Informatique et Programmation

Cours 8

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-py

Plan

- portée des variables en Python
- graphes
- implémentation des graphes
- recherche de chemins dans un graphe
  - exploration en profondeur d'abord
  - exploration en largeur d'abord
  - plus court chemin

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

Recap

- mots clés en Python (déjà vus en rouge)

```
>>> help()
help> keywords

Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not
```

Recap

- déclaration des variables en Python
  - pas de déclaration !
  - déclaration implicite dans la fonction englobante
- mot clé **global**
  - variable déclarée dans le module courant
  - dans une fonction, on peut qualifier une variable globale
- mot clé **nonlocal**
  - variable locale d'une fonction contenant la fonction courante

```
x = 4

def f() :
    print (x)

def g() :
    x = 3; print (x)

f(); g(); f()
```

→ 4 3 4

```
x = 4

def f() :
    print (x)

def g() :
    global x
    x = 3; print (x)

f(); g(); f()
```

→ 4 3 3

```
def f() :
  def g() :
    x = 3; print (x)
  x = 4
  g()
  print (x)

f()
```

→ 3 4

```
def f() :
  def g() :
    nonlocal x
    x = 3; print (x)
  x = 4
  g()
  print (x)

f()
```

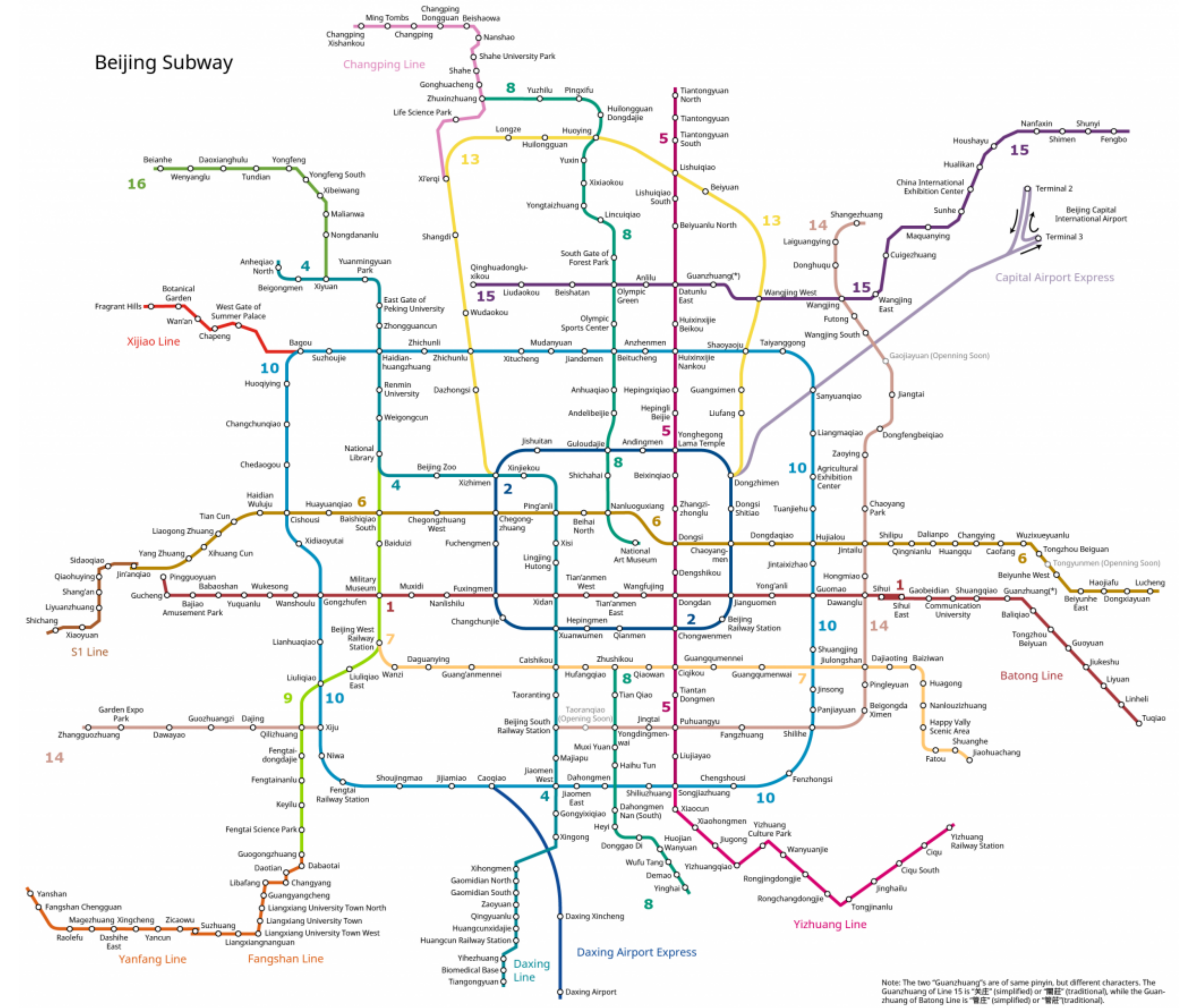
→ 3 3

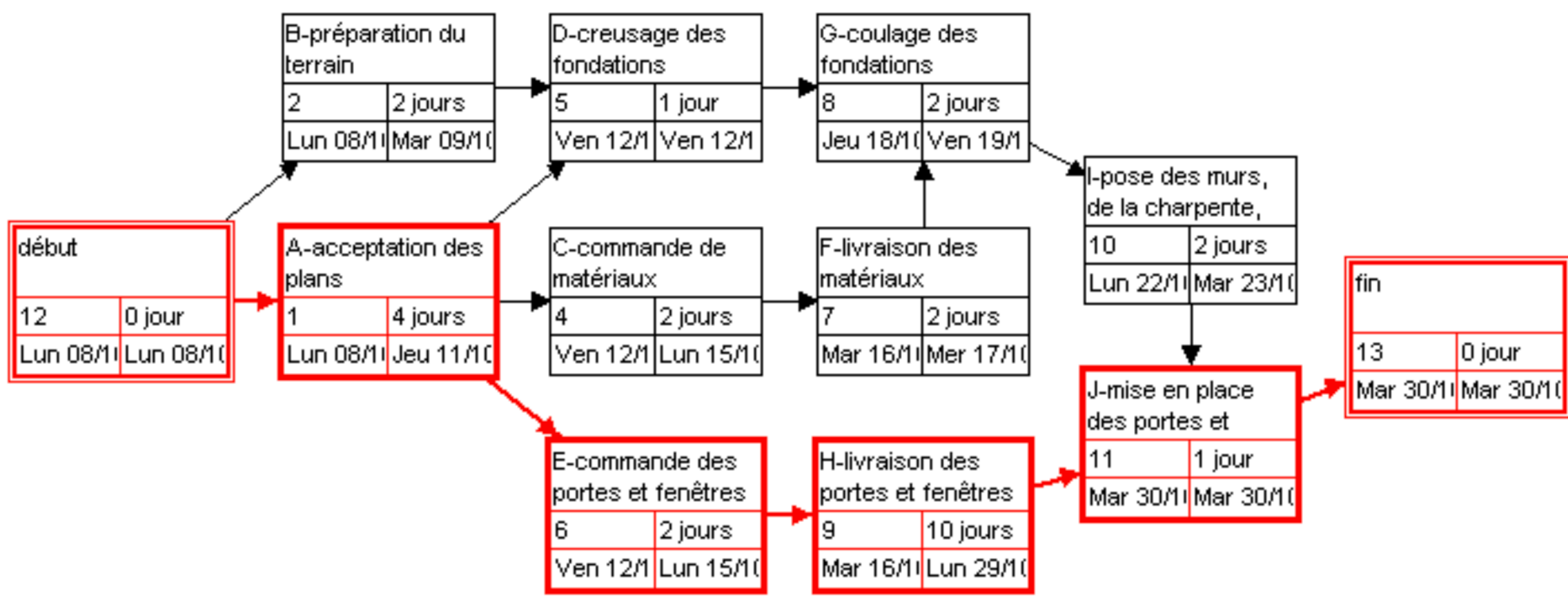
Recap

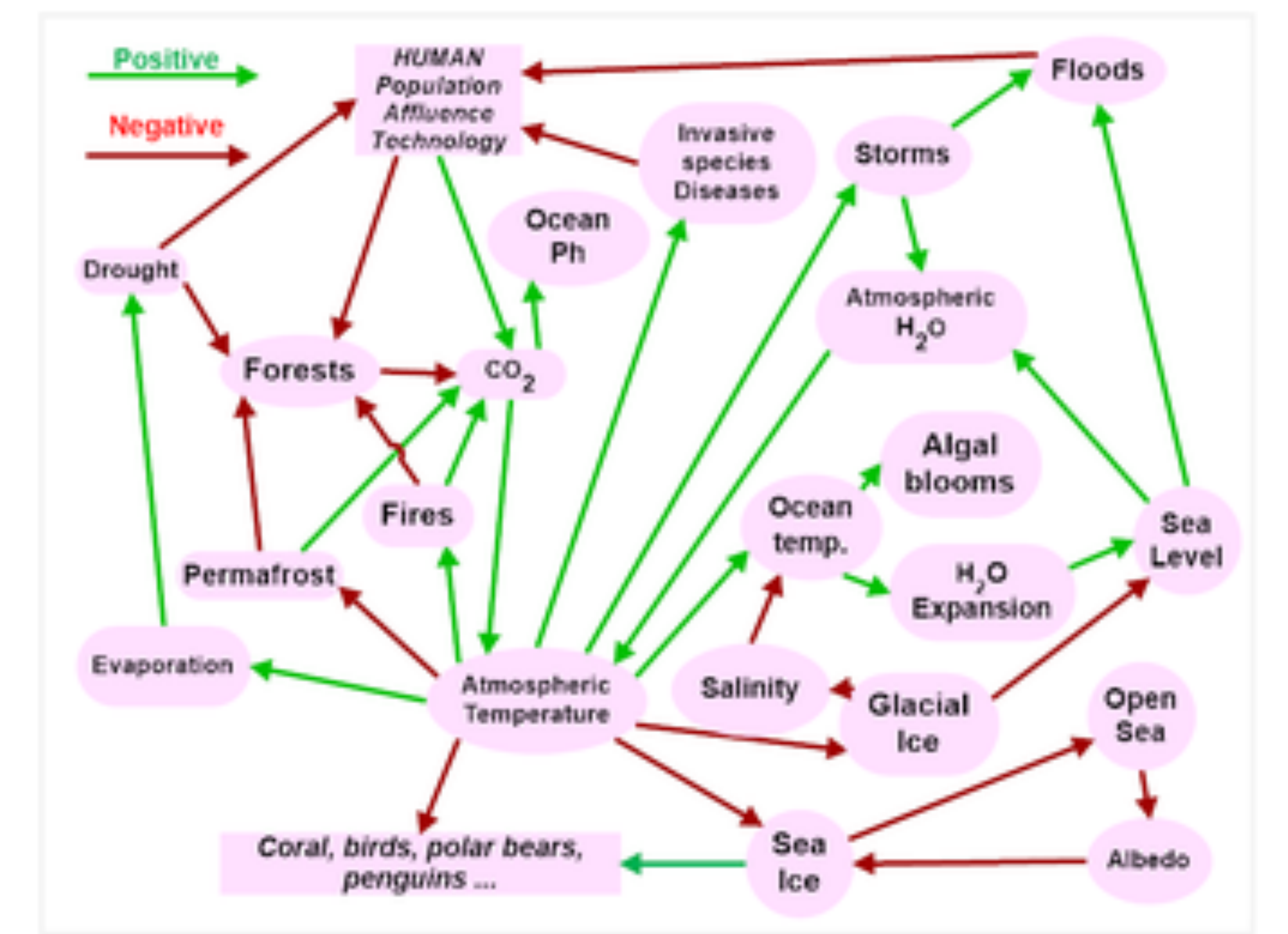
- mots clés en Python (déjà vus en rouge)

```
>>> help()
help> keywords

Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not
```

