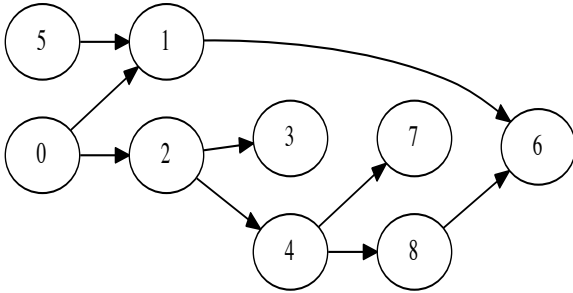


TP - Parcours de graphes

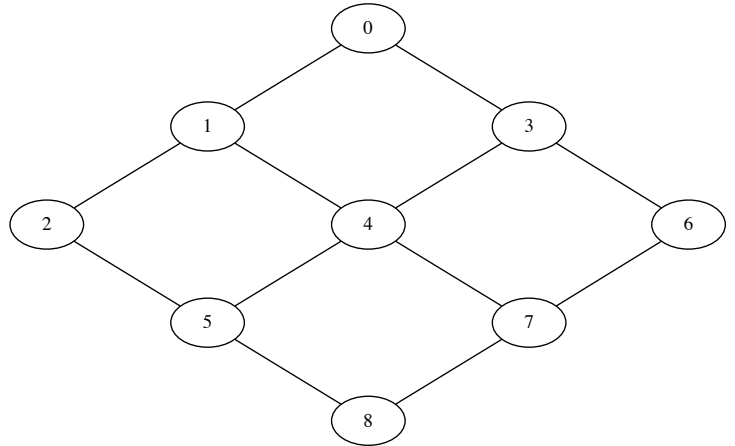
Comme il est de coutume, on lance spyder, et on poursuit sa lecture. On téléchargera depuis l'adresse <https://cahier-de-prepa.fr/pcsi-bdb/docs?rep=103> le script `tpParcours.py`, le fichier `sgb-words.txt` et on enregistrera ces deux fichiers dans un même dossier (qui pourrait bien s'appeler `tpParcours`)

1 Quelques rappels

Un graphe est la donnée d'un ensemble S de sommets, quelquefois appelés aussi nœuds, et d'un ensemble A d'arêtes. Les arêtes relient deux sommets, et peuvent être orientées (on parle alors de graphe orienté) ou non.



G_1 : un graphe orienté



G_2 : un graphe non-orienté

Les arêtes peuvent aussi être munies de poids (par exemple, le coût correspondant à son parcours, comme pourrait l'être une distance), ou non. Dans ce TP, nous n'étudierons que des graphes non pondérés, à l'image des deux exemples ci-dessus.

Nous coderons par ailleurs les graphes étudiés à l'aide de listes d'adjacence, où à chaque sommet on associe la liste de ses sommets voisins. On peut coder cela en python ou bien avec une liste de listes, en supposant les sommets numérotés à partir de 0 ou bien à l'aide d'un dictionnaire (ce qui ne suppose plus nécessairement les étiquettes des sommets formées d'entiers consécutifs, mais pouvant être aussi, par exemple, des chaînes de caractères)

Par exemple, le graphe G_1 peut être représenté par la liste `L1` valant `[[1, 2], [6], [3, 4], [], [7, 8], [1], [], [], [6]]`, ainsi par exemple `L1[4] = [7, 8]` ce qui indique que les deux sommets joints au sommet 4 sont les sommets 7 et 8, ou bien par le dictionnaire `D1` valant `{0: [1, 2], 1: [6], 2: [3, 4], 3: [], 4: [7, 8], 5: [1], 6: [], 7: [], 8: [6]}`

Parcourir un graphe depuis un sommet donné est déterminer l'ensemble des sommets que l'on peut atteindre depuis le sommet de départ en suivant les arêtes du graphe. Quel que soit l'algorithme employé, on devra toujours trouver les mêmes sommets, mais l'ordre d'obtention de ces sommets pourra, lui, différer. Par exemple, pour le graphe G_1 et depuis le sommet 0, tous les sommets du graphe seront obtenus, à l'exception du sommet 5. Depuis le sommet 3, aucun autre sommet que 3 ne sera obtenu.

Au contraire, depuis n'importe quel sommet de G_2 , le parcours de ce graphe conduira à lister tous les sommets de G_2 .

