Graphes et plus courts chemins

1. Être familiarisé avec le vocabulaire des graphes : graphe orienté, non orienté ; sommet, arc, arête ; degré (entrant, sortant) ; chemin d'un sommet à un autre. Cycle. Connexité dans les graphes non orientés.
2. Listes d'adjacence (dans une liste ou un dictionnaire) et matrices d'adjacence.
3. Pondération d'un graphe.
4. (Découverte). Parcours d'un graphe : en profondeur, en largeur. Piles et Files.
5. (Découverte). Recherche d'un plus court chemin dans un graphe pondéré avec des poids positifs. Algorithme de Dijkstra.

---

1 Vocabulaire

- Un *graphe* est décrit par un ensemble S de sommets et un ensemble A d'arêtes reliant ces sommets.
- Un graphe est *non orienté* si ses arêtes ne sont pas orientées. Les arêtes sont des ensembles $\{x, y\}$ avec x et y dans S.
- Un graphe est *orienté* si ses arêtes sont orientées. Les arêtes orientées, ou *arcs*, sont des couples (x, y) avec x et y dans S.

graphe orienté	**graphe non orienté**
arc (s, s') : couple	arête $\{s, s'\}$: ensemble
boucle : arête (s, s)	
s' est un successeur de s lorsque (s, s') est un arc s' est un prédécesseur de s lorsque (s', s) est un arc	s' est voisin de s lorsque $\{s, s'\}$ est une arête
$d_+(s)$, degré sortant, nombre d'arcs partant de s $d_-(s)$, degré entrant, nombre d'arcs arrivant en s $d(s) = d_+(s) + d_-(s)$ degré du sommet s	$d(s)$ degré de s, nombre d'arêtes contenant le sommet s
chemin $(a_1, \ldots, a_p)$: p-uplet d'arcs avec $(a_i, a_{i+1}) \in A$	chemin $(a_1, \ldots, a_p)$: p-uplet d'arêtes tels que $\{a_i, a_{i+1}\} \in A$

- Un cycle est un chemin dont les sommets de départ et d'arrivée sont égaux.
- Un graphe non orienté est *connexe* quand deux sommets quelconques sont reliés par un chemin.
- Un graphe est *pondéré* lorsqu'un poids est associé à chacune de ses arêtes.

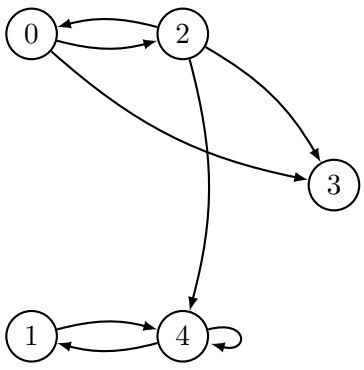
2 Représentation des graphes

On numérote les sommets de 0 à $n-1$ lorsqu'on veut utiliser des listes. On peut aussi utiliser des dictionnaires si on veut conserver des noms de sommets avec des caractères.

2.1 matrice d'adjacence

La matrice d'adjacence A d'un graphe est une matrice remplie de 0 et 1, avec $a_{i,j} = 1$ lorsqu'il y a un arc/une arête entre les sommets i et j.
En Python, on enregistre une matrice par une liste de listes et on accède à $a_{i,j}$ par `A[i][j]` en débutant i et j à 0.

- La matrice est symétrique si le graphe est non orienté.
- À la place de mettre 1, on peut placer le poids de l'arc/arête pour un graphe pondéré
- Avantages :
  - on sait rapidement s'il y a un arc/une arête entre 2 sommets
  - on gère facilement les poids
  - on peut obtenir les voisins d'un sommet, on parcourt la ligne correspondante.
  - le nombre de chemins de longueur k partant de i et arrivant en j est égal à $m_{i,j}$, où $A^k = (m_{i,j})_{0 \leqslant i,j \leqslant n-1}$ (démonstration par récurrence).
- Inconvénients :
  - La matrice d'adjacence peut occuper beaucoup de place en mémoire.
  - Pour obtenir tous les voisins d'un sommet, on parcourt toute la ligne correspondante : complexité $\Theta(n)$.



$$A = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

et pour un graphe pondéré, par exemple :

$$P = \begin{pmatrix} 0 & 0 & 3 & 1/2 & 0 \\ 0 & 0 & 0 & 0 & 10 \\ 2 & 0 & 0 & 7 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1/3 & 0 & 0 & 1 \end{pmatrix}$$

