Séq. 12 – Algorithmes portant sur les Graphes

Objectifs

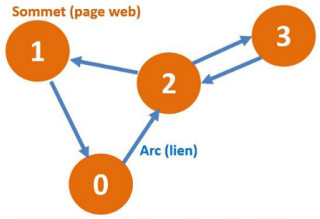
1. Parcourir un graphe en profondeur d'abord, en largeur d'abord (exemple de parcours d' un labyrinthe)
2. Repérer la présence d'un cycle dans un graphe
3. Chercher un chemin dans un graphe
4. Modéliser à l'aide de classes Python
5. Faire un lien avec les protocoles de routage

Cette séquence s'appuie sur :

- https://www.nsi-ljm.fr/NSI-TLE/res/res_bfs.pdf

1 Introduction

On peut modéliser un réseau de pages web reliées entre elles par des hyperliens à l'aide d'un graphe orienté. Dans l'exemple ci-dessous, la page web 2 possède deux liens sortants (un vers la page 1 et un vers la page 3) et deux liens entrants (un depuis la page 0 et un depuis la page 3).



Modélisation d'un réseau de pages web par un graphe orienté

Pour la suite des activités suivantes, on admettra que ce graphe est connexe, c'est-à-dire qu'à partir d'une page donnée, toute autre page est accessible par une série de liens (une chaîne).

L'objectif est de mettre en œuvre les techniques permettant de tester toutes les pages d'une structure modélisable sous forme d'un graphe comme celui-ci.

C'est-à-dire à parcourir ce graphe en passant par tous ses sommets une seule fois, soit pour lister simplement tous ses sommets, soit pour trouver un chemin reliant un sommet à un autre.

Nous avons déjà vu comment parcourir un arbre. Le principe est identique.

Une nuance importe toutefois, dans un arbre, il est impossible de passer plusieurs fois par le même nœud, dans un graphe ce risque existe …

Il faut donc trouver un moyen de marquer les sommets rencontrés afin de ne plus les explorer.

Autre nuance importante, plusieurs arêtes peuvent mener à un même sommet, lors d'un parcours, on cherche à visiter chaque sommet, pas chaque arête.

2 Méthodes de parcours de graphe

2.1 Parcours en largeur ou Breadth First Search (BFS)

2.1.1 Principe résumé :

Pour le parcours en largeur (BFS pour Breadth First Search), on commence avec un nœud donné et on explore chacun de ses voisins avant d'explorer leurs enfants. Autrement dit, on commence d'abord par aller sur la plus grande largeur possible. Son implémentation repose sur une file (queue) dont le principe du premier entré, premier sorti (FIFO : First In / First Out) permet de s'assurer que les nœuds découverts d'abord seront explorés en premier.

2.1.2 Principe détaillé:

On procède à un parcours en largeur du graphe en mettant les sommets successifs dans une file (structure FIFO, First In, First Out).

Rappel :
La structure de file est celle d'une file d'attente à un guichet : les nouvelles personnes qui arrivent se rangent à la fin de la file d'attente. La personne servie est celle qui est arrivée en premier dans la file.

Voici la description intuitive de l'algorithme :

1. On enfile le sommet de départ (on visite la page d'accueil du site).
2. On enfile les sommets adjacents à la tête de file (on visite les pages ciblées par la page d'accueil) s'ils ne sont pas déjà présents dans la file.
3. On défile (c'est-à-dire on supprime la tête de file).
4. Tant que la file n'est pas vide, on réitère les points 2 et 3.

En d'autres termes, on défile toujours prioritairement les sommets (les pages) les plus tôt découverts.

On donne le nom de parcours en largeur d'un graphe car cet algorithme va visiter tous les sommets à distance 1 du sommet de départ puis tous les sommets à distance 2 du sommet de départ puis tous les sommets à distance 3 du

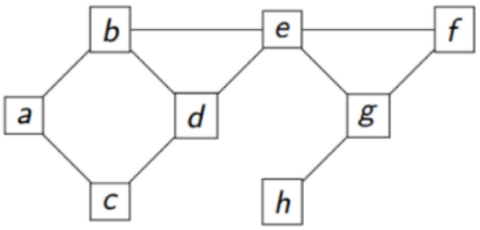
sommet de départ etc...

Si le graphe est connexe, le parcours obtenu décrit un arbre de type arbre couvrant.

A faire vous même 1.

Pour mieux comprendre voir la vidéo : https://www.youtube.com/watch?v=NrQGxfFMYzs

2.1.3 Mise en œuvre à la main:



Reprenons le graphe connexe déjà utilisé au chapitre précédent.

