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

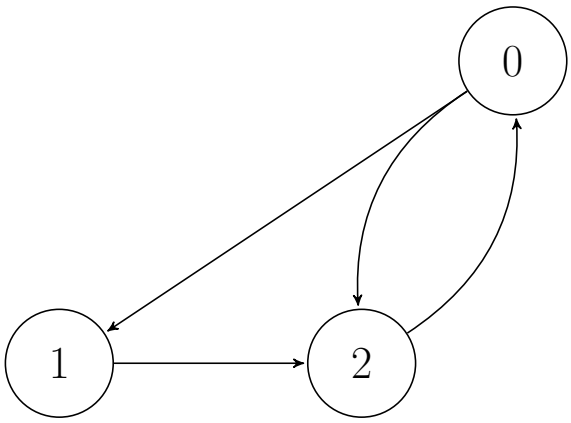
Lorsque entre deux sommets quelconque d'un graphe il existe au moins un chemin les reliant on dit que le graphe est **connexe**

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 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

4 Implémentation en Python

```
class Graphe:
    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):
         pass

     def __str__(self):
        pass
```

5 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 ?

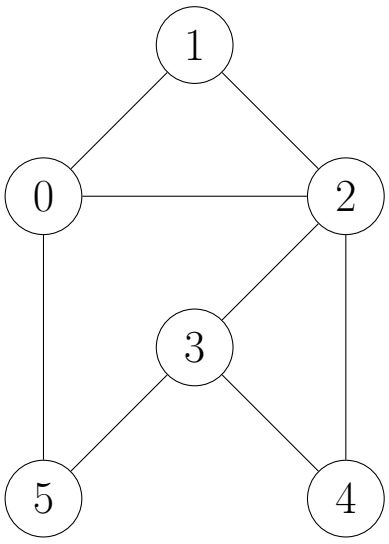
5.1 Parcours en profondeur

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  |  si non (s est vide) alors
4  |  |  pour chaque voisin de s non marqué faire
5  |  |  |  parcours_profondeur (G,voisin)
6  |  |  fin
7  |  fin
   fin
```

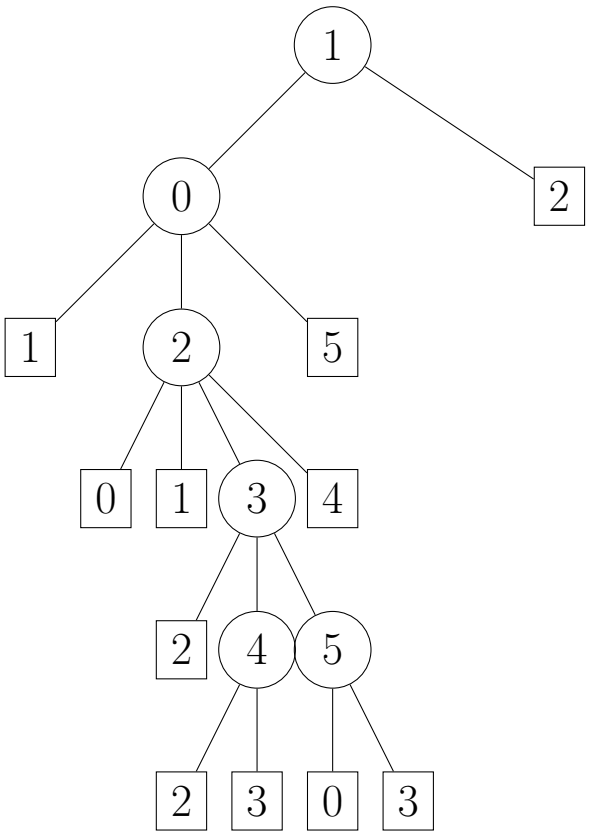
Par exemple



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 orienté

Voici une implémentation en Python (PP pour parcours en profondeur)

```
class PP:
    def __init__(self,G,s):
        self.visites = [False]*G.nb_sommets
```

