TP (S2) 4 — Parcours de graphes non pondérés & Applications

Objectifs

- S'approprier les algorithmes de parcours de graphe en profondeur et en largeur.
- Modifier les algorithmes de parcours de graphe pour répondre à de demandes spécifiques.

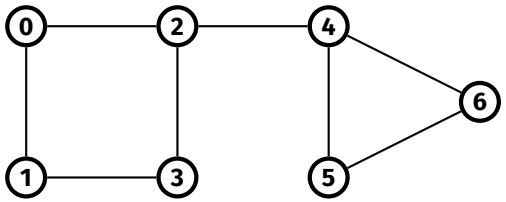
Fichier externe ?

OUI `parcours.py` *(présent(s) dans le répertoire partagé de la classe)*

Les algorithmes de parcours vus en cours sont fournis dans le fichier `parcours.py` se trouvant dans le répertoire partagé de la classe. Ce fichier est à copier sur votre espace personnel pour utilisation.

1. AUTOUR DES DEUX PARCOURS

Dans cette partie, nous allons proposer des versions modifiées des deux parcours vus en cours. Afin de tester les modifications, nous utiliserons comme graphe exemple le graphe G_1 suivant :



Exercice 1 **Parcours de graphes implémentés par matrice d'adjacence** [Sol 1]
Les algorithmes de parcours vus en cours ont été présentés pour des graphes implémentés par des listes d'adjacence, et doivent être légèrement remaniés si on désire les appliquer à des graphes implémentés par matrice d'adjacence. Pour un graphe $G = (S, A)$ à n sommets, on notera `T` le tableau numpy de dimension (n, n) contenant la matrice d'adjacence de G, tel que `T[i,j]` contient `True` si il existe un arc de i vers j, `False` sinon. On notera que dans cette implémentation, les sommets sont repérés par des entiers. Cette modification peut être effectuée pour chacune des fonctions `bfs` et `dfs`, mais on se limite ici à la modification de la fonction `dfs`.

1. Reprendre le code de la fonction `dfs` vue en cours, et préciser à quel(s) endroit(s) la nature de l'implémentation du graphe jour un rôle. Modifier le code fourni et proposer le code d'une fonction `dfs_M(grph:np.array, v:int)->list` qui réalise le parcours en profondeur d'un graphe représenté par sa matrice d'adjacence, à partir d'un sommet v, et qui renvoie la liste des sommets parcourus.

2. Afin de tester cette fonction et de la comparer avec la fonction `dfs`, il est nécessaire de disposer à la fois de la liste d'adjacence (implémentée sous forme de dictionnaire) et la matrice d'adjacence (implémentée sous forme de tableau numpy) associée à un même graphe $G = (S, A)$.

   2.1) On suppose connu le dictionnaire `D` associé à la liste d'adjacence d'un graphe G, pour lequel chaque sommet est désigné par l'entier qui lui correspond. Proposer une fonction `D2M(D:dict)->np.array` qui prend comme argument `D` et qui renvoie le tableau numpy associé à la matrice d'adjacence de G. On pourra utiliser l'instruction `np.zeros((n,n),dtype=bool)` pour initialiser un tableau (n, n) contenant `False` pour chaque élément.

   2.2) Pour le graphe exemple G_1, on donne le dictionnaire d'adjacence `D1` associé au graphe. Créer la matrice d'adjacence `T1` associée et comparer les résultats des fonctions `dfs` et `dfs_M` appliquées respectivement sur `D1` et `T1`.

Exercice 2 **Parcours dfs récursif.** [Sol 2] Le parcours DFS permet de progresser dans un graphe « vers l'avant » en cherchant, à partir d'un sommet courant u, un sommet non encore visité v, puis si v existe, en réitérant l'opération à partir de v, et si v n'existe pas, en remontant au dernier sommet visité possédant des voisins non visités. Il est donc possible de coder une version récursive de cet algorithme, la pile des

sommets découverts introduite dans l'algorithme itératif étant ici remplacée par la pile des appels récursifs (pile automatiquement gérée par le langage). On se propose de coder une fonction récursive `dfs_rec(grph,s)`, où `graph` et `s` sont respectivement un dictionnaire d'adjacence et un sommet d'un graphe $G = (S, A)$, qui renvoie la liste des sommets visités dans l'ordre du parcours. On aura besoin d'utiliser le dictionnaire `visited` indiquant le statut visité (`True` ou `False`) de chaque sommet ainsi que la liste `lst_visited` des sommets visités dans l'ordre de leur visite.