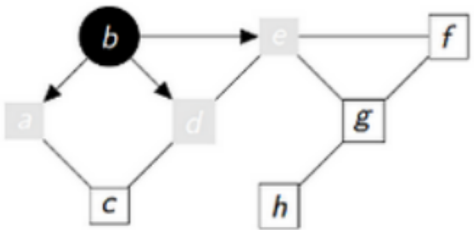
On dispose d'un graphe G, d'une liste sommets_visités et d'une file f pour enfiler et défiler. Le sommet de départ est par exemple 'b'.

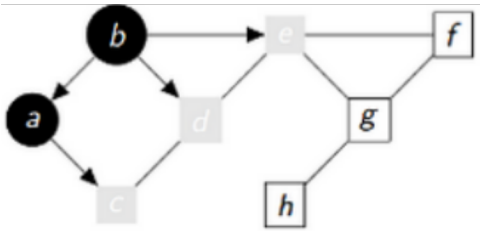
```
On enfile le sommet de départ
Puis tant que la file f n'est pas vide :
      On défile f dans une variable tmp
      Si tmp n'est pas dans sommets_visités :
            On l'ajoute à sommets_visités
            Pour chaque voisin de tmp :
                  S'il n'est ni dans sommets_visités ni
                  dans la file f :
                        On l'enfile dans la file f
      On renvoie sommets_visités
```

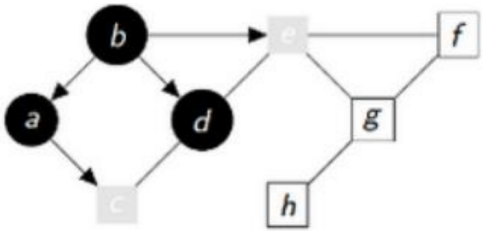
On donne dans le tableau ci-dessous les deux premières lignes complétées à partir de l'algorithme décrit ci-dessus. Le code couleur est le suivant : blanc si le sommet S n'est pas passé dans la file, gris si le sommet S est dans la file et noir si le sommet S est sorti de la file.

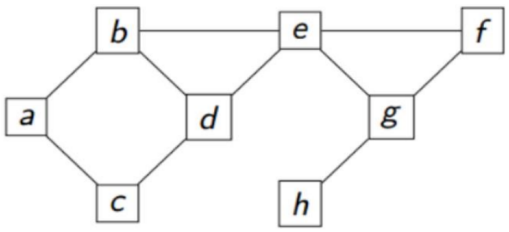
A faire vous même 2.

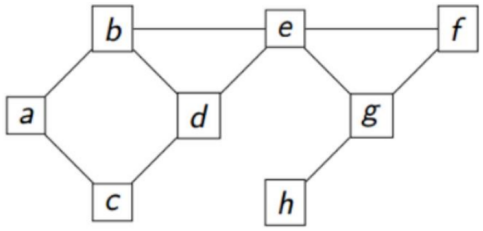
Compléter le tableau ci-dessous. Reprenez les colorations du graphe à partir du 4ème tour de boucle.

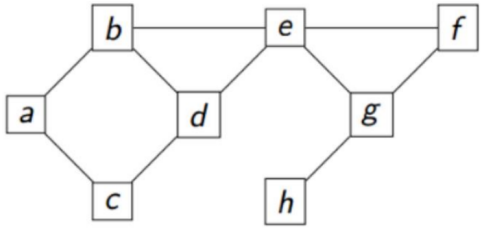
Tour de boucle 1		`tmp = 'b'`
		`sommets_visités = ['b']`
		`f = 'e' , 'd' , 'a'`

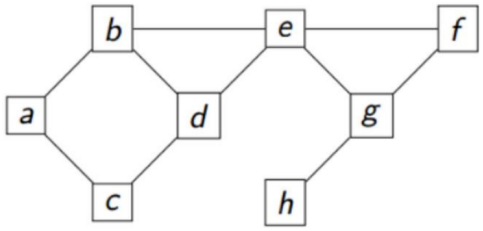
Tour de boucle 2		`tmp = 'a'`
		`sommets_visités = ['b', 'a']`
		`f = 'c' , 'e' , 'd'`

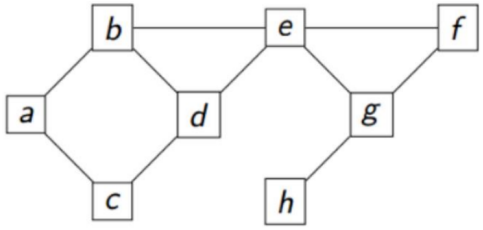
Tour de boucle 3		`tmp =`
		`sommets_visités =`
		`f=`

