Algorithmes, Programmation, IA

Cours 4

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

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

Plan

- rappels
- piles, files d’attente, files d’attente avec priorités
- parcours de graphe
- recherche de chemins
- plus court chemin

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

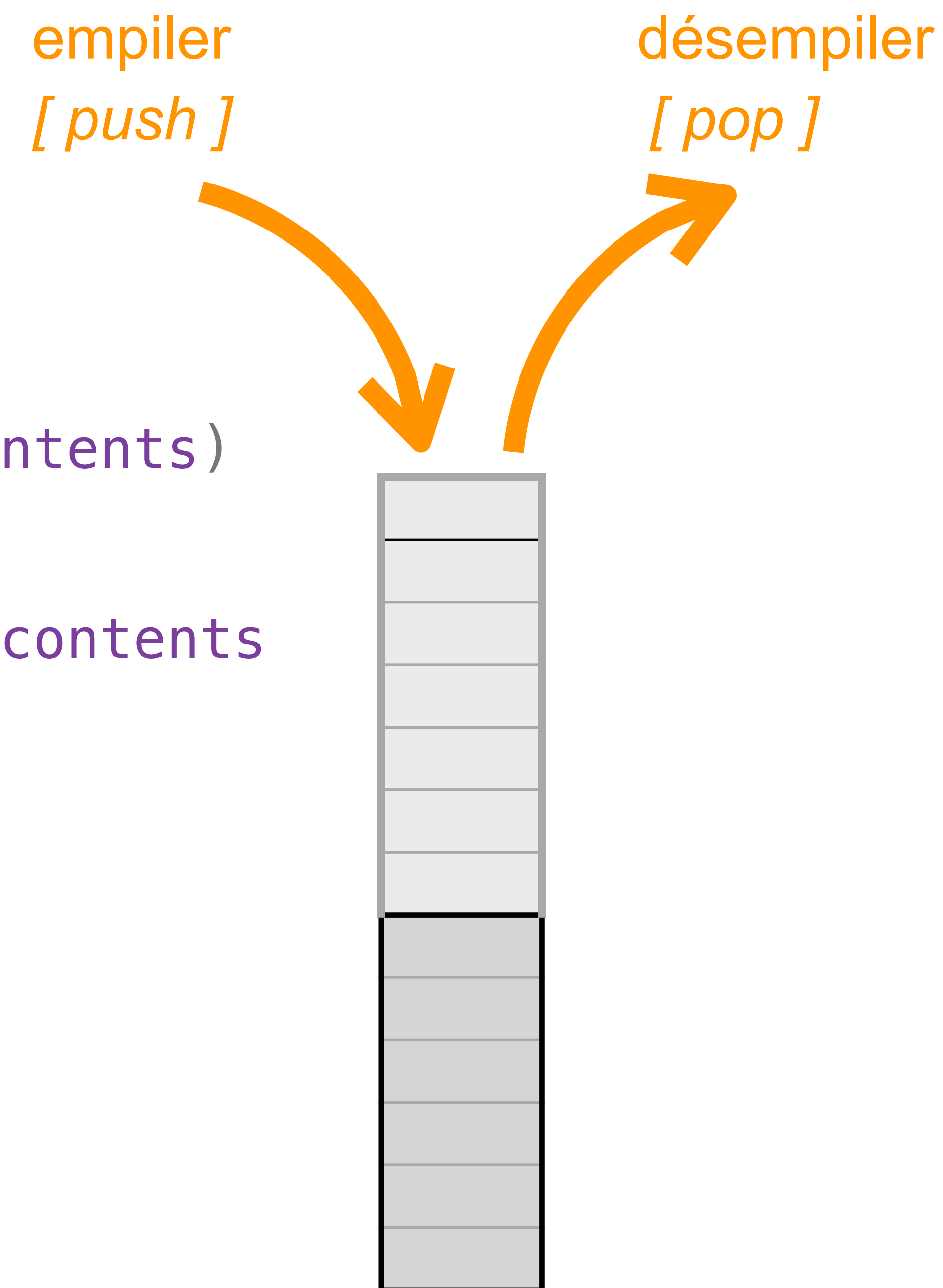
Piles et files d’attente

- classe des piles (LIFO)

```
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 == [ ]
```

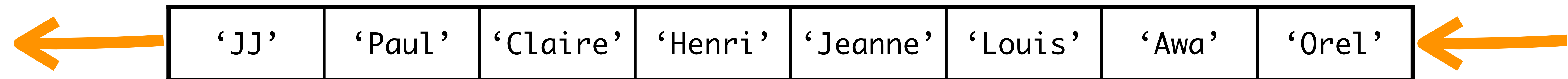
Last In First Out

opérations des piles en O(n)



- classe des files d’attente (FIFO)

```
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 == [ ]
```



First In First Out

opérations des files en O(n) amorti
avec des doubles-listes

Files de priorité

- file de priorité

```
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 = self.contents.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 == [ ]
```

enlever le plus petit
[deq]

20	2	5	7	13	24
‘JJ’	‘Paul’	‘Claire’	‘Henri’	‘Jeanne’	‘Louis’

ajouter
[enq]

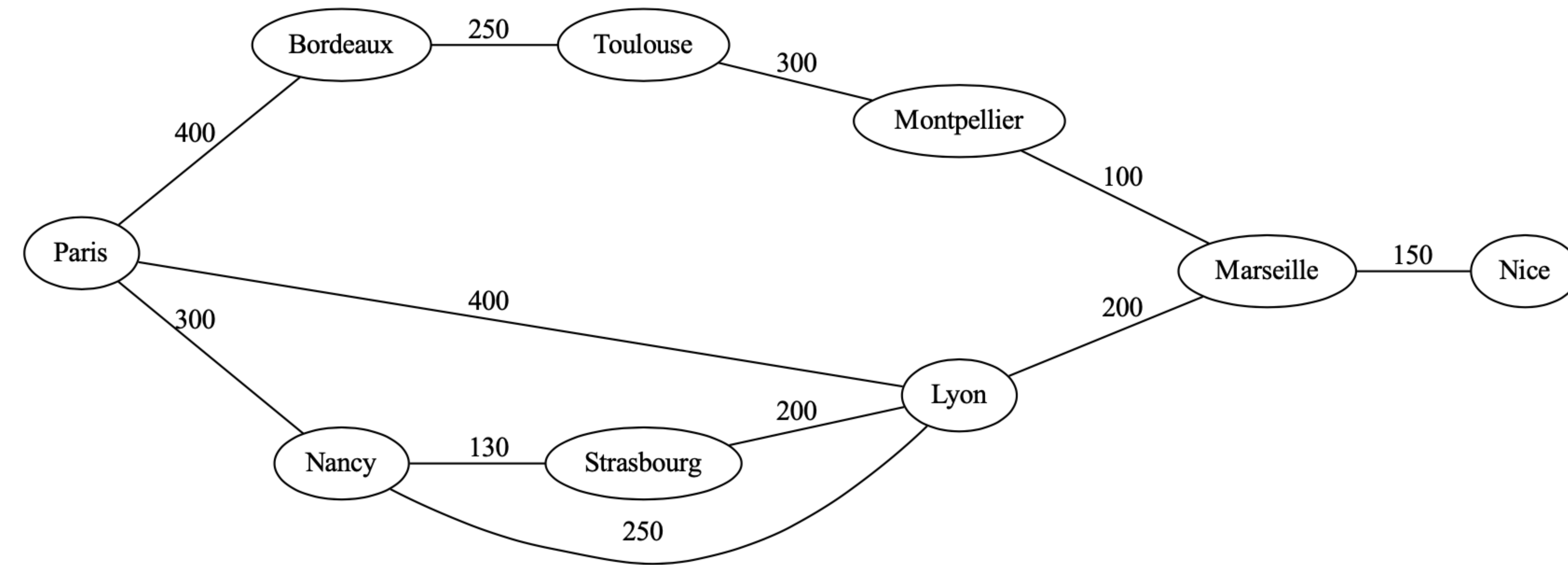
```
fp = PQueue()
for x in [(20, 'JJ'), (2, 'Paul'), (5, 'Claire'), (13, 'Robert')] :
    fp.enq(x)
print (fp)
(n, s) = fp.deq()
print (n, s, fp)
```

opérations des files de priorité en O(n (log n))
avec des tas

Représentation des graphes

- on définit une classe pour les graphes

```
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]
```



- et on construit un graphe (sans les distances)

```
g = Graph (['paris', 'lyon', 'marseille', 'toulouse', ‘nancy’, 'nice'])
g.add_edge ('paris', 'lyon')
g.add_edge ('lyon', ‘nancy')
. . .
```

Représentation des graphes

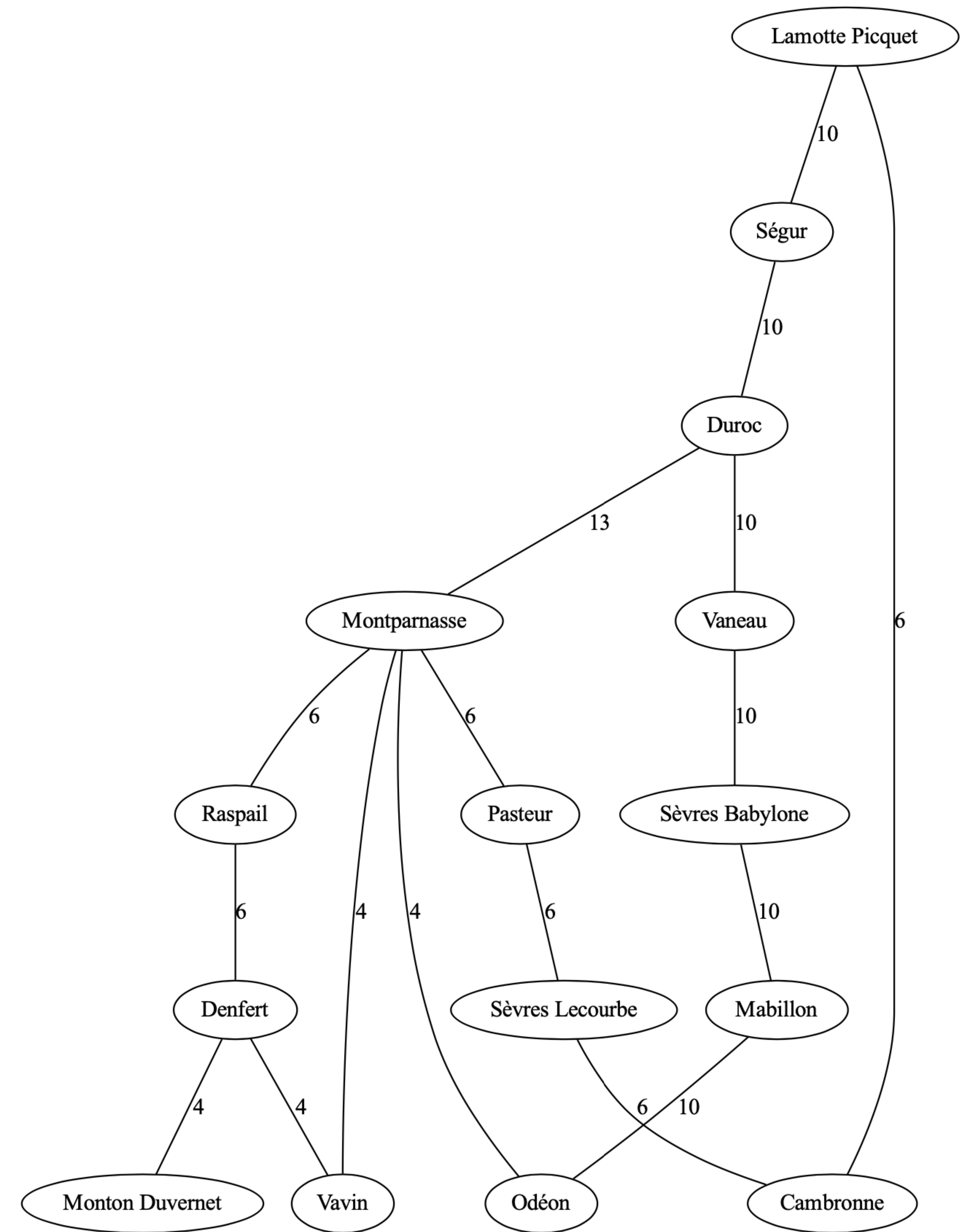
- on construit le graphe du métro parisien

```
stations = ['lamotte-picquet', 'segur', 'duroc', 'montparnasse' ]
metro = Graph (stations)
metro.add_edge ('segur', 'duroc')
metro.add_edge ('lamotte-picquet', 'segur')
metro.add_edge ('segur', 'duroc')
```