```
        self.source = s
        self.peres = [s]*G.nb_sommets
        self.pp(G,s)

    def pp(self,G,s):
        self.visites[s] = True
        for v in G.voisins(s):
            if not(self.visites[v]):
                pp(G,v)
                self.peres[v] = s

    def a_un_chemin_vers(v):
        """
        retourne vrai si il y a
        un chemin de s vers v
        """
        pass

    def chemin_vers(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
        """
        pass
```

Application : Détection de cycle dans un graphe

On veut définir une fonction qui détecte le premier cycle rencontré et retourne Vrai

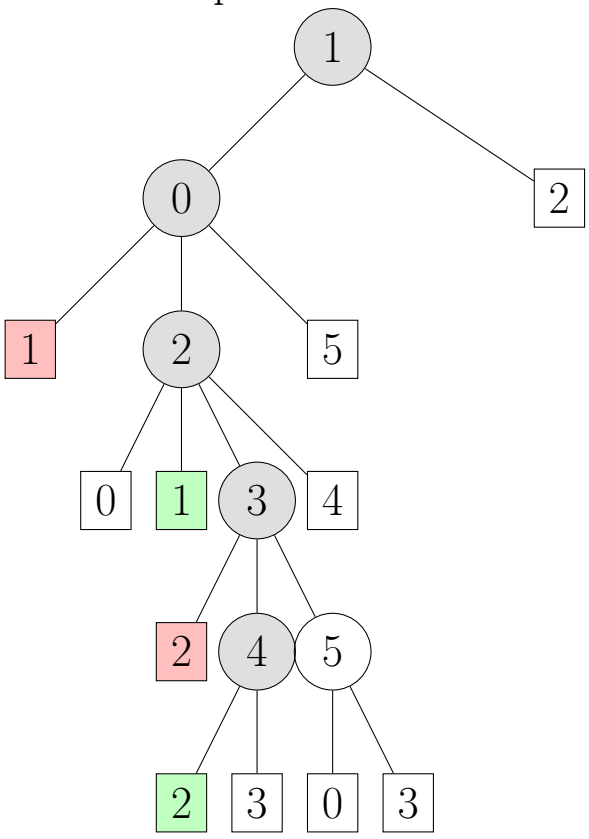
En regardant de plus près l'arbre des appels récursifs vu ci-dessus on se rend compte que les cycles seraient détectables

ainsi le premier cycle 1-0-2-1 (en gris) serait détectable **à condition que le sommet 1 ne soit plus marqué comme visité dans le parcours en profondeur**

Par contre 1-0-1 ne doit pas être considéré comme un cycle (le sommet 1 en rouge)

Pour mieux saisir le problème regardons un autre cas, le cycle 2-3-4-2 :

Dans le parcours en profondeur, la première fois où 2 est visité, son père est le noeud 0, que l'on mémorise
Puis 2 aurait pu être visité à partir du noeud 3 (à éviter) , par contre lorsque 2 est visité à partir du sommet 4 on détecte un cycle car son prédécesseur 0 mémorisé est différent de 4



L'idée est la suivante, s'il y a un cycle dans un graphe **connexe** alors de la source à n'importe quel point du cycle il y a au moins deux chemins possibles

Pour détecter la présence de plusieurs chemins possibles il suffit de modifier un peu l'algorithme de parcours en profondeur

pour définir une méthode `a_un_cycle()` dans la classe `Graphe`

`a_un_cycle()` va appeler une méthode **interne** `_a_un_cycle(s)` récursive à partir de s = 0 (on choisit une fois pour toute le sommet 0)

On n'utilise plus de tableau pour marquer les sommets visités, le tableau peres, initialisé dans chaque cellule par -1 et tel que peres[0] = 0 suffit :

On regarde tous les voisins v de s

1. si le sommet v n'a pas été visité, si peres[v] == -1 dans ce cas on appelle récursivement la fonction `_a_un_cycle(v)` sur v en partant de s après avoir fait `peres[v] = s`
2. sinon si `peres[s] != v` (pour éviter une séquence x - v - s - v qui n'est pas un cycle) et si `peres[v] != s` alors c'est un cycle

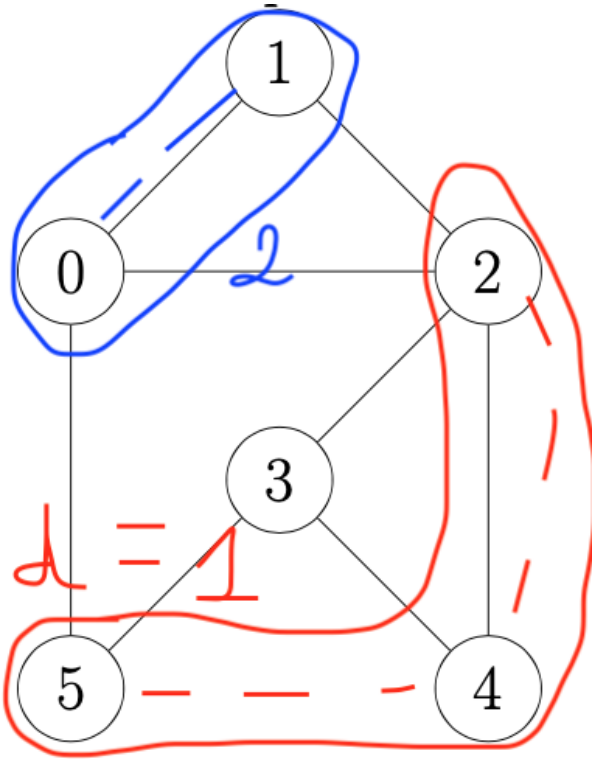
```
class Graphe:
    def __init__(self,nb_sommets):
        self.nb_sommets = nb_sommets
        self.nb_arcs = 0
        self.arcs =  [[]for i in range(nb_sommets)]
        self.peres = [-1]*G.nb_sommets
        self.peres[0] = 0

