Graphes

Nous avons déjà vu en Première la structure de graphe orienté et non orienté lorsque nous avons vu des exemples d'algorithmes gloutons (algorithme de Dijkstra)

Dans un premier temps on rappelle le vocabulaire

1 Vocabulaire

Un graphe **non orienté** est un ensemble d'éléments appelés **noeuds** ou **sommets** reliés ou non entre eux par des **arêtes**

Par exemple on peut regarder la carte du métro de Paris comme un graphe non orienté où les sommets sont les stations et les arêtes les connexions directes entre deux stations

Il y a une arête entre la station du Luxembourg et celle de Port-Royal, on dit aussi que les sommets Luxembourg et Port-Royal sont **voisins** mais il n'y a pas d'arête entre la station de

Luxembourg et celle de Rennes

Par contre il y a un **chemin** entre ces deux stations

Par exemple Luxembourg – Port Royal – Denfert – Raspail – Vavin – Montparnasse Bienvenue – Notre Dame des Champs – Rennes

On dit que ce chemin est de **longueur** 7 car il y a 7 arêtes

Existe -t-il un chemin de longueur moindre entre ces deux stations ? Nous reviendrons sur ce problème plus loin

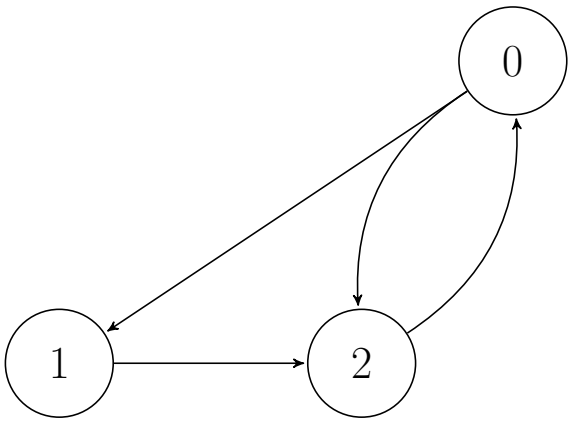
Lorsque entre deux sommets quelconque d'un graphe il existe au moins un chemin les reliant on dit que le graphe est **connexe**

Parfois le lien entre deux sommets nécessite une orientation, par exemple lorsque ce lien représente une rue en sens unique

Par exemple , voici une carte d'un quartier de Palaiseau où il y a des rues en sens unique

que l'on peut schématiser par un graphe **orienté**

Les liens orientés sont appelés des **arcs**



On peut parler ici du **chemin** $0 \to 1 \to 2$

Lorsqu'un chemin commence et se termine au même sommet on parle de **cycle**

Par exemple le cycle $0 \to 1 \to 2 \to 0$

La notion de cycle est valable aussi pour les graphes non orientés

2 Représentation des graphes en mémoire

Comment représenter un graphe en mémoire ?

En général les sommets du graphe sont numérotés de 0 jusqu' à $n-1$ quelque soit la nature du graphe

Ainsi si c'est le graphe associé aux stations de métro on va par l'intermédiaire d'un dictionnaire associer un entier à un nom de station

2.1 Liste de voisinage

On associe à chaque entier i représentant un sommet une liste des voisins de i

Par exemple pour le graphe orienté précédent , :

1. les voisins du sommet 0 sont les sommets 1 et 2 que l'on matérialise par la liste [1,2]

2. les voisins du sommet 1 est le sommet 2 que l'on matérialise par la liste [2]
3. les voisins du sommet 2 est le sommet 0 que l'on matérialise par la liste [0]

La liste de listes [[1,2], [2], [0]] contient l'information au sujet des arcs du graphe orienté

Ceci fonctionne aussi pour les graphes non orientés

2.2 Matrice d'adjacence

Dans un tableau à deux dimensions T on conserve l'information suivante :

j est voisin de i lorsque T[i][j] = 1, 0 sinon

Si le graphe est non orienté La relation de voisinage est symétrique et dans ce cas T[i][j] = T[j][i] on dit alors que la matrice est **symétrique**

Pour notre exemple ci-dessus le tableau T est :

[[0,1,1], [0,0,1], [1,0,0]]

2.3 Avantage et inconvénients des deux représentations

1. **Place en mémoire** : Plus le nombre de sommets est grand plus une représentation par matrice d'adjacence prendra plus de place que par liste d'adjacence
2. La requête **Est ce que j est voisin de i ?** prend moins de temps par une représentation par matrice d'adjacence que par liste d'adjacence

Néanmoins on préfère utiliser la plupart du temps une liste d'adjacence car en pratique la plupart des graphes sont peu "denses" dans le sens où il y a un grand nombre de sommets et la plupart des sommets ont peu de voisins relativement au nombre total de sommets

