# Algorithmes, Programmation, IA

## Cours 3

**Jean-Jacques Lévy**

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/algo-prog-ia-25

# Plan

- rappels
- classes et objets
- arbres de syntaxe abstraite
- belle impression
- graphes
- représentation des graphes avec listes de successeurs

dès maintenant: télécharger Python 3 en `http://www.python.org`

# Quelques rappels

- le cours utilise le langage Python et l'environnement Visual Studio Code. (`vscode`)

- et pour la partie IA, la bibliothèque Pytorch

# Procédures ou Méthodes

- programmation procédurale
  - on regroupe les opérations à l'intérieur du corps de la fonction
  - on fonctionne par induction structurelle
  - si on modifie la classe, on doit changer toutes les fonctions

- programmation orientée-objet. (*OO programming*)
  - chaque classe a une méthode spécifique
  - l'objet applique la méthode de sa classe
  - si on modifie la méthode, on doit changer la même méthode dans toutes les classes

# Programmation fonctionnelle ou impérative

- programmation fonctionnelle
  - on ne modifie pas les arbres
  - on rajoute de nouveaux noeuds
  - et on partage les sous-arbres (non modifiés)

- programmation impérative
  - on fait des effets de bord sur les arbres
  - on modifie donc leur structure
  - on optimise la place mémoire en ne créant pas de nouveaux noeuds
  - danger… danger !!

# Arbres de syntaxe abstraite (ASA)

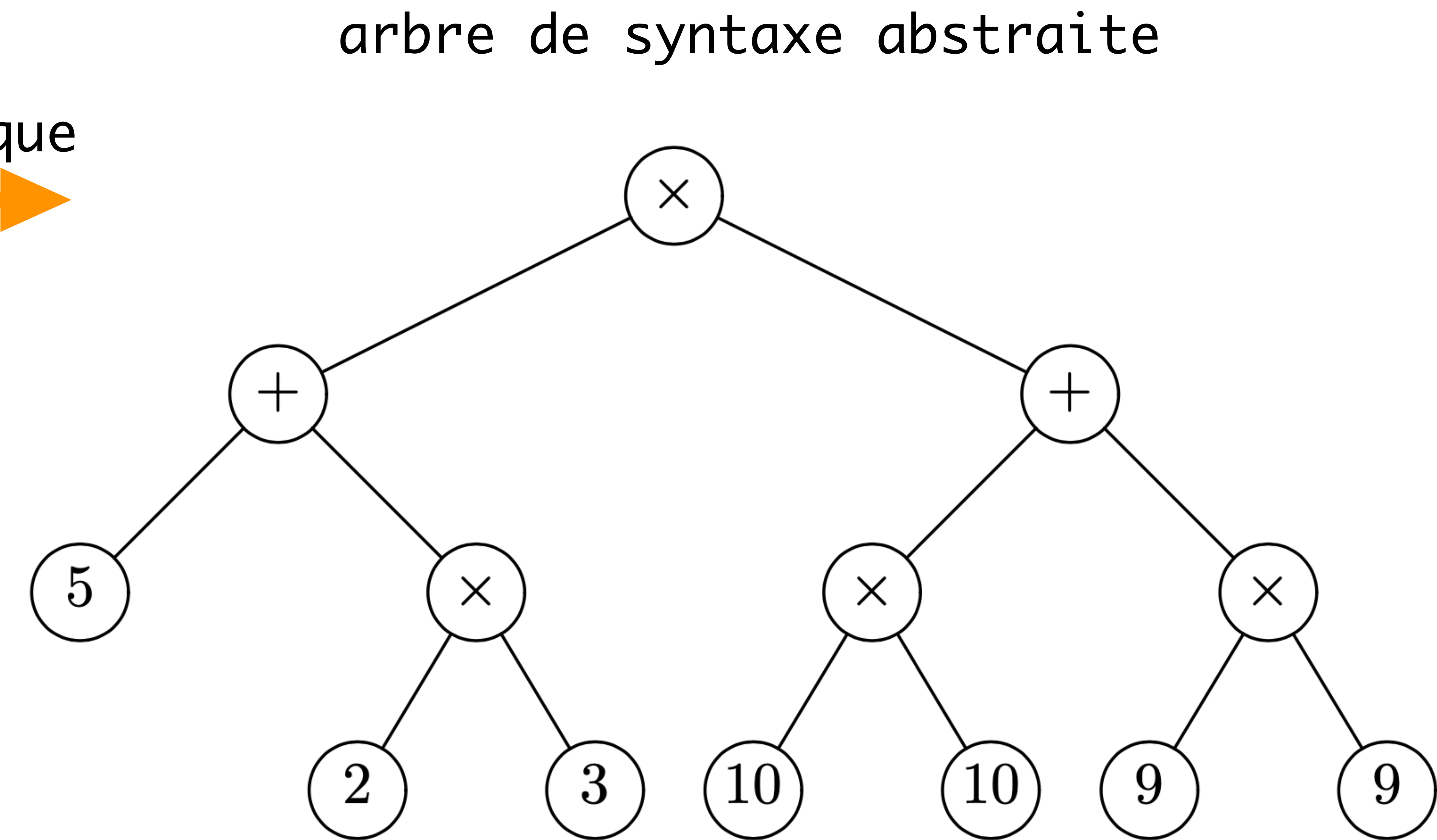
• analyse syntaxique

analyse syntaxique

chaîne de caractères

(5 + 2 * 3) * (10 * 10 + 9 * 9)

belle impression




• structure arborescente des systèmes de fichiers

• les arbres sont à la base des algorithmes de l’informatique

# Arbres de syntaxe abstraite (ASA)

- les ASA représentent des expressions arithmétiques

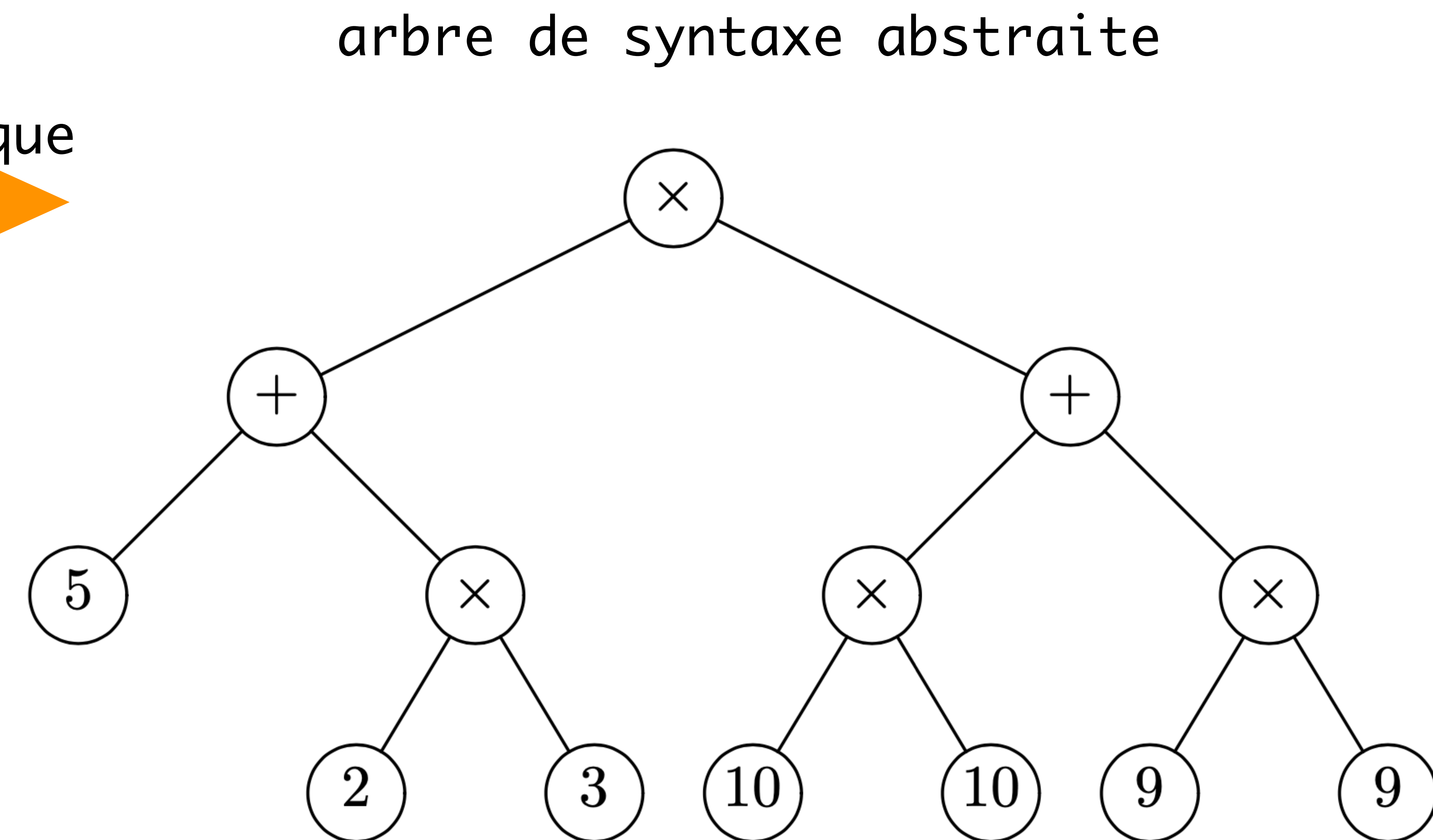
- analyse syntaxique

chaîne de caractères

(5 + 2 * 3) * (10 * 10 + 9 * 9)

analyse syntaxique →

← belle impression




5 2 3 * + 10 10 * 9 9 * + *

* + 5 * 2 3 + * 10 10 * 9 9

**Exercice**: Imprimer en notation polonaise préfixe

**Exercice**: Imprimer en notation polonaise postfixe

**Exercice**: Imprimer en notation infixe sans parenthèses

**Exercice**: Imprimer en notation infixe avec parenthèses

***Hint***: on tiendra compte de la précédence des opérateurs

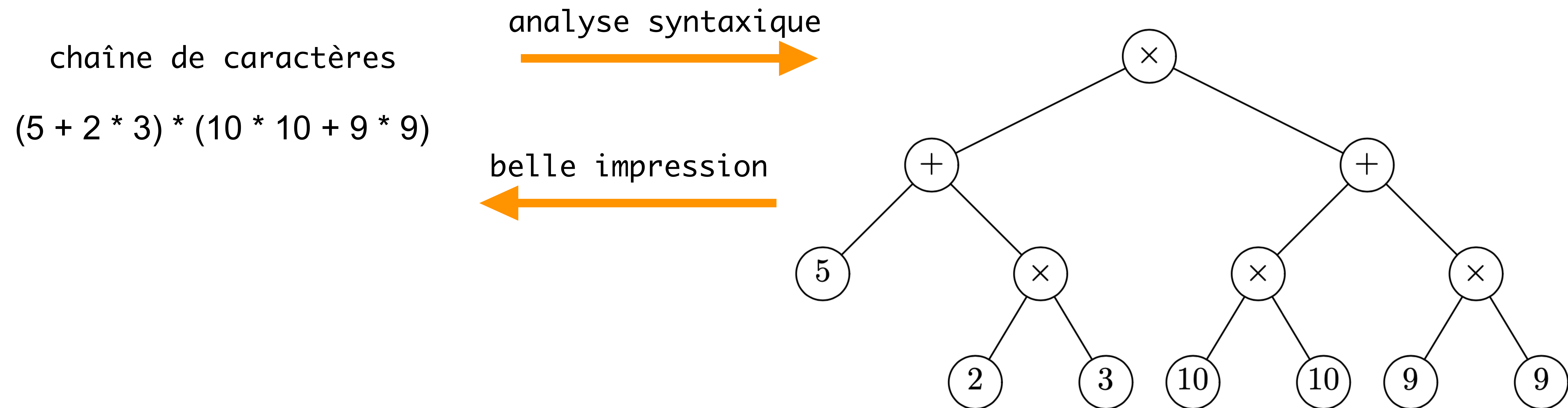
```
precedence = {'+': 1, '*': 2, '-': 1, ‘/‘:2, ‘**’:3}
```

***Question ++***: comment privilégier l’assiociation à gauche ou à droite pour les opérateurs de même précédence ?

# Arbres de syntaxe abstraite (ASA)

- les ASA représentent des expressions arithmétiques

- analyse syntaxique




**Exercice**: Evaluer le résultat d'un arbre ASA dans un environnement donné.

*Hint*: on représentera l'environnement par un dictionnaire associant une valeur à toute variable

```
env = {'x': 1, 'y': -2, 'z': 10}
```