Exemples de graphes

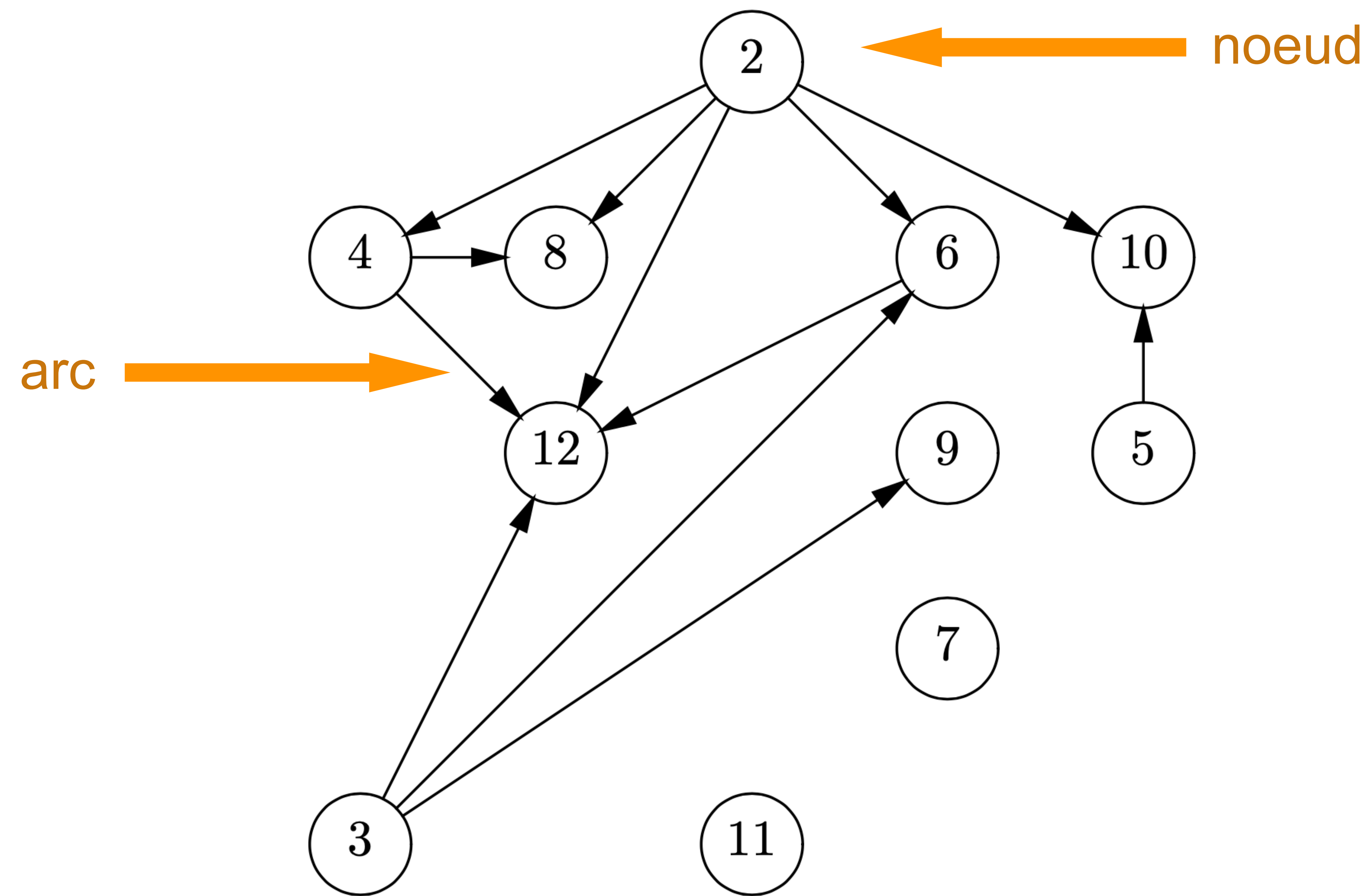




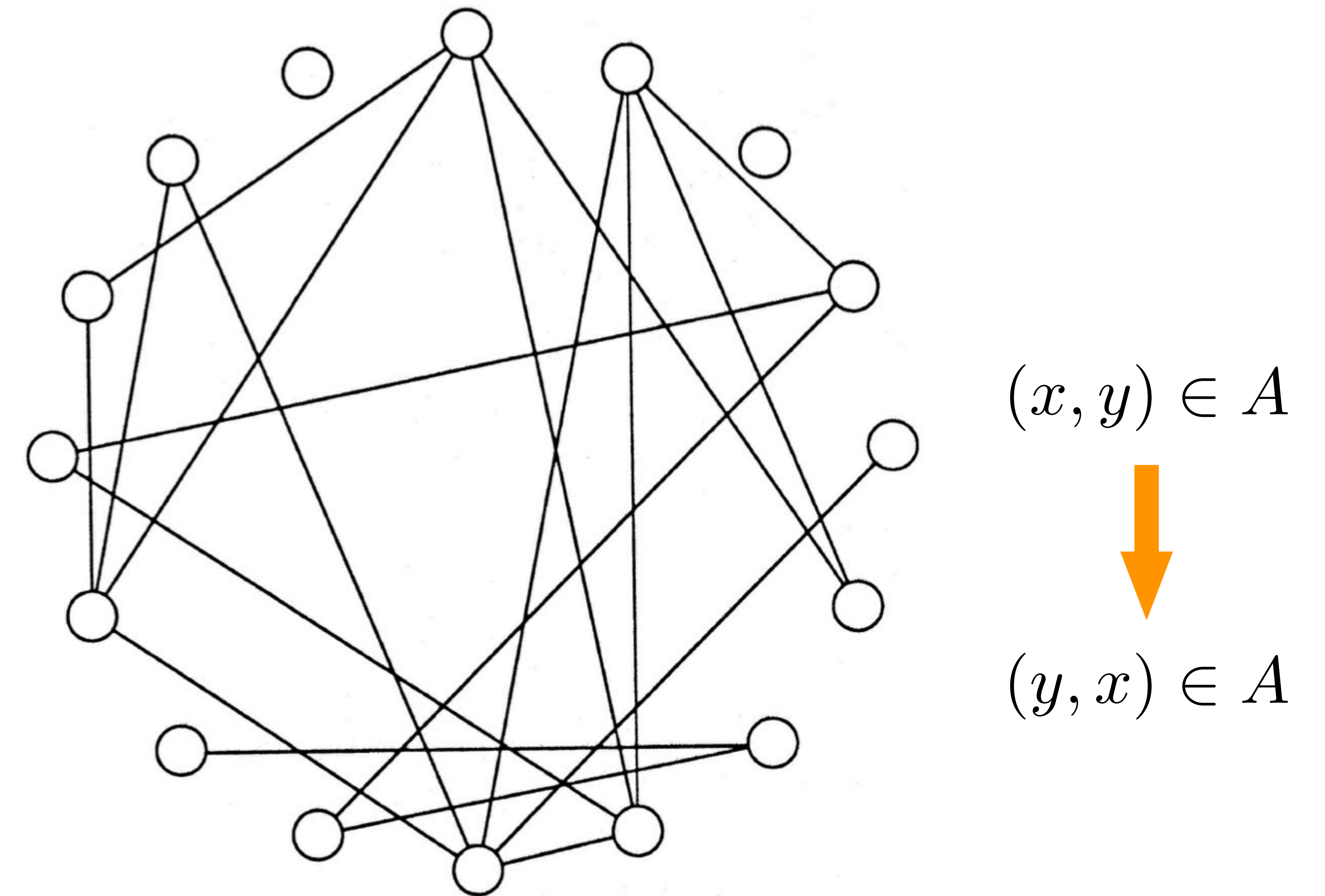


Exemples de graphes

- un graphe est donné par une paire (*G, A*) où *G* est un ensemble de **noeuds** et *A* est un ensemble d'**arcs**
- les arcs relient 2 noeuds (source et destination)



graphe orienté (*directed graph*)



$(x, y) \in A$

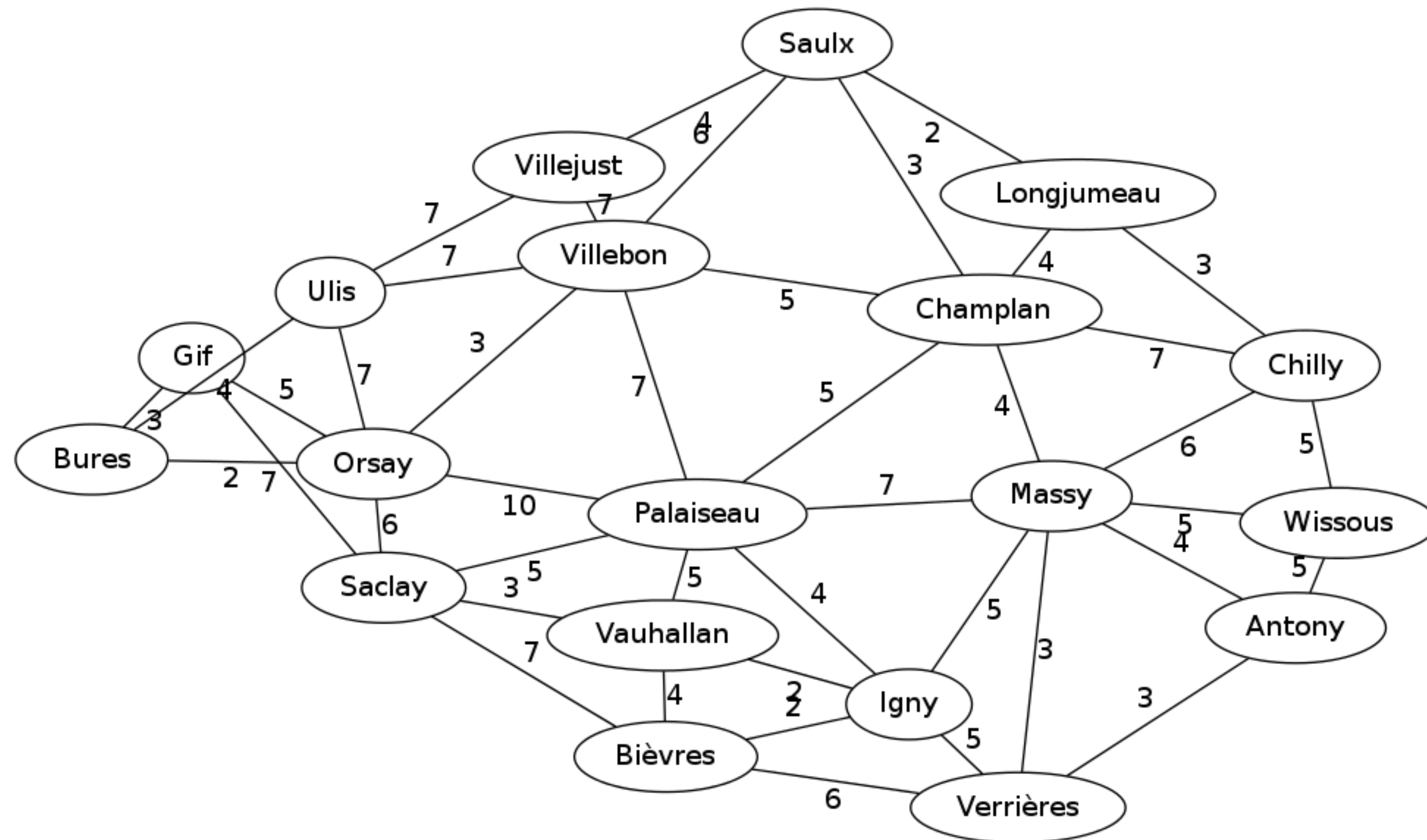
$(y, x) \in A$

graphe non orienté (*undirected graph*)

