est celui de son parcours (un analogue de boucle `for` en quelque sorte, mais adaptée aux graphes), c'est ce que nous voyons à présent.

1.1. Objectif

Parcourir un graphe, c'est visiter ses différents sommets, en partant d'un sommet quelconque, afin de pouvoir opérer une action tour à tour sur eux.

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, trouver la sortie d'un labyrinthe...

Le choix d'un parcours dépend essentiellement du contexte et des attentes d'un problème. Cette description sommaire des parcours révèle une autre différence essentielle entre les graphes et les structures de données linéaires (comme les listes). En raison de l'existence de cycles, un parcours peut ramener à un sommet déjà visité. Pour en tenir compte et ne pas recommencer un parcours déjà fait, il convient de garder une information sur le fait qu'un sommet ait été visité ou pas lors d'un parcours. Cette information peut être stockée dans une structure de données adaptée.

Cadre

Dans la suite, nous supposons le graphe simple ***(sans arête multiple ou boucle)*** **et non orienté .**

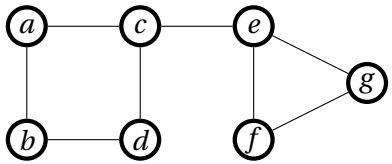
Il existe deux manières principales de parcourir ainsi les sommets d'un graphe non pondéré :

- le parcours en profondeur, que l'on utilise par exemple pour explorer un labyrinthe. Ce parcours sera vu en 2ème année.
- Le parcours en largeur, que l'on étudie dans cette section.

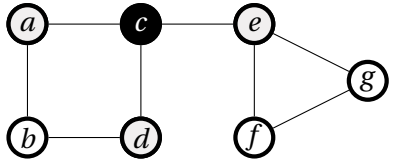
En 2ème année, vous verrez une généralisation du parcours en largeur aux graphes pondérés (algorithmes de Dijkstra).

Exemple de parcours en largeur d'un graphe non-orienté. Le parcours en largeur d'un graphe à partir d'un sommet donné consiste à commencer par explorer les voisins du sommet, puis les voisins de ses voisins, puis les voisins des voisins de ses voisins, et ainsi de suite.

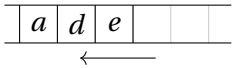
Considérons le graphe ci-dessous. Déroulons cet algorithme de parcours par exemple à partir du sommet c.



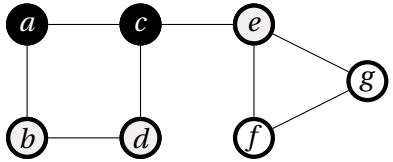
On marque c comme *visité*, puis ses sommets adjacents sont découverts, marqués en gris clair et mémorisés dans ce que l'on appelle une *file* (indiquée sous le graphe, la terminologie « file » sera expliquée plus tard) : a d'abord, d ensuite, e enfin (par exemple dans l'ordre alphabétique, mais cet ordre n'est pas important). On obtient donc le résultat suivant :



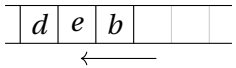
sommets visités : `['c']`



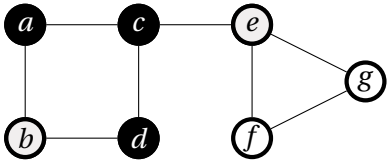
Il faut ensuite recommencer le parcours depuis l'un des sommets en gris clair (l'un des sommets de la file). On recommence avec le sommet a à gauche (c'est-à-dire le premier sommet que l'on a ajouté dans la file) et on le marque comme *visité*. On ajoute alors dans la file les voisins de a **qui n'ont pas encore été visités** et **non encore présents dans la file** : seul b est ajouté.



sommets visités : `['c','a']`

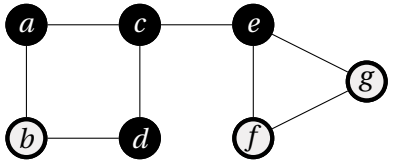


On recommence avec le sommet d à gauche, il est marqué comme visité. Rien n'est ajouté dans la file, puisque b l'était déjà.

