Graphes non orientés

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**

Si un graphe n'est pas connexe, il est la réunion disjointe de parties connexes appelées **composantes connexes**.

Le nombre de voisins d'un sommet est le **degré** du sommet.

Par exemple la station Denfert Rochereau est de degré 5.

Il peut exister dans un graphe une **boucle**, c'est à dire une arête reliant un sommet à lui-même.

Il peut y avoir aussi des **arêtes parallèles**, par exemple, entre Denfert-Rochereau et Raspail il y a deux arêtes (lignes 6 et 13).

Un **cycle** est un chemin qui commence et se termine par le même sommet.

Par exemple Denfert Rochereau – Raspail – Vavin – Montparnasse par la ligne 13 puis Montparnasse–Edgar Quinet–Raspail–Denfert Rochereau par la ligne 6

Un **graphe acyclique** est un graphe sans cycle.

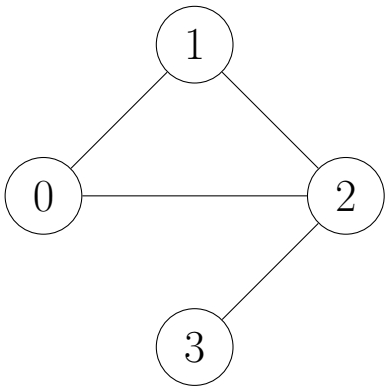
Un **arbre** est un graphe connexe acyclique.

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

1.1 Liste de voisinage



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

Par exemple pour le graphe non 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 sont les sommets 0 et 2 que l'on matérialise par la liste [0,2]
3. les voisins du sommet 2 sont les sommets 0,1 et 3 que l'on matérialise par la liste [0,1,3]
4. Le voisin du sommet 3 est le sommet 2 que l'on matérialise par la liste [2]

La liste de listes [[1,2], [0,2], [0,1,3],[2]] contient l'information au sujet des arêtes du graphe non orienté.

1.2 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 :

$$\begin{vmatrix} T_{00} & T_{01} & T_{02} & T_{03} \\ T_{10} & T_{11} & T_{12} & T_{13} \\ T_{20} & T_{21} & T_{22} & T_{23} \\ T_{30} & T_{31} & T_{32} & T_{33} \end{vmatrix} = \begin{vmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{vmatrix}$$

1.3 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

2 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`

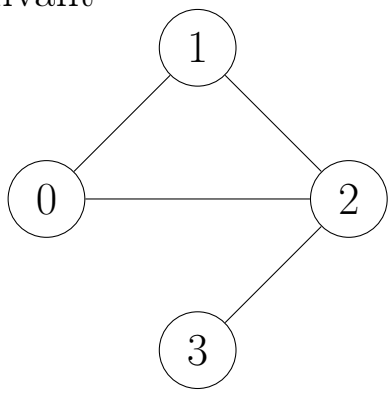
`voisins(i)` retourne la liste des voisins de i

3 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
```

4 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 ?

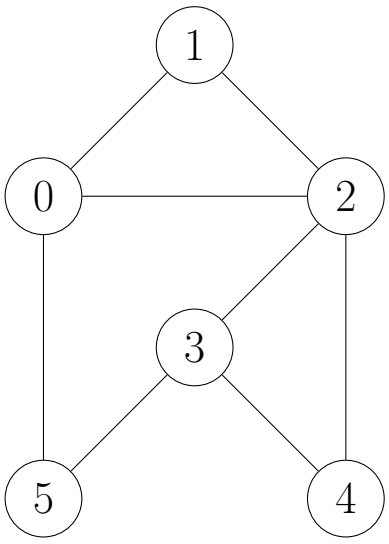
4.1 Parcours en profondeur dans un graphe

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
```

