Informatique tronc commun — MP*

# Révisions sur les graphes

# Introduction

# Le programme (première année)

### 2.3 Bases des graphes, plus courts chemins

Il s'agit de définir le modèle des graphes, leurs représentations et leurs manipulations.
On s'efforce de mettre en avant des applications importantes et si possible modernes : réseau de transport, graphe du web, réseaux sociaux, bio-informatique. On précise autant que possible la taille typique de tels graphes.

| **Notions** | **Commentaires** |
|---|---|
| Vocabulaire des graphes. | Graphe orienté, graphe non orienté. Sommet (ou nœud) ; arc, arête. Boucle. Degré (entrant et sortant). Chemin d'un sommet à un autre. Cycle. Connexité dans les graphes non orientés.<br>On présente l'implémentation des graphes à l'aide de listes d'adjacence (rassemblées par exemple dans une liste ou dans un dictionnaire) et de matrice d'adjacence. On n'évoque ni multi-arcs ni multi-arêtes. |
| Notations. | Graphe $G = (S, A)$, degrés $d(s)$ (pour un graphe non orienté), $d_+(s)$ et $d_-(s)$ (pour un graphe orienté). |
| Pondération d'un graphe. Étiquettes des arcs ou des arêtes d'un graphe. | On motive l'ajout d'information à un graphe par des exemples concrets. |
| Parcours d'un graphe. | On introduit à cette occasion les piles et les files ; on souligne les problèmes d'efficacité posés par l'implémentation des files par les listes de Python et l'avantage d'utiliser un module dédié tel que `collections.deque`.<br>Détection de la présence de cycles ou de la connexité d'un graphe non orienté. |
| Recherche d'un plus court chemin dans un graphe pondéré avec des poids positifs. | Algorithme de Dijkstra. On peut se contenter d'un modèle de file de priorité naïf pour extraire l'élément minimum d'une collection. Sur des exemples, on s'appuie sur l'algorithme A* vu comme variante de celui de Dijkstra pour une première sensibilisation à la notion d'heuristique. |

# Aux concours : en informatique

| Sujet | Le graphe | Ce qui est demandé |
|---|---|---|
| X-ENS 2016 | réseau social, liste d'arêtes | fonctions de base et complexité, structure de partition |
| X-ENS 2019 | grille de cases colorées | région unicolore (récursif), terminaison, critique d'un code |
| Mines 2022 | grille de spins | voisins d'une case, exploration d'un domaine, pile |
| X-ENS 2023 | grille implicite (`successeurs`) | Dijkstra fourni : que calcule-t-il ? complexité, heuristique |
| Mines 2024 | graphe en couches, probabilités | compter, dessiner, plus court chemin, prog. dynamique |
| X-ENS 2025 | grille avec sauts | chemins, absence de cycle, mémoïsation |
| X-ENS 2026 | dictionnaire d'adjacence, orienté | parcours en profondeur fourni : dérouler, prouver, compter |

souvent un « graphe déguisé », dont la représentation est fournie

# Aux concours : en mathématiques aussi

| Sujet | Le graphe | Ce qui est demandé |
|---|---|---|
| Centrale 2016, maths 1 | graphe d'une matrice positive | chemins et coefficients de $A^m$ |
| ENS 2016, maths D | graphe et matrice d'adjacence | constante de Cheeger |
| Centrale 2021, maths 1 (PSI) | marche aléatoire sur un graphe | matrices stochastiques, PageRank |
| Mines 2024, maths 2 | graphes aléatoires | rang, polynôme caractéristique, seuil |
| Centrale 2025, maths 2 | modèle d'Ising : cycle, grille | écrire `mat_adj(graphe)` en Python, valeurs propres |

# L'idée du chapitre (Python)

```python
def parcours_sac(G, depart):
    vus = [False for s in range(len(G))]
    ordre = []
    sac = sac_vide()
    ajouter(sac, depart)
    while len(sac) > 0:
        s = retirer(sac)
        if not vus[s]:
            vus[s] = True
            ordre.append(s)
            for t in G[s]:
                ajouter(sac, t)
    return ordre
```