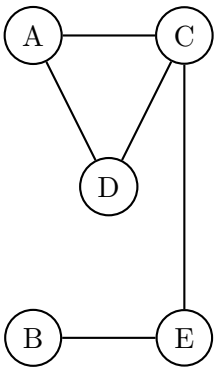
2.2 liste d'adjacence

Dans un graphe non orienté, la liste d'adjacence d'un sommet est la liste de ses voisins. Dans un graphe orienté, la liste d'adjacence d'un sommet est la liste de ses successeurs. La *liste d'adjacence d'un graphe* est la liste (ou le dictionnaire) constituée de la liste d'adjacence de ses sommets.

- Avantages :
  - on a directement la liste des sommets successeurs (ce qui est utile, par exemple, pour tous les algorithmes de parcours),

— il est plus ou moins facile de savoir si un arc/une arête existe.

- Inconvénients : difficile d'avoir les sommets prédécesseurs d'un sommet dans le cas d'un graphe orienté.



Liste d'adjacence (dictionnaire ici) :

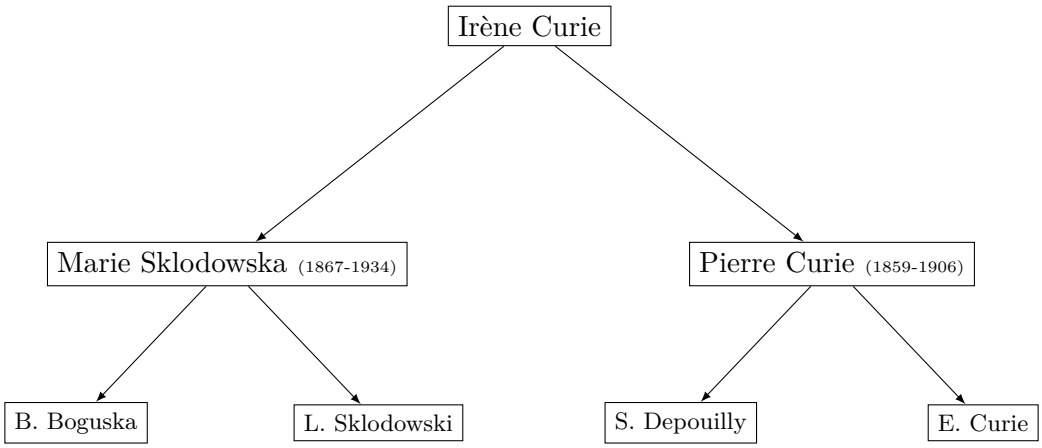
```
L    =  {
'A'   :  ['C','D']
'B'   :  ['E']
'C'   :  ['A','E','D']
'D'   :  ['A','C']
'E'   :  ['B','C']
}
```

ou (liste ici)

```
[['C','D'], ['E'], ['A','E','D'], ['A','C'],
['B','C']]
```

3 Parcours d'un graphe en profondeur

Si on adopte l'image d'un arbre généalogique (le descendant est ici mis en tête et les ancêtres en-dessous), un parcours en profondeur est celui où on parcourt toute une branche généalogique avant de passer à la suivante :



3.1 la notion de Pile : LIFO

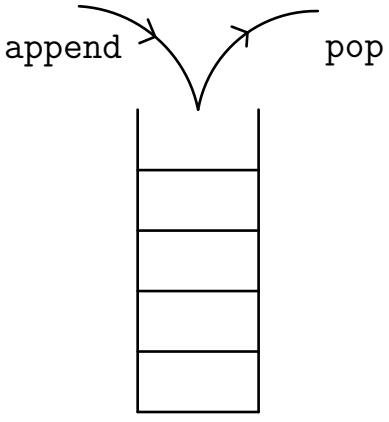
Imaginez des assiettes sur une étagère, des habits dans un sac de linge sale, des livres sur un bureau... Étant donné une pile d'assiettes, vous pouvez :

— vous demander si la pile d'assiettes est vide,

— **prendre l'assiette en haut de la pile** (dépiler),

— **déposer une assiette en haut de la pile**.