3 Interface d'un graphe non orienté

Voici une interface pour un graphe non orienté

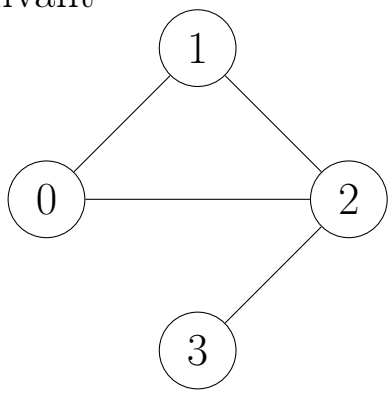
Interface d'un graphe non orienté
Opérations :
•`creer_graphe: entier` $\rightarrow$ `Graphe`
`creer_graphe(nb_sommets)` crée un graphe à `nb_sommets` sans arêtes
•`ajoute_arete: Couple d'Entiers` $\rightarrow \phi$
`ajoute_arête(i,j)` mémorise l'arête $i - j$
•`nb_arêtes: Graphe` $\rightarrow$ `Entier`
`nb_arêtes(G)` retourne le nombre d'arêtes du Graphe G
•`voisins: Entier` $\rightarrow$ `Liste d'Entiers`
`voisins(i)` retourne la liste des voisins de i

4 Implémentation en Python

```
class Graphe_NO:
    def __init__(self,nb_sommets:int) :
        self.nb_sommets = nb_sommets
        self.nb_aretes = 0
        self.aretes = [[] for i in range(self.nb_so

    def ajoute_arete(self,u:int,v:int) -> None:
        self.nb_aretes += 1
        self.aretes[u].append(v)
        self.aretes[v].append(u)

    def voisins(self,i : int) -> list:
        return self.aretes[i]

    def __str__(self) -> str:
```

```
        ch = str(self.nb_sommets)+" sommets "+\
        str(self.nb_aretes)+" aretes \n"
        for i in range(self.nb_sommets):
            ch = ch + str(i) + " -- "
            for sommet in self.aretes[i]:
                ch = ch + str(sommet) + " "
            ch = ch +"\n"
        return ch
```

Voici un exemple d'utilisation pour le graphe non orienté suivant




```
g = Graphe_NO(4)
g.ajoute_arete(0,1)
g.ajoute_arete(0,2)
g.ajoute_arete(2,1)
g.ajoute_arete(2,3)
print(g)
```

On obtient dans la console

```
4 sommets 4 aretes
0 -- 1 2
1 -- 0 2
2 -- 0 1 3
3 -- 2
```

5 Interface d'un graphe orienté

Voici une interface pour un graphe orienté

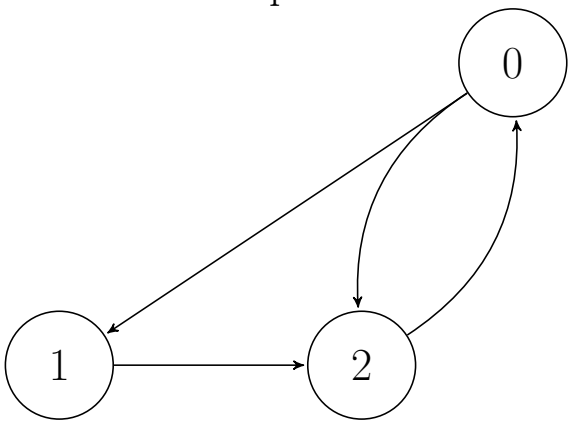
Interface d'un graphe orienté
Opérations : •`creer_graphe: entier` $\rightarrow$ `Graphe` `creer_graphe(nb_sommets)` crée un graphe à `nb_sommets` sans arcs •`ajoute_arc: Couple d'Entiers` $\rightarrow \phi$ `ajoute_arc(i,j)` mémorise l'arc $i \rightarrow j$ •`nb_arcs: Graphe` $\rightarrow$ `Entier` `nb_arcs(G)` retourne le nombre d'arcs du Graphe G •`voisins: Entier` $\rightarrow$ `Liste d'Entiers` `voisins(i)` retourne la liste des voisins de i

6 Implémentation en Python

```
class Graphe_O:
    def __init__(self,nb_sommets):
        self.nb_sommets = nb_sommets
        self.arcs = 0
        self.arcs =  [[]for i in range(nb_sommets)]

    def ajoute_arcs(self,i,j):
        self.arcs[i] .append( j)
        self.arcs += 1

    def voisins(self,i):
        return self.arcs[i]

    def __str__(self):
        ch = str(self.nb_sommets)+" sommets "+str(s
```

```
        for i in range(self.nb_sommets):
            ch = ch + str(i) + " --> "
            for sommet in self.arcs[i]:
                ch = ch + str(sommet) + " "
            ch = ch +"\n"
        return ch
```

