Algorithmes, Programmation, IA

Cours 5

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

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

Plan

- graphes acyclique
- tri topologique
- composantes connexes
- points d’articulation
- matrices d’adjacence

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

Représentation des graphes

- fichier mes_graphes.py

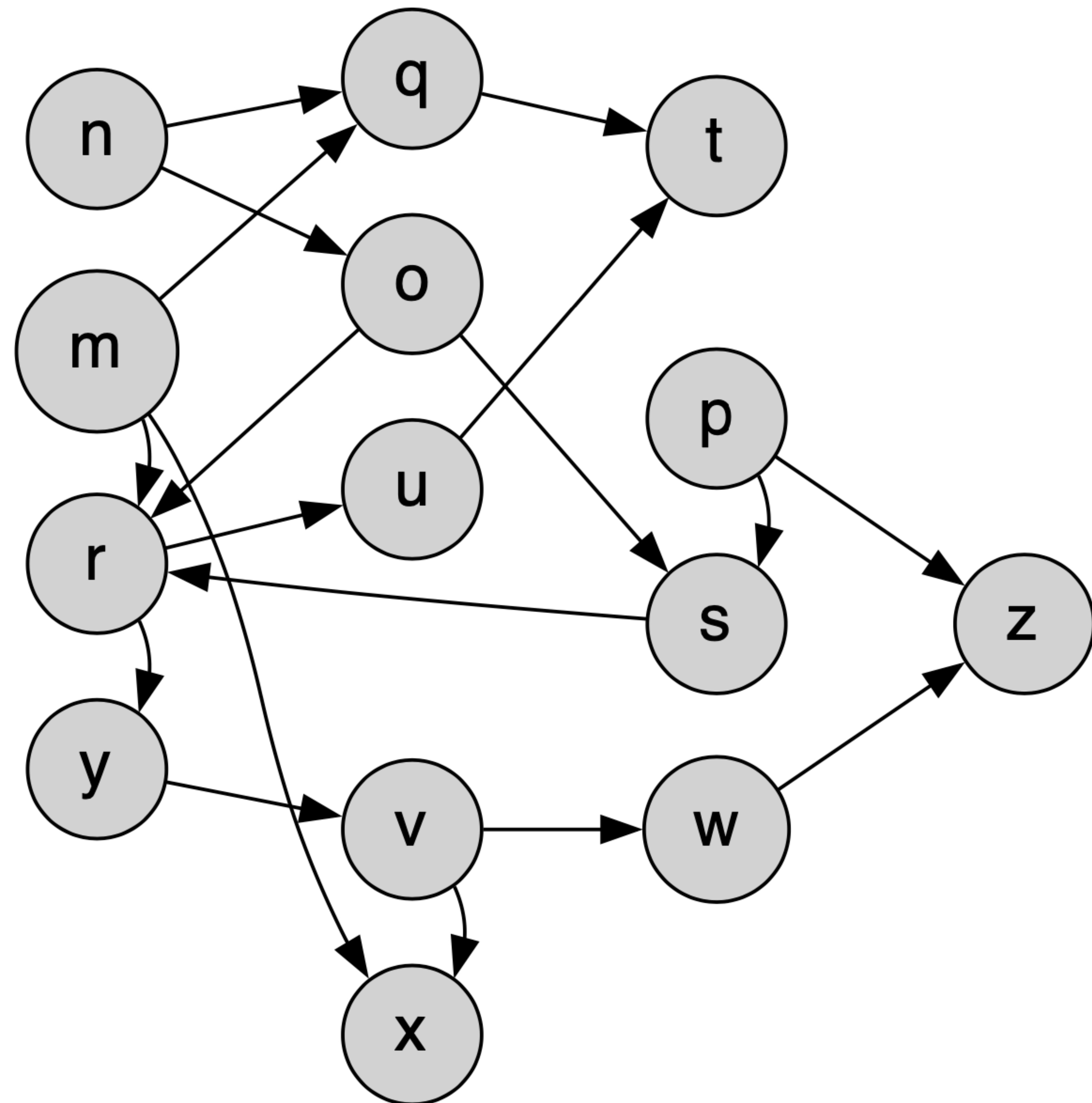
```python
class Graph :
    def __init__ (self, vs) :
        self.vertices = vs
        self.succ = {w:[ ] for w in vs}

    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]


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)]
```