Tour de boucle 4		`tmp =`
		`sommets_visités =`
		`f=`

Tour de boucle 5		`tmp =`
		`sommets_visités =`
		`f=`

Tour de boucle 6		`tmp =`
		`sommets_visités =`
		`f=`

Tour de boucle 7		`tmp =`
		`sommets_visités =`
		`f=`

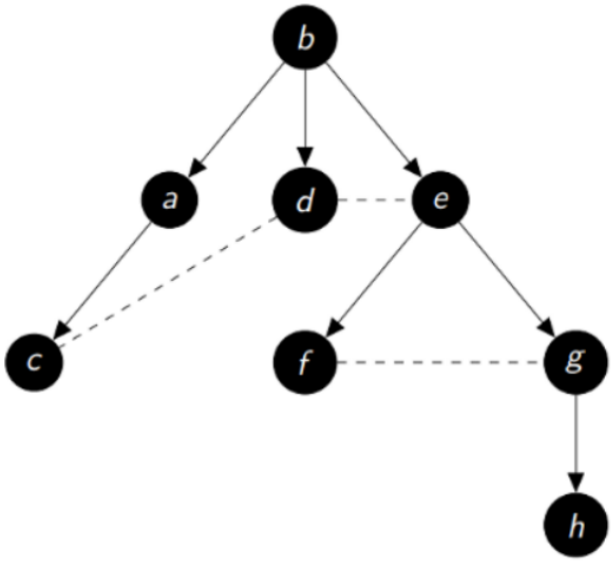
Tour de boucle 8		`tmp =`
		`sommets_visités =`
		`f=`

A faire vous même 3.

Donner l'arbre couvrant montrant l'arborescence associée au parcours en largeur effectué précédemment, ceci sous une forme semblable à celle indiquée dans la vidéo ci-dessus (on gardera en pointillés les arcs supprimables).

Au final l'arborescence associée au parcours peut donc être modélisée de la façon suivante :

['b', 'a', 'd', 'e', 'c', 'f', 'g', 'h']



2.1.4 Programmation en Python du BFS

Voici une classe `File` dans laquelle on a rajouté une méthode `present(self,x)` qui renvoie vrai si `x` est dans la file.

```
class File:
    """ classe File
    création d'une instance File avec une liste
    """
    def __init__(self):
        self.L = []
    def vide(self):
        return self.L == []
    def defiler(self):
        assert not self.vide(), "file vide"
        return self.L.pop(0)
    def enfiler(self,x):
        self.L.append(x)
```

```
    def taille(self):
        return len(self.L)
    def sommet(self):
        return self.L[0]
    def present(self,x):
        return x in self.L
```

Voici le code pour la création du dictionnaire qui représente le graphe G et une fonction qui renvoie les voisins d'un sommet

```
G = dict()
G['a'] = ['b','c']
G['b'] = ['a','d','e']
G['c'] = ['a','d']
G['d'] = ['b','c','e']
G['e'] = ['b','d','f','g']
G['f'] = ['e','g']
G['g'] = ['e','f','h']
G['h'] = ['g']
def voisins(G,sommet):
    return G[sommet]
```

A faire vous même 4.

Implémentez l' algorithme BFS décrit ci-avant en Python et testez-le sur notre graphe G.

2.1.5 Programmation récursive du BFS

La présence d'une boucle `while` nous suggère la version récursive de cet algorithme.

On dispose d'un graphe, d'une File contenant le sommet de départ, d'une liste contenant le sommet de départ et qui nous servira à marquer les sommets visités.

Le processus :

1. on défile la File dans une variable (tmp) (on l'affiche)
2. pour chaque voisins non déjà visité de tmp
3. on le note comme visité
4. on l'enfile
5. on recommence du 1

Le processus s'arrête quand la File est vide

Voici le programme :

```
def bfs_recur(G,f,sommets_visites):
    if f.vide():
        return None
    tmp=f.defiler()
    print(tmp,end=" ")
    for u in voisins(G,tmp):
        if u not in sommets_visites:
            sommets_visites.append(u)
            f.enfiler(u)
    bfs_recur(G,f,sommets_visites)
f=File()
sommets_visites=[]
sommet='b'
f.enfiler(sommet)
sommets_visites.append(sommet)
bfs_recur(G,f,sommets_visites)
```

A faire vous même 5.

Complétez (pre/post-conditions, docstring et quelques commentaires) et faites fonctionner ce programme pour notre graphe.

A faire vous même 6. Pour les rapides (bfs1.py)

Au chapitre précédent, vous avez vu que tout graphe est caractérisé par sa matrice d'adjacence composée de 1 et de 0 selon que deux sommets sont ou ne sont pas reliés par une arête et qu'une façon d'encoder un graphe sous Python est d'utiliser un dictionnaire qui est la représentation de sa matrice d'adjacence. La clé associée à chaque sommet est la liste des sommets adjacents.

