Informatique et Programmation

Cours 16

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

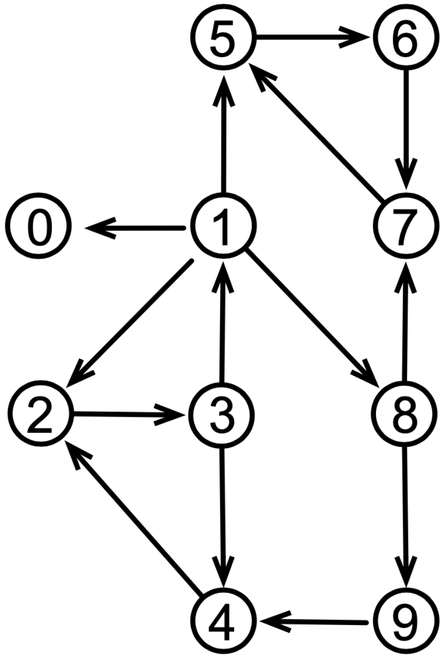
http://jeanjacqueslevy.net/prog-py

Plan

- graphes orientés
- parcours de graphes
- arbres de recouvrement
- connexité
- tri topologique

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

Graphes orientés

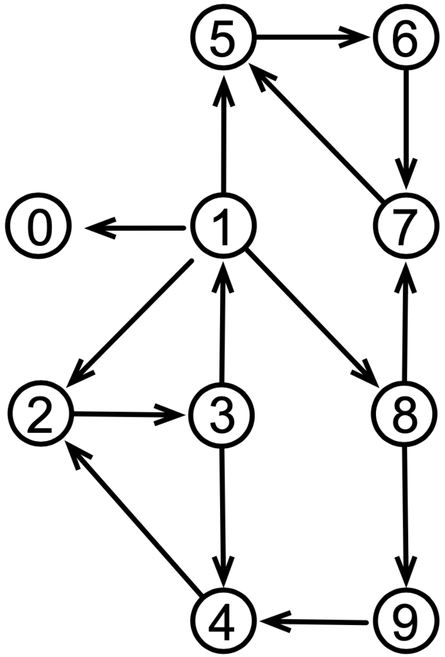



```
class Sommet :
    def __init__ (self, s, l) :
        self.nom = s
        self.voisins = l
    #
    def __str__ (self) :
        return "Sommet ('{}', {})".format (self.nom, self.voisins)



graphe = [
    Sommet ('0', []),
    Sommet ('1', [0, 2, 5, 8]),
    Sommet ('2', [3]),
    Sommet ('3', [1, 4]),
    Sommet ('4', [2]),
    Sommet ('5', [6]),
    Sommet ('6', [7]),
    Sommet ('7', [5]),
    Sommet ('8', [7, 9]),
    Sommet ('9', [4])
    ]
```

Graphes orientés




```python
class Sommet :
    def __init__ (self, s, l) :
        self.nom = s
        self.voisins = l
    #
    def __str__ (self) :
        return "Sommet ('{}', {})".format (self.nom, self.voisins)


def print_graphe (g) :
    print ('[', end ='\n\t')
    print (*g, sep = ',\n\t')
    print (']')
```

impression starred expression

```
>>> print (*[1, 2, 3], sep = ':: ')
1:: 2:: 3
>>> print (*[1, 2, 3])
1 2 3
```

caractères spéciaux

`'\n'` linefeed

`'\t'` tabulation

Connexité

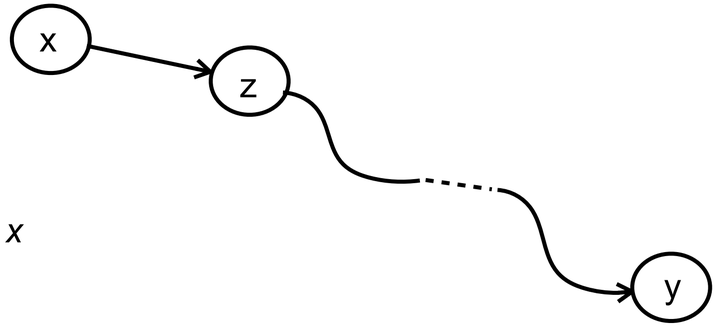
- cakculer un chemin possible pour aller du sommet *x* au sommet *y*