Voici un exemple d'utilisation pour le graphe orienté suivant




```
g = Graphe_O(3)
g.ajoute_arc(0,1)
g.ajoute_arc(0,2)
g.ajoute_arc(2,0)
g.ajoute_arc(1,2)
print(g)
```

On obtient dans la console

```
3 sommets 4 arcs
0 --> 1 2
1 --> 2
2 --> 0
```

7 Algorithmes

Voici quelques problèmes qui peuvent être posés concernant un graphe :

1. Existe-t-il un chemin reliant deux sommets i et j donnés ?
2. Quels sont tous les sommets accessibles à partir de i ?
3. Le graphe est il connexe ?
4. Existe-t-il un cycle dans le graphe ?
5. Quels sont les longueurs des chemins entre i et j donnés ?

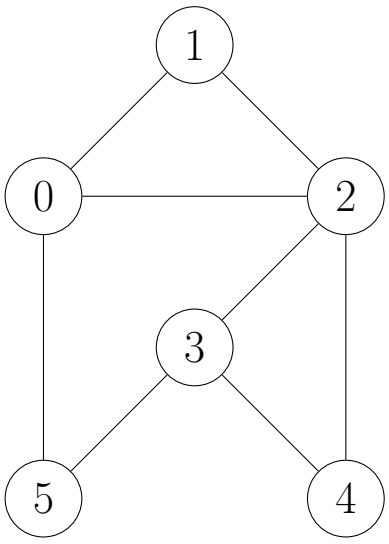
7.1 Parcours en profondeur dans un graphe

On adapte le parcours en profondeur vu pour les arbres aux graphes afin d'éviter de tourner en rond **en marquant les sommets déjà visités** à l'aide d'un tableau de booléens

Algorithme 1 : Parcours en profondeur d'un graphe G au départ d'une source s

```
   parcours_profondeur (G,s)
   début
   |  Données : Une source s
   |  Résultat : Rien
1  |  marquer(s)
2  |  afficher(s)
3  |  pour chaque voisin de s non marqué faire
4  |  |  parcours_profondeur (G,voisin)
5  |  fin
   fin
```

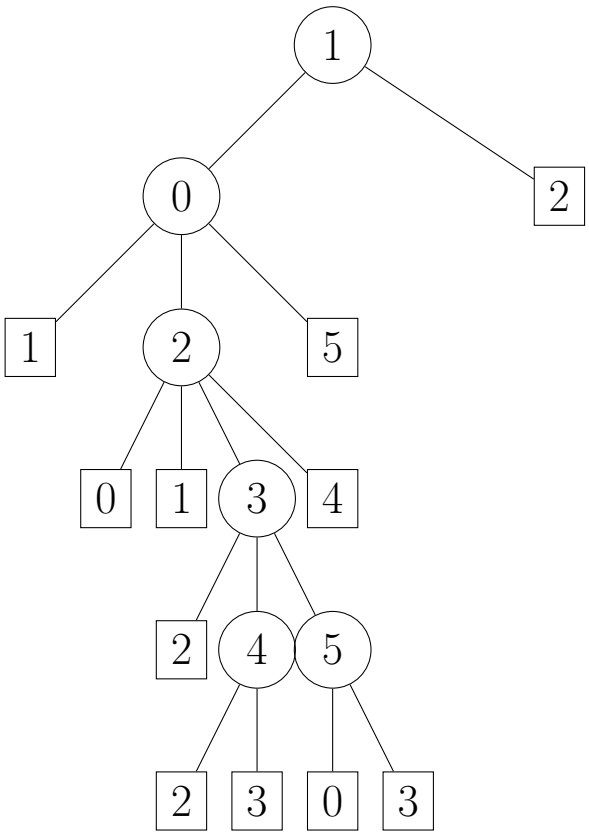
Par exemple



On suppose que ce graphe est représenté par une liste d'adjacence [[1,2,5], [0,2],[0,1,3,4],[2,4,5],[2,3],[0,3]]

Si on part de la source 1 on visualise une partie de l'arbre des appels récursifs par l'arbre ci-dessous où les noeuds entourés d'un carré ne font pas partie de l'arbre mais aide à la compréhension

Ainsi lorsque la fonction `parcours_profondeur` est appelée sur le sommet 0, le sommet 1 a déjà été visité c'est pour cela qu'il est entouré d'un carré et qu'il n'y aura pas d'appel récursif de `parcours_profondeur` sur le sommet 1 mais sur le sommet 2 qui lui est un voisin du sommet 0 pas encore visité



Par conséquent le parcours en profondeur en partant de 1, avec la liste d'adjacence fournie, est :

1 - 0 - 2 - 3 - 4 - 5

Attention ! Dans certaines situations un graphe peut être une structure de données assez volumineuse et il serait maladroit de répéter des parcours en profondeurs pour chaque requête concernant le graphe en partant de la source s