    def _a_un_cycle(self,s):
        for v in self.aretes[s]:
            if self.peres[v] == -1:
                self.peres[v] = s
                if self._a_un_cycle(v):
                    return True
            elif self.peres[s] != v and \
            self.peres[v] != s:
```

```
                return True
        return False

    def a_un_cycle(self):
        return self._a_un_cycle(0)
```

5.2 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)

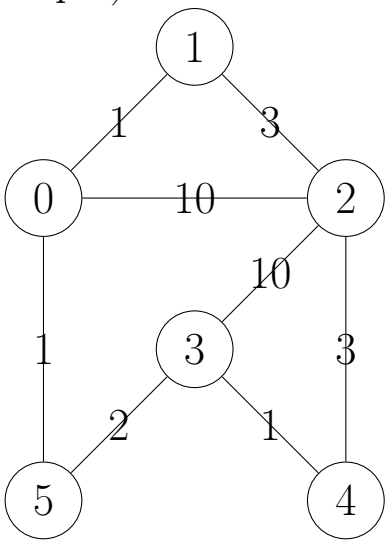
```
#on utilise l'objet deque de Python
class PL:
    def __init__(self,G,s):
        self.visites = [False]*G.nb_sommets
        self.source = s
        self.peres = [s]*G.nb_sommets
```

```
        self.pl(G,s)

    def pl(self,G,s):
        file = deque()
        file.appendleft(s)
        self.visites[s] = True
        while len(f) > 0:
            x = file.pop()
            print(x)
            for voisin in G.voisins(x):
                if not(self.visites[voisin]):
                    file.appendleft(voisin)
                    self.visites[voisin] = True
                    self.peres[voisin] = s

     def nb_sauts(v):
         """
         retourne le nombre de sauts de
         la source vers v
         """
         pass