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

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

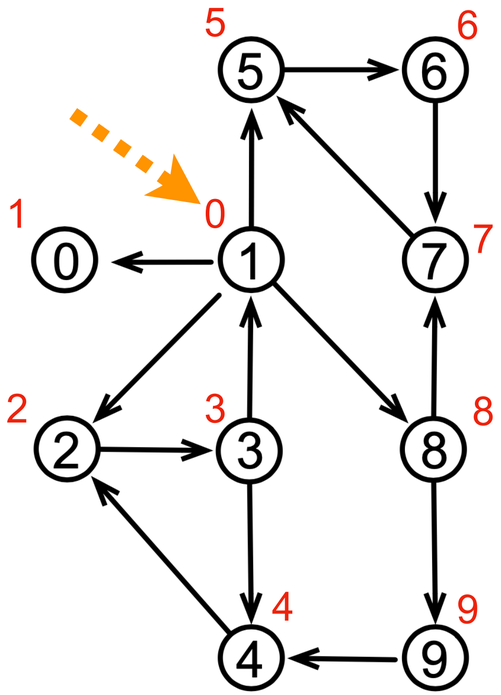
- même programme que pour les graphes non orientés

- mais s'il existe un chemin de *x* à *y*, il peut ne pas y avoir de chemin de *y* à *x*

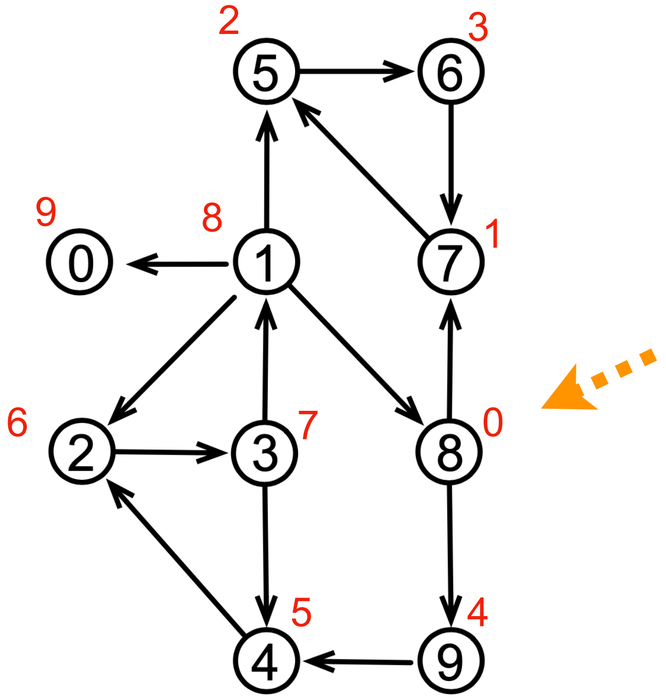


Parcours de graphe

- parcours en **profondeur d'abord** (*depth first search — DFS*)