Avec une telle représentation d'un graphe, la programmation du BFS peut se réaliser avec une fonction `bfs(graphe, sommet de départ)` dont les variables seront les suivantes :

- Un dictionnaire P tel que, en fin de parcours, pour tout sommet S du graphe, P[S] sera le père de S, c'est-à-dire le sommet à partir duquel le sommet S a été découvert lors du parcours.
- Un dictionnaire couleur tel que, pour tout sommet S, couleur[S] est :
  - blanc si le sommet S n'est pas passé dans la file,

- gris si le sommet S est dans la file,
- noir si le sommet S est sorti de la file.

Une liste Q utilisée comme file (FIFO) : on enfile un sommet lorsqu'il est découvert et on le défile lorsqu'il est terminé (traitement prioritaire des sommets découverts au plus tôt).

Implémentez cet algorithme et testez-le sur notre graphe G.

Tracez l' arbre couvrant avec Graphviz.

2.1.6 Arbre couvrant et affichage d' un chemin entre deux sommets.

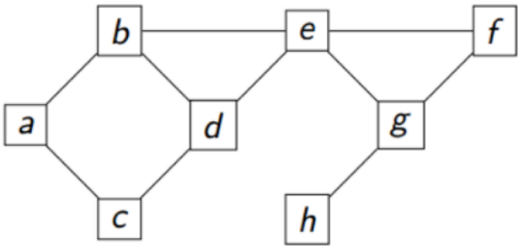
L'objectif est de faire afficher un chemin entre deux sommets d'un graphe.

Par exemple :

a-b-e-g-h est l'un des chemins possible entre a et h

La méthode consiste à mémoriser les sommets voisins du sommet visité comme clés d'un dictionnaire et ayant pour valeur son parent(le sommet visité).

Le sommet de départ n'aura bien entendu pas de parent (None)



À la fin notre dictionnaire parents sera:

```
{'a': None, 'b': 'a', 'c': 'a', 'd': 'b', 'e': 'b', 'f': 'e', 'g': 'e', 'h': 'g'}
```

Il nous faudra lire ce dictionnaire pour pouvoir établir le chemin entre `'a'` et `'h'`

`h` a pour parent `g` qui a pour parent `e` qui a pour parent `b` qui a pour parent `a` d'où le chemin : a - b - e - g - h

A faire vous même 7.

Complétez les lignes de code de la fonction `solution`, puis testez la dans votre script (ou le srcipt `bfs1.py`) avec le même graphe.

```
def Solution(end, parents):
    chemin = []
    courant = end
    while . . .:
        chemin = . . .
        courant = . . .
    return chemin
```

```
def Solution(end, parents):
    chemin = []
    courant = end
    while courant != None:
        chemin = [courant] + chemin
        courant = parents[courant]
    return chemin
```

2.2 Parcours en profondeur ou Depth First Search (DFS)

2.2.1 Principe résumé :

Pour le parcours en profondeur (DFS pour Depth First Search), on commence avec un nœud donné et on explore chaque branche complètement avant de passer à la suivante. Autrement dit, on commence d'abord par aller le plus profond possible. Cet algorithme s'écrit naturellement de manière récursive et permet de trouver tous les nœuds auquel un nœud est connecté.

2.2.2 Principe détaillé:

On procède à un parcours en profondeur du graphe en mettant les sommets successifs dans une pile (structure LIFO, Last In, First Out).

Rappel :

La structure de pile est celle d'une pile d'assiettes : Pour ranger les assiettes, on les empile les unes sur les autres.

Lorsqu'on veut utiliser une assiette, c'est l'assiette qui a été empilée en dernier qui est utilisée.

Voici la description intuitive de l'algorithme :

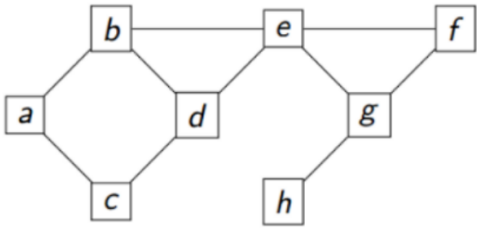
```
On empile le sommet de départ (on visite la page d'accueil du site).
Tant que la pile n'est pas vide :
    Si le sommet de la pile possède des voisins qui ne sont pas dans la pile, ni déjà passés dans la pile, alors on sélectionne l'un de ces voisins et on l'empile (en le marquant de son numéro de découverte)
    Sinon on dépile (c'est-à-dire on supprime l'élément au sommet de la pile).
```

En d'autres termes, on traite toujours en priorité les liens des pages les plus tard découvertes.

Si le graphe est connexe, le parcours obtenu décrit un arbre de type arbre couvrant.

Pour mieux comprendre : https://youtu.be/kcedjJOjDpg

2.2.3 Mise en œuvre à la main:



Reprenons le graphe connexe déjà utilisé au chapitre précédent.

On dispose d'un graphe `G`, de deux listes, `sommets_visités` `sommets_fermés` et d'une pile `p` pour empiler et dépiler.