La manière dont les assiettes sont empilées assure que la première assiette que vous pouvez prendre est la dernière qui a été déposée. Cette structure est appelée LIFO pour *Last In First Out*.



Opérations sur les piles

En Python, les piles sont implémentées par des listes. On peut :

- créer une pile vide : `P = []`
- ajouter (empiler) un élément `a` : `P.append(a)`
- enlever et retourner un élément (dépiler) : `x = P.pop()`
- tester si une pile est vide : `if P == []`

Toutes ces opérations se font en temps constant (indépendant de la longueur de la liste, complexité $\Theta(1)$).

3.2 algorithme de parcours d'un graphe en profondeur

parcours en profondeur

L'algorithme de parcours en profondeur d'un graphe consiste, partant d'un sommet de départ, à visiter les sommets du graphe en favorisant la génération de sommets suivante par rapport à la génération actuelle.

1. Créer une pile comportant le sommet de départ.
2. Tant que la pile n'est pas vide :
   - dépiler le sommet
   - ajouter à la pile les voisins de ce sommet qui n'ont pas encore été rencontrés.

   On maintient pour cela une liste `sommets_deja_mis_en_attente` qui enregistre les sommets qui ont été mis à un moment en Pile d'attente.

Le programme suivant parcourt en profondeur le graphe G (donné par sa liste d'adjacence) depuis le sommet `depart`.

```
def parcours_profondeur(G, depart):
    '''
    G: liste d'adjacence (liste de listes d'entiers)
    depart: entier sommet de départ
    renvoie le parcours en profondeur des sommets (liste d'entiers)
    '''
    # on maintient une liste de sommets deja mis en attente dans la Pile

    parcours = []
    pileAtraiter = [depart]
    sommets_deja_mis_en_attente = [depart]
    while ..........:
        sommet = ...................
        parcours.append(sommet)
        voisins = [v for v in G[sommet] if v not in sommets_deja_mis_en_attente]
        for s in voisins:
            # rajout de ces sommets dans la Pile
            ................................
            ................................
    return parcours
G = [[1,2,3],[0,4],[0,4,5],[0],[1,2,5],[2,4,6],[5,7],[6]]
parcours_profondeur(G,0)
```

4 Parcours en largeur

Si on reprend l'analogie de l'arbre généalogique, un parcours en largeur consiste à parcourir l'arbre génération par génération.

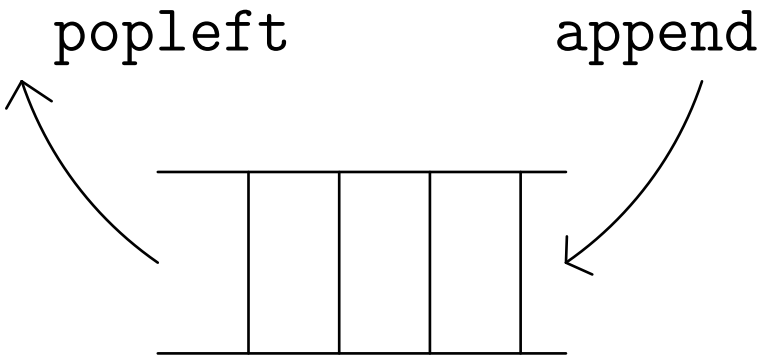
4.1 la notion de File : FIFO

Imaginez que vous faites la queue à un guichet. Étant donné une file d'attente, on peut :

— se demander si la file est vide,

— traiter la demande du client en tête de file,

— placer un client à la queue de la file pour qu'il attende son tour.



La manière dont les clients sont disposés dans la file assure que le premier client qui verra sa demande traitée est le premier arrivé. Cette structure est appelée FIFO pour *First In First Out*.
Si on utilisait une liste pour une file, le fait de traiter une demande (supprimer le premier élément) demanderait de copier et décaler tous les autres, ce qui aurait une complexité linéaire en la taille de la liste. C'est pourquoi nous utiliserons le module `deque` (prononcer dèque en français ou deqwou en anglais) dans lequel les implémentations ont été prévues pour que les opérations se fassent en temps constant.

Opérations sur les files

En Python, avec :

```
from collections import deque
```

les files sont des structures de données mutables sur lesquelles on peut :

- créer une file vide : `F = deque([])`
- ajouter un élément `a` : `F.append(a)`
- enlever le premier élément : `x = F.popleft()`
- tester si une file est vide : `if len(F)==0`

Toutes ces opérations se font en temps constant (indépendant de la longueur de la liste, complexité $\Theta(1)$).