`METRO-PARIS`

fichier texte qui contient les lignes de métro parisien

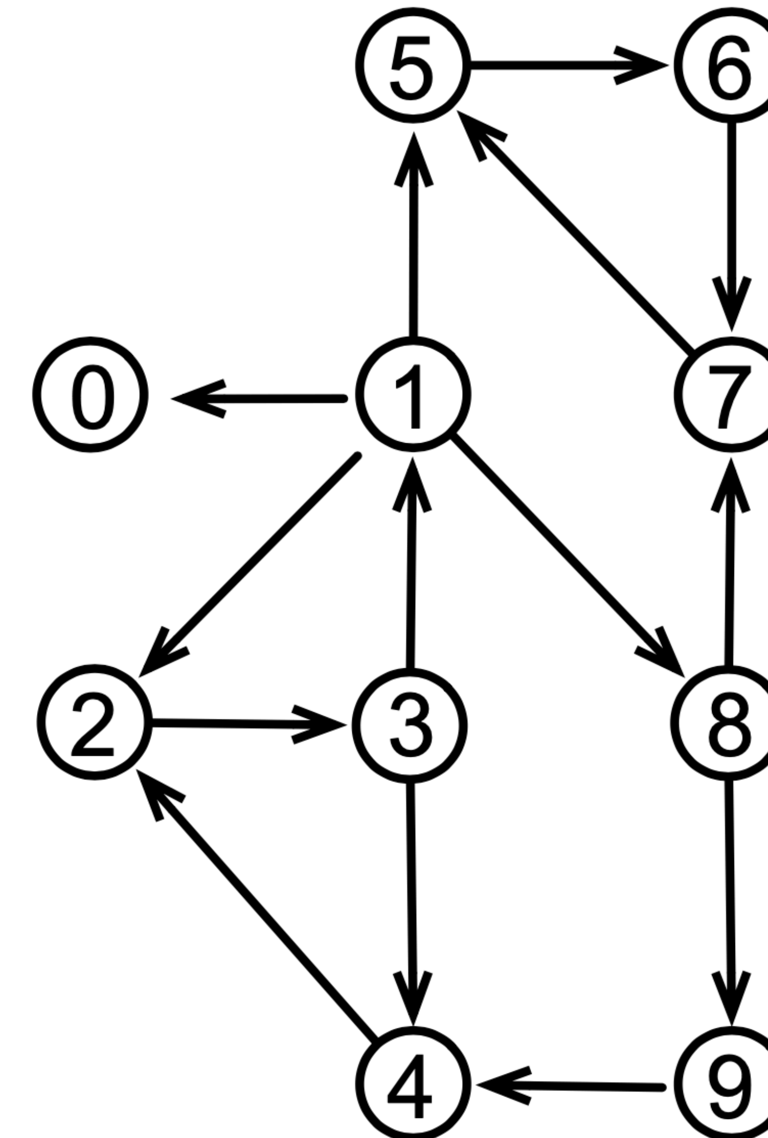
`METRO-PARIS via GraphViz`



Représentation des graphes

• un graphe plus abstrait

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


g2 = Graph ([i for i in range(10)])
add_edges (g2, [(1,0), (1,2), (1,5), (1,8), (2,3),
  (3,1), (3,4), (4,2), (5,6), (6,7),
  (7,5), (8,7), (8,9), (9,4)])
```



• quelques remarques sur la représentation :

- utilisation des dictionnaires : représentation plus concise

- utilisation des dictionnaires : représentation plus polymorphe

- autre représentation : objets pour les sommets avec attributs nom et `successors`

- autre représentation : sommets sont des nombres entiers et tableau de listes de successeurs

Exercice Ecrire la représentation pour ces deux dernières représentations.

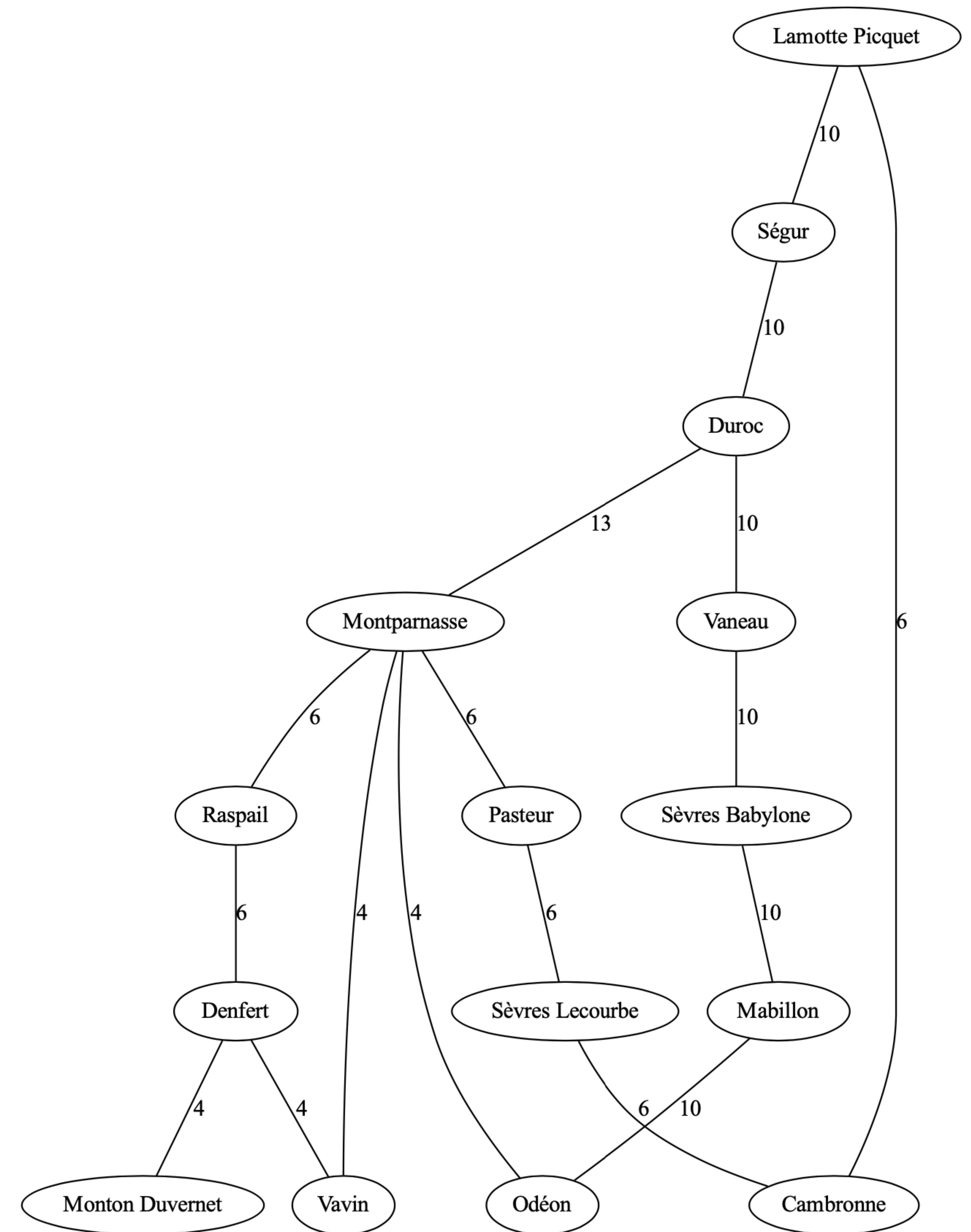
Représentation des graphes

- on définit une sous-classe pour les graphes valués

```
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)]
```

- et on construit un graphe (sans les distances)

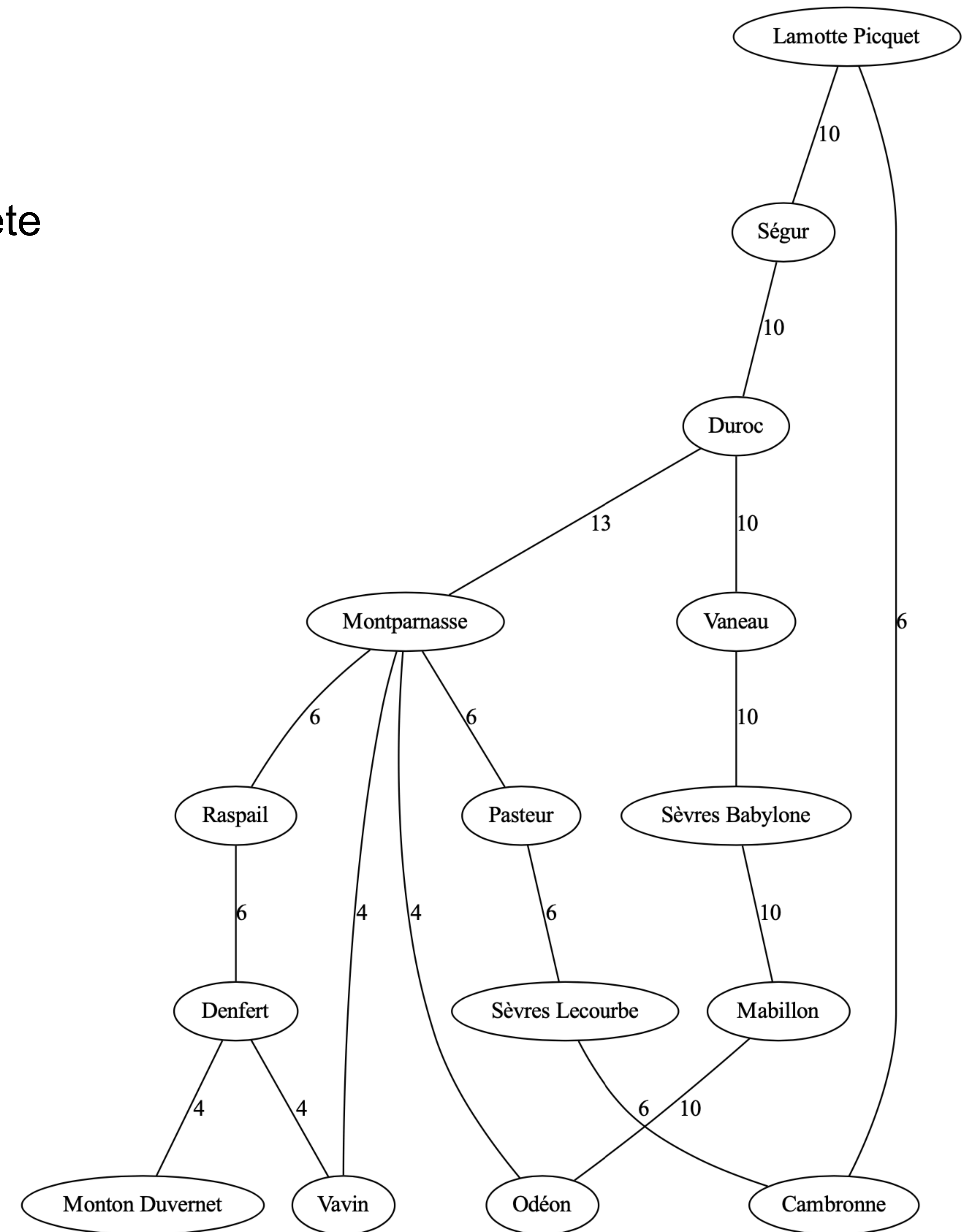
```
metro = VGraph (stations)
metro.add_edge ('segur', 'duroc', 10)
metro.add_edge ('lamotte-picquet', 'segur', 10)
metro.add_edge ('segur', 'duroc', 10)
print (metro)
```



Représentation des graphes

- un graphe non dirigé a des arrêtes **bi-directionnelles**
- si (u, v) est une arrête d'un graphe non dirigé, alors (v, u) est aussi une arrête

```
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)
```



Représentation des graphes

- fichier mes_graphes.py