1. On considère une fonction auxiliaire

   ```
   dfs_rec_aux(grph,u,visited,lst_visited)
   ```

   qui, à partir d'un sommet `u` non visité, modifie le statut visité de `u`, place `u` dans la liste des sommets visités, et réitère l'opération sur un des voisins non visités de `u` (si `u` n'a pas de voisin cette fonction ne fait rien). Lors de l'exécution de cette fonction, le contenu des variables `visited` et `lst_visited` évolue par effet de bord. Écrire le script de cette fonction.

2. Utiliser la fonction précédente pour écrire le script de la fonction `dfs_rec(grph,s)` présentée en introduction.

3. Tester la fonction sur le graphe G_1, et comparer avec les résultats renvoyés par la version non récursive `dfs`. Expliquer la différence de comportement.

Exercice 3 Dictionnaire des prédécesseurs.

[Sol 3] Dans les algorithmes vus en cours, les fonctions `bfs` et `dfs` renvoient la liste des sommets visités dans l'ordre dans lequel ils sont visités. Cette information ne permet cependant pas de reconstituer les embranchements qu'il peut y avoir lors du parcours d'un graphe.

- Lorsque qu'un sommet `u` est déclaré visité lors d'un parcours (c'est-à-dire qu'il vient d'être dépilé ou défilé), le sommet `u` avait donc été découvert depuis un sommet `w` (que l'on pourrait garder en mémoire, voir ci-après). On appelle alors *prédécesseur de* `u` le sommet `w` qui a permis de découvrir `u` lors de l'ajout dans la pile (ou la file).
- On peut conserver cette information en utilisant un dictionnaire `pred` tel que pour deux sommets `(w,u)` de type (parent, enfant), `w` soit la valeur associée à la clé `u` dans le dictionnaire `pred`, autrement dit, `pred` est le dictionnaire des parents de chaque sommet. On affectera par convention la valeur `None` au sommet de départ.
- Comme un même sommet `u` peut être découvert plusieurs fois lors d'un parcours (à partir de sommets différents, il peut être empilé/enfilé plusieurs fois), il faut alors déterminer quel sommet retenir comme parent de `u`.

Dans les parcours de l'exercice, on supposera que les sommets sont empilés/enfilés par numéros croissants.

1. **[Parcours en profondeur]**

   1.1) Écrire l'arbre couvrant de l'algorithme de parcours en profondeur sur le graphe `G1` à partir du sommet `2` et vérifier que, sur cet exemple, le parent du sommet `0` dans le parcours est le sommet à partir duquel `0` est découvert **pour la dernière fois** lors du parcours. *On peut montrer que ceci est valable quelque soit le graphe que l'on parcourt en profondeur, quelque soit le sommet d'origine, et quelque soit le sommet étudié.*

   1.2) Lors de chaque **découverte** d'un sommet u à partir d'un sommet w, on peut affecter à `pred[u]` la valeur `w`. En effet d'après la question précédente, seule la dernière affectation correspondra au parent de `u` dans le parcours final. Modifier la fonction `dfs(grph:dict, v)->list` fournie pour créer une fonction `dfs2(grph:dict, v)->dict` qui n'utilise pas de liste `lst_visited`, mais qui renvoie le dictionnaire des prédécesseurs `pred` comme indiqué précédemment.

   1.3) Tester cette fonction sur le dictionnaire `D1` associé au graphe `G1`.

2. **[Parcours en largeur]**

   2.1) Écrire l'arbre couvrant de l'algorithme de parcours en largeur sur le graphe `G1` à partir du sommet `2` et vérifier que, sur cet exemple, le parent du sommet `1` dans le parcours est le sommet à partir duquel `1` est découvert **pour la première fois** lors du parcours. *On peut montrer que ceci est valable quelque soit le graphe que l'on parcourt en largeur, quelque soit le sommet d'origine, et quelque soit le sommet étudié*

   2.2) Modifier la fonction `bfs(grph:dict, v)->list` fournie pour créer une fonction `bfs2(grph:dict, v)->dict` qui n'utilise pas de liste `lst_visited`, mais qui renvoie le dictionnaire des prédécesseurs `pred` comme indiqué précédemment.

   2.3) Tester cette fonction sur le dictionnaire `D1` associé au graphe `G1`.

3. Pour deux sommets quelconques `u` et `v` d'un graphe, le dictionnaire des prédécesseurs obtenu lors d'un parcours permet de déterminer simplement si lors du parcours, `u` et `v` sont reliés par un chemin uniquement descendant (pas de retour en arrière au niveau d'embranchement)[1].