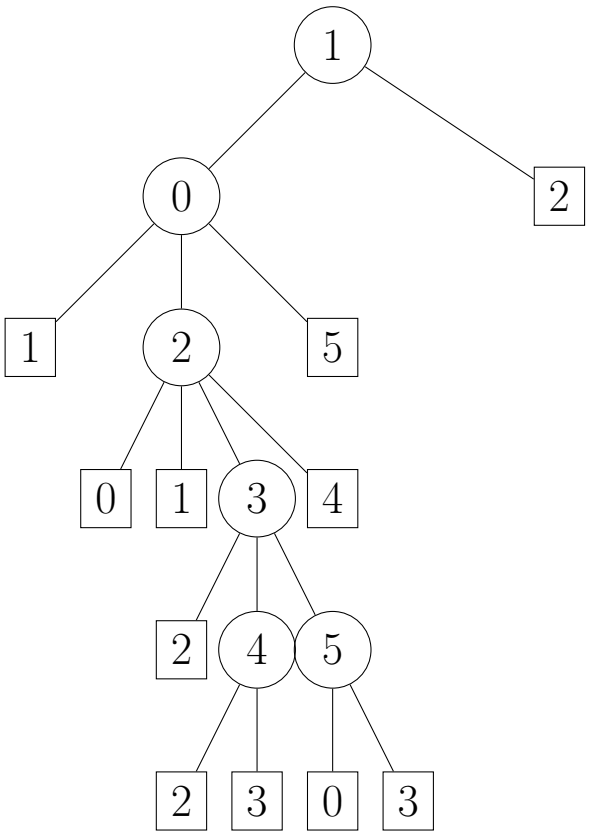
Par exemple



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):
        pass

    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
```

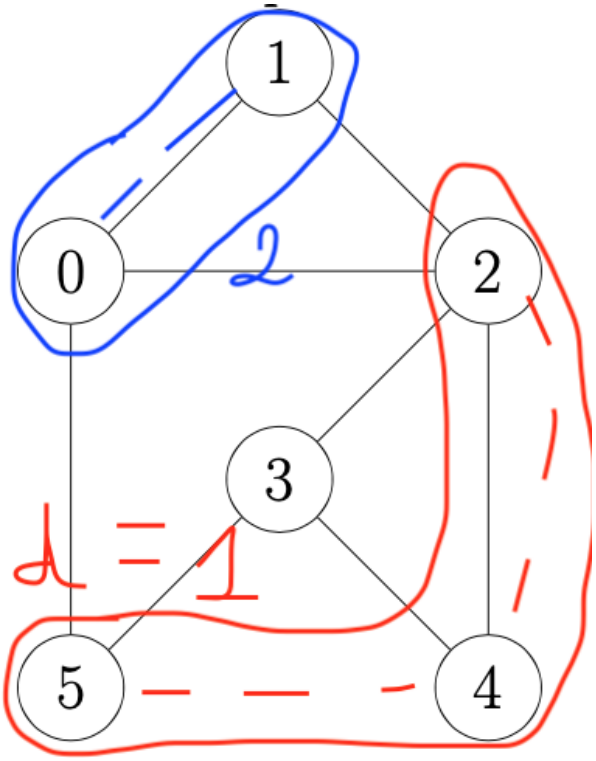
Applications

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

4.2 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) valable à la fois pour les graphes orientés et non orientés

```
class PL:
    """
```

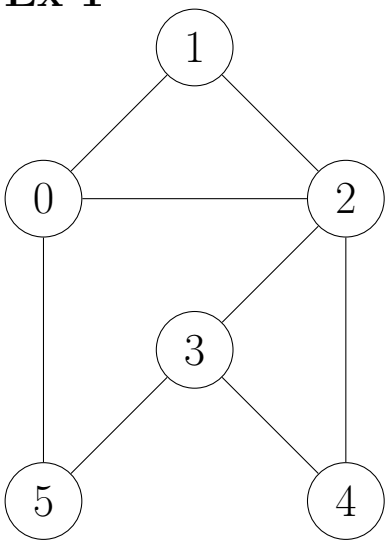
```
    pour graphe orienté et non orienté
    """
    def __init__(self,graphe,source):
        self.visites = [False]*graphe.nb_sommets
        self.peres = [source]*graphe.nb_sommets
        self.source = source
        self.pl(graphe,source)