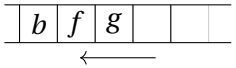


sommets visités : `['c','a','d']`

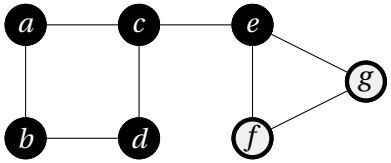
On recommence avec le sommet e à gauche, il est marqué comme visité. Les sommets f et g sont placés à leur tour dans la file.



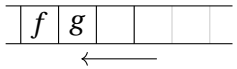
sommets visités : `['c','a','d','e']`



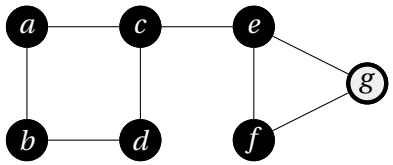
On recommence avec le sommet b, il est marqué comme visité. Il n'a plus de sommets voisins non visités, aucun sommet n'est rajouté dans la file.



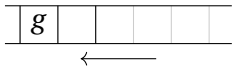
sommets visités : `['c','a','d','e','b']`



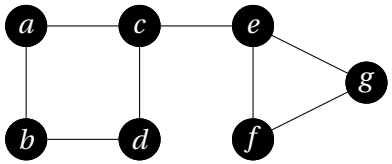
On recommence avec le sommet f, on le marque comme visité, aucun sommet n'est placé dans la file.



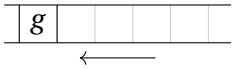
sommets visités : `['c','a','d','e','b','f']`



On recommence avec le sommet g, on le marque comme visité. Aucun autre sommet n'est placé dans la file



sommets visités : `['c','a','d','e','b','f','g']`



Désormais la file d'attente est vide, le parcours est terminé. À l'issue de l'algorithme, on a un parcours dans un certain ordre de tous les sommets *via* la liste des sommets visités. Ce parcours correspond bien à un parcours en largeur : on a d'abord le sommet à distance 0 de la source (`c`), puis à distance 1 (`a, d, e`), et enfin à distance 2 (`b, f, g`).

STRUCTURES DE DONNÉES UTILISÉES. Analysons à partir de l'exemple précédent les structures de données qui seront utilisées. On a besoin :

- d'une liste `L_Visites`, initialement vide, contenant tous les sommets que l'on a déjà visités dans le parcours. C'est la liste que l'on renverra à la fin du parcours : elle contiendra alors tous les sommets du graphe $\mathcal{G}$ qui sont accessibles depuis le sommet s.
- La liste `File` sera la file d'attente des sommets (représentée sur le dessin plus haut en-dessous du graphe) que l'on doit encore visiter : les premiers sommets que l'on visitera seront les premiers de la file, et les nouveaux sommets à visiter seront ajoutés en queue de liste (avec `append`). Initialement, la liste `File` contiendra le sommet de départ s.

Note | *C'est pour cette raison que l'on qualifie cette liste de « file » : les sommets sont traités selon la priorité « premier arrivé premier servi », comme une file d'attente dans la vie courante.*

- À une étape donnée, on a besoin de tester souvent si un sommet est découvert/visité ou non (c'est-à-dire coloré en gris ou noir, ou non). On peut tout simplement tester l'appartenance à `L_Visites` ou à la file, mais cela est coûteux en temps. Il est préférable d'introduire un dictionnaire `est_decouvert` de clefs les sommets et valeurs `True` si le sommet a déjà été visité ou découvert, `False` dans le cas contraire. On modifiera donc ce dictionnaire à chaque coloration en gris ou noir d'un sommet.