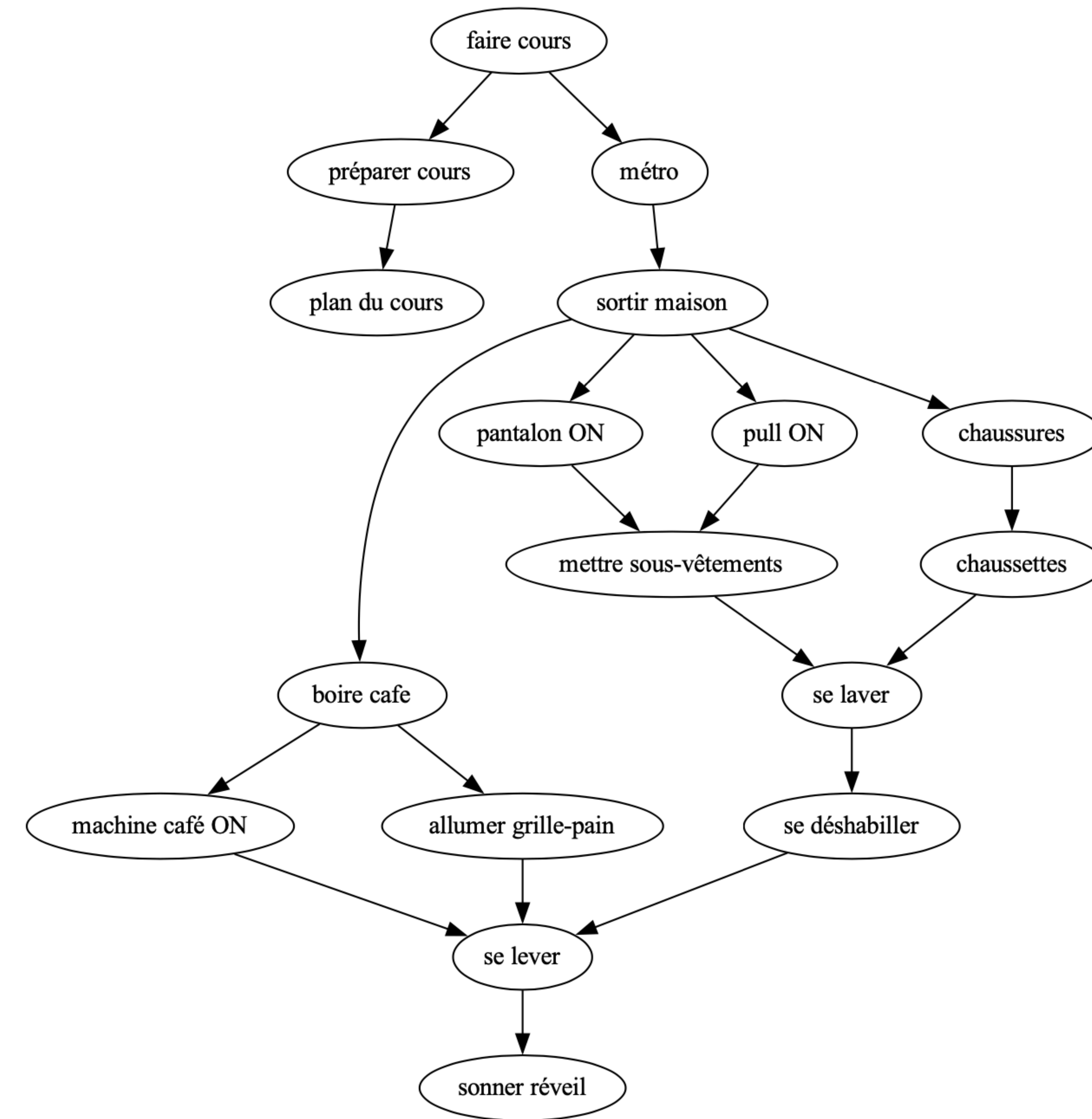
```python
def add_edges (g, edges) :
    for (v, w) in edges :
        g.add_edge (v, w)

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

def add_vedges (g, edges) :
    for (v, w, weight) in edges :
        g.add_edge (v, w, weight)

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

```python
class Stack :
    def __init__(self) :
        self.contents = []

    def __str__(self) :
        return '{}'.format (self.contents)

    def push (self, x) :
        self.contents = [x] + self.contents

    def pop (self) :
        x = self.contents[0]
        del self.contents[0]
        return x

    def is_empty (self) :
        return self.contents == [ ]


class Queue :
    def __init__(self) :
        self.contents = []

    def __str__(self) :
        return '{}'.format (self.contents)

    def enq (self, x) :
        self.contents = self.contents + [x]

    def deq (self) :
        x = self.contents[0]
        del self.contents[0]
        return x

    def is_empty (self) :
        return self.contents == [ ]
```

```python
class PQueue :
    def __init__(self) :
        self.contents = []

    def __str__(self) :
        return '{}'.format (self.contents)

    def enq (self, x) :
        self.contents = self.contents + [x]

    def deq (self) :
        a = self.contents
        i = a.index(min(a))
        x = self.contents[i]
        del self.contents[i]
        return x

    def update (self, x, y) :
        i = self.contents.index(x)
        self.contents[i] = y