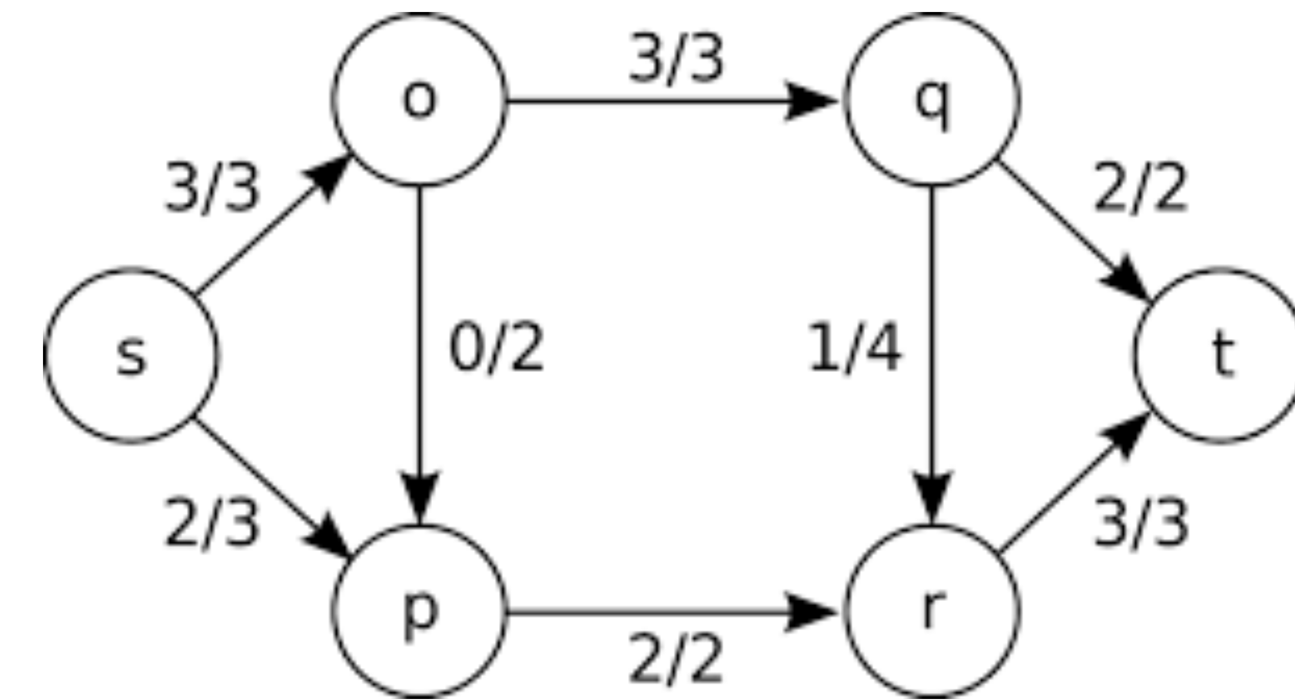
[**pour les mathématiciens, un graphe est une relation binaire entre les noeuds**]

Exemples de graphes

- les arcs peuvent aussi contenir des valeurs. Alors *A* est un ensemble d'arcs (x, y, v) où x est le noeud source, y est le noeud destination, v est une valeur associée à cet arc



nombre de kms entre 2 noeuds

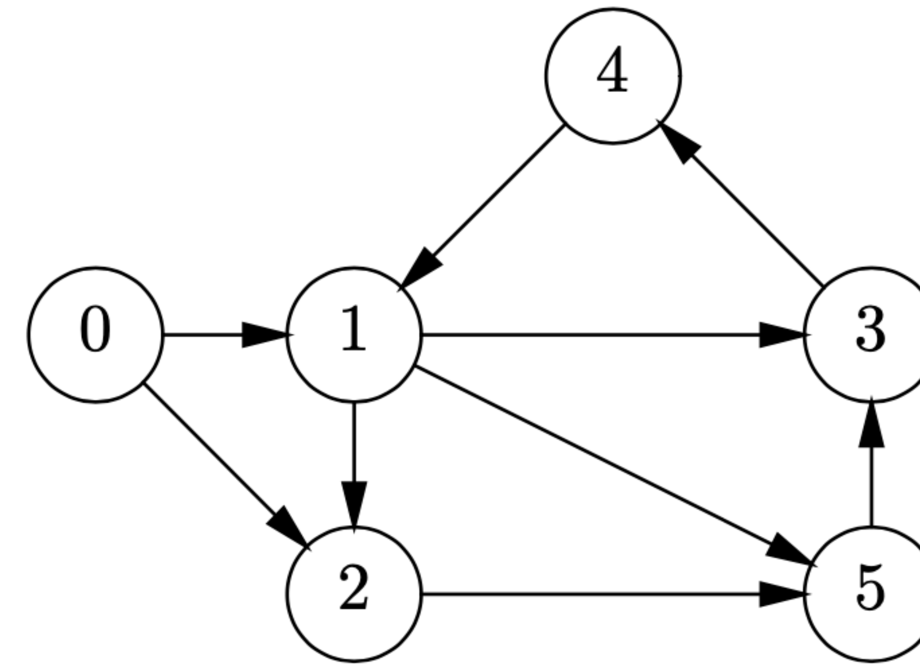


graphe de flots entre *s* et *t* où *f* / *c* représente le flot courant sur l'arc et sa capacité maximum

Représentation des graphes

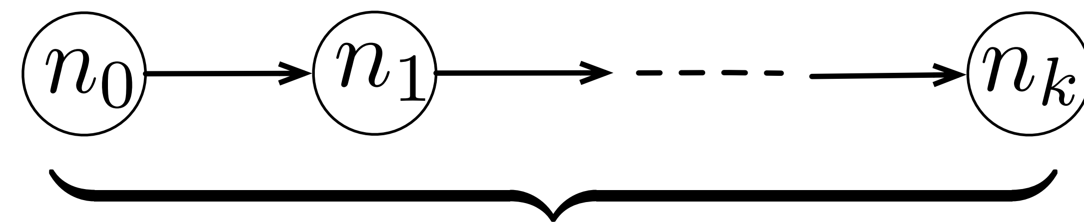
- matrice d'adjacence (matrice booléenne)

$M_{i,j} = 1$ si et seulement s'il existe un arc de i à j



$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

- un chemin dans un graphe est une suite d'arcs adjacents



chemin de longueur k
$(k \geq 0)$

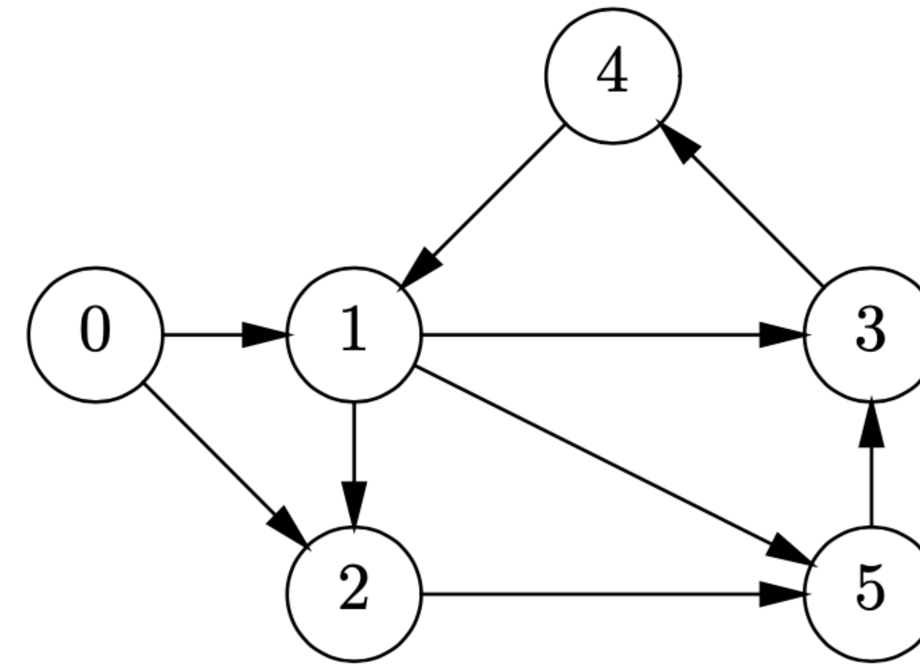
I	chemins de longueur 0
M	chemins de longueur 1
$M \times M$	chemins de longueur 2
$M \times M \times M$	chemins de longueur 3

Fermeture transitive

- existe-t-il un chemin entre les noeuds *x* et *y* ?

- on calcule une matrice M^*:

$M^*_{i,j} = 1$ si et seulement s'il existe un chemin de i à j



$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

- M^* est la fermeture transitive de M

$$M^* = I + M + M^2 + M^3 + \cdots + M^n$$

où n est le nombre de noeuds du graphe

- cela demande n^4 opérations !!

- l'algorithme de `Warshall` ramène ce calcul à n^3 opérations

- faire mieux ?

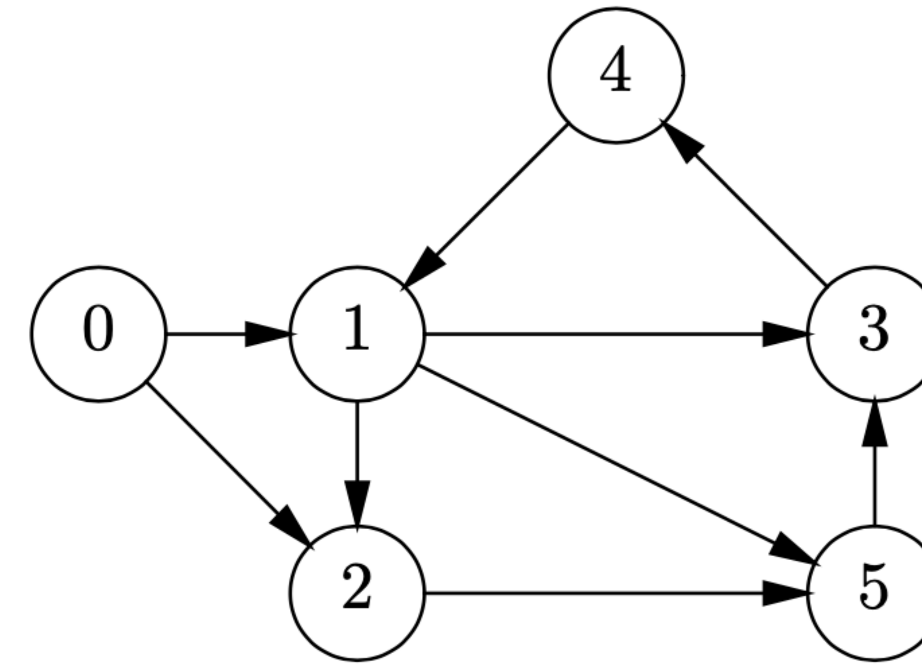
Exercice trouver l'algorithme de Warshall et écrire le programme.

Indice calculer les chemins ne passant que par des noeuds inférieurs où égal à *k* pour *k* variant de 0 à *n*

Représentation des graphes

- avec un classe noeud et des listes de successeurs

[représentation plus compacte si peu d'arcs]




```
class Noeud :
   def __init__ (self, x, a) :
      self.val = x
      self.succ = a
```

```
n0 = Noeud (0, [ ])
n1 = Noeud (1, [ ])
n2 = Noeud (2, [ ])
n3 = Noeud (3, [ ])
n4 = Noeud (4, [ ])
n5 = Noeud (5, [ ])
```

il faut créer les noeuds avant de les référencer !