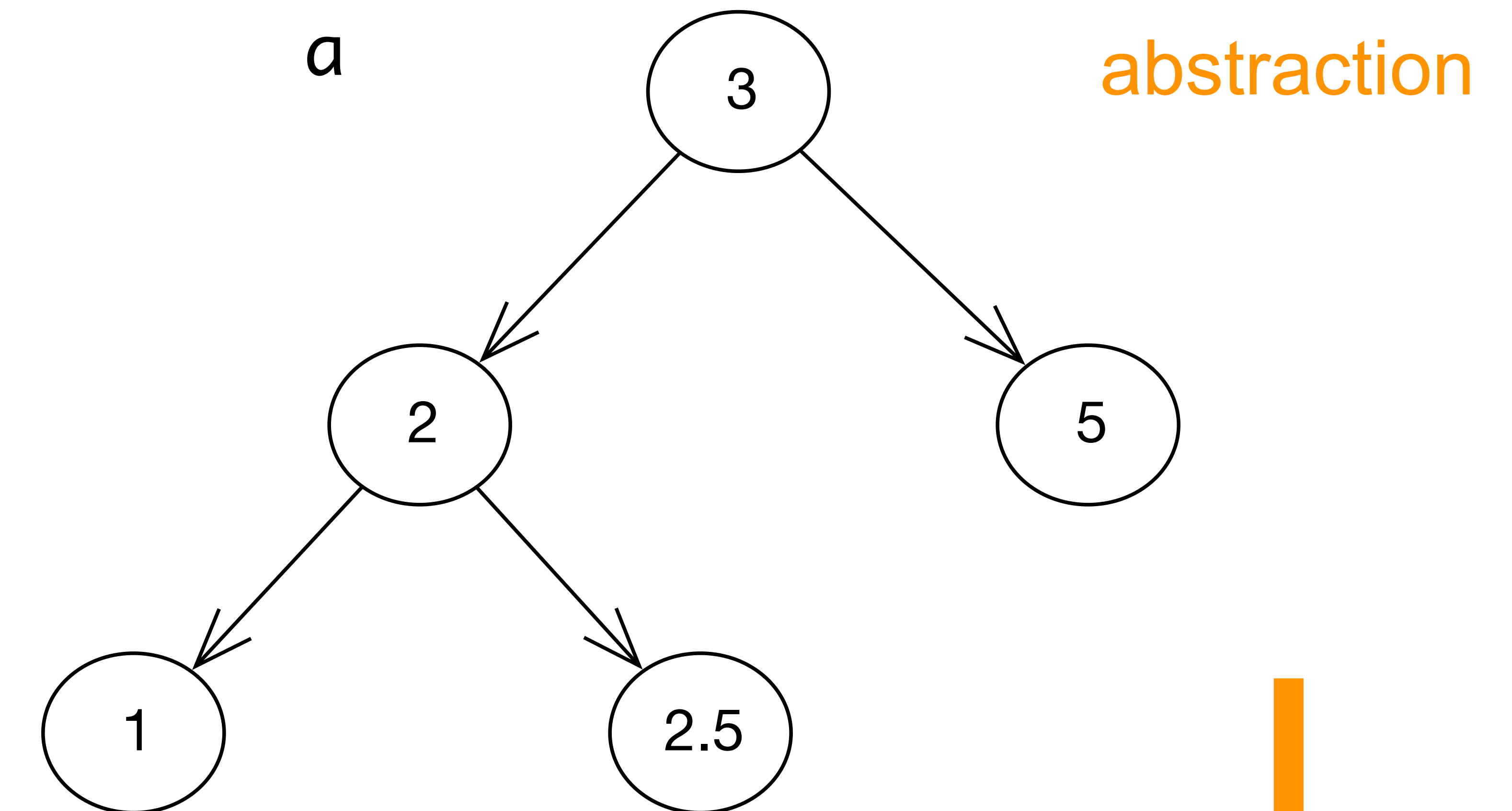
# Représentation des arbres

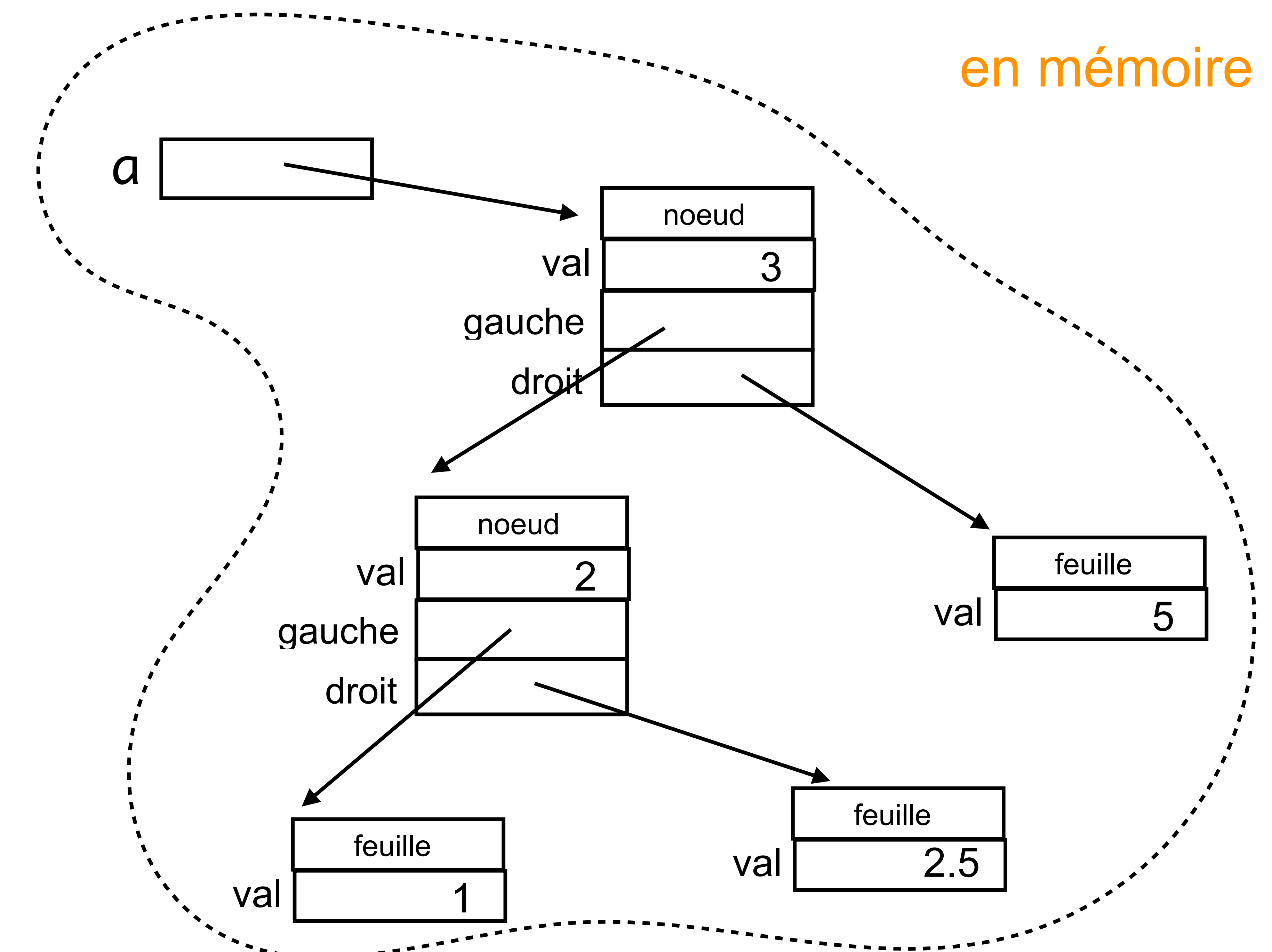
- on définit une classe pour les noeuds et pour les feuilles

```
class Noeud:
    def __init__ (self, x, g, d) :
        self.val = x
        self.gauche = g
        self.droit = d

class Feuille:
    def __init__ (self, x) :
        self.val = x
```

- et on construit des arbres

```
a = Noeud (3, Noeud (2, Feuille (1), Feuille (2.5)), Feuille (5))
```





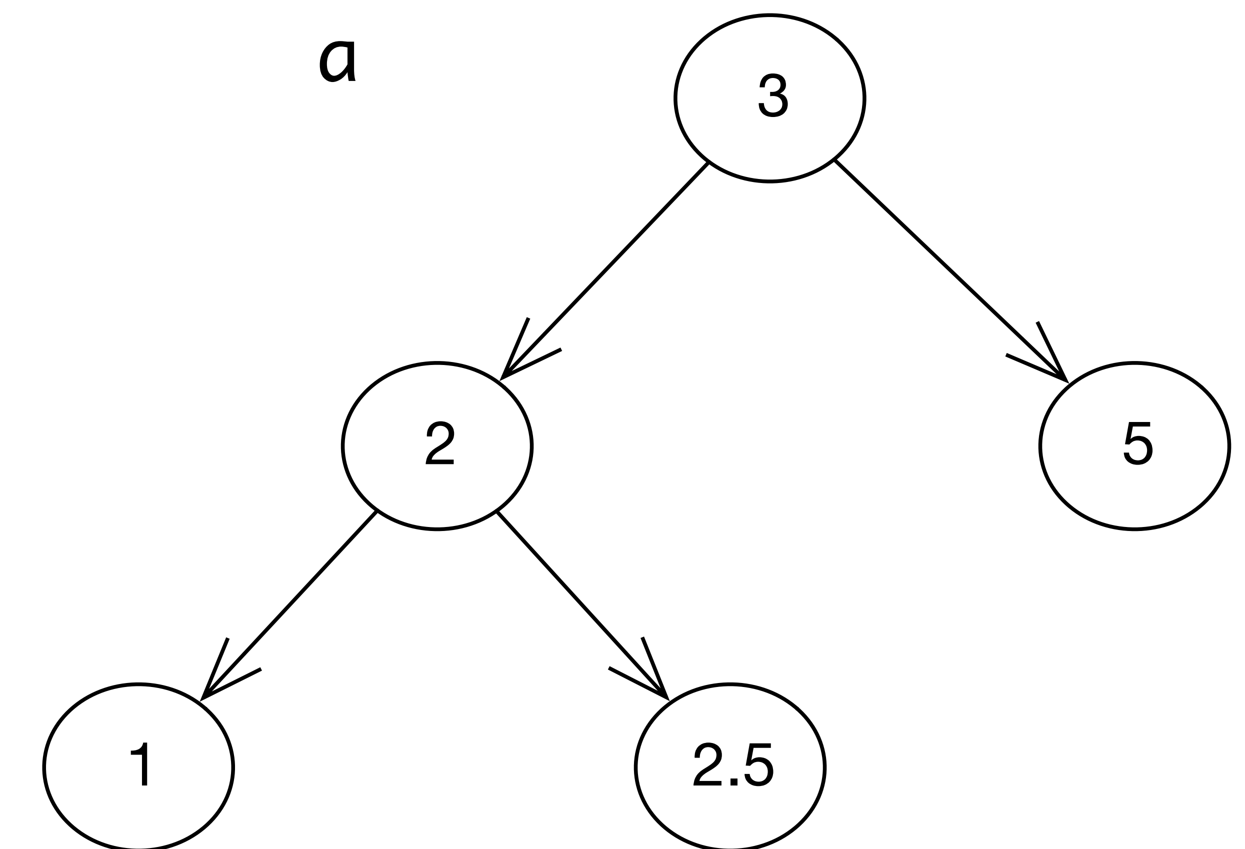
# Représentation des arbres

- on définit une méthode pour l'impression des noeuds et des feuilles

```
class Noeud:
    # ...

    def __str__ (self) :
        return "Noeud ({}, {}, {})".format (self.val, self.gauche, self.droit)


class Feuille:
    # ...

    def __str__ (self) :
        return "Feuille ({})".format (self.val)
```

- on construit et imprime des arbres

```
  a = Noeud (3, Noeud (2, Feuille (1), Feuille (2.5)), Feuille (5))
  print (a)
➡ Noeud (3, Noeud (2, Feuille (1), Feuille (2.5)), Feuille (5))

  print (a.droit)
➡ Feuille (5)

  print (a.gauche)
➡ Noeud (2, Feuille (1), Feuille (2.5))
```