Il est préférable de faire une fois pour toutes le traitement d'où la nécessité de créer une interface spécifique pour le parcours en profondeur sur un graphe en partant d'une source s **différente** de l'interface du graphe

Voici une implémentation en Python (PP pour parcours en profondeur) qui est valable à la fois pour les graphes orientés et non orientés

```
class PP:
    """
```

```
    valable pour G orienté ou non orienté
    """
    def __init__(self,graphe,source):
        self.visites = [False]*graphe.nb_sommets
        self.ancetres = [0]*graphe.nb_sommets
        self.source = source
        self.pp(graphe,source)

    def pp(self,graphe,v):
        self.visites[v] = True
        for w in graphe.voisins(v):
            if not self.visites[w]:
                self.ancetres[w] = v
                self.pp(graphe,w)

    def pp_iter(self,graphe,v):
        self.visites[v] = True
        pile = [v]
        while len(pile) != 0:
            x = pile.pop()
            for w in graphe.voisins(x):
                if not self.visites[w]:
                    self.visites[w] = True
                    self.ancetres[w] = x
                    pile.append(w)

    def a_un_chemin_vers(self,v):
        return self.visites[v]

    def chemin_vers(self,v):
        """
```

```
        retourne le chemin s'il existe de s vers v
        forme d'une liste d'entiers
        s'il n'existe pas retourne une liste vide
        """

        if not(self.visites[v]):
            return []
        chemin = [v]
        sommet = v
        while sommet != self.source:
            sommet = self.ancetres[sommet]
            chemin.insert(0,sommet)
        return chemin
```

Applications

1. Connexité d'un graphe , composantes connexes d'un graphe
2. Détection de cycle dans un graphe

Pour les composantes connexes voir Exercices

7.2 Détection d'un cycle dans un graphe orienté

On veut définir une fonction qui détecte le premier cycle rencontré dans un graphe orienté et retourne Vrai

On ne peut plus utiliser un tableau pour marquer les sommets visités

Par contre au lieu de considérer qu'un sommet a deux états, visité ou non visité on associe à un sommet trois états ou couleurs :

1. BLANC ou non visité
2. GRIS , visité mais l'exploration des voisins est en cours

3. NOIR visité et l'exploration des voisins est terminée

On détecte un cycle quand dans un parcours en profondeur, un sommet w GRIS a un de ses voisins v GRIS (cela signifie que "l'on revient sur ses pas")

On ajoute deux méthodes à la classe Graphe orienté

1. Une méthode **interne** `_a_un_cycle(v)` qui fait un parcours en profondeur à partir d'un sommet v et qui retourne vrai s'il détecte un cycle
2. Une méthode `a_un_cycle()` qui appelle la précédente sur chaque sommet de couleur BLANC

```
def _a_un_cycle(self,s:int)->bool:
    if self.couleurs[s] == GRIS:
        return True
    if self.couleurs[s] == NOIR:
        return False
    self.couleurs[s] = GRIS
    for v in self.voisins(s):
        #en profondeur
        if self._a_un_cycle(v):
            return True
    self.couleurs[s] = NOIR
    return False

def a_un_cycle(self):

    for s in range(self.nb_sommets):
        if  self.couleurs[s] == BLANC and self.
            return True
    return False
```

7.3 Détection de cycle dans un graphe non orienté

On ne peut plus utiliser la technique précédente car une arête entre deux sommets serait considérée comme un cycle

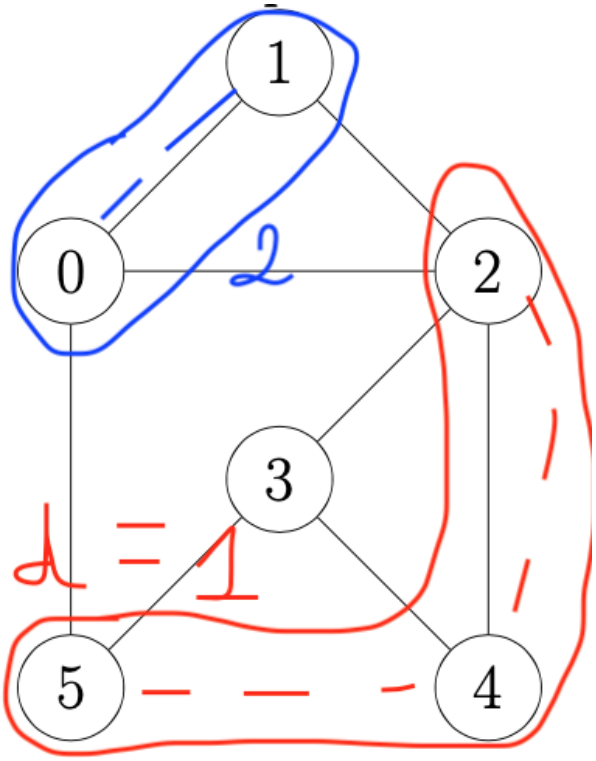
Pour empêcher cela on fait en sorte qu'il n'y a pas de retour en arrière et on ajoute les méthodes suivantes à la classe Graphe non orienté

```
def _a_un_cycle(self,s:int) -> bool:
    self.couleurs[s] = GRIS
    for v in self.voisins(s):
        self.peres[v] = s
        if v != self.peres[s] and self.couleurs
            return True
        if v != self.peres[s] and self.couleurs
            return False
        #en profondeur
        if v != self.peres[s] and self.couleurs
            return self._a_un_cycle(v)
    self.couleurs[s] = NOIR


def a_un_cycle(self) -> bool:
    for s in range(self.nb_sommets):
        if  self.couleurs[s] == BLANC and self.
            return True
    return False
```