```
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)]
```

```
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)
```

```
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 == [ ]
```

```
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 == [ ]
```

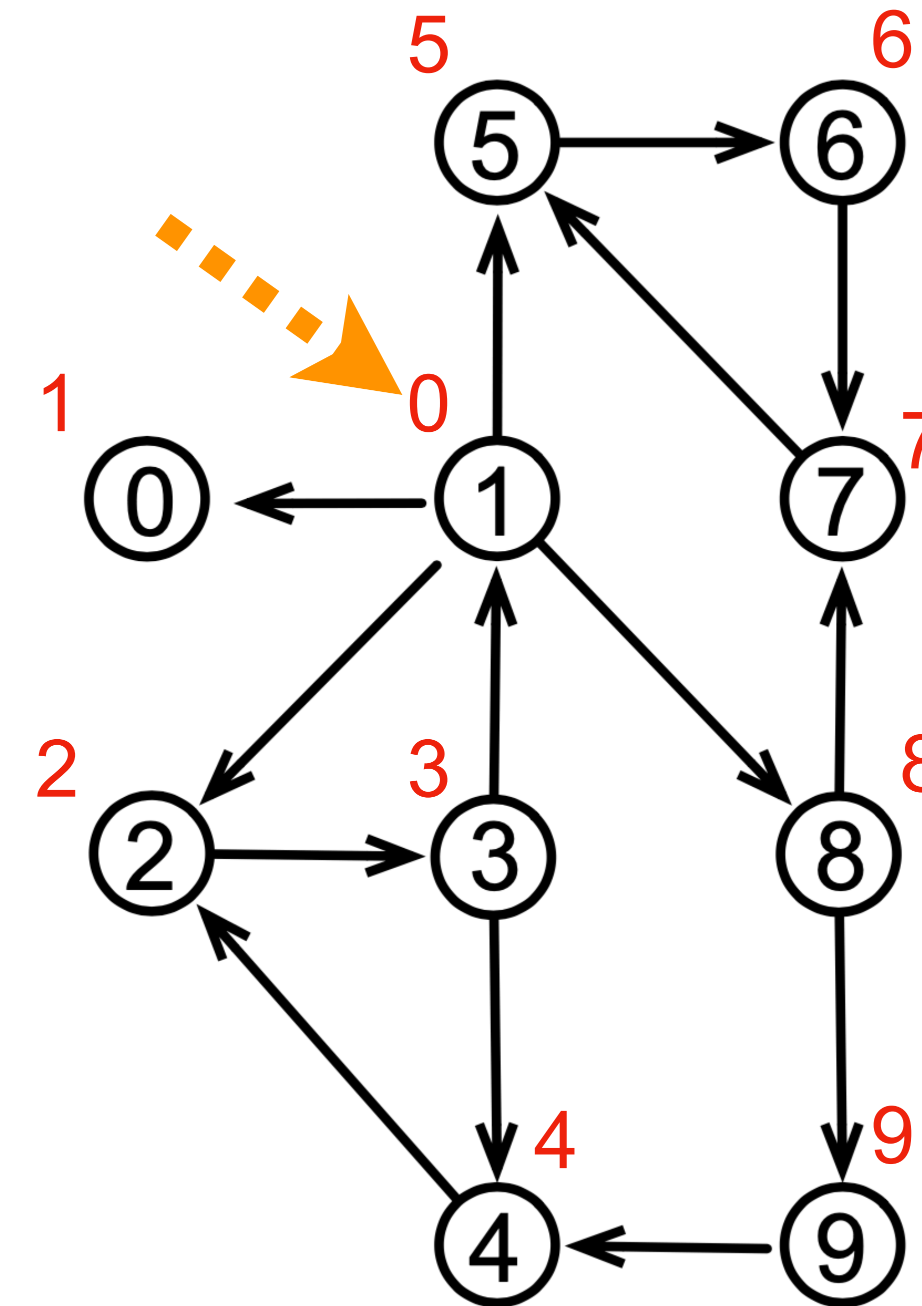
Parcours de graphe en profondeur d'abord

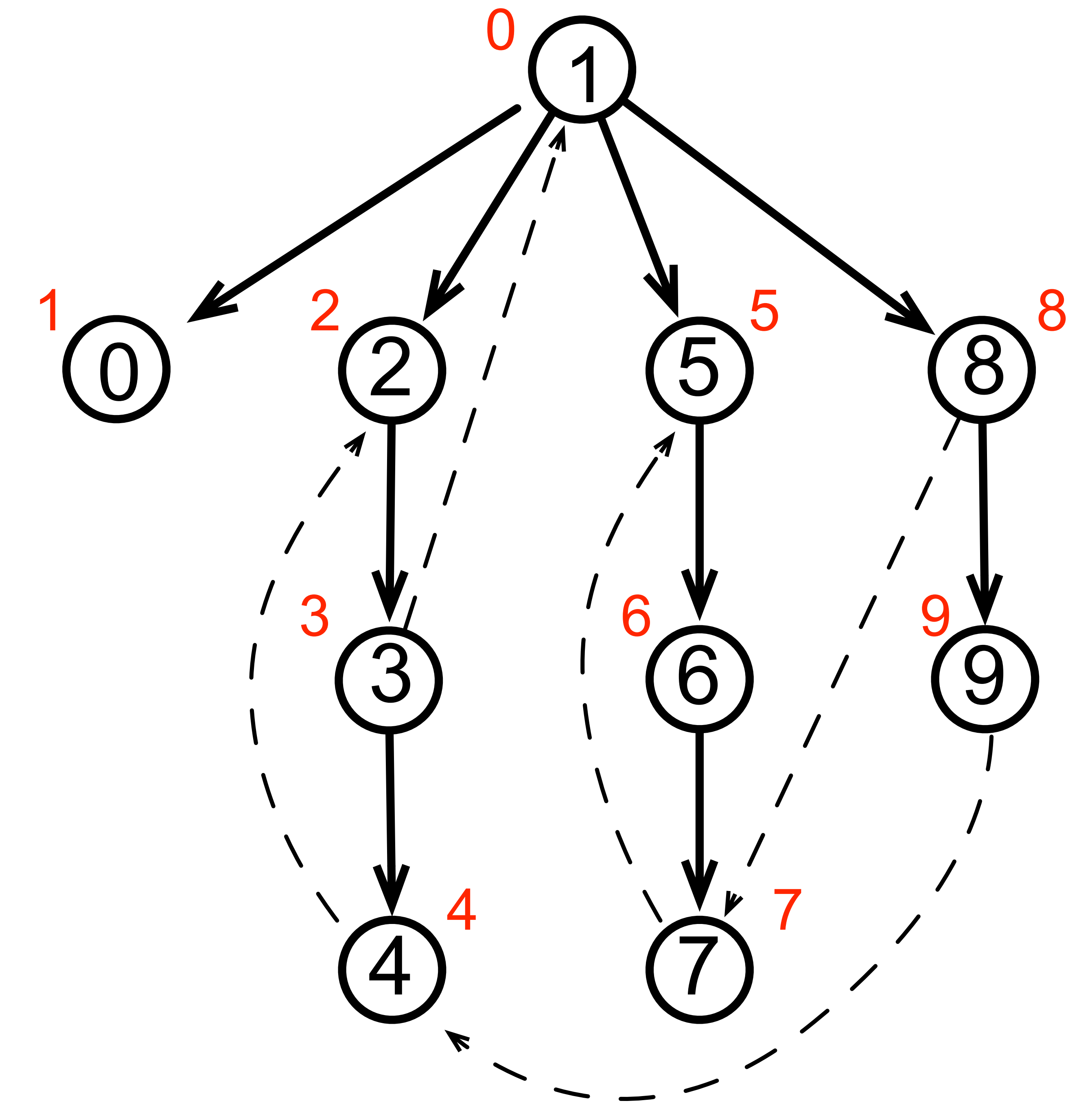
- *depth first search*

```
def dfs1 (g, v, visited) :
    visited[v] = True
    print (v, end =' ')
    for w in g.successors(v) :
        if not visited[w] :
            dfs1 (g, w, visited)

def dfs (g) :
    visited = {v: False for v in g.vertices}
    for v in g.vertices :
        if not visited[v] :
           dfs1 (g, v, visited)
```

dfs en *O(V + E)*

et donc linéaire en nombre de sommets et d'arrêtes





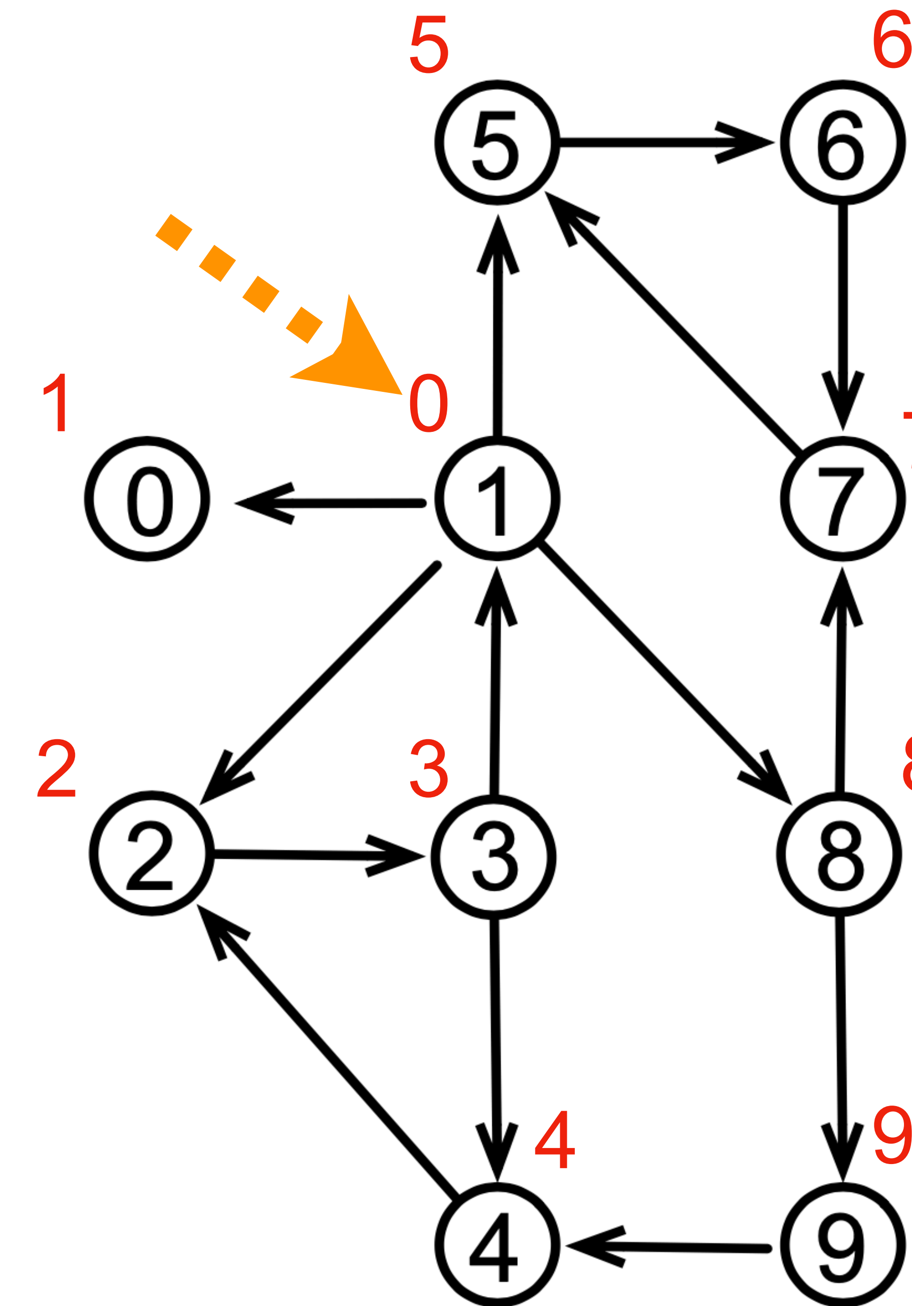
Arbre de recouvrement

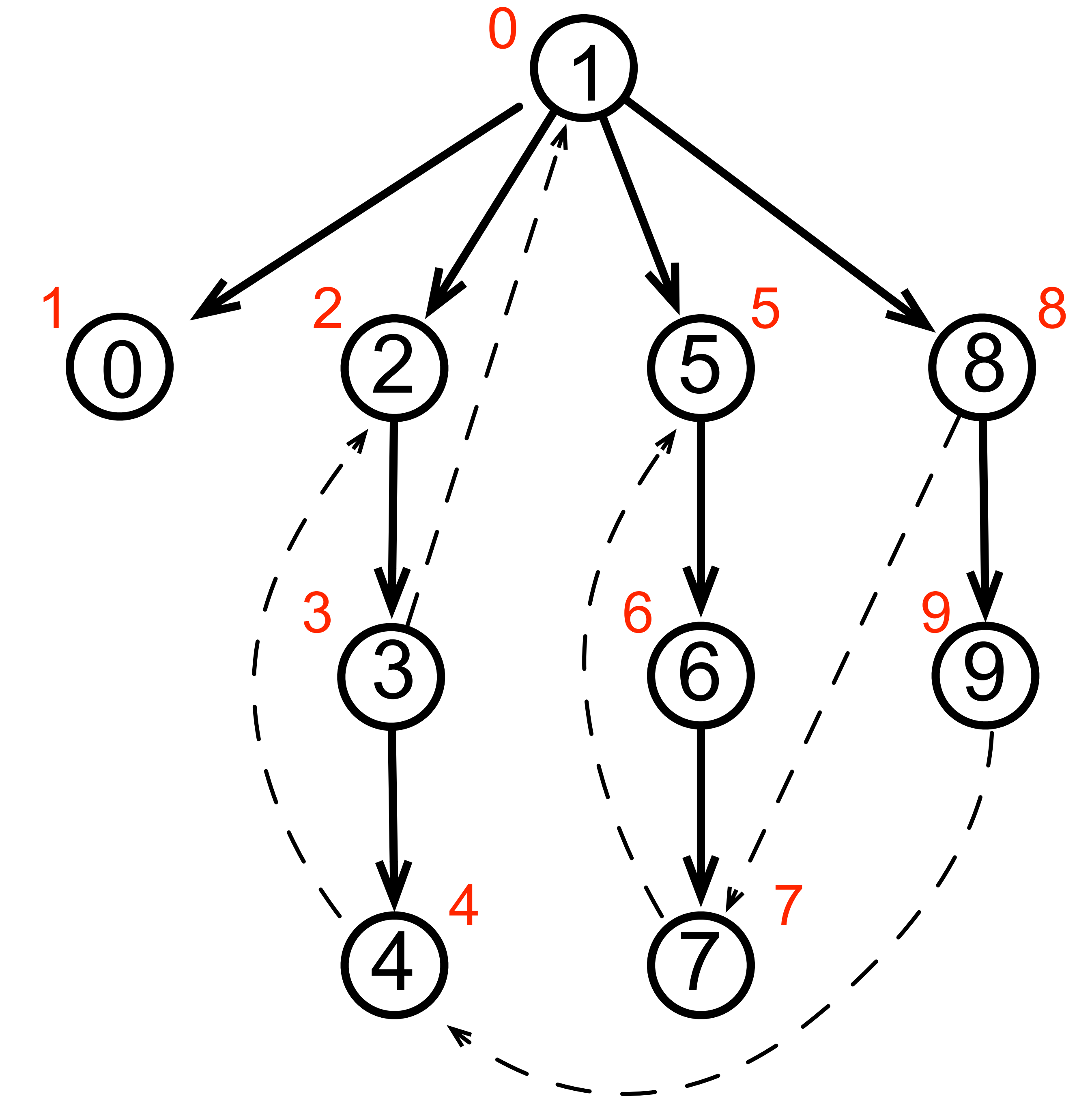
[*spanning tree*]

Parcours de graphe en profondeur d’abord

- *depth first search*