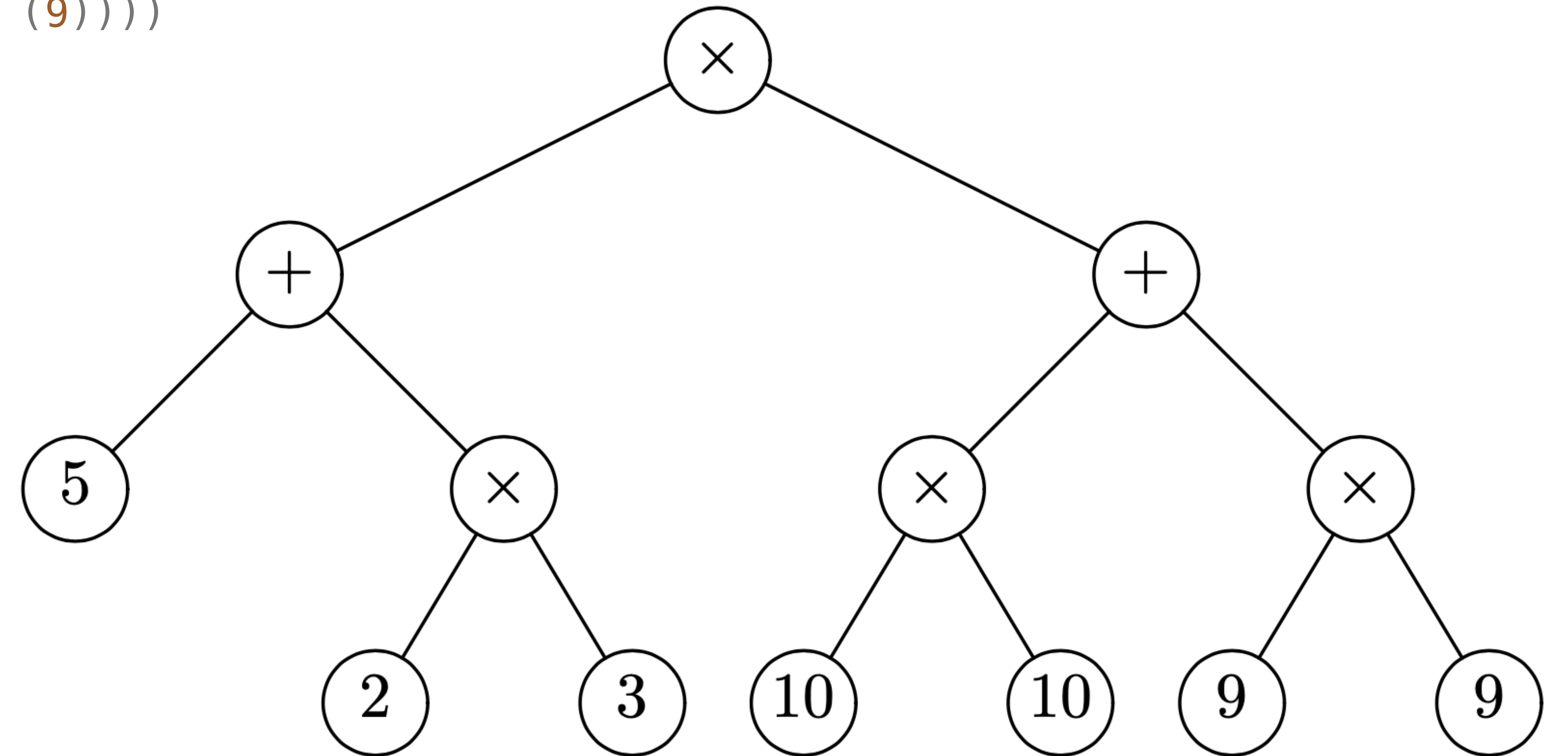


# Représentation des arbres

- on construit et imprime des arbres

```
b = Noeud ('*', Noeud ('+', Feuille (5),
                            Noeud ('*', Feuille (2), Feuille (3))),
                Noeud ('+', Noeud ('*', Feuille (10), Feuille (10)),
                            Noeud ('*', Feuille (9), Feuille (9))))
```

```
print (b.gauche.gauche)
Feuille (5)
print (b.gauche.droit)
Noeud ('*', Feuille (2), Feuille (3))
```



# Arbres de syntaxe abstraite (ASA)

- les ASA représentent des expressions arithmétiques

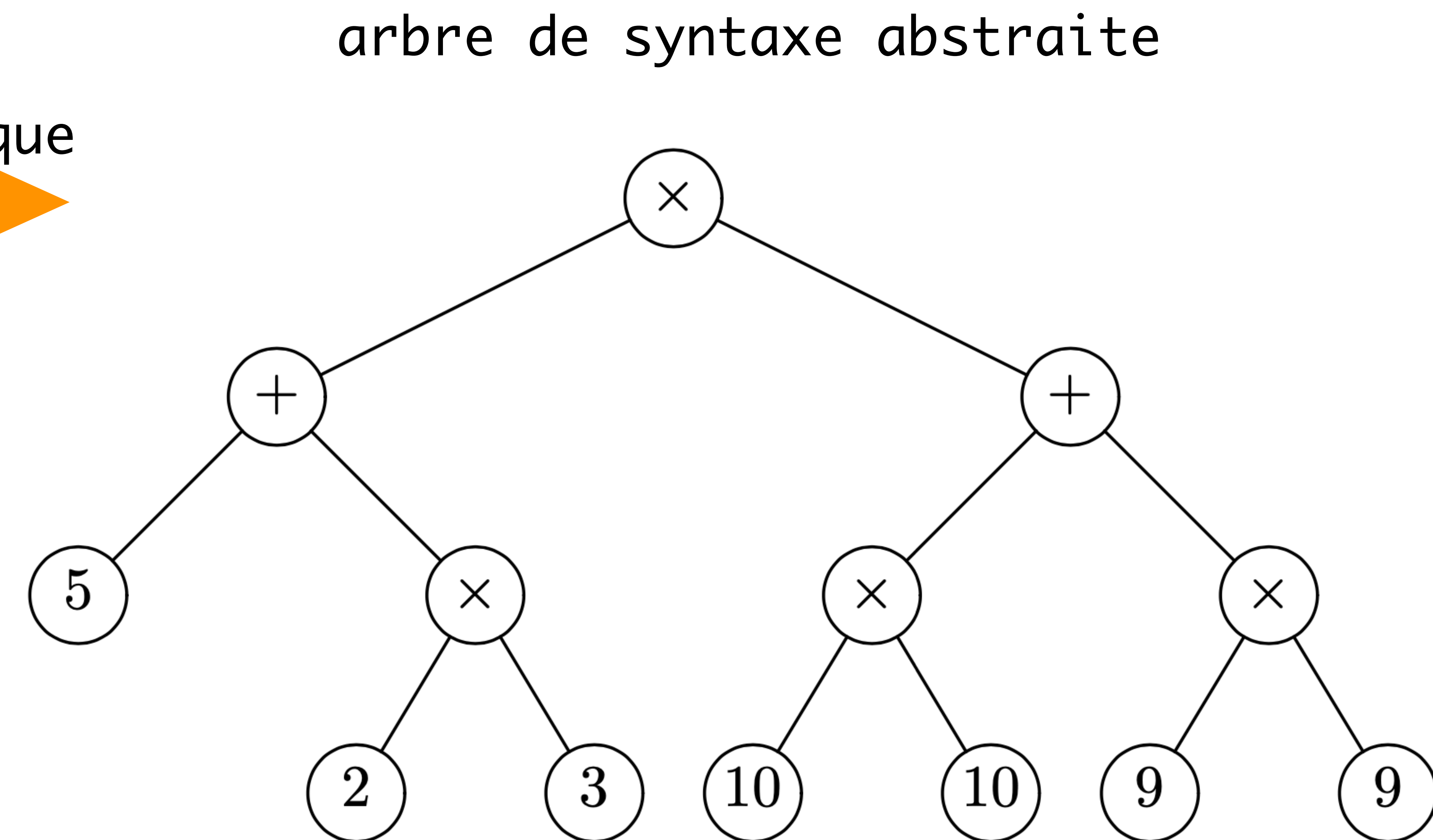
- analyse syntaxique

chaîne de caractères

(5 + 2 * 3) * (10 * 10 + 9 * 9)

analyse syntaxique →

belle impression ←




5 2 3 * + 10 10 * 9 9 * + *

* + 5 * 2 3 + * 10 10 * 9 9

**Exercice**: Imprimer en notation polonaise préfixe

**Exercice**: Imprimer en notation polonaise postfixe

**Exercice**: Imprimer en notation infixe sans parenthèses

**Exercice**: Imprimer en notation infixe avec parenthèses

***Hint***: on tiendra compte de la précédence des opérateurs

```
precedence = {'+': 1, '*': 2, '-': 1, '/':2, '**':3}
```

***Question ++***: comment privilégier l'assiociation à gauche ou à droite pour les opérateurs de même précédence ?

# Python ++

- traitement des exceptions
  - `try:` début d'un bloc avec exception possible
  - `except IOError:` récupère l'exception `IOError`
  - `except:` récupère toutes les exceptions
  - `finally:` pour le traitement normal **et** le traitement exceptionnel

```
def lire_lignes (nom) :
    try:
       f = open (nom, 'r')
       return f.read().splitlines()
    except IOError:
       print("Fichier '%s' inexistant." % nom)

lire_lignes('abc')
```

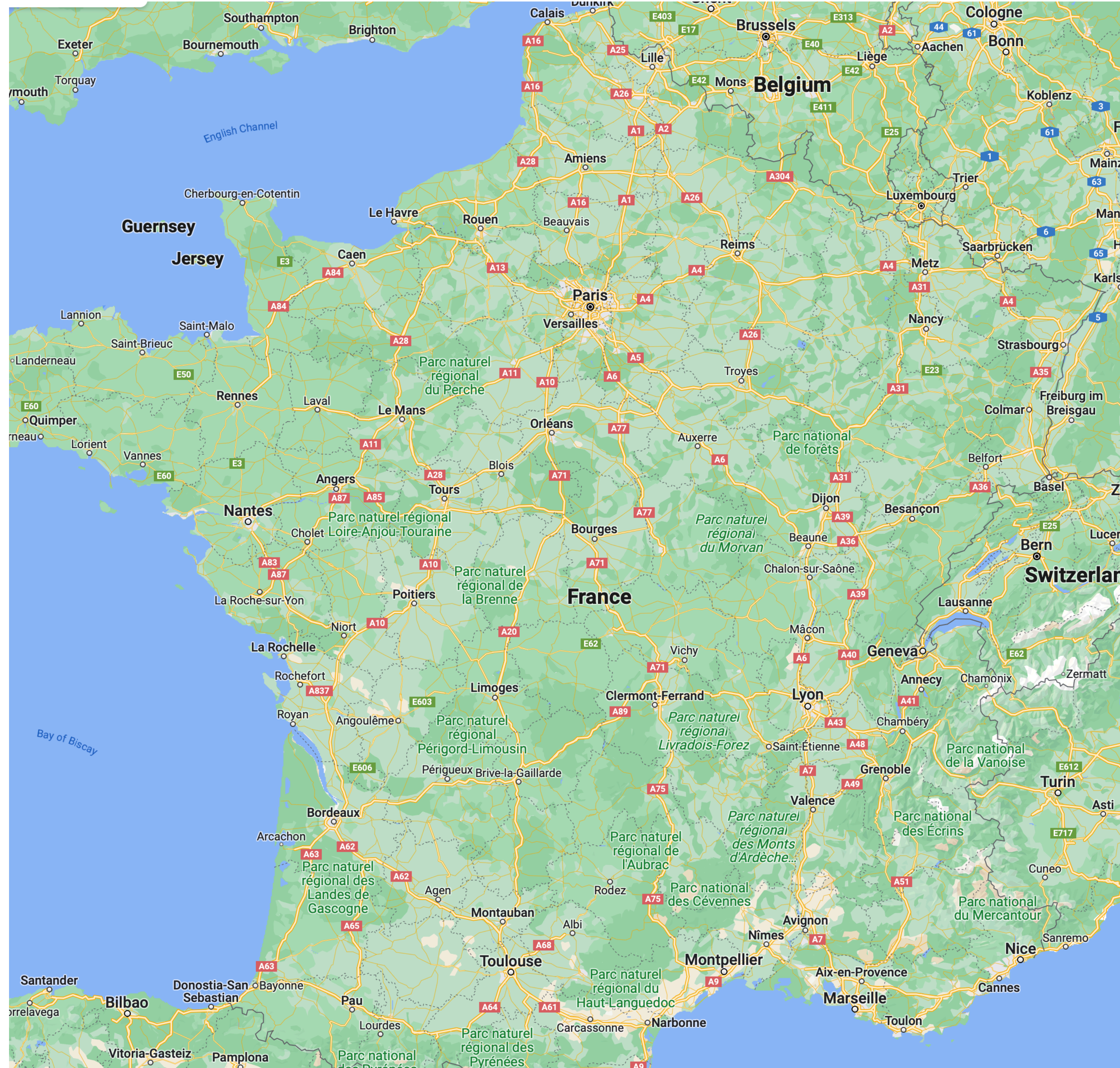
➡ Fichier 'abc' inexistant.

