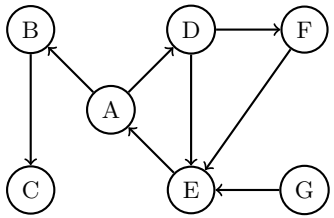
TP Python 10 - Graphes

Exercice 1 On considère le graphe suivant :



1) Représenter ce graphe en Python par un dictionnaire de listes d'adjacence (les sommets sont des caractères : `'A'`, `'B'`, etc.).

2) Écrire sa matrice d'adjacence (la numérotation des sommets doit correspondre à l'ordre alphabétique).

3) a) Parcourir à la main le graphe en profondeur en partant de A, puis en partant de G.

b) Même question avec un parcours en largeur.

Exercice 2

Les exemples correspondent à ceux du cours. On pourra aussi tester les fonctions sur le graphe de l'exercice 1 dont les sommets sont des caractères.

1) Dans cette question un graphe est représenté par le dictionnaire de ses listes d'adjacence.

a) Écrire une fonction `degre` qui, recevant un graphe non orienté et un de ses sommets, renvoie son degré.

```
>>> G = {1: [2], 2: [1, 4, 5], 3: [4], 4: [2, 3, 5, 6], 5: [2, 4], 6: [4, 6]}
>>> degre(G, 2)
3
```

b) Écrire deux fonctions `degre_sortant` et `degre_entrant` qui, recevant un graphe orienté et un de ses sommets, renvoient respectivement son degré sortant et son degré entrant.

```
>>> G = {1: [3], 2: [4], 3: [5], 4: [], 5: [1]}
>>> degre_sortant(G, 2)
1
>>> degre_entrant(G, 2)
0
```

c) Écrire une fonction `est_chemin` qui, recevant un graphe et une liste de sommets, renvoie un booléen indiquant si cette liste définit un chemin ou non.

```
>>> G = {1: [2], 2: [1, 4, 5], 3: [4], 4: [2, 3, 5, 6], 5: [2, 4], 6: [4, 6]}
>>> est_chemin(G, [1, 2, 4, 3])
True
>>> est_chemin(G, [1, 2, 5, 6])
False
```

2) Reprendre les questions précédentes lorsque les graphes sont représentés par leurs matrices d'adjacence. On supposera que les sommets sont numérotés à partir de 1.

```
>>> M = [[0, 1, 0, 0, 0, 0],
         [1, 0, 0, 1, 1, 0],
         [0, 0, 0, 1, 0, 0],
         [0, 1, 1, 0, 1, 1],
         [0, 1, 0, 1, 0, 0],
         [0, 0, 0, 1, 0, 1]]    # premier graphe du cours
>>> degre(M, 2)
3
>>> est_chemin(M, [1, 2, 4, 3])
True
>>> est_chemin(M, [1, 2, 5, 6])
False
```

3) a) Écrire une fonction `poids` qui, recevant un graphe pondéré représenté par un dictionnaire de dictionnaires d'adjacence et un chemin, renvoie le poids de celui-ci, s'il est valide, et `-1` sinon.

```
>>> G = {1: {2: 7, 3: 10}, 2: {1: 7, 3: 6, 4: 5, 5: 3}, 3: {1: 10, 2: 6, 5: 4},
         4: {2: 5}, 5: {2: 3, 3: 4}}  # troisième graphe du cours
>>> poids(G, [1, 2, 3, 5])
17
>>> poids(G, [1, 2, 3, 4])
-1
```

b) Même question lorsque le graphe est donné par sa matrice d'adjacence. On supposera que les sommets sont numérotés à partir de 1.

```
>>> M = [[0, 7, 10, 0, 0],
         [7, 0, 6, 5, 3],
         [10, 6, 0, 0, 4],
         [0, 5, 0, 0, 0],
         [0, 3, 4, 0, 0]]   # troisième graphe du cours
>>> poids(M, [1, 2, 3, 5])
17
>>> poids(M, [1, 2, 3, 4])
-1
```

Exercice 3

Dans cet exercice on supposera que les sommets sont numérotés à partir de 1.

1) Écrire une fonction `matrice_adjacence` qui, recevant le dictionnaire des listes d'adjacence d'un graphe non pondéré, renvoie sa matrice d'adjacence.

