Informatique et Programmation

Cours 13

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-py

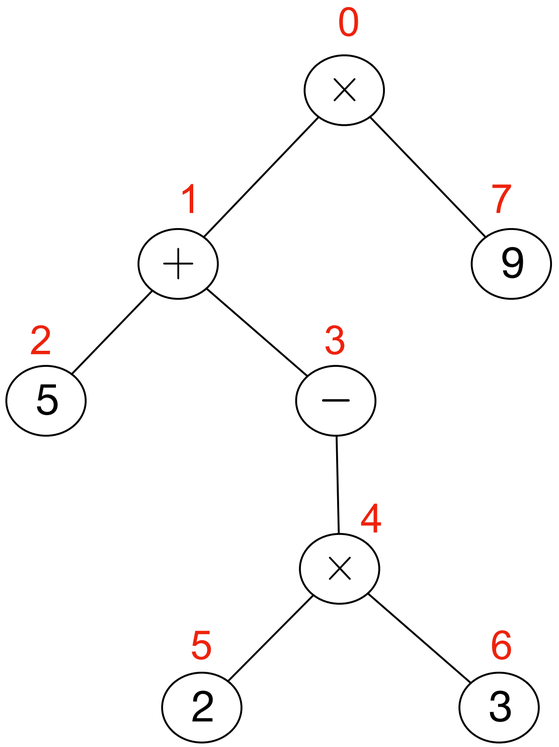
Plan

- ordres de parcours d’arbre
- belle impression
- arbres de syntaxe abstraite
- graphes
- graphes (2 représentations)

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

Parcours d’arbre

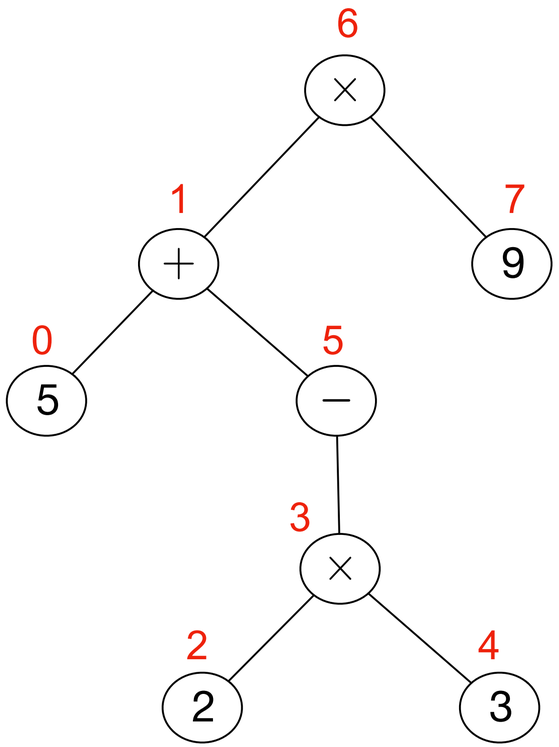
• 3 parcours d’arbre (préfixe, infixe, postfixe)



préfixe

• notation polonaise préfixe

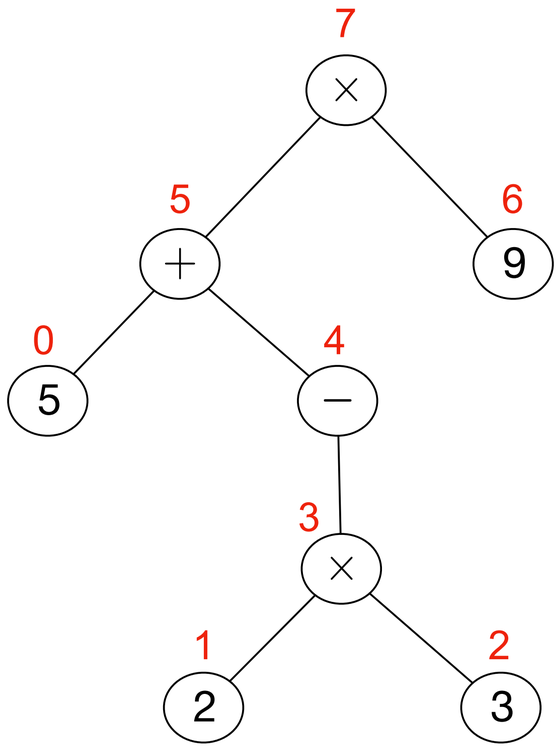
x + 5 - x 2 3 9



infixe

• notation arithmétique

(5 + - (2 x 3)) x 9



postfixe

• notation polonaise postfixe

5 2 3 x - + 9 x

Parcours d'arbre

- générer les notations préfixe, postfixe et infixe

```
def polprefix (a) :
    if isinstance (a, Feuille) :
        return a.val
    elif isinstance (a, Noeud_Un) :
        return a.val + ' ' + polprefix (a.fils)
    else :
        return a.val \
              + ' ' + polprefix (a.gauche) \
              + ' ' + polprefix (a.droit)
```

```
def polpostfix (a) :
    if isinstance (a, Feuille) :
        return a.val
    elif isinstance (a, Noeud_Un) :
        return polpostfix (a.fils) + ' ' + a.val
    else :
        return polpostfix (a.gauche) \
             + ' ' + polpostfix (a.droit) \
             + ' ' + a.val
```

```
def notinfixe (a) :
    if isinstance (a, Feuille) :
        return a.val
    elif isinstance (a, Noeud_Un) :
        return '(' + a.val + ' ' + notinfixe (a.fils) + ')'
    else :
        return '(' + notinfixe (a.gauche) \
             + ' ' + a.val \
             + ' ' + notinfixe (a.droit) + ')'
```

trop de parenthèses

[on verra plus tard pour les enlever]