Remarque : `deque` est l'abréviation de *double-ended queue*.

4.2 algorithme de parcours d'un graphe en largeur

> Pour effectuer un parcours en largeur du graphe, on change simplement la pile de l'algorithme de parcours en profondeur en file. Conséquence : les nouveaux voisins sont traités après les anciens, et les sommets d'une même génération sont privilégiés par rapport à ceux des générations suivantes.

Voici l'adaptation du parcours en profondeur déjà vu pour transformer la pile utilisée en file.

```
from collections import deque
def parcours_largeur(G, depart):
    '''
    G: liste d'adjacence
    depart: sommet de départ
    renvoie le parcours en largeur des sommets
    '''
    # on maintient une liste de sommets deja mis en attente dans la File

    parcours = []
    fileAtraiter = deque([depart])
    sommets_deja_mis_en_attente = [depart]
    while .....:
        sommet = .....
        parcours.append(sommet)
        voisins = [v for v in G[sommet] if v not in sommets_deja_mis_en_attente]
        for s in voisins:
            # rajout de ces sommets dans la File
            .....
            .....
    return parcours
G = [[1,2,3],[0,4],[0,4,5],[0],[1,2,5],[2,4,6],[5,7],[6]]
parcours_largeur(G,0)
```

Visualisation : On considère un graphe non orienté de liste d'adjacence

$$G = [[2,3,4],[2,3,4],[0,1],[0,1],[0,1]]$$

Déterminer, sans ordinateur mais en faisant un schéma des piles ou files,

1. le parcours en profondeur du graphe partant de 0,
2. le parcours en largeur du graphe partant de 0,

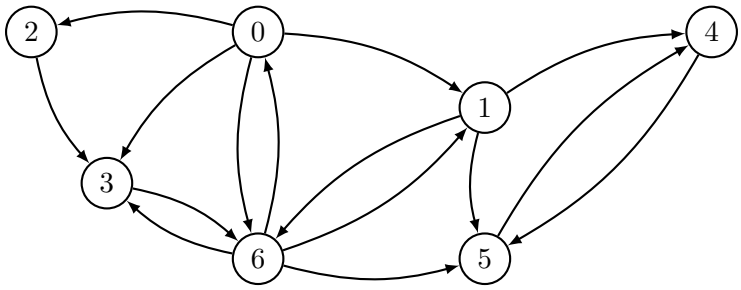
et vérifier vos réponses à l'aide des fonctions `parcours_profondeur` et `parcours_largeur`.

5 Plus court chemin dans un graphe non pondéré

5.1 distance minimale de d à a

> — On part du sommet d. On va parcourir le graphe en largeur.
> — On visite d'abord les voisins/successeurs de d (1^{re} génération, distance 1).
> — On visite ensuite les voisins de ces voisins (2^{e} génération, distance 2).
> — Etc. jusqu'à arriver à a.
> — Il faut aussi prévoir qu'on peut ne pas atteindre a, auquel cas on renvoie -1.

Exemple : On considère le graphe suivant :



1. Avec le graphe, donner la distance de 0 à 5 et la distance de 4 à 6.
2. On modifie la fonction `parcours_largeur` pour renvoyer la distance de `d` à `a` avec les indications suivantes :
   — on maintient une liste `statut` qui nous sert à savoir si un sommet a déjà été atteint, ou a déjà été mis en attente, ou n'a pas été vu du tout,
   — on rajoute une liste `distance` de longueur le nombre de sommets,
   — quand on visite un sommet `s`, on affecte à chacun de ses voisins/successeurs pas encore mis en attente la distance `distance[s]+1`.

```
from collections import deque
def distance_min(G, depart, a):
    '''
    G liste d'adjacence
    depart sommet de départ, a sommet d'arrivée
    renvoie la distance minimale de d à a
    et -1 si d ne mène pas à a
    '''
    # Chaque sommet se voit attribuer 3 statuts
    # 0 : non vu / 1 : en cours sur le chemin / 2 : définitivement vu
    statut = [0 for k in range(len(G))]
    distance = [0 for k in range(len(G))]
    file = deque([depart])
    while len(file)!=0 and statut[a]!=2:
        sommet = file.popleft()
        statut[sommet] = .........
        voisins = [v for v in G[sommet] if statut[v] == ........]
        for s in voisins:
            # actualisation de distance
            distance[s] = ..........................
            # rajout de ces sommets dans la File
            statut[s] = ............................
            file.append(s)
    if statut[a]!=2:
        return ......................
    return ..........................
```