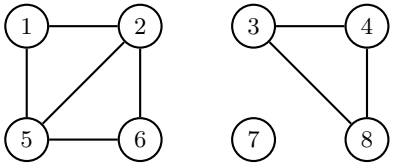
```
>>> G = {1: [2], 2: [1, 4, 5], 3: [4], 4: [2, 3, 5, 6], 5: [2, 4], 6: [4, 6]}
>>> matrice_adjacence(G)
[[0, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0], [0, 0, 0, 1, 0, 0], [0, 1, 1, 0, 1, 1],
 [0, 1, 0, 1, 0, 0], [0, 0, 0, 1, 0, 1]]
```

2) Écrire une fonction `listes_adjacence` qui, recevant la matrice d'adjacence d'un graphe non pondéré, renvoie le dictionnaire de ses listes d'adjacence.

```
>>> M = [[0, 1, 0, 0, 0, 0],
         [1, 0, 0, 1, 1, 0],
         [0, 0, 0, 1, 0, 0],
         [0, 1, 1, 0, 1, 1],
         [0, 1, 0, 1, 0, 0],
         [0, 0, 0, 1, 0, 1]]
>>> listes_adjacence(M)
{1: [2], 2: [1, 4, 5], 3: [4], 4: [2, 3, 5, 6], 5: [2, 4], 6: [4, 6]}
```

Exercice 4

La **composante connexe** d'un sommet S d'un graphe non orienté G est l'ensemble des sommets reliés à S par un chemin de G. Le graphe ci-dessous a trois composantes connexes.



1) Écrire une fonction `composante_connexe` qui, recevant le dictionnaire des listes d'adjacence d'un graphe non orienté et un de ses sommets, renvoie la composante connexe de celui-ci.

```
>>> G = {1: [2, 5], 2: [1, 5, 6], 3: [4, 8], 4: [3, 8], 5: [1, 2, 6],
         6: [2, 5], 7: [], 8: [3, 4]}
>>> composante_connexe(G, 1)
[1, 2, 5, 6]
```

2) Écrire une fonction `est_connexe` qui, recevant le dictionnaire des listes d'adjacence d'un graphe non orienté, renvoie un booléen indiquant s'il est connexe ou non.

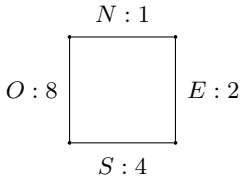
```
>>> est_connexe(G)
False
```

3) Écrire une fonction `composantes_connexes` qui, recevant le dictionnaire des listes d'adjacence d'un graphe non orienté, renvoie la liste de ses composantes connexes.

```
>>> composantes_connexes(G)
[[1, 2, 5, 6], [3, 4, 8], [7]]
```

Exercice 5 - Labyrinthe

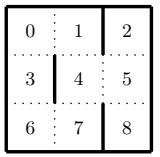
Dans cet exercice on définit un labyrinthe comme une liste d'entiers compris entre 0 et 15. Chaque entier représente une case. L'entier associé à une case est la somme des valeurs de ses murs données par le schéma suivant :



Par exemple, la case ⌜ est représentée par le nombre $1 + 8 = 9$. On peut ainsi tester facilement la présence d'un mur en utilisant l'opérateur `&` (ET binaire). Par exemple, l'écriture binaire de 9 est 1001 donc :

```
>>> 9 & 1     # 1001 ET 0001 = 0001
1
>>> 9 & 2     # 1001 ET 0010 = 0000
0
>>> 9 & 4     # 1001 ET 0100 = 0000
0
>>> 9 & 8     # 1001 ET 1000 = 1000
8
```

Le labyrinthe (qui sera toujours carré) est donc défini par la liste des entiers associés aux cases en commençant par celle en haut à gauche. Par exemple, `[9, 3, 11, 10, 8, 2, 12, 6, 14]` représente le labyrinthe suivant :



1) Donner le début de la liste associée au labyrinthe de la page suivante (la liste complète est dans le dossier de la classe).

On peut associer à un labyrinthe un graphe non orienté dont les sommets sont les cases et tel qu'il existe une arête entre deux cases s'il n'y a pas de mur entre elles.

2) a) Parcourir en profondeur puis en largeur (à la main) le labyrinthe 3×3 précédent en partant de la case 0.

b) Appliquer (à la main) l'algorithme de Dijkstra à ce labyrinthe pour trouver le plus court chemin entre les cases 0 et 8.