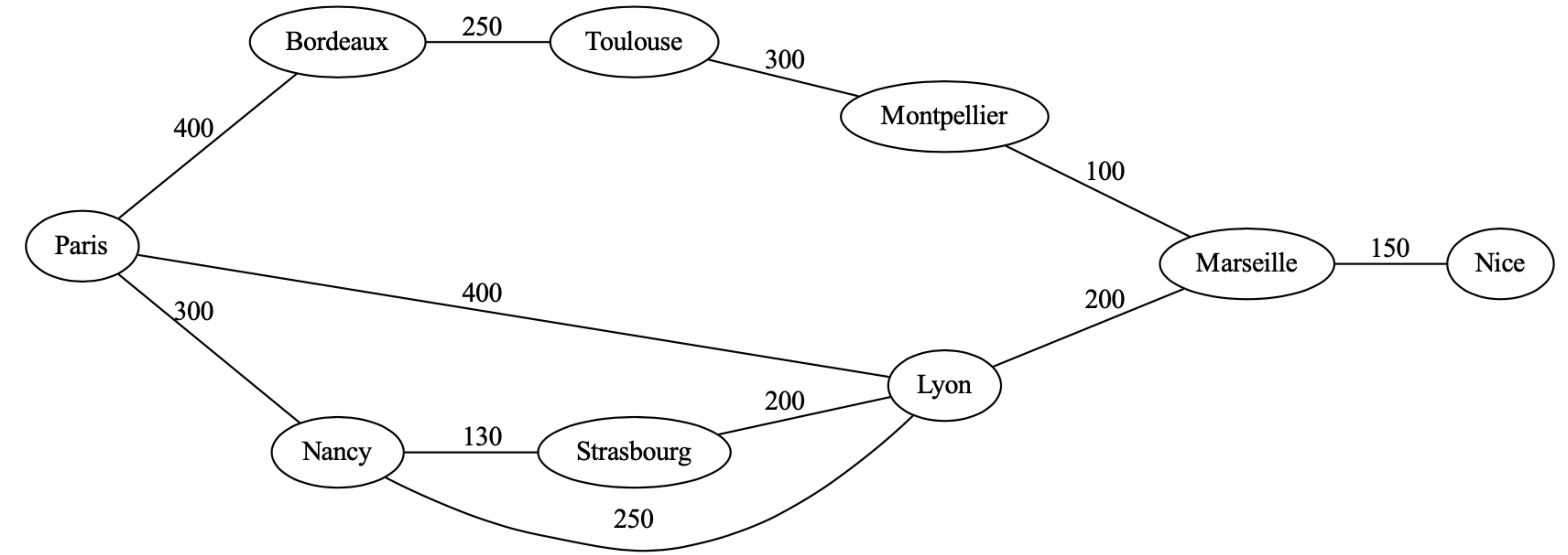
- alias de module
  - `as` déclare un alias pour un module (par exemple si le nom est trop long)

```
import random as r

r.choice (['a', 'b', 'c'])
```

➡ 'a'