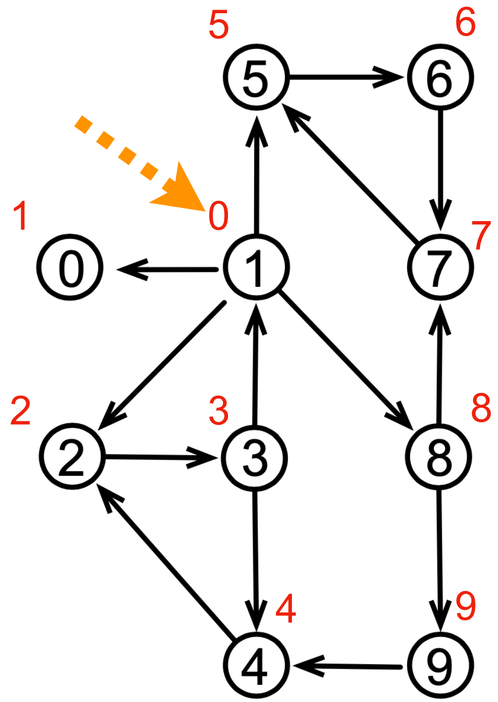
- début du parcours en 1



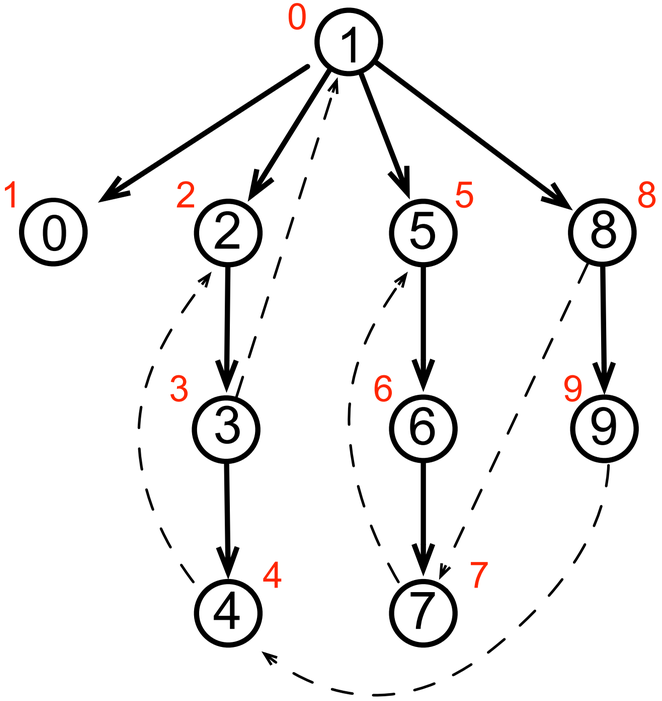
- début du parcours en 8

Parcours de graphe

- parcours en **profondeur d'abord** (*depth first search — DFS*)



arbre de recouvrement
[spanning tree]

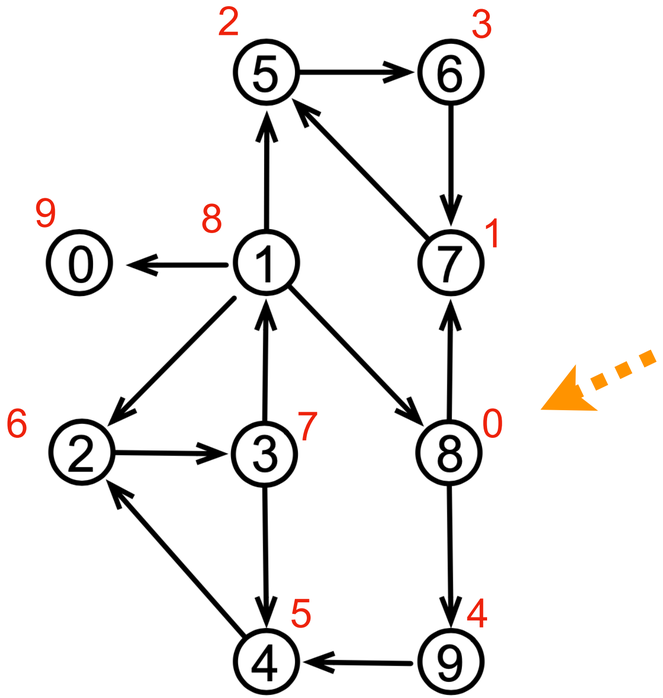


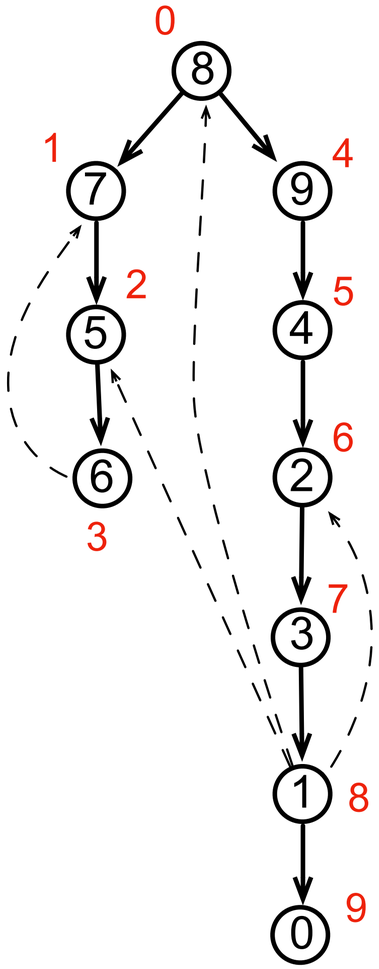
- début du parcours en 1

- les numéros du parcours en profondeur d'abord sont les numéros de l'ordre préfixe sur l'arbre de recouvrement