    def is_empty (self) :
        return self.contents == [ ]
```

Graphe acyclique



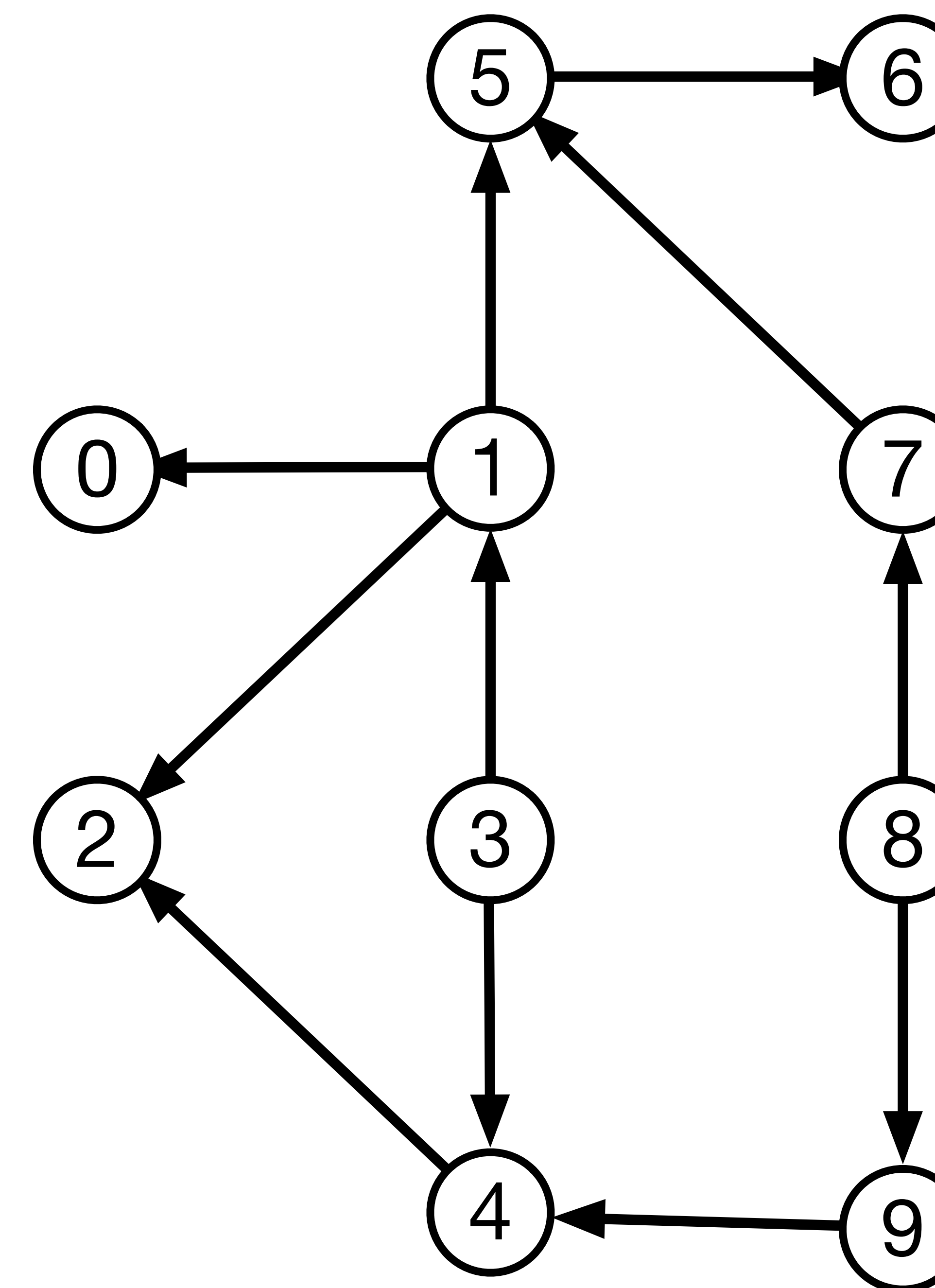


Exercice Ecrire un programme qui teste si un graphe est sans cycle

Graphe acyclique

- graphe sans cycle *(dag, directed acyclic graph)*
- tester si un graphe est acyclique avec depth first search tricolore

```
WHITE = 0; GRAY = 1; BLACK = 2

def is_dag1 (g, v, color) :
    res = True;
    color[v] = GRAY
    for w in g.successors(v) :
        if color[w] == GRAY :
            print ('cycle en ', w)
            return False
        if color[w] == WHITE :
            res = res and is_dag1 (g, w, color, p)
    color[v] = BLACK
    return res