1.2. Implémentation en Python

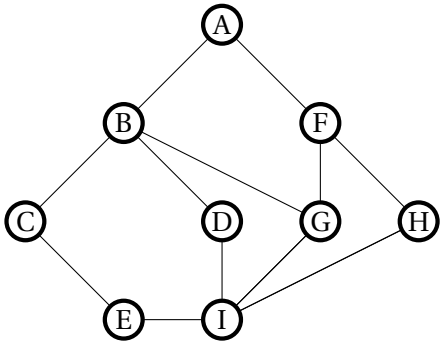
Récapitulons l'algorithme de parcours en largeur présenté dans l'exemple précédent.

(Parcours en largeur) Tant que la liste `File` n'est pas vide, c'est-à-dire tant qu'il reste dans la file d'attente des sommets à explorer, on effectue les actions suivantes :

- on visite le premier sommet (donc celui de gauche sur les dessins précédents) de la file d'attente `File` :
  - ◇ on l'ajoute à la liste `L_Visites` des sommets visités ;
  - ◇ dans le dictionnaire `est_decouvert`, on passe à `True` la valeur du sommet ;
  - ◇ on le supprime de la file d'attente `File` ;
- on ajoute à la file d'attente `File` tous les voisins de ce sommet qui ne sont pas déjà visités ou découverts. *(Ainsi, un sommet ne peut pas apparaitre plusieurs fois dans la file)*

Exercice 1 | Solution Étudier l'algorithme ci-dessus et appliquez-le à la fin (dans un tableau) au graphe ci-dessous avec comme sommet de départ le sommet A. Vous noterez à chaque étape, dans un tableau :

- les sommets atteints (liste `L_Visites`)
- les sommets présents dans la file d'attente (liste `File`).



Ci-après le début du tableau :

L_Visites	File
[]	[A]
[A]	[B,F]
⋮	⋮

Exercice 2 | Code du parcours en largeur Solution

1. Compléter le code ci–dessous :

```
def Parcours_largeur(G, s):
    """
    Arguments :
    - un graphe donné sous forme d'un
    dictionnaire G des voisins ;
    - un sommet s du graphe.
    Sortie : la liste des sommets accessibles depuis le sommet \
    ↪ s, dans l'ordre du parcours en largeur.
    """
    est_decouvert = {}
    for x in G:
        est_decouvert[x] = __________

    File = [s]
    est_decouvert[s] = True

    L_Visites = []
    while len(File) != __________ :
        sommet = __________ # on récupère le premier sommet \
        ↪ de la file et on le supprime
        L_Visites.append(__________)
        est_decouvert[sommet] = __________
        for voisin in G[sommet]:
            if __________:
                File.append(__________)
                est_decouvert[voisin] = __________
    return __________
```

On rappelle que la méthode `L.pop(k)` renvoie l'élément d'indice `k` d'une `L` et l'enlève de `L`.

2. Implémenter le graphe de l'exercice précédent (à l'aide d'un dictionnaire), et tester la fonction sur celui-ci.

Exercice 3 | Chemin reliant deux sommets Solution

1. Écrire une fonction `existe_chemin(G, s, t)` qui prend en arguments un graphe et deux sommets s et t du graphe, et qui détermine s'il existe un chemin du sommet s vers le sommet t dans le graphe. *Il suffit par exemple de parcourir le graphe en largeur à partir du sommet s et de regarder si t est dans la liste des sommets découverts. Le code ne doit pas prendre plus de 3 lignes !*
2. Si cela est possible, on désire alors renvoyer un chemin du sommet s vers le sommet t. Pour cela il nous faut connaitre pour chaque sommet découvert, le « père » (c'est-à-dire sommet dont il a été identifié comme voisin).
   - **2.1)** Modifier la fonction d'en-tête `parcours_largeur(G, s)` en `parcours_largeur_pred(G, s, t)` pour quelle renvoie, en plus des sommets visités, le père de chaque sommet découvert. *On pourra stocker cette information dans un dictionnaire `pred` de clefs les sommets, et valeurs le père en question. Il n'y a que deux ou trois lignes à modifier !*
   - **2.2)** Écrire alors une fonction `chemin(G, s, t)` qui renvoie un chemin de s à t. On pourra utiliser la méthode `L.insert(i,elem)` permettant d'insérer à l'indice `i` d'une liste `L` l'élément `elem`.
   - **2.3)** Tester la fonction chemin sur le graphe précédent pour obtenir un chemin de A à I (par exemple).