2 Parcours en largeur

Le parcours en largeur revient à lister dans un premier temps tous les sommets voisins du sommet initial, puis tous ceux à une distance de 2 arêtes, puis de 3 et ainsi de suite. Par exemple, pour le graphe G_2 depuis le sommet 0, un parcours en largeur pourrait être (il n'y a pas unicité d'un tel parcours) : (0, 1, 3, 2, 4, 6, 5, 7, 8).

La méthode classique pour implémenter un parcours en largeur consiste à utiliser une file d'attente (voir l'annexe pour quelques rappels sur `deque`). On introduira également une liste qui sera formée des sommets obtenus durant notre parcours. Il nous faut également garder trace des sommets déjà ajoutés à notre file pour ne pas tourner en rond !

Entrée : un graphe, donné par une liste d'adjacence LA (liste ou dictionnaire), et un sommet s
 Sortie : une liste L des sommets parcourus (dans l'ordre)
 On initialise une file d'attente q, une liste L (sommets parcourus) et une autre vus
 On ajoute s à q et à vus
 Tant que q n'est pas vide :
 On retire un élément s_1 de q (on peut réutiliser s si on veut) qu'on ajoute à L
 Pour chaque sommet s_2 voisin de s_1 , s'il n'est pas déjà dans vus:
 on ajoute s_2 à q et à vus.

Question 1. Implémenter l'algorithme présenté ci-dessus dans une fonction d'entête `def BFS(LA, s)`. Bien sûr, on testera avec les deux graphes G_1 et G_2 .

Utiliser la liste L1 ou le dictionnaire D1 pour coder le graphe G_1 change-t-il ici quelque chose ?

3 Parcours en profondeur : version itérative

Pour parcourir un graphe G en profondeur depuis un sommet donné, on tâche de se déplacer depuis le sommet initial, en suivant les arêtes du graphe. Ce n'est qu'en parvenant à un cul-de-sac que l'on revient alors sur ses pas, jusqu'à trouver une arête non encore explorée, et on continue. Pour le graphe G_1 , depuis le sommet 0, un parcours en profondeur pourrait être : (0, 1, 6, 2, 3, 4, 7, 8).

Question 2. Pour le graphe G_1 , (0, 1, 6, 2, 3, 4, 7, 8) n'est pas le seul parcours en profondeur valable depuis le sommet 0. Donnez au moins un autre parcours possible. Combien de parcours en profondeur existe-t-il ?

Pour le graphe G_2 , si un parcours en profondeur commence par (0, 1, 4, 7, 6, 3), comment termine-t-il ?

Pour un parcours en profondeur, l'idée est de remplacer la file d'attente utilisée pour le parcours en largeur par une pile (où le premier élément extrait est le dernier ajouté, et non le premier comme pour les files d'attente) mais quelques autres petits changements sont à faire, car si on recopie à l'identique l'algorithme précédent en remplaçant la file d'attente par une pile, le parcours de G_2 depuis le noeud 0 conduit à (0, 3, 6, 7, 8, 5, 2, 4, 1) qui n'est pas un parcours en profondeur. (L'explication est que le sommet 1 est ajouté dès la première itération à notre pile comme voisin du noeud 0 et qu'étant alors marqué comme vu, le parcours en profondeur croit être parvenu à un cul-de-sac.) La solution consiste à ne marquer comme vus que les sommets qui font effectivement partie de notre parcours (ceux qui ont été extraits de notre pile). L'écueil est qu'un même sommet pourra alors être ajouté plusieurs fois à notre pile (un nombre de fois au plus égal à son degré) et c'est en dépilant un sommet de la pile qu'il faut s'assurer qu'il n'a pas encore été visité. L'algorithme devient donc :

Entrée : un graphe, donné par une liste d'adjacence LA, et un sommet s (son indice)
 Sortie : une liste L des sommets parcourus (dans l'ordre)
 On initialise une pile p, une liste L (sommets parcourus) et une autre vus
 On ajoute s à p
 Tant que p n'est pas vide :
 On dépile un élément s_1 de p
 S'il n'est pas déjà dans vus:
 On l'ajoute à L et à vus
 Pour chaque sommet s_2 voisin de s_1 :
 on l'empile dans p.

Remarque 1. Si les objets L et vus sont comme proposés ci-dessus définis sous la même forme de listes en python, alors ils sont à tout moment identiques et il n'est pas utile de définir les deux, et on pourra donc n'introduire que l'un des deux dans la fonction qui suit. Il reste intéressant d'écrire cet algorithme avec ces deux variables dans la mesure où leur rôle diffère un peu : L contient le parcours de notre graphe et l'ordre de ses termes est important, tandis que dans vus, l'ordre n'a pas d'importance. On verra plus tard qu'une autre structure qu'une liste pour vus peut permettre d'améliorer la complexité de l'algorithme précédent (et justifie donc d'introduire à la fois L et vus).

