Informatique et Programmation

Cours 15

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-py

Plan

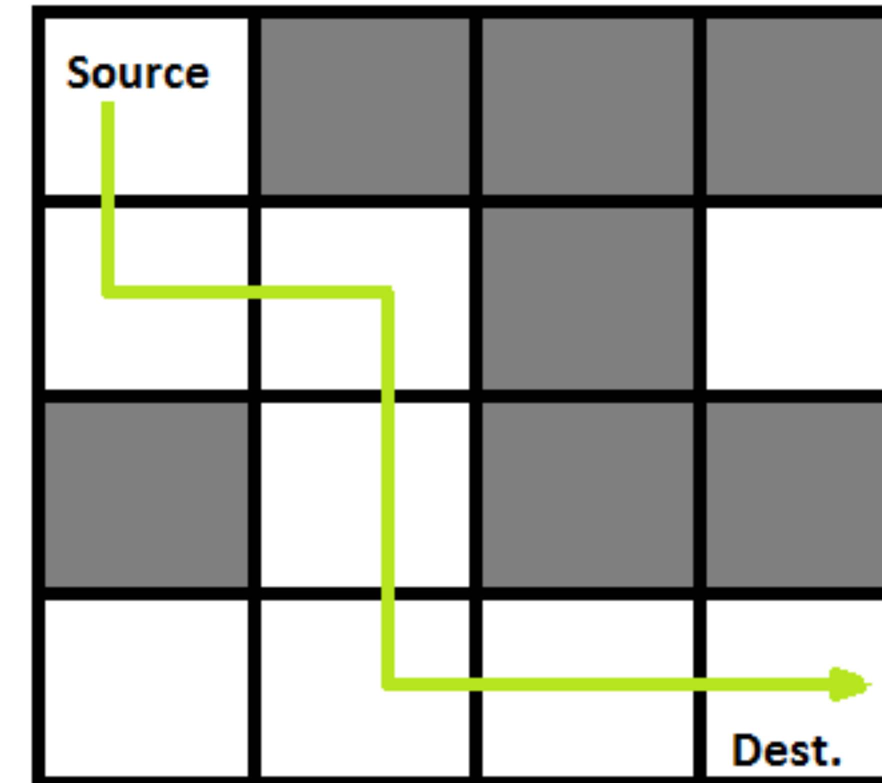
- graphes dirigés
- arbres de recouvrement
- parcours de graphes
- composantes connexes

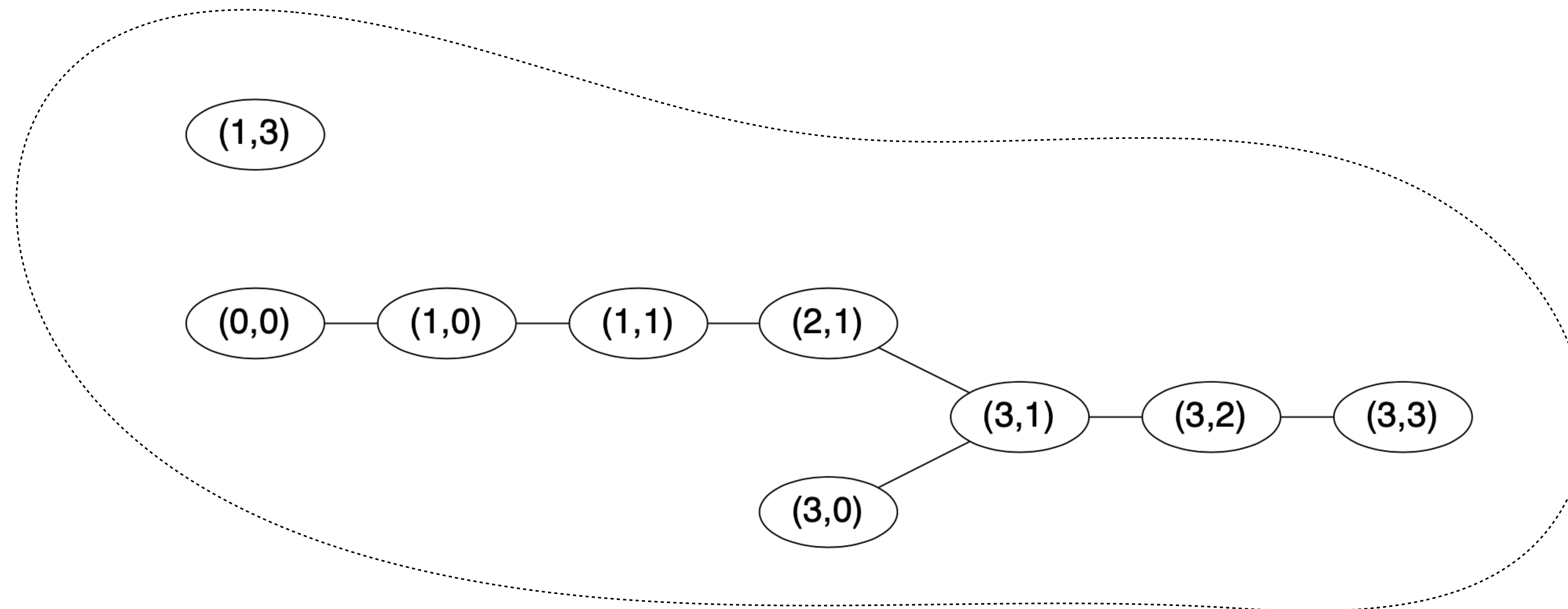
dès maintenant: télécharger Python 3 en `http://www.python.org`