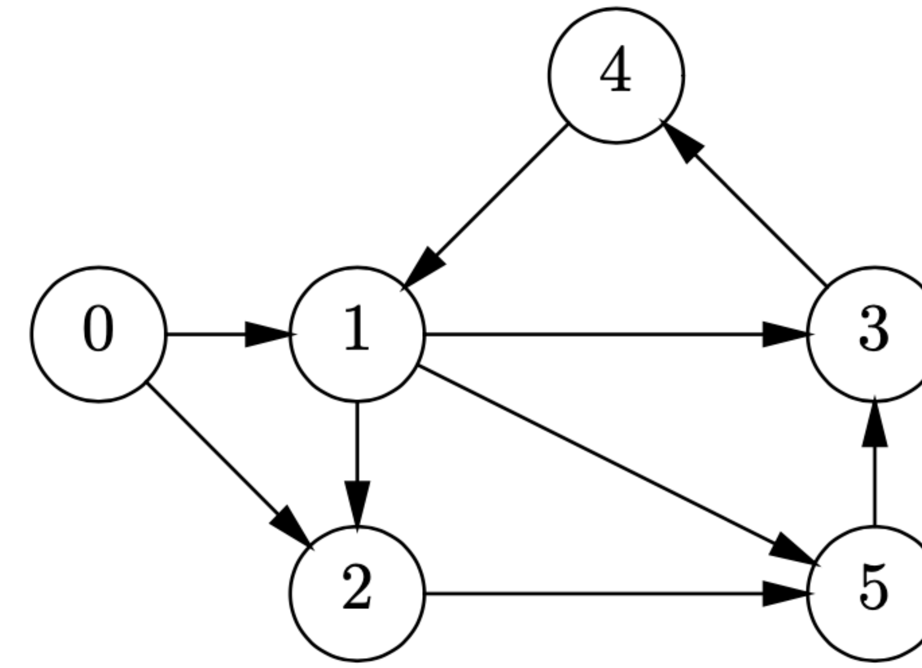
```
n0.succ = [n1, n2]
n1.succ = [n2, n5, n3]
n2.succ = [n5]
n3.succ = [n4]
n4.succ = [n1]
n5.succ = [n3]

G = [n0, n1, n2, n3, n4, n5]
```

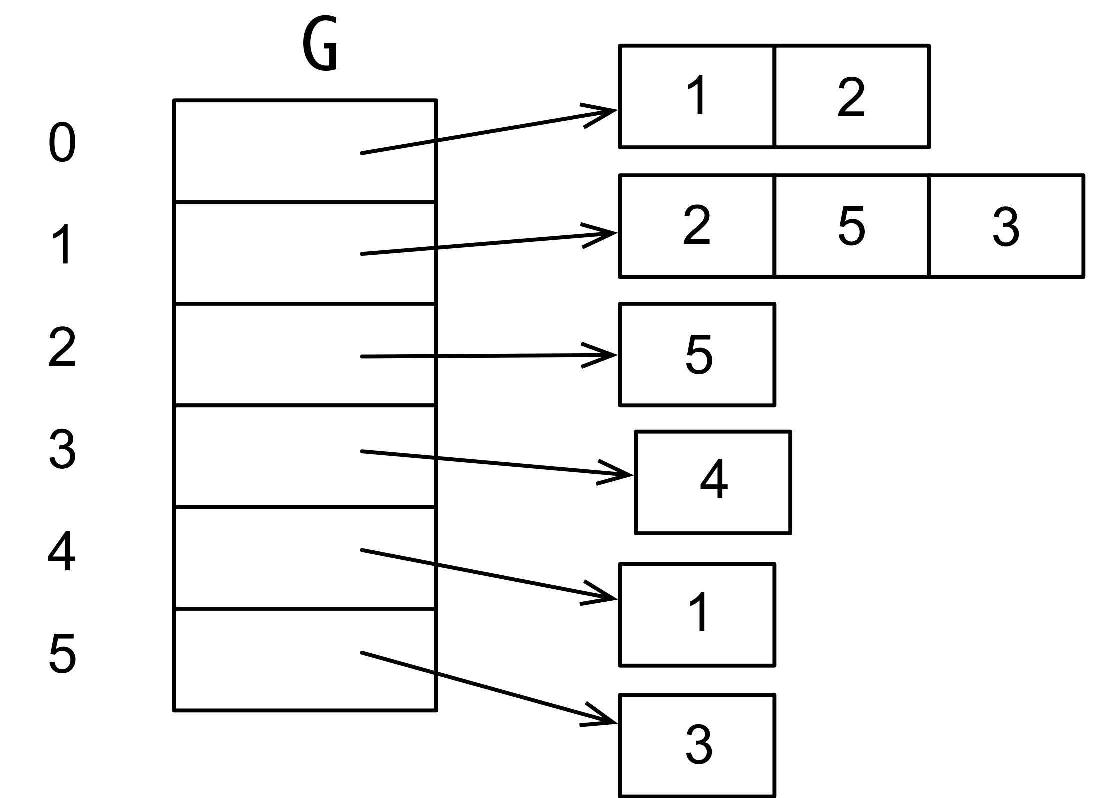
le graphe est donné par la liste de ses noeuds

Représentation des graphes

- ici on simplifiera en désignant les noeuds par des entiers et en ne considérant que les listes de successeurs

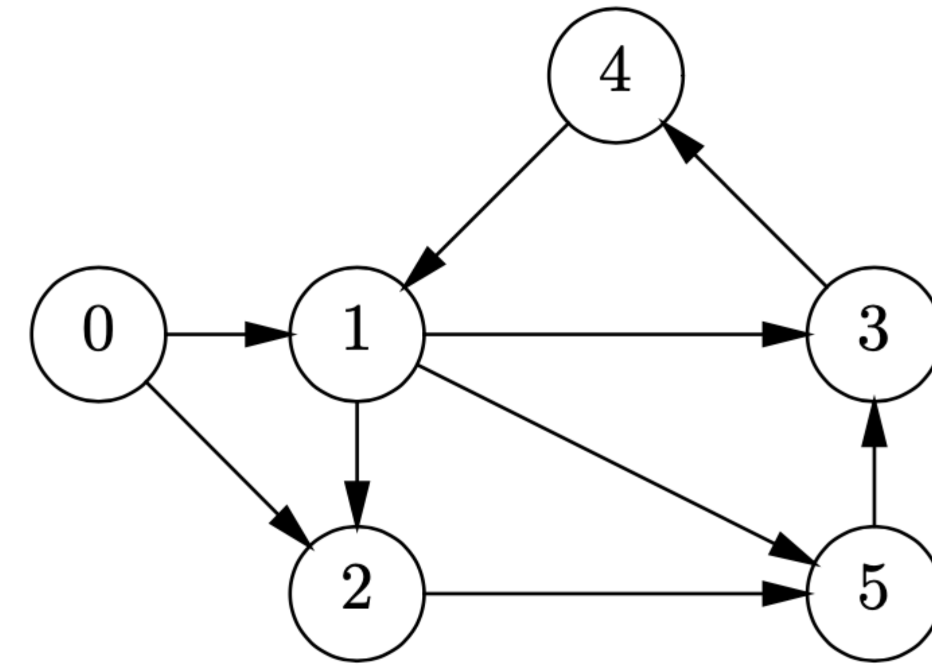
```
G = [[1,2], [2,5,3], [5], [4], [1], [3]]
```

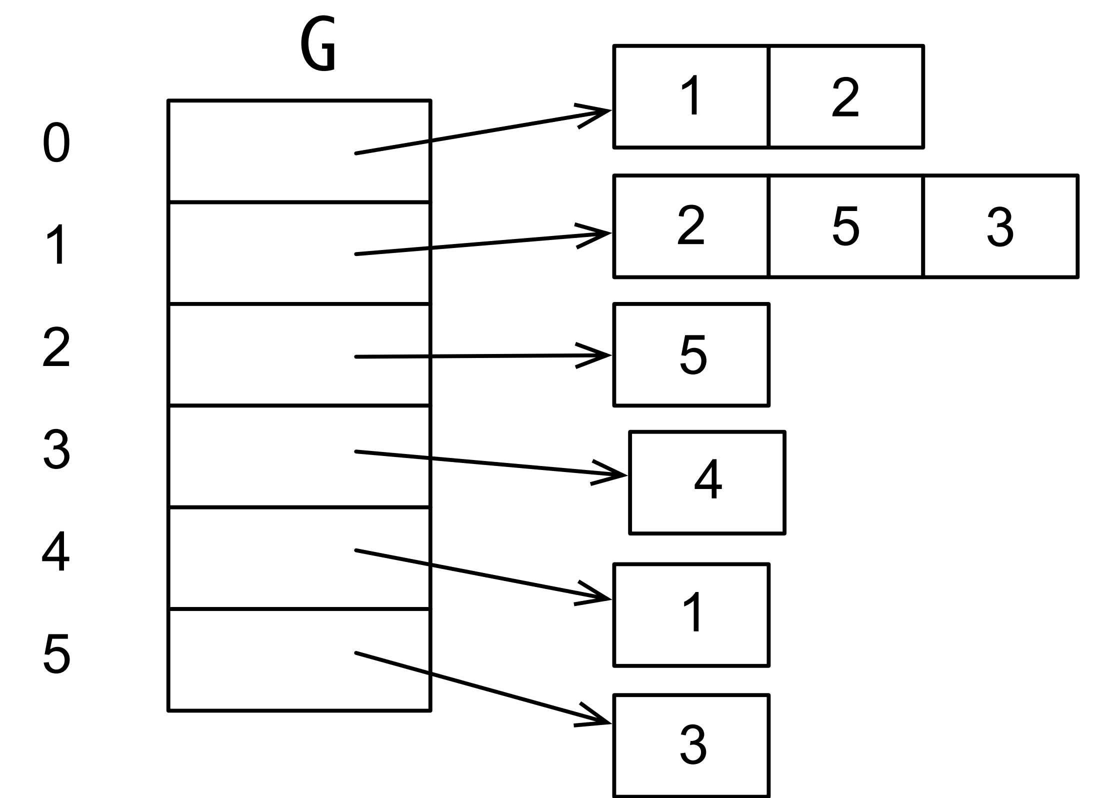




Connectivité

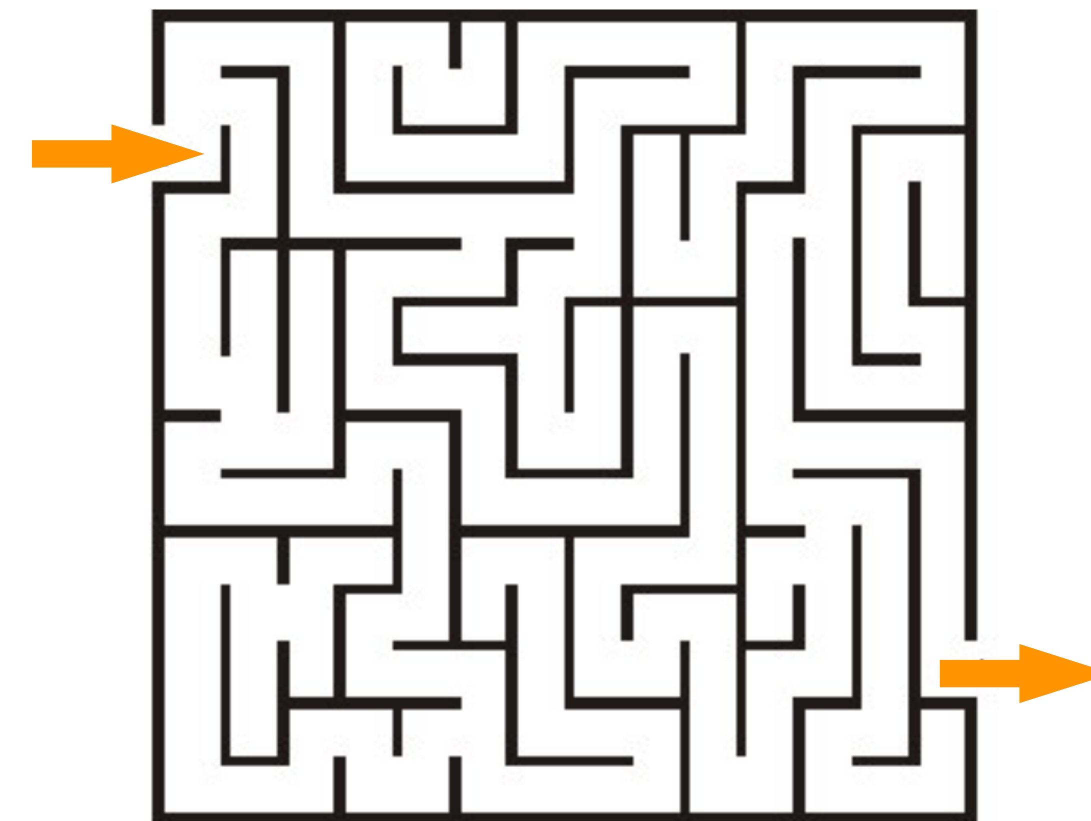
- la fermeture transitive cherche s'il existe un chemin entre les noeuds i et j pour toutes les paires (i, j)