- notation polonaise préfixe

  x + 5 - x 2 3 9

- notation infixe

  (5 + - (2 x 3)) x 9

- notation polonaise postfixe

  5 2 3 x - + 9 x

Parcours d'arbre

• générer les notations **préfixe**, postfixe et infixe

```
def polprefix (a) :
    if isinstance (a, Feuille) :
        return a.val
    elif isinstance (a, Noeud_Un) :
        return a.val + ' ' + polprefix (a.fils)
    else :
        return a.val \
              + ' ' + polprefix (a.gauche) \
              + ' ' + polprefix (a.droit)
```

concaténation des chaînes de caractères

rappel Python: (surcharge de +)

```
3 + 5          8

'a' + 'b'      'ab'

[1, 2] + [3, 4]      [1, 2, 3, 4]
```

• correspond au parcours **préfixe**

• notation polonaise préfixe

x + 5 - x 2 3 9

Parcours d’arbre

- générer les notations préfixe, postfixe et infixe

```
def polpostfix (a) :
    if isinstance (a, Feuille) :
        return a.val
    elif isinstance (a, Noeud_Un) :
        return polpostfix (a.fils) + ' ' + a.val
    else :
        return polpostfix (a.gauche) \
       + ' ' + polpostfix (a.droit) \
       + ' ' + a.val
```

- correspond au parcours **postfixe**

- notation polonaise postfixe

  5 2 3 x - + 9 x

Parcours d'arbre

- générer les notations préfixe, postfixe et infixe

```
def notinfixe (a) :
    if isinstance (a, Feuille) :
        return a.val
    elif isinstance (a, Noeud_Un) :
        return '(' + a.val + ' ' + notinfixe (a.fils) + ')'
    else :
        return '(' + notinfixe (a.gauche) \
       + ' ' + a.val \
       + ' ' + notinfixe (a.droit) + ')'
```

trop de parenthèses

[on verra plus tard pour les enlever]

- correspond au parcours **infixe**

- notation infixe

  (5 + - (2 x 3)) x 9

Arbres - Belle impression

- on peut réduire le nombre de parenthèses si on connait la précédence des opérateurs

- en mathématiques, ‘*’ a une plus forte précédence que ‘+’

$$3 + 4 \times 5 \quad \equiv \quad 3 + (4 \times 5)$$

$$(11 + 3) * 4 \quad \not\equiv \quad 11 + 3 * 4$$

- on peut donc faire le dictionnaire suivant de précédences: `preds = {'+': 0, '*': 2, '-': 3}`

- la fonction d’impression met des parenthèses si la précédence est inférieure à la précédence du contenant

Arbres - Belle impression

- on peut réduire le nombre de parenthèses si on connait la précédence des opérateurs

```
preds = {'+': 0, '*': 2, '-': 3}

def notinfix (a, p) :
    if isinstance (a, Feuille) :
        return a.val
    elif isinstance (a, Noeud_Un) :
        q = preds[a.val]
        if p > q :
            return '(' + ' ' + a.val \
         + ' ' + notinfix (a.fils, q) + ')'
        else :
            return a.val + ' ' + notinfix (a.fils, q)
    else :
        q = preds[a.val]
        if p > q :
          return '(' + notinfix (a.gauche, q) \
         + ' ' + a.val \
         + ' ' + notinfix (a.droit, q) + ')'
        else :
            return notinfix (a.gauche, q) \
         + ' ' + a.val \
         + ' ' + notinfix (a.droit, q)
```

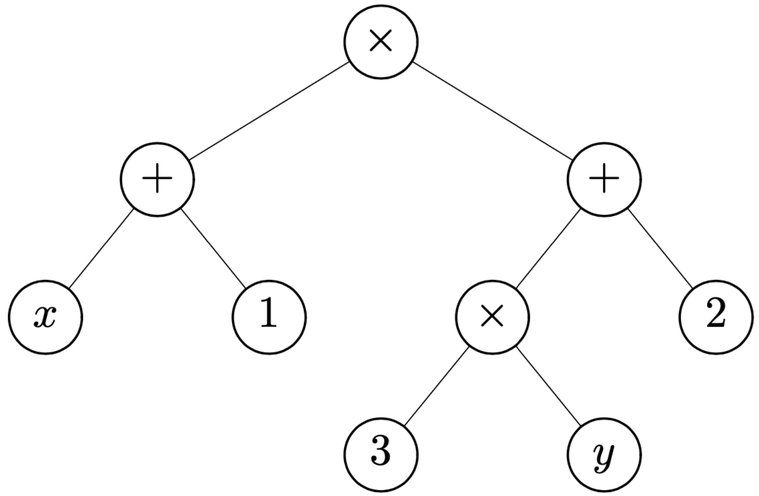
- et on imprime avec la précédence 0

Arbres de syntaxe abstraite

- passer d'une chaîne de caractères à un arbre (syntaxe astraite) est plus difficile