Question 3. Implémenter l'algorithme présenté ci-dessus dans une fonction d'entête `def DFS(LA, s)`. Tester et comparer avec BFS (sur les graphes G_1 et G_2 , par exemple `DFS(L1, 0)` et `BFS(L1, 0)`)

4 Parcours en profondeur : version récursive

Une implémentation élégante du parcours en profondeur est d'utiliser une fonction récursive. Parvenu à un sommet donné s , l'idée est de traverser chacune des arêtes issues de s (et que la liste d'adjacence nous donne) et pour ce faire, à chaque sommet voisin de s , notre fonction de parcours va s'invoquer elle-même. Le parcours depuis le sommet s va être la concaténation des parcours depuis chacun des voisins de s (du moins ceux que l'on va effectivement visiter, autrement dit ceux qui n'ont pas déjà été marqués comme visités). On rappelle que concaténer deux listes L1 et L2 est aussi simple que : $L1 + L2$.

Bien entendu, il ne faut pas passer deux fois sur un même sommet, ici encore il va falloir garder trace des sommets parcourus. Pour ce faire, on va introduire une variable non locale et qui formera plutôt un argument de notre fonction récursive et qui permettra de stocker ceux-ci. Ici encore, on utilisera pour simplifier une liste (initialement vide) qui se remplit des indices de sommets visités (voir en fin de sujet pour une amélioration possible).

Question 4. Compléter la fonction suivante pour parcourir en profondeur un graphe connu par sa liste d'adjacence depuis un sommet donné :

```
def DFS2(LA, s):
    vus = [] # vus est une variable locale à DFS2, mais pas à DFSrec qui suit :
    def DFSrec(LA, s, vus):
        parcours = [s]
        ...# on rajoute le sommet s à notre liste de sommets vus
        ...# pour tous les sommets voisins de s, on les visite (appel récursif)
        ...# s'ils ne l'ont pas encore été...
        return parcours

    return DFSrec(LA, s, vus)
```

Bien sûr, on testera avec G_1 et G_2 . Comparer par exemple $\text{DFS}(D1, 0)$ et $\text{DFS2}(D1, 0)$. Etait-ce prévisible ?

5 Construction d'un (gros) graphe

On va construire un graphe dont les sommets sont formés des 5757 mots de 5 lettres de la langue anglaise, qu'on trouvera dans le fichier 'sgb-words.txt' que vous avez téléchargé. Reste à rajouter les arêtes, et on va construire un graphe non orienté où deux mots sont reliés dès lors qu'une seule lettre les distingue. Vous noterez déjà que le script comporte une fonction du nom de `chargeMots` et que, pourvu que vous ayez enregistré (et exécuté) ce script dans un même dossier que le fichier 'sgb-words.txt', alors en exécutant `mots = chargeMots('sgb-words.txt')` vous obtiendrez une liste `mots` formée de ces 5757 mots.

Question 5. Ecrire une fonction d'entête `def sontVoisins(mot1, mot2)` qui renvoie `True` si les deux mots qu'on pourra supposer de 5 lettres `mot1` et `mot2` diffèrent d'exactly une lettre, et `False` sinon.

Question 6. Ecrire alors une fonction d'entête `def creeGraphe(listeMots)` qui prend en argument une liste de mots `listeMots` et qui renvoie une liste d'adjacence `D` telle que `D[mot]` est la liste des mots de `listeMots` qui sont voisins de `listeMots[i]`. Par exemple :

```
>>> creeGraphe(['tract', 'trace', 'troop', 'brace', 'grace'])
{'tract': ['trace'],
 'trace': ['tract', 'brace', 'grace'],
 'troop': [],
 'brace': ['trace', 'grace'],
 'grace': ['trace', 'brace']}
```

De ce qui précède, on peut alors créer notre gros graphe (du moins ses arêtes) en exécutant :

```
>>> mots = chargeMots('sgb-words.txt')
>>> LA = creeGraphe(mots)
```

Ne soyez pas surpris : une vingtaine de secondes seront peut-être nécessaires pour exécuter cette dernière instruction, ce qui n'est pas surprenant quand on considère qu'il y a $\frac{5757 \times 5756}{2} = 16568646$ couples de mots à comparer.