1. Cette fonction ne permet pas de déterminer le chemin entre `u` et `v` dans le cas où `u` et `v` sont séparés par un embranchement, *i.e.* des sommets qui ne sont pas dans une même branche de l'arbre couvrant associé au parcours

3.1) Écrire une fonction `chemin(u,v,pred)` qui prend en argument deux sommets `u` et `v` d'un graphe, et le dictionnaire `pred` des prédécesseurs obtenu lors d'un parcours, et qui renvoie, s'il existe, le chemin descendant entre `u` et `v` sous la forme d'une liste $[u,\dots,v]$. Si un tel chemin n'existe pas, la fonction devra renvoyer la liste vide.

3.2) Tester la fonction `chemin`, pour un parcours en profondeur du graphe G_1 à partir du sommet 2, en cherchant le chemin descendant entre 2 et 5, puis entre 0 et 5.

Exercice 4 **Optimisation des ajouts dans la pile ou la file.** [Sol 4] On a vu en cours que les parcours DFS et BFS pouvaient conduire à ajouter à la variable qui stocke les sommets découverts (pile ou file suivant les cas) plusieurs fois le même sommet, ce qui peut provoquer une augmentation de la taille de cette variable, surtout pour les graphes ayant une grande quantité d'arêtes. Pour éviter cela, on peut choisir d'utiliser un dictionnaire des sommets découverts, appelé `discovered`, avec pour clés les sommets, tel que pour chaque sommet `s`, `discovered[s]` est initialisé à `False`, et passe à `True` lors de la découverte de `s`. *Il serait inefficace de tester à chaque étape l'appartenance à la pile ou la file...*

1. Créer une fonction :

```
s_disp(u, grph:dict, discovered:dict, visited:dict)
```

qui étant donnés un graphe `grph`, un sommet `u` et les deux dictionnaires `discovered` et `visited`, renvoie un sommet voisin de `u` n'ayant été ni visité, ni découvert (qui est un sommet disponible pour poursuivre le parcours). Si plusieurs sommets sont possibles, on prendra ici celui qui, dans la liste `grph[u]`, possède l'indice le plus grand. Si aucun sommet n'est disponible, la fonction devra renvoyer `None`.

2. On cherche à modifier la fonction `dfs2` vue précédemment qui réalise le parcours en profondeur de sorte que chaque sommet soit ajouté une seule fois dans la pile qui sert lors du parcours. Pour cela, lors de chaque itération lors du parcours,

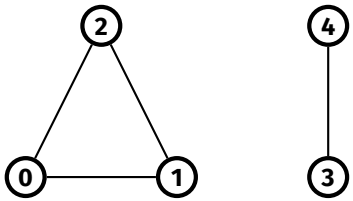
- si le sommet `u` en haut de la pile possède un voisin `w` qui n'a été ni découvert, ni visité, alors on empile `w`, on le marque comme découvert et on affecte `u` comme prédécesseur de `w`,
- sinon, on dépile `u` et on le marque comme visité.

Créer une fonction `dfs3` qui modifie la fonction `dfs2` en tenant compte des recommandations précédentes, et qui permet en plus de visualiser l'état de la pile au cours de l'exécution.

3. Exécuter `dfs3` sur le graphe G_1 pour un parcours à partir du sommet 2, et comparer avec les résultats renvoyés la fonction `dfs2` que l'on aura aussi modifiée pour afficher l'état de la pile à chaque itération.

2. APPLICATION DES PARCOURS

Exercice 5 **Recherche des composantes connexes d'un graphe.** [Sol 5] Le but de cet exercice est de créer une fonction permettant de déterminer les composantes connexes d'un graphe non orienté donné (une composante connexe d'un graphe G = (S,A) non orienté est l'ensemble des points qui sont reliés deux à deux par un chemin). On peut alors regrouper les sommets appartenant à une même composante connexe dans une liste L_i, et chercher à déterminer la liste L de toutes les listes L_i associées à toutes les composantes connexes de G. Par exemple, pour le graphe à 5 sommets suivant :



la liste `L` des composantes connexes est : `[[0,1,2],[3,4]]`.

Pour déterminer les composantes connexes, on peut utiliser une fonction de parcours de graphe à partir d'un sommet `s` pris au hasard. Tous les sommets visités à partir de `s` forment alors une composante connexe de G. On peut alors recommencer l'opération à partir d'un sommet de G qui n'a pas encore été visité, et ce jusqu'à ce que tous les sommets aient été visités. Le choix du type de parcours importe peu ici, cependant, afin de pouvoir facilement tester le fonction sur des graphes volumineux, on utilisera la fonction `dfs_M(grph:np.array, v:int)->list` écrite précédemment, dans laquelle la variable `grph` est le tableau numpy qui contient la matrice d'adjacence du graphe G.