$(x + 1) * (3 * y + 2)$

Analyse syntaxique



Belle impression

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

class Op_un:
    def __init__ (self, x, a) :
        self.val = x
        self.fils = a

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

```
t = Op_bi ('*', Op_bi ('+', CVar ('x'), CVar (1)),
          Op_bi ('+', Op_bi ('*', CVar (3), CVar ('y')),
                      CVar (2)))
```

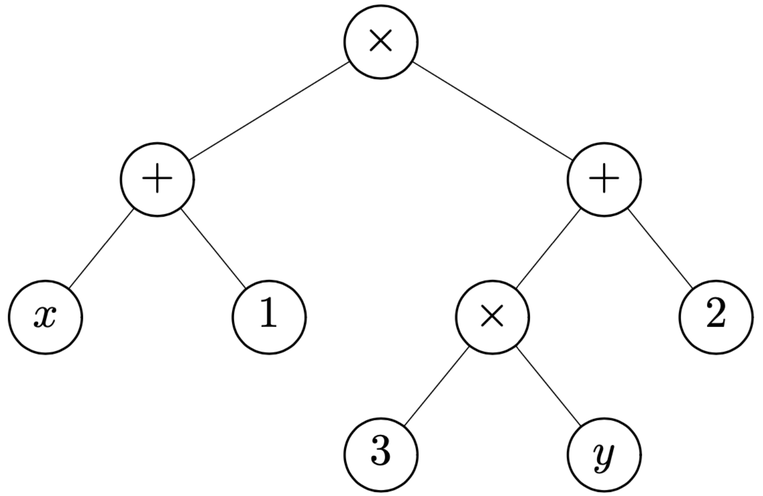
Arbres de syntaxe abstraite

- on peut évaluer sa valeur en donnant une valeur aux variables *x* et *y*

- on définit l'environnement par le dictionnaire:

```
e = {'x' : 20, 'y' : -20}
```

```
def eval (t, e) :
    if isinstance (t, Op_bi) :
        if t.val == '+' :
            return eval (t.gauche, e) + eval (t.droit, e)
        elif t.val =='*' :
            return eval (t.gauche, e) * eval (t.droit, e)
    elif isinstance (t, Op_un) :
        return - eval (t.fils, e)
    elif isinstance (t.val, int) :
        return t.val
    else :
        return e[t.val]
```




```
t = Op_bi ('*', Op_bi ('+', CVar ('x'), CVar (1)),
           Op_bi ('+', Op_bi ('*', CVar (3), CVar ('y')),
                       CVar (2)))
```

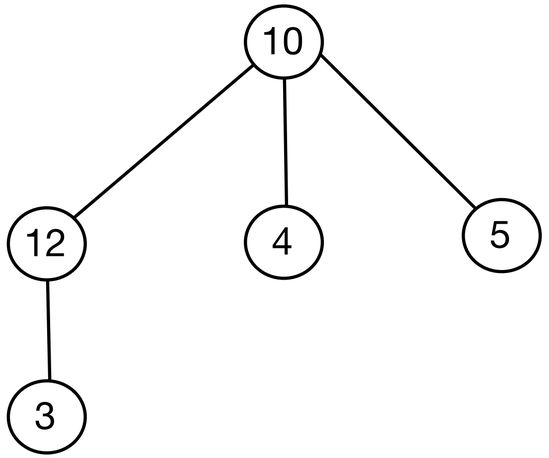
```
print (eval (t, e))
```

Arbres (représentation 5)

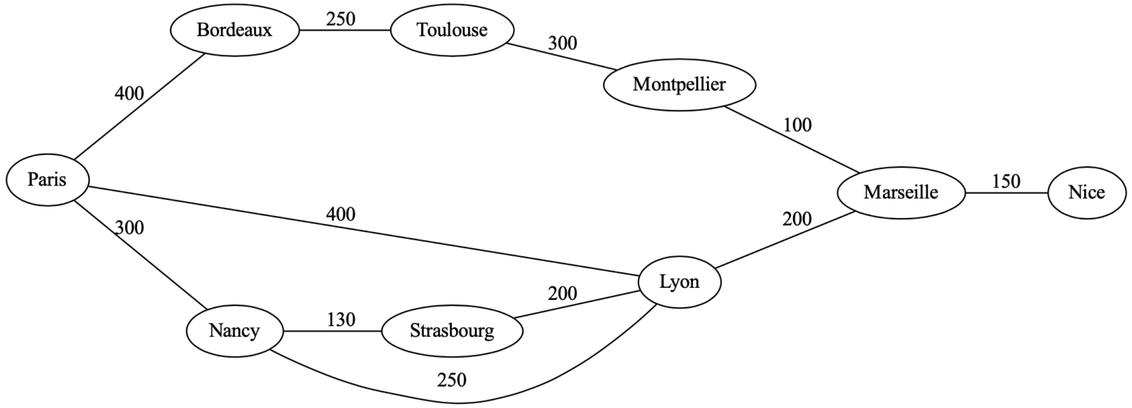
- Arbres n-aires avec nombre arbitraire de fils (rangés dans une liste)

```
class Noeud:
    def __init__ (self, x, l) :
        self.val = x
        self.fils = l
    #
    def __str__ (self) :
        r = ''
        for a in self.fils :
            r = r + ', ' + str(a)
        return "Noeud ({}, [ {} ])".format (self.val, r[2:])

class Feuille:
    def __init__ (self, x) :
        self.val = x
    #
    def __str__ (self) :
        return "Feuille ({})".format (self.val)

a = Noeud (10,
        [ Noeud (12, [ Feuille (3) ]),
          Feuille (4), Feuille (5) ])
print (a)
```

```
def __str__ (self) :
    r = ', '.join (map(str, self.fils))
    return "Noeud ({}, [ {} ])".format (self.val, r)
```

