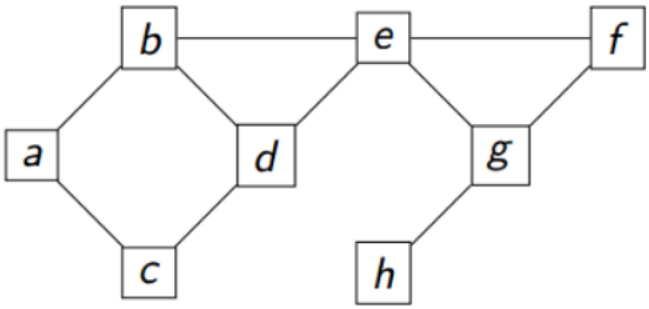
Lycée Jean Moulin **Draguignan**
NSI *Terminale* Année .....

TD - Parcours en profondeur d'un graphe - DFS

La méthode - DFS(Depth First Search)

Parcourir un graphe en profondeur à partir d'un sommet, consiste à explorer le graphe en suivant un chemin.
Lorsqu'on arrive sur un sommet qui n'a plus de voisins non visités, on le marque.
Puis on remonte dans le chemin pour explorer les voisins non visités d'un autre sommet...

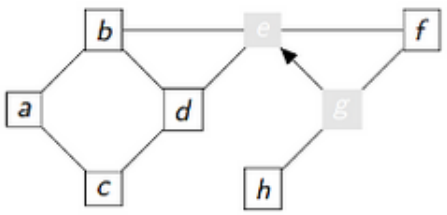
On utilise une pile et deux listes
Prenons en exemple ce graphe :



On dispose d'un graphe(G), de deux listes (sommets_visités , sommets_fermés) et d'une pile(p)
Le sommet de départ est par exemple 'g', **on l'empile**.
On met le sommet de départ dans la liste sommets_visités
Puis **tant que la pile n'est pas vide** :

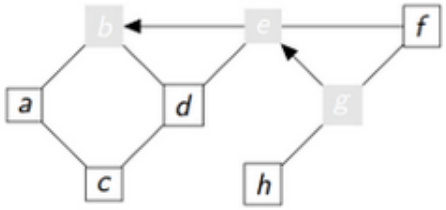
- On récupère **le sommet de la pile** dans une variable tmp
- **voisins reçoit la liste des voisins de tmp non déjà visités**
- **Si voisins n'est pas vide :**
  - **v ← un voisin choisi au hasard**
  - **sommets_visités ← v**
  - On **empile** v
- **Sinon :**
  - **sommets_fermés ← tmp**
  - On **dépile**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

Voici les contenus des variables au premier tour de la boucle tant que :



1 tour
tmp : g
voisins : ['e', 'f', 'h']
v : e
sommets_visités : ['g', 'e']
pile : ['g', 'e']
sommets_fermés : []

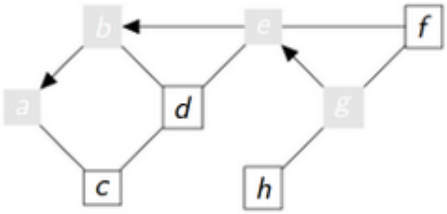
Voici les contenus des variables au second tour de la boucle tant que :



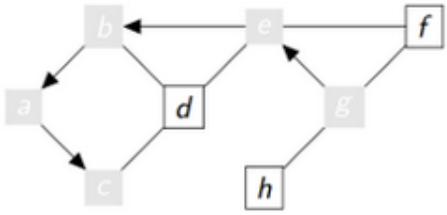
2nd tour
tmp : e
voisins : ['b', 'd', 'f']
v : b
sommets_visités : ['g', 'e', 'b']
pile : ['g', 'e', 'b']
sommets_fermés : []

À faire 1:

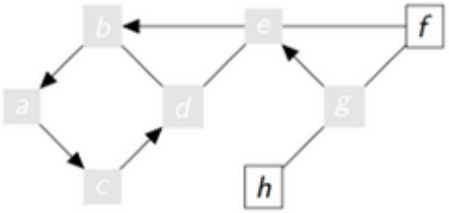
Compléter les contenus des variables:



3ème tour
tmp :
voisins :
v :
sommets_visités :
pile :
sommets_fermés :

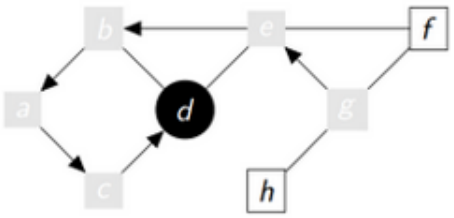


4ème tour
tmp :
voisins :
v :
sommets_visités :
pile :
sommets_fermés :



5ème tour
tmp :
voisins :
v :
sommets_visités :
pile :
sommets_fermés :

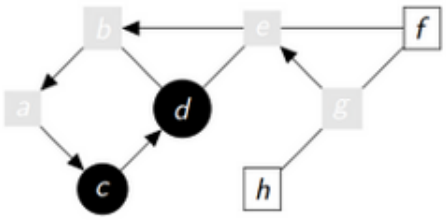
Voilà le contenu des variables au 6ème tour :



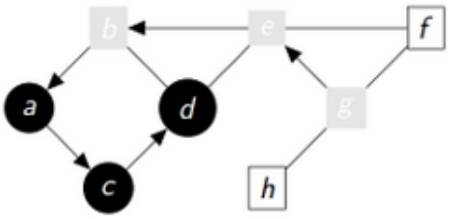
6ème tour
tmp : d
voisins : []
v : d
sommets_visités : ['g', 'e', 'b', 'a', 'c', 'd']
pile : ['g', 'e', 'b', 'a', 'c']
sommets_fermés : ['d']

À FAIRE 2:

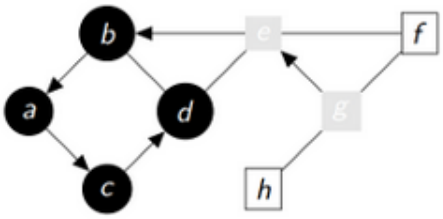
Poursuivez...



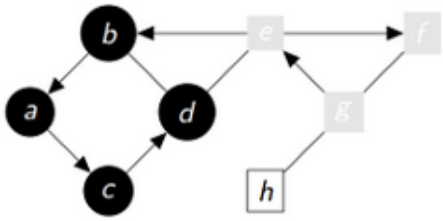
7ème tour
tmp :
voisins :
v :
sommets_visités :
pile :
sommets_fermés :



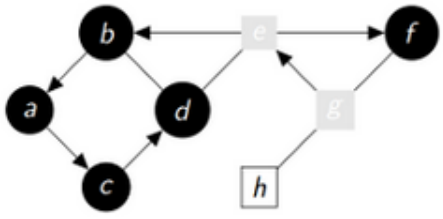
8ème tour
tmp :
voisins :
v :
sommets_visités :
pile :
sommets_fermés :



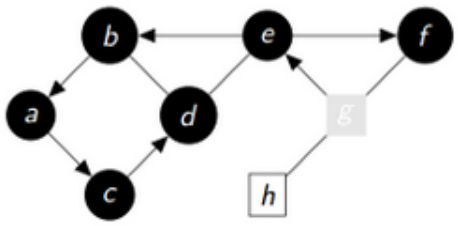
9ème tour
tmp :
voisins :
v :
sommets_visités :
pile :
sommets_fermés :



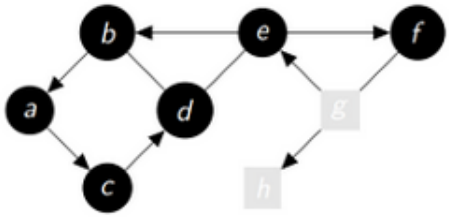
10ème tour
tmp :
voisins :
v :
sommets_visités :
pile :
sommets_fermés :



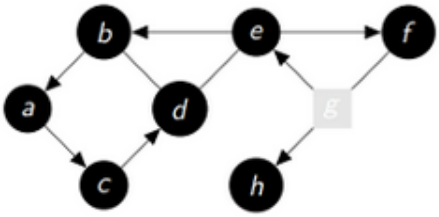
11ème tour
tmp :
voisins :
v :
sommets_visités :
pile :
sommets_fermés :