```
def dfs1 (g, v, visited) :
    visited[v] = True
    print (v, end =' ')
    dfs_vs (g, g.successors(v), visited)

def dfs_vs (g, vs, visited) :
    for v in vs :
        if not visited[v] :
            dfs1 (g, v, visited)

def dfs (g) :
    visited = {v: False for v in g.vertices}
    dfs_vs (g, g.vertices, visited)
```





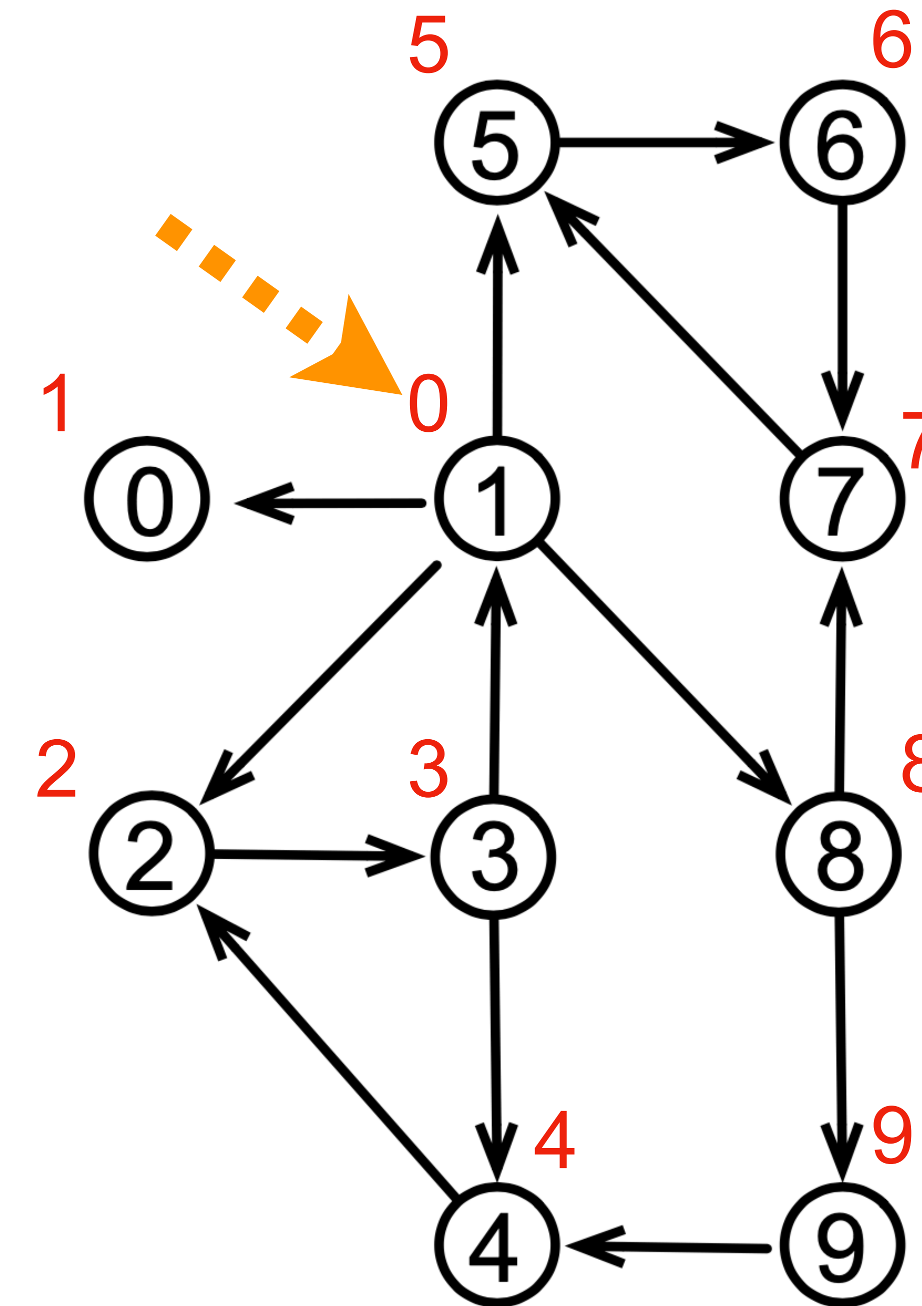
Arbre de recouvrement

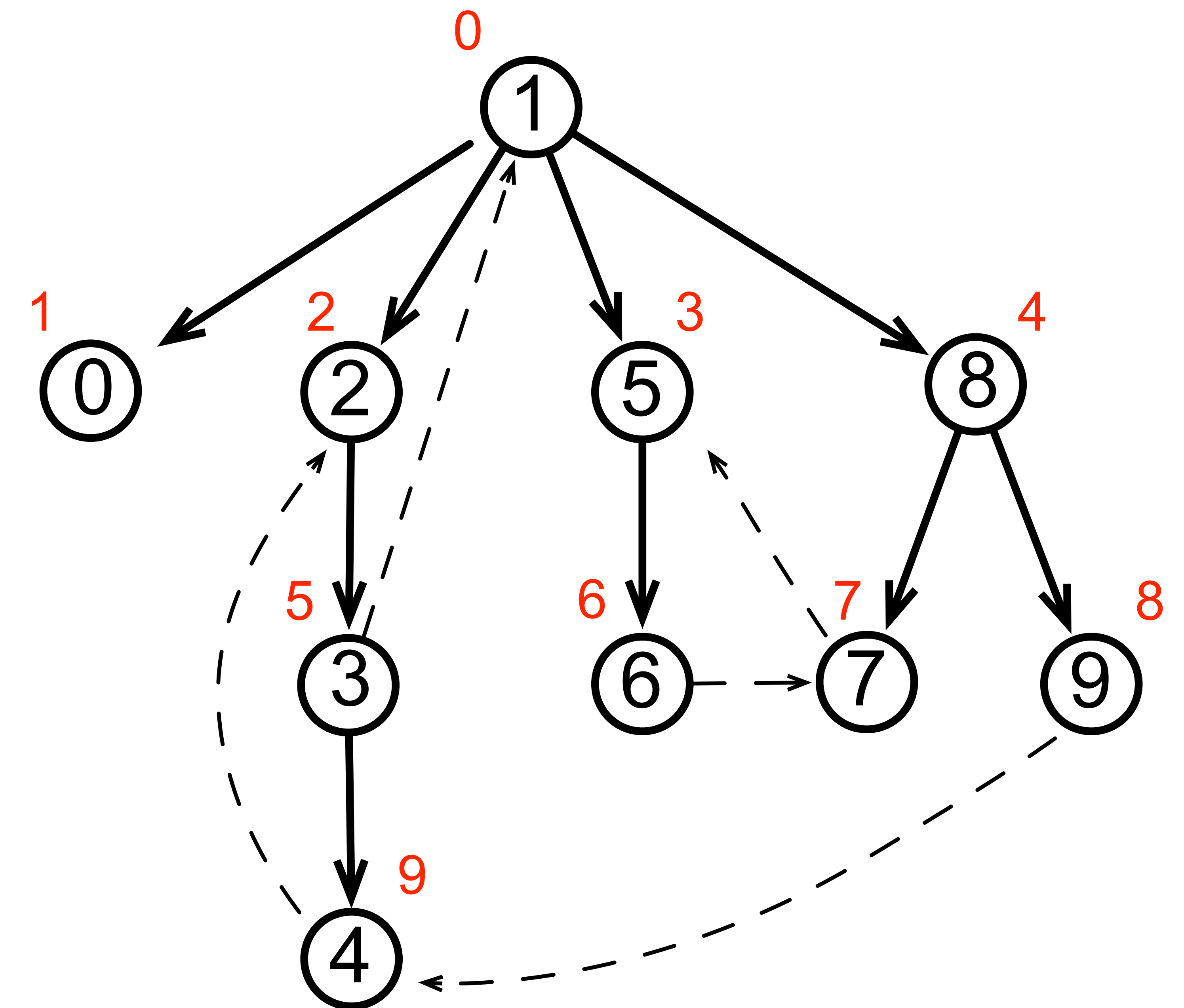
[*spanning tree*]

Parcours de graphe en largeur d'abord

- *breadth first search*

```
def bfs1 (g, visited, f) :
    while not f.is_empty() :
        v = f.deq()
        print (v, end=' ')
        for w in g.successors(v) :
            if not visited[w] :
                f.enq(w); visited[w] = True

def bfs (g) :
    visited = {v: False for v in g.vertices}
    f = Queue()
    for v in g.vertices :
        if not visited[v] :
            f.enq(v); visited[v] = True
            bfs1 (g, visited, f)
```





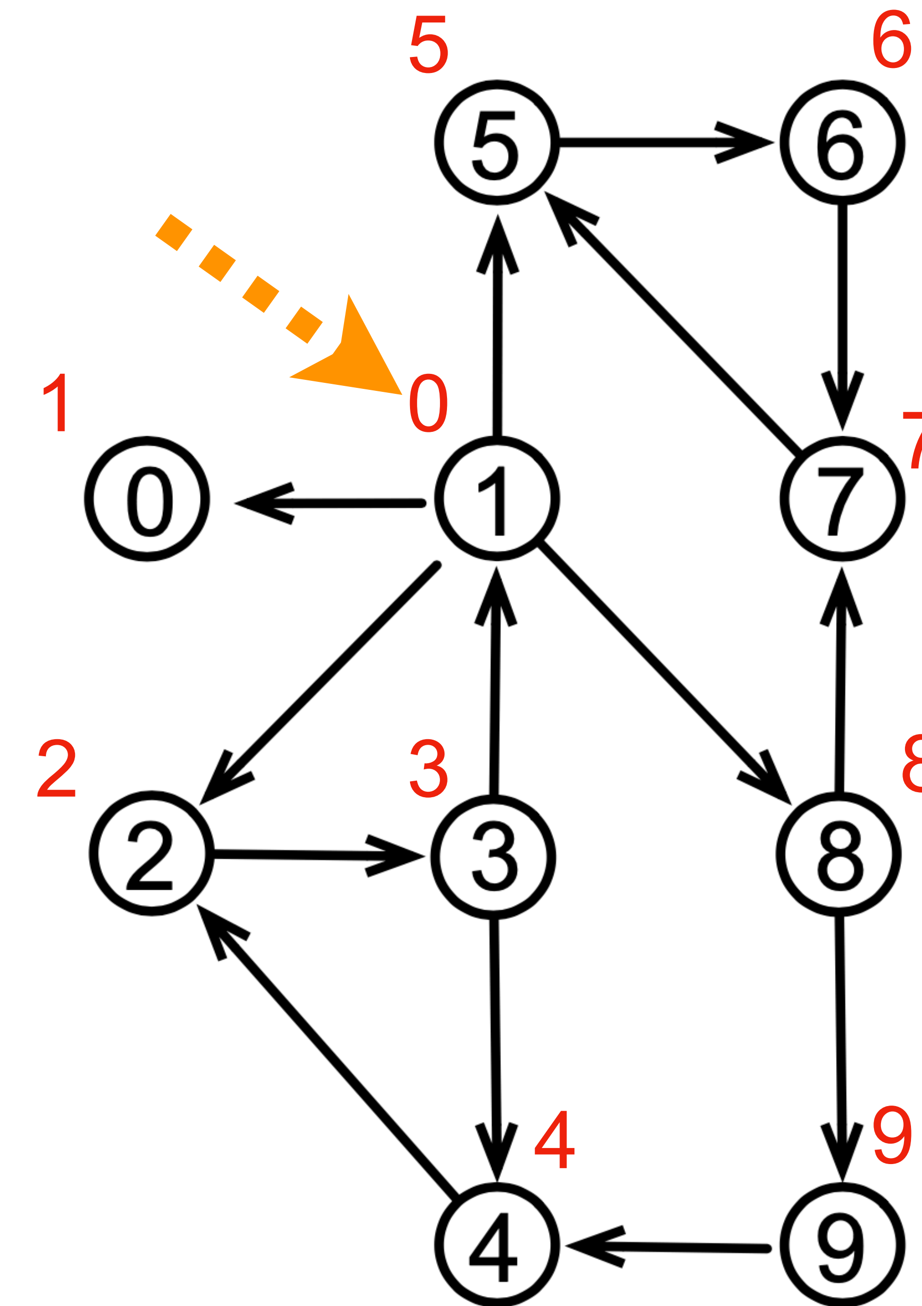
Arbre de recouvrement

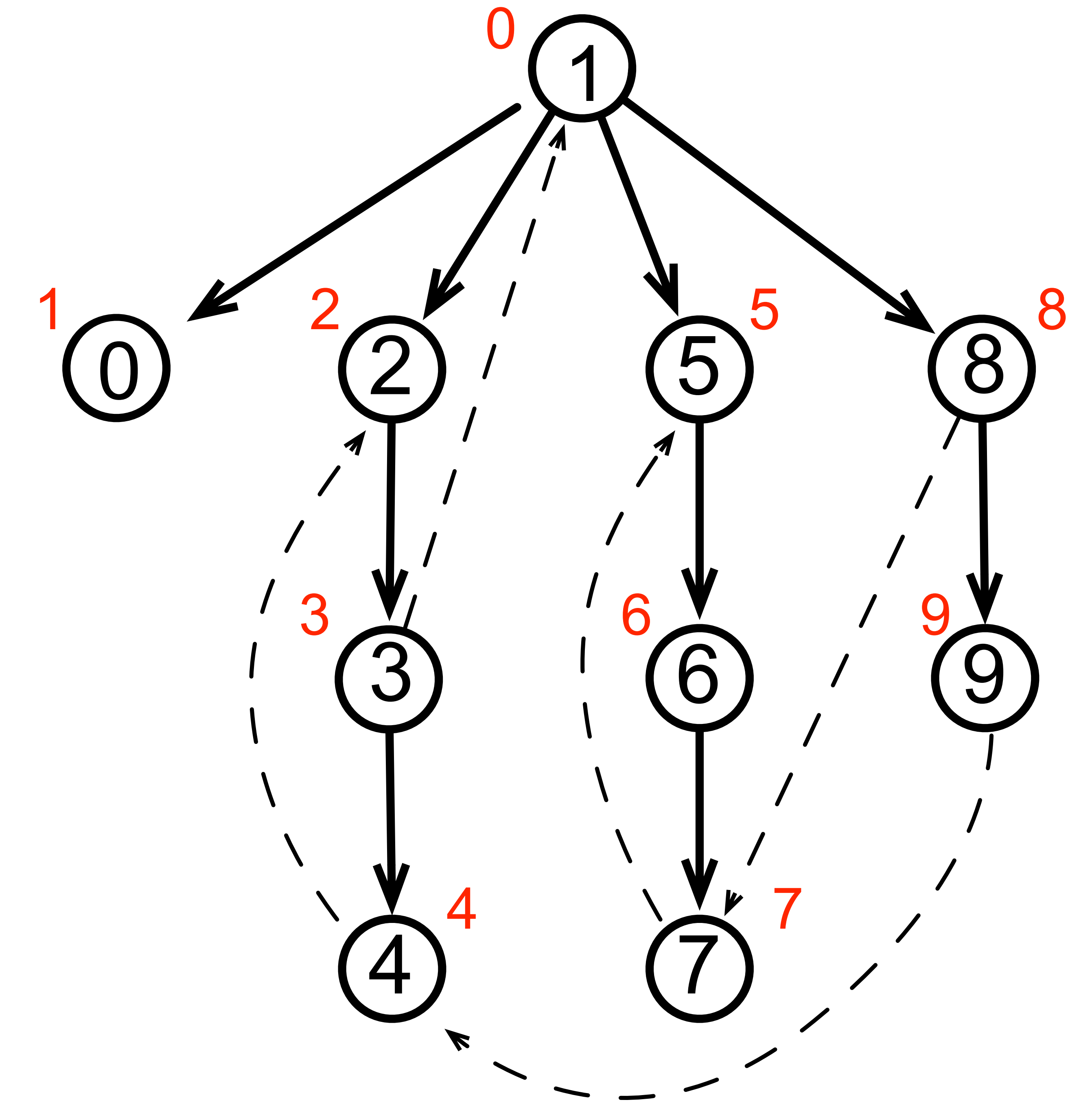
[spanning tree]

Parcours de graphe en profondeur d'abord

- *depth first search* (itératif)