3. Pour terminer, on souhaite obtenir un plus court chemin de d à a. Pour cela, on reprend la fonction précédente en la modifiant pour qu'elle enregistre dans une liste le prédécesseur de la génération $n-1$ (« Père ») de chaque sommet de la génération n (« Fils »).
   La nouvelle fonction est `def chemin_min(G:[[int]],d:int,a:int)->[int]:`
   Pour cette fonction :
   — Deux lignes sont à ajouter : l'initialisation de `Pere` et la ligne où on indique le prédécesseur des sommets qui vont être rajoutés dans la file.
   — La remontée de l'arrivée vers le départ à l'aide de la liste `Pere` en fin de fonction, qui peut suivre la trame suivante :

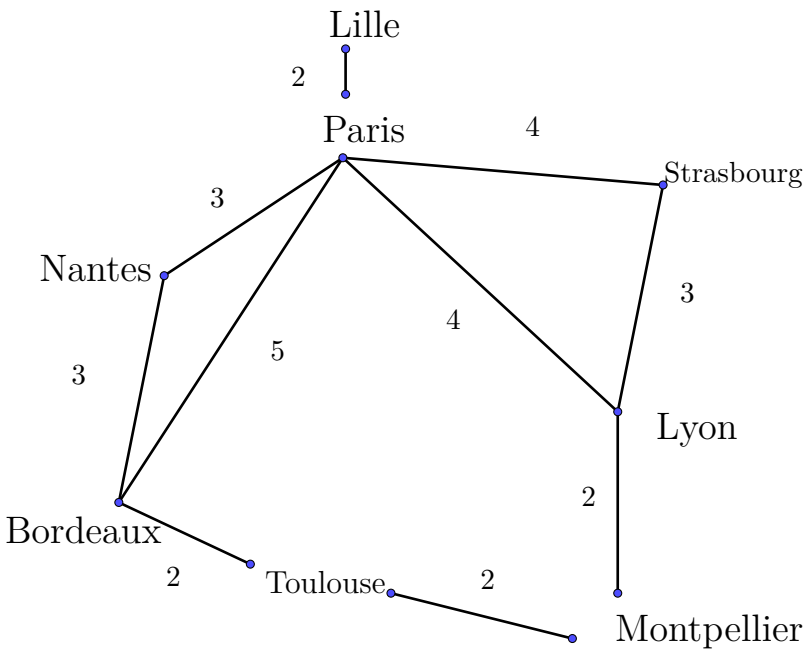
```python
from collections import deque
def chemin_min(G, d ,a):
    '''
    G liste d'adjacence
    depart sommet de départ, a sommet d'arrivée
    renvoie UN plus court chemin  de d à a
    et -1 si d ne mène pas à a
    '''
    # Chaque sommet se voit attribuer 3 statuts
    # 0 : non vu / 1 : en cours sur le chemin / 2 : définitivement vu
    # Pere[i] est un prédecesseur (génération n-1) de i, sommet fils de la génération n
    statut = [0 for k in range(len(G))]
    distance = [0 for k in range(len(G))]
    Pere = [None for k in range(len(G))]
    file = deque([d])
    while len(file)!=0 and statut[a]!=2:
        sommet = file.popleft()
        statut[sommet] = ......
        voisins = [v for v in G[sommet] if statut[v] == .....]
        for s in voisins:
            # actualisation de distance
            distance[s] = ...................
            # rajout de ces sommets dans la File
            statut[s] = ..............................
            Pere[s] = ............ # s a pour prédecesseur sommet
            file.append(s)
    if statut[a]!=2:
        return ..............
    else:
        # on remonte de a vers d en utilisant de proche en proche chaque père
        chemin = .......
        sommet = .......
        while Pere[sommet]!= .......:
            chemin = [Pere[sommet]] + chemin #concaténation de listes
            sommet = ...............
        chemin = [d]+chemin
    return chemin
```