     def chemin_vers(v):
         """
         retourne le chemin le plus court s'il exis
         forme d'une liste d'entiers
         s'il n'existe pas retourne une liste vide
         """
         pass
```

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

d est initialisé par $+\infty$ pour tous les sommets et par 0 pour la source elle-même

Nous avons étudié en TP les files de priorité et nous avons vu que les clés évoluent avec le temps

Ainsi une clé en diminuant "va remonter dans la file" comment faire le lien avec le sommet associé ?

On va envelopper les informations sommet,cle et position par une référence commune en créant un type `Sommet` et pour retrouver cette référence à partir d'un sommet du graphe on va utiliser un dictionnaire

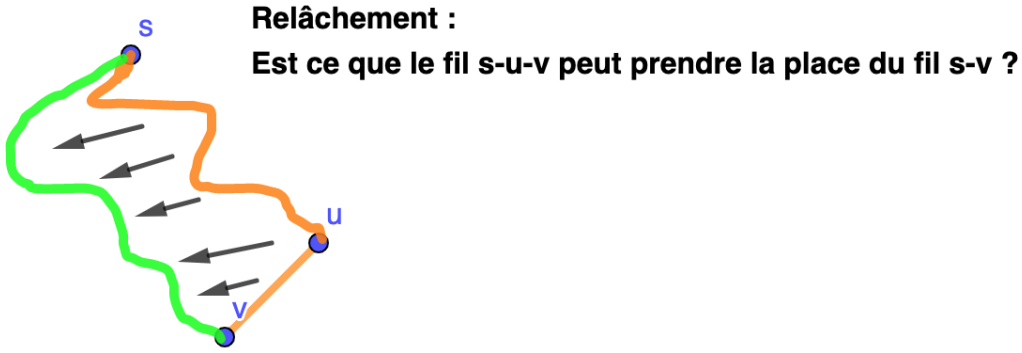
```
class Sommet:
    """
    un objet de type Sommet a trois attributs:
    -un sommet = un entier de 0 à n-1  = un sommet
    pondéré
    -une cle = la distance à la source du sommet
    en question (évolue avec le temps)
    -pos = position de l'objet dans la file min
     (évolue avec le temps)
    """
    def __init__(self,sommet,cle,pos):
        self.sommet = sommet
        self.cle = cle
        self.pos = pos

class File_min:
    """
    Une file de priorité min pour l'algorithme
    de Dijkstra
    la file contient des références concernant
    des objets de type Sommet
    les objets sont placés dans la file min
    à partir de leur clé
    """

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

    def inserer(self,Sommet):
```

```
        self.taille += 1
        self.tab.append(Sommet)
        Sommet.pos = self.taille
        self.percoler(self.taille)

    def tamiser(self,i):
        """
        la valeur du sommet i est  strictement plus
        grande que l'un de ses enfants
        On fait descendre cette valeur dans l'arbre
        jusqu'à ce qu'elle trouve sa place
        complexité logarithmique
        """
        enfant_gauche = i << 1
        enfant_droit = enfant_gauche + 1
        if enfant_gauche <= self.taille and\
        self.tab[enfant_gauche].cle < self.tab[i].c
            max = enfant_gauche
        else:
            max = i
        if enfant_droit <= self.taille and\
        self.tab[enfant_droit].cle < self.tab[max].
            max = enfant_droit
        if max != i:
            echanger(self.tab,i,max)
            self.tab[i].pos = i
            self.tab[max].pos = max
            self.tamiser(max)

    def percoler(self,i):
        """
```

```
        La valeur du sommet i est strictement plus
        petite que celle de son parent
        On fait monter cette valeur dans l'arbre
        jusqu'à ce qu'elle retrouve sa place
        """
        pos = i
        while pos > 1 and \
        self.tab[pos].cle < self.tab[pos >> 1].cle:
            echanger(self.tab,pos,pos >> 1)
            self.tab[pos].pos = pos
            self.tab[pos >> 1].pos = pos >> 1
            pos = pos >> 1

    def minimum(self):
        return self.tab[1].sommet

    def cree_tas_min(self,tableau):
        for i in range(self.taille):
            self.tab.append(tableau[i])
        for i in range(self.taille//2,0,-1):
            self.tamiser(i)

    def elimine_racine(self):
        racine = self.tab[1]
        self.tab[1] = self.tab[self.taille]
        #on met à jour la position du Sommet
        #qui prend la place de la racine
        self.tab[1].pos = 1
        #on élimine la racine
        self.tab.pop()
        self.taille -= 1
```

```
        self.tamiser(1)
        return racine.sommet

    def __len__(self):
        return len(self.tab)

    def __str__(self):
        ch =''
        for i in range(1,len(self.tab)):
            ch = ch + str(self.tab[i].sommet)+" "
        return ch
```

Ici diminuer la clé c'est diminuer la distance à la source à un voisin v d'un sommet u, par le processus de relâchement, illustré ci-dessous



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

Maintenant on modifie l'algorithme de parcours en largeur en créant une classe `Dijkstra`

Voici une implémentation en Python