Graphes

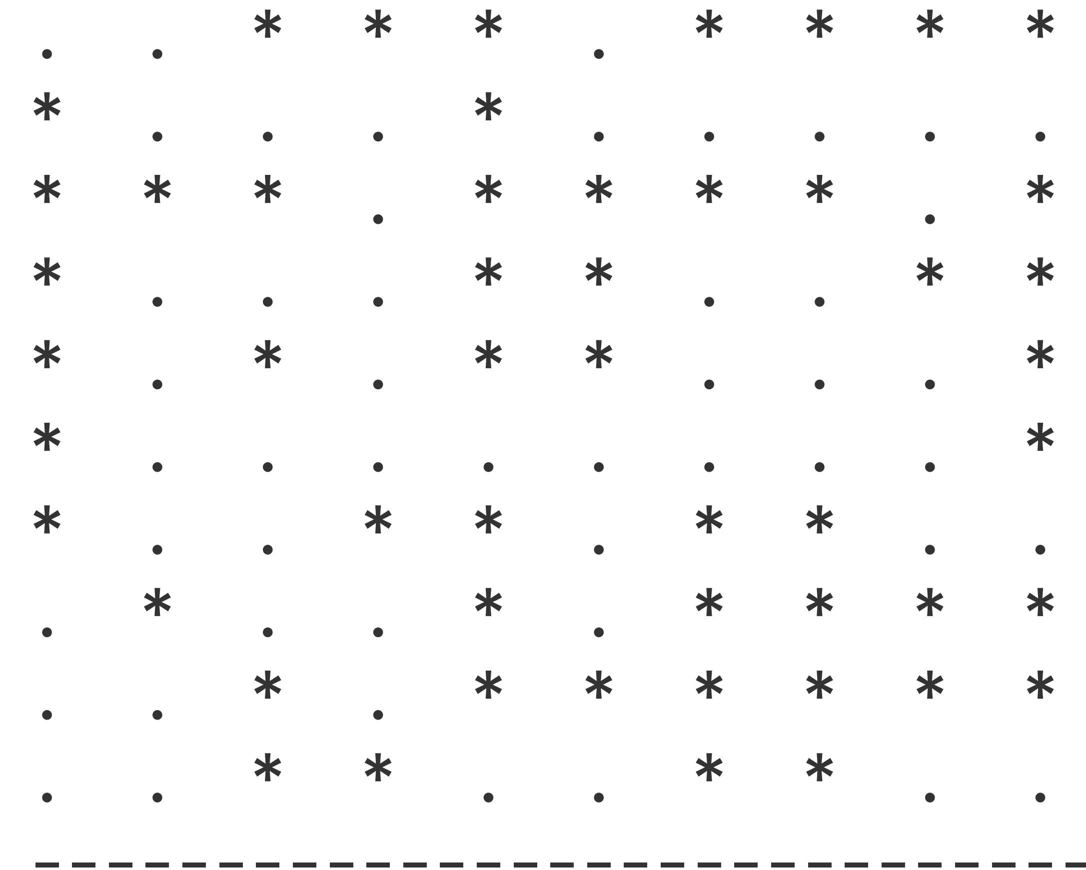
- Graphes de connexion des cases d'un labyrinthe

```
maze = [[0, 1, 1, 1],
        [0, 0, 1, 0],
        [1, 0, 1, 1],
        [0, 0, 0, 0]]
```





Graphes




```
m1 = [[0, 0, 1, 1, 1, 0, 1, 1, 1, 1],
      [1, 0, 0, 0, 1, 0, 0, 0, 0, 0],
      [1, 1, 1, 0, 1, 1, 1, 1, 0, 1],
      [1, 0, 0, 0, 1, 1, 0, 0, 1, 1],
      [1, 0, 1, 0, 1, 1, 0, 0, 0, 1],
      [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
      [1, 0, 0, 1, 1, 0, 1, 1, 0, 0],
      [0, 1, 0, 0, 1, 0, 1, 1, 1, 1],
      [0, 0, 1, 0, 1, 1, 1, 1, 1, 1],
      [0, 0, 1, 1, 0, 0, 1, 1, 0, 0]]
```

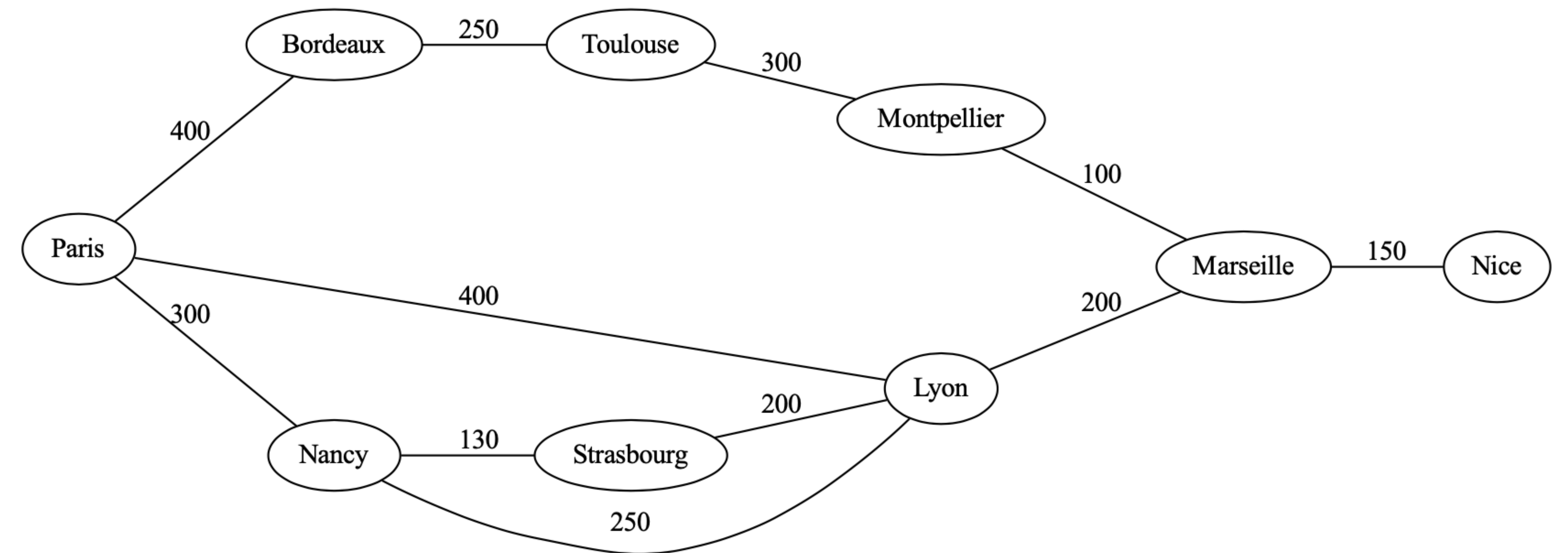
- Graphes de connexion des cases d’un labyrinthe