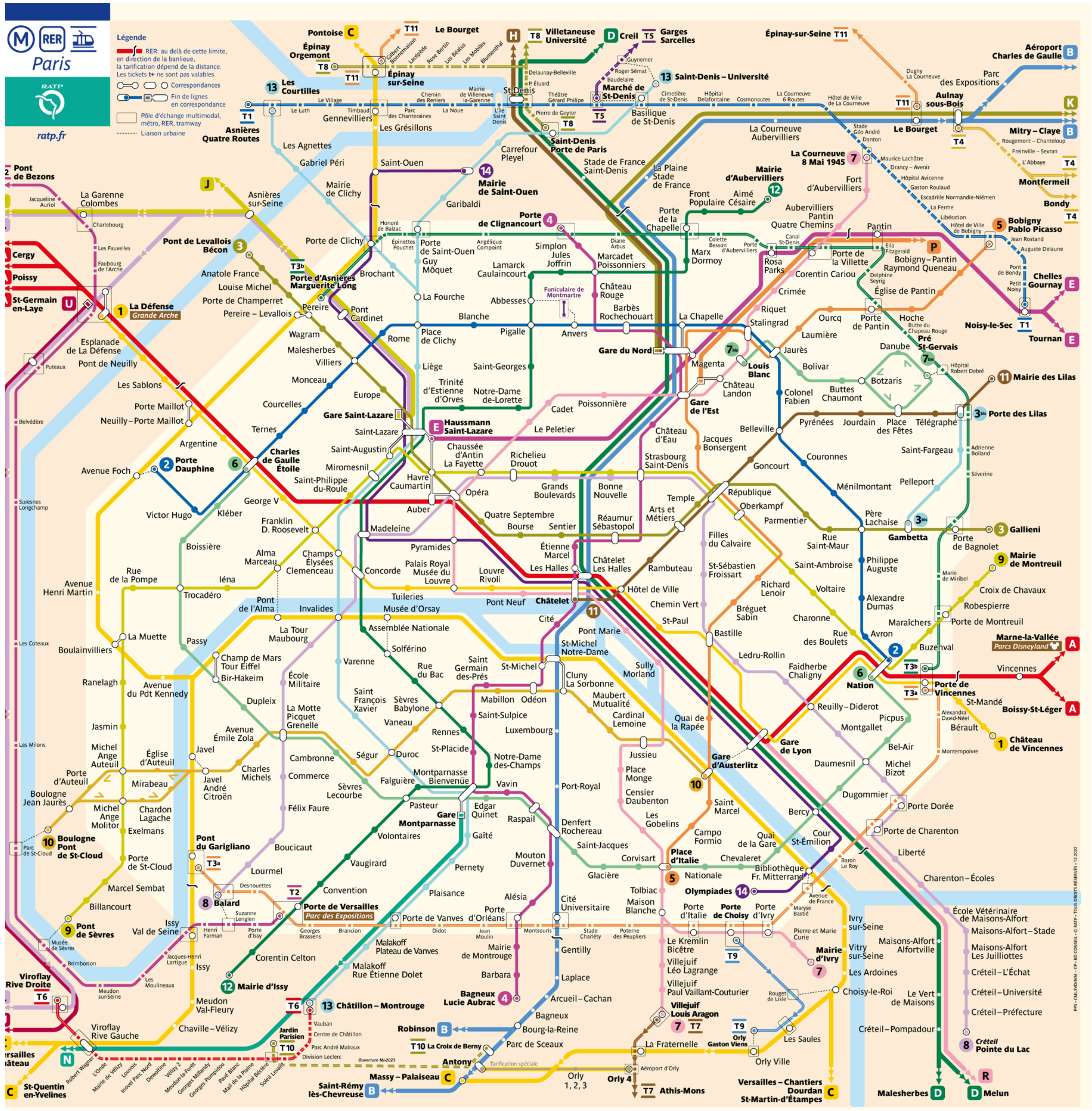


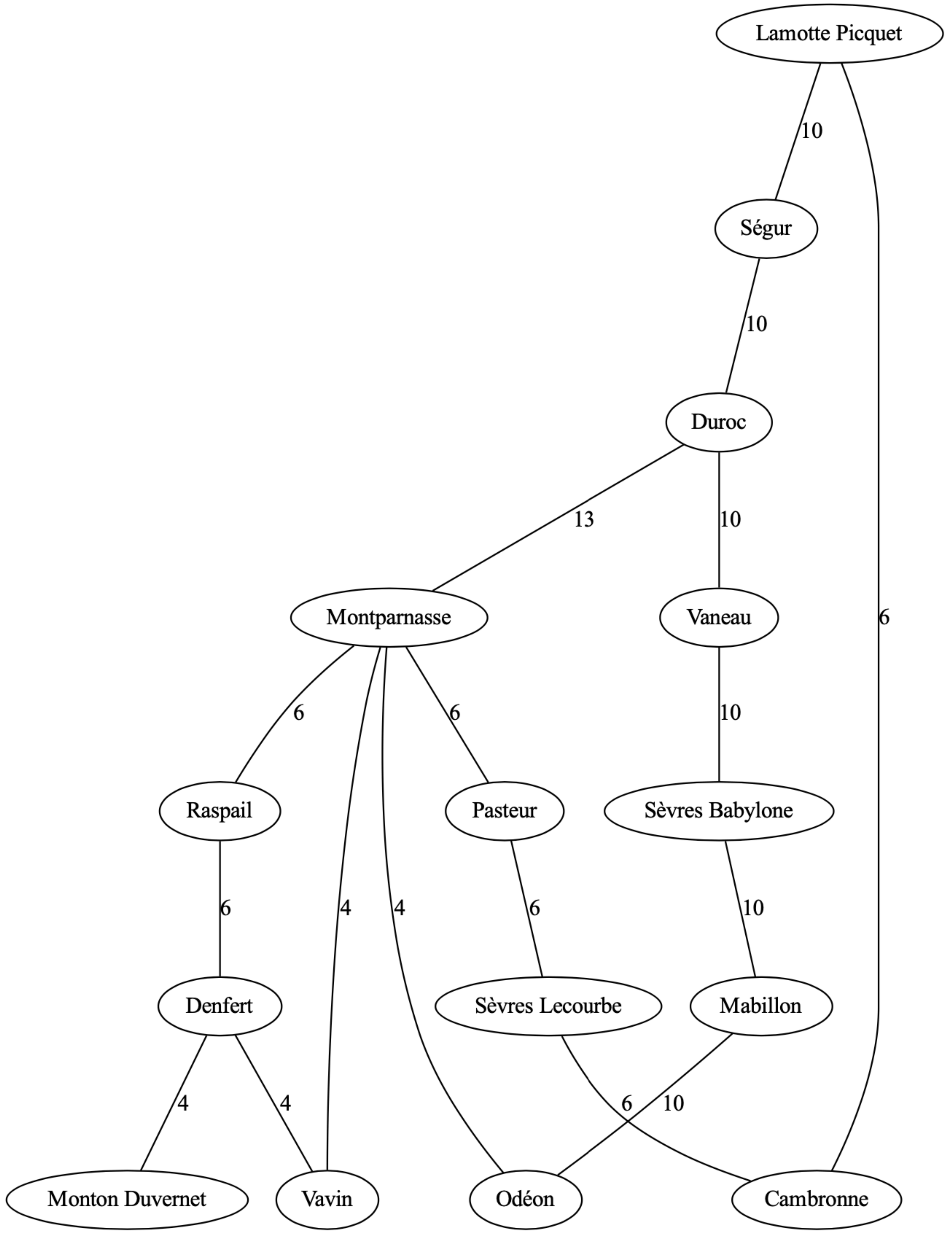
Graphes



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

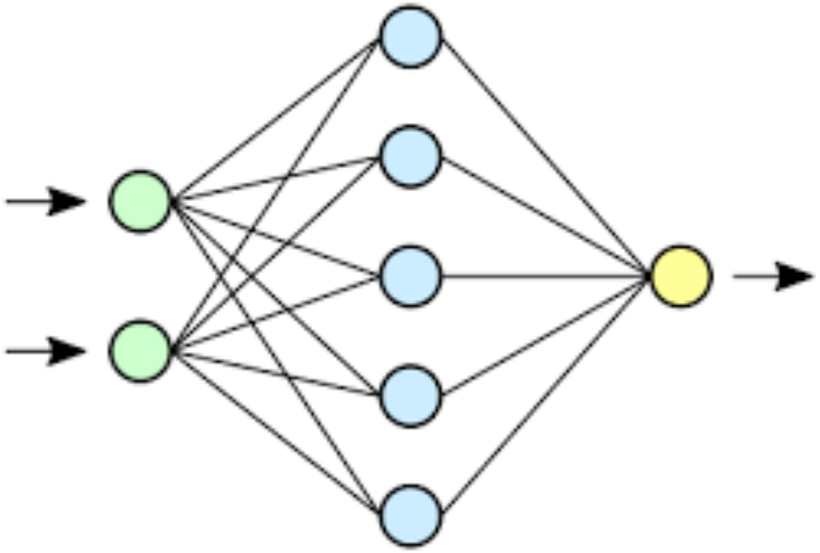
Graphes

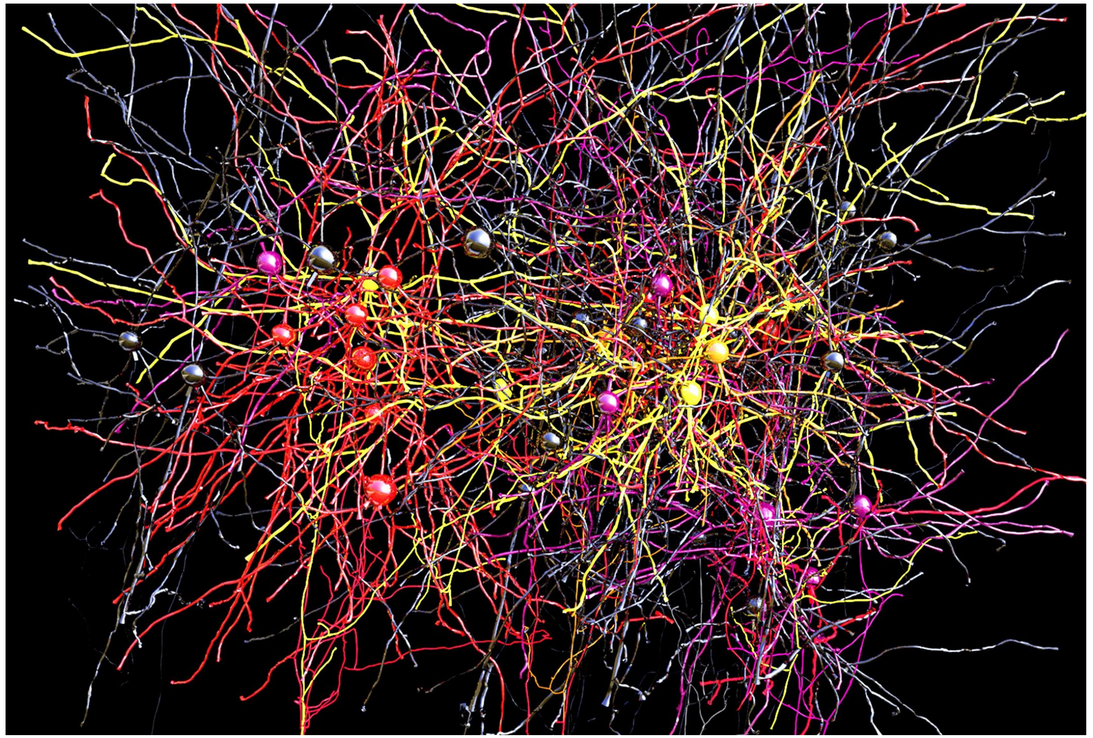




- plan du métro et le graphe qui relie les différentes stations

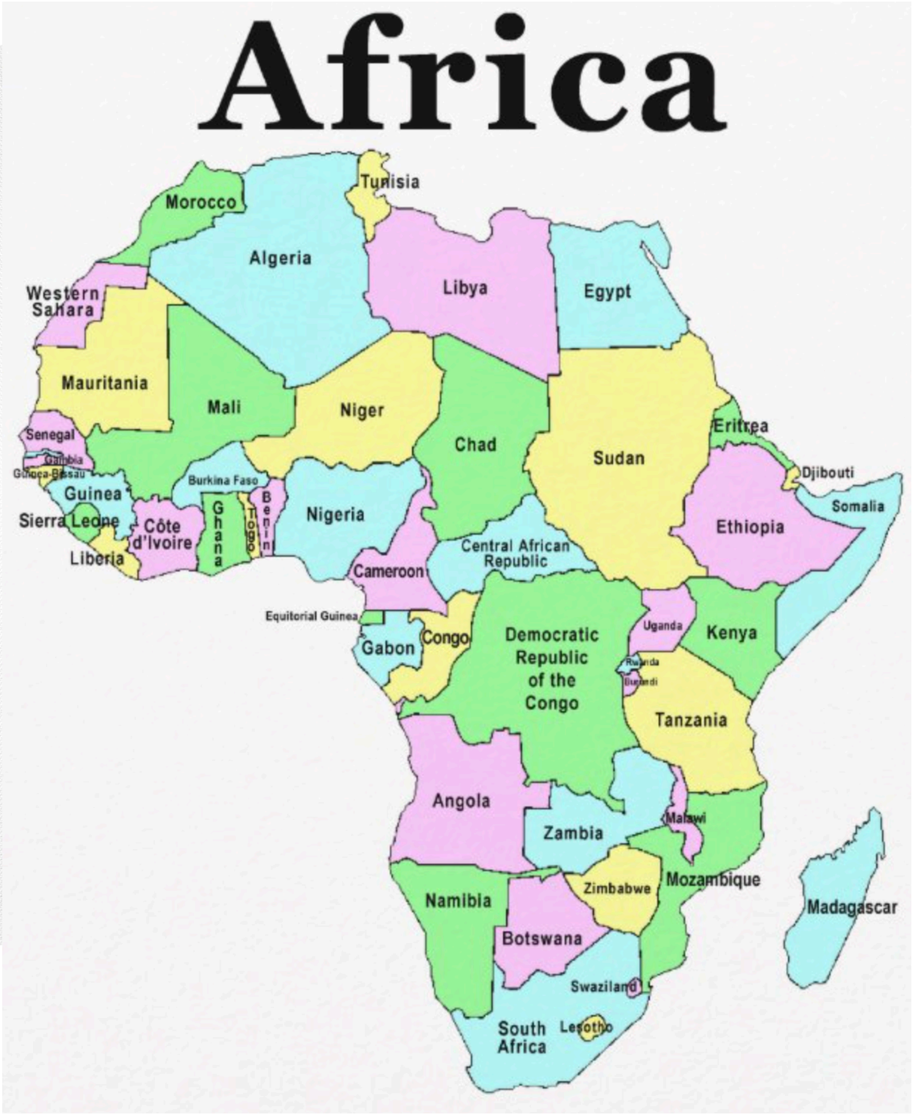
Graphes





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

