TP 20
Parcours de graphes

Dans la continuité du TP précédent, nous nous intéressons aux questions de parcours de graphes. **Parcourir** un graphe, c'est écrire une fonction qui passe successivement d'un sommet à l'autre, dans un certain ordre, éventuellement pour y effectuer certaines opérations. On peut penser à une ville qu'on veut visiter entièrement, en passant d'un point à l'autre via les rues ; ou bien à un labyrinthe dont il faut sortir. Cela permet d'espérer répondre aux questions suivantes :

- Partant d'un sommet x, peut-on toujours atteindre un autre sommet y ? Par exemple, peut-on trouver un chemin pour sortir d'un labyrinthe ?
- Peut-on déterminer un chemin le plus court possible entre deux sommets ?

Cependant il y a plusieurs stratégies possibles, qui sont plusieurs ordres de parcours… Dans chacune de ces stratégies il y a une notion de sommets « déjà vus » (ceux sur lesquels nous sommes déjà passés, et nous n'avons pas particulièrement besoin d'y revenir) et de sommets « à voir » (ceux qui sont reliés aux sommets vus mais sur lesquels nous ne sommes pas encore passés).

On pourra utiliser comme exemple le graphe orienté ci-dessous, donné comme un dictionnaire d'adjacence :

```
G_exemple = {'A': ['B', 'D'], 'B': ['C'], 'C': ['E', 'F'], 'D': ['E'], 'E': ['B'], 'F': [],
'G': ['C']}
```

Exercice 1
Dessiner ce graphe sur feuille et le garder sous les yeux pour les deux premières parties du TP.

I Parcours en profondeur

Le parcours en profondeur correspond à explorer une ville où, dès qu'on trouve une rue nouvelle, on y va ! Jusqu'au bout tant qu'il y a encore à explorer. On ne revient en arrière que si on se trouve à un point où, tout autour, on a déjà tout vu.

Cela s'écrit simplement en Python avec une fonction récursive. Partant d'un somme initial x, on regarde les sommets vers lesquels on peut accéder (c'est facile si le graphe est donné par un dictionnaire d'adjacence), et on saute (en appelant la fonction récursivement) sur le premier venu… parmi ceux qui n'ont pas déjà été vus ! La fonction s'appelle donc `parcours_profondeur(G, x)`, prenant comme argument un graphe `G` donné comme dictionnaire d'adjacence, et un sommet `x` de `G`, et elle manipule une liste `vus` de sommets déjà vus qui est définie *en dehors de la fonction* — c'est cela qui est un peu subtil à cause de la récursivité. Au départ cette liste est vide ; à la fin de l'exécution, elle contient la liste des sommets sur lesquels la fonction est passée, et dans l'ordre.

On rappelle à ce propos les manipulations suivantes de listes :
- `vus.append(x)` : ajoute l'élément `x` à la fin de la liste `vus`.
- `if x in vus` : teste si l'élément `x` est dans la liste `vus`.
- `if x not in vus` : teste si l'élément `x` n'est pas dans la liste `vus`.

L'algorithme se décrit ainsi :
- Partant d'un sommet `x`, on le marque comme vu.
- Puis on regarde la liste des sommets auxquels on peut accéder depuis `x`.
  - Si on en trouve un, **et** qu'il n'a pas déjà été vu, on y va avec un appel récursif.
  - Sinon, la fonction s'arrête.

Sur notre graphe d'exemple, on a la séquence suivante, partant de `"A"`, où `x` est le sommet sur lequel on se trouve :
1. Parcours à partir de A : `vus = ["A"]`, `x = "A"`
2. Découvre B, parcours à partir de B : `vus = ["A", "B"]`, `x = "B"`
3. Découvre C, parcours à partir de C : `vus = ["A", "B", "C"]`, `x = "C"`
4. Découvre E, parcours à partir de E : `vus = ["A", "B", "C", "E"]`, `x = "E"`
5. Retour à parcours à partir de C : `vus = ["A", "B", "C", "E"]`, `x = "C"`
6. Découvre F, parcours à partir de F : `vus = ["A", "B", "C", "E", "F"]`, `x = "F"`

7. Retour à parcours à partir de C : `vus = ["A", "B", "C", "E", "F"]`, `x = "C"`
8. Retour à parcours à partir de B : `vus = ["A", "B", "C", "E", "F"]`, `x = "B"`
9. Retour à parcours à partir de A : `vus = ["A", "B", "C", "E", "F"]`, `x = "A"`
10. Découvre D, parcours à partir de D : `vus = ["A", "B", "C", "E", "F", "D"]`, `x ="D"`
11. Retour à parcours à partir de A : `vus = ["A", "B", "C", "E", "F", "D"]`, `x ="A"`
12. Fin

En fait, pour pouvoir renvoyer la liste `vus` définie en dehors de la fonction, on utilisera plutôt une *sous-fonction récursive* (la fonction `parcours_profondeur` contient la liste `vus`, et à l'intérieur, la fonction `aux` est récursive).