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


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



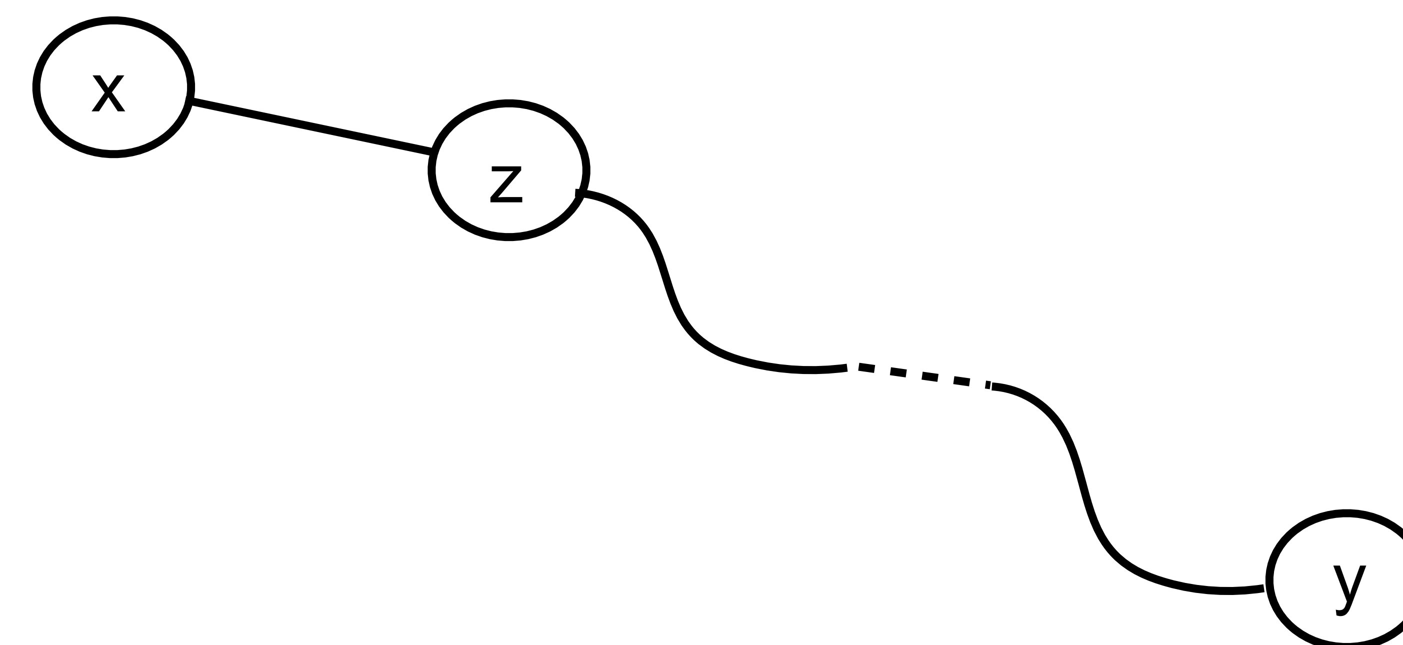
Graphes

- cakculer un chemin possible pour aller d’une ville à une autre

```
def chemin (g, x, y, dejaVu) :
    dejaVu [x] = True
    if x == y :
        return [x]
    for p in g[x].voisins :
        z = p[0] #d = p[1]
        if not dejaVu [z] :
            ch = chemin (g, z, y, dejaVu)
            if ch != [] :
                return [z] + ch
    return []
```

```
def uneSolution (g, x, y) :
    n = len (g)
    dejaVu = n*[False]
    ch = chemin (g, x, y, dejaVu)
    if ch != [] :
        return ([x] + ch)[:-1]
    return []
```

- même programme que pour sortie de labyrinthe (cours 10)