1. Expliquer, sous formes d'étapes structurées en tirets, la fonction :

```
comp_connexe(grph:np.array)->list
```

qui prend comme argument le tableau `grph` qui contient la matrice d'adjacence d'un graphe, et qui renvoie la liste `L` précédemment définie.

2. Écrire le script de la fonction `comp_connexe`, en indiquant une docstring.

3. Afin de pouvoir tester la fonction précédente, on doit disposer de graphes variés comportant éventuellement plusieurs composantes connexes. On cherche dans cette question à écrire une fonction `init_mat_g(n:int,alpha:float)->np.array` où `n` est un entier correspondant au nombre de sommets du graphe voulu, `alpha` est un réel compris entre 0 et 1 fixant le nombre n_a d'arêtes du graphe ($n_a = \lfloor \alpha n(n-1)/2 \rfloor$), et qui renvoie un tableau numpy contenant la matrice d'adjacence du graphe généré. On pourra utiliser la fonction `np.random.randint(n)` qui renvoie aléatoirement un entier compris entre 0 et $n-1$.

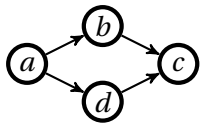
3.1) Écrire un script pour la fonction `init_mat_g` présentée précédemment (on ne cherchera pas ici à optimiser le temps d'exécution de cette fonction).

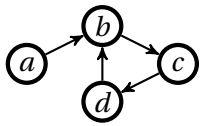
3.2) Générer un graphe de 10 sommets avec $\alpha = 0.07$, afficher la matrice d'adjacence associée, appliquer la fonction `comp_connexe` et contrôler son résultat à l'aide d'une étude « à la main » du graphe.

4. [Application] L'archipel HATARTE est formé de 11 îles numérotées de 0 à 10, qui sortent durement éprouvées d'une longue guerre civile, à l'issue de laquelle de nombreux ponts ont été détruits. Les ponts restés intacts sont les suivants : $[0,2]$, $[0,1]$, $[1,2]$, $[1,3]$, $[2,4]$, $[5,6]$, $[10,9]$, $[8,10]$. Combien de ponts faut-il reconstruire au minimum pour pouvoir relier toutes les îles ?

Exercice 6 Détection de cycle. [Sol 6]

En marquant les sommets visités, le parcours en profondeur évite de tourner en rond dans un cycle. Ainsi, retomber sur un sommet déjà visité lors d'un parcours peut indiquer la présence d'un cycle dans un graphe. Toutefois, ce retour vers un sommet ne se traduit par toujours par un parcours cyclique. Deux cas de figure mènent à une telle situation. Soit le sommet visité est effectivement découvert après un parcours cyclique. Soit il l'est après deux parcours parallèles. Sur la figure de gauche ci-dessous, un parcours en profondeur découvre les sommets a, b et c. Le parcours reprend ensuite à l'embranchement a en visitant ensuite les sommets d et c. La deuxième découverte du sommet c n'est pas liée à l'existence d'un cycle mais à deux parcours parallèles y menant. Sur la figure de droite, un premier parcours découvre le sommet b. Un second parcours y revient en raison, ici, d'un parcours cyclique.





Pour tenir compte de ces situations, les sommets sont marqués par trois **couleurs**.

- Un sommet non découvert est marqué en **blanc**.
- Un sommet blanc découvert est marqué en **gris**.
- Un sommet gris découvert est marqué en **noir**.

Dès lors, le parcours en profondeur peut être modifié de la façon suivante. Quand on découvre un sommet :

- s'il est **gris**, c'est qu'on vient de découvrir un cycle ;
- s'il est **noir**, on ne fait rien (ce marquage sert pour les parcours suivants) ;
- s'il est blanc :
  - ◇ on le colore en **gris** ;
  - ◇ récursivement, on visite tous ses voisins ;
  - ◇ enfin, on le colore en **noir**.

1. Écrire une fonction `dfs_cycle(grph, color, v)` qui renvoie le booléen `True` si le graphe `grph` contient un cycle issu du sommet `v`, le dictionnaire `color` contenant les couleurs des différents sommets au fur et à mesure du parcours.

2. En déduire une fonction `cycle(grph)` qui détecte la présence d'un cycle dans le graphe `grph` en appelant la fonction `dfs_cycle`.

