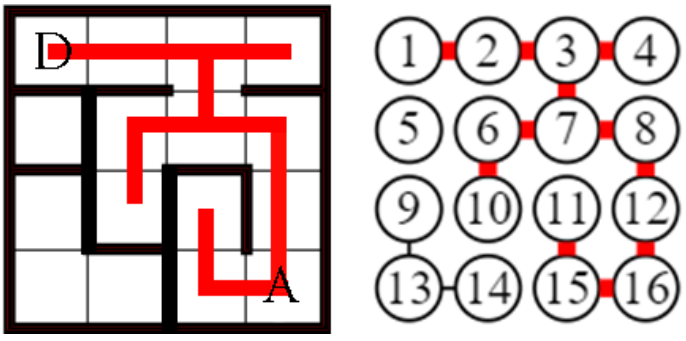
GRAPHES : Parcours en profondeur

- Dans ce document , nous allons découvrir un algorithme fondamental lié aux graphes, que nous avons déjà aperçu dans le cas des arbres.
- Le parcours en profondeur s'applique à n'importe quel graphe (connexe ou non, orienté ou non) et permet de déterminer tous les sommets atteignables depuis un sommet de départ.
- Il permet de déterminer s'il existe un chemin entre deux sommets donnés, la présence d'un cycle ou encore de savoir si le graphe est connexe.

- Ci-dessous :
  - On modélise un labyrinthe(à gauche) par un graphe non orienté(à droite) ou chaque "case" est un sommet ou les arcs indiquent les accès aux cases voisines .
  - En rouge , apparaît le résultat du parcours en profondeur du graphe à partir du sommet 1(case départ). Ce chemin passe par le sommet 16 (case d'arrivée).
  - Un autre parcours(parcours en largeur), qui sera abordé ultérieurement, permettra de déterminer le plus court chemin entre deux sommets.



1. Algorithme

Principes :

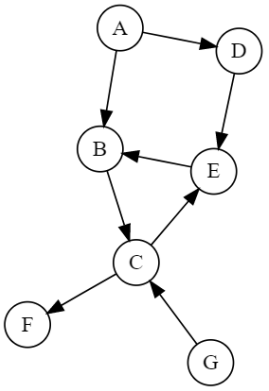
- On marque un sommet S dès qu'on le visite.
- On visite récursivement chacun des voisins de S en les marquant.
- Lorsqu'un sommet visité est déjà marqué, on remonte au sommet précédemment visité.

En d'autres termes :

- Dès que l'on trouve un voisin (peu importe lequel), on le visite.
- Quand il n'y a plus de voisins à visiter, on revient ensuite au dernier sommet visité.

Exemple :

Considérons le graphe orienté ci-contre.



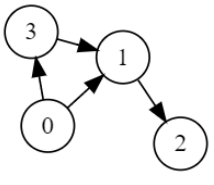
Le parcours en profondeur à partir du sommet A révèle les sommets A - B - C - E - F - D :

- Dans cet ordre.
- Ou encore dans cet ordre : A - D - E - B - C - F.

Le sommet G n'est pas découvert lors du parcours, car il n'y a pas de chemin qui relie A et G.

Exercice 1 :

Dans le graphe orienté ci-contre, indiquer un résultat du parcours en profondeur à partir de :



- 0 :
- 1 :
- 2 :
- 3 :

2. Implémentation

On utilise la classe déjà vue précédemment qui permet de représenter un graphe avec un dictionnaire d'adjacence. Le graphe créé ci-après est celui montré en exemple.