6 Plus court chemin dans un graphe pondéré

6.1 exemple et vocabulaire

- Dans un graphe pondéré, on dispose d'une application poids qui à chaque arête associe un réel positif. Le poids d'un chemin est la somme des poids des arêtes de ce chemin. Chercher un plus court chemin, c'est chercher un chemin de plus faible poids.
- Pour la liste d'adjacence d'un graphe orienté, pour chaque sommet s, on donne la liste des couples (t, p) où t est un sommet successeur de s et p le poids de l'arc (s, t).
  Pour la liste d'adjacence d'un graphe non orienté, pour chaque sommet s, on donne la liste des couples (t, p) où t est un voisin de s et p le poids de l'arête $\{s, t\}$.
- Pour la matrice d'adjacence, à la place des 1, on met le poids de l'arc correspondant.

Voici un exemple de villes de France séparées par des autoroutes (kilométrage simplifié). On recherche les distances minimales depuis Paris.

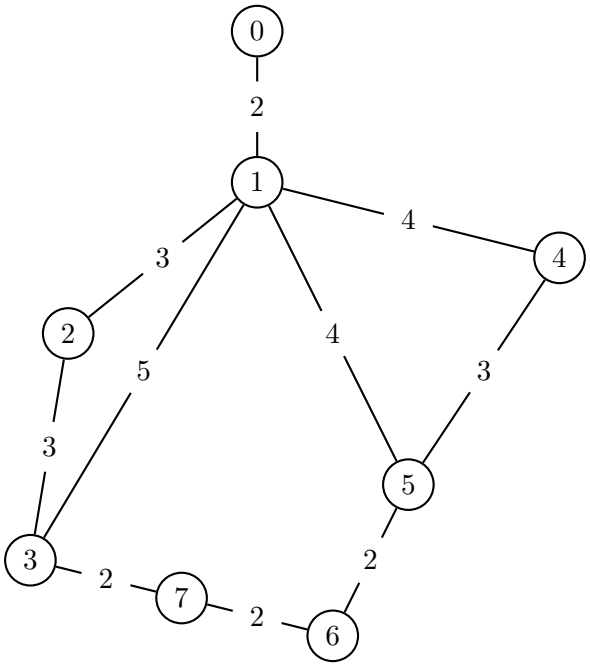


Le graphe est donné par sa liste d'adjacence représentée par un dictionnaire.

6.2 algorithme de Dijkstra

L'algorithme de Dijkstra permet de déterminer la distance minimale (et éventuellement un plus court chemin) entre deux sommets d'un graphe dont les arêtes ont des poids positifs. Son principe repose sur une progression en largeur où on visite en priorité les sommets les plus proches : **on progresse de plus proche en plus proche**. On calcule les distances entre le sommet de départ et chacun des sommets du graphe, ce qui effectue donc beaucoup de calculs « inutiles », puisque tous les points du graphe sont visités.
C'est un algorithme glouton (à chaque étape on choisit la situation la plus favorable) et qui fournit bien un résultat optimal. Il y a plusieurs façons d'implémenter cet algorithme, et il y a plusieurs variantes avec des améliorations.

Pour travailler plus simplement sur des listes, on numérote les villes comme suit :




```python
# Liste d'adjacence
# couples (voisin, poids vers ce voisin)
LG = [
    [(1,2)],
    [(2,3), (0,2), (3,5), (5,4), (4,4)],
    [(1,3), (3,3)],
    [(2,3), (7,2), (1,5)],
    [(1,4), (5,3)],
    [(1,4), (4,3), (6,2)],
    [(7,2), (5,2)],
    [(3,2),  (6,2)]
]
```

On considère un sommet de départ et on se donne les variables :

— `aTraiter` : une liste de couples (s, δ) où les s sont les sommets à traiter et δ le poids d'un chemin menant du départ au sommet s. `aTraiter` est initialisée avec le couple (ville de départ, 0).

— `vu` : une liste contenant les villes vues. `vu` est initialisée vide.

— `R` : une liste qui contiendra les résultats (sommet, distance depuis le départ). `R` est initialisée vide.

Algorithme de Dijkstra

Tant que `aTraiter` n'est pas vide :

✓ On enlève un tuple (s, δ) de `aTraiter` ayant la plus petite distance δ et on récupère s.