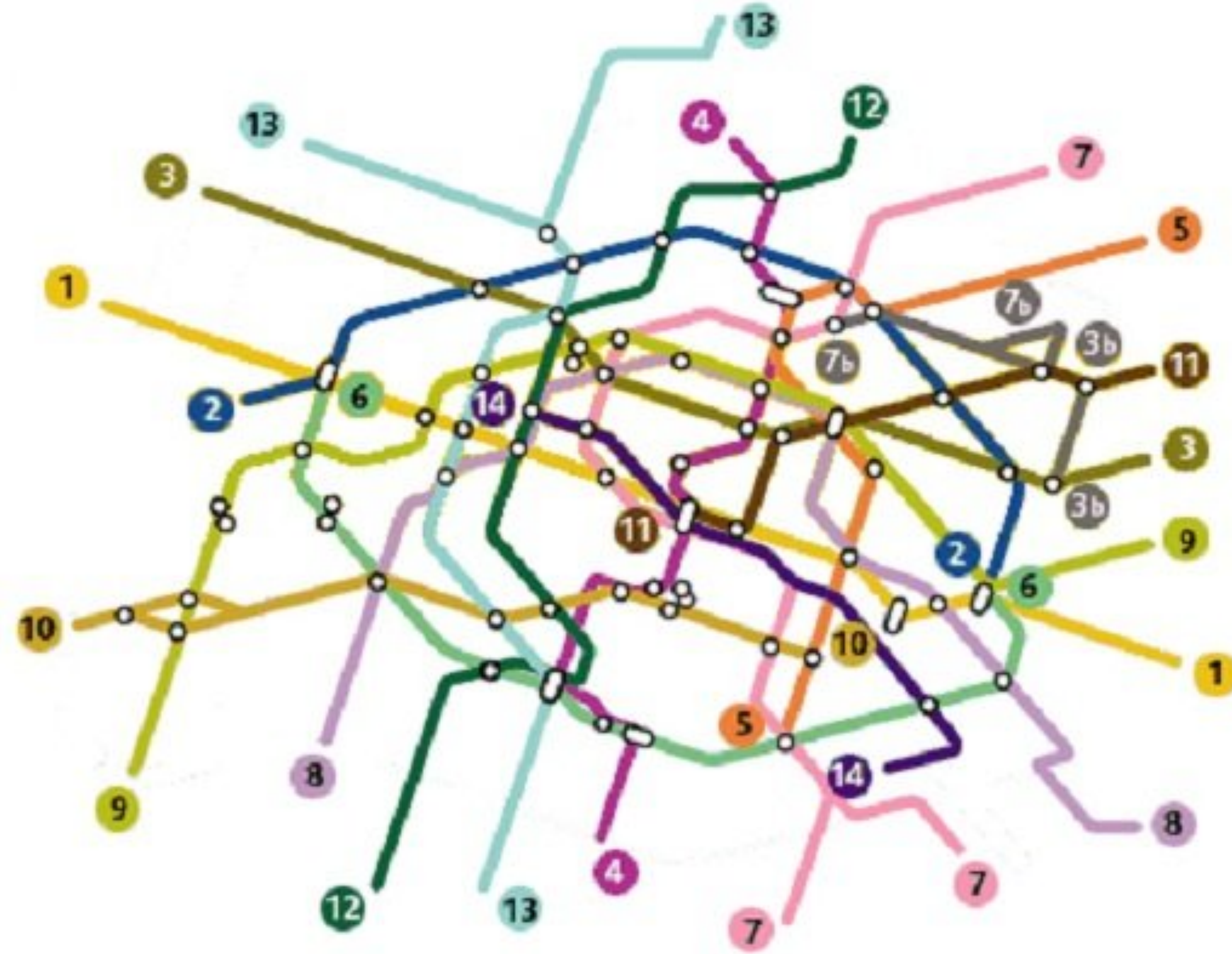
# Graphes

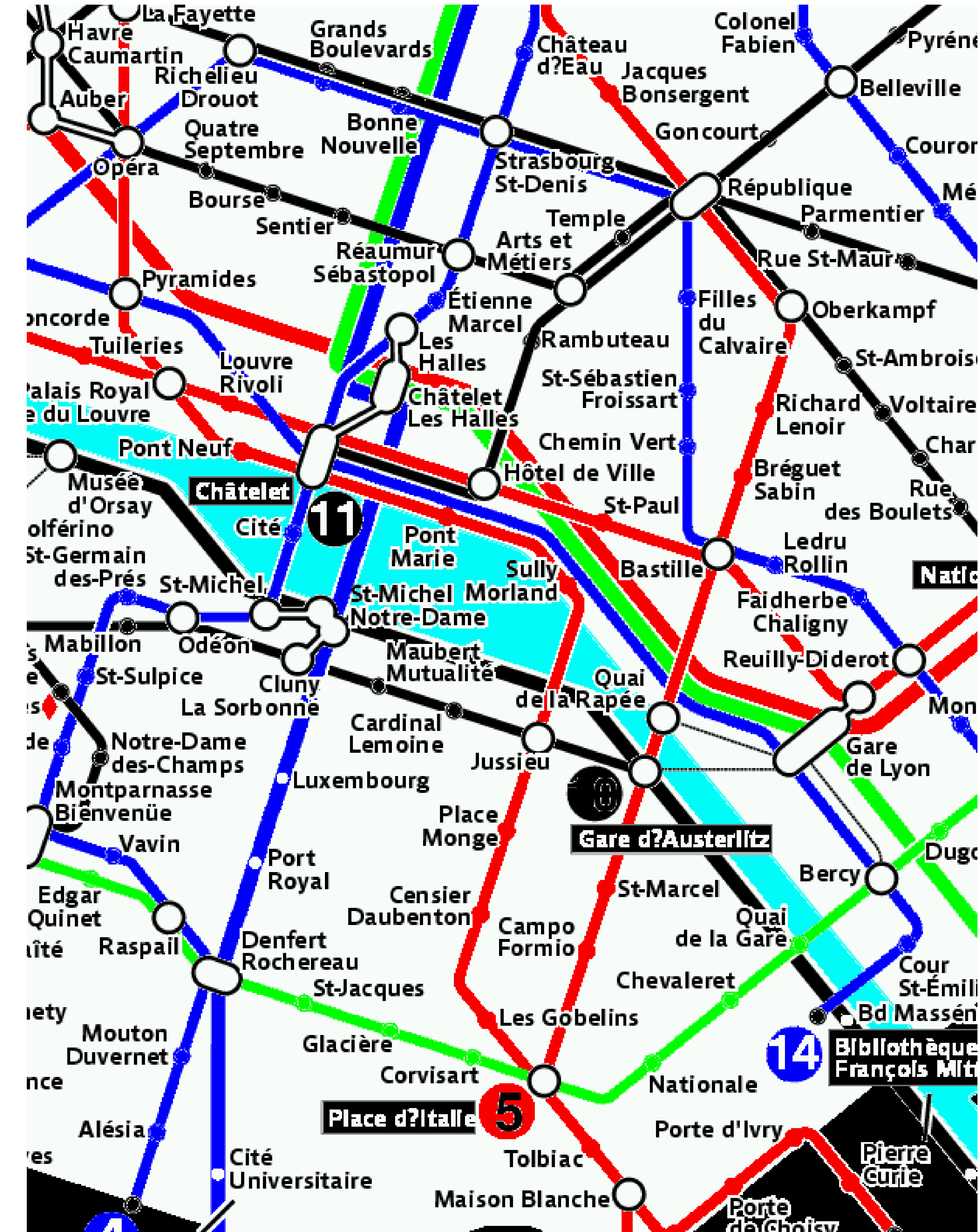


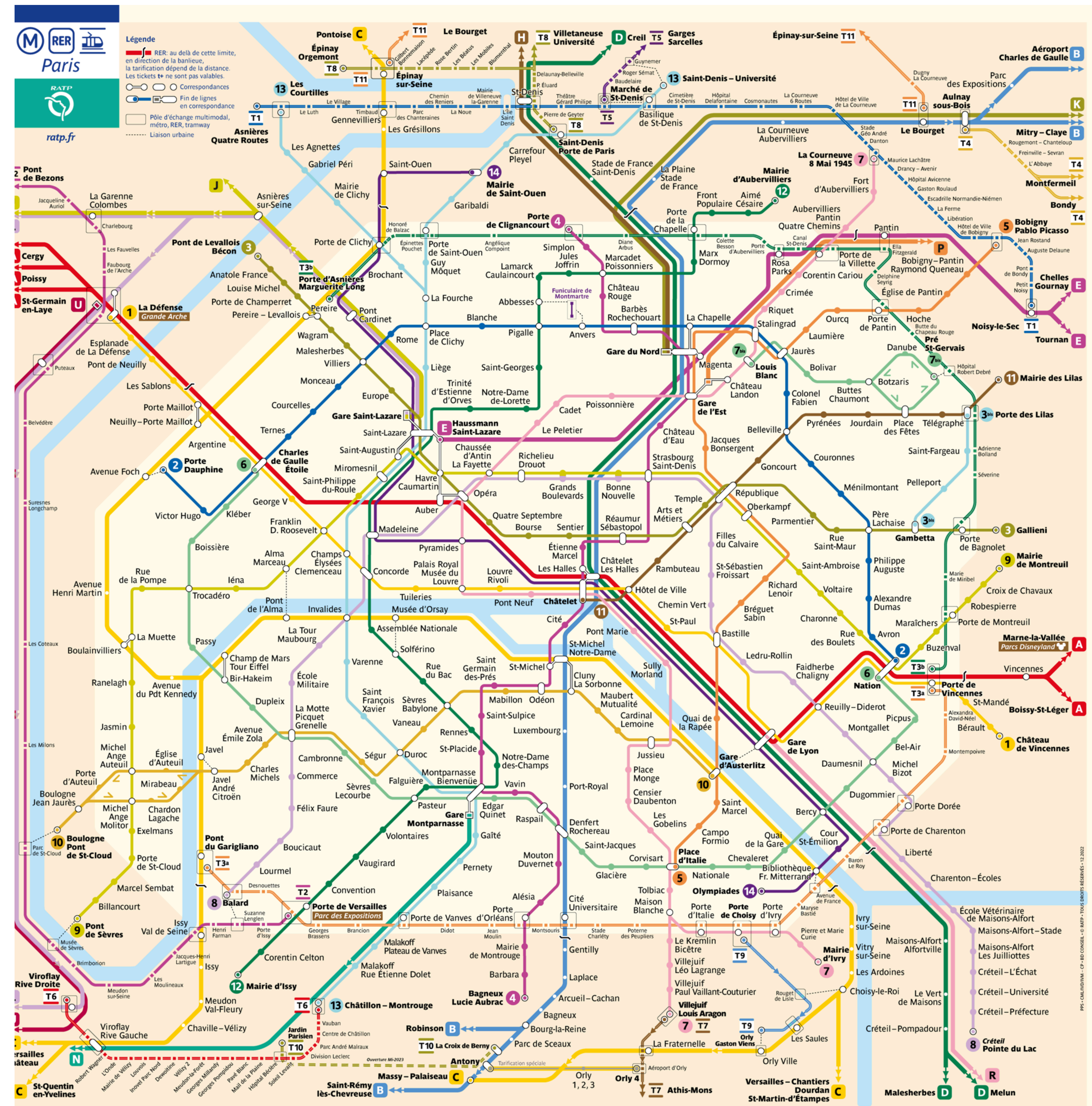

- carte routière et graphe des connexions entre villes avec la distance en km

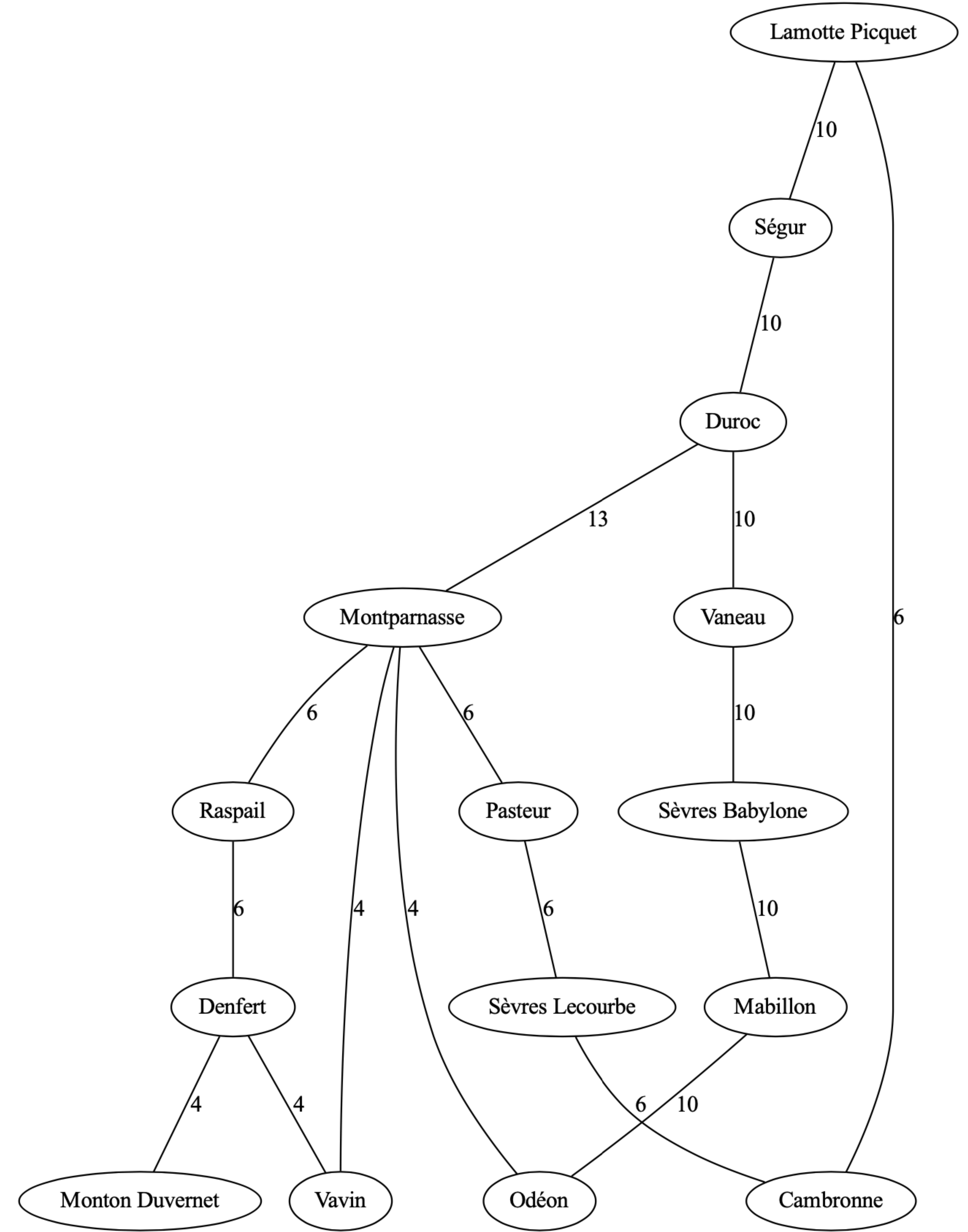
# Graphes



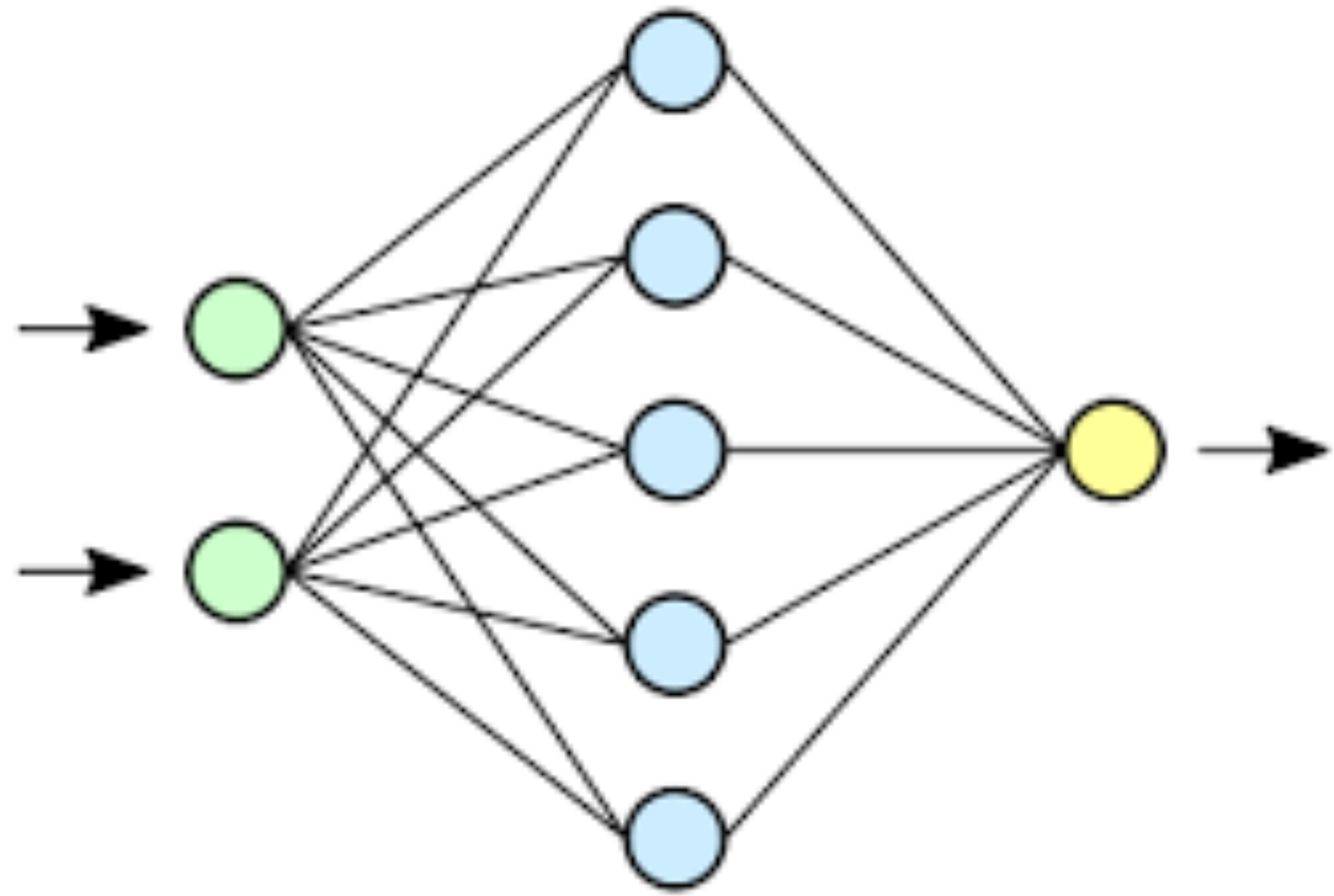