In [ ]:

```
class Graphe:
    '''Graphe représenté par un dictionnaire d'adjacence'''
    def __init__(self):
        self.adj={}

    def affiche(self):
        for k in self.adj:
            print (k,self.adj[k])

    def ajouter_sommet(self,s):
        if s not in self.adj:
            self.adj[s]=[]

    def ajouter_arc(self, s1, s2):
        self.ajouter_sommet(s1)
        self.ajouter_sommet(s2)
        self.adj[s1].append(s2)
        self.adj[s2].append(s1)

    def voisins(self, s):
        return self.adj[s]

    def sommets(self):
        s=[]
        for k in self.adj:
            s.append(k)
        return s

    #Ex2
    def parcours_profondeur(self,vus, s):
        pass

    #Ex3 (code à intégrer)




    #Ex4
    def existe_chemin(self,x,y):
        vus=[]
        pass


    def parcours_cycle(self,couleur,s):
        '''parcours en profondeur depuis le sommet s
        parametres:
        ----------
        couleur : dictionnaire des couleurs des sommets (NOIR=0, GRIS=1, BLANC=2)
        s :sommet de départ
        '''
        if couleur[s]==1:
            return True
        elif couleur[s]==0:
            return False
        else:
            couleur[s]=1
            for v in self.voisins(s):
                if self.parcours_cycle(couleur,v)==True:
                    return True
            couleur[s]=0
            return False

    #Ex5
    def cycle(self):
        couleur={}
        pass
```

```
    #Partie B 1.
    def parcours_chemin(self,vus,origine,s):
        if s not in vus:
            vus[s]=origine
            origine=s
            for v in self.voisins(s):
                self.parcours_chemin( vus, origine,v)

    #Partie B 2.
    def chemin(self,s1,s2):
        vus={}
        chem=[]


        return chem
```

In [ ]:
```
#graphe montré en exemple
g2=Graphe()

g2.ajouter_arc('A','B')
g2.ajouter_arc('A','D')
g2.ajouter_arc('B','C')
g2.ajouter_arc('C','E')
g2.ajouter_arc('C','F')
g2.ajouter_arc('D','E')
g2.ajouter_arc('E','B')
g2.ajouter_arc('G','C')
g2.affiche()
```

Avec une fonction récursive

- Les sommets marqués seront ajoutés dans une liste python passée en paramètre appelée `vus` .
- Principe : Si le sommet `s` n'est pas dans `vus` , l'y ajouter et parcourir récursivement tous ses voisins.

Exercice 2 :

Compléter la méthode `parcours_profondeur(self, vus,s)` ou `s` est le sommet de départ et `vus` la liste qui contient les sommets marqués.

In [ ]:
```
#test parcours profondeur
vus=[]
g2.parcours_profondeur(vus,'A')
print(vus) #['A','B','C','E','F','D']
```

Remarques :

- La fonction étant récursive , il y a un risque de dépassement de capacité(du fait du trop grand nombre d'appels récursifs) si le graphe contient beaucoup de sommets.
- Un autre possibilité d'implémentation est d'utiliser une autre structure de données déjà vue : la pile.

Avec une pile

On utilise la classe `Pile` ci-dessous, déjà vue dans un document précédent.

In [ ]:
```
class Pile:
    def __init__(self, valeurs=[]):
        self.valeurs=valeurs

    def est_vide(self):
        return self.valeurs == []

    def empiler(self,a):
        self.valeurs.append(a)

    def depiler(self):
        if self.est_vide() == False:
            return self.valeurs.pop()
```

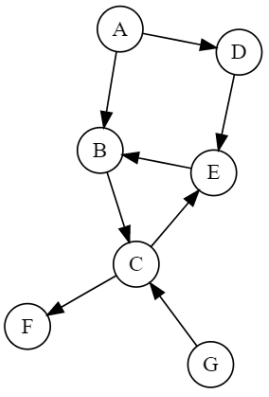
Exercice 3 :

Voici le code de la méthode qui permet d'effectuer ce parcours en profondeur

```
def parcours_profondeur_pile(self,s):
        vus=[]
        pile=Pile()
        pile.empiler(s)
        while not pile.est_vide():
            s=pile.depiler()
            if s not in vus:
                vus.append(s)
                for v in self.voisins(s):
                    pile.empiler(v)

        return vus
```