Exercice 2
Écrire la fonction `parcours_profondeur(G, e)`, (on nomme `e` le sommet de départ) et tester avec le graphe d'exemple et plusieurs sommets de départ différents.

Par construction, un sommet `y` est accessible depuis `x` s'il apparait dans la liste `vus` en effectuant le parcours depuis `x`. On peut aussi arrêter la fonction plus tôt, dès qu'on trouve `y`.

Exercice 3
En déduire une fonction `existe_chemin(G, x, y)`, qui renvoie `True` s'il existe un chemin, suivant le sens des arêtes, du sommet `x` vers le sommet `y`, et `False` sinon.

II Parcours en largeur

Dans le parcours en largeur, on imagine qu'on visite une ville mais en restant à chaque fois le plus proche possible de son point de départ, tant qu'il reste des rues à explorer. Chaque fois qu'en prenant une rue on découvre un nouvel endroit, on ne va pas, comme dans le parcours en profondeur, y sauter tout de suite, mais on va le mettre en « liste d'attente », et on y reviendra quand on aura terminé d'explorer ce qui est déjà en attente. Ainsi on visite des zones de plus en plus larges autour de son point de départ.

La fonction ne s'écrit pas de façon récursive. Elle manipule une liste `L` et un indice `i` dans `L` tel que les sommets d'indice avant `i` ont déjà été vus, et ceux d'indice après `i` sont ceux à voir en liste d'attente. L'algorithme se décrit ainsi :

- Partant d'un sommet `x`, au départ `L = [x]` tout seul et `i = 0`.
- On lit la liste des sommets accessibles depuis `x`, et on les ajoute à `L`. Ce sont les sommets « à voir » qui sont en liste d'attente.
- Si on a bien ajouté des sommets à l'étape précédente (on est à `i = 0` et `L` est de longueur au moins 2) :
  - Alors on passe à `i = 1`, on considère qu'on est sur le sommet `L[1]`.
  - On recommence, en ajoutant à `L` tous les sommets accessibles depuis `L[1]`, ce sont les « nouveaux sommets découverts » qu'on met en file d'attente.
- Si `i` arrive au bout de la liste, et qu'il n'y a rien après, on s'arrête : on n'a plus rien en liste d'attente.

Sur notre graphe d'exemple, on a la séquence suivante en partant de `"A"` :

1. Début en A : `L = ["A"]`, `i = 0`
2. Découvre B et D : `L = ["A", "B", "D"]`, `i = 0`
3. Passe à B : `L = ["A", "B", "D"]`, `i = 1`
4. Découvre C : `L = ["A", "B", "D", "C"]`, `i = 1`
5. Passe à D : `L = ["A", "B", "D", "C"]`, `i = 2`
6. Découvre E : `L = ["A", "B", "D", "C", "E"]`, `i = 2`
7. Passe à C : `L = ["A", "B", "D", "C", "E"]`, `i = 3`
8. Découvre F : `L = ["A", "B", "D", "C", "E", "F"]`, `i = 3`
9. Passe à E : `L = ["A", "B", "D", "C", "E", "F"]`, `i = 4`
10. Passe à F : `L = ["A", "B", "D", "C", "E", "F"]`, `i = 5`
11. Fin

Exercice 4
Écrire la fonction `parcours_largeur(G, e)`, qui prend en argument un graphe `G` représenté par un dictionnaire d'adjacence, et un sommet de départ `e`, et renvoie la liste des sommets vus ; et tester avec plusieurs sommets de départs.