Graphes

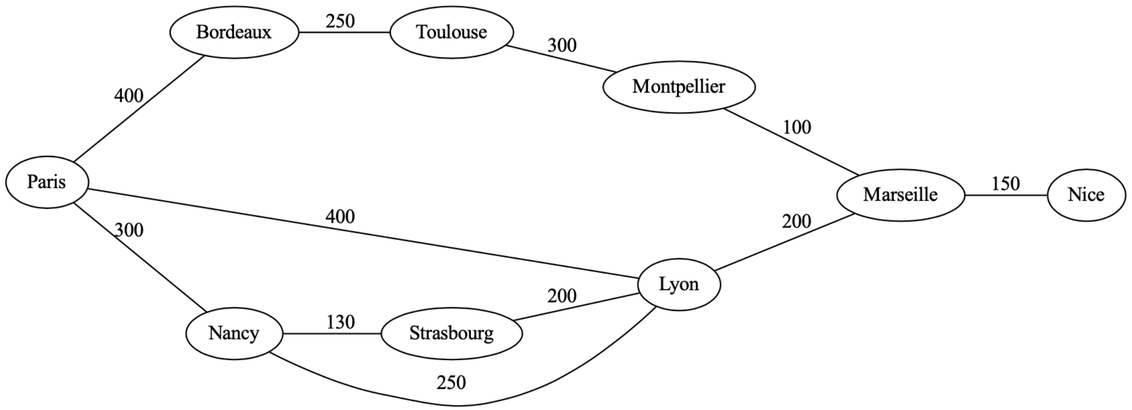


- Graphes planaires coloriables avec 4 couleurs

Graphes (représentation 1)

- représentation par matrice d'adjacence

```
villes = [ 'Paris', 'Bordeaux', 'Toulouse', 'Montpellier',
     'Marseille', 'Nancy', 'Strasbourg', 'Lyon', 'Nice']
nVilles = len (villes)


graphe =  new_matrix (nVilles, nVilles, None)


def arc (g, v1, v2, d) :