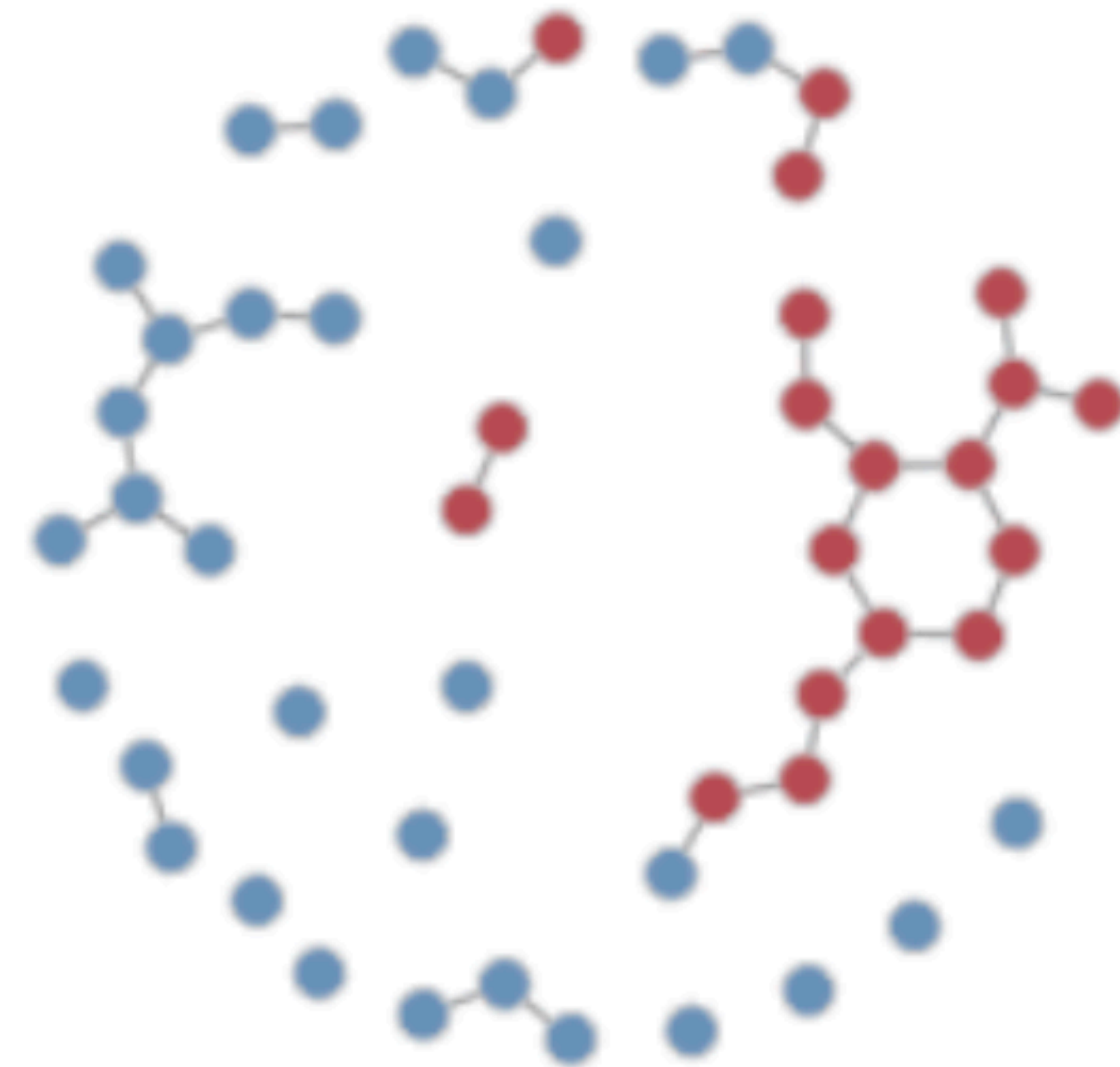


Composantes connexes

- on oublie les distances et on ne considère que les connexions

- le graphe précédent devient:

```
g = [
    Sommet ('a', [5, 7, 1]),
    Sommet ('b', [2, 0]),
    Sommet ('c', [3, 1]),
    Sommet ('d', [4, 2]),
    Sommet ('e', [8, 7, 3]),
    Sommet ('f', [7, 6, 0]),
    Sommet ('g', [7, 5]),
    Sommet ('h', [4, 5, 6, 0]),
    Sommet ('i', [4])
    ]
```

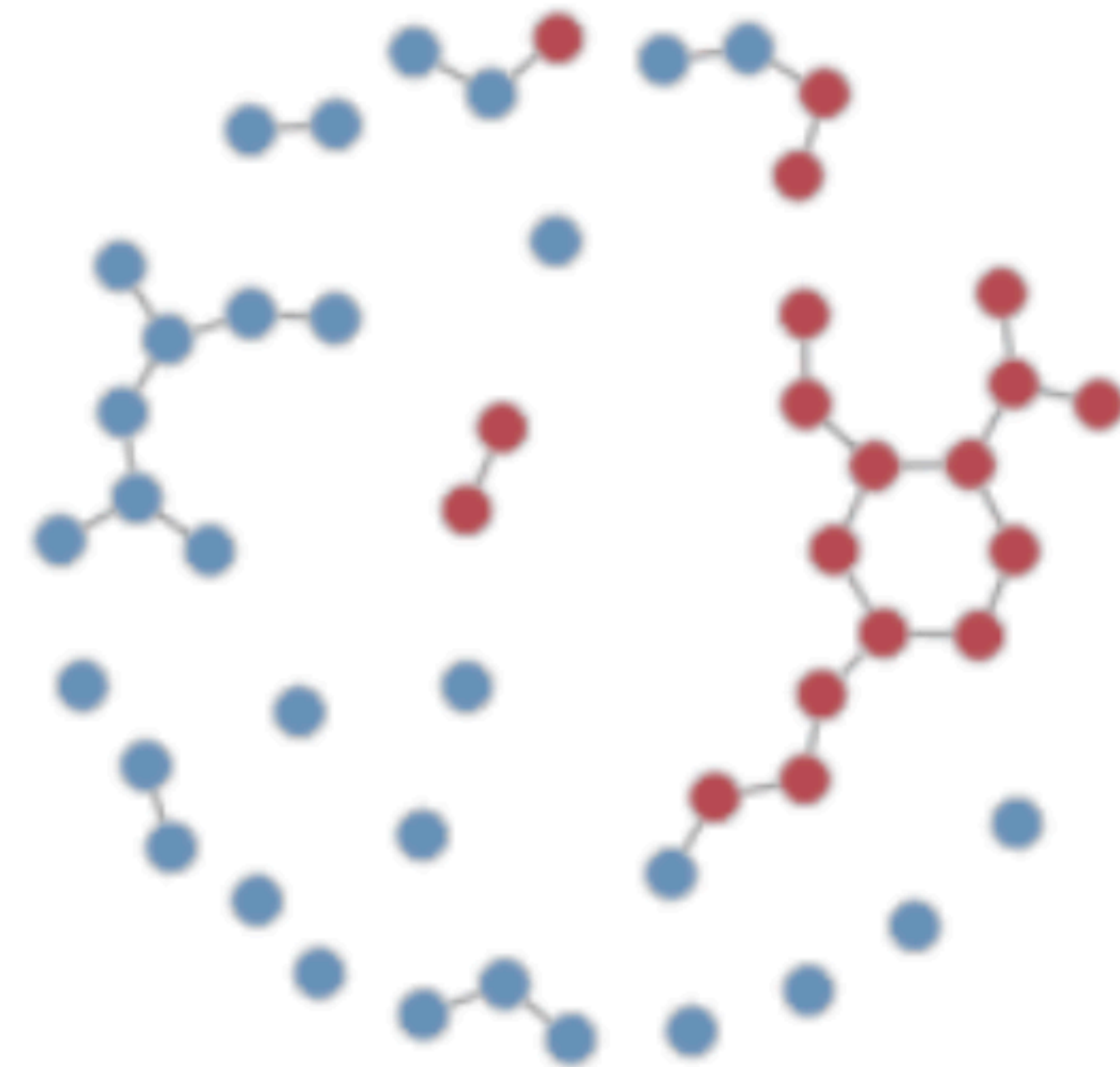


Composantes connexes

- une composante connexe est un ensemble maximal de sommets tous reliés entre eux.