Un avantage du parcours en largeur, c'est que les sommets sont automatiquement vus dans l'ordre en fonction de leur distance au point de départ (les zones visitées forment des cercles de plus en plus larges autour du départ). Seul le sommet `x` est à distance 0 de lui-même ; ensuite, seuls les nouveaux sommets découvert depuis `x` sont à distance exactement 1 de `x`, etc. Pour calculer les distances, il est nécessaire de calculer, en parallèle (et aussi avec des `append`) de la liste `L`, une liste `D` où `D[i]` est la distance à `x` du sommet `L[i]`. Alors quand on découvre depuis `L[i]` des sommets, on les ajoute à `L`, et on ajoute en même temps leur distance à `D`, leur distance est `D[i] + 1`. Éventuellement les sommets qui ne sont pas accessibles depuis `x` n'ont pas de distance bien définie à `x` (la valeur mathématiquement cohérente est $+\infty$).

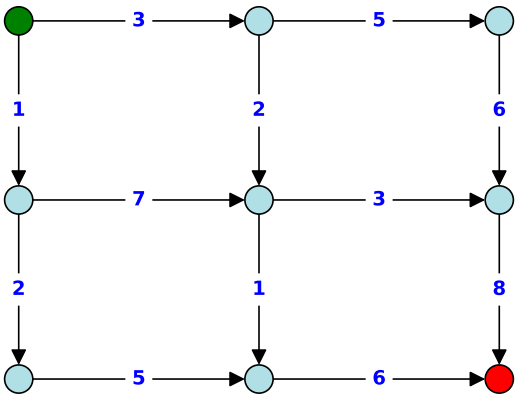
Exercice 5
En déduire une fonction `distance(G, x, y)` qui renvoie la distance du sommet `x` au sommet `y`, et `None` s'il n'y a pas de chemin de `x` à `y`.

III Plus court chemin : un modèle simplifié

Le très célèbre **algorithme de Dijkstra** permet de déterminer la plus courte distance entre deux sommets dans un graphe orienté pondéré, où chaque arête porte un nombre indiquant sa longueur. On peut y penser très concrètement comme une carte d'une ville, indiquant des rues et leur temps de parcours, et il faut aller d'un point à l'autre le plus rapidement possible.

Le problème est en général compliqué, dû au fait que pour choisir globalement le chemin le plus court, on ne peut pas forcément choisir *à chaque étape* l'arête la plus courte qui nous fait avancer vers notre but…

On se propose d'étudier une situation simplifiée dans laquelle notre graphe a la forme d'un quadrillage : on part d'en haut à gauche, pour aller en bas à droite, et à chaque arête on peut aller soit en bas soit à droite mais on ne peut jamais revenir en arrière (c'est bel et bien un graphe orienté, dont les sommets sont les croisements d'une grille quadrillée). Sur les arêtes sont indiquées des valeurs, qu'on suppose être des nombres réels strictement positifs. On peut y penser comme à une ville américaine, bien quadrillée, mais avec des embouteillages, et les valeurs indiquent le temps de parcours de chaque rue. Voici un exemple de tel graphe :



Exercice 6
Pouvez-vous trouver le plus court chemin dans ce graphe, entre le sommet de départ (en vert) et celui d'arrivée (en rouge) ?

Observez que le plus court chemin n'est pas obtenu en choisissant à chaque intersection l'arête la plus courte…

On représentera un tel graphe en Python par la donnée de *deux* tableaux : `V` est le tableau de toutes les arêtes verticales, et `H` est le tableau de toutes les arêtes horizontales. Ainsi du sommet (i, j) on peut aller au

sommet $(i+1, j)$ avec la distance `V[i][j]`, ou bien au sommet $(i, j+1)$ avec la distance `H[i][j]`. Pour l'exemple précédent les tableaux sont :

```
V = [[1, 2, 6], [2, 1, 8]]
H = [[3, 5], [7, 3], [5, 6]]
```

Dans le cas général, notre quadrillage a n lignes et p colonnes, ainsi sur une verticale on a n arêtes et $n+1$ sommets, et sur une horizontale on a p arêtes et $p+1$ sommets. Le sommet en haut à gauche est $(0, 0)$, celui en bas à droite est (n, p). Le tableau `V` est de taille $(n, p+1)$ et `H` est de taille $(n+1, p)$.

L'idée est de calculer un tableau complet `T` de taille $(n+1, p+1)$ où `T[i][j]` indique la distance la plus courte pour aller de tout en haut à gauche jusqu'au sommet (i, j). Ce tableau contient *plus* d'information que notre seul but (qui est le coefficient `T[n][p]`) mais se calcule naturellement pas à pas. Pour notre exemple, on peut vérifier que le tableau est

$$\texttt{T} = \begin{bmatrix} 0 & 3 & 8 \\ 1 & 5 & 8 \\ 3 & 6 & 12 \end{bmatrix}$$

et donc la distance la plus courte jusqu'en bas à droite est 12.