1. En déroulant ce code à la main , écrire la liste des sommets qui sera contenue dans la variable `vus` à la fin de ce parcours lorsque le départ est le sommet A sur le graphe donné en exemple(redonné ci-contre):



Réponse :

1. Intégrer ce code à la classe `Graphe_D` et vérifier la réponse

In [ ]:
```
#test parcours profondeur pile
g2.parcours_profondeur_pile('A')
```

3 . Applications

Chemins

- Une première application est que le parcours en profondeur permet de savoir s'il existe un chemin entre un sommet x et un sommet y.
- Il suffit pour cela d'effectuer un parcours en profondeur à partir de x. Si y est dans la liste des sommets atteints par ce parcours, alors le chemin de x vers y existe, sinon , il n'existe pas.

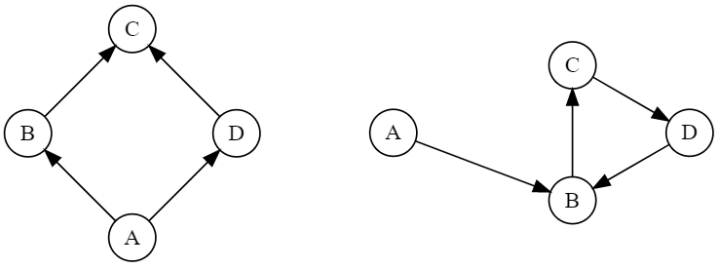
Exercice 4 :

Ecrire la méthode `existe_chemin(self, x, y)` qui prend en paramètres deux sommets x et y et qui renvoie `True` si le chemin de x vers y existe et `False` sinon.

In [ ]:
```
#test existe_chemin
assert(g2.existe_chemin('A','E')==True)
assert(g2.existe_chemin('A','G')==False)
```

Cycles

- Le parcours en profondeur permet également de détecter la présence d'un cycle dans un graphe orienté. La première idée est d'utiliser le fait que lors du parcours, on teste si le sommet a déjà été visité. Cela ne suffit cependant pas à justifier la présence d'un cycle.
- ##### Exemples :



  - Dans le graphe de gauche, le parcours en profondeur à partir de A fait que l'on passe deux fois par C, mais il n'y a pas de cycle.
  - Dans le graphe de droite, on passe deux fois par B et il y a un cycle(B - C - D - B).
- Pour pouvoir détecter un cycle, nous allons distinguer dans la liste des sommets marqués, deux sortes de sommets : ceux pour lesquels le parcours n'est pas encore terminé(il reste des voisins à explorer) et ceux pour lesquels il est terminé (tous les voisins ont été explorés).
- Il existe plusieurs solutions d'implémentation mais généralement, on utilise un marquage à trois "couleurs" :
  - BLANC pour les sommets non encore atteints.
  - GRIS pour les sommets en cours d'exploration(dont les voisins ne sont pas tous explorés).
  - NOIR pour les sommets dont le parcours est terminé(tousles voisins ont été explorés).
- Plus précisément, lorsque l'on visite un sommet :
  - S'il est gris, c'est que l'on vient de découvrir un cycle.
  - S'il est noir, on ne fait rien.
  - Sinon, c'est qu'il est blanc:
    - On colorie le sommet en gris.
    - On visite tous ses voisins, récursivement.
    - On colorie le sommet en noir.
- Ainsi, les voisins d'un sommet `s` sont explorés après que `s` ait été colorié en gris et avant qu'il ait été colorié en noir.

Exercice 5 :

- Dans la classe `Graphe` , la méthode `parcours_cycle(self,couleur,s)` détecte s'il y a un cycle à partir du sommet `s` . Elle suit la description de l'algorithme faite précédemment. `couleur` est ici un dictionnaire qui contient la couleur des différents sommets.
- Cependant, le fait qu'il n'y ait pas de cycle à partir d'un sommet `s` donné ne suffit pas à affirmer qu'il n'y en a pas dans le graphe.Il peut y en avoir un à partir d'un autre sommet.