- on peut chercher s'il existe un chemin entre 2 noeuds x et y donnés

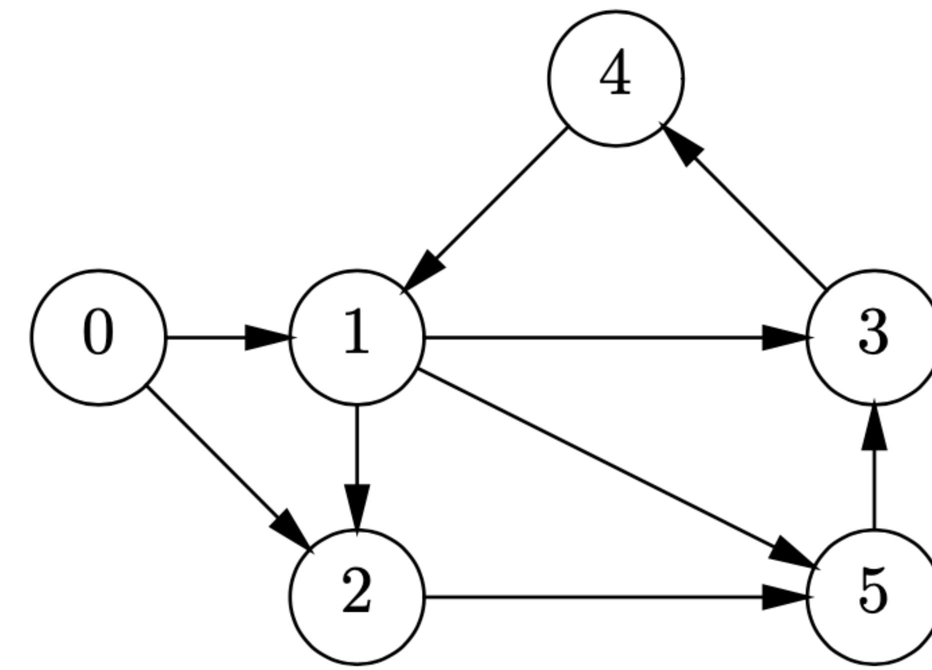
- méthode du fil d'Ariane (pour sortir d'un labyrinthe)

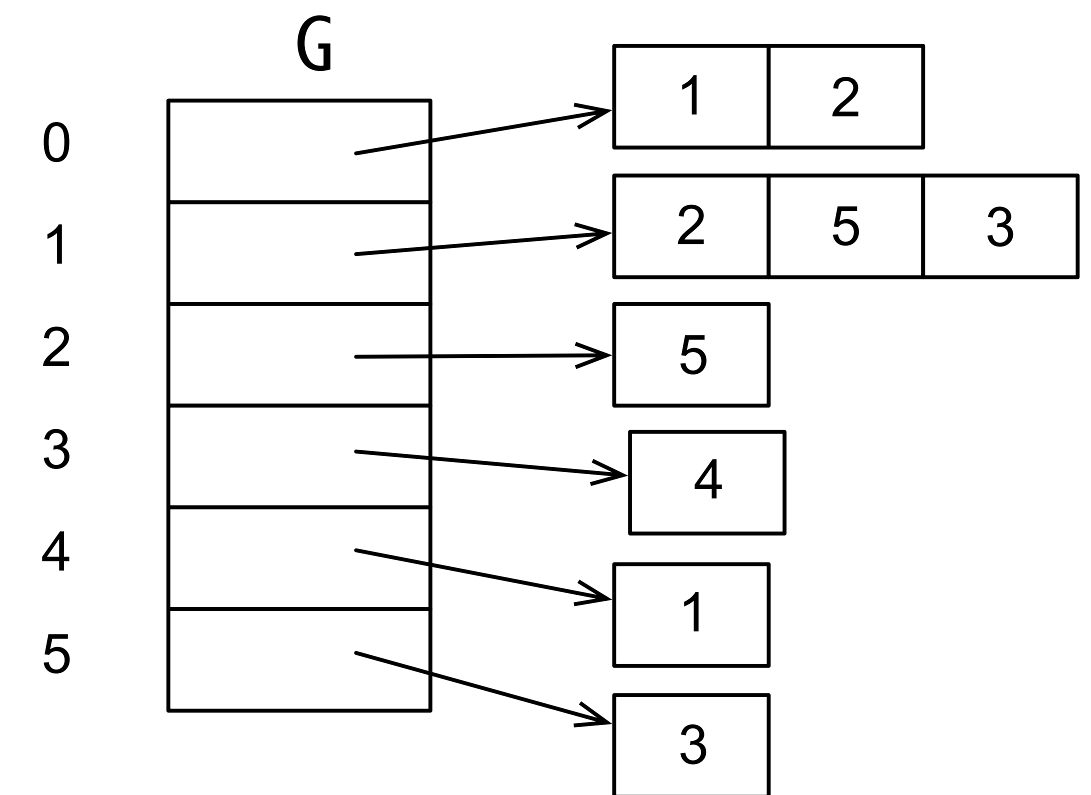


Connectivité

- recherche d'un chemin

```
G = [[1,2], [2,5,3], [5], [4], [1], [3]]

def reset () :
  global vu
  vu = (len (G)) * [False]


def existe_chemin (x, y) :
  global G, vu
  vu[x] = True
  if x == y :
      return True
  for z in G[x] :
      if not vu[z] :
         if existe_chemin (z, y) :
           return True
  return False
```





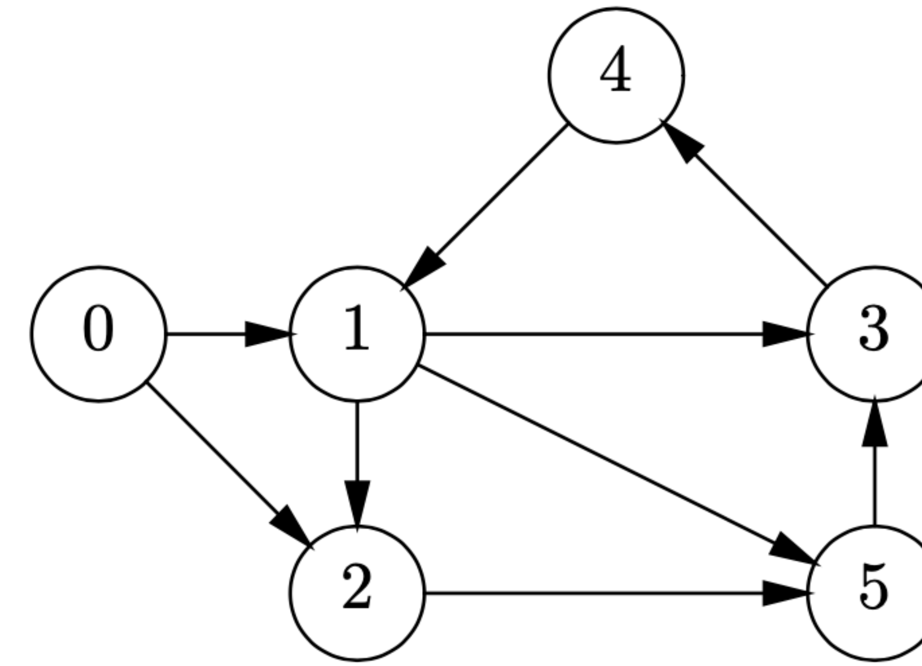
- vu représente l'ensemble des noeuds **visités**
- on ne visite qu'**une seule fois** chaque noeud (et chaque arc)
- la fonction existe_chemin prend donc un temps **linéaire** dans le nombre de noeuds et d'arcs

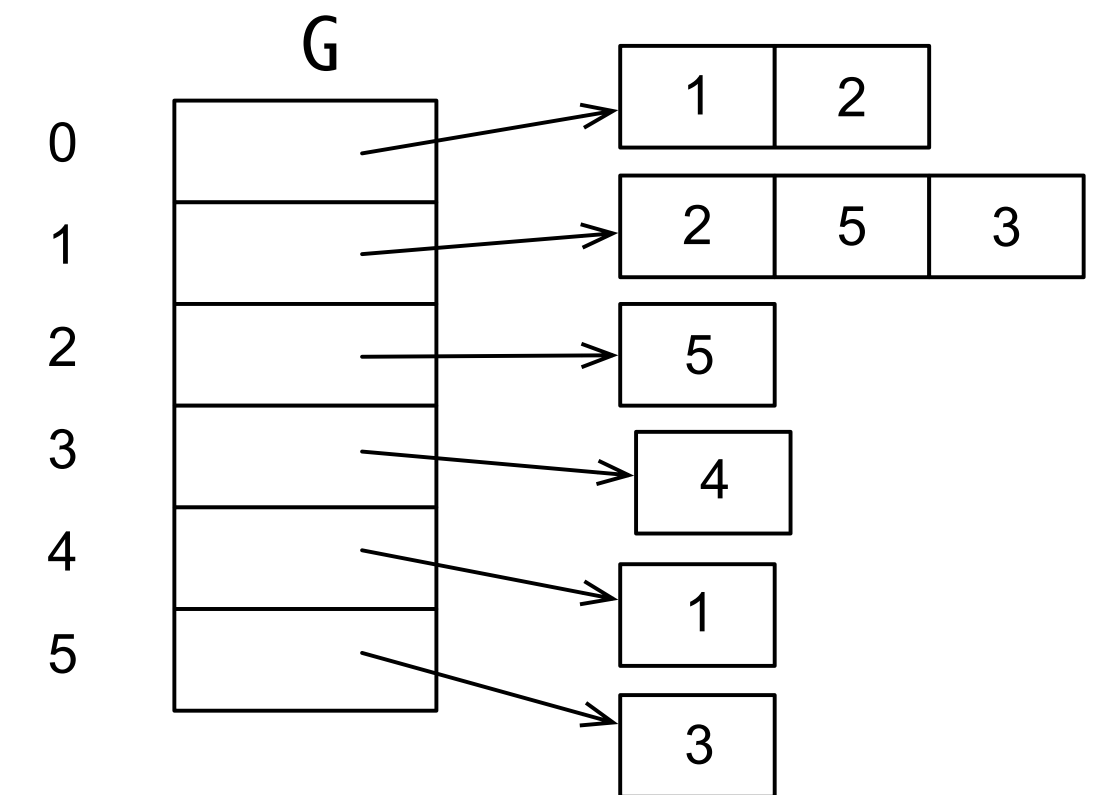
```
>>> existe_chemin (0, 3)
True
```

Connectivité

- recherche d'un chemin

```
G = [[1,2], [2,5,3], [5], [4], [1], [3]]

def reset () :
  global vu
  vu = (len (G)) * [False]


def chemin (x, y) :
  global G, vu
  vu[x] = True
  if x == y :
      return [x]
  for z in G[x] :
      if not vu[z] :
         r = chemin (z, y)
         if r != [ ] :
           return [x] + r
  return [ ]
```