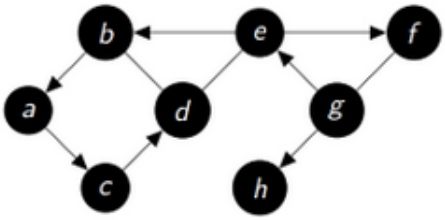
12ème tour
tmp :
voisins :
v :
sommets_visités :
pile :
sommets_fermés :



13ème tour
tmp :
voisins :
v :
sommets_visités :
pile :
sommets_fermés :



14ème tour
tmp :
voisins :
v :
sommets_visités :
pile :
sommets_fermés :

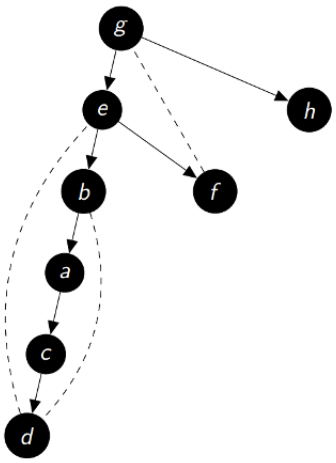


15ème et dernier tour
tmp :
voisins :
v :
sommets_visités :
pile :
sommets_fermés :

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

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

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



Implémentation 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)
```

Vous pouvez également utiliser votre propre classe Pile.
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 voisin(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
```

À FAIRE 3:

Implémenter cet algorithme en Python et tester le sur notre graphe G.
Remarque : Comme les choix dans la liste des voisins sont aléatoires, il y a plusieurs parcours possibles.

DFS - version 2

Voici une version sans utiliser de liste fermée.

```
def dfs_bis(G,sommet):
    p=Pile()
    sommets_visites=[]
    p.empiler(sommet)
    while p.vide()==False:
        tmp=p.depiler()
        if tmp not in sommets_visites:
            sommets_visites.append(tmp)
            print(tmp,end=" ")
        voisins=[y for y in voisin(G,tmp) if y not in sommets_visites]
        for vois in voisins:
            p.empiler(vois)
    return sommets_visites

c=dfs_bis(G,'g')
print('\n')
print(c)
```

À FAIRE 4:

tester cette version sur notre graphe G.

DFS - Version récursive

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

À FAIRE 5:

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

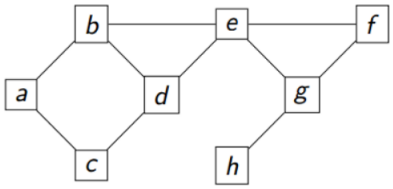
remarque : Le choix du 1er voisin est le premier de la liste voisins qui correspond à celle implantée lors de la création du graphe: `G['g'] = ['e','f','h']`, en la modifiant par `G['g'] = ['f','e','h']`, vous obtiendrez un autre parcours...

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.

Application : 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)

L'algorithme du DFS modifié

Données : G est un graphe
départ :(un sommet du graphe)
arrivee :(un sommet du graphe)
sommets_visités est une liste
parents = dict() (un dictionnaire)
parents[depart]=None
fonction dfs2(G,depart):
Si depart n'est pas dans la liste sommets_visités
alors
    On le met dans la liste
voisins ← la liste des voisins de depart non déjà visités
Pour chaque voisin dans voisins **faire**
    parents[voisin]=depart
    dfs2(G,voisin)
renvoyer parents

Cette fonction renvoie un dictionnaire qui contient les sommets visités(clés) et leurs parents(valeurs).
Il faut maintenant exploiter ce dictionnaire pour faire afficher un chemin entre deux sommets
C'est ce que réalise cette fonction qui prend en paramètre l'arrivée et le dictionnaire parents.

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

À FAIRE 6:

Implémenter l'algorithme en Python et la fonction Solution pour faire afficher un chemin entre les sommets b et h de notre graphe G.

À FAIRE 7:

Reprendre ce dernier travail en utilisant et modifiant la fonction dfs_bis.