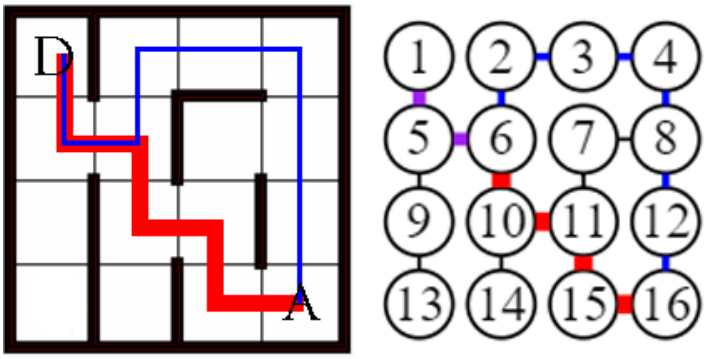
GRAPHES : Parcours en largeur

- Le parcours en profondeur étudié précédemment nous permet de déterminer l'existence d'un chemin entre deux sommets, voire d'en construire un, mais il ne détermine pas nécéssairement le plus court chemin entre deux sommets, c'est à dire leur distance. En effet le parcours en profondeur détermine un chemin arbitraire.
- Dans ce document, nous allons découvrir un autre algorithme fondamental lié aux graphes, que nous avons également déjà aperçu dans le cas des arbres : Le parcours en largeur.
- Il permet à l'instar du parcours en profondeur de découvrir les sommets atteignables à partir d'un sommet de départ, mais dans un ordre établi.
- Le parcours en largeur permet de déterminer le plus court chemin entre deux sommets d'un graphe (leur distance).
- 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 .
  - Il existe plusieurs chemins de la case départ à la case d'arrivée.
  - Le plus court a pour distance 6.



1. Algorithme

Principes :

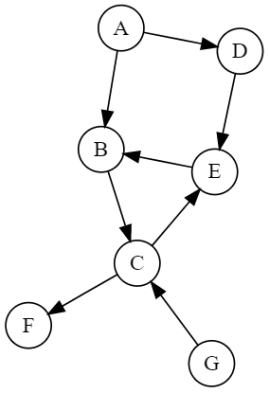
- Comme pour le parcours en profondeur, on se donne un sommet de départ (la source) pour intier le parcours.
- Et comme pour le parcours en profondeur, on va déterminer peu à peu tous les sommets atteignables à partir de ce sommet atteignable.
- Mais dans le parcours en largeur, on visite les sommets dans un ordre établi.
- D'abord les sommets à une distance 1 de la source, puis les sommets non encore visités situés à une distance 2, etc...Jusqu'à ce qu'il n'y ait plus de sommets à explorer.

Remarque :

- On mémorise ainsi la distance de chause sommet à la source, ce qui permettra ensuite de déterminer le plus court chemin d'un sommet à un autre.

Exemple :

Considérons le graphe orienté ci-contre.



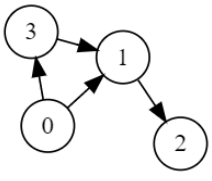
Le parcours en largeur à partir du sommet A révèle les sommets :

- Situés à une distance 1 de A : B et D.
- Puis situés à une distance 2 de A(et non encore visités) : C et E
- Puis situés à une distance 3 de A(et non encore visités) : 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 le résultat du parcours en largeur (en notant les distances de chasue sommet à la source entre parenthèses) 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)
        #graphe non orienté
        #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_largeur(self, source):
        dist={source:0}
        courant=[source]
        suivant=[]

        pass


        return dist

    #Ex3
    def parcours_largeur_file(self,source):
        dist={source:0}
        file=File()
        file.enfiler(source)


        pass


        return dist

    #Ex4
    def distance(self, s1,s2):

        pass




    #Ex5
    def parcours_chemin(self,source):
        vus={source:(0,None)}
```

```
        courant=[source]
        suivant=[]


        pass



        return vus

    def chemin(self,s1,s2):
        vus=self.parcours_chemin(s1)
        chem=[]

        pass

        chem.reverse()
        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 deux listes python et un dictionnaire.

- La premiere liste `courant` contient des sommets situés à une distance d de la source. C'est dans cette liste quel'on prend le prochaon sommet à examiner.
- La seconde liste `suivant` contient des sommets situés à une distance $d+1$ de la source, que l'on examinera après ceux de la liste `courant` .
- Le dictionnaire `dist` associe à chaque sommet atteint sa distance à la source.