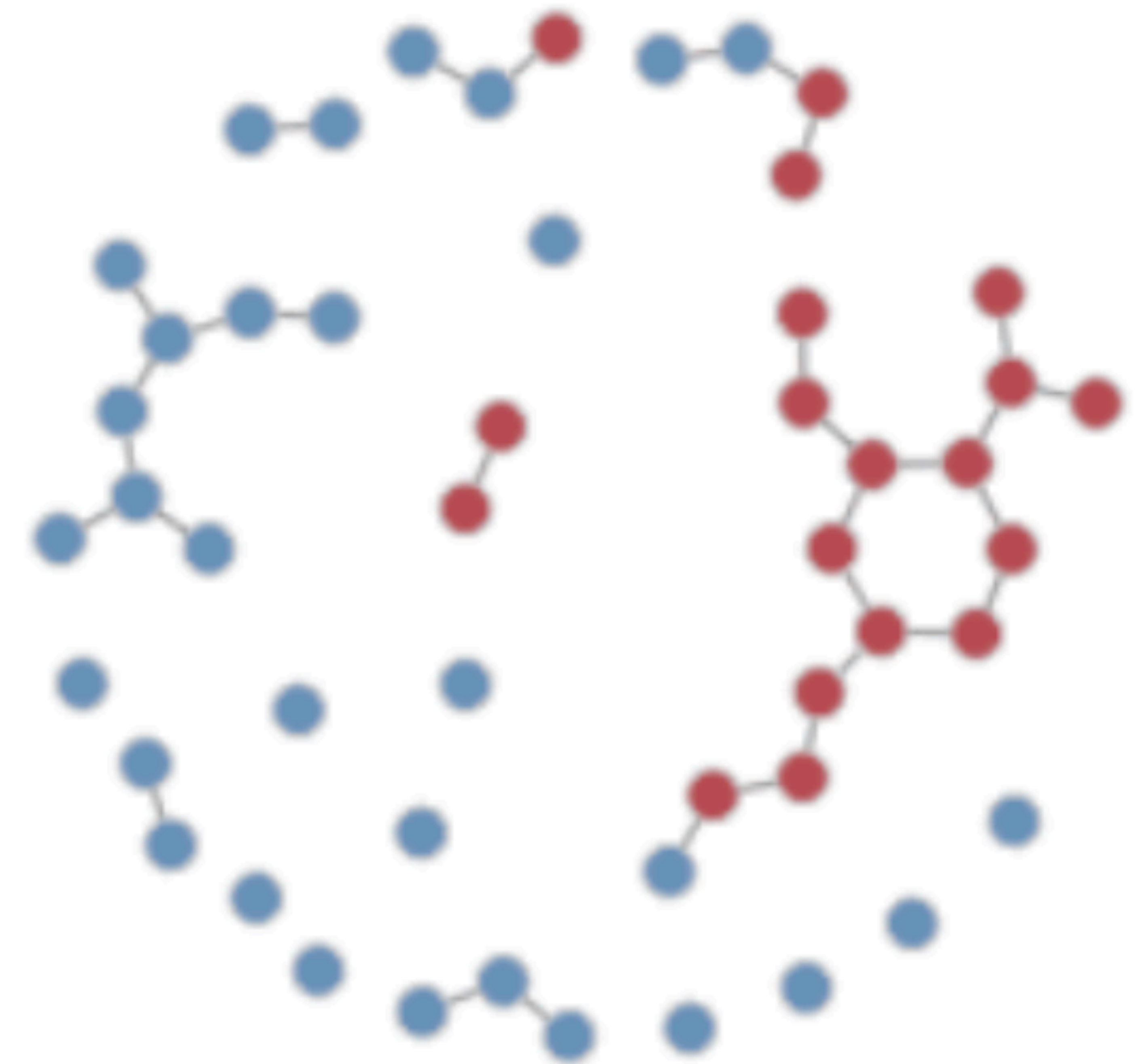
- un graphe peut contenir plusieurs composantes connexes
  - villes d’un même continent
  - sommets d’un labyrinthe
  - groupes d’amis dans un réseau social

- comment calculer les composantes connexes ?