```
def dfs1 (g, visited, p) :
    while not p.is_empty() :
        v = p.pop()
        print (v, end=' ')
        for w in g.successors(v) :
            if not visited[w] :
                p.push(w); visited[w] = True

def dfs (g) :
    visited = {v: False for v in g.vertices}
    p = Stack()
    for v in g.vertices :
        if not visited[v] :
            p.push(v); visited[v] = True
            dfs1 (g, visited, p)
```



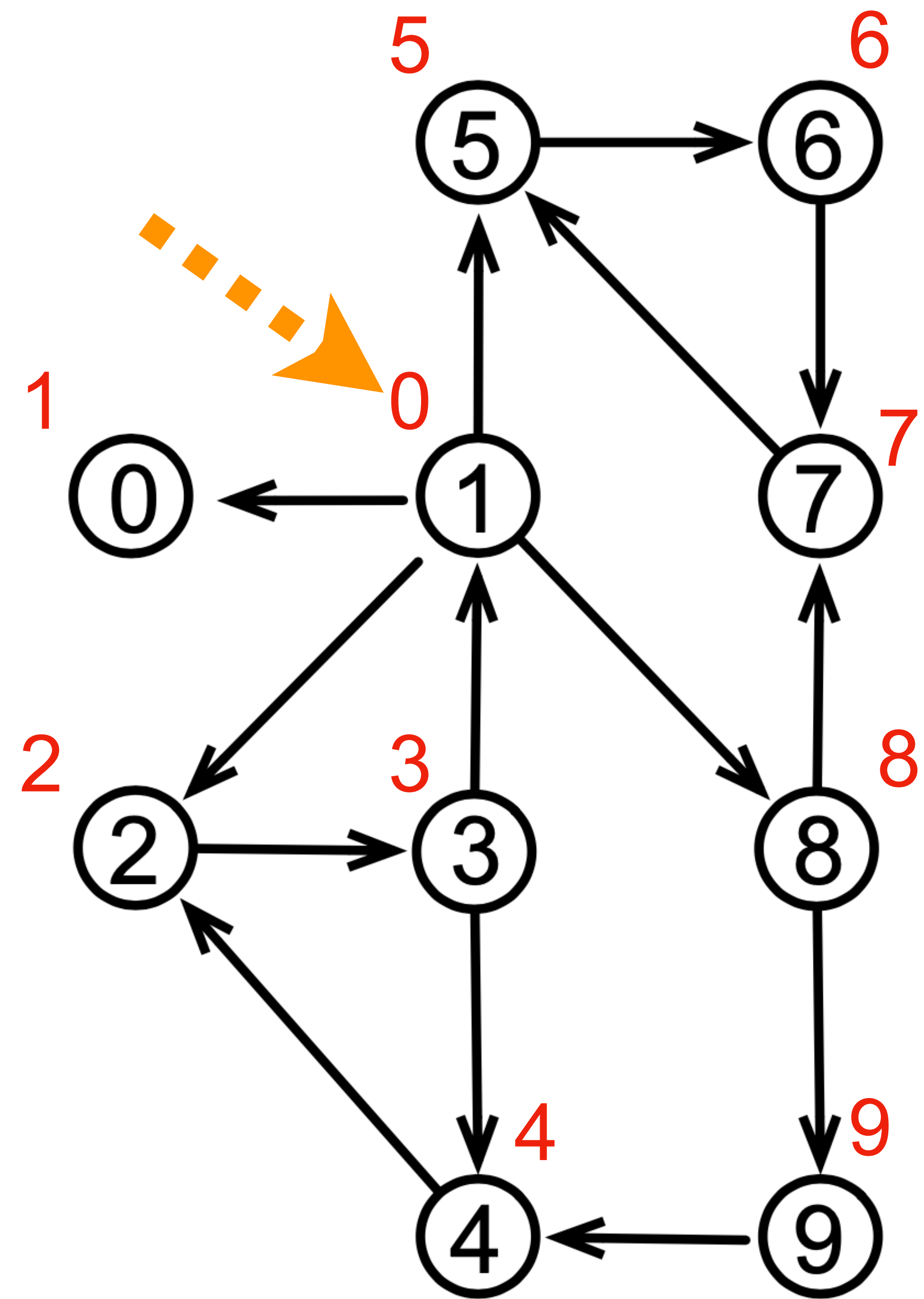


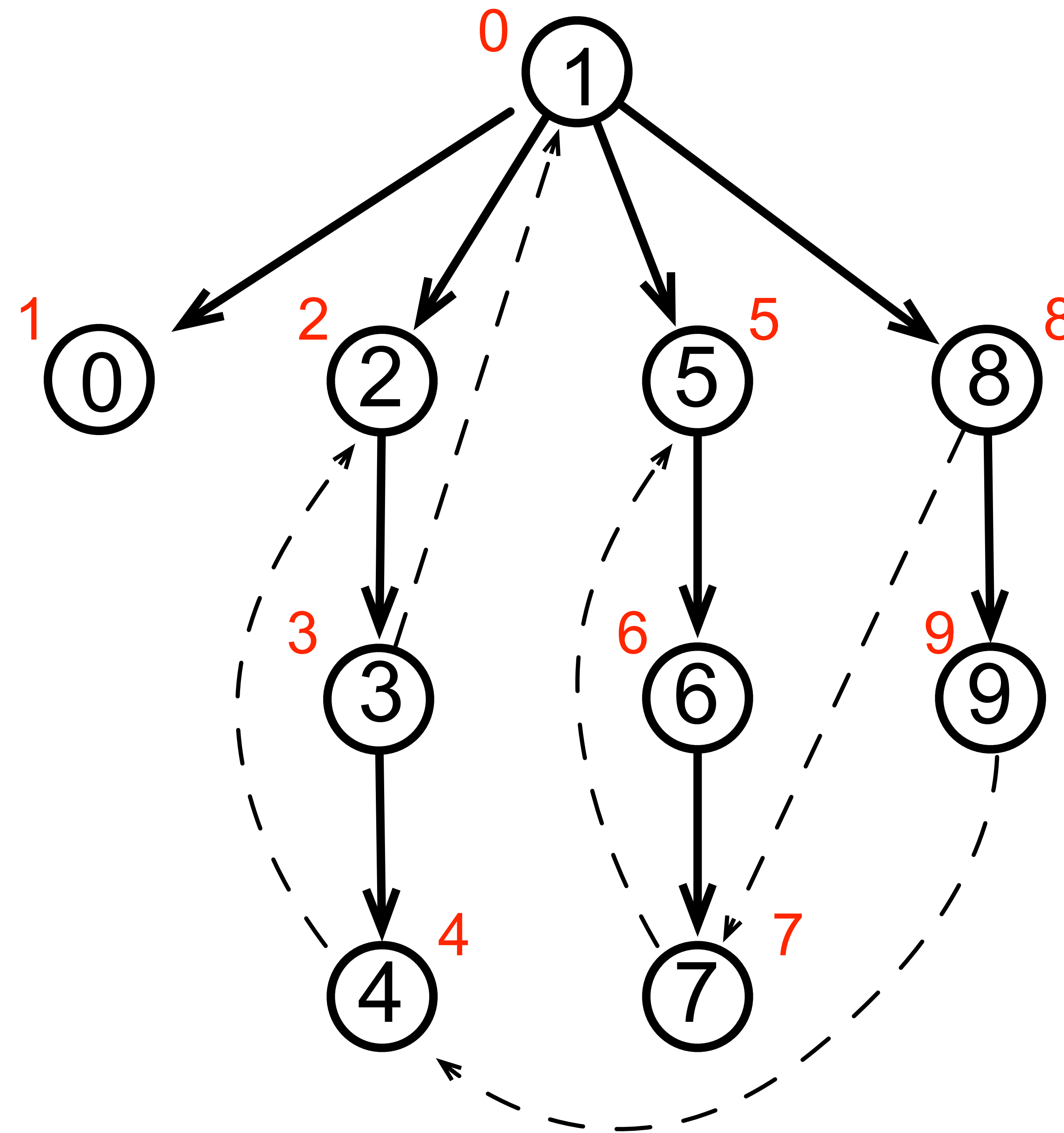
Arbre de recouvrement

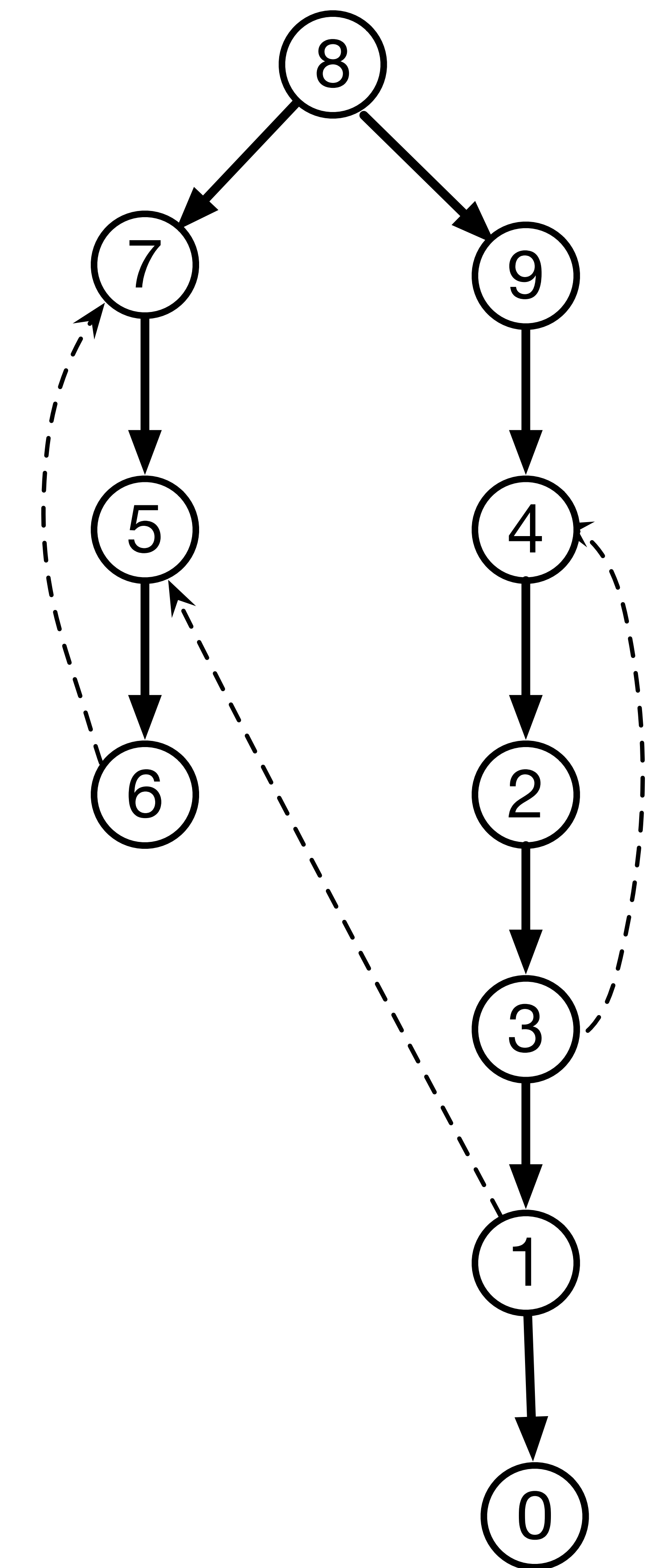
[spanning tree]

Parcours de graphe en profondeur d'abord

- *depth first search:* l'arbre de recouvrement dépend du sommet de départ







Arbres de recouvrement

[*spanning trees*]

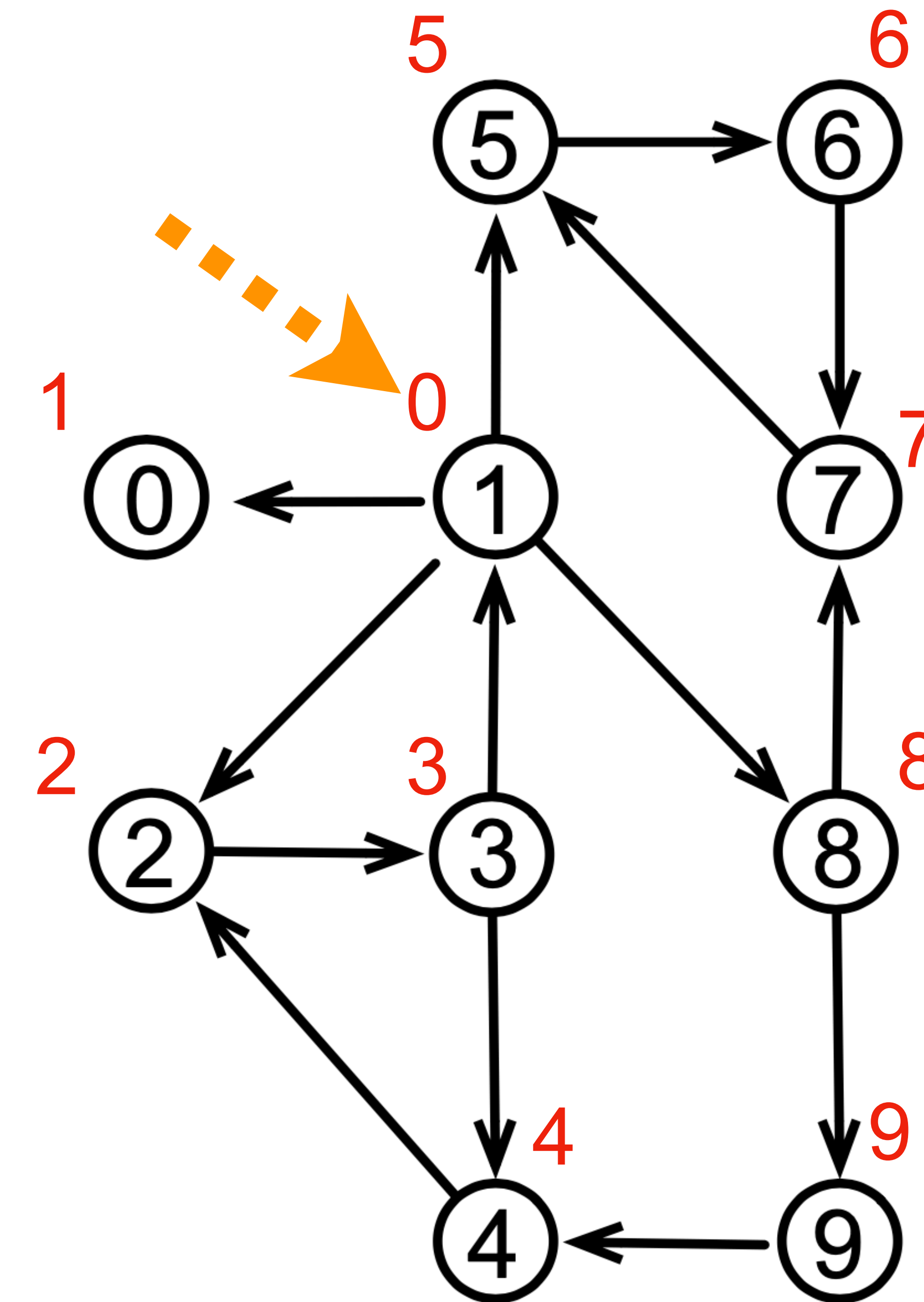
Quelques remarques

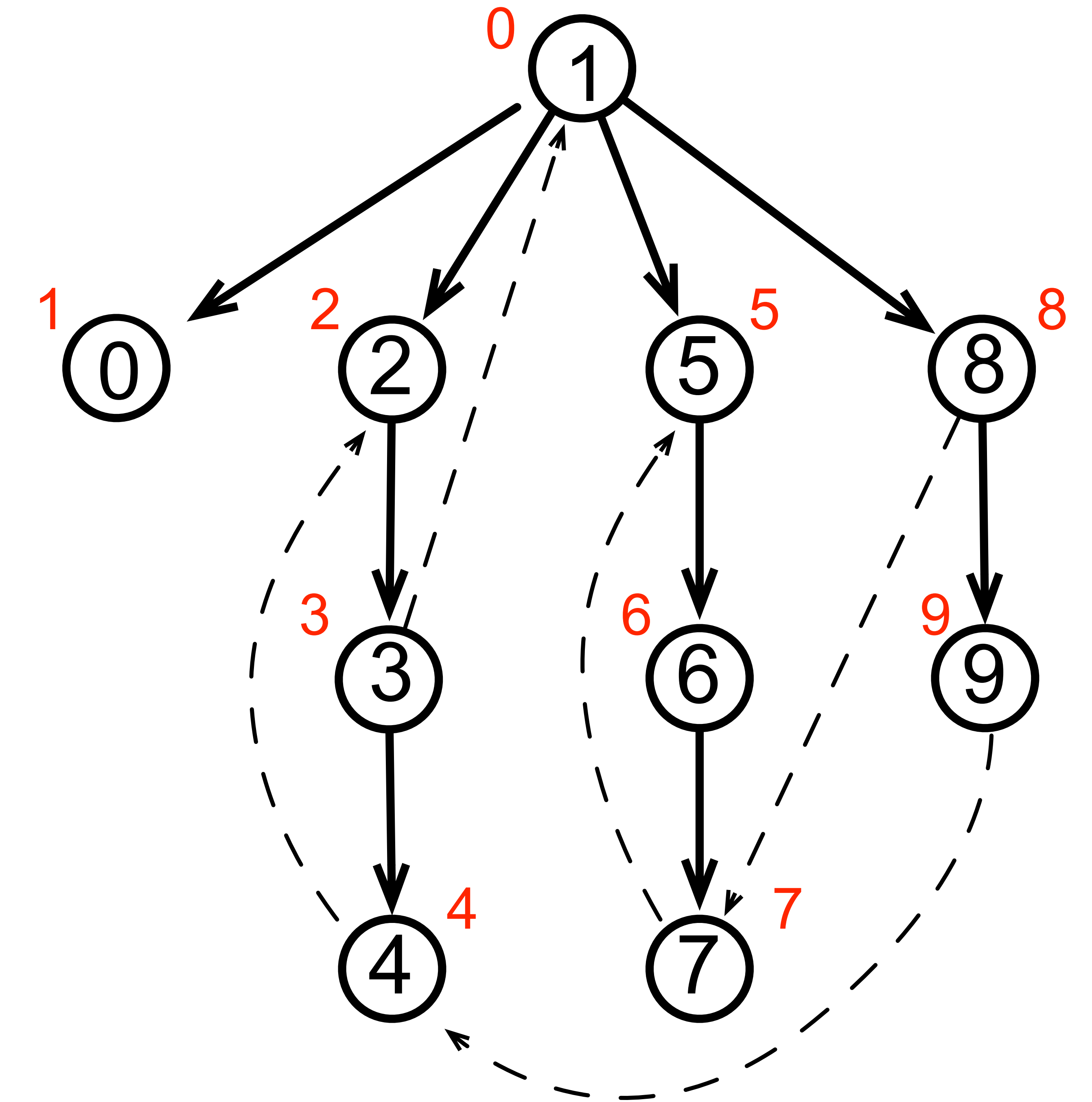
- le parcours en **profondeur** d’abord s’écrit élégamment avec la **récursivité**
- le parcours en **largeur** d’abord s’écrit qu’avec des itérations et une **file d’attente**
- il faut un drapeau *visited* pour éviter de boucler
- le parcours en profondeur d’abord est naturellement **linéaire** en nombre d’opérations
  [on ne passe qu’une seule fois sur chaque arrête]
- le parcours en profondeur d’abord induit beaucoup d’algorithmes `[Robert Tarjan]`

Recherche de chemin

- recherche de chemin avec *dfs*