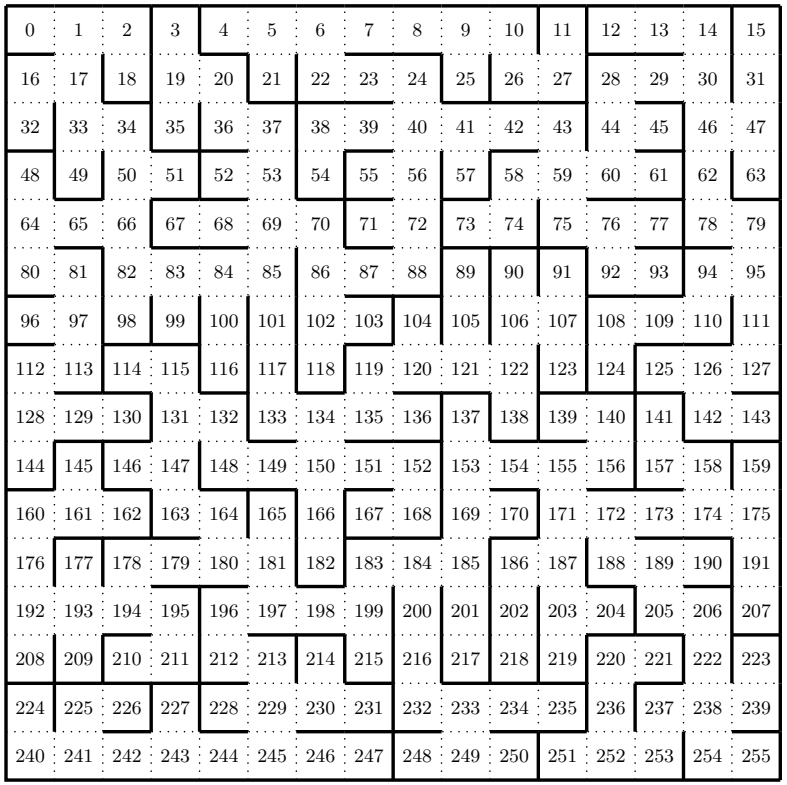
3) Écrire une fonction `graphe` qui, recevant la liste représentant un labyrinthe, renvoie le graphe associé sous forme du dictionnaire de ses listes d'adjacence.

```
>>> graphe([9, 3, 11, 10, 8, 2, 12, 6, 14])
{0: [1, 3], 1: [4, 0], 2: [5], 3: [0, 6], 4: [1, 5, 7], 5: [2, 8, 4],
 6: [3, 7], 7: [4, 6], 8: [5]}
```

4) Vérifier les résultats de la question 2 en utilisant les fonctions du cours (en les adaptant éventuellement).

```
>>> G = graphe([9, 3, 11, 10, 8, 2, 12, 6, 14])
>>> DFS(G, 0, set())
0 1 4 5 2 8 7 6 3
>>> BFS(G, 0)
0 1 3 4 6 5 7 2 8
>>> plus_court_chemin(G, 0, 8)
[0, 1, 4, 5, 8]
```

5) Vérifier que le labyrinthe 16×16 ci-dessous est connexe et trouver (avec l'ordinateur) un chemin reliant les cases 0 et 255.



Exercice 6 - Promenade dans le dictionnaire - Un gros graphe

Dans cet exercice on utilisera le fichier `mots.txt` du TP 8. On ne travaillera qu'avec des mots de cinq lettres.

1) Créer la liste des mots de cinq lettres du fichier.

2) Si cela n'a pas été fait au TP 8, écrire une fonction `voisins` qui, recevant une chaîne de caractères `mot`, renvoie la liste des mots qui diffèrent de `mot` d'une et une seule lettre.

```
>>> voisins('liste')
['ciste', 'leste', 'lifte', 'lisse', 'lista', 'piste']
```

On peut ainsi définir un graphe dont les sommets sont les mots et où deux sommets sont reliés par une arête s'ils sont voisins au sens de la fonction précédente.

3) Ce graphe est gros et déterminer les voisins d'un mot prend du temps. On va créer son dictionnaire des listes d'adjacence une fois pour toutes et le stocker dans un fichier texte pour pouvoir le récupérer ensuite sans avoir à le recréer.

a) Écrire une fonction `creer_graphe` (sans paramètre) qui renvoie le dictionnaire des listes d'adjacence du graphe. Ne surtout pas afficher le résultat (on le stockera dans une variable `G`).

```
>>> G = creer_graphe() # Cela peut prendre plusieurs minutes.
>>> G['liste']
['ciste', 'leste', 'lifte', 'lisse', 'lista', 'piste']
```