Composantes connexes

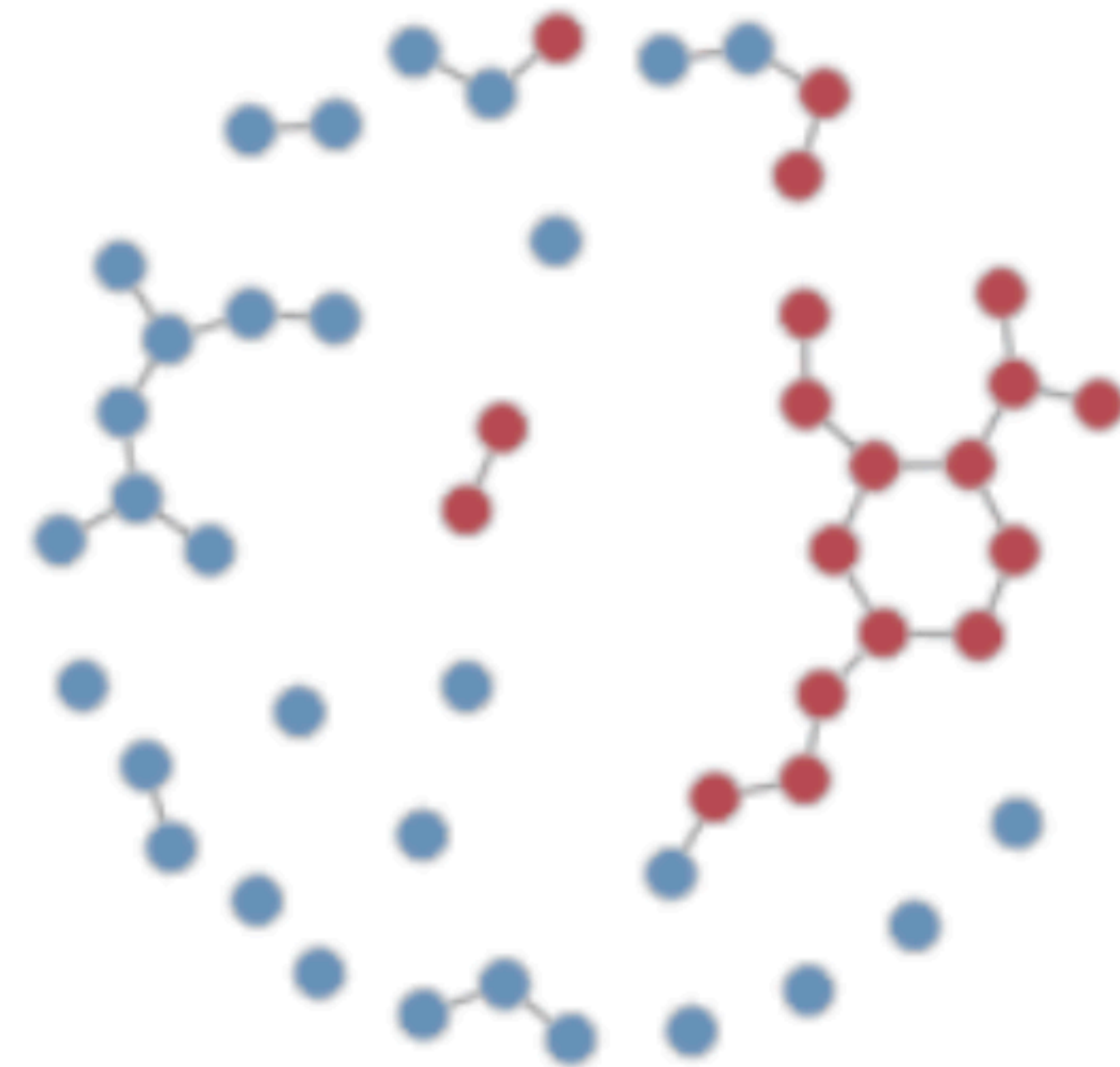
- au début, aucun sommet n'est visité et on itère l'algorithm suivant:
  - initialiser le tableau `dejaVu` à `False`
- à partir d'un sommet non visité,
  - on explore tous les sommets accesibles par un chemin
- et on recommence



Composantes connexes

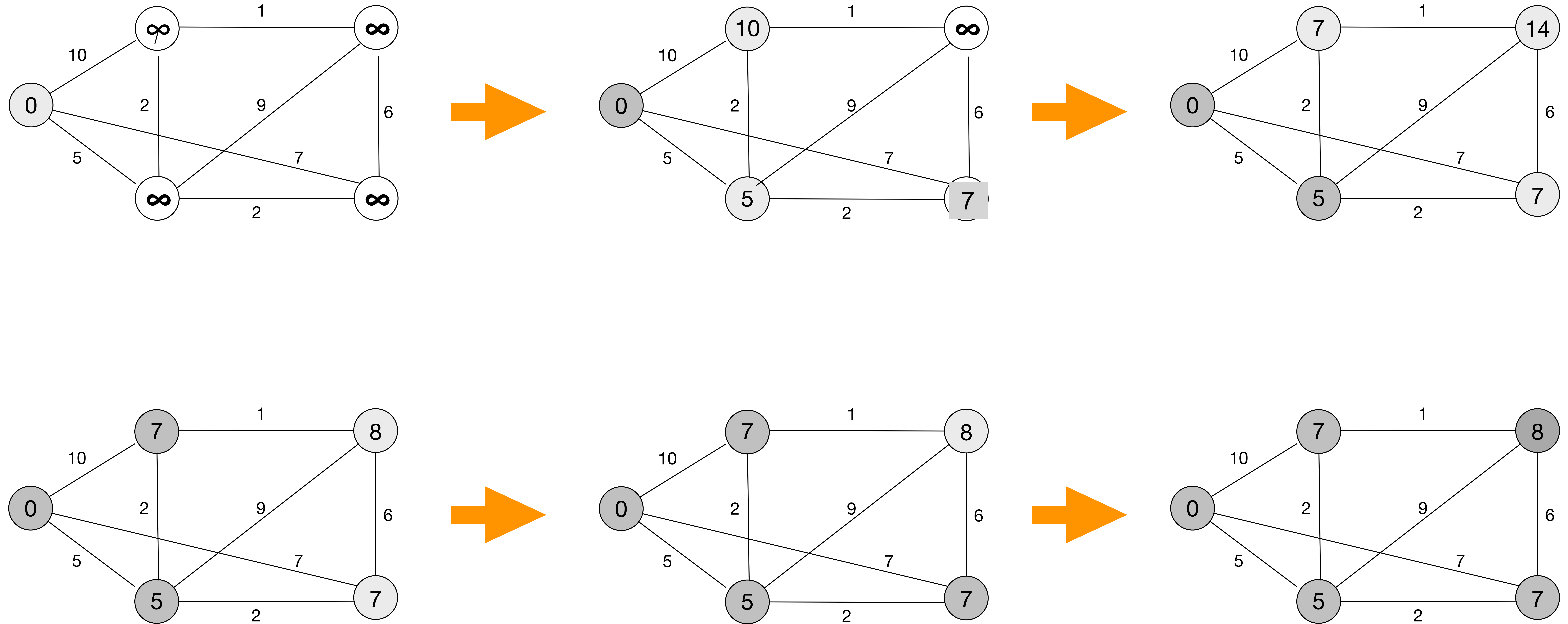
```
def cconn_de (g, x, dejaVu) :
    dejaVu [x] = True
    r = [x]
    for y in g[x].voisins :
        if not dejaVu [y] :
            r = r + cconn_de (g, y, dejaVu)
    return r
```

```
def composantes_de (g) :
    n = len (g)
    dejaVu = n*[False]
    r = []
    for x in range(n) :
        if not dejaVu[x] :
            r = r + [cconn_de (g, x, dejaVu)]
    return r
```