def is_dag (g) :
    color = {v: WHITE for v in g.vertices}
    res = True
    for v in g.vertices :
        if color[v] == WHITE :
            res = res and is_dag1 (g, v, color)
    return res
```



is_dag en $O(V + E)$

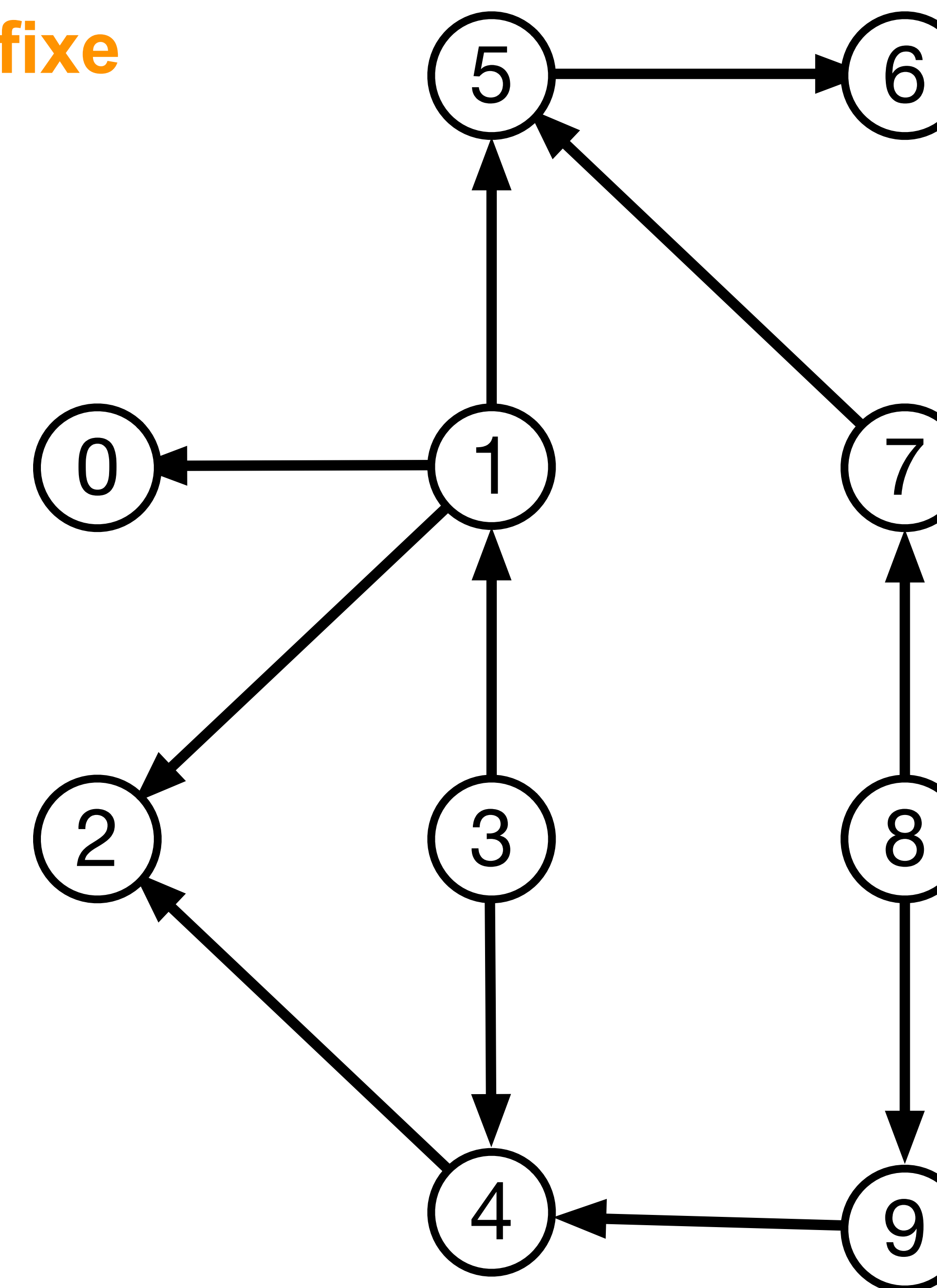
et donc linéaire en nombre de sommets et d'arrêtes

Tri topologique

- le graphe de dépendances de tâches est acyclique
- trouver l'ordre dans lequel il faut faire un projet

```
def tsort1 (g, v, visited) :
    visited[v] = True
    return tsort_vs (g, g.successors (v), visited) + [v]

def tsort_vs (g, vs, visited) :
    res = [ ]
    for v in vs :
        if not visited[v] :
            res += tsort1 (g, v, visited)
    return res