    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)

5 Exercices

Ex 1



1. Donner un chemin qui part du sommet 1 vers le sommet 3
2. Ce graphe est il connexe ?
3. Donner les degrés de chaque sommet du graphe ci-dessus.
4. Ce graphe est il acyclique ?
5. Implémenter en Python une méthode `degre(i)` qui retourne le degré du sommet i.
6. Implémenter en Python une méthode `max_degre()` qui retourne le degré maximal du graphe.
7. Justifier que : Dans un graphe, la somme des degrés des sommets est le double du nombre d'arêtes.
8. Justifier que le degré moyen d'un graphe non orienté vérifie : $\overline{d} = \dfrac{2|\text{arêtes}|}{|\text{sommets}|}$
9. Dans une salle, un certain nombre de personnes se serrent la main. Montrer que le nombre de personnes qui ont serré un nombre impair de mains est un nombre pair.

Ex 2

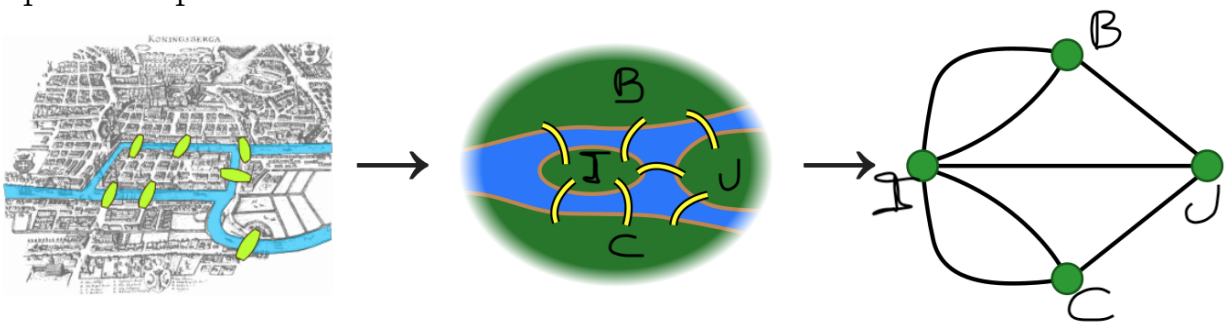
Voici un problème connu sous le nom du problème des sept ponts de Königsberg, raconté par le mathématicien Euler(1707-1783) :

"Dans la ville de Königsberg, en Prusse, se trouve une île appelée Kneiphof, ceinte par les deux bras de la rivière Pregel. Sept ponts enjambent les deux bras de la rivière. La question consiste à savoir si une personne peut, au cours de sa promenade, ne franchir chacun des ponts une seule fois...(1736)"

Voici une carte et une modélisation sous forme de graphe non orienté, des connexions entre l'ile I et les trois bords B,C et J.

Le but est de trouver un **cycle eulérien** dans ce graphe.

Un **cycle eulérien** est un cycle qui passe une fois et une seule par chaque arête du sommet.



1. En faisant une démonstration par l'absurde prouver qu'il n'existe pas de cycle eulérien dans ce graphe.
2. Donner les degrés de chaque sommet du graphe ci-dessus.
3.

Ex 3

Il s'agit de démontrer le théorème suivant :

Théorème 1. *Un graphe connexe contient un cycle eulérien si et seulement si chacun de ses sommets a un degré pair*

1. Supposons qu'au moins un sommet du graphe a un nombre

impair de voisins, prouver qu'il ne peut pas avoir de cycle eulérien. Qu'a-t-on démontré ?