Graphes

- cakculer le chemin le plus court pour aller d’une ville à toutes les autres `[Dijkstra]`



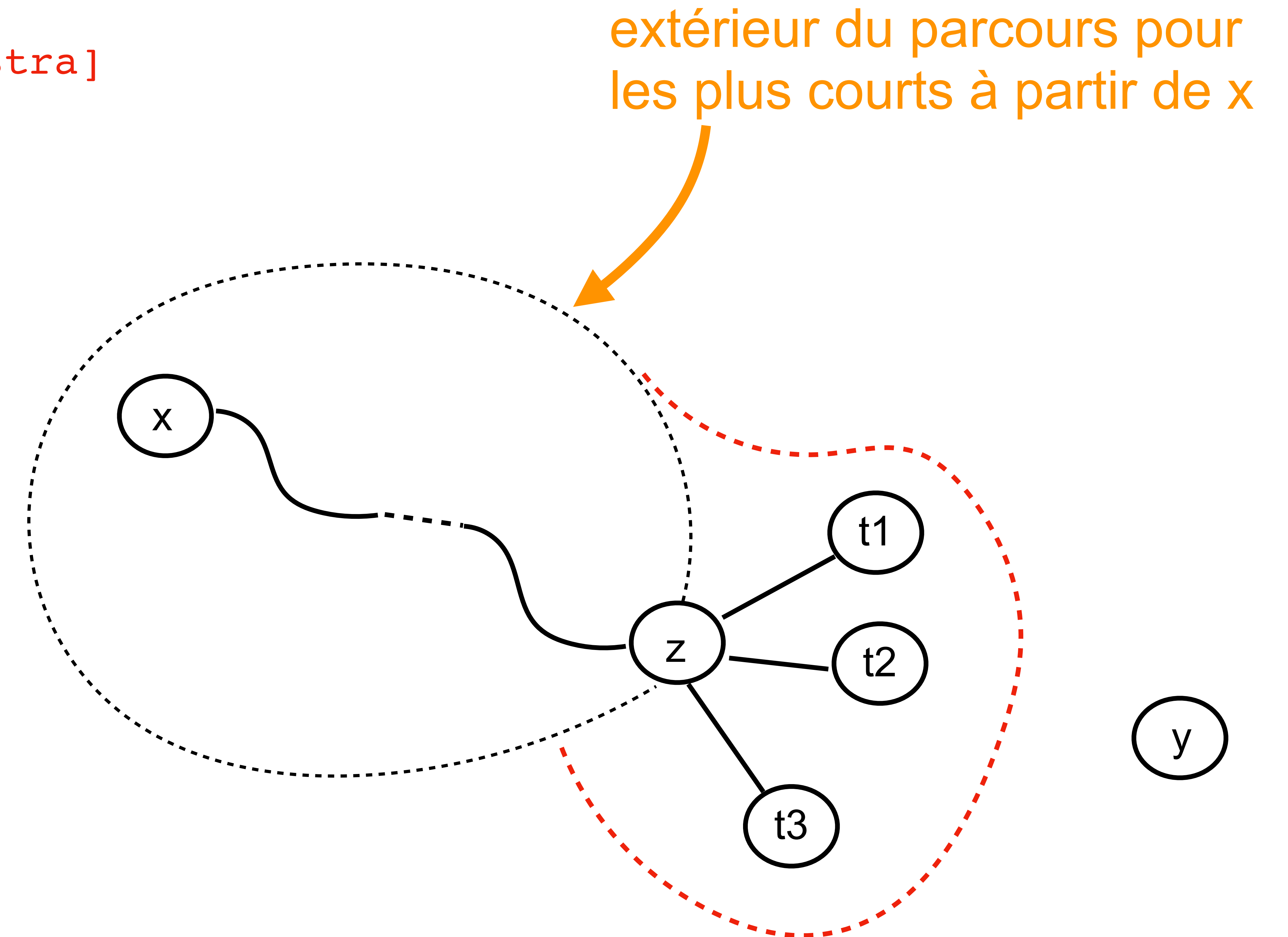
Graphes

- cakculer le chemin le plus court pour aller d'une ville à une autre [Dijkstra]

```
def distance_min (g, x, y) :
    n = len (g)

    ext = n*[False]; ext[x] = True
    xdist = n*[infini]; xdist[x] = 0
    z = x
    while z != y :
        ext[z] = False
        for p in g[z].voisins :
            t = p[0]; d = p[1]
            if (xdist[z] + d < xdist[t]) :
               xdist[t] = xdist[z] + d
               ext[t] = True

        z = imin (ext, xdist)
    return (xdist[y])
```




```
def imin (f, xdist) :
    # assert : len (f) == len (xdist)
    n = len (f)
    dmin = infini
    for i in range (n) :
        if f[i] and xdist[i] < dmin :
            r = i; dmin = xdist[i]
    return r
```