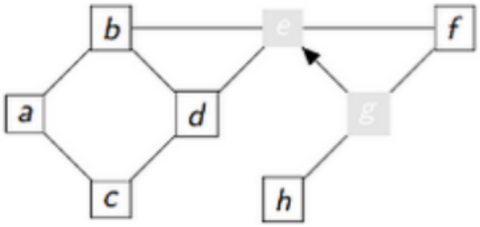
Le sommet de départ est par exemple 'g'.

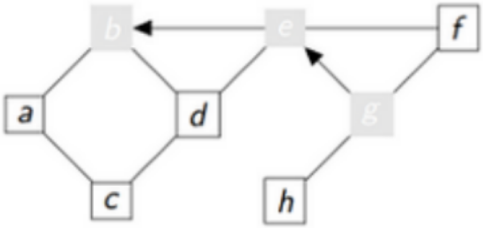
```
On empile le sommet de départ.
On met le sommet de départ dans la liste sommets_visités
Puis tant que la pile p n'est pas vide :
      On récupère le sommet de la pile p dans une variable tmp
      voisins reçoit la liste des voisins de tmp non déjà visités
      Si voisins n'est pas vide :
            v ← un voisin choisi au hasard
            sommets_visités ← v
            On empile v
      Sinon :
            sommets_fermés ← tmp
            On dépile p
```

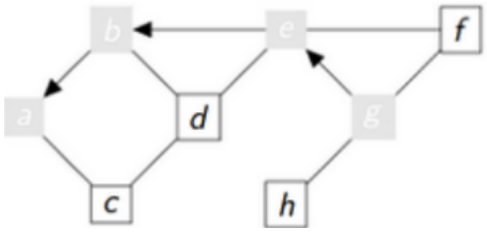
On donne dans le tableau ci-dessous les deux premières lignes complétées à partir de l'algorithme décrit ci-dessus. Le code couleur est le suivant : blanc si le sommet S n'est pas passé dans la file, gris si le sommet S est dans la file et noir si le sommet S est sorti de la file.

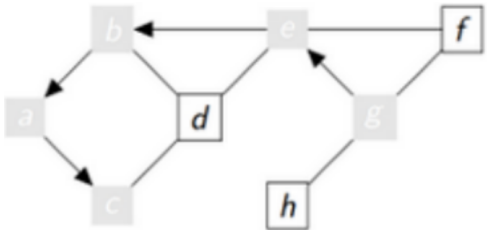
A faire vous même 8.

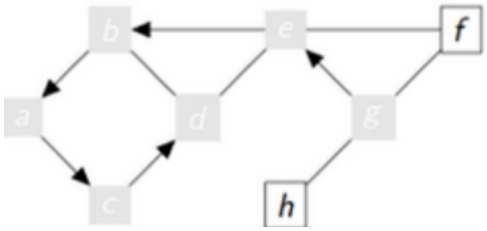
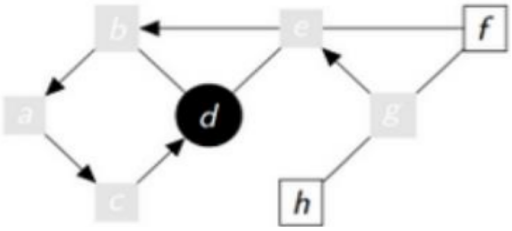
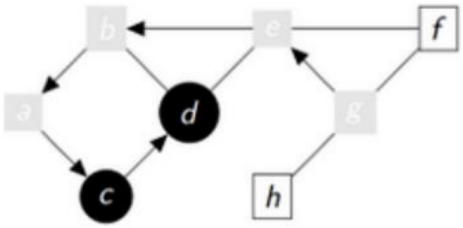
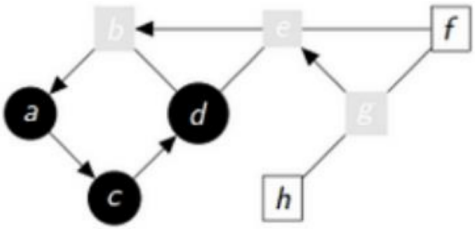
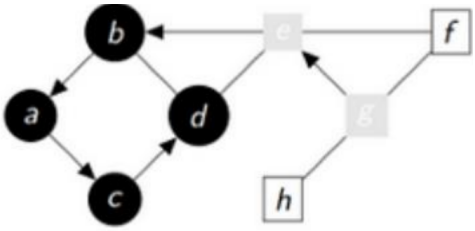
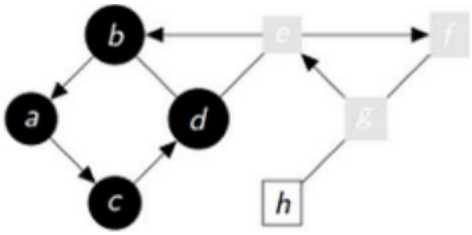
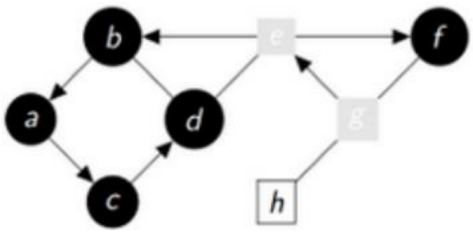
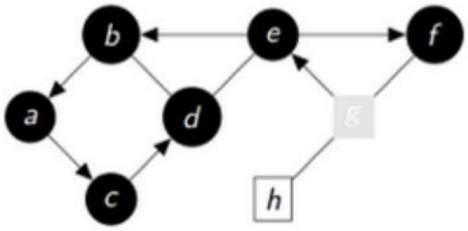
Compléter le tableau ci-dessous. Reprenez les colorations du graphe quand nécessaire.