# Graphes

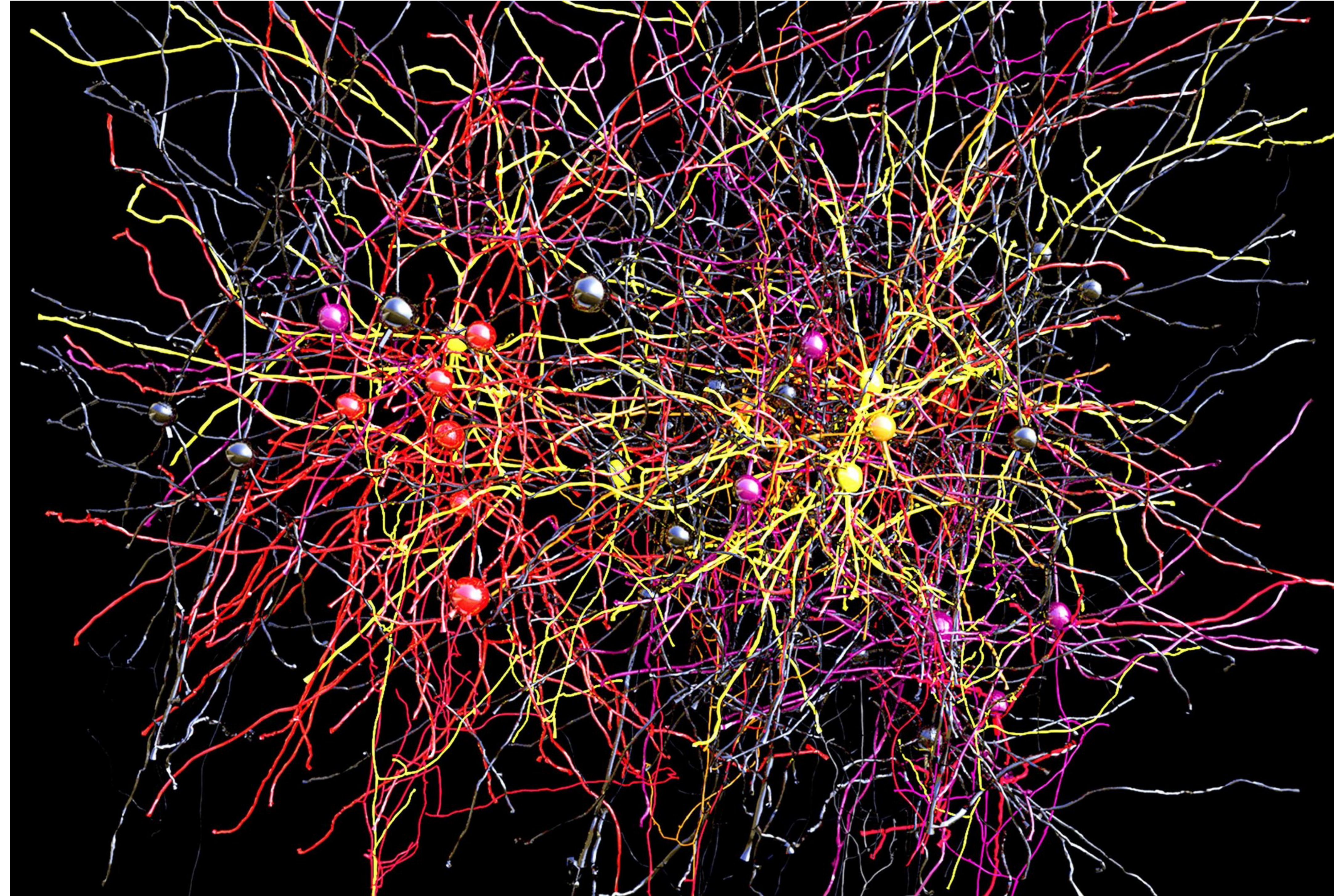





- plan du métro et le graphe qui relie les différentes stations

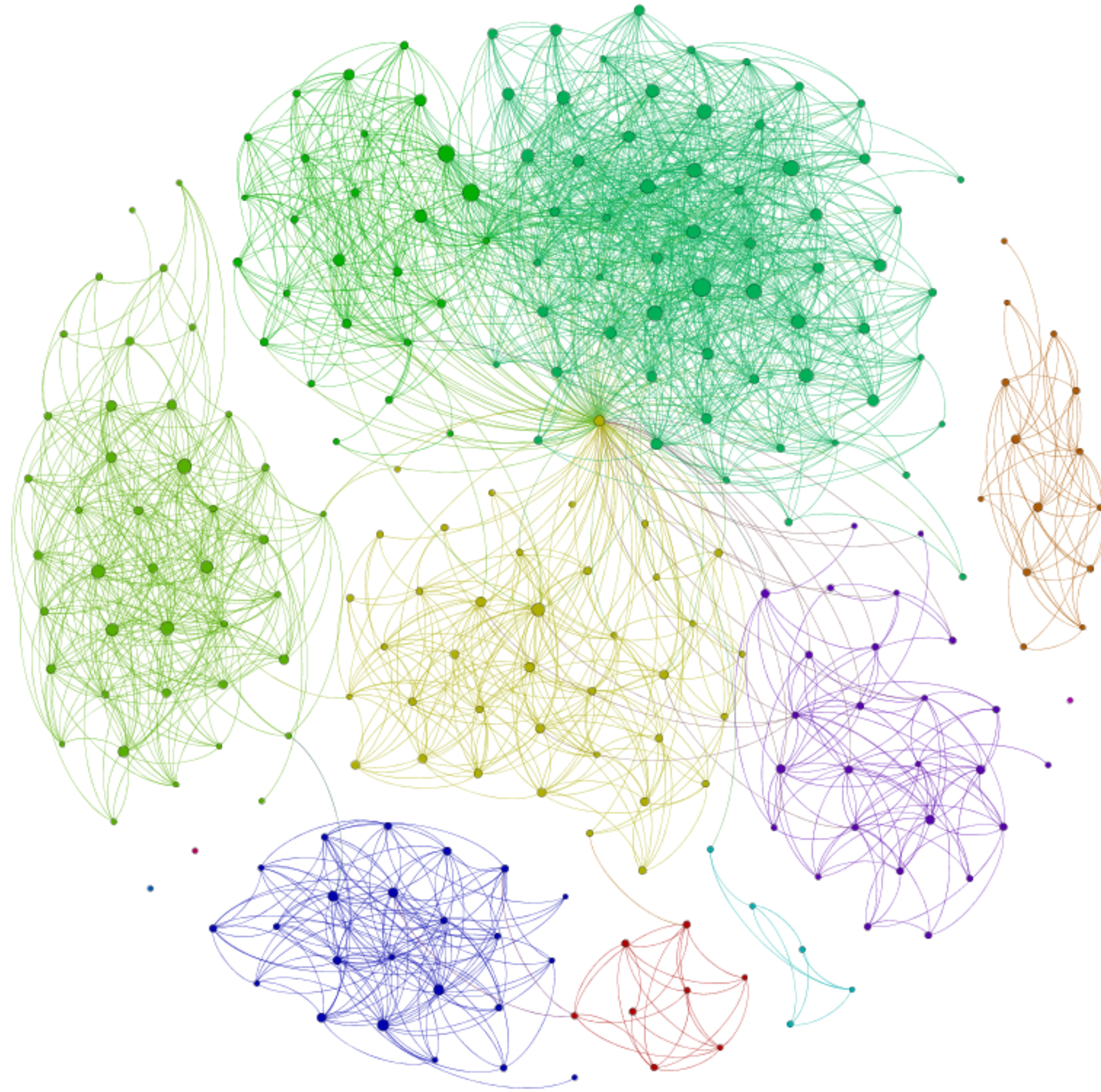
# Graphes





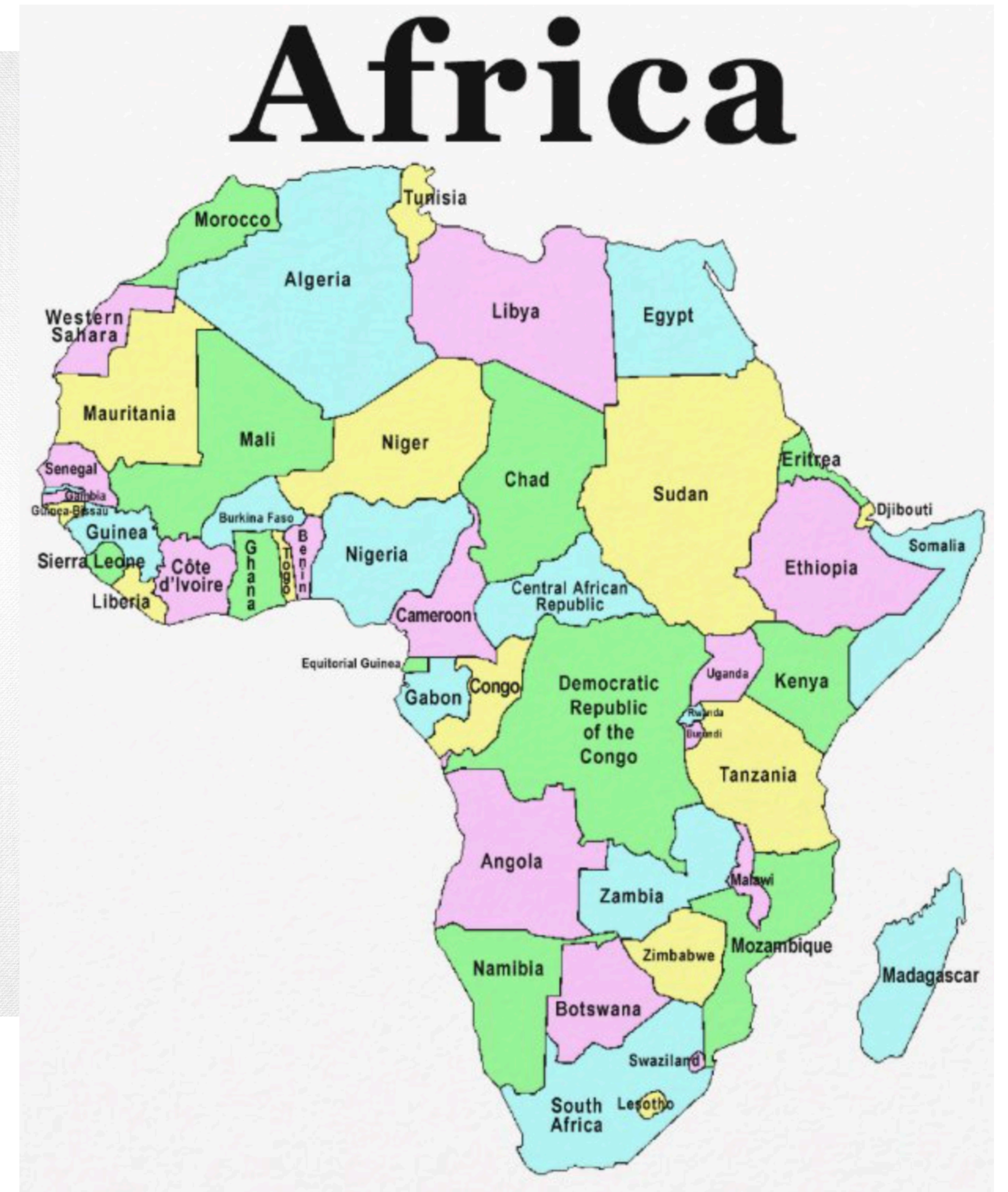
- Réseaux de neurone 2 couches et vrais réseaux de neurones biologiques