def tsort (g) :
    visited = {v: False for v in g.vertices}
    return tsort_vs (g, g.vertices, visited)
```

résultat postfixe



- simple dfs avec résultat postfixe

tsort en $O(V + E)$

et donc linéaire en nombre de sommets et d'arrêtes

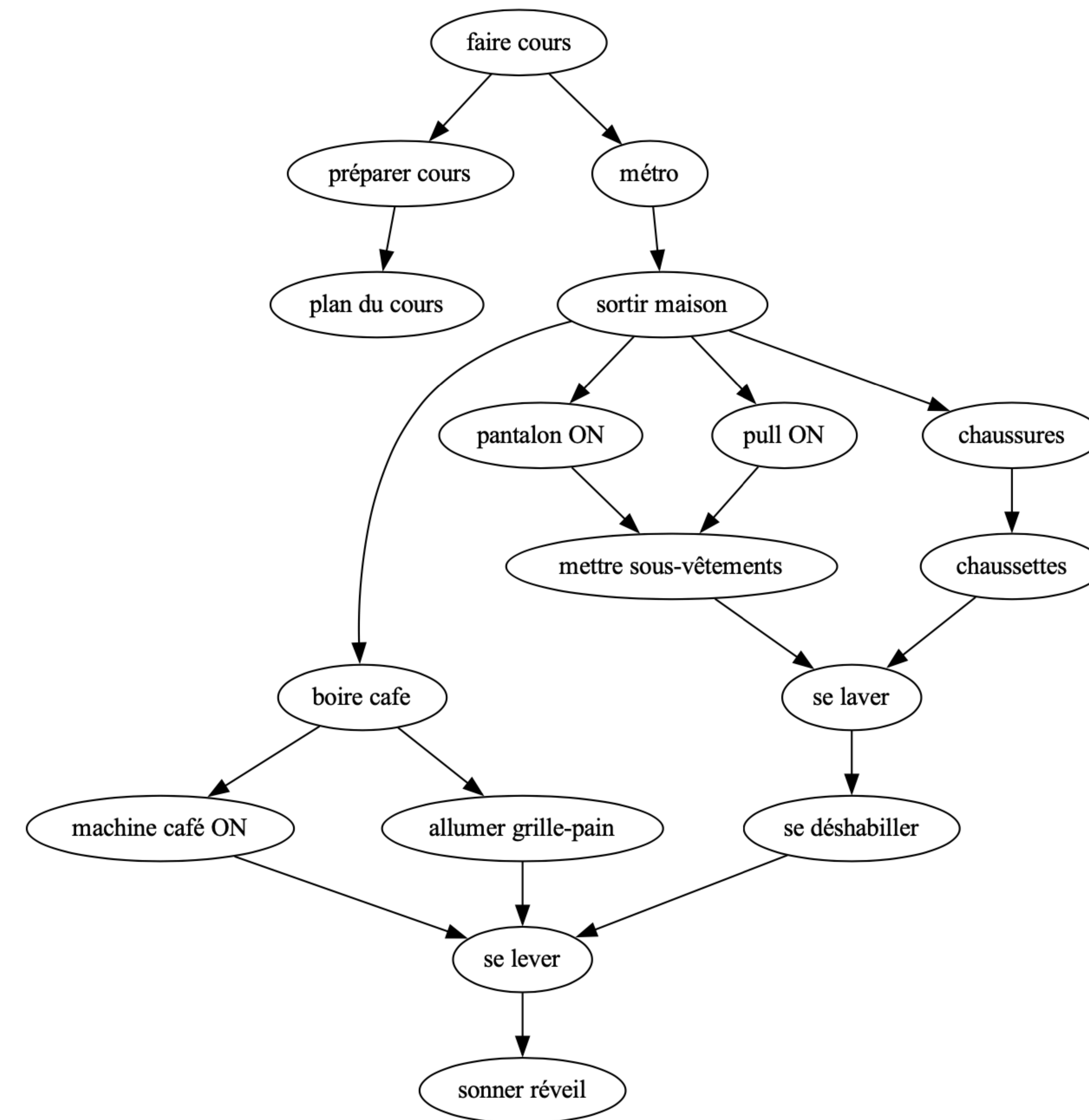
Tri topologique

- résultats possibles (liste des tâches à faire dans l'ordre inverse des dépendances)