Exercice 7

1. Justifier que le tableau `T[i][j]` vérifie les relations suivantes, permettant de le calculer entièrement pas à pas :
   - (i) `T[0][0]` $= 0$,
   - (ii) $\forall 1 \leqslant i \leqslant n$, `T[i][0]` $=$ `T[i-1][0]` $+$ `V[i-1][0]`,
   - (iii) $\forall 1 \leqslant j \leqslant p$, `T[0][j]` $=$ `T[0][j-1]` $+$ `H[0][j-1]`,
   - (iv) $\forall 1 \leqslant i \leqslant n$, $\forall 1 \leqslant j \leqslant p$, `T[i][j]` $=$ Min(`T[i-1][j]` $+$ `V[i-1][j]`, `T[i][j-1]` $+$ `H[i][j-1]`).
2. En déduire une fonction `distances_minimales(V, H)` qui prend en argument les deux tableaux `V` et `H` décrivant entièrement notre graphe, et qui renvoie le tableau de toutes les distances minimales.

À retenir

Pour calculer *le* plus court chemin, l'algorithme consiste à calculer les longueurs de *tous* les plus courts chemins en même temps. On calcule plus de choses que demandé pour le résultat final, mais la méthode est plus efficace.

La méthode précédente donne la plus courte distance, mais ne montre pas par *quel* chemin on y accède. Une façon naturelle de s'y prendre — quoique pas la plus économe — consiste à calculer, en parallèle du tableau `T`, un tableau `C` de chemins (un chemin est représenté comme une liste de couples (i, j), indiquant les sommets par lesquels il passe, donc `T` est une… liste de liste de listes de couples) où `C[i][j]` est un chemin le plus court pour aller d'en haut à gauche jusqu'à (i, j). Au départ `C` est un tableau de valeurs `None` et `C[0][0]` est la liste `[(0, 0)]`, puis `C` se remplit au fur et à mesure qu'on remplit `T` en ajoutant (avec l'opération `+` sur les listes, par exemple `C[i][j] = C[...][...] + [(i, j)]`) le sommet par lequel on passe.

Exercice 8

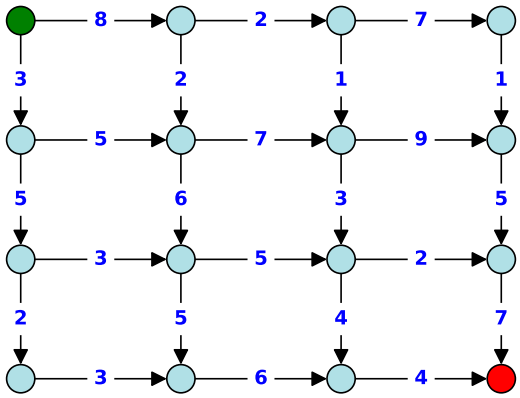
Écrire la fonction `plus_court_chemin(V, H)` qui renvoie le chemin le plus court, d'en haut à gauche jusqu'à en bas à droite.

Une autre façon naturelle pour trouver le plus court chemin est d'utiliser le tableau `T` et de remonter en partant du coefficient en bas à droite, en déduisant *d'où* provenait le chemin le plus court. On obtiendra alors le chemin rangé en ordre inverse. Plus précisément, on initialise des variables `i` et `j` représentant le coefficient sur lequel on se trouve, et un chemin `C`, en démarrant du bas à droite :

- Si on est est remonté jusqu'en haut à gauche : c'est fini.
- Si on se trouve sur la première colonne : le chemin le plus court provenait nécessairement du haut.
- Si on se trouve sur la première ligne : le chemin le plus court provenait nécessairement de la gauche.
- Sinon :
  - Si `T[i][j-1]` $+$ `H[i][j-1]` $<$ `T[i-1][j]` $+$ `V[i-1][j]` : le chemin le plus court provenait de le gauche.

- Sinon : il provenait d'en haut.
- À chaque étape on utilise `C.append((i, j))` pour ajouter au chemin la case par laquelle on passe.

Exercice 9
Écrire la fonction `plus_court_chemin_inverse(V, H)` qui renvoie le plus court chemin, en ordre inverse, obtenu en partant du coefficient en bas à droite.

Vous avez tout compris ?

Exercice 10
Calculer le tableau des distances minimales, puis trouver le plus court chemin, pour l'exemple suivant :