Parcours de graphe

- parcours en profondeur d'abord (*depth first search — DFS*)





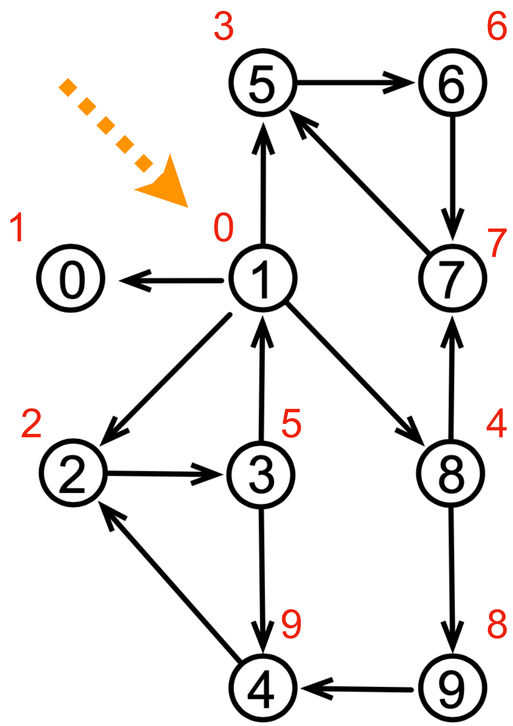
arbre de recouvrement
[spanning tree]

- début du parcours en 8

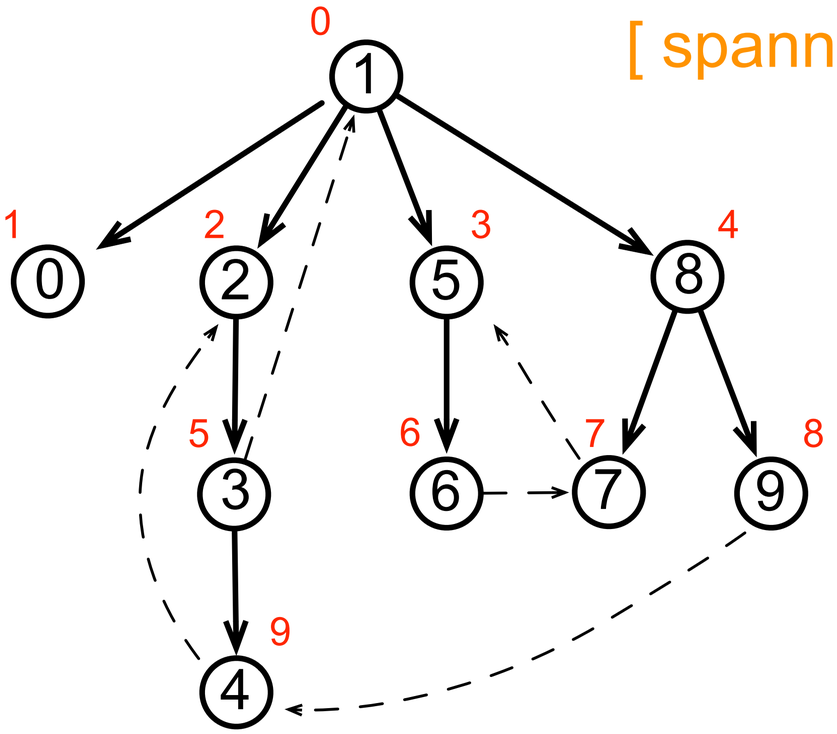
- les numéros du parcours en profondeur d'abord sont les numéros de l'ordre préfixe sur l'arbre de recouvrement

Parcours de graphe

- parcours en largeur d'abord (*breadth first search — BFS*)