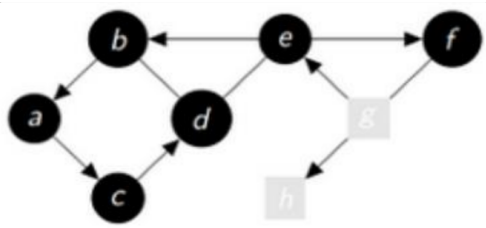
Tour de boucle 1		`tmp = 'g'`
		`voisins = ['e', 'f', 'h']`
		`v = 'e'`
		`sommets_visités = ['g', 'e']`
		`p = ['g', 'e']`
		`sommets_fermés = [ ]`

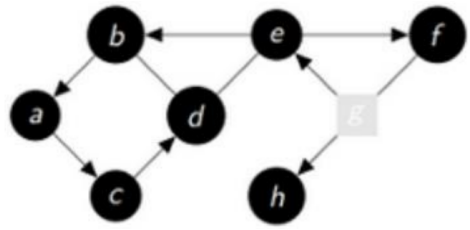
Tour de boucle 2		`tmp = 'e'`
		`voisins = ['b', 'd', 'f']`
		`v = 'b'`
		`sommets_visités = ['g', 'e', 'b']`
		`p = ['g', 'e', 'b']`
		`sommets_fermés = [ ]`

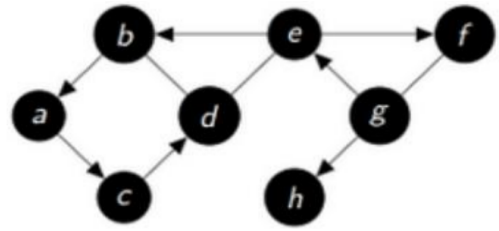
Tour de boucle 3		`tmp =`
		`voisins =`
		`v =`
		`sommets_visités =`
		`p =`
		`sommets_fermés =`

Tour de boucle 4		`tmp =`
		`voisins =`
		`v =`
		`sommets_visités =`
		`p =`
		`sommets_fermés =`

Tour de boucle 5		tmp = voisins = v = sommets_visités = p = sommets_fermés =
Tour de boucle 6		tmp = voisins = v = sommets_visités = p = sommets_fermés =
Tour de boucle 7		tmp = voisins = v = sommets_visités = p = sommets_fermés =
Tour de boucle 8		tmp = voisins = v = sommets_visités = p = sommets_fermés =
Tour de boucle 9		tmp = voisins = v = sommets_visités = p = sommets_fermés =
Tour de boucle 10		tmp = voisins = v = sommets_visités = p = sommets_fermés =
Tour de boucle 11		tmp = voisins = v = sommets_visités = p = sommets_fermés =
Tour de boucle 12		tmp = voisins = v = sommets_visités = p = sommets_fermés =

Tour de boucle 13		`tmp =`
		`voisins =`
		`v =`
		`sommets_visités =`
		`p =`
		`sommets_fermés =`

Tour de boucle 14		`tmp =`
		`voisins =`
		`v =`
		`sommets_visités =`
		`p =`
		`sommets_fermés =`

Tour de boucle 15		`tmp =`
		`voisins =`
		`v =`
		`sommets_visités =`
		`p =`
		`sommets_fermés =`

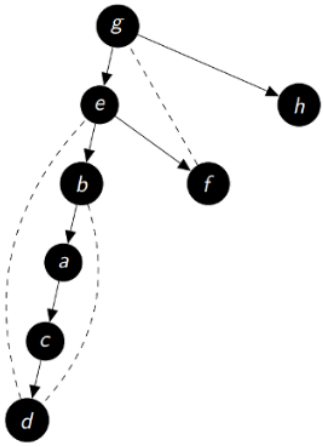
Remarque :
Comme les choix dans la liste des voisins sont aléatoires, il y a plusieurs parcours possibles.

A faire vous même 9.