SOLUTIONS DES EXERCICES

Solution 1

1. il suffit de modifier la ligne qui correspond à la définition du dictionnaire `visited`, ainsi que les instructions de la boucle `for`. On obtient le code suivant :

```
def dfs_M(grph:np.array, v:int)->list:
    n = len(grph)                          nombre de sommets
    s = deque()                            création d'une pile vide
    visited = {x : False \
        for x in range(n)}                 dictionnaire de booléens des sommets visités
    lst_visited = []                       liste des sommets visités
    s.append(v)                            empilement du sommet de départ
    while len(s) > 0:                      parcours des sommets non visités
        w = s.pop()                        dépilement du sommet de s
        if not visited[w]:                 si w non déjà visité
            visited[w] = True              w marqué comme visité
            lst_visited.append(w)          ajout de w à la liste des sommets visités
            for u in range(n):             parcours des voisins u de w
                if grph[w, u] \
                and not visited[u]:        si u non déjà visité
                    s.append(u)            empilement de u
    return lst_visited
```

2. **2.1)** Pour créer la matrice d'adjacence, il faut parcourir les clés de `D`, puis pour chaque clé `u`, parcourir les voisins `v` contenus dans `D[u]` (`u` et `v` sont de type `int`), et modifier l'élément `T[u,v]` correspondant. On obtient le code suivant :

```
def D2M(D:dict)->np.array:
    """ Construit le tableau T de la matrice d'adjacence \
    ↪ d'un graphe à partir du dictionnaire 'D' de sa liste \
    ↪ d'adjacence """
    n = len(D)
    T = np.zeros((n,n),dtype=bool)
    for u in D.keys():
        for v in D[u]:
            T[u,v] = True
    return T
```

Testons par exemple sur `D1`.

```
D1 = {
  0 : [1, 2],
  1 : [0, 3],
  2 : [0, 3, 4],
  3 : [1, 2],
  4 : [2,5,6],
  5 : [4,6],
  6 : [4,5],
}
```

2.2)

```
def dfs(grph, v):
    s = deque()                            création d'une pile vide
    visited = {x : False \
        for x in grph.keys()}              dictionnaire de booléens des sommets visités
    lst_visited = []                       liste des sommets visités
    s.append(v)                            empilement du sommet de départ
    while len(s) > 0:                      parcours des sommets non visités
        w = s.pop()                        dépilement du sommet de s
        if not visited[w]:                 si w non déjà visité
            visited[w] = True              w marqué comme visité
            lst_visited.append(w)          ajout de w à la liste des sommets visités
            for u in grph[w]:              parcours des voisins u de w
                if not visited[u]:         si u non déjà visité
                    s.append(u)            empilement de u
    return lst_visited
```

```
>>> D1 = {0: [2, 1], 1: [0, 3], 2:[0, 3, 4], 3:[2, 1], \
↪ 4:[2, 5, 6], 5:[4, 6], 6:[4, 5]}
>>> T1 = D2M(D1)
>>> dfs(D1, 2)
[2, 4, 6, 5, 3, 1, 0]
>>> dfs_M(T1, 2)
[2, 4, 6, 5, 3, 1, 0]
```

Solution 2

1. Fonction `dfs_rec_aux` :

```
def dfs_rec_aux(grph,u,visited,lst_visited):
    visited[u] = True
    lst_visited.append(u)
    for v in grph[u]:
```

```
        if not visited[v]:
            dfs_rec_aux(grph,v,visited,lst_visited)
```

La gestion des appels récursifs correspond bien à un parcours dfs : en effet, dès le premier appel dans la boucle principal,

2. Fonction globale

```
def dfs_rec(grph,s):
    """
    algorithme dfs récursif

    Parameters
    ----------
        grph : dict : dictionnaire de la liste d'adjacence du \
        ↪ graphe
        s : int ou str : sommet de départ

    Returns
    -------
        liste des sommets visités
    """
    # dictionnaire de booléens des sommets visités
    visited = {x : False for x in grph}
    # liste des sommets visités
    lst_visited = []
    dfs_rec_aux(grph,s,visited,lst_visited)
    return lst_visited
```

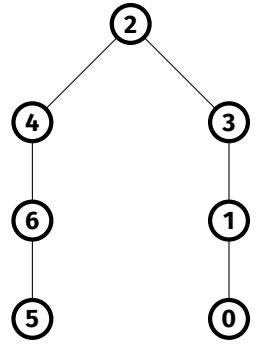
3.

```
>>> dfs_rec(G1,2)
[2, 0, 1, 3, 4, 5, 6]
>>> dfs(G1,2)
[2, 4, 6, 5, 3, 1, 0]
```