Dans la classe `Graphe` , compléter la méthode `cycle(self)` qui renvoie `True` s'il y a un cycle dans la graphe et `False` sinon.

Aides :

- On appellera la méthode `parcours_cycle` à partir de tous les sommets du graphe.
- Au départ, tous les sommets sont BLANC.

Les tests sont effectués sur les graphes donnés en exemple précédemment.

In [ ]:
```
## construction des graphes donnés en exemple
g3=Graphe()
g3.ajouter_arc('A','B')
g3.ajouter_arc('A','D')
g3.ajouter_arc('B','C')
g3.ajouter_arc('D','C')
g3.affiche()
print('---')
g4=Graphe()
g4.ajouter_arc('A','B')
g4.ajouter_arc('B','C')
g4.ajouter_arc('C','D')
g4.ajouter_arc('D','B')
g4.affiche()
```

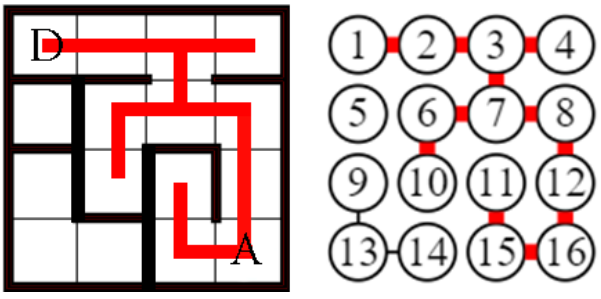
In [ ]:
```
# test cycle
assert(g3.cycle()==False)
assert(g4.cycle()==True)
```

4. Exercice de synthèse

ATTENTION :

- Dans cet exercice, on travaille avec un graphe non orienté.
- Pour pouvoir utiliser correctement les méthodes de la classe `Graphe` , il faut décommenter la ligne `#self.adj[s2].append(s1)` dans la méthode `ajouter_arc(self,s1,s2)` .

On reprend le labyrinthe présenté en introduction.



Partie A :

1. Construire et afficher le graphe associé à ce labyrinthe.

In [ ]:
```
#1.
laby=Graphe()



laby.affiche()
```

1. Vérifier qu'il existe un chemin entre la case départ et la case d'arrivée.

In [ ]:
```
#2.
```

1. Le graphe est-il connexe ? Justifier

In [ ]:
```
#3.
vus=[]


print(vus)
```

Réponse :

Partie B :

Dans cette partie, on cherche à utiliser le parcours en profondeur pour construire un chemin entre deux sommets du graphe, lorsque c'est possible.

Pour cela , on ajoute deux méthodes à la classe `Graphe` :

1. La méthode `parcours_chemin(self, vus, origine, s)` :

   - Elle ressemble beaucoup à la méthode `parcours_profondeur` .
   - Le paramètre `vus` est cette fois un dictionnaire qui associe à chaque sommet visité `s` , le sommet qui a permis de l'atteindre, `origine` .
   - Ainsi après le parcours, on pourra utiliser le contenu du dictionnaire pour remonter d'un sommet à un autre.

Cette méthode est déjà écrite dans la classe `Graphe` . Afficher le parcours à partir du sommet 1 (on utilisera `None` comme origine)

In [ ]:
```
#1.
vus={}

print(vus)
```

1. La méthode `chemin(self,s1,s2)` . Cette fonction renvoie un chemin entre s1 et s2 sous forme d'une liste de sommets.

Compléter cette méthode.

Aides :

- On lance un parcours en profondeur à partir du sommet s1.
- Si le sommet s2 a été atteint (c'est à dire s'il se trouve dans `vus`):
  - Construire le chemin dans une liste en remontant le dictionnaire de s2 à s1

In [ ]:
```
#test chemin entre case départ et case arrivée
laby.chemin(1,16) #[1, 2, 3, 7, 8, 12, 16]
```