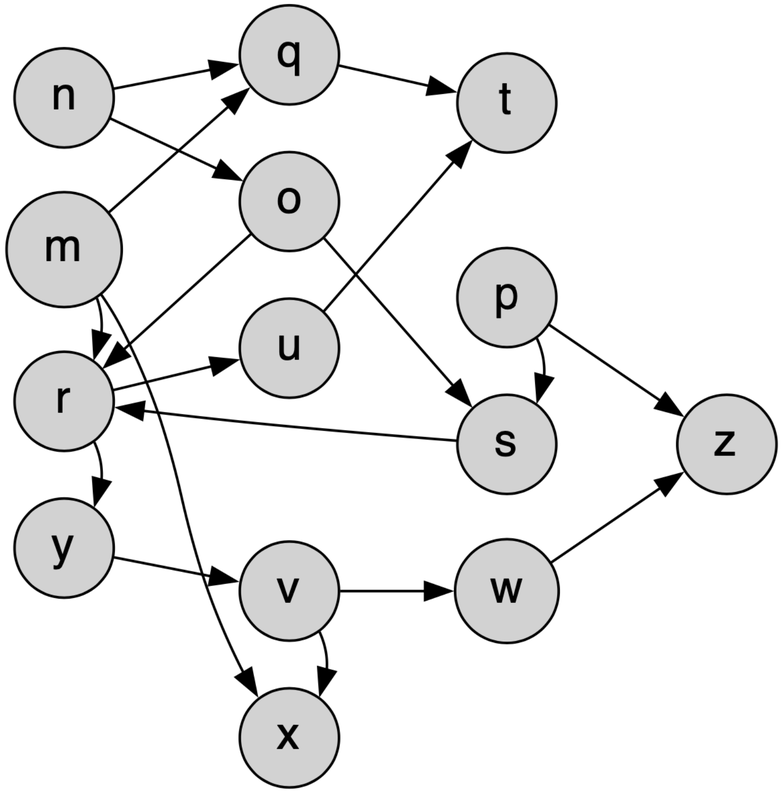
arbre de recouvrement
[spanning tree]

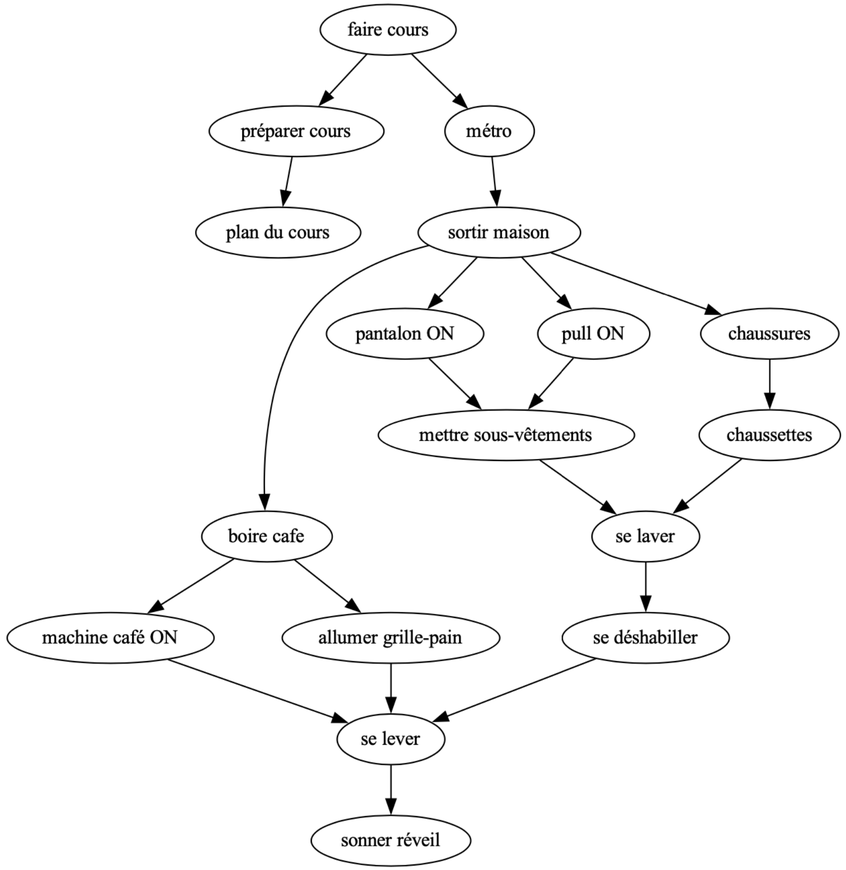


- début du parcours en 1

- sur l'arbre de recouvrement, l'ordre de parcours est l'ordre militaire (selon la distance à partir de la racine)