Tester votre graphe avec :

```
>>> BFS(LA, 'would')
['would', 'could', 'world', ..., 'court'] (14 mots)
```

Et noter que $\text{DFS}(\text{LA}, \text{'would'})$ conduit à une liste constituée des mêmes 14 éléments, mais dans un ordre différent.

Remarque 2. On peut noter que le graphe construit n'est pas orienté, ni **connexe**, puisque le parcours du graphe depuis donc le sommet 'would' ne comporte que 14 sommets, et non les 5757 que comporte notre graphe.

6 Recherche de chemins

Les algorithmes de parcours étudiés permettent déjà de répondre à la question suivante : étant donnés deux sommets **s1** et **s2**, existe-t-il un chemin de **s1** vers **s2** ? En effet, il suffit de parcourir notre graphe à partir du sommet **s1**, et la réponse sera positive pourvu que **s2** fasse partie du parcours obtenu, et négative sinon puisque tous les sommets atteignables depuis **s1** l'ont été, dans un ordre ou un autre. Maintenant, on pourrait souhaiter connaître un chemin effectif conduisant du sommet **s1** au sommet **s2**. Il n'y a pas grand chose à modifier dans nos fonctions pour parvenir à cela, il suffira, au fur et à mesure de notre parcours, de garder trace pour chaque sommet ajouté à notre sommet, de l'indice du sommet d'où l'on venait. (Attention, ce n'est pas toujours le sommet juste précédent dans la liste obtenue dans le parcours effectué, il n'y a qu'à voir les exemples de parcours donnés pour les graphes G_1 et G_2).

On va reprendre les fonctions **BFS** et **DFS** en rajoutant un dictionnaire **predecesseur**. Initialement, le dictionnaire aura pour clés les sommets du graphe, et à chaque'un d'entre eux, on associera la valeur **None** puis, au fur et à mesure de notre parcours, pour chaque nouveau sommet vu à notre parcours, on mettra à jour **predecesseur** de sorte qu'on puisse associer à tout sommet parcouru son sommet père et ainsi remonter le trajet depuis le sommet atteint jusqu'au sommet initial. (N.B.: dans la version itérative du parcours en profondeur, un même sommet peut être ajouté plusieurs fois à notre pile, et on mettra donc pour ce sommet plusieurs fois à jour la liste **predecesseur**)

Par exemple, dans le parcours en largeur de G_2 depuis le sommet 0, à savoir (0, 1, 3, 2, 4, 6, 5, 7, 8), le dictionnaire **predecesseur** vaudra, à la fin du parcours : {0: **None**, 1: 0, 2: 1, 3: 0, 4: 1, 5: 2, 6: 3, 7: 4, 8: 5} car les sommets 1 et 3 auront été ajoutés depuis le sommet 0, les sommets 2 et 4 depuis le sommet 1, le sommet 6 depuis le sommet 3, 5 depuis le sommet 2, 7 depuis le sommet 4 et 8 depuis le sommet 5. Il devient facile de construire alors un chemin depuis le sommet 0 vers n'importe quel autre sommet. Par exemple pour un chemin depuis le sommet 0 vers le sommet 8, on part de la fin : le sommet 8, et on remonte : son prédécesseur est le sommet 5, puis celui de 5 est 2, puis celui de 2 est 1 et celui de 1 est 0. 0 n'ayant pas lui-même de prédécesseur, on a fini de reconstruire notre chemin, lequel est donc $0 \rightarrow 1 \rightarrow 2 \rightarrow 5 \rightarrow 8$ qu'on présentera sous la forme de la liste [0, 1, 2, 5, 8].

Question 7. A l'aide des fonctions **BFS** et **DFS**, écrire des fonctions d'entête **def cheminBFS(LA, s1, s2)** et **def cheminDFS(LA, s1, s2)** qui déterminent un chemin, s'il existe, de **s1** à **s2**. En l'absence d'un tel chemin, on renverra **None**.

Ici, la nature de l'objet **LA** (liste ou dictionnaire) a son importance, et il est pénible d'écrire un code qui s'adapte aux deux situations, donc on considèrera désormais que la liste d'adjacence est donnée sous la forme d'un dictionnaire.

Si **LA** est le graphe obtenu dans la partie précédente (mots de 5 lettres), tester avec **cheminBFS(LA, 'other', 'thing')** et **cheminDFS(LA, 'other', 'thing')**. (On obtient ['other', 'otter', ..., 'thins', 'thing'] (12 termes))

Commentaire ?