7.4 Parcours en largeur

Cette fois ci on veut **avancer en partant de la source en vagues concentriques**

Par exemple sur le graphe ci-dessous la source est le sommet 3 qui est à distance 0 de lui-même



1. Ensuite on visite les sommets à distance 1 de la source autrement dit les sommets 2,4 et 5 qui sont les voisins de la source 3
2. Puis on visite les sommets à distance 2 de la source , c'est à dire les sommets 0 et 1

Comment faire cela ?

On l'a déjà vu pour les arbres on utilise une file et une liste de booléens pour marquer les sommets visités, d'où l'algorithme

Algorithme 2 : Parcours en largeur d'un graphe à partir d'une source

```
parcours_largeur (G,s)
début
    Données : Un graphe G, une source s un entier
    Résultat : Rien
1   f = file_vide()
2   enfiler(f,s)
3   marquer(s)
4   tant que non (est-vide(f)) faire
5       x = defiler(f)
6       afficher(x)
7       pour chaque voisin de x non marqué faire
8           enfiler(f,voisin)
9           marquer(voisin)
10      fin
11  fin
fin
```

Comme pour le parcours en profondeur on va définir une structure différente de celle de graphe

Voici une implémentation en Python (PL pour parcours en largeur) valable à la fois pour les graphes orientés et non orientés

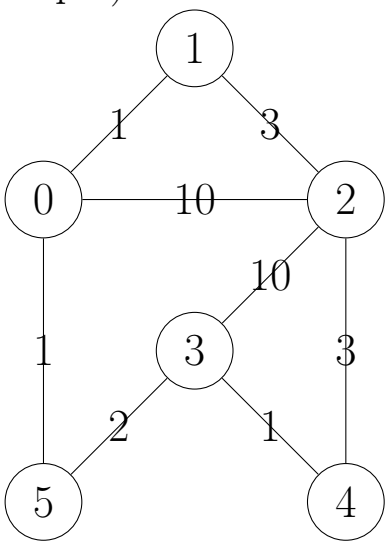
```
class PL:
    """
    pour graphe orienté et non orienté
    """
    def __init__(self,graphe,source):
        self.visites = [False]*graphe.nb_sommets
        self.peres = [source]*graphe.nb_sommets
        self.source = source
```

```
        self.pl(graphe,source)

    def pl(self,graphe,v):
        self.visites[v] = True
        file = [v]
        while len(file) != 0:
            x = file.pop()
            for w in graphe.voisins(x):
                if not self.visites[w]:
                    self.visites[w] = True
                    self.peres[w] = x
                    file.insert(0,w)

    def a_un_chemin_vers(self,v):
        return self.visites[v]

    def chemin_vers(self,v):
        """
        retourne le chemin s'il existe de s vers v
        forme d'une liste d'entiers
        s'il n'existe pas retourne une liste vide
        """
        if not(self.visites[v]):
            return []
        chemin = [v]
        sommet = v
        while sommet != self.source:
            sommet = self.peres[sommet]
            chemin.insert(0,sommet)
        return chemin
```

```
def distance(self,v):
    """
    Le parcours en largeur détermine les chemin
    entre la source et un sommet visité
    """
    ch = self.chemin_vers(v)
    if len(ch) == 0:
        return inf
    return len(ch) - 1
```

Le parcours en largeur donne le chemin le plus court de la source à un sommet donné (si ce chemin existe)

On peut généraliser au cas où le graphe est **pondéré** c'est à dire le cas où les arêtes ont un poids positif : (une distance par exemple)



Précédemment le parcours en largeur avait trouvé comme chemin le plus court de la source 3 au sommet 2 le chemin 3 -> 2, ici ce sera 3 -> 4 -> 2 pour une distance de 4 alors que le chemin 3 -> 2 a une distance de 10