Donner l'arbre couvrant montrant l'arborescence associée au parcours en largeur effectué précédemment, ceci sous une forme semblable à celle indiquée dans la vidéo ci-dessus (on gardera en pointillés les arcs supprimables).

Au final l'arborescence associée au parcours peut donc être modélisée de la façon suivante :

['d', 'c', 'a', 'b', 'f', 'e', 'h', 'g']



2.2.4 Programmation en Python du DFS

Voici une classe `Pile` .

```
class Pile:
    """ classe Pile
    création d'une instance Pile avec une liste
    """
    def __init__(self):
        self.L = []
    def vide(self):
        return self.L == []
    def depiler(self):
        assert not self.vide(),"Pile vide"
        return self.L.pop()
    def sommet(self):
        assert not self.vide(),"Pile vide"
        return self.L[-1]
    def empiler(self,x):
        self.L.append(x)
```

Voici le code pour la création du dictionnaire qui représente le graphe G et une fonction qui renvoie les voisins d'un sommet

```
G = dict()
G['a'] = ['b','c']
G['b'] = ['a','d','e']
G['c'] = ['a','d']
G['d'] = ['b','c','e']
G['e'] = ['b','d','f','g']
G['f'] = ['e','g']
```

```
G['g'] = ['e','f','h']
G['h'] = ['g']
def voisins(G,sommet):
    return G[sommet]
```

Voici une ligne de code qui permet de récupérer les voisins de tmp non déjà visités:

```
voisins=[y for y in voisin(G,tmp) if y not in sommets_visites]
```

La bibliothèque `random` permet un choix aléatoire dans une liste:

```
import random
v=random.choice(voisins)
```

L'algorithme du DFS

```
fonction parcours_profondeur(G,sommet):
sommets_visités ← []
sommets_fermés ← []
p ← Pile()
sommets_visités ← sommet
On empile le sommet dans p
Tant que p n'est pas vide faire
      tmp ← le sommet de la pile
      voisins ← la liste des voisins de tmp non déjà visités
      Si voisins n'est pas vide alors
            v ← un voisin au hasard
            sommets_visités ← v
            On empile v
      Sinon
            sommets_fermés ← tmp
            on dépile p
fin tant que
renvoyer sommets_fermés
```

A faire vous même 10.

Implémentez l' algorithme DFS décrit ci-avant en Python et testez-le sur notre graphe G.

A faire vous même 11. Pour les rapides

Reprenons l'implémentation pour le graphe précédente pour la programmation du DFS qui peut se réaliser avec une fonction `dfs(graphe, sommet de départ)` dont les variables seront les suivantes :

- Un dictionnaire `P` tel que, en fin de parcours, pour tout sommet `S` du graphe, `P[S]` sera le père de S, c'est-à-dire le sommet à partir duquel le sommet `S` a été découvert lors du parcours.
- Un dictionnaire couleur tel que, pour tout sommet `S`, `couleur[S]` est :
  - blanc si le sommet `S` n'est pas passé dans la pile,
  - gris si le sommet `S` est dans la pile,
  - noir si le sommet `S` est sorti de la pile.
- Une liste `Q` utilisée comme pile (LIFO) : on désempile un sommet et on empile ses voisins non encore explorés.

Testez le script `dfs1.py` (http://ninoo.fr/LC/Term_NSI/seq12_algorithmes_des_graphes/dfs1.py) en utilisant le graphe avant.

1. Le résultat présenté est-il conforme à l'algorithme ?
2. Documentez la fonction `dfs` et commentez chaque ligne de code. Pour cela, entre autres, reprenez les éléments explicatifs fournis.

2.2.5 Programmation récursive du DFS

On peut utiliser un algorithme récursif pour parcourir un graphe en profondeur.

En voici la description:

1. On part d'un nœud du graphe.
2. On le marque comme visité s'il ne l'est pas déjà.
3. Pour chacun de ses voisins non visités, on reprend à partir du 1.

Il y a une "boucle" du 3 au 1. Cela présage une méthode récursive. Voici l'algorithme davantage détaillé:

```
Données : G est un graphe
sommet est un sommet du graphe
sommets_visités est une liste
fonction dfs(G,sommet):
Si le sommet n'est pas dans la liste sommets_visités
alors
      On le met dans la liste
```

```
    voisins ← la liste des voisins de sommet non déjà visités
    Pour chaque voisin dans voisins faire
        dfs(G,voisin)
    renvoyer sommets_visités
```

A faire vous même 12.

Écrire la fonction dfs et la faire fonctionner avec notre graphe avec comme sommet de départ 'g'.

Remarque :
Le choix du 1er voisin est le premier de la liste voisins qui correspond à celle implantée lors de la création du graphe: `G['g'] = ['e','f','h']`, en la modifiant par `G['g'] = ['f','e','h']`, vous obtiendrez un autre parcours...