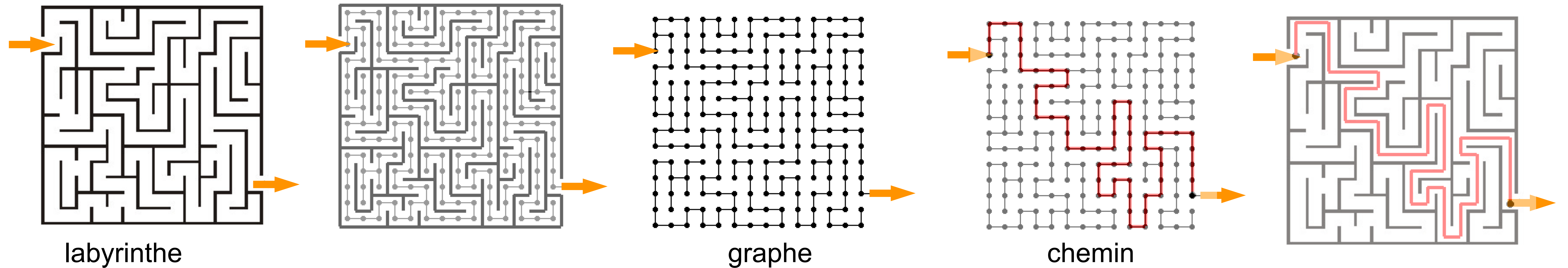




- on trouve un chemin en temps linéaire en les nombres de noeuds et d'arcs

```
>>> chemin (0, 3)
[0, 1, 2, 5, 3]
```

Labyrinthes



- ici, le labyrinthe est représenté par une matrice $n \times n$ (ici $n = 14$) dont chaque élément est un nombre binaire sur 4 bits représentant sa connexion avec ses voisins

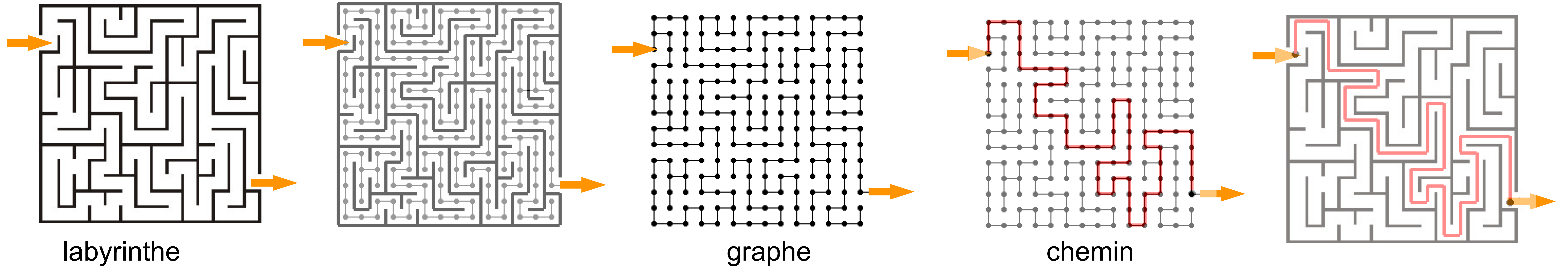
```
a = [[ 9,  5,  3,  9,  3, 11,  9,  5,  5,  3,  9,  5,  5,  3 ],
     [ 8,  3, 10, 10, 12, ...
```

- les noeuds du graphe associé sont des paires (i, j) où $0 \leq i < n, \quad 0 \leq j < n$

- le chemin vers la sortie est :

```
[(2, 0), (1, 0), (0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (3, 2), (3, 3), (3, 4),
 (3, 5), (4, 5), (4, 4), (4, 3) . . . (9, 13), (10, 13), (11, 13)]
```

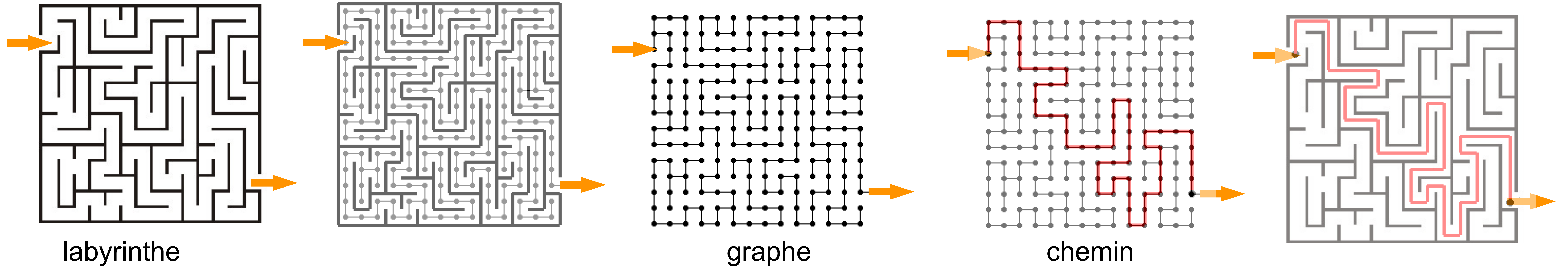
Labyrinthes



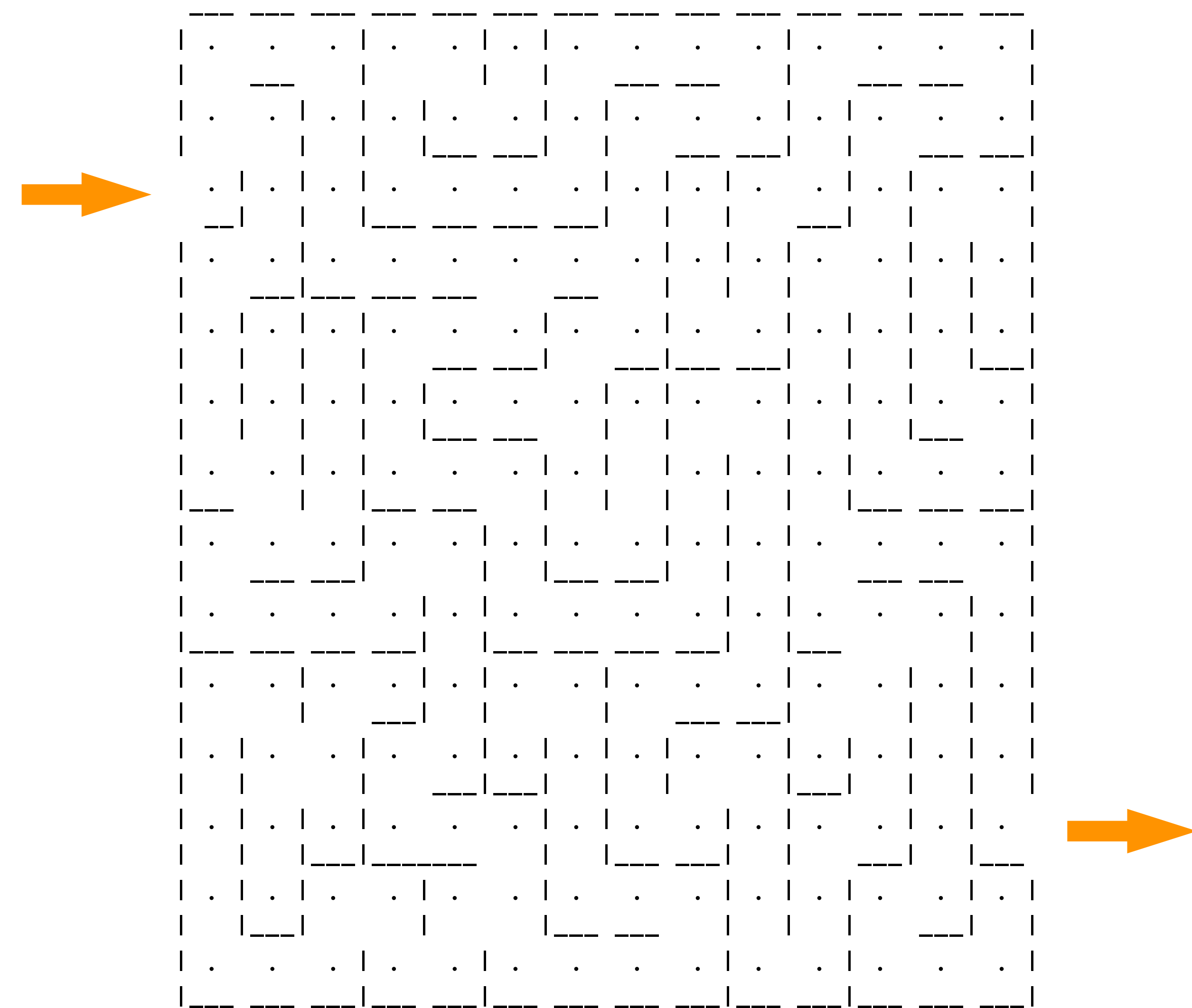
Exercice (facile) Entrer manuellement le graphe correspondant à un labyrinthe et écrire le programme qui trouve le chemin de sortie

Exercice (plus difficile) Ecrire le programme qui affiche le labyrinthe et le chemin de sortie

Labyrinthes



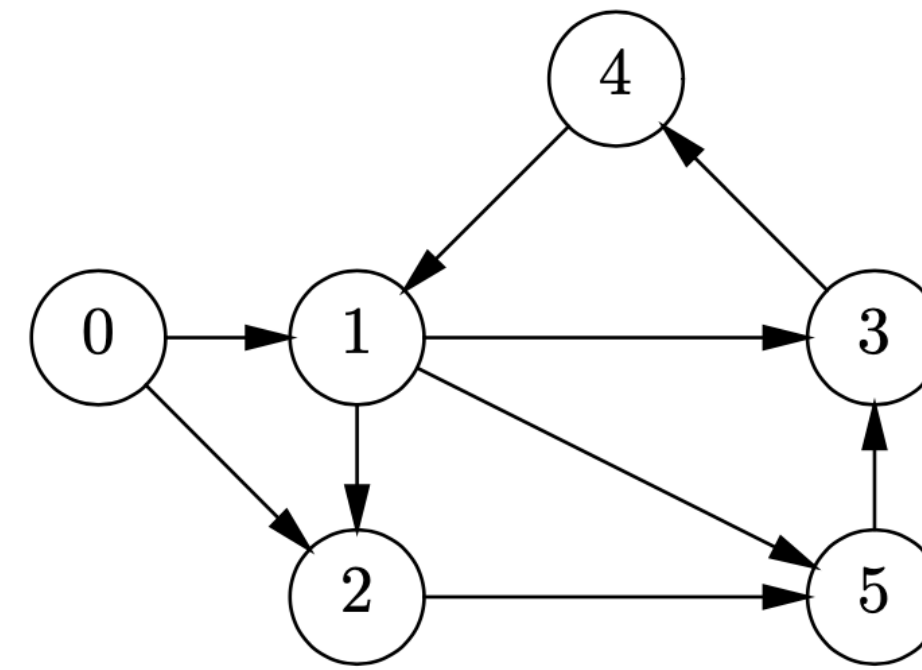
Exercice (plus difficile) Ecrire le programme qui affiche le labyrinthe et le chemin de sortie en alphanumérique