```
[ "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" ]
```



Composantes connexes

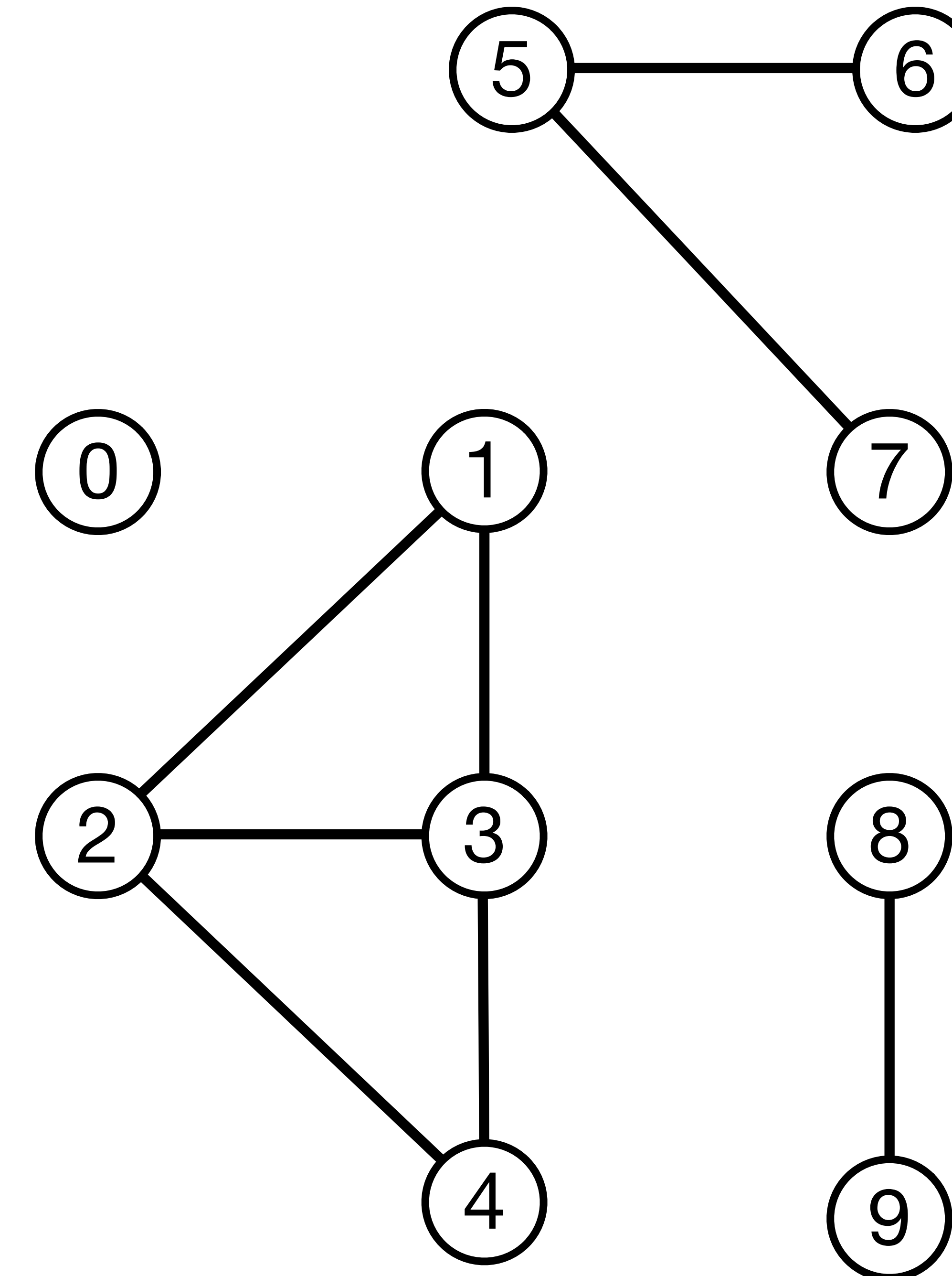
- trouver les ensemble de sommets interconnectés
- parcours en profondeur d'abord dans un graphe non dirigé

```
def comp_conn1 (g, v, visited) :
    visited[v] = True
    res = [v]
    for v in g.successors(v) :
        if not visited[v] :
            res += comp_conn1 (g, v, visited)
    return res

def comp_conn (g) :
    visited = {v: False for v in g.vertices}
    res = []
    for v in g.vertices :
        if not visited[v] :
            res += [comp_conn1 (g, v, visited)]
    return res
```

liste de listes

- graphe non dirigé

```
g3 = Graph ([i for i in range(10)])
add_uedges (g3, [(1,2),
  (3,1), (3,4), (4,2), (5,6),
  (7,5), (8,9)])