Exercice 4 | Le chou, la chèvre et le loup Solution

Une devinette raconte l'histoire d'un berger qui possède un chou, une chèvre et un loup. En sa présence, la chèvre n'ose pas manger le chou, pas plus que le loup n'ose manger la chèvre, mais ils n'hésiteraient pas à satisfaire leur appétit si l'homme tournait le dos. Ce berger doit traverser une rivière avec sa petite troupe et il ne dispose que d'une barque, dans laquelle il peut naviguer avec un seul de ses compagnons. Comment doit-il s'y prendre pour amener tout le monde sur l'autre rive ? Par exemple, il doit éviter de laisser la chèvre et le loup seuls sur une rive pendant une traversée.

Ce problème peut se modéliser comme un parcours de graphe. Tout d'abord, chacun des quatre protagonistes pouvant se trouver sur une rive ou l'autre, en enlevant les cas dramatiques, il nous reste dix états :

	Rive de départ	Rive d'arrivée
0	–	chou, chèvre, loup, berger
1	chou	chèvre, loup, berger
2	chèvre	chou, loup, berger
3	loup	chou, chèvre, berger
4	chou, loup	chèvre, berger
5	chèvre, berger	chou, loup
6	chou, chèvre, berger	loup
7	Chou, loup, berger	chèvre
8	chèvre, loup, berger	chou
9	chou, chèvre, loup, berger	–

1. Ces dix états numérotés de 0 à 9 constituent les sommets du graphe. Deux sommets sont voisins si l'on peut passer d'un état à l'autre par une traversée en bateau. Dessiner le graphe correspondant.

2. Résoudre le problème à l'aide d'un parcours de graphe.

2. SOLUTIONS DES EXERCICES

Solution (exercice 1) Énoncé

L_Visites	File
[]	[A]
[A]	[B,F]
[A,B]	[F,C,D,G]
[A,B,F]	[C,D,G,H]
[A,B,F,C]	[D,G,H,E]
[A,B,F,C,D]	[G,H,E,I]
[A,B,F,C,D,G]	[H,E,I]
[A,B,F,C,D,G,H]	[E,I]
[A,B,F,C,D,G,H,E]	[I]
[A,B,F,C,D,G,H,E,I]	[]

Remarque : on a choisi ici de ranger dans la file d'attente les voisins dans l'ordre lexicographique, ce choix est fait au moment du codage du dictionnaire des voisins. Le parcours est donc `[A,B,F,C,D,G,H,E,I]`.

Solution (exercice 2) Énoncé

```
def Parcours_largeur(G, s):
    """
    Arguments :
    - un graphe donné sous forme d'un
    dictionnaire G des voisins ;
    - un sommet s du graphe.
    Sortie : la liste des sommets accessibles depuis le \
    ↪ sommet s, dans l'ordre du parcours en largeur.
    """
    est_decouvert = {}
    for x in G:
        est_decouvert[x] = False

    File = [s]
    est_decouvert[s] = True
    L_Visites = []
    while len(File) != 0:
        sommet = File.pop(0)
        L_Visites.append(sommet)
        est_decouvert[sommet] = True
        for voisin in G[sommet]:
            if not est_decouvert[voisin]:
                File.append(voisin)
                est_decouvert[voisin] = True
    return L_Visites

G = {}
G["A"] = ["B","F"]
G["B"] = ["A","C","D","G"]
G["C"] = ["E"]
G["D"] = ["B","I"]
G["E"] = ["C","I"]
G["F"] = ["A","G","H"]
G["G"] = ["B","F","I"]
G["H"] = ["F","I"]
G["I"] = ["D","E","G","H"]
>>> Parcours_largeur(G, "A")
['A', 'B', 'F', 'C', 'D', 'G', 'H', 'E', 'I']
```