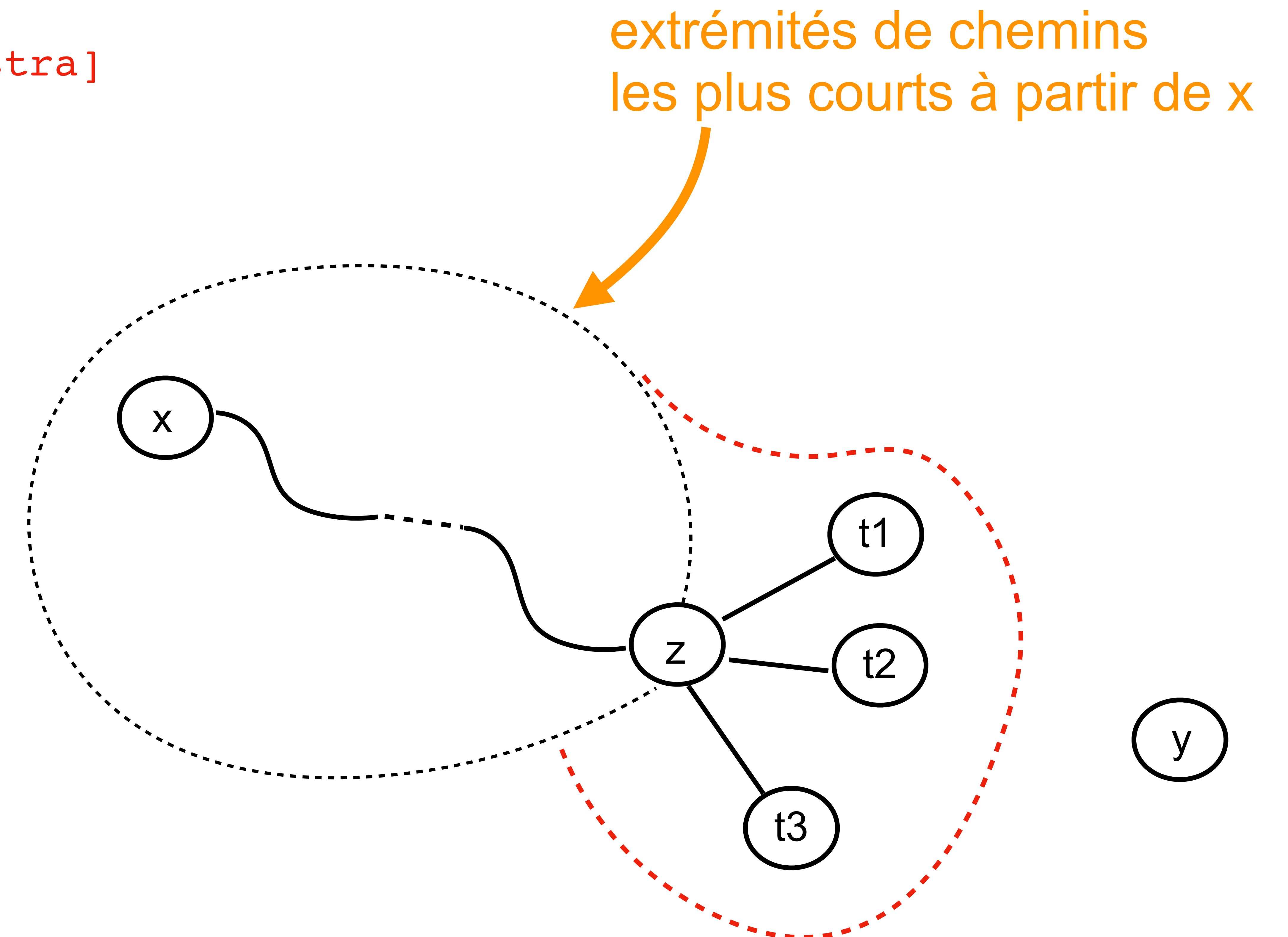
- de l'ordre de n^2 opérations

- algorithme glouton (optimisation locale)

Graphes

- cakculer le chemin le plus court pour aller d’une ville à une autre `[Dijkstra]`

```
def distance_min (g, x, y) :
    n = len (g)
    pred = n*[0]; pred[x] = -1
    ext = n*[False]; ext[x] = True
    xdist = n*[infini]; xdist[x] = 0
    z = x
    while z != y :
        ext[z] = False
        for p in g[z].voisins :
            t = p[0]; d = p[1]
            if (xdist[z] + d < xdist[t]) :
               xdist[t] = xdist[z] + d
               ext[t] = True
               pred[t] = z
        z = imin (ext, xdist)
    return (xdist[y], pred)
```




```
def imin (f, xdist) :
    # assert : len (f) == len (xdist)
    n = len (f)
    dmin = infini
    for i in range (n) :
        if f[i] and xdist[i] < dmin :
            r = i; dmin = xdist[i]
    return r
```

- de l’ordre de n^2 opérations

- algorithme glouton (optimisation locale)

Graphes

- cakculer le chemin le plus court pour aller d'une ville à une autre `[Dijkstra]`

```
def chemin_min (g, x, y) :
    dch = distance_min (g, x, y)
    dist = dch[0]; pred = dch[1]
    r = []
    z = y
    while z != -1 :
        r = [g[z].nom] + r
        z = pred[z]
    r = [dist] + r
    return r


print (chemin_min (graphe,
                   villes.index('Paris'),
                   villes.index ('Strasbourg')))
```

Graphes

- cakculer le chemin le plus court pour aller d'une ville à une autre `[Dijkstra]`

```
def distance_min (g, x, y) :
    n = len (g)
    pred = n*[0]; pred[x] = -1
    f = new_file(); f = add_file (f, x)
    xdist = n*[infini]; xdist[x] = 0
    z = x
    while z != y :
        print (f)
        f = del_file (f, z)
        for p in g[z].voisins :
            t = p[0]; d = p[1]
            if (xdist[z] + d < xdist[t]) :
               xdist[t] = xdist[z] + d
               f = add_file (f, t)
               pred[t] = z
        z = imin (f, xdist)
    return (xdist[y], pred)
```

même programme avec une file

(de priorité)

- de l'ordre de $n \log n$ opérations

```
def imin (f, xdist) :
    # assert : len (file) == len (xdist)
    dmin = infini
    for x in f :
        if xdist[x] < dmin :
            r = x; dmin = xdist[x]
    return r
```

à faire

- graphes dirigés
- 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