print (comp_conn (g3))
```



4 composantes connexes

Point d'articulation

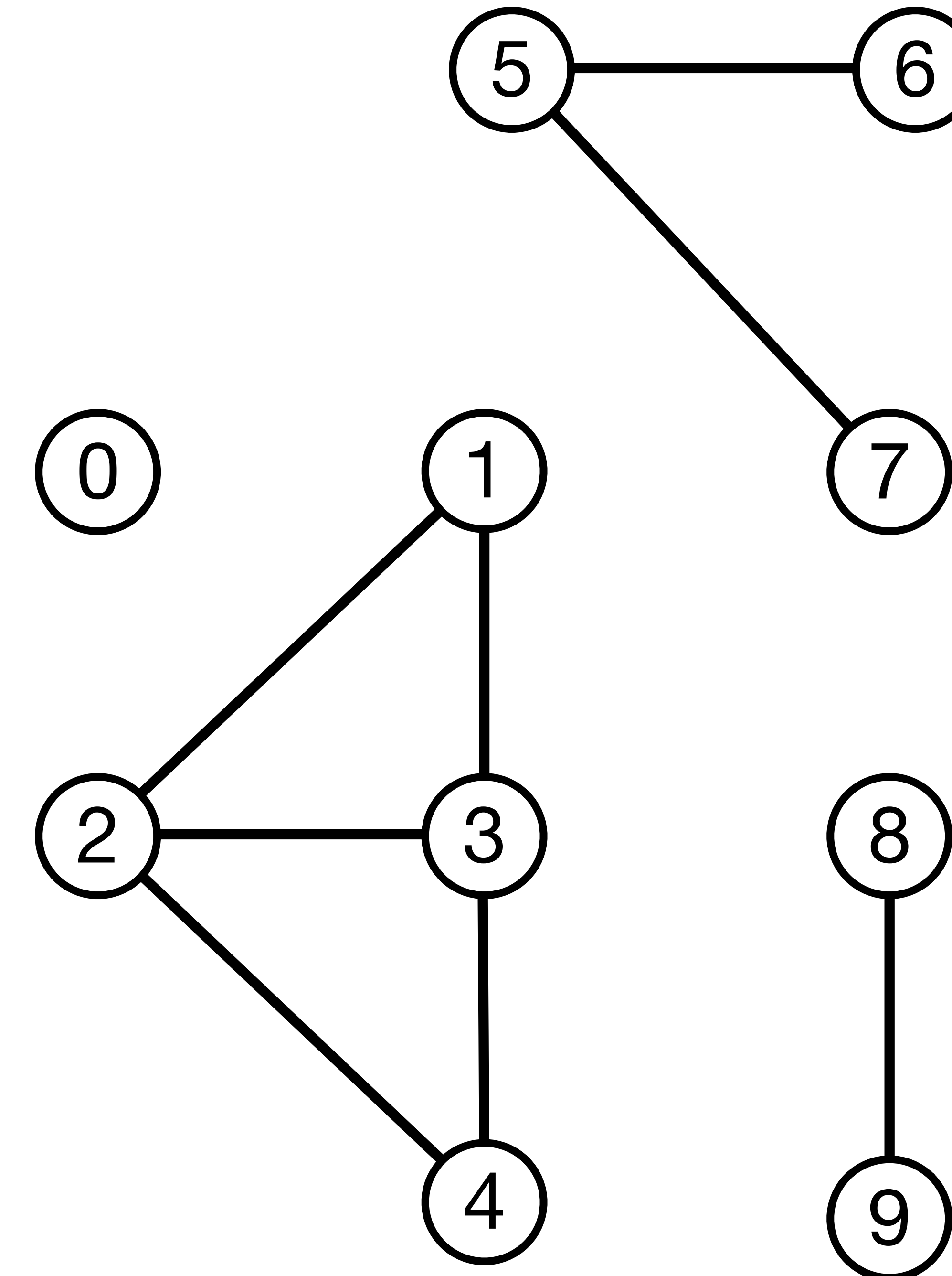
- sommet qui déconnecte une partie du graphe s'il disparait
- dans un graphe non dirigé

```
def comp_conn1 (g, v, visited) :
    visited[v] = True
    res = [v]
    for v in g.successors(v) :
        if not visited[v] :
            res += comp_conn1 (g, v, visited)
    return res

def comp_conn (g) :
    visited = {v: False for v in g.vertices}
    res = []
    for v in g.vertices :
        if not visited[v] :
            res += [comp_conn1 (g, v, visited)]
    return res
```

liste de listes

- graphe non dirigé

```
g3 = Graph ([i for i in range(10)])
add_uedges (g3, [(1,2),
  (3,1), (3,4), (4,2), (5,6),
  (7,5), (8,9)])