On modifie l'algorithme de parcours en largeur en utilisant une **file de priorité minimale** la clé étant la distance minimale d à la source s sous la forme d'un tableau de taille égale

au nombre de sommets

8 File de priorité

Une file de priorité est une structure de données qui permet de gérer un ensemble d'éléments F dont chacun a une valeur associée baptisée **clé**

Par exemple les processus en mode Prêt sont gérés dans une file de priorité

Une **file de priorité min** reconnaît les opérations suivantes :

1. `insérer(x,F)` l'élément x dans l'ensemble F
2. `minimum(F)` retourne l'élément de F ayant la clé minimale
3. `supprimer_min(F)` supprime et retourne l'élément de F ayant la clé minimale
4. `diminuer_clé(F,x,k)` diminue la clé de l'élément x en lui attribuant une nouvelle valeur k inférieure à la précédente

On va modifier la classe **Tas min** (vu en TP) pour implémenter une classe **FP min** (file de priorité min)

On a presque toutes les fonctions ci-dessus sauf celle qui permet de diminuer la valeur d'une clé d'un élément de la file de priorité

```
def echanger(t,i,j):
    temp = t[i]
    t[i] = t[j]
    t[j] = temp


class Tas_min:
    """
    arbre binaire essentiellement complet rangé dan
    de l'indice 1 jusqu'à l'indice taille = n
```

```
    le fils gauche de i est en 2*i
    le fils droit de i est en 2*i + 1
    T[i] <= T[2i] et T[i] <= T[2i+1] pour 1 <= i <=

    Un tas est créé à partir d'une liste en tamisan
partir des noeuds
    en descendant
    """

    def __init__(self,n:int):
        self.taille = n
        self.tableau = [n]

    def tamiser(self,i):
        j = i
        while 2*j <= self.taille:
            k = 2*j
            if k < self.taille and self.tableau[k+1
                k = k+1
            if self.tableau[j] > self.tableau[k]:
                echanger(self.tableau,j,k)
            else:
                break
            j = k

    def percoler(self,i):
        j = i
        while j > 1 and self.tableau[j] < self.tabl
            echanger(self.tableau,j,j//2)
            j = j//2
```

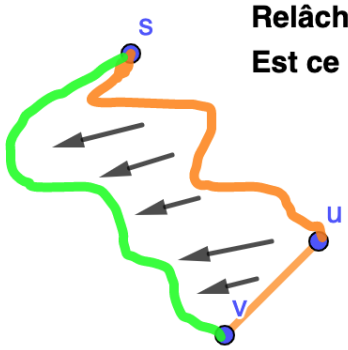
```
def cree_tas(self,tab:list):
    for elt in tab:
        self.tableau.append(elt)
    for i in range(self.taille//2,0,-1):
        self.tamiser(i)

def get_min(self):
    return self.tableau[1]

def inserer(self,k):
    """
    on insère la clé à la fin du tas
    ce qui probablement viole la structure du t
    Il faut donc percoler cette clé afin que l'
    la structure du tas
    """
    self.taille += 1
    self.tableau[0] = self.taille
    self.tableau.append(k)
    self.percoler(self.taille)

def supprimer_min(self):
    """
    On échange le sommet d'indice 1 (le minimum
    élément
    On tamise le sommet
    On renvoie le minimum
    """
    echanger(self.tableau,1,
    self.taille)
    self.taille -= 1
```

```
        self.tamiser(1)
        self.tableau[0] = self.taille
        return self.tableau.pop()
```

Quelles modifications ?

1. Dans la file il y avait des entiers, dans la file de priorité il y aura des objets. Ce peut être une liste dont le premier élément est la clé pour faire les comparaisons. Le deuxième élément est un sommet du graphe, la clé est la distance de ce sommet à la source
2. A l'initialisation on va insérer dans la file **tous** les objets de telle sorte que la clé des objets vaut $+\infty$ et la clé de la source vaut 0 et on va le faire avec la méthode `cree_tas(liste)` de telle sorte que l'on n'a pas bes
3. Ensuite à chaque tour de boucle on va extraire de la file la racine qui est l'élément dont la clé est minimale jusqu'à ce que la file soit vide. Au début ce sera la source avec la clé 0 mais ensuite puisque les valeurs des clés vont diminuer ce sera d'autres sommets

   Imaginons que l'on extrait l'objet [d[u],u], on note d[u] la clé associé au sommet u, c'est la distance de la source à u
4. On va ensuite essayer de diminuer les clés des voisins v de u qui sont encore dans la file de priorité, par le processus de relâchement, illustré ci-dessous



Relâchement :
Est ce que le fil s-u-v peut prendre la place du fil s-v ?