```
def dfs_path1 (g, v, w, visited) :
    visited [v] = True
    if v == w :
        return [v]
    else :
        for x in g.successors(v) :
            if not visited[x] :
                ch = dfs_path1 (g, x, w, visited)
                if ch != [ ] :
                    return [v] + ch
        return [ ]

def dfs_path (g, v, w) :
    return dfs_path1 (g, v, w,
                      {v:False for v in g.vertices})
```





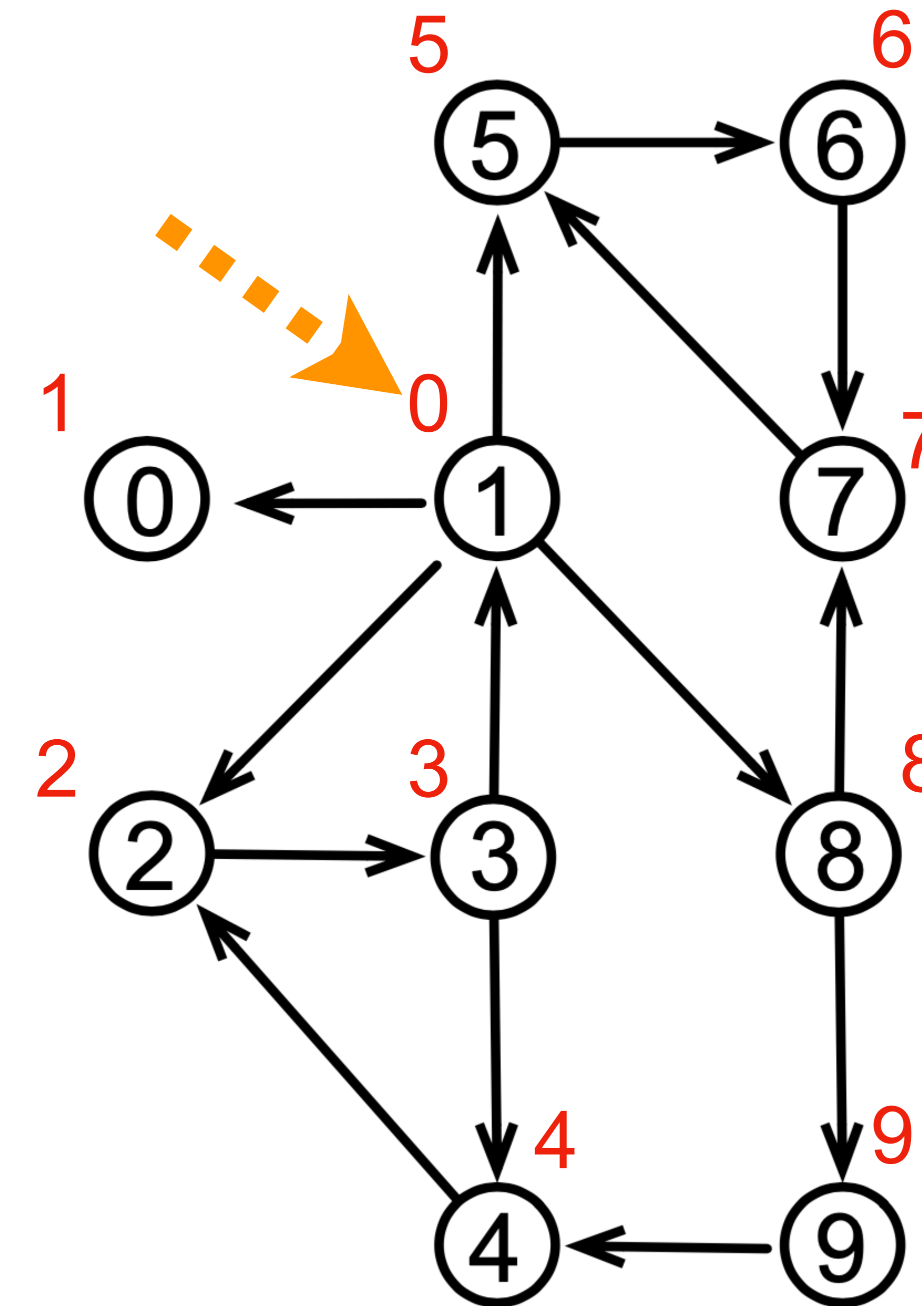
Arbre de recouvrement

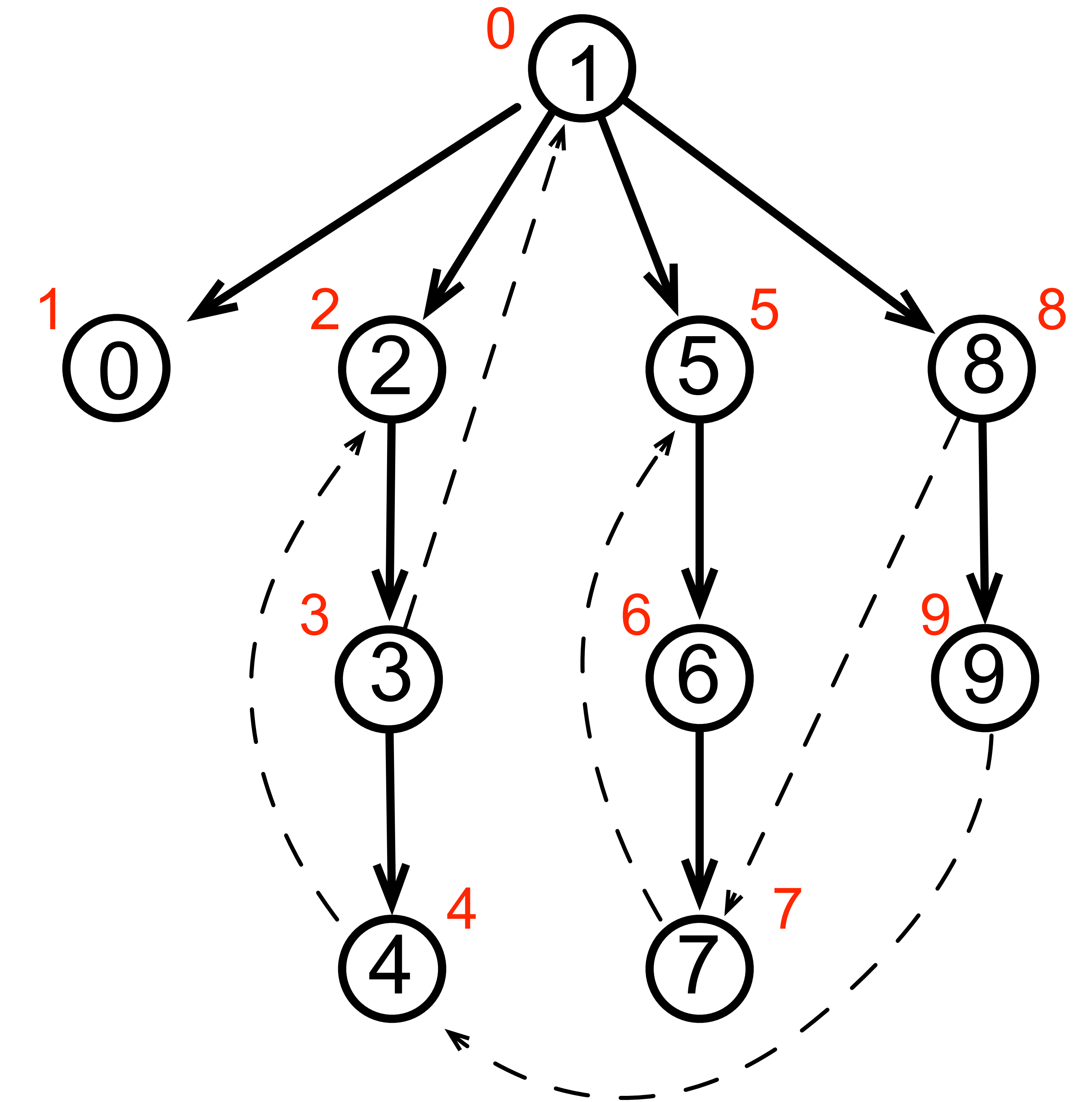
[*spanning tree*]

Recherche de chemin

- recherche de chemin avec *bfs*