L'appel `d` renvoie la liste `[2, 0, 1, 3, 4, 5, 6]`, qui correspond bien à un parcours en profondeur à partir du sommet 2. L'appel `dfs(G1,2)` renvoie la liste `[2, 4, 6, 5, 3, 1, 0]` qui est une liste différente de la précédente mais qui correspond aussi à un parcours en profondeur à partir du sommet 2. La différence vient du fait que pour les appels récursifs à partir d'un sommet `u`, le sommet visité suivant est celui de plus petit numéro (le premier rencontré lors du parcours de `grph[u]` dans la boucle **`for`**), alors que pour l'algorithme itératif, c'est le dernier (celui qui est en haut de la pile des sommets découverts).

Solution 3

1. **1.1)** Puisqu'on empile les sommets par numéros croissants, on part au début dans la branche de droite du graphe. On obtient alors l'arbre couvrant suivant associé au parcours :



Dans ce parcours, le parent de `0` est bien `1`. Et cela correspond à la dernière fois que l'on a découvert `0` (on l'avait découvert une première fois partant de `2`, puis une seconde partant de `1`).

1.2) Modification du parcours en profondeur :

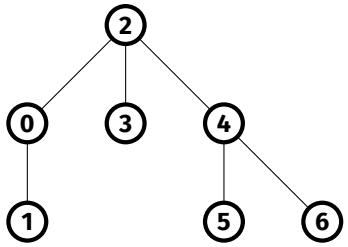
```
def dfs2(grph:dict, v)->dict:
    """
    renvoie le dictionnaire des prédécesseurs pour le \
    ↪ parcours en profondeur de 'grph'
    à partir de 'v'
    """
    s = deque()
    visited = {x : False for x in grph.keys()}
    pred = {}
    pred[v] = None
    s.append(v)

    while len(s) > 0:
        w = s.pop()
        if not visited[w]:
            visited[w] = True
            for u in grph[w]:
                if not visited[u]:
                    s.append(u)
                    pred[u] = w
    return pred
```

1.3) L'appel demandé conduit à

```
>>> dfs2(D1, 2)
{2: None, 0: 1, 3: 2, 4: 2, 5: 6, 6: 4, 1: 3}
```

2. 2.1) Puisqu'on enfile les sommets par numéros croissants, on obtient l'arbre ci-après :



Dans ce parcours, le parent de 1 est bien 0. Et cela correspond à la 1ère fois que l'on découvre 1 (on le découvre 2 fois, 1 première fois depuis 0, 1 seconde fois depuis 3).

2.2) Modification du parcours en largeur avec affectation du prédécesseur uniquement lors de la première découverte :

```
def bfs2(grph:dict, v)->dict:
    """
    renvoie le dictionnaire des prédécesseurs pour le \
    ↪ parcours en profondeur de 'grph' à partir de 'v'
    """
    q = deque()
    visited = {x : False for x in grph.keys()}
    pred = {}
    pred[v] = None
    q.append(v)

    while len(q) > 0:
        w = q.popleft()
        if not visited[w]:
            visited[w] = True
            for u in grph[w]:
                if not visited[u]:
                    q.append(u)
                    if u not in pred:
                        pred[u] = w
    return pred
```

2.3) Le résultat est alors :

```
>>> bfs2(D1,2)
{2: None, 0: 2, 3: 2, 4: 2, 1: 0, 5: 4, 6: 4}
```

3. 3.1) On peut proposer pour la fonction `chemin`

```
def chemin(u,v,pred:dict)->None:
    """ Renvoie s'il existe, le chemin qui a permis de \
    ↪ visiter 'v' à partir de 'u', sans embranchement, \
    ↪ pour un dictionnaire des prédécesseurs 'pred' donné, \
    ↪ sous forme d'une liste [u,....,v]
    """
    S = [] # liste des sommets de u à v
    if u in pred and v in pred:
        s = v
    while s != u and s != None:
        # on remonte à partir du sommet v
        S = [s] + S
        s = pred[s]
    if s == u: # arrivée au sommet de départ
        S = [s] + S # il manquait encore le départ
        return S
    else: # la remontée ne passe pas par u
        return []
    return [] # u et v non visités lors de ce parcours
```

3.2) On utilise les fonctions précédentes

```
>>> pred1 = dfs2(D1, 2)
>>> chemin(2,5,pred1)
[2, 4, 6, 5]
>>> chemin(0,5,pred1)
[]
```