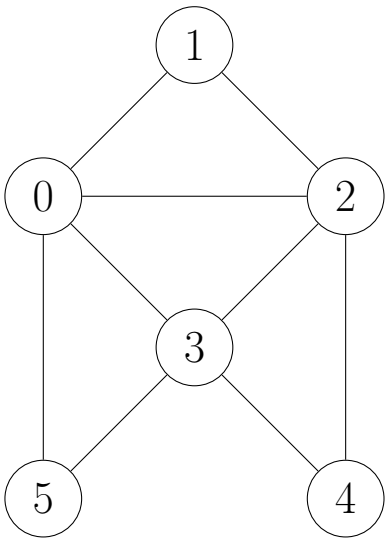
2. On suppose que tous les sommets ont un degré pair.

   L'algorithme suivant trouve un cycle eulérien. (On parle de **preuve constructive**)

Algorithme 3 : Création d'un cycle eulérien dans un graphe connexe dont chaque sommet a un degré pair

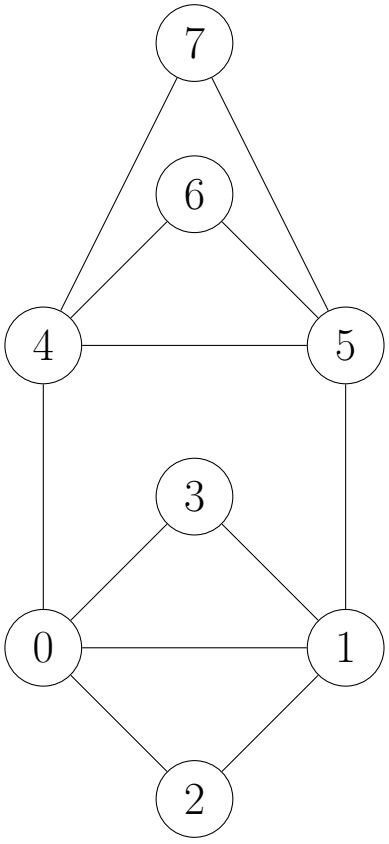
```
cycle_eulerien (G)
début
    Données : Un graphe G connexe
    Résultat : Un cycle eulérien C
1   // à chaque fois qu'on sélectionne une arête
2   // il faut mettre à jour le degré des sommets visités
3   // et les arêtes restantes
4   S ← un sommet de degré le + élevé
5   C ← S
6   tant que il y a des arêtes non visitées dans G faire
7       Partir du sommet S où on se trouve dans C et aller
         vers le sommet de degré le plus élevé S' dans G
8       Avancer au hasard jusqu'à retomber sur S et faire
         un cycle C'
9       (Tout en veillant à ce que le graphe lorsqu'on
         enlève les arêtes reste connexe)
10      C ← C ∪ C'
11  fin
12  retourner C
fin
```

Pour expliciter l'algorithme on va l'exécuter sur le graphe suivant :