```
def bfs_path1 (g, w, visited, f) :
    pred = {x:[ ] for x in g.vertices }
    while not f.is_empty() :
        v = f.deq()
        if v == w :
            return pred[v] + [v]
        else :
            for x in g.successors(v) :
                if not visited[x] :
                    f.enq(x); visited[x] = True
                    pred[x] = pred[v] + [v]
    return [ ]

def bfs_path (g, v, w) :
    visited = {x: False for x in g.vertices}
    f = Queue()
    f.enq(v); visited[v] = True
    return bfs_path1 (g, w, visited, f)
```





Arbre de recouvrement

[*spanning tree*]

Recherche du plus court chemin

In 1956, 26-year-old Edsger Dijkstra invented a classic path-finding algorithm while out with his fiancée at a café in Amsterdam. It all happened in his head: "Without pencil and paper you are almost forced to avoid all avoidable complexities," he said

AvG45a/EWD901a - 0

A simple fixpoint argument without the restriction to continuity

by

Edsger W. Dijkstra & A.J.M. van Gasteren

Abstract

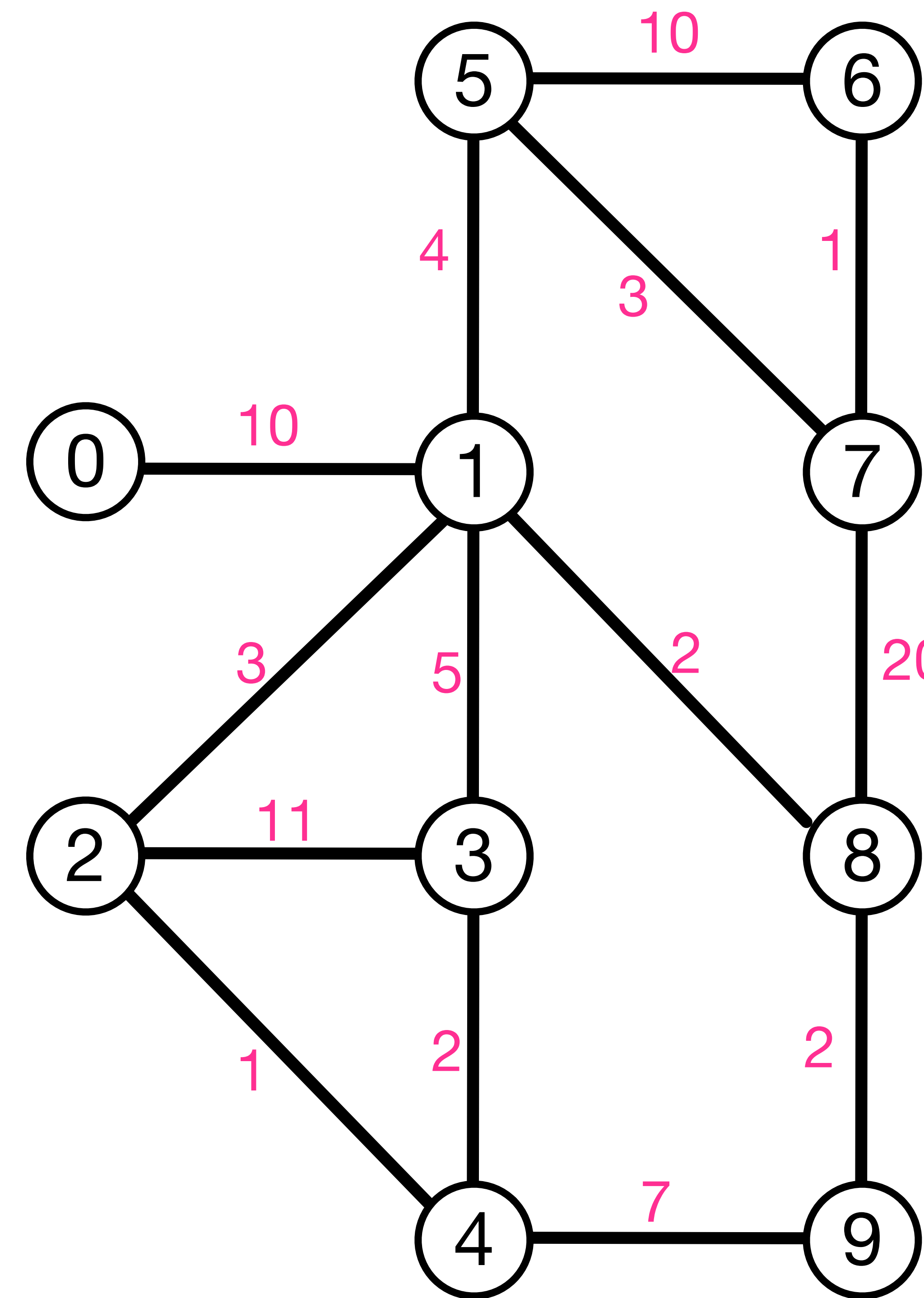
In programming language semantics, the introduction of unbounded nondeterminacy, which amounts to the introduction of noncontinuous predicate transformers, is needed for dealing with such concepts as fair interleaving. With the semantics of the repetition given as the strongest solution of a fixpoint equation, the weakest precondition expressed in closed form would then require transfinite ordinals. Here, however, it is shown that, even in the case of unbounded nondeterminacy, the fundamental theorem about the repetition can be proved by a simple and quite elementary argument.

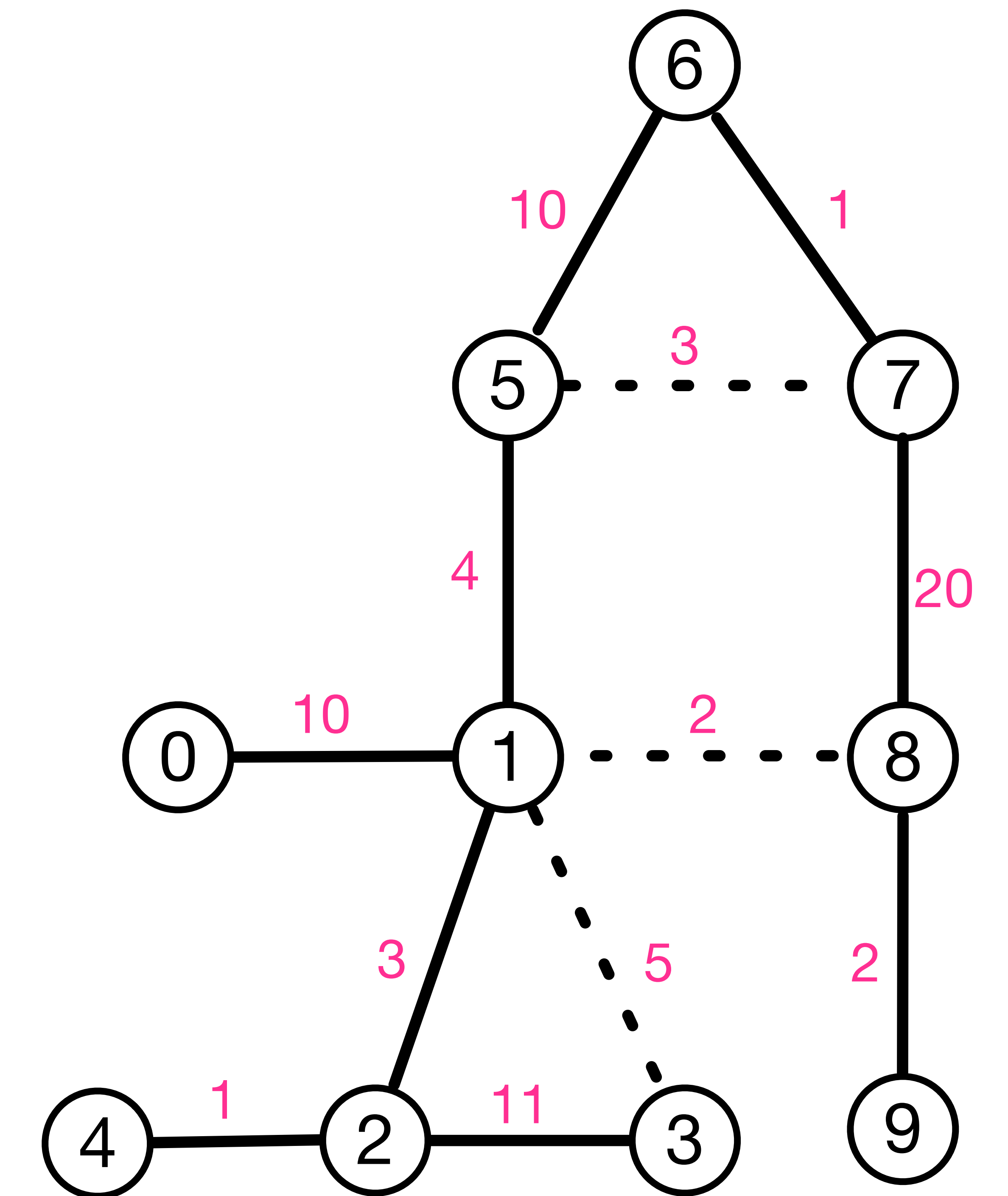
https://www.cs.utexas.edu/~EWD/indexEWDnums.html

Recherche du plus court chemin

- recherche du plus court chemin avec [Dijkstra 1959]

```
def shortest_path1 (g, w, dist, fp) :
    pred = {x:[ ] for x in g.vertices }
    while not fp.is_empty() :
        (d, v) = fp.deq()
        if v == w :
            return (d, pred[v] + [v])
        else :
            for (x, dvx) in g.successors(v) :
                dx = dist[x]
                dnewx = dist[v] + dvx
                if dnewx < dx :
                    pred[x] = pred[v] + [v]
                    if dx == max_int :
                        fp.enq ((dnewx, x))
                    else :
                        fp.update ((dx, x), (dnewx, x))
                    dist[x] = dnewx
    return [ ]

def shortest_path (g, v, w) :
    dist = {x: 0 if x == v else max_int for x in g.vertices}
    fp = PQueue()
    fp.enq ((0, v))
    return shortest_path1 (g, w, dist, fp)
```

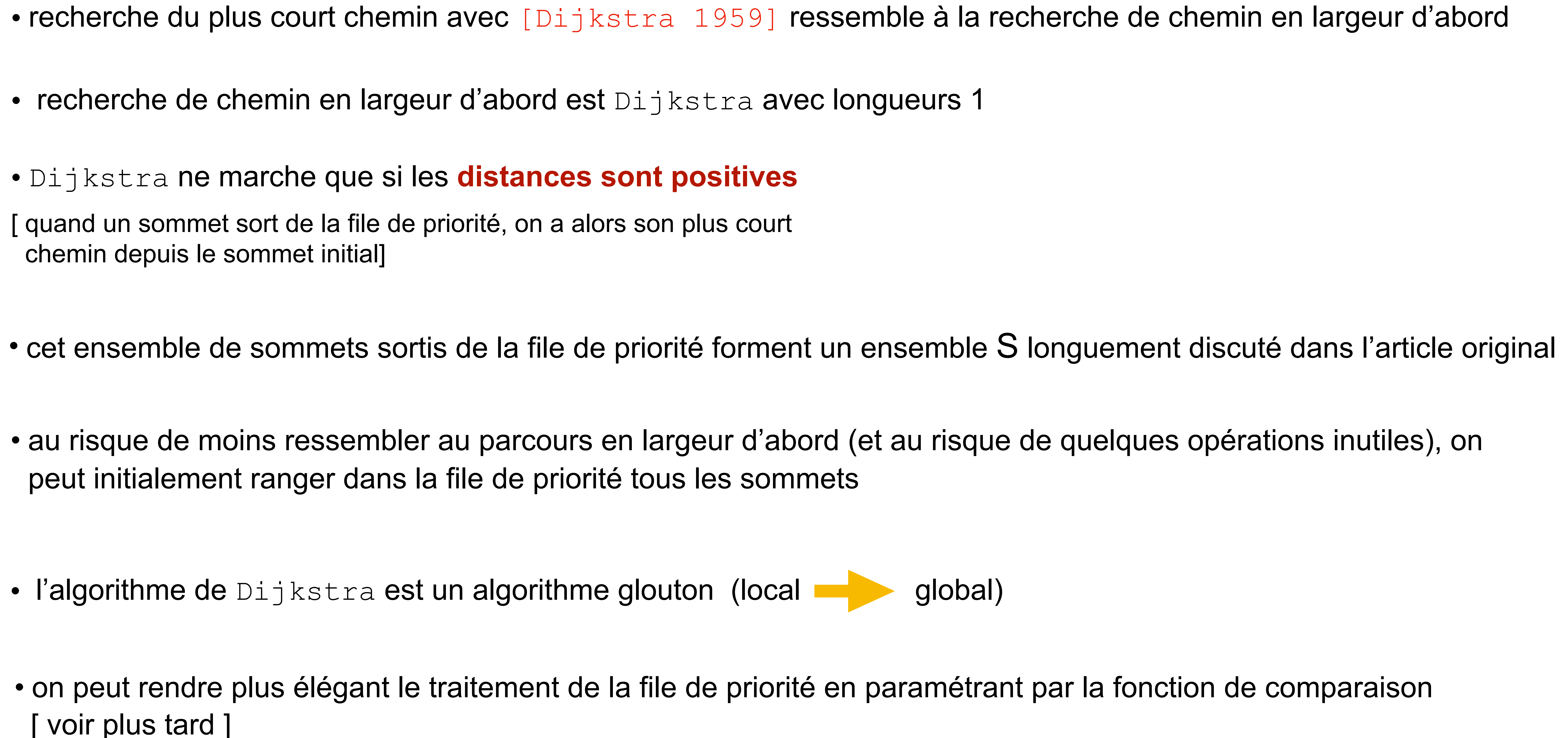





```
print (shortest_path (g3, 6, 3))