- L'algorithme procède ainsi :
  - Initialement, la source est placée dans `courant` et dans `dist` associé à sa distance à elle-même qui est 0.
  - Tant que l'ensemble `courant` n'est pas vide :
    - On en retire un sommet `s` .
    - Pour chaque voisin `v` de `s` qui n'est pas encore dans `dist` :
      - On ajoute `v` à l'ensemble suivant.
      - On fixe `dist[v]` à `dist[s]+1` .
    - Si l'ensemble `courant` est vide, on l'échange avec `suivant` .
  - On renvoie `dist`

Exercice 2 :

En utilisant l'algorithme décrit ci-dessus, compléter la méthode `parcours_largeur(self, source)` ou `source` est le sommet de départ.

In [ ]:
```
#test parcours largeur
print(g2.parcours_largeur('A'))
```

Avec une file

- Le parcours en largeur est traditionnellement réalisé avec une file dans laquelle on ajoute à la fin les nouveaux sommets(ceux que le code précédent place dans `suivant`) et dans laquelle on retire au début le prochaon sommet à examiner (celui que le code précédent retire de l'ensemble `courant`).
- On utilise la classe `File` ci-dessous, déjà vue dans un document précédent.

In [ ]:
```
class File:

    def __init__(self, valeurs=[]):
        self.valeurs=valeurs

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

    def enfiler(self,a):
        self.valeurs.insert(0,a)

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

Exercice 3 :

Compléter la méthode `parcours_largeur_file(self, source)` et vérifier que le parcours en largeur donne bien le même résultat qu'avec la méthode précédente.

In [ ]:
```
#test parcours profondeur file
g2.parcours_largeur_file('A')
```

3 . Distance entre deux sommets

- Le parcours en largeur permet ainsi de calculer la distance entre deux sommets `s1` et `s2` , c'est à dire le plus court chemin qui les relie.
- On lance un parcours en largeur à partir du sommet `s1` .
- On renvoie la valeur de `dist[s2]` si `s2` est atteignable.

Exercice 4:

Compléter la méthode `distance(self, s1,s2)` qui renvoie la distance entre les sommets `s1` et `s2` .Si cette distance n'existe pas, on renvoie `None` .

In [ ]:
```
#test distance
assert(g2.distance('A','F')==3)
assert(g2.distance('A','A')==0)
assert(g2.distance('B','F')==2)
assert(g2.distance('A','G')==None)
```

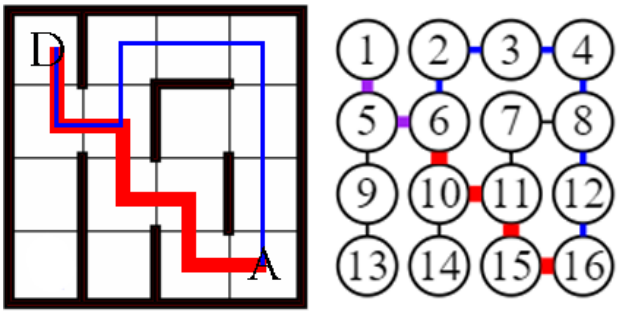
4. Exercice de synthèse

Exercice 5 :

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 que le plus court chemin entre le départ et l'arrivée est égal à 6.

In [ ]:
```
#2.
laby.distance(1,16)
```

Partie B :

Dans cette partie, on cherche à utiliser le parcours en largeur pour construire le plus court 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, source)` :
    - Elle ressemble beaucoup à la méthode `parcours_largeur` .
    - L'idée est de remplacer le dictionnaire `dist` par un dictionnaire `vus` de telle sorte que l'on associe à chaque sommet visité, en plus de la distance à la source, le sommet qui a permis de l'atteindre pendant le parcours en largeur. Pour le sommet source, on lui associe la valeur `None` . Ainsi les clés de ce dictionnaire sont les sommets et les valeurs associés peuvent un tuple contenant la distance à la source et le sommet origine.
    - Après le parcours, on pourra utiliser le contenu du dictionnaire pour remonter d'un sommet à un autre.

Compléter cette méthode dans la classe `Graphe` .

In [ ]:
```
#1.test parcours_chemin
print(laby.parcours_chemin(1))
```

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

Compléter cette méthode.

Aides :

- On lance un parcours en largeur 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 + court chemin entre case départ et case arrivée
laby.chemin(1,16) #[1, 5, 6, 10, 11, 15, 16]
```