# Graphes



- Réseaux d’amis sur Facebook

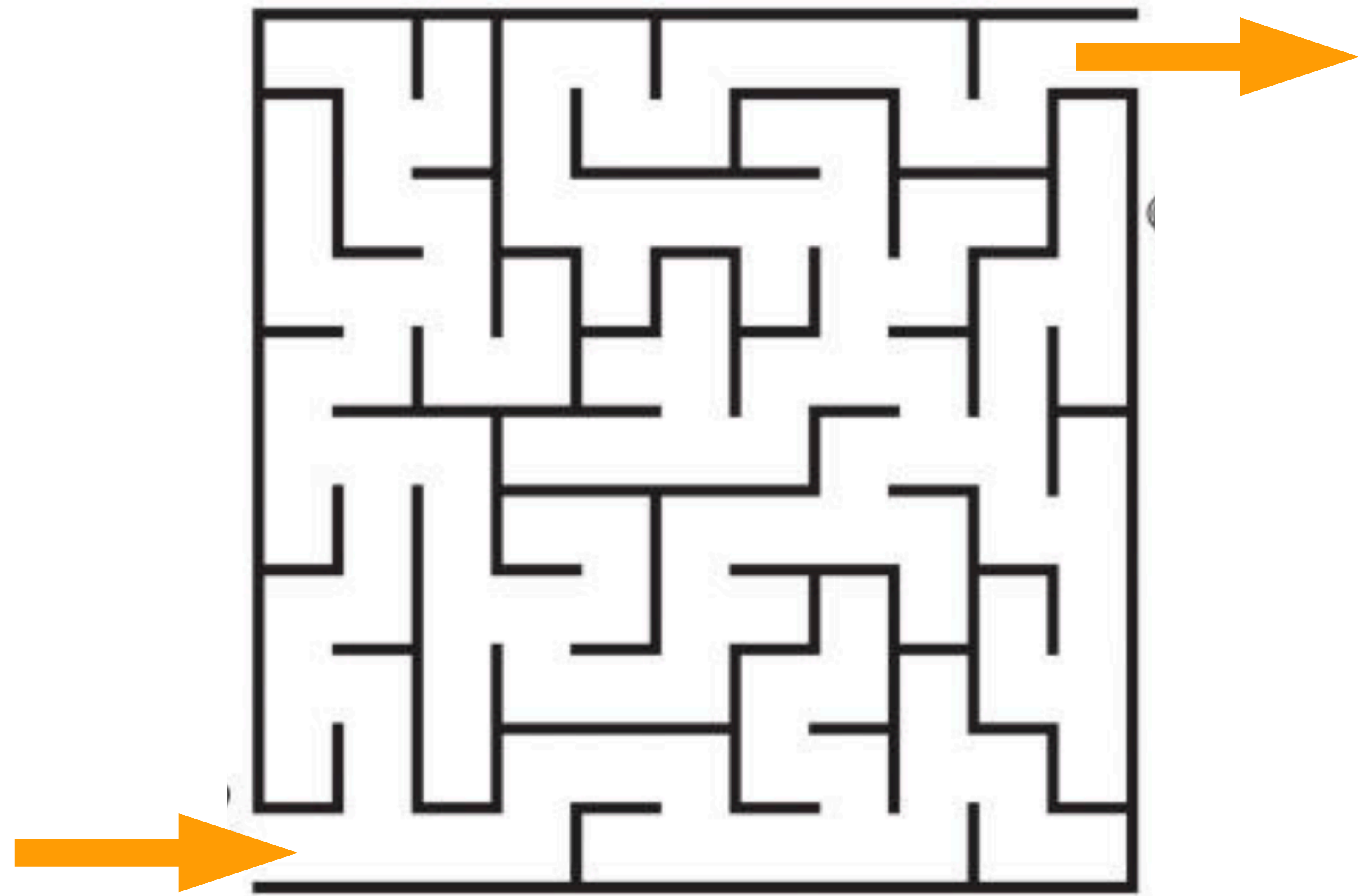
# Graphes

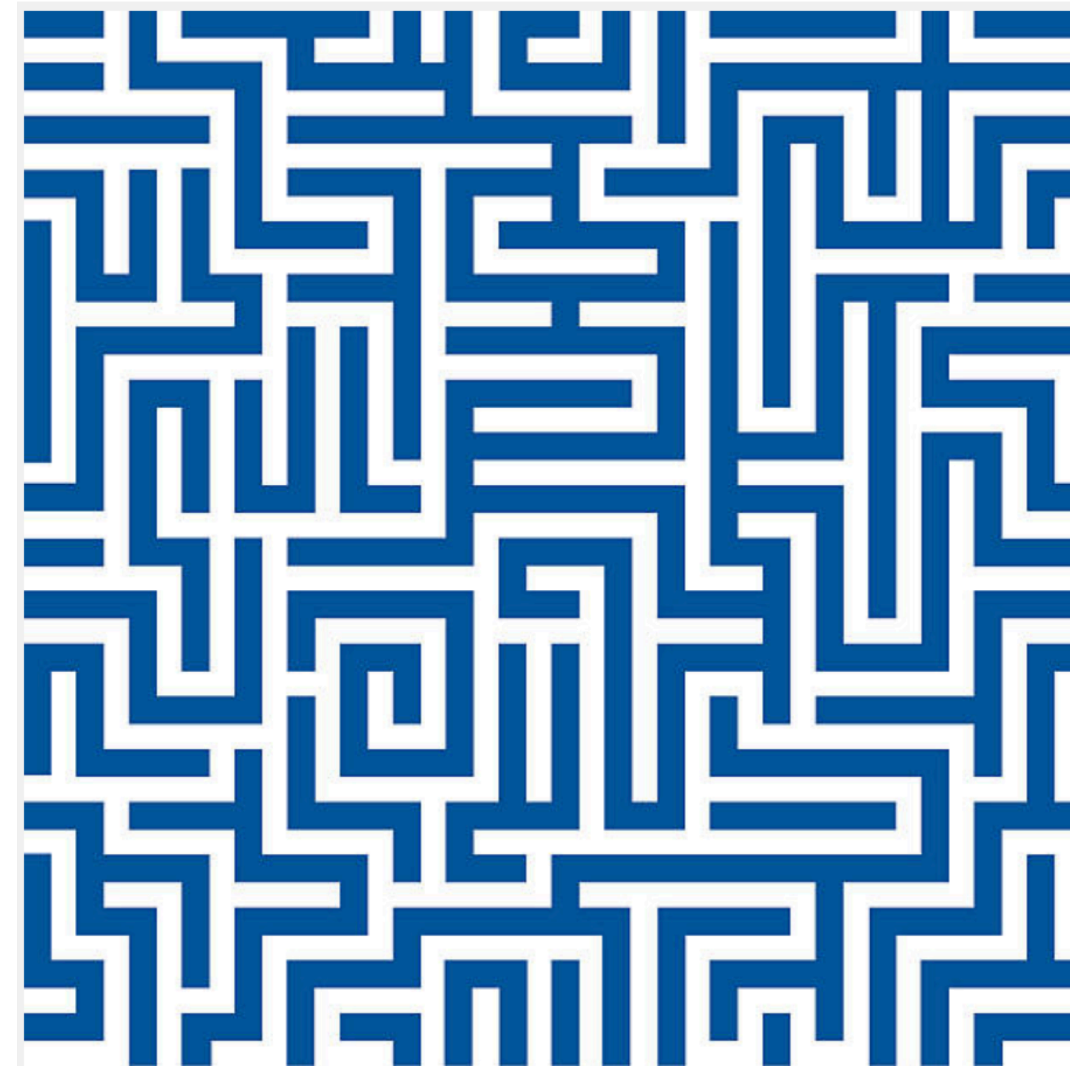


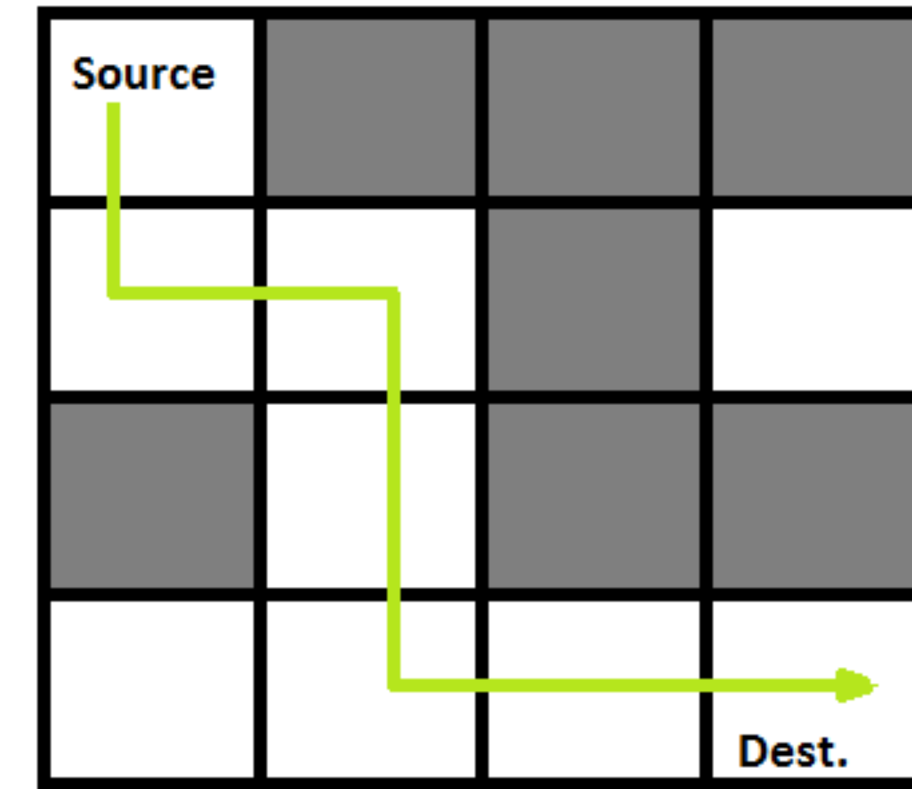

- Graphes planaires coloriables avec 4 couleurs

# Labyrinthes

- trouver le chemin vers la sortie !







# Graphes

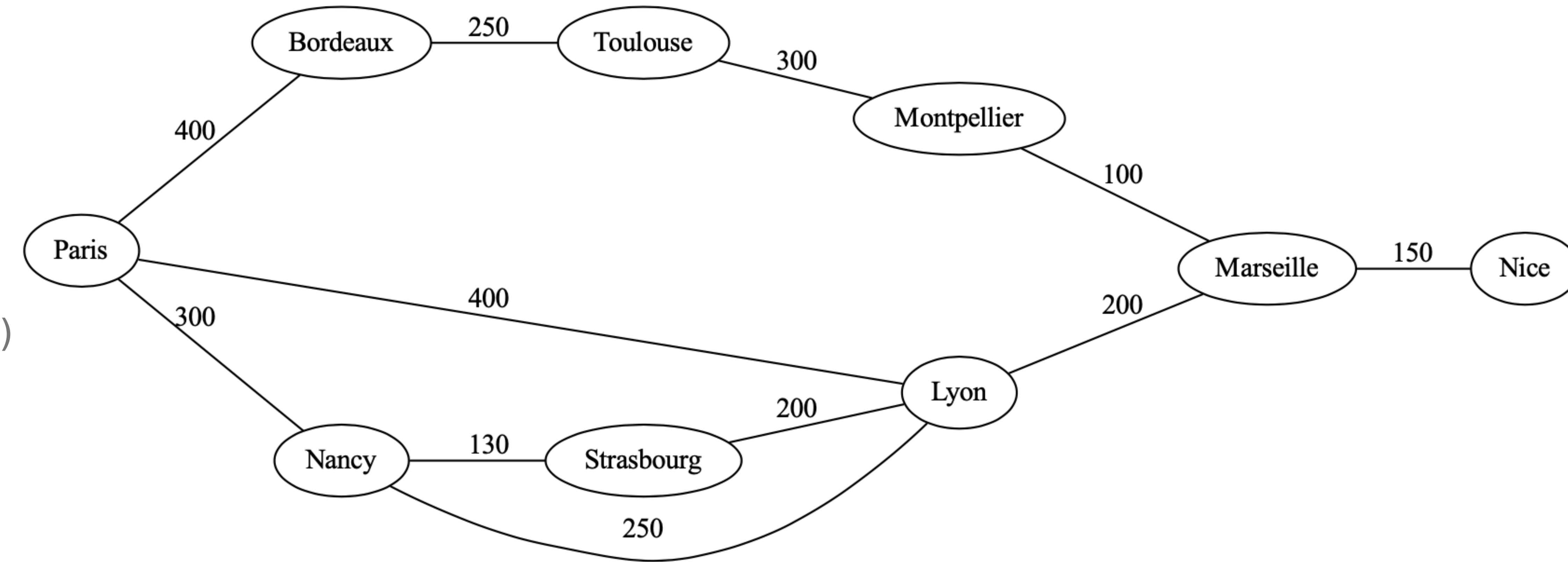
- un graphe est formé par un ensemble de **sommets** (*vertices*) et d'**arrêtes** (*edges*)
- une arrête relie 2 sommets (**origine** et **extrémité**)
- une graphe peut être **non dirigé** s'il existe une arrête (w, v) pour toute arrête (v, w)
- parfois on parle aussi d'**arcs** pour les arrêtes et de **noeuds** pour les sommets

**Remarque**: un arbre est un cas particulier d'un graphe non-dirigé

# Représentation des graphes

- on définit une classe pour les graphes

```
class Graph :
    def __init__ (self, vs) :
        self.vertices = vs
        self.succ = {w:[ ] for w in vs}

    def __str__ (self) :
        return 'sommets = {}\narretes = {}'.format (self.vertices, self.succ)

    def order (self) :
        return len (self.vertices)

    def add_edge (self, v, w) :
        self.succ[v] += [w]

    def successors (self, v) :
        return self.succ[v]
```



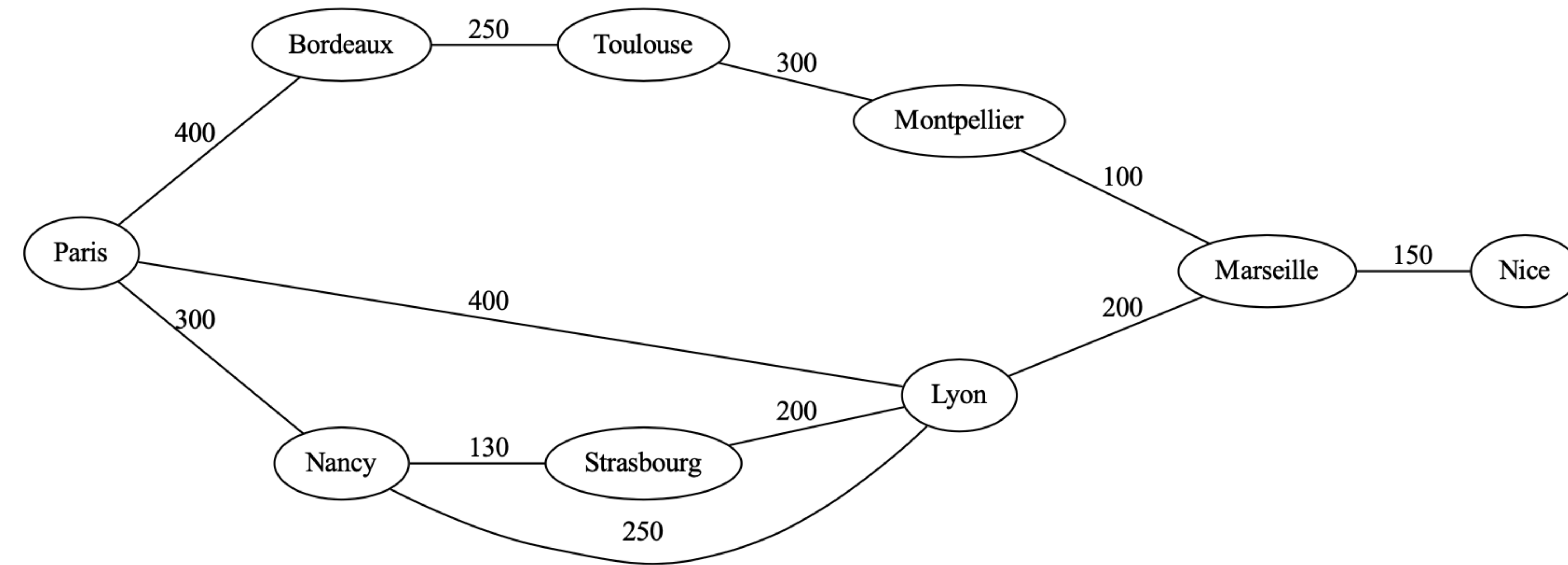

- et on construit un graphe (sans les distances)

```
g = Graph (['paris', 'lyon', 'marseille', 'toulouse', ‘nancy’, 'nice'])
g.add_edge ('paris', 'lyon')
g.add_edge ('lyon', ‘nancy')
. . .
```

# Représentation des graphes

- on définit une classe pour les graphes

```
class Graph :
    def __init__ (self, vs) :
        self.vertices = vs
        self.succ = {w:[ ] for w in vs}

    def __str__ (self) :
        res = 'sommets = {}\n'.format (self.vertices)
        res += 'arretes = '
        for v in self.vertices :
            for w in self.succ[v] :
                res += '{} '.format ((v, w))
        return res

    def order (self) :
        return len (self.vertices)

    def add_edge (self, v, w) :
        self.succ[v] += [w]

    def successors (self, v) :
        return self.succ[v]
```