Autrement dit si $d[v] > d[u] + l(u, v)$ alors la nouvelle clé de v (plus petite) est $d[v] = d[u] + l(u, v)$ où $l(u, v)$ est la longueur de l'arc ou arête u-v

5. Lorsqu'on extrait u on a les voisins v de u. Pour pouvoir faire le processus de relâchement on doit savoir si v est dans la file ou pas et si c'est la cas à quel indice i du tas pour avoir accès à d[v]

   Ceci peut être fait avec une liste `position` de telle sorte que `position[v] = i`, où v est un sommet et i est l'indice où on trouve [d[v],v] dans le tas min

   Si d[v] a été diminué alors il faudra percoler l'élément d'indice i du tas pour rétablir la structure de tas min

   Ce faisant l'indice i va changer, il faut donc mettre à jour la liste `position` régulièrement à chaque échange de position dans le tas dans la méthode `percoler`

6. D'où la classe FPmin

```
class FP_min:
    """
    une file de priorité min est un tas min
    contenant des objets comparables
    par exemple des processus
    Ces objets ont un attribut permettant
    de les comparer entre eux
```

```
    Ils sont donc rangés dans un tas min
    selon cet attribut
    Au cours du temps les attributs peuvent
    changer et par conséquent ces
    objets vont changer de place
    dans la file de priorité
    Ici un objet sera une liste [d(i,s),i]
    où i est le numéro du
    sommet d'un graphe 0 <= i <= n

    En procédant ainsi [d,i] < [d',j]
    quand d < d'
    """
    def __init__(self,n:int):
        self.taille = n
        self.tableau = [n]
        self.position = [0]*self.taille

    def tamiser(self,i):
        j = i
        while 2*j <= self.taille:
            k = 2*j
            if k < self.taille and \
            self.tableau[k+1] < self.tableau[k]:
                k = k+1
            if self.tableau[j] > self.tableau[k]
                echanger(self.tableau,j,k)
                self.position[self.tableau[k][1]
                self.position[self.tableau[j][1]
            else:
                break
```

```
            j = k

    def percoler(self,i):
        j = i
        while j > 1 and \
        self.tableau[j] < self.tableau[j>>1] :
            echanger(self.tableau,j,j>>1)
            self.position[self.tableau[j][1]] =
            self.position[self.tableau[j>>1][1]]
            j = j>>1

    def cree_file_min(self,tab:list):
        """
        On insère tous les sommets avec une clé
        la source avec une clé nulle
        """
        for elt in tab:
            self.position[elt[1]] = len(self.tab
            self.tableau.append(elt)
        for i in range(self.taille//2,0,-1):
            self.tamiser(i)

    def get_min(self):
        return self.tableau[1]


    def supprimer_min(self):
        """
        On échange le sommet d'indice 1 (le mini
        et le dernier élément
        On tamise le sommet
```

```
            On renvoie le minimum
            """
            echanger(self.tableau,1,
            self.taille)
            self.position[self.tableau[1][1]] = 1
            #-1 signifie que l'élément est sorti de
            self.position[self.tableau[self.taille]
            self.taille -= 1
            self.tamiser(1)
            self.tableau[0] = self.taille
            return self.tableau.pop()

        def diminuer_cle(self,v,n_cle):
            self.tableau[self.position[v]][0] = n_cl
            self.percoler(self.position[v])
```

7. D'où l'algorithme de Dijkstra vu comme un parcours en largeur d'un graphe pondéré

Algorithme 3 : Algorithme de Dijkstra

```
dijkstra (G,L,s)
début
    Données : Un graphe G, une tableau L de longueur
              des arêtes, une source s
    Résultat : Rien
1   f = creer file de priorité(G)
2   tant que non (est-vide(f)) faire
3       x = supprimer_min(f)
4       pour chaque voisin v de x faire
5           relacher(x,v,L)
6       fin
7   fin
fin
```

Voici une implémentation en Python

```
class Dijkstra:
    """
    L est une liste de longueur des arêtes ou des a
    """
    def __init__(self,graphe,L,source):
        self.source = source
        self.ancetres = [source]*graphe.nb_sommets
        self.solutions = {}
        self.dijkstra(graphe,L,source)

    def relacher(self,x,v,L,file_min):
        """
        Ici x est un objet de clé minimal de type [
        """
        d_x = x[0]
```

