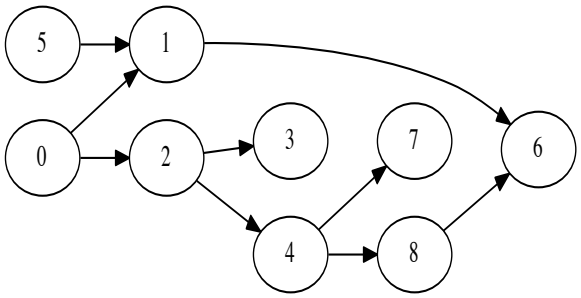
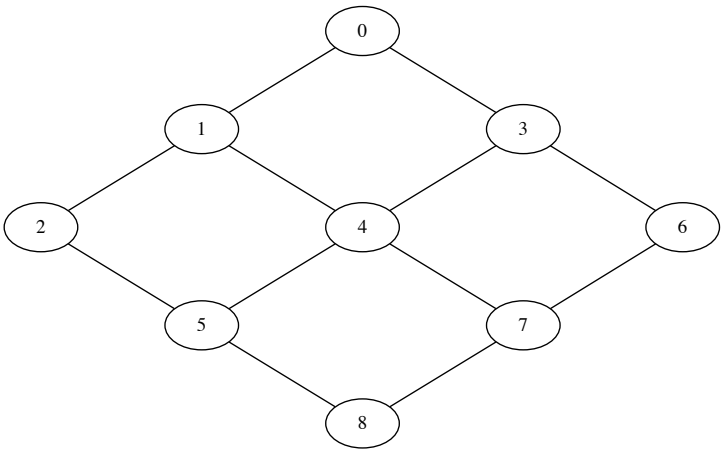
TP - Parcours de graphes (2/2)

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=118` le fichier `sgb-words.txt` et le script `tpParcours.py` (qui contient déjà les fonctions de parcours en largeur et profondeur abordées dans le TP précédent) et on enregistrera ces deux fichiers dans un même dossier.

Les deux graphes vus en exemple dans le TP précédent se trouvent encore, par leurs listes d'adjacence (présentées comme des listes de listes ou des dictionnaires) sous la forme des objets `L1, D1, L2, D2`, ce qui permettra de tester nos algorithmes :



G_1 : un graphe orienté



G_2 : un graphe non-orienté

1 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 1. Ecrire une fonction d'entête `def sontVoisins(mot1, mot2)` qui renvoie `True` si les deux mots qu'on pourra supposer (il n'est donc pas demandé se s'en assurer) de 5 lettres `mot1` et `mot2` diffèrent d'exactement d'une lettre, et `False` sinon.

Question 2. 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'] (17 mots)
```

Et noter que `DFS(LA, 'would')` conduit à une liste constituée des mêmes 17 éléments, mais dans un ordre différent.

Remarque 1. 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 17 sommets, et non les 5757 que comporte notre graphe.

2 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`. On pourra initialiser notre dictionnaire avec une seule clé et valeur associée : `{s1 : None}` puis, au fur et à mesure de notre parcours, pour chaque nouveau sommet vu à notre parcours, on ajoutera un couple `(s : parent)` à notre dictionnaire `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.

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 dicionnaire `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 $0 \to 1 \to 2 \to 5 \to 8$ qu'on présentera sous la forme de la liste $[0, 1, 2, 5, 8]$.

Question 3. 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`.

Si `LA` est le graphe obtenu dans la partie précédente (mots de 5 lettres), tester avec `cheminBFS(LA, 'other', 'thing')` (On devrait obtenir une liste de 12 termes : `['other', 'otter', ..., 'thins', thing']`) puis tester avec `cheminDFS(LA, 'other', 'thing')`.

Commentaires ?

3 Amélioration(s)

Question 4. 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.

Mettre en œuvre les améliorations proposées, et tester vos fonctions.

4 Annexe : rappels de syntaxe python

4.1 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`.

4.2 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():`

4.3 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(onj)` 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)