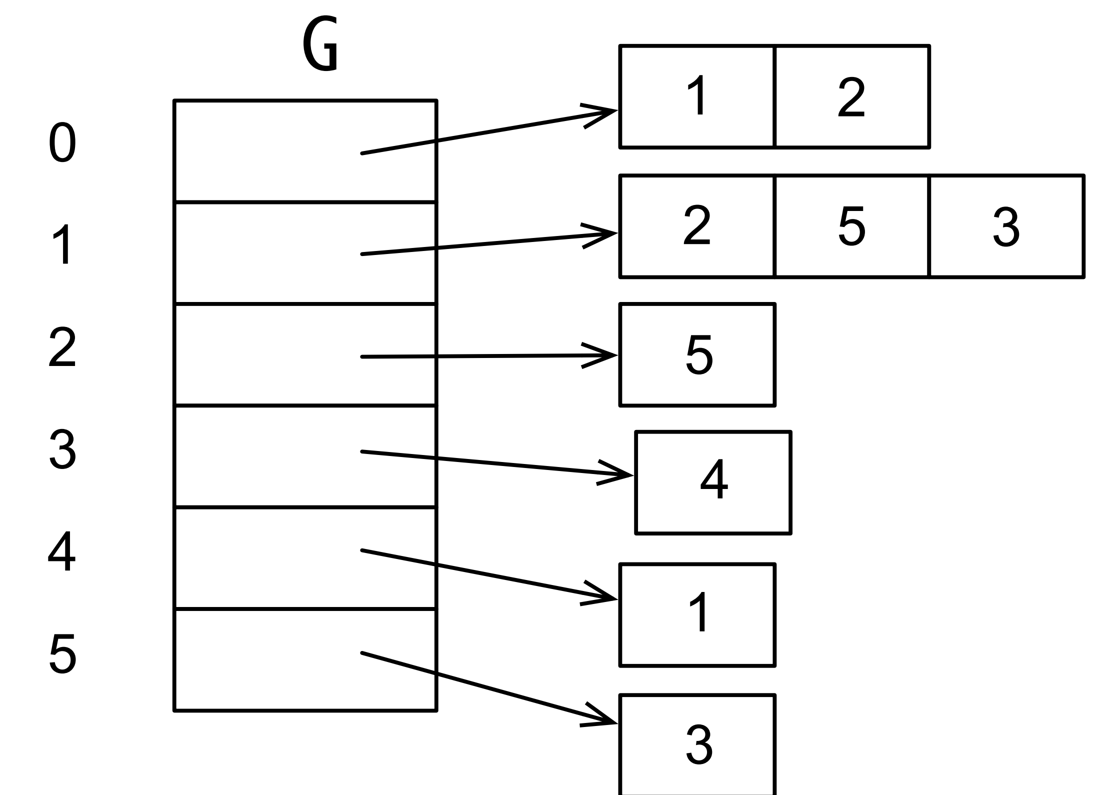


Exercice (difficile) Ecrire le programme qui génère des labyrinthes

Connectivité

- recherche du chemin le plus court






```
G = [[1,2], [2,5,3], [5], [4], [1], [3]]

def chemin_plus_court (x0, x1) :
  global G
  n = len (G); vu = n * [False]; p = n * [-1]
  fifo = [x0]
  vu[x0] = True
  while fifo != [ ] :
      x = fifo[0]; del fifo[0]
      if x == x1 :
        return chemin_arriere (p, x0, x1)
      for z in G[x] :
        if not vu[z] :
          fifo = fifo + [z]
          vu[z] = True; p[z] = x
  return [ ]
```

`fifo` est une file d'attente (*first in, first out*)

on note le père de z dans le tableau p

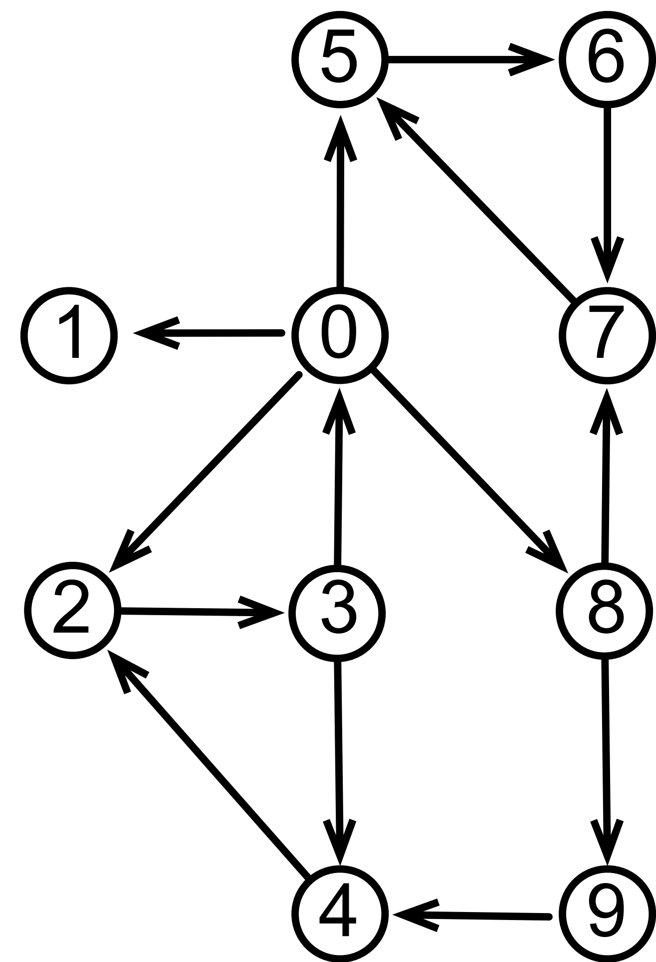
```
def chemin_arriere (p, x, y) :
   if x == y :
     return [x]
   else :
     return chemin_arriere (p, x, p[y]) + [y]
```

on restitue le chemin de x à y grâce au tableau p

- on trouve un chemin en temps linéaire en les nombres de noeuds et d'arcs

```
>>> chemin_plus_court (0, 3)
[0, 1, 3]
```

Parcours de graphes



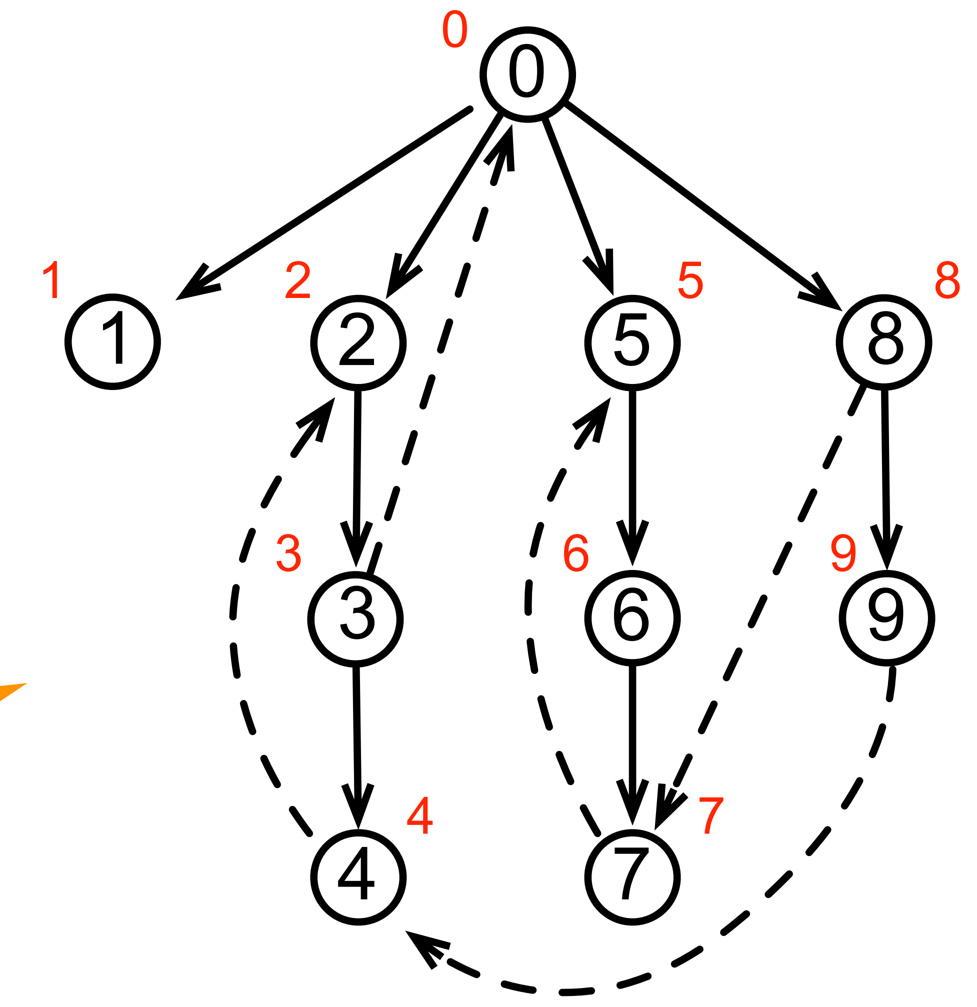
graphe

• dans la recherche du chemin, on effectue un parcourt du graphe en **profondeur** d'abord (*depth-first search*)

[on parcourt le graphe dans l'ordre des numéros rouges]

[avec une fonction récursive]

arbre de recouvrement
(*spanning tree*)

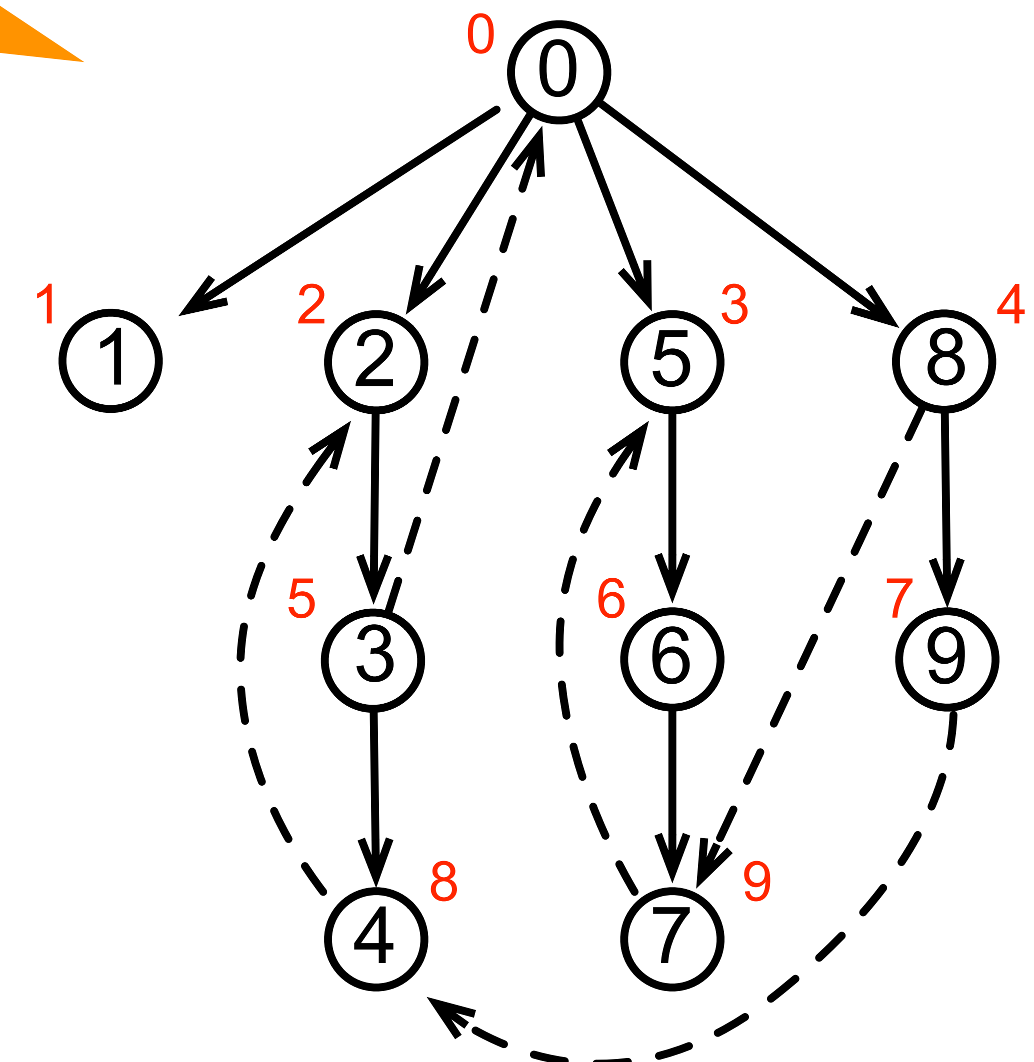


profondeur d'abord

• dans la recherche du chemin le plus court, on effectue un parcourt du graphe en **largeur** d'abord (*breadth-first search*)

[on parcourt le graphe dans l'ordre des numéros rouges]

[avec une file d'attente `fifo`]