Solution 4

1. On parcourt les sommets voisins de u en partant de la fin de la liste LV des voisins de u.

```
def s_disp(u,grph:dict,discovered:dict,visited:dict):
    """
```

```
    Pour le graphe 'grph' fourni, détermine, s'il existe, un \
    ↪ sommet voisin du sommet 'u' qui n'a été ni visité, ni \
    ↪ découvert. En cas de plusieurs possibilités, renvoie le \
    ↪ sommet d'étiquette la plus grande.
    Renvoie None si aucun sommet ne satisfait aux critères.
    """
    LV = grph[u]
    j = len(LV)-1
    while j > -1 and (discovered[LV[j]] or visited[LV[j]]):
        j = j-1
    if j > -1:
        return LV[j]
    else:
        return None
```

2. En appliquant les modifications indiquées, et en n'oubliant pas d'initialiser le tableau `discovered` nécessaire, on obtient :

```
def dfs3(grph:dict,v)->dict:
    """
    Pour le graphe 'grph' fourni, renvoie le dictionnaire \
    ↪ 'pred' des prédécesseurs
    obtenus lors du parcours en profondeur de 'grph' à partir \
    ↪ du sommet 'u'
    Dans cette version, on empile une seule fois chaque sommet.
    """
    s = deque()
    visited = {x:False for x in grph.keys()}
    discovered = {x:False for x in grph.keys()}
    pred = {x:None for x in grph.keys()}

    s.append(v)
    discovered[v] = True

    while len(s) > 0:
        print(s)
        u = s[-1]
        w = s_disp(u,grph,discovered,visited)
        if w != None:
            s.append(w)
            discovered[w] = True
            pred[w] = u
        else:
            s.pop()
            visited[u] = True
    return pred
```

3. On peut modifier `dfs3` en faisant afficher la pile à chaque itération. On obtient alors pour les deux appels demandés :

```
>>> dfs3(D1,2)
deque([2])
deque([2, 4])
deque([2, 4, 6])
deque([2, 4, 6, 5])
deque([2, 4, 6])
deque([2, 4])
deque([2])
deque([2, 3])
deque([2, 3, 1])
deque([2, 3, 1, 0])
deque([2, 3, 1])
deque([2, 3])
deque([2])
{0: 1, 1: 3, 2: None, 3: 2, 4: 2, 5: 6, 6: 4}
```

et

```
>>> dfs2(D1,2)
{2: None, 0: 1, 3: 2, 4: 2, 5: 6, 6: 4, 1: 3}
```

Les deux dictionnaires obtenus contiennent la même information, et les piles ont bien le comportement attendu (les sommets 5 et 0 ne sont pas empilés plusieurs fois dans le cas `dfs3`).

Solution 5

1. Raffinage de la fonction `comp_connexe` :

1. initialiser une liste vide `L` destinée à contenir les listes des différentes composantes connexes,
2. initialiser une liste `S` des sommets non encore visités,
3. tant que `S` n'est pas vide :
   - **3.1)** choisir un sommet `s0` de `S`,
   - **3.2)** initialiser une liste annexe vide `L1`,

3.3) appliquer un parcours de graphe pour déterminer la liste `L1` des sommets reliés à `s0`,
3.4) retirer de `S` les sommets de `L1`,
3.5) ajouter `L1` à `L`.

2. Script de la fonction `comp_connexe`

```
def comp_connexe(grph:np.array)->list:
    """
    Recherche les composantes connexes d'un graphe, par un parcours en profondeur

    Parameters
    -------
    grph : np.array
        tableau numpy de la matrice d'adjacence du graphe considéré

    Returns
    ---------
    L : list
        liste des listes des différentes composantes connexes
    """
    L = [] # liste de sortie
    n = len(grph)
    S = [i for i in range(n)] # liste des sommets non encore regroupés
    while S != []:
        s = S[0]
        L1 = dfs_M(grph, s) # liste des sommets de même composante que le sommet s
        for e in L1:
            S.remove(e)
        L.append(L1)
    return L
```

3. **3.1)** Code de la fonction `init_mat_g`

```
def init_mat_g(n,alpha):
    """
    renvoie le tableau numpy de la matrice d'adjacence d'un graphe de n sommets
    avec un taux de remplissage de alpha, soit N = E( alpha*n(n-1)/2 ) arêtes
    """
    T = np.zeros((n,n),dtype=bool)
    N = int(alpha*n*(n-1)/2) #nombres d'arêtes final
    c = 0 # compteur des arêtes
    while c < N:
        i = np.random.randint(n) # renvoie un entier i tq 0 <= i <= n-1
        j = np.random.randint(n)
        while i == j or T[i,j] == 1:
            i = np.random.randint(n)
            j = np.random.randint(n)
        c += 1
        T[i,j] = True
        T[j,i] = True
    return T
```