Solution (exercice 3) Énoncé

1. On parcours le graphe et on s'arrête dès que l'on a trouvé t.

```
def existe_chemin(G, s, t):
    """
    Arguments :
    - un graphe donné sous forme d'un
    dictionnaire D d'adjacence ;
    - deux sommets s et t du graphe.
    Sortie : True si il existe un chemin de s vers t
    False si non
    """
    L_Visites = Parcours_largeur(G, s)
    return t in L_Visites
```

Par exemple, reprenons le graphe de l'exercice précédent.

```
>>> existe_chemin(G, "A", "C")
```

```
True
```

La fonction indique bien qu'il existe un chemin menant de A à C. En revanche, si l'on construit un graphe à trois sommets A, B, C non connexe avec uniquement une arête reliant A et B, alors cette fonction renvoie **False** pour l'existence d'un chemin entre A et C.

```
Gbis = {}
Gbis["A"]=["B"]
Gbis["B"]=["A"]
>>> existe_chemin(Gbis, "A", "C")
False
```

2. **2.1)**

```
def parcours_largeur_pred(G, s, t):
    """
    Arguments :
    - un graphe donné sous forme d'un dictionnaire des voisins
    - deux sommets s et t du graphe.
    Sortie : les sommets visités et le dictionnaire des prédécesseurs
    """
    est_decouvert = {}
    for x in G:
        est_decouvert[x] = False
    pred = {}

    File = [s]
    est_decouvert[s] = True

    L_Visites = []
    while len(File) != 0:
        sommet = File.pop(0)
        L_Visites.append(sommet)
        est_decouvert[sommet] = True
        for voisin in G[sommet]:
            if not est_decouvert[voisin]:
                File.append(voisin)
                est_decouvert[voisin] = True
                pred[voisin] = sommet
    return L_Visites, pred
```

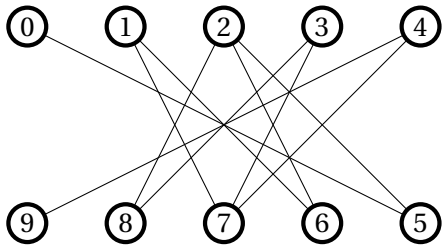
2.2) Pour reconsituer le chemin, on remonte alors la liste des prédécesseurs jusqu'à revenir à l'arrivée.

```
def chemin(G, s, t):
    if existe_chemin(G, s, t):
        #si le chemin est possible
        L_Visites, pred = parcours_largeur_pred(G,s,t)
        chemin = [t]
        sommet = t
        while sommet != s:
            sommet = pred[sommet]
            chemin.insert(0, sommet) # ajout en première position de chemin
        return chemin
```

Par exemple, reprenons le graphe de l'exercice précédent.

```
>>> chemin(G, "A", "I")
['A', 'B', 'D', 'I']
```

Solution (exercice 4) Énoncé



Pour résoudre le problème, il s'agit de savoir s'il existe un chemin menant de l'état 9 (tout le monde est sur la rive de départ), à l'état 0 (tout le monde est sur la rive d'arrivée).

```
G_berger = {0:[5], 1:[6,7], 2:[5,6,8], 3:[7,8], 4:[7,9], 5:[0,2],
    6:[1,2], 7:[1,3,4], 8:[2,3], 9:[4]}
```

On en déduit alors la solution :

```
>>> chemin(G_berger, 9, 0)
[9, 4, 7, 1, 6, 2, 5, 0]
```