```
class Dijkstra:
    def __init__(self,graphe,source,l):
```

```
        self.source = source
        self.distance = [inf]*graphe.nb_sommets
        self.distance[source] = 0
        self.ancetres = [source]*graphe.nb_sommets
        self.dijkstra(graphe,source,l)

    def relachement(self,v,x,l,file_min,
    dictionnaire):
        if self.distance[v] + l[v][x] < \
        self.distance[x]:
            self.ancetres[x] = v
            self.distance[x] = \
            self.distance[v] + l[v][x]
            #p est la position dans la file
            #du Sommet associé
            #au sommet x du graphe
            p = dictionnaire[x].pos
            file_min.tab[p].cle = self.distance[x]
            file_min.percoler(p)


    def dijkstra(self,graphe,source,l):
        liste = [Sommet(i,self.distance[i],i+1) \
        for i in range(graphe.nb_sommets)]
        dictionnaire = {i:liste[i] \
        for i in range(graphe.nb_sommets)}
        file_min = File_min(graphe.nb_sommets)
        file_min.cree_tas_min(liste)
        while len(file_min) > 1:
            v = file_min.elimine_racine()
            for x in graphe.aretes[v]:
```

```
                self.relachement(v,x,l,file_min,
                dictionnaire)

    def dist(self,sommet):
        pass

    def a_un_chemin_vers(self,v):
        pass
    def chemin_vers(self,v):
        pass
```

Explications

1. On enfile tous les sommets du graphe au début de l'algorithme avec un clé infini sauf la source avec une clé nulle
2. Ensuite tant que la file min n'est pas vide (attention une file min est implémentée avec une liste à partir de l'indice 1 d'où `while len(file_min) > 1`), on enlève la racine v de la file min et on relâche le chemin source - v - x pour tout voisin x de v afin de minimiser la distance de la source à x

6 Exercices

Ex 1

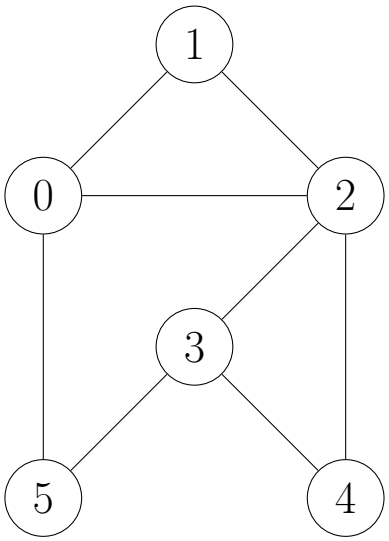
Implémenter les méthodes `nb_arcs()`, `voisins(i)` et `__str()__`de la classe Graphe du cours

Ex 2

1. Ecrire l'interface d'un graphe **non orienté**
2. Implémenter l'interface `Graphe_NO` pour non orienté en Python

Ex 3

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 4

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 5

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

Ex 6

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

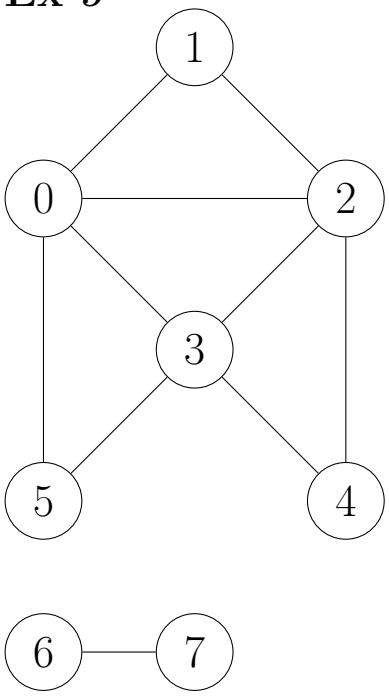
Ex 7

1. Implémenter dans la classe `PP` les méthodes `a_un_chemin_vers(v)` et `chemin_vers(v)`
2. Implémenter une méthode **itérative** `a_un_cycle()`

Ex 8

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

Ex 9



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 10

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)