    i = index_of (villes, v1)
    j = index_of (villes, v2)
    g[i][j] = g[j][i] = d


arc (graphe, 'Paris', 'Bordeaux', 400)
arc (graphe, 'Bordeaux', 'Toulouse', 250)
arc (graphe, 'Toulouse', 'Montpellier', 300)
arc (graphe, 'Montpellier', 'Marseille', 100)
arc (graphe, 'Paris', 'Lyon', 400)
arc (graphe, 'Paris', 'Nancy', 300)
arc (graphe, 'Nancy', 'Strasbourg', 130)
arc (graphe, 'Strasbourg', 'Lyon', 200)
arc (graphe, 'Nancy', 'Lyon', 250)
arc (graphe, 'Lyon', 'Marseille', 200)
arc (graphe, 'Marseille', 'Nice', 150)
```

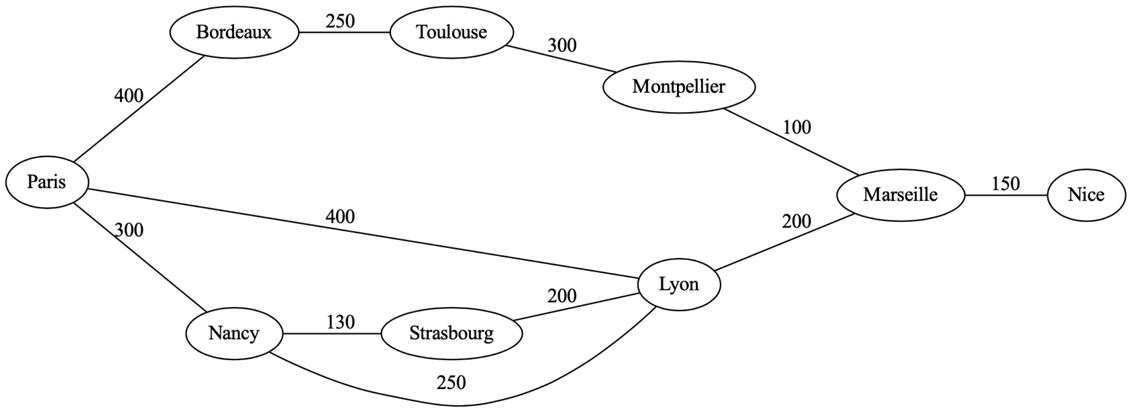



```
def new_matrix (m, n, v) :
    a = [ [v for j in range (n) ]
          for i in range (m) ]
    return a

def index_of (a, v) :
     for i in range (len(a)) :
        if a[i] == v :
           return i
     return None
```