Graphe sans cycle — dag (directed acyclic graph)

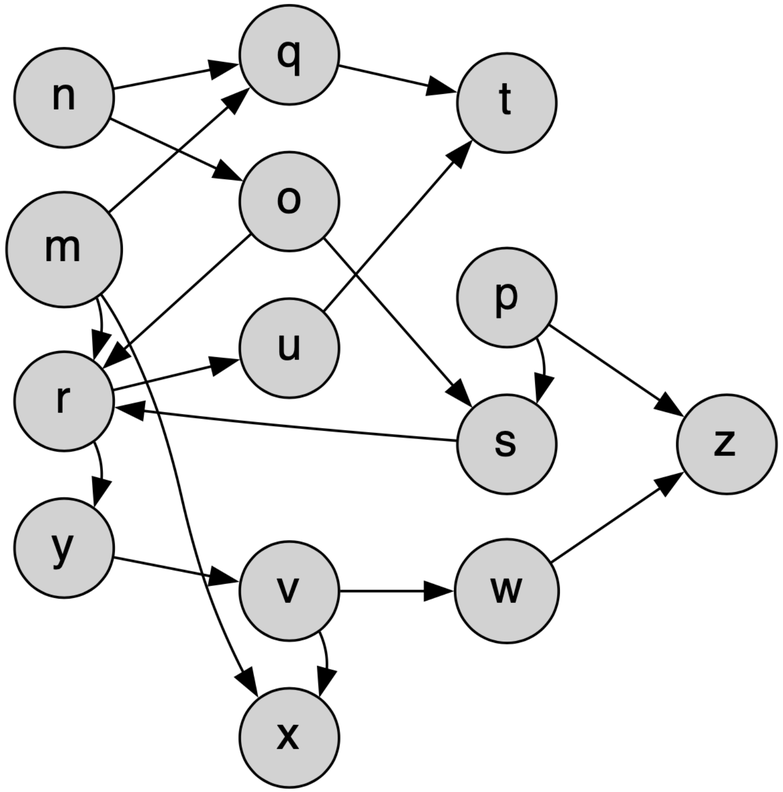


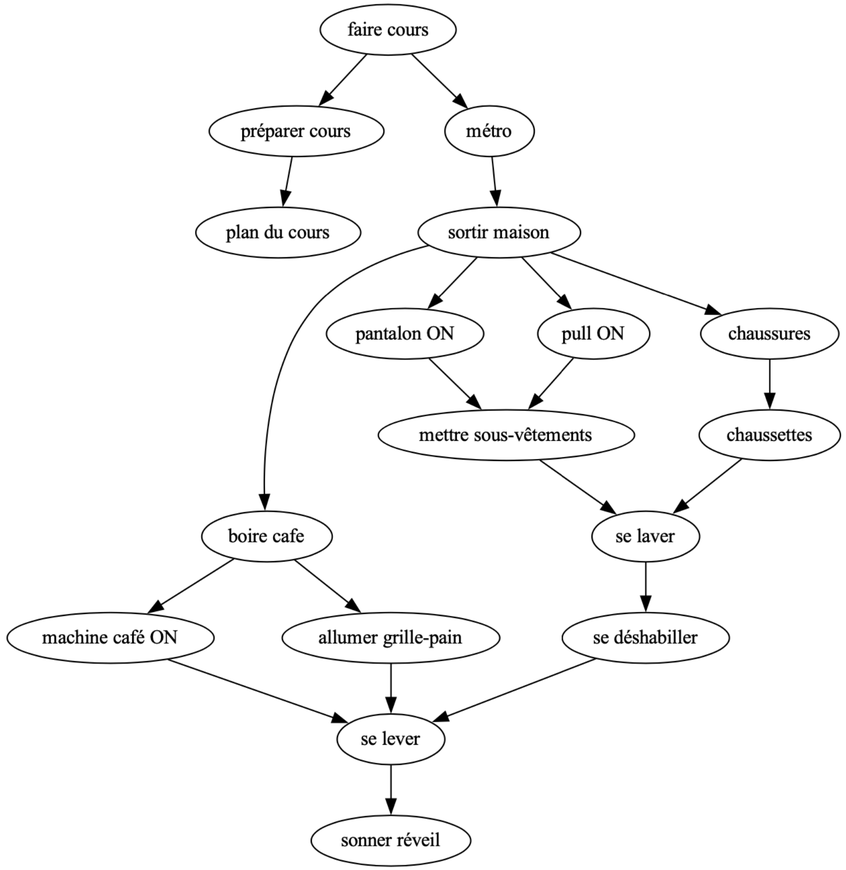


Exercice Ecrire un programme qui teste si un graphe est sans cycle

Tri topologique

- tri topologique liste les sommets dans un ordre cohérent avec les dépendancesi dans un graphe sans cycle





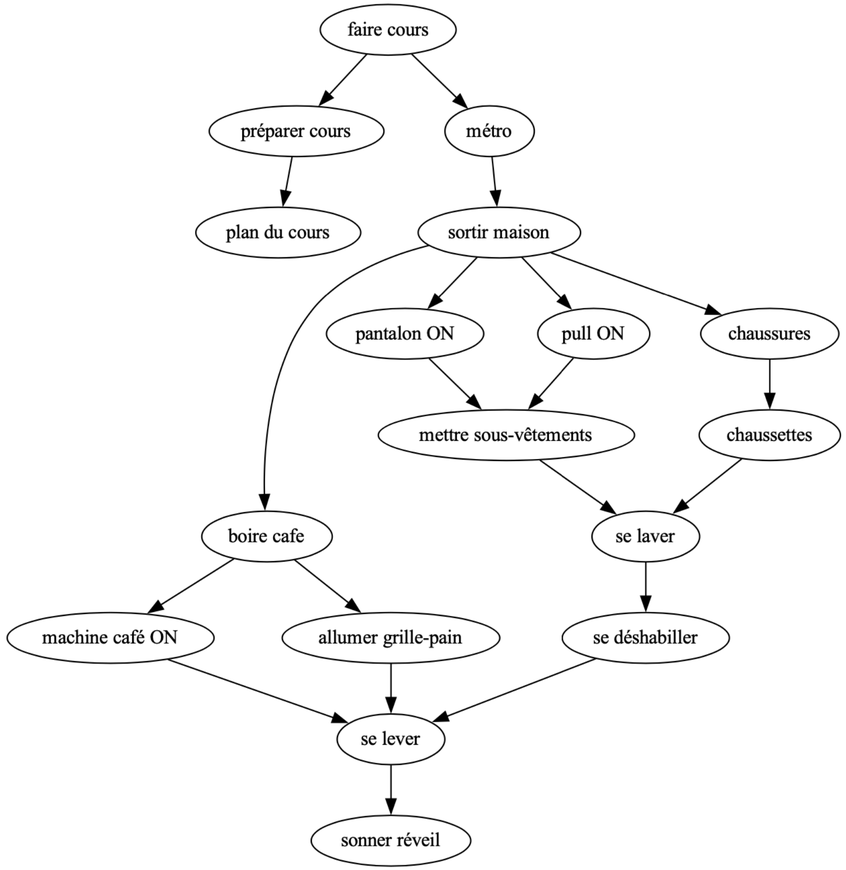
Tri topologique

- tri topologique liste les sommets dans un ordre cohérent avec les dépendancesi dans un graphe sans cycle

```
[ "plan du cours", "préparer cours",
 "sonner réveil", "se lever", "se déshabiller",
 "se laver", "chaussettes",
 "mettre sous-vêtements", "pull ON",
 "pantalon ON",
 "allumer grille-pain", "machine café ON",
 "boire cafe",
 "chaussures", "sortir maison",
 "métro", "faire cours" ]
```

ou encore

```
[ "plan du cours", "sonner réveil", "se lever",
 "se déshabiller", "se laver", "chaussettes",
 "mettre sous-vêtements", "pull ON",
 "pantalon ON",
 "allumer grille-pain", "machine café ON",
 "préparer cours",
 "boire cafe",
 "chaussures", "sortir maison",
 "métro", "faire cours" ]
```