- et on construit un graphe (sans les distances)

```
g = Graph (['paris', 'lyon', 'marseille', 'toulouse', ‘nancy’, 'nice'])
g.add_edge ('paris', 'lyon')
g.add_edge ('lyon', ‘nancy')
. . .
```

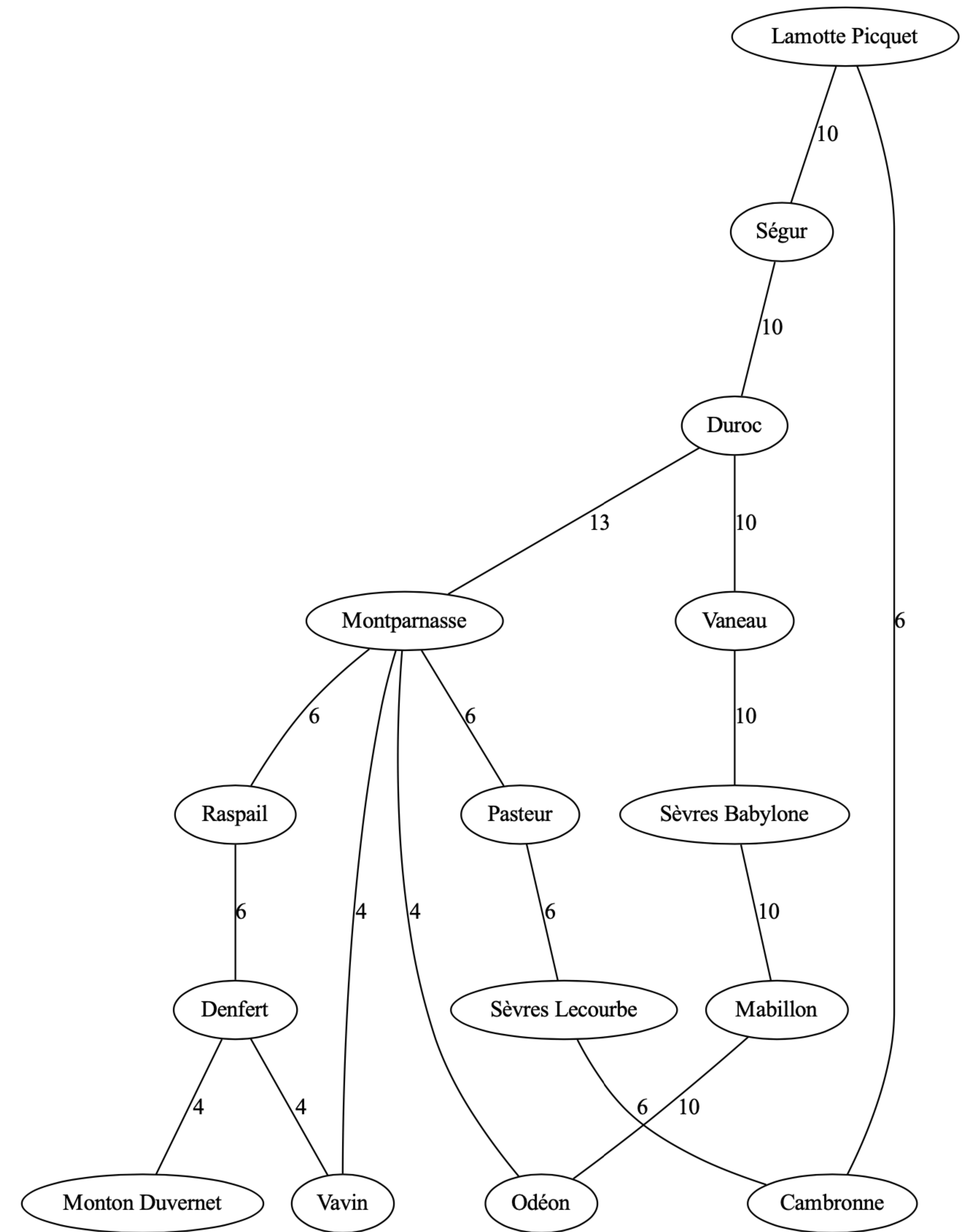
# Représentation des graphes

- on construit le graphe du métro parisien

```
stations = ['lamotte-picquet', 'segur', 'duroc', 'montparnasse' ]
metro = Graph (stations)
metro.add_edge ('segur', 'duroc')
metro.add_edge ('lamotte-picquet', 'segur')
metro.add_edge ('segur', 'duroc')
```

METRO-PARIS

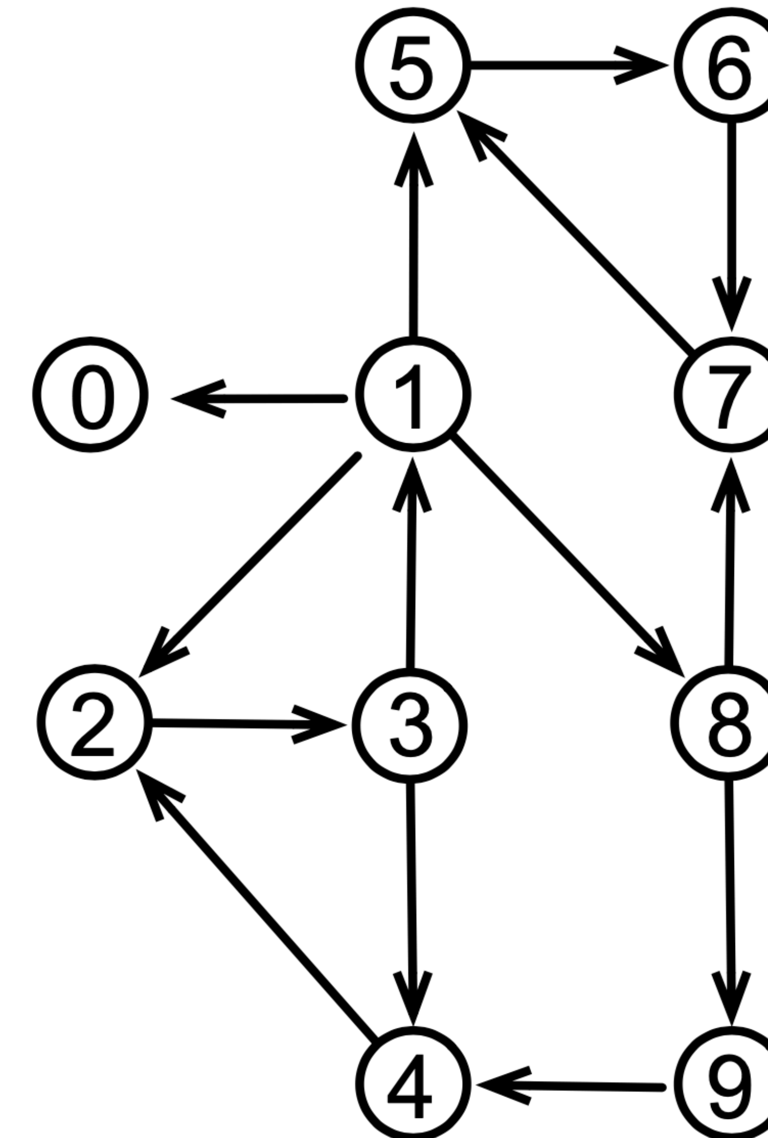
fichier texte qui contient les lignes de métro parisien



# Représentation des graphes

- un graphe plus abstrait

```
def add_edges (g, edges) :
    for (v, w) in edges :
        g.add_edge (v, w)


g2 = Graph ([i for i in range(10)])
add_edges (g2, [(1,0), (1,2), (1,5), (1,8), (2,3),
  (3,1), (3,4), (4,2), (5,6), (6,7),
  (7,5), (8,7), (8,9), (9,4)])
```




- quelques remarques sur la représentation :

  - utilisation des dictionnaires : représentation plus concise

  - utilisation des dictionnaires : représentation plus polymorphe

  - autre représentation : objets pour les sommets avec attributs `nom` et `successors`

  - autre représentation : sommets sont des nombres entiers et tableau de listes de successeurs

**Exercice** Ecrire la représentation pour ces deux dernières représentations.

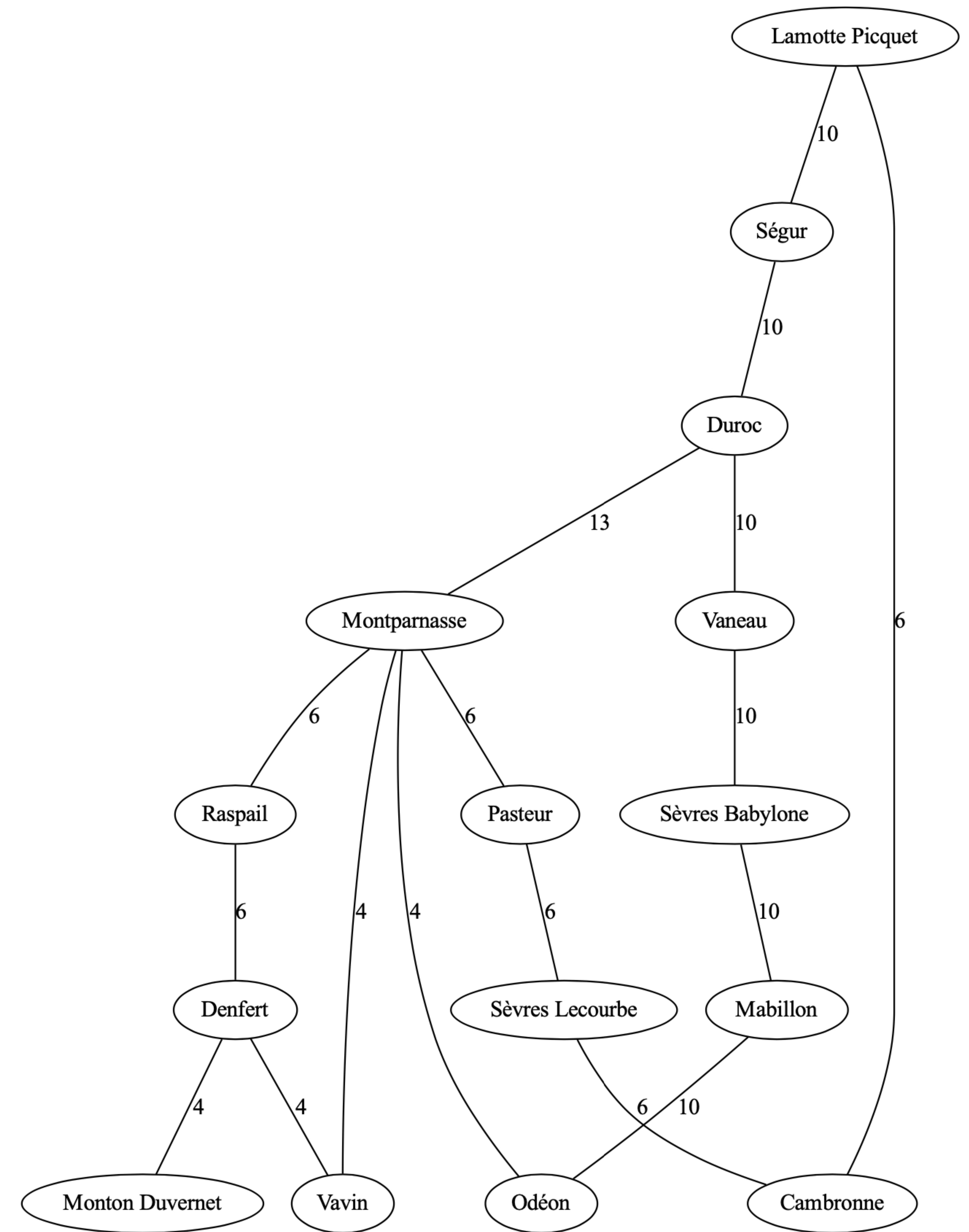
# Représentation des graphes

- on définit une sous-classe pour les graphes valués

```
class VGraph (Graph):
    def __init__ (self, vs) :
        super().__init__ (vs)

    def __str__ (self) :
        res = 'sommets = {}\n'.format (self.vertices)
        res += 'arretes = '
        for v in self.vertices :
            for (w, weight) in self.successors(v) :
                res += '{} '.format ((v, w, weight))
        return res

    def add_edge (self, v, w, weight) :
        self.succ[v] += [(w, weight)]
```

- et on construit un graphe (sans les distances)

```
metro = VGraph (stations)
metro.add_edge ('segur', 'duroc', 10)
metro.add_edge ('lamotte-picquet', 'segur', 10)
metro.add_edge ('segur', 'duroc', 10)
print (metro)
```



# Prochain cours

- graphes et matrices d’adjacences
- parcours de graphe
- arbres de recouvrement
- recherche de chemins