TP13 - Parcours de graphes

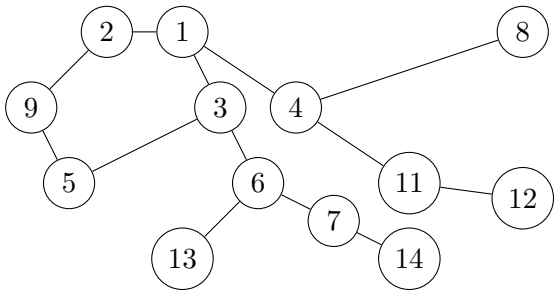
Dans ce TP, on souhaite écrire différents algorithmes de parcours de graphes.
On utilisera un marquage des sommets à l'aide de trois couleurs : blanc, gris et noir.

— blanc : le sommet n'**a pas encore été visité**.

— gris : le sommet est **en cours d'exploitation**.

— noir : le sommet a été **totalement exploité**.

On considère le graphe non orienté connexe suivant :



Q1. Ecrire la variable `g_test` représentant le graphe ci-dessus.
`g_test` est un dictionnaire dont les clés sont les sommets du graphe (de type `int`) et les valeurs les listes d'adjacence (de type `list`).

I Parcours en largeur

Q2. Donner le parcours en largeur de ce graphe à partir du sommet 1, puis à partir du 13.

Pour le parcours en largeur, on utilise une file pour gérer les sommets visités :

- on enfile le sommet de départ. (initialisation)
- on regarde la tête de la file pour obtenir le sommet sélectionné.
- on enfile les sommets que l'on visite.
- on défile la file lorsque le sommet a été totalement exploité, c'est-à-dire lorsque tous ses voisins ont été visités.

Q3. Tester ces quelques lignes pour se familiariser avec les files.

```
from queue import PriorityQueue
f = deque()
f.append(1)
f.append(2)
a = f.popleft()
print(a)
f.append(3)
print(f[0])
print(len(f))
```

Q4. Compléter la fonction `parcours_largeur(g,s)` qui a pour paramètres un graphe `g` de type `dict` et un sommet `s` de `g` et qui renvoie la liste des sommets lorsqu'on parcourt le graphe en largeur.

```
def parcours_largeur(g, s):
    #initialisation
    attente = deque()#la file des sommets en cours d'exploitation
    visite = [s]#le parcours
    attente.append(s)
    marque = {}#on stocke les sommets visités et leur couleur
    marque[s] = 'gris'
    while len(attente) != 0:
        ...

    return visite
```

Tester la fonction avec les exemples précédents.

II Parcours en profondeur

Q5. Donner le parcours en profondeur de ce graphe à partir du sommet 1, puis à partir du 13.

Pour le parcours en profondeur, on utilise une pile pour gérer les sommets visités :

- on empile le sommet de départ. (initialisation)
- on regarde le sommet de la pile pour obtenir le sommet sélectionné.
- on empile les sommets que l'on visite.
- on dépile la pile lorsque le sommet de la pile a été totalement exploité, c'est-à-dire lorsque tous ses voisins ont été visités.

Q6. En utilisant une pile (pouvant être codée par une liste), écrire la fonction `parcours_profondeur(g, s)` qui a pour paramètres un graphe `g` de type `dict` et un sommet `s` de `g` et qui renvoie la liste des sommets lorsqu'on parcourt le graphe en profondeur.
Indication : On pourra écrire la **liste par compréhension** des voisins du sommet sélectionné qui n'on pas été visités.

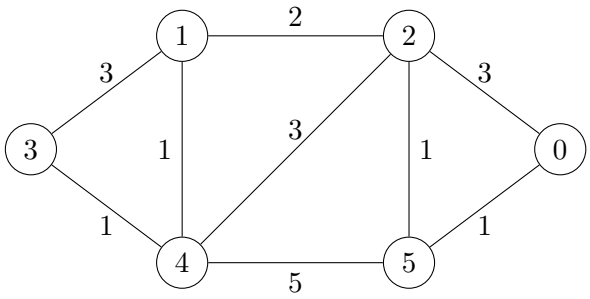
On peut réécrire la fonction précédente mais sans l'utilisation de pile et en utilisant une fonction récursive.
Q7. Comléter la fonction récursive `_parcours_prof(g, s, visite, marque)` qui a pour paramètres un graphe `g` de type `dict`, un sommet `s`, une liste `visite` correspondant au parcours déjà effectué, un dictionnaire `marque` pour stocker la couleur des sommets visités et qui renvoie le parcours en profondeur de `g`.

```
def parcours_p(g, s):
    return _parcours_prof_rec(g, s, [s], {s:'gris'})

def _parcours_prof_rec(g, s, visite , marque):
    for voisin in g[s]:
        if voisin not in marque:
```

III Algorithme de Dijkstra

On considère le graphe connexe pondéré suivant :



Q8. Importer `numpy` et écrire la variable `g_dijkstra` représentant le graphe ci-dessus.
`g_dijkstra` est une matrice contenant les poids du graphe pondéré.
Indication : `np.inf` est de type `float` et permet de représenter $+\infty$...

Dans l'algorithme de Dijkstra, on doit sélectionner à chaque fois le sommet qui a la plus petite distance au sommet de départ.
Pour cela, on utilise une file « prioritaire ».

- A l'initialisation, on stocke dans cette file le couple `(0, s)` où 0 est la distance au sommet de départ et `s` le sommet de départ.
- Tant que la file « prioritaire » n'est pas vide :
  - on défile le couple ayant la plus petite distance pour sélectionner le sommet ;
  - A partir de ce sommet, on met à jour `dist`, `pere` et on stocke dans la file « prioritaire » ces mise-à-jour sous forme de couple `(d, s)`.

Q9. Pour implémenter cette file « prioritaire », tester le code suivant :

```
from queue import PriorityQueue

file_prio = PriorityQueue()
file_prio.put((5, 'turing'))
file_prio.put((0, 'alan'))
file_prio.put((2, 'dijkstra'))
while not file_prio.empty():
     print(file_prio.get())
```

Q10. Effectuer à la main l'algorithme de Dijkstra en partant du sommet 4.

Q11. Compléter la fonction `dijkstra(g, s)` qui a pour paramètres une matrice `g` représentant un graphe pondéré et un sommet `s` et qui renvoie deux dictionnaires :

- `dist` où les clés sont les sommets et les valeurs les distances entre `s` et le sommet (cad la clé) ;
- `pere` où les clés sont les sommets et les valeurs les pères du sommet (cad la clé) ;

```
def dijkstra(g, s):
    #initialisation
    n = len(g)
    dist = {s : np.inf for s in range(n)}
    dist[s] = 0
    pere = {s : None for s in range(n)}
    f = PriorityQueue()
    f.put((0, s))
    #
    while not f.empty():
        ....
    return dist, pere
```

Tester la fonction.

Q12. Ecrire la fonction `plus_court_trajet(g, s1, s2)` qui a pour paramètres une matrice `g` représentant un graphe pondéré et deux sommets `s1` et `s2` et qui renvoie la liste correspondant au plus court chemin pour aller de `s1` à `s2`.
Tester la fonction.