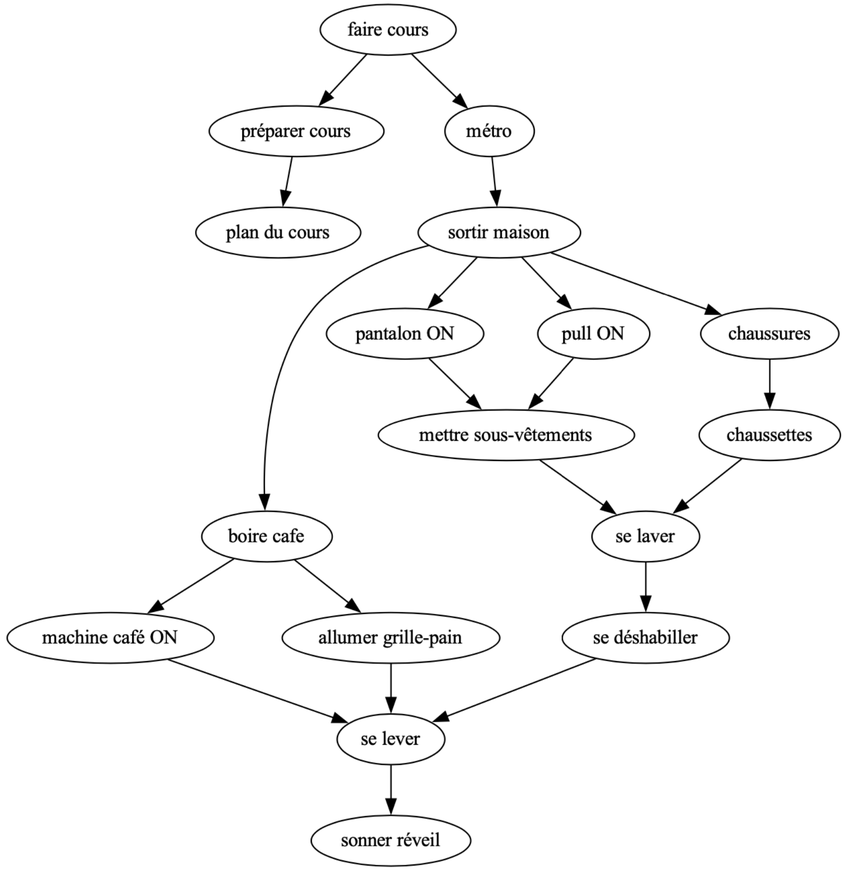


Tri topologique

- tri topologique liste les sommets dans un ordre cohérent avec les dépendancesi dans un graphe sans cycle

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


def un_ordre_possible (g) :
    n = len (g); r = [ ]
    dejaVu = [False] * n
    for x in range(n) :
        r = r + tri_topologique (g, x, dejaVu)
    return r
```



Python ++

- fonctions anonymes — lambda expression

```
noms = [ "plan du cours",
 "chaussures", "sortir maison",
 "sonner réveil", .. ]

dag = list(map (lambda nx: Sommet (nx, [ ]), noms))
```

fonction anonyme

```
(lambda nx: Sommet (nx, [ ]) ("chaussures")
```

```
Sommet ("chaussures", [ ])
```

- opérateur fonctionnel : map

`map f [1, 2, 3]` identique à `[ f(1), f(2), f(3) ]`

[en fait, ca donne un objet qu'on transforme en liste avec `list ( )`]

- lire les tutoriels, par exemple `http://www.w3schools.com/python/`

Python ++

- beaucoup d'exemples en python avec Google ou autre indexeur
- lire les tutoriels, par exemple `http://www.w3schools.com/python/`
- utiliser `help()` en mode terminal
- sous `help()` et taper `modules` pour avoir la liste des modules
- faire `import random` pour charger le module
- et `help(random)` donne une info sur ce module
- aussi `dir(random)` donne toutes les méthodes et attributs de la classe `random`

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