Le résultat attendu est : `['g', 'e', 'b', 'a', 'c', 'd', 'f', 'h']`

Et en renversant la liste `['h', 'f', 'd', 'c', 'a', 'b', 'e', 'g']` pour obtenir la liste des sommets fermés de la version itérative.

2.2.6 Arbre couvrant et affichage d' un chemin entre deux sommets.

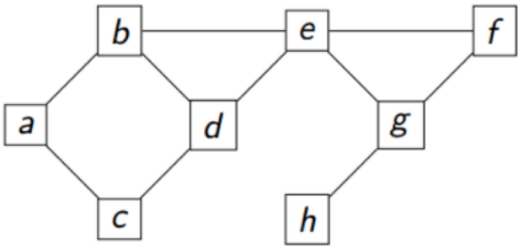
L'objectif est de faire afficher un chemin entre deux sommets d'un graphe.

Par exemple :

a-b-e-g-h est l'un des chemins possible entre a et h

La méthode consiste à mémoriser les sommets voisins du sommet visité comme clés d'un dictionnaire et ayant pour valeur son parent(le sommet visité).

Le sommet de départ n'aura bien entendu pas de parent (None)



A faire vous même 13.

Reprenez la fonction `solution`, puis testez la dans votre script (ou le script `dfs1.py`) avec le même graphe.

```
def Solution(end, parents):
    chemin = []
    courant = end
    while courant != None:
        chemin = [courant] + chemin
        courant = parents[courant]
    return chemin
```

3 Présence d'un cycle dans un graphe

Il existe beaucoup de variantes. Nous n'en présentons qu'une.

On utilise un parcours en largeur (en profondeur c'est pareil).

La différence est le tableau des drapeaux.

Cette fois, lorsqu'on dépile, on ajoute la règle suivante :

- `lorsqu'on dépile un élément, on passe le drapeau à 1.`

Et lorsqu'on cherche à empiler les voisins, on ajoute la règle suivante :

- `si un voisin rencontré a un drapeau à 0, c'est qu'il y a un cycle.`

3.1 Algorithme complet

```
source                          (un noeud du graphe)
file: [Source]                  (une file)
drapeaux : [-1, -1, etc., -1] (un tableau avec -1 pour chaque indice de sommet)

Dans le tableau drapeaux, si un sommet est d'indice 2,
drapeaux[2] = -1 signifie qu'on ne l'a pas encore ajouté à la file.
drapeaux[2] = 0 signifie qu'on l'a déjà ajouté à la file mais pas encore visité.
drapeaux[2] = 1 signifie qu'on l'a déjà visité le sommet.

Fonction Contient un cycle (graphe)
      Choisir un sommet (n'importe lequel) et l'ajouter à la file.
      Tant que la file n'est pas vide faire :
            courant = défiler()
            passer le drapeau de courant à 1.
            Pour chaque voisin de courant :
```

```
                        Si son drapeau vaut 0:
                              On a déjà rencontré ce sommet ! Il y a un cycle.
                              Cycle_present = Vrai
                        Si son drapeau vaut -1 :
                              l'ajouter à la file.
                              Changer son drapeaux en 0.
            Retourner Cycle_present
```

3.2 Exemples détaillés

3.2.1 Graphe n°1

```
file = [0], drapeaux = [-1, -1, -1, -1, -1, -1], Cycle_present = faux
   1. courant = 0. Voisins = 1, 2. drapeaux = [1, 0, 0, -1, -1, -1]. File = [1, 2]
   2. courant = 1. Voisins = 2, 5.
      Le drapeau de 2 vaut 0 !!! Il y a un cycle.
      Cycle_present = Vrai
      ... le parcours se continue …
   3. On retourne Vrai
```

3.2.2 Graphe n° 2

```
file = [0], drapeaux = [-1, -1, -1, -1, -1, -1], Cycle_present = Faux
   1. courant = 0. Voisins = 1, 2. drapeaux = [1, 0, 0, -1, -1, -1]. File = [1, 2]
   2. courant = 1. Voisins = 0, 5. drapeaux = [1, 1, 0, -1, -1, 0]. File = [2, 5]
   3. courant = 2. Voisins = 0, 3, 4. drapeaux = [1, 1, 1, 0, 0, 0]. File = [5, 3, 4]
   4. courant = 5. Voisins = 1. drapeaux = [1, 1, 1, 0, 0, 1]. File = [3, 4]
   5. courant = 3. Voisins = 2. drapeaux = [1, 1, 1, 1, 0, 1]. File = [4]
   6. courant = 4. Voisins = 2. drapeaux = [1, 1, 1, 1, 1, 1]. File = []
On retourne Faux
```

À aucun moment la variable `Cycle_present` n'a changé d'état.

1.

2.

3.