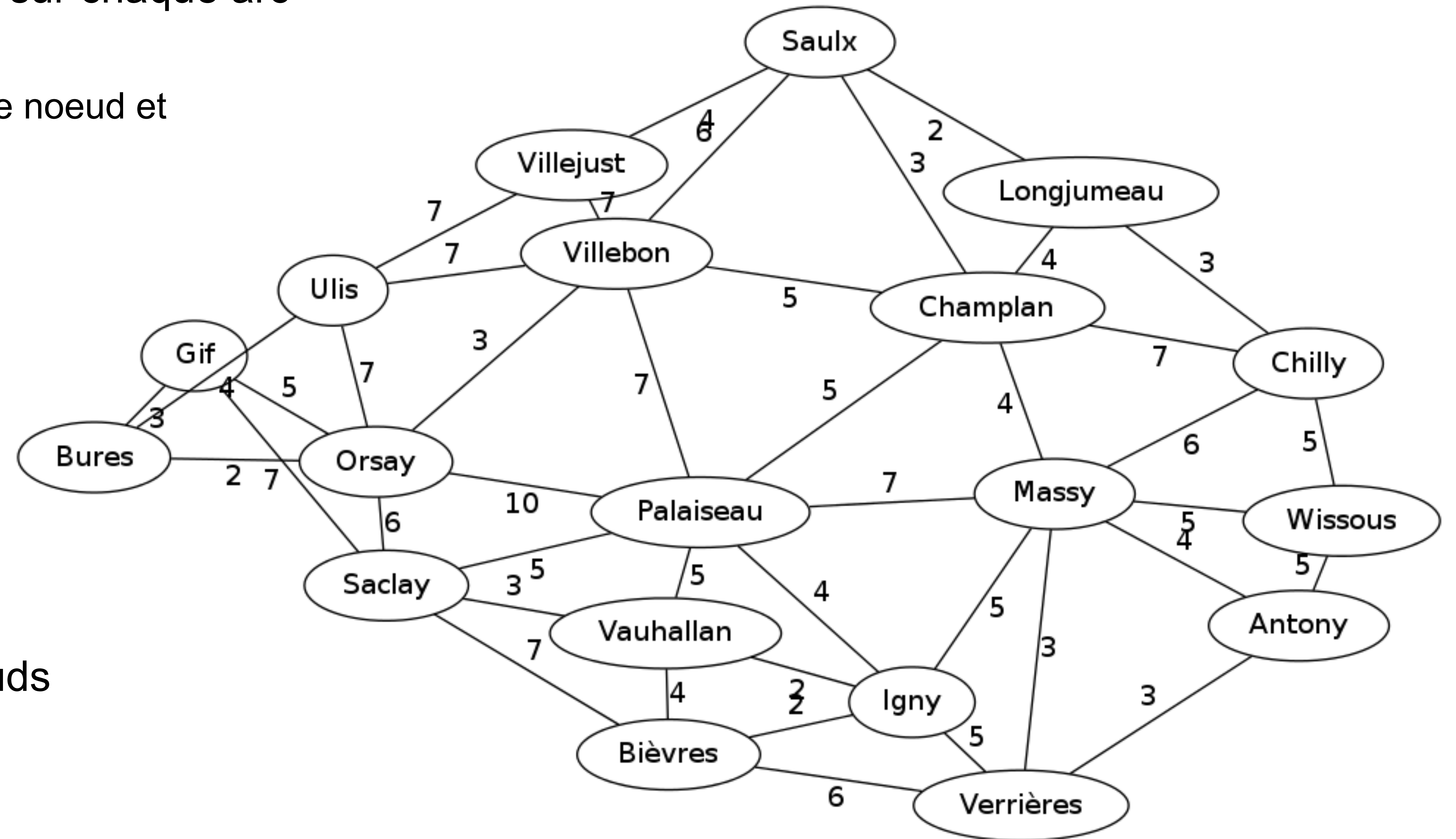
largeur d'abord

Plus court chemin dans un graphe valué

Question quel est le court chemin pour aller de Villebon à Bièvres ?

- le graphe (non orienté) a des distances (positives) sur chaque arc

[le tableau `succ` donne pour les successeurs de chaque noeud et les distances qui y mènent]



- on cherche le plus court chemin entre deux noeuds

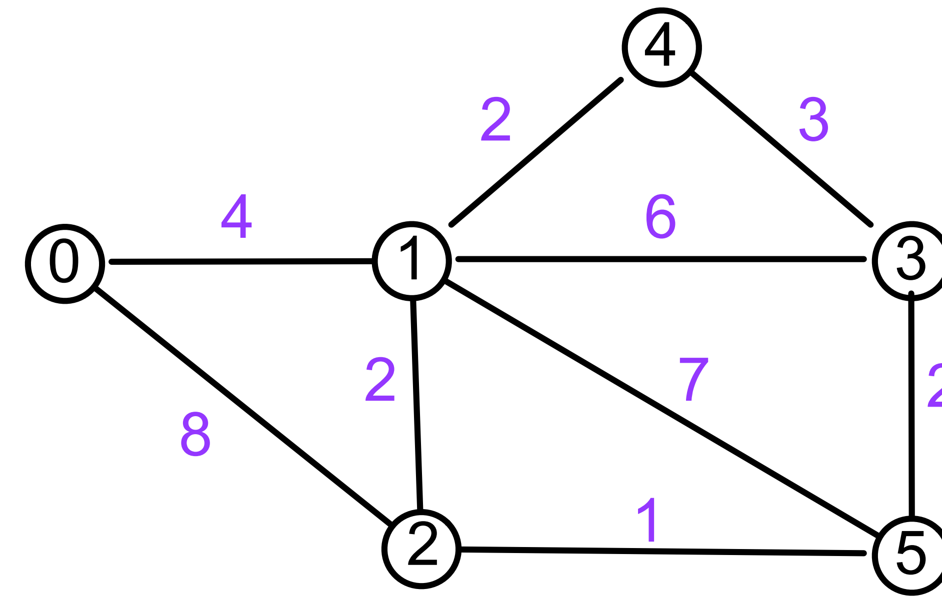
nombre de kms entre 2 noeuds

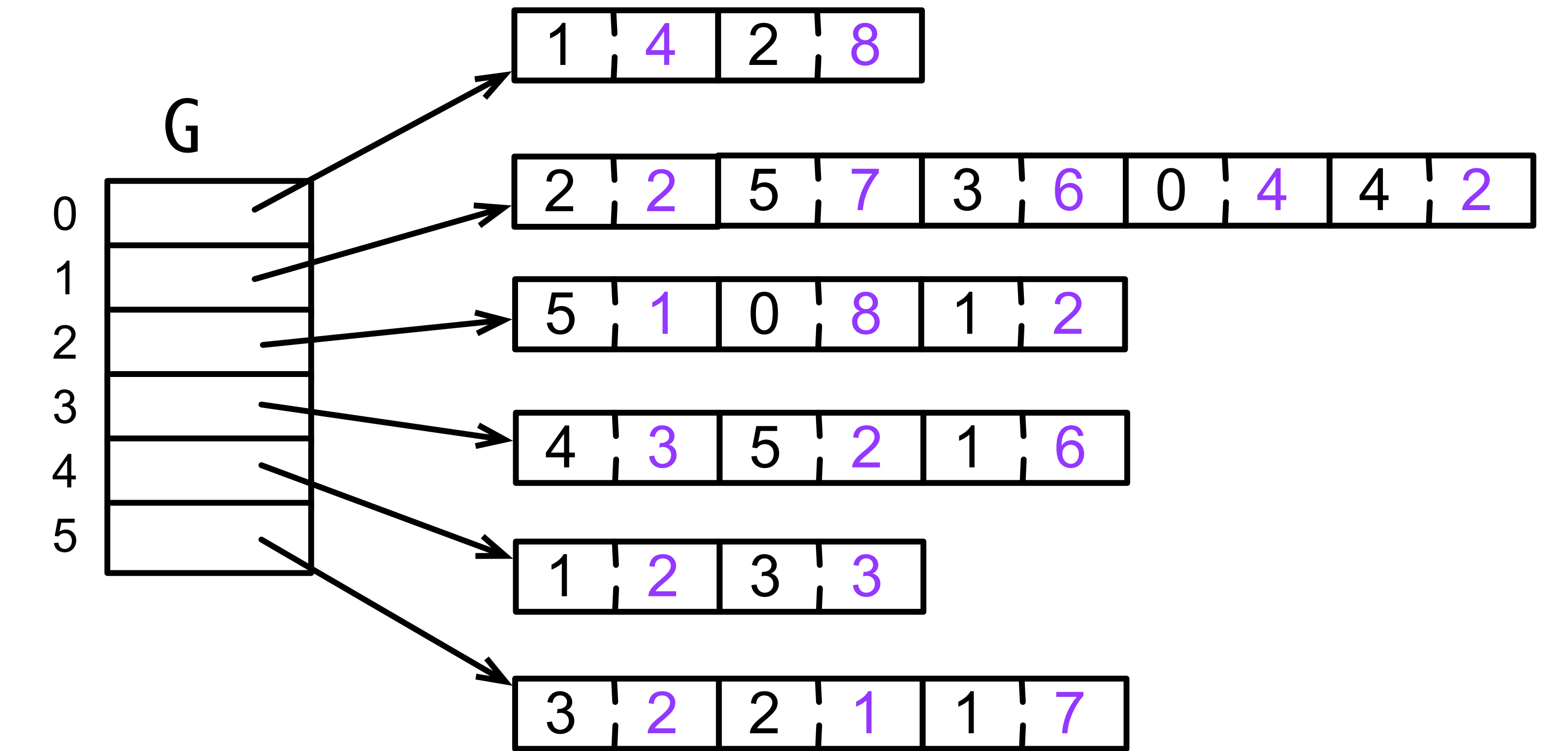
Plus court chemin dans un graphe valué

- le graphe non orienté a des distances sur chaque arc

Dijkstra, 1959

```
import sys; infini = sys.maxsize


def chemin_plus_court (x0, x1) :
  global G
  n = len (G); vu = n * [False]; p = n * [-1];
  d = n * [infini]
  d[x0] = 0
  while min(d) < infini :
      x = d.index (min(d))
      vu[x] = True
      if x == x1 :
        return chemin_arriere (p, x0, x1)
      for xz in G[x] :
        z = xz[0]; dxz = xz[1]
        if not vu[z] :
          if d[z] > d[x] + dxz :
            d[z] = d[x] + dxz
            p[z] = x
      d[x] = infini
  return [ ]
```






```
G = [[(1, 4), (2, 8)],
     [(2, 2), (5, 7), (3, 6), (0, 4), (4, 2)],
     [(5, 1), (0, 8), (1, 2)],
     [(4, 3), (5, 2), (1, 6)],
     [(1, 2), (3, 3)],
     [(3, 2), (2, 1), (1, 7)]]
```

```
>>> chemin_plus_court(0, 3)
[0, 1, 4, 3]
```

Plus court chemin dans un graphe valué

- le graphe non orienté a des distances sur chaque arc

Dijkstra, 1959

```
def chemin_plus_court (x0, x1) :
  global G
  n = len (G); vu = n * [False]; p = n * [-1];
  d = n * [infini]
  d[x0] = 0; f = FileP (x0, d)
  while not f.isEmpty(f) :
      x = f.popMax ()
      vu[x] = True
      if x == x1 :
        return chemin_arriere (p, x0, x1)
      for xz in G[x] :
        z = xz[0]; dxz = xz[1]
        if not vu[z] :
          if d[z] > d[x] + dxz :
            d[z] = d[x] + dxz
            p[z] = x; f.add(z)
      d[x] = infini
  return [ ]
```

f est une file de Priorité
(ordonnée selon le tableau d)

```
def chemin_plus_court (x0, x1) :
  global G
  n = len (G); vu = n * [False]; p = n * [-1]
  fifo = [x0]
  vu[x0] = True
  while fifo != [ ] :
      x = fifo[0]; del fifo[0]
      if x == x1 :
        return chemin_arriere (p, x0, x1)
      for z in G[x] :
        if not vu[z] :
          fifo = fifo + [z]
          vu[z] = True; p[z] = x
  return [ ]
```

- en utilisant une file de priorité, on ne fait pas plus que $n \log n$ où n est la taille du graphe

Prochainement

- exploration et *backtracking*