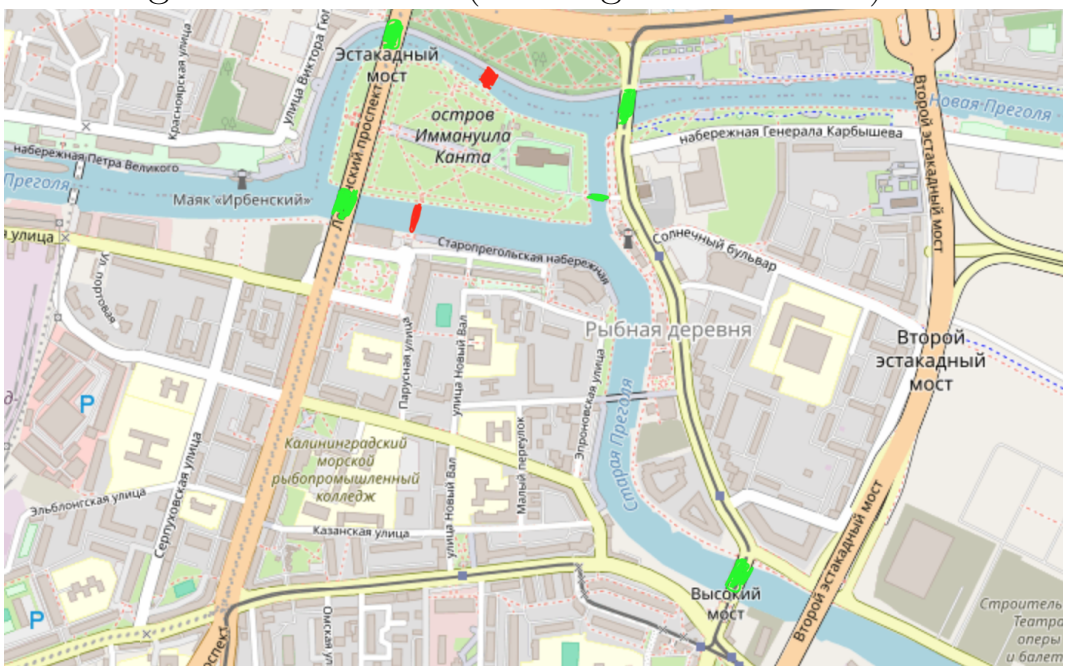
On part de C = S = 3 dont le degré est 4
On peut aller vers S' = 0 ou 2 de degré 4,
supposons 3-2-1-0-3 c'est le cycle C'
On n'a pas épuisé toutes les arêtes de G on recommence
C = 3-2-1-0-3
On peut aller vers 4 ou 5 de degré 2
On ne peut pas aller vers 0 ou 2 qui sont de degré 2 mais il n'y a plus de connexion vers 3
On peut faire par exemple
C' = 3-5-0-2-4-3 et on a épuisé toutes les arêtes
Conclusion le cycle eulérien est 3-2-1-0-3-5-0-2-4-3

3. Exécuter l'algorithme sur le graphe suivant :



Ex 4

Aujourd'hui Königsberg est une ville russe appelée Kaliningrad. Parmi les sept ponts, deux ont été détruits au cours de la deuxième guerre mondiale (en rouge sur la carte)

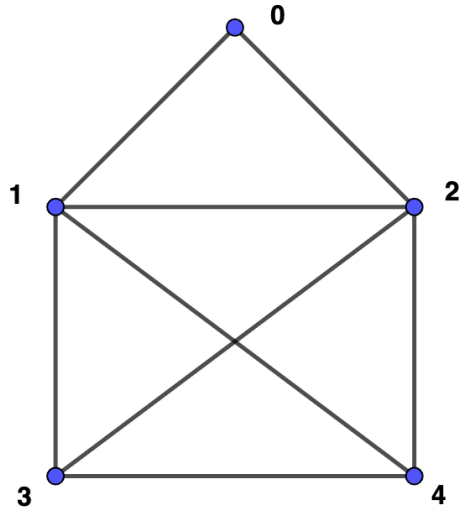


Un illustre philosophe allemand est né et a vécu toute sa vie

à Königsberg, qui est-il ?

Ex 5

Un chemin dans un graphe est eulérien s'il passe une fois et une seule par chaque arête du graphe (sans forcément être un cycle).



1. Y-a-t-il un cycle eulérien dans le graphe ci-dessus ?
2. Y-a-t-il un chemin eulérien dans le graphe ci-dessus ?

Ex 6

"A graph can be drawn as a set of points, called nodes, and a set of lines joining the nodes, called edges. Details such as the length and shape of the edges aren't part of the graph itself, though ; the only thing that distinguishes one graph from another are the connections between the nodes. The number of edges that meet at any given node is known as its valence" (Greg Egan) Voir `https://www.gregegan.net/SCHILD/Connect/Connect.html`

Ex 7

DFS -> Lire `https://xkcd.com/761/`

Ex 8 : Application du parcours en profondeur : connexité

1. Ajouter une méthode `est_connexe(g)` à la classe `PP` qui

renvoie Vrai si le graphe non orienté g est connexe et Faux sinon

2. 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).