✓ Si s n'a pas été vu, :

- on ajoute (s, δ) à `R` et on marque s comme vu,
- pour chaque (t, d_t) voisin pondéré de s, on le rajoute dans la liste `aTraiter` avec la bonne distance minimale, à savoir $d_t + \delta$.

TRAVAIL PERSONNEL POUR CEUX QUI SONT MOTIVÉS PAR CE THÈME (NON FAIT EN CLASSE)

Pour vous familiariser avec l'algorithme, remplir sur feuille annexe le tableau d'exécution suivant. Combien d'étapes y a-t-il en tout ?

	R	`aTraiter`	Li	P	N	Ly	S	B	T	M	`aTraiter`
initialisation	vide										(Paris,0)
étape 1	(Paris, 0)			×							(Li,2), (N,3), (B,5), (Ly,4), (S,4)
⋮											

1. Écrire une fonction annexe qui prend en entrée une liste de couples (ville, valeur) où les valeurs sont entières et qui renvoie un couple (ville, val) pour laquelle `val` est la valeur minimale.
2. Programmer alors l'algorithme de Dijkstra présenté ci-dessus en utilisant la trame suivante :

```
# Algorithme de Dijkstra pour déterminer les distances
# vers tous les sommets depuis un sommet "départ"

def couple_min(L):
    '''
    renvoie un (s,y) pour lequel y est minimal
    '''
    (.................)

depart = 1 # au choix !
R = [] #liste des résultats
# R contiendra à la fin les couples (sommet, distance depuis départ)
aTraiter = [(depart,0)]
vu = [] # liste des sommets marqués

while aTraiter !=[]:
    # parmi les sommets à traiter, on privilégie le plus proche
    (s, delta) = couple_min(aTraiter)
    # on retire ce couple de aTraiter
    aTraiter = [x for x in aTraiter if x!=(s, delta)]
    if s not in vu:
        ..............  # on l'ajoute à R
        ..............  # on marque s comme vu
        # on prépare les voisins de s pas encore vus
```

```
25          voisins_ponderes = [v for v in LG[s] if v[0] not in vu]
26          for v in voisins_ponderes:
27              aTraiter.append((v[0], v[1] + delta))
28  # t est atteint en delta (pour aller jusque s) plus p (poids de s à t)
29
30  print(R)
```

Avec comme point de départ 1 (Paris), on obtient :
`[(1, 0), (0, 2), (2, 3), (5, 4), (4, 4), (3, 5), (6, 6), (7, 7)]`

Remarques :

— L'algorithme peut être modifié pour afficher un plus court chemin d'un sommet à l'autre.

— Dans la recherche d'un itinéraire de réseau routier (avec Waze, Google Maps, Via Michelin...), on ne souhaite pas *toutes* les distances minimales partant de `depart` vers tous les sommets. On peut arrêter l'algorithme dès qu'on a atteint la ville de destination.

6.3 algorithme A^*

L'algorithme de Dijkstra a pour inconvénient, pour un sommet, de partir vers le point le plus proche : ainsi, dans notre exemple, pour aller de Paris à Montpellier, l'exploration a tout de même commencé en direction de Lille. L'algorithme de Dijkstra peut s'éloigner énormément de la trajectoire qui semble la plus courte.
L'algorithme A^* rajoute une pénalisation des sommets du graphe qui n'appartiennent probablement pas au chemin cherché. Il utilise une *fonction heuristique* qui, à un sommet s, associe un coût $f(s)$ (dépendant du but `arrivee` à atteindre). Par exemple, dans le cas d'un itinéraire routier,

— f peut pénaliser une mauvaise direction (Nord si on veut aller de Paris à Montpellier),

— si on cherche à minimiser le temps de trajet, f peut être la distance à vol d'oiseau divisée par la vitesse maximale autorisée (cela privilégie le choix d'une autoroute),

— f peut pénaliser des directions difficiles (zones montagneuses, risque d'embouteillages).

L'algorithme A^* fonctionne comme celui de Dijkstra, hormis le fait qu'on retire de `aTraiter` un sommet s tel que $f(s) + \text{poids}(s)$ est minimal (si f est la fonction nulle, on retrouve l'algorithme de Dijkstra).

Nous rencontrons à plusieurs reprises le mot *heuristique* dans le programme. Nom féminin : partie de la science qui a pour objet les procédures de recherche et de découverte. Adjectif : qui sert à la découverte.