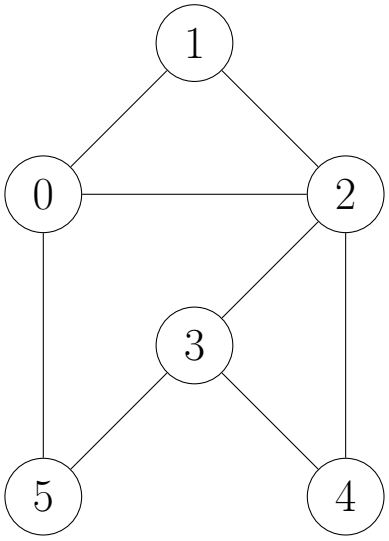
```
        #on ne relache pas sur un voisin
        #déjà sorti de la file
        if file_min.position[v] > 0:
            d_v = file_min.tableau[file_min.positio
            n_cle =  d_x + L[x[1]][v]
            if n_cle < d_v:
                self.ancetres[v] = x[1]
                file_min.diminuer_cle(v,n_cle)



    def dijkstra(self,graphe,L,source):
        liste = [[inf,i] for i in range(graphe.nb_s
        liste[source][0] = 0

        file_min = FP_min(graphe.nb_sommets)
        file_min.cree_file_min(liste)
        while len(file_min.tableau) > 1:
            x = file_min.supprimer_min()
            self.solutions[x[1]] = x[0]
            for v in graphe.voisins(x[1]):
                self.relacher(x,v,L,file_min)


    def dist(self,sommet):
        return self.solutions[sommet]

    def a_un_chemin_vers(self,v):
        return self.dist(v) < inf

    def chemin_vers(self,v):
```

```
    """
    retourne le chemin s'il existe de s vers v
    forme d'une liste d'entiers
    s'il n'existe pas retourne une liste vide
    """
    if not(self.a_un_chemin_vers(v)):
        return []
    chemin = [v]
    sommet = v
    while sommet != self.source:
        sommet = self.ancetres[sommet]
        chemin.insert(0,sommet)
    return chemin
```

9 Exercices

Ex 1

Le **degré** d'un sommet d'un graphe **non orienté** est le nombre de voisins de ce sommet

1. 

   Donner les degrés de chaque sommet du graphe ci-dessus

2. Implémenter en Python une méthode `degre(i)` qui retourne le degré du sommet i
3. Implémenter en Python une méthode `max_degre()` qui retourne le degré maximal du graphe
4. Justifier que le degré moyen d'un graphe non orienté vérifie : $\overline{d} = \dfrac{2|\text{arêtes}|}{|\text{sommets}|}$

Ex 2

Si G est un graphe orienté, le graphe inverse de G noté $\overline{G}$ est le graphe ayant les même sommets que G mais dont les arcs sont orientés dans le sens inverse que ceux de G

Implémenter une méthode `inverse()` dans la classe `Graphe` qui crée le graphe inverse de self

Ex 3

Définir un algorithme **itératif** `parcours_profondeur(s)`

Ex 4

Lire `https://xkcd.com/761/`

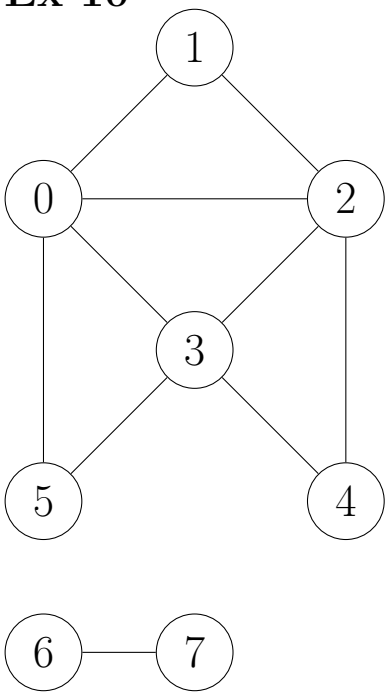
Ex 5

Implémenter la classe `CC` dont le constructeur à partir d'un graphe donné G détermine les composantes connexes de G non orienté en associant à chaque sommet d'une même composante connexe le même identifiant (un entier)

Ex 9

Implémenter dans la classe `PL` la méthode `nb_sauts(v)`

Ex 10



En supposant que ce graphe est représenté par la liste d'adjacence [[1,2,3,5], [0,2],[0,1,3,4],[0,2,4,5],[2,3],[0,3],[7],[6]]

1. Donner l'affichage après un parcours en profondeur en partant de 0, puis de 3, puis de 6
2. Donner l'affichage après un parcours en largeur en partant de 0, puis de 3, puis de 6

Ex 11

1. Implémenter les trois dernières méthodes de la classe Dijkstra

2. Faire tourner à la main l'algorithme sur l'exemple du cours

Défis

Proposer **votre** solution personnelle même imparfaite à :

1. La **coloration d'un graphe avec deux couleurs** de telle sorte que tout lien se fait entre deux sommets de couleur différente
2. Un **pont** est une arête telle que si elle est enlevée le graphe n'est plus connexe . Un **point d'articulation** est un sommet tel que s'il est enlevé ainsi que toutes les arêtes associées alors le graphe n'est plus connexe. Détecter les ponts (et ou les points d'articulation)

(Ne regarder ni les livres ni le Web)