>>> (13, [6, 7, 5, 1, 3])
```

complexité en *O((V + E) log(V))*

Recherche du plus court chemin



- recherche du plus court chemin avec `[Dijkstra 1959]` ressemble à la recherche de chemin en largeur d'abord

- recherche de chemin en largeur d'abord est `Dijkstra` avec longueurs 1

- `Dijkstra` ne marche que si les **distances sont positives**

[quand un sommet sort de la file de priorité, on a alors son plus court chemin depuis le sommet initial]

- cet ensemble de sommets sortis de la file de priorité forment un ensemble S longuement discuté dans l'article original

- au risque de moins ressembler au parcours en largeur d'abord (et au risque de quelques opérations inutiles), on peut initialement ranger dans la file de priorité tous les sommets

- l'algorithme de `Dijkstra` est un algorithme glouton (local ➜ global)

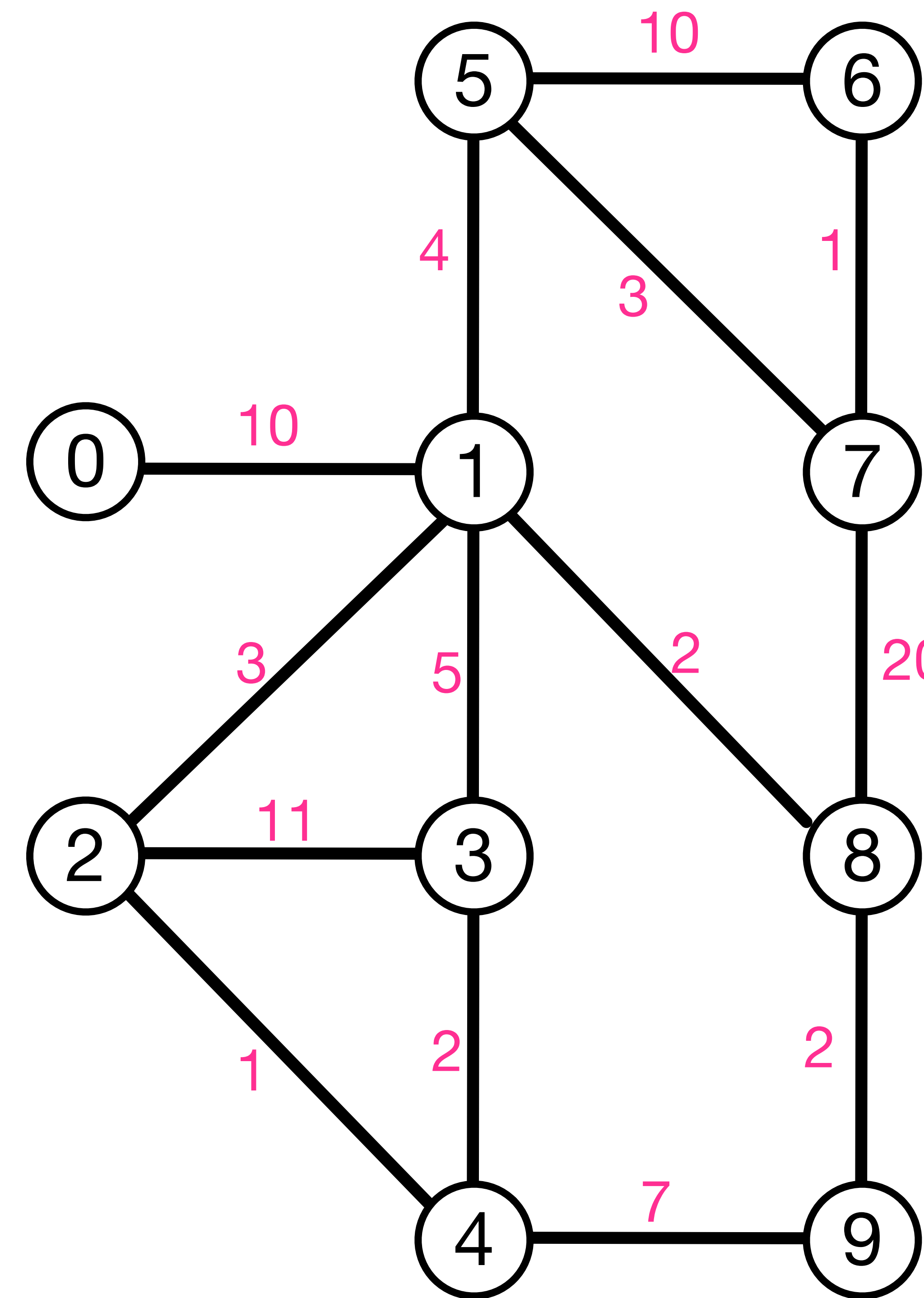
- on peut rendre plus élégant le traitement de la file de priorité en paramétrant par la fonction de comparaison [voir plus tard]

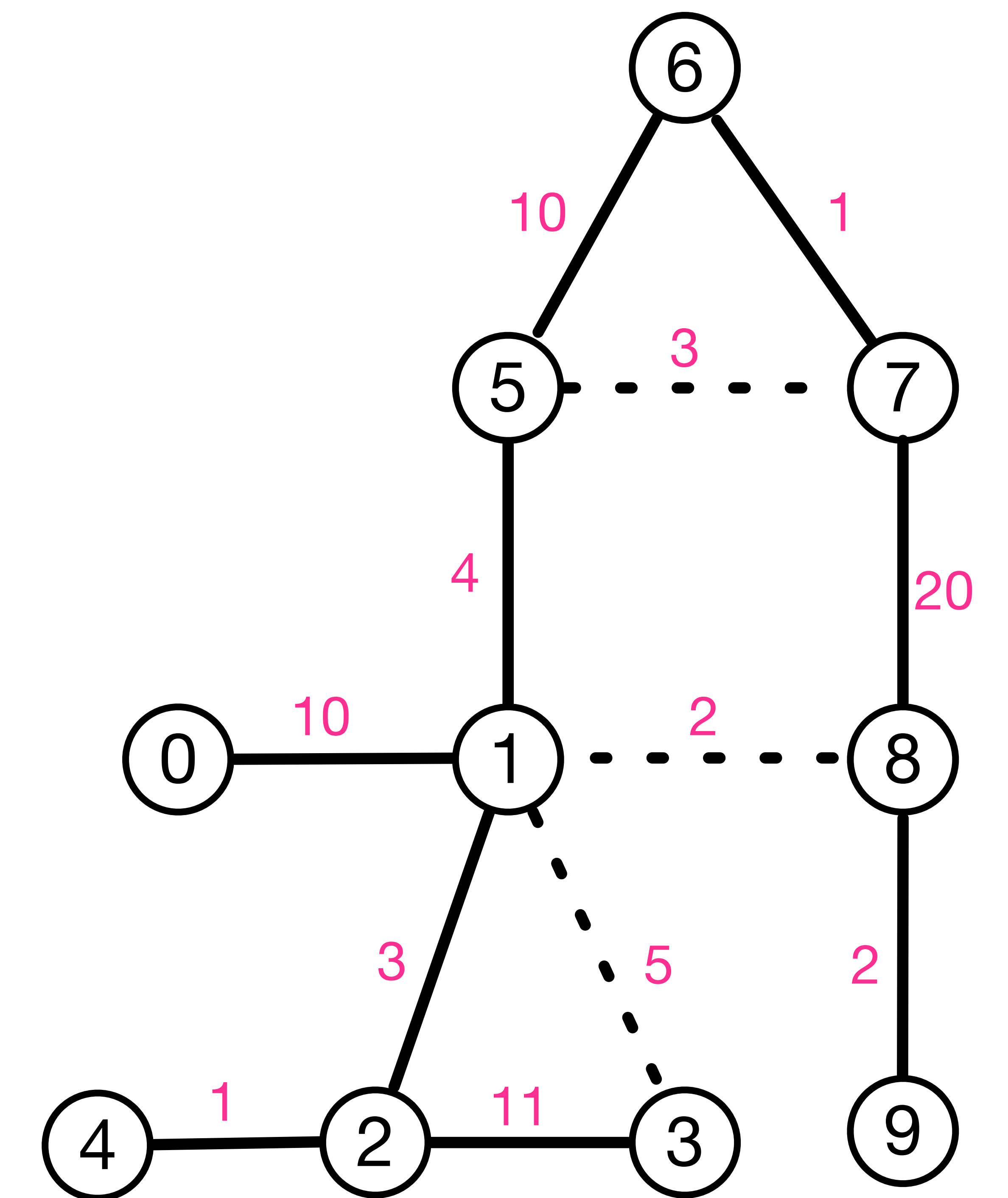
complexité en *O((V + E) log(V))*

Recherche du plus court chemin

- recherche du plus court chemin avec [Dijkstra 1959]

```
def shortest_path1 (g, w, dist, fp) :
    pred = {x:[ ] for x in g.vertices }
    while not fp.is_empty() :
        (d, v) = fp.deq()
        if v == w :
            return (d, pred[v] + [v])
        else :
            for (x, dvx) in g.successors(v) :
                dx = dist[x]
                dnewx = dist[v] + dvx
                if dnewx < dx :
                    pred[x] = pred[v] + [v]
                    fp.update ((dx, x), (dnewx, x))
                    dist[x] = dnewx
    return [ ]

def shortest_path (g, v, w) :
    dist = {x: 0 if x == v else max_int for x in g.vertices}
    fp = PQueue()
    for x in g.vertices : fp.enq ((dist[x], x))
    return shortest_path1 (g, w, dist, fp)
```






```
print (shortest_path (g3, 6, 3))
>>> (13, [6, 7, 5, 1, 3])
```

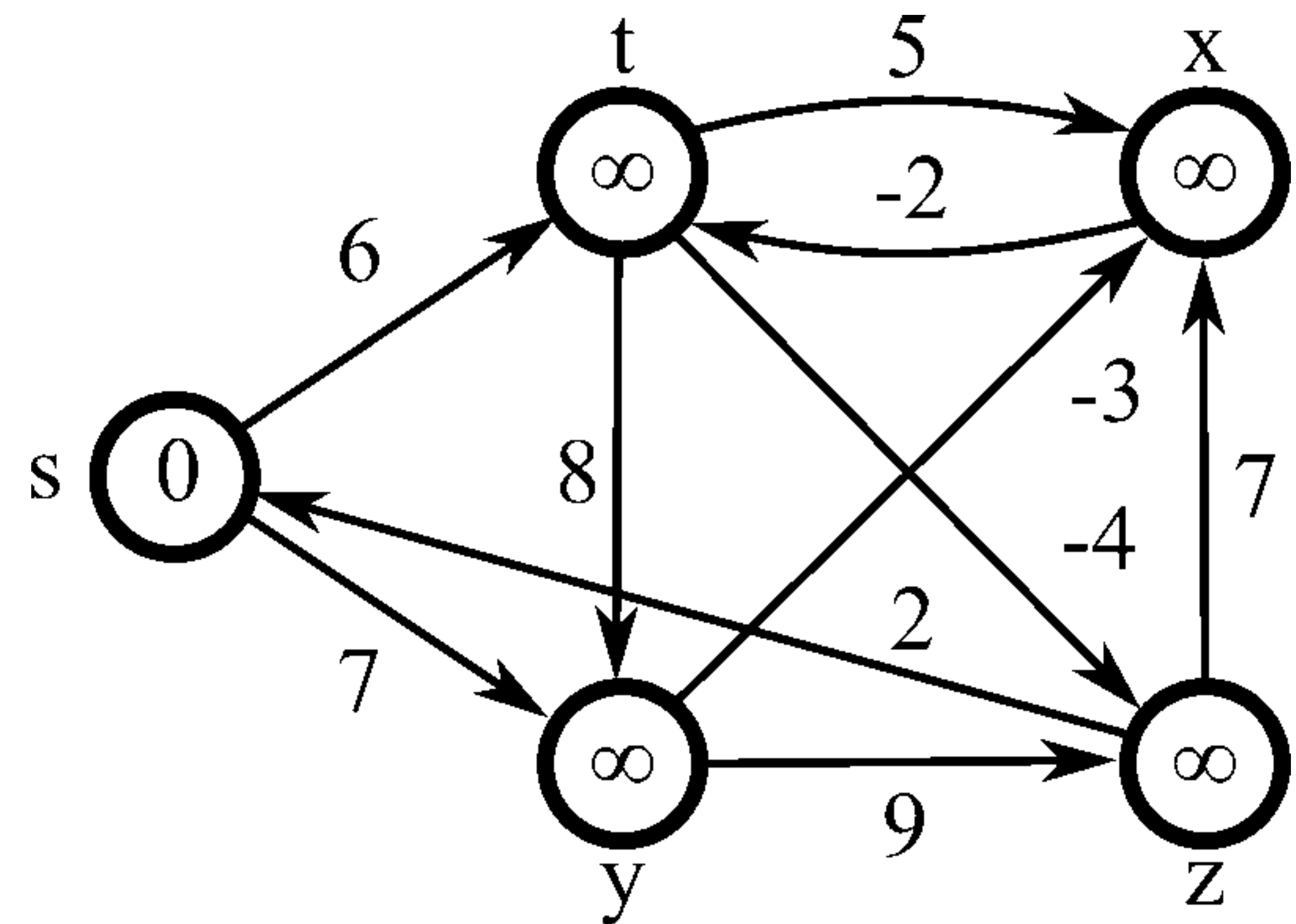
complexité en *O((V + E) log(V))*

```
max_int = 100000
```

Recherche du plus court chemin

- si les distances négatives
- algorithme de `Bellman-Ford`
- il ne marche que s'il n'existe pas de cycle complètement négatif
- donc pour un graphe valué dirigé

```
def bellman_ford (g, u0, u1) :
    dist = {x: max_int for x in g.vertices}
    pred = {x: [] for x in g.vertices}
    dist[u0] = 0
    for k in range (g.order() - 1) :
        for u in g.vertices :
            for (v, d) in g.successors (u) :
                if dist[u] + d < dist[v] :
                    dist[v] = dist[u] + d
                    pred[v] = pred[u] + [u]
    return (dist[u1], pred[u1])
```



complexité en *O((V x E))*

Recherche du plus court chemin

Exercice Modifier `Bellman-Ford` pour tester s'il existe un cycle négatif.

Exercice Donner un idée d'algorithme pour calculer les plus courtes distances entre tous les sommets.

Prochain cours

- graphes et matrices d’adjacences
- tri topologique
- composantes connexes
- composantes fortement connexes