Graphes (représentation 2)

- représentation par tableau de sommets et listes d'adjacence

```
class Sommet :
    def __init__ (self, s, l) :
        self.nom = s
        self.voisins = l
    #
    def __str__ (self) :
        return self.nom + ' [' + \
          ', '.join (map(str, self.voisins)) + ']'

def new_graph (elts) :
    n = len (elts)
    return [Sommet (elts[i], []) for i in range (n)]
```

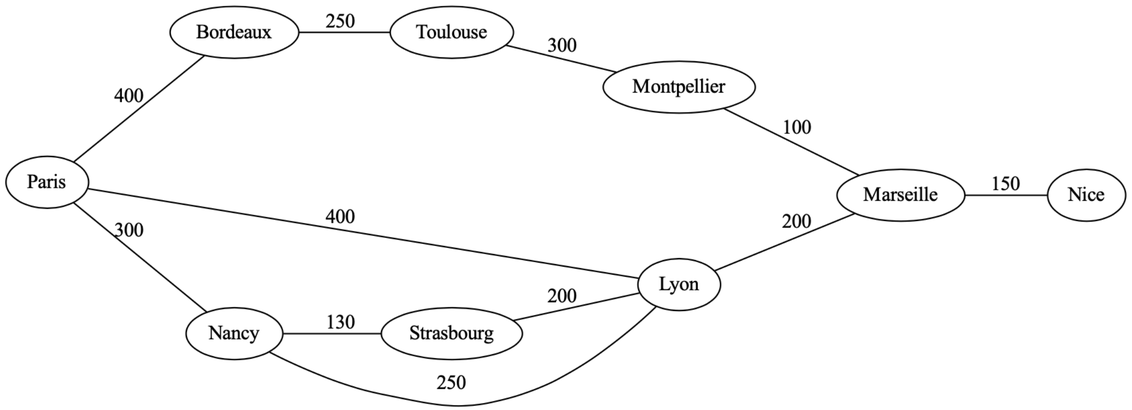



```
0: Paris [(5, 300), (7, 400), (1, 400)]
1: Bordeaux [(2, 250), (0, 400)]
2: Toulouse [(3, 300), (1, 250)]
3: Montpellier [(4, 100), (2, 300)]
4: Marseille [(8, 150), (7, 200), (3, 100)]
5: Nancy [(7, 250), (6, 130), (0, 300)]
6: Strasbourg [(7, 200), (5, 130)]
7: Lyon [(4, 200), (5, 250), (6, 200), (0, 400)]
8: Nice [(4, 150)]
```

Graphes (représentation 2)

- représentation par tableau de sommets et listes d'adjacence

```
class Sommet :
    def __init__ (self, s, l) :
        self.nom = s
        self.voisins = l
    #
    def __str__ (self) :
        return self.nom + ' [' + \
          ', '.join (map(str, self.voisins)) + ']'

def new_graph (elts) :
    n = len (elts)
    return [Sommet (elts[i], []) for i in range (n)]
```




```
graphe = new_graph (villes)
ajouter_arc (graphe, 'Paris', 'Bordeaux', 400)
ajouter_arc (graphe, 'Bordeaux', 'Toulouse', 250)
ajouter_arc (graphe, 'Toulouse', 'Montpellier', 300)
ajouter_arc (graphe, 'Montpellier', 'Marseille', 100)
ajouter_arc (graphe, 'Paris', 'Lyon', 400)
ajouter_arc (graphe, 'Paris', 'Nancy', 300)
ajouter_arc (graphe, 'Nancy', 'Strasbourg', 130)
ajouter_arc (graphe, 'Strasbourg', 'Lyon', 200)
ajouter_arc (graphe, 'Nancy', 'Lyon', 250)
ajouter_arc (graphe, 'Lyon', 'Marseille', 200)
ajouter_arc (graphe, 'Marseille', 'Nice', 150)
```

```
def ajouter_arc (g, v1, v2, d) :
    i = villes.index(v1)
    j = villes.index(v2)
    g[i].voisins = [(j, d)] + g[i].voisins
    g[j].voisins = [(i, d)] + g[j].voisins
```

Graphes

Exercice Calculer une distance possible pour aller d'une ville à une autre

à faire

- analyses lexicale et syntaxique
- modularité et programmation objet
- programmation graphique
- algorithmes géométriques
- calculs flottants et méthodes numériques
- programmation de plusieurs fils de calcul
- assertions et logique des programmes
- introduction à l'informatique théorique
- etc

vive l’informatique

et

la programmation !