print (comp_conn (g3))
```

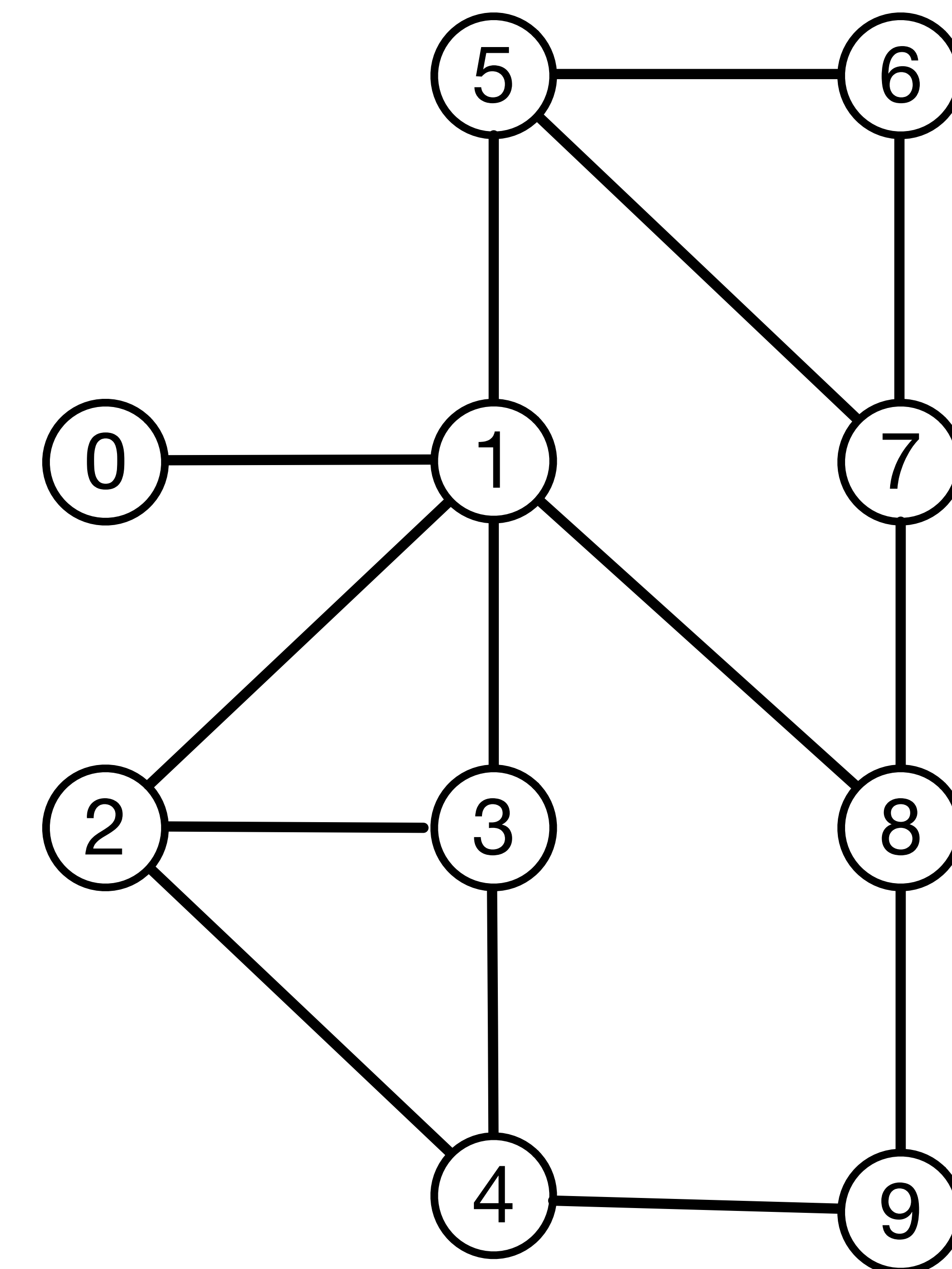


4 composantes connexes

Point d’articulation

- sommet qui déconnecte une partie du graphe s’il disparait
- dans un graphe non dirigé

```
def num_anchor (g, v, num, numOrdre, arts) :
    numOrdre += 1; num[v] = numOrdre
    res = num[v]
    for w in g.successors(v) :
        if num[w] == -1 :
            m = num_anchor (g, w, num, numOrdre, arts)
            if m >= num[v] :
                arts.add(v)
        else :
            m = num[w]
        res = min (res, m)
    return res

def articulations (g) :
    arts = set()
    numOrdre = -1; num = {v: -1 for v in g.vertices}
    for v in g.vertices :
        if num[v] == -1 :
            nFils = 0
            numOrdre += 1; num[v] = numOrdre
            for w in g.successors (v) :
                if num[w] == -1 :
                    nFils += 1
                    num_anchor (g, w, num, numOrdre, arts)
            if nFils > 1 :
                arts.add(v)
    return arts
```




```
print (articulations (g2))
```

```
>>> {1}
```

Biconnexité

- un graphe **non-orienté** sans point d'articulation est biconnexe
- dans un graphe biconnexe, il existe toujours 2 chemins totalement différents entre 2 sommets
- composantes **biconnexes**, triconnexes, etc… k-connexes

Exercice Donner un algorithme pour trouver les composantes biconnexes d'un graphe

Exercice Ecrire le programme correspondant

- dans un **graphe dirigé** fortement connexe, il existe toujours 2 chemins totalement différents entre 2 sommets
- composantes **fortement connexes**
  - `Tarjan` sait les calculer en temps linéaire sur une variation de `dfs` (ou de `articulations`)

Autre représentation des graphes

- on peut se passer des dictionnaires de Python
- pour un graphe dont les sommets sont entiers
- et ne considérer que des listes d'adjacences

```
class GraphInt :
    def __init__ (self, n) :
        self.vertices = [i for i in range(n)]
        self.succ = [[ ] for w in range(n)]

    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]
```

Autre représentation des graphes

- on peut aussi considérer une matrice d'adjacence au lieu de listes d'adjacence

```
class GraphIntMat :
    def __init__ (self, n) :
        self.vertices = [i for i in range(n)]
        self.adj = [[0 for j in range(n)] for i in range(n)]

    def __str__ (self) :
        res = 'sommets = {}\n'.format (self.vertices)
        res += 'adjacence = {}'.format (self.adj)
        return res

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

    def add_edge (self, v, w) :
        self.adj[v][w] = 1

    def successors (self, v) :
        n = self.order()
        return [w for w in range(n) if self.adj[v][w] != 0]
```

- algorithme de `Warshall` pour la fermeture transitive en $O(n^3)$

- algorithme de `Warshall` pour les plus courts chemins entre tous les sommets

Prochain cours

- analyse lexicale
- analyse syntaxique (méthode récursive descendante)
- évaluation d’arbres de syntaxe abstraite (ASA)