7 Amélioration(s)

Question 8. Il est un point un peu coûteux en termes de complexité dans nos implémentations de parcours de graphes : c'est le test d'appartenance d'un élément **s** à la liste **vus**. En effet, réaliser le test **if s in vus** a un coût proportionnel en la longueur de **vus**. On s'en doute en effet, la liste **vus** n'étant pas supposée triée, tester l'appartenance d'un objet à celle-ci, s'il n'en fait pas partie, oblige à parcourir l'intégralité de la liste **vus**.

Proposer une modification pour que le fait de tester si un sommet a été visité se fasse en temps constant. (On considèrera ici que le graphe est codé sous la forme d'un dictionnaire)

Autre amélioration possible : en cherchant un chemin d'un sommet **s1** à un sommet **s2**, dès lors que le sommet **s2** a été atteint en parcourant notre graphe depuis le sommet **s1**, alors il n'est pas nécessaire de poursuivre le parcours. On ne cherche pas ici en effet tous les chemins conduisant à **s2**, mais juste l'un d'eux.

8 Annexe : rappels de syntaxe python

8.1 Listes

- longueur d'une liste : `len(L)`, on rappelle que les éléments de `L` ont pour indices les entiers 0 à `len(L)-1`.
- Concaténer deux listes : si `L1` et `L2` sont deux listes, l'opération `L1 + L2` renvoie une nouvelle liste, formée des éléments de `L1` suivis de ceux de `L2`. De ce fait, l'instruction `L1 = L1 + L2` remplace la liste `L1` par une nouvelle liste.

A noter que l'instruction `L1 += L2` est un peu différente, puisqu'ici la liste `L1` n'est pas détruite pour être remplacée par une nouvelle liste, mais elle est modifiée, étendue donc, en y ajoutant, à la fin, les éléments de `L2`.

8.2 Dictionnaires

- Créer un dictionnaire vide : `d = dict()`, ou bien `d = {}`.
- Créer un dictionnaire avec déjà quelques associations (clés, valeurs) : `d = {'clé': [], 0: 'valeur'}`
- Ajouter une association (clé, valeur) à un dictionnaire : `d[nouvelle_clé] = valeur`
- Lire la valeur associée à une clé : `d[clé]` (laquelle donne la valeur associée, mais attention, si `clé` ne figure pas parmi les clés du dictionnaire, ceci conduit à une erreur de type `KeyError`)
- Vérifier la présence d'une clé : `if clé in d:` (contrairement aux listes, le test `clé in d` opère en temps constant, quelle que soit la taille du dictionnaire)
- Créer un dictionnaire avec une liste de clés `L`, en association une même valeur pour chaque clé :

```
d = {}
for i in L:
    d[i] = None # ou toute autre valeur bien sûr
# ou bien
d = {i: None for i in L} # version courte du code précédent
```

- Itérer sur les clés d'un dictionnaire : `for clé in d:` ou, ce qui revient au même : `for clé in d.keys():`

8.3 Piles, files d'attente, deque

On rappelle le type `deque` de python, qu'on importe par `from collections import deque`

On rappelle qu'avec un objet de type `deque`, on peut, de manière plus efficace qu'avec les listes : ajouter et retirer des éléments aussi bien à gauche qu'à droite, ce qui permet d'implémenter de manière efficace une file d'attente où les nouveaux éléments sont ajoutés d'un côté, et retirés de l'autre (peu importe bien sûr si on décide d'ajouter à droite et de retirer à gauche, ou de faire le contraire). On rappelle ainsi que, si `q` a été défini par `q = deque()` alors

- `q.append(obj)` rajoute `obj` à l'extrémité droite de `q`
- `q.appendleft(obj)` rajoute `obj` à l'extrémité gauche de `q`
- `q.pop()` retire l'objet situé à l'extrémité droite de `q` et renvoie celui-ci. Une erreur est déclenchée si `q` est vide.
- `q.popleft()` retire l'objet à l'extrémité gauche de `q` et renvoie celui-ci. Une erreur est déclenchée si `q` est vide.
- `len(q)` renvoie le nombre d'éléments contenus dans `q`.

Une pile peut quant à elle être implémentée, de manière équivalente, ou bien par une liste, avec les méthodes `append` et `pop`, ou bien par un objet `deque` (en prenant soin ou bien d'ajouter à droite et retirer à droite, ou bien d'ajouter à gauche et retirer à gauche)