Attention à ne pas exécuter à nouveau le fichier sinon `G` sera effacé et il faudra le recréer.

b) Créer un fichier texte `graphe_mots.txt`, l'ouvrir en mode écriture et y stocker `G` (utiliser `str` pour convertir `G` en chaîne de caractères). La taille du fichier devrait être 380 kilo-octets.

c) Écrire une fonction `recuperer_graphe` (sans paramètre) qui ouvre le fichier `graphe_mots.txt` et renvoie le dictionnaire des listes d'adjacence du graphe. On peut utiliser `eval` pour reconvertir la chaîne de caractère en dictionnaire.

```
>>> G = recuperer_graphe()
>>> G['liste']
['ciste', 'leste', 'lifte', 'lisse', 'lista', 'piste']
```

4) a) Combien le graphe a-t-il de sommets ? Combien d'arêtes ? En moyenne, combien chaque sommet a-t-il de voisins ?

b) Combien de mots n'ont pas de voisins ?

c) Combien le graphe a-t-il de composantes connexes ? Quelle est la taille de sa plus grande composante connexe ?

5) Écrire une fonction `chemin` qui, recevant deux mots de cinq lettres, renvoie le plus court chemin qui les relie, s'il existe, et affiche un message sinon.

```
>>> chemin('maths', 'gauss')
['maths', 'matas', 'hatas', 'haras', 'harts', 'hauts', 'sauts', 'saurs',
 'gaurs', 'gauss']
>>> chemin('alpha', 'omega')
Pas de chemin
```

Exercice 7 - Promenade sur Wikipedia - Un très gros graphe

La version anglaise de l'encyclopédie en ligne Wikipedia contient plus de six millions d'articles. Ces articles contiennent des liens vers d'autres articles. On peut donc considérer le graphe orienté dont les sommets sont les articles et les arcs les liens entre eux. On se propose de programmer un parcours en profondeur dans ce graphe.

Vu la taille du graphe (plus de six millions de sommets), il est hors de question de construire le dictionnaire de ses listes d'adjacences et encore moins sa matrice d'adjacence. On va construire les listes d'adjacence des sommets au fur et à mesure du parcours.

1) On va d'abord définir une fonction qui va **parser** une page Wikipedia, i.e. en effectuer une analyse syntaxique, et renvoyer la liste de ses liens vers d'autres articles. On travaille avec la version anglaise pour éviter les problèmes d'encodage liés aux accents.

Si on observe le code-source d'une page quelconque, par exemple

`https://en.wikipedia.org/wiki/Depth-first_search`

(clic droit sur la page pour afficher le code-source), on constate que les liens vers d'autres articles correspondent toujours à des chaînes du type :

`<a href="/wiki/Algorithm" title="Algorithm">`.

On éliminera les chaînes qui contiennent `':'` et celles qui contiennent `'disambiguation'` car elles correspondent à des pages spéciales de Wikipedia.

a) Écrire une fonction `extraire` qui, recevant une chaîne de caractères de la forme précédente, renvoie le nom de l'article correspondant (situé après `wiki/`).

```
>>> extraire('<a href="/wiki/Algorithm" title="Algorithm">')
Algorithm
```

b) Avec le module `urllib.request` (qu'on importe de la manière habituelle) on peut récupérer le contenu html d'une page web. Ainsi les instructions suivantes :

```
with urllib.request.urlopen(lien) as response:
    html = response.read().decode('utf-8')
```

où `lien` est l'adresse internet de la page désirée, stockent dans la variable `html` le code-source de cette page.

Écrire une fonction `code_source` qui, recevant le titre d'un article de Wikipedia, renvoie le code-source de la page web correspondante.

```
>>> code = code_source('Depth-first_search')
>>> code[:20]
'<!DOCTYPE html>\n<htm'
```

c) Il reste à parser le code-source : on le parcourt pour récupérer ses sous-chaînes de la forme précédente, on en extrait les titres des articles et on les met dans une liste.

Écrire une fonction `parser` qui, recevant le titre d'un article, renvoie la liste des titres des articles liés.

```
>>> parser('Depth-first_search')
['Search_algorithm', 'Graph_(data_structure)', 'Best,_worst_and_average_case',
 'Time_complexity', ...]
```

2) Adapter la fonction `DFS` du cours afin que, recevant le titre d'un article, elle affiche les titres des articles visités par un parcours en profondeur.

```
>>> DFS('Depth-first_search', set())
Search_algorithm
Hash_table
Hash_list
Computer_science
...
```