   Cette classe a trois attributs :

   (a) `visites` un tableau pour marquer les sommets visités au cours du parcours en profondeur

   (b) un entier `nb_comp` qui mémorise le nombre de composantes connexes.

   (c) un tableau `id` qui donne un identifiant à chaque sommet.

   Deux sommets appartenant à la même composante connexe ont le même identifiant.

   Si le graphe est connexe, tous les sommets ont pour identifiant 1.

   Indication : Dans le constructeur, parcourir l'ensemble des sommets, et ne lancer un parcours en profondeur que si le sommet n'a pas été visité. Après chaque parcours en profondeur augmenter de 1 l'attribut `nb_comp`.

   Attribuer à chaque composante connexe, la valeur **actuelle** de l'attribut `nb_comp`.

Ex 9 :Application du parcours en profondeur : acyclicité

En vous inspirant de ce qui a été fait pour les composantes connexes, créer une classe `Cycle` dont le but est de détecter la présence d'un cycle dans un graphe en faisant des parcours en profondeur sur chaque composante connexe.

Définir les attributs et les méthodes de cette classe.

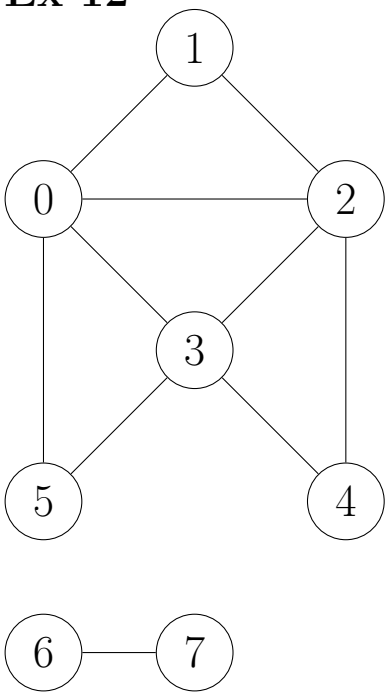
Ex 10

Ecrire une méthode **itérative** `pp_iter()` de la classe `PP`

Ex 11

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

Ex 12



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 13

1. L' implémentation non récursive du parcours en profondeur avec une pile, symétrique du parcours en largeur avec une file ne marche pas

```
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 pp(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.peres[w] = x
                    pile.append(w)
```

Pour s'en convaincre prendre un contre-exemple simple.

2. La raison principale est qu'en procédant ainsi on ne traduit pas la pile des appels récursifs.

   On pourrait en utilisant un itérateur mimer la pile des appels récursifs.

   On peut aussi procéder autrement. On ne cherche plus à mimer la pile des appels récursifs mais mettre dans la pile les sommets vus dès la première fois :

   On procède ainsi :

   (a) On affiche la source

(b) On marque la source et on empile tous les voisins de la source.

(c) Tant que la pile n'est pas vide, on dépile le sommet x

Si x est marqué cela signifie que l'on a déjà vu,on ignore la suite des instructions avec `continue`

Sinon on affiche x, on marque x et on empile tous ses voisins.

```
def pp_iter(self,graphe,source):
    pile = []
    print(source)
    self.visites[source] = True
    for voisin in graphe.voisins(source):
        pile.append(voisin)
        self.ancetres[voisin] = source
    while len(pile) != 0:
        sommet = pile.pop()
        if self.visites[sommet]:
            continue
        print(sommet)
        self.visites[sommet] = True
        for voisin in graphe.voisins(sommet):
            pile.append(voisin)
            self.ancetres[voisin] = sommet
```