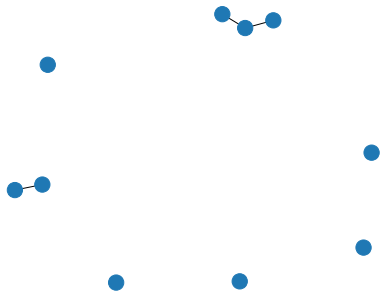
On peut remarque que pour α proche de 1, la deuxième boucle `while` peut être amenée à s'exécuter un grand nombre de fois avant de trouver un couple (i,j) correspondant à une arête non encore initialisée.

3.2) On exécute les lignes suivantes

```
>>> TG1 = init_mat_g(10,0.07)
>>> TG1
array([[False,  True, False, False, False, False, False, False, False,
        False],
       [ True, False, False, False, False, False, False, False, False,
         True],
       [False, False, False,  True, False, False, False, False, False,
        False],
       [False, False,  True, False, False, False, False, False, False,
        False],
       [False, False, False, False, False, False, False, False, False,
        False],
```

```
       [False, False, False, False, False, False, False, \
       ↪ False, False,
        False],
       [False, False, False, False, False, False, False, \
       ↪ False, False,
        False],
       [False, False, False, False, False, False, False, \
       ↪ False, False,
        False],
       [False, False, False, False, False, False, False, \
       ↪ False, False,
        False],
       [False,  True, False, False, False, False, False, \
       ↪ False, False,
        False]])
>>> comp_connexe(TG1)
[[0, 1, 9], [2, 3], [4], [5], [6], [7], [8]]
```

Il ne reste plus qu'à représenter le graphe à partir du tableau `TG1` et de vérifier que les composantes connexes renvoyées sont les bonnes :



4. On veut déterminer la liste des composantes connexes. On peut ici soit le faire à la main, soit créer le tableau `TGa` stockant la matrice d'adjacence du graphe des iles (chaque ile est un sommet, un pont est une arête). On écrit pour cela :

```
>>> TGa = np.zeros((11,11), dtype = bool)
>>> TGa[0,2] = True
>>> TGa[2,0] = True
>>> TGa[0,1] = True
>>> TGa[1,0] = True
>>> TGa[1,2] = True
>>> TGa[2,1] = True
>>> TGa[1,3] = True
>>> TGa[3,1] = True
>>> TGa[2,4] = True
>>> TGa[4,2] = True
>>> TGa[5,6] = True
>>> TGa[6,5] = True
>>> TGa[10,9] = True
>>> TGa[9,10] = True
>>> TGa[8,10] = True
>>> TGa[10,8] = True
```

Dans tous les cas, on obtient la liste des composantes connexes : `L = [[0, 1, 2, 4, 3], [5, 6], [7], [8, 10, 9]]`. Il y a quatre composantes connexes, donc il suffit de reconstruire trois ponts pour rendre l'ensemble connexe.

Solution 6

1. Trois variables globales sont définies.

```
BLANC, GRIS, NOIR = 1, 2, 3

def dfs_cycle(g, color, v):
    if color[v] == GRIS:
        return True
    if color[v] == NOIR:
        return False
    color[v] = GRIS
    for u in g.keys():
        if dfs_cycle(g, color, u):
            return True
        color[v] = NOIR
    return False
```

2. En entrant dans la fonction `cycle`, un dictionnaire `color` des couleurs est initialisé, chaque clé étant un sommet, chaque valeur étant `BLANC`. La boucle qui suit ne fait que parcourir l'ensemble des sommets du graphe et appelle la fonction `dfs_cycle` sur chacun des sommets.

```
def cycle(g):
    color = {v : BLANC for v in g.keys()}
    for v in g.keys():
        if dfs_cycle(g, color, v):
            return True
```

```
    return False
```

La boucle lance un parcours en profondeur à partir de chacun des sommets du graphe. Après quelques tours de boucle, pour beaucoup de ces sommets, le parcours est déjà passé par eux car ils étaient accessibles depuis des sommets déjà rencontrés. La fonction `dfs_cycle` peut se terminer sans rien faire. Le temps passé à détecter un cycle est alors pratiquement proportionnel à la taille du graphe.