**Un seul parcours, trois « sacs »**

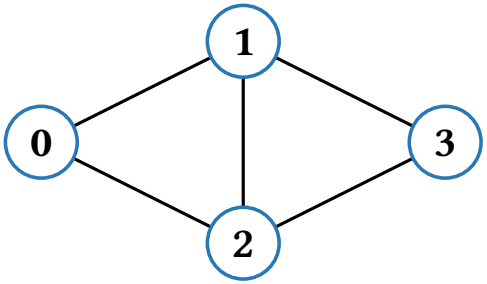
| Le sac | On obtient |
|---|---|
| pile | profondeur |
| file | largeur |
| file de priorité | Dijkstra |

chaque problème classique = ce parcours + **une information** (prédécesseur, distance, numéro)

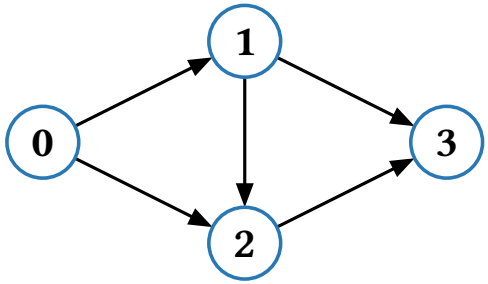
# Vocabulaire et représentations

# Quatre sortes de graphes

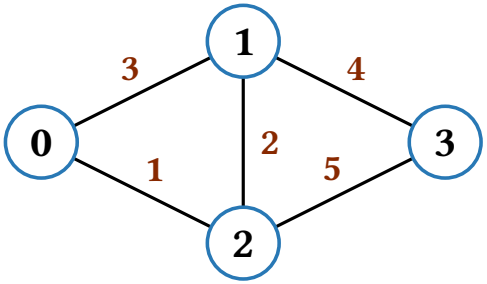
**non orienté**



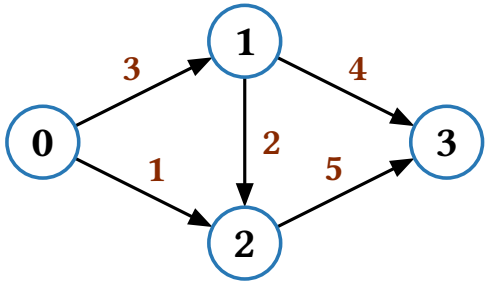

**orienté**



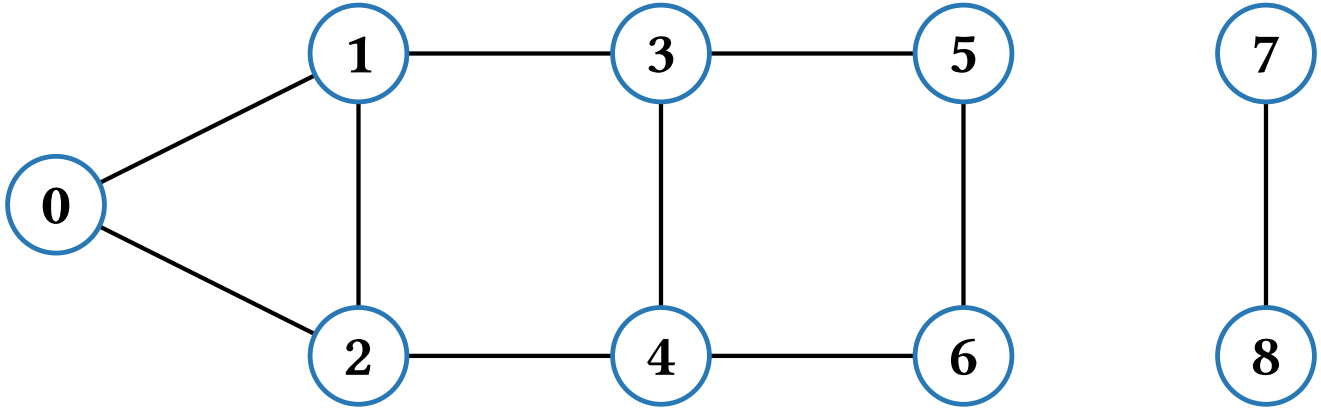

**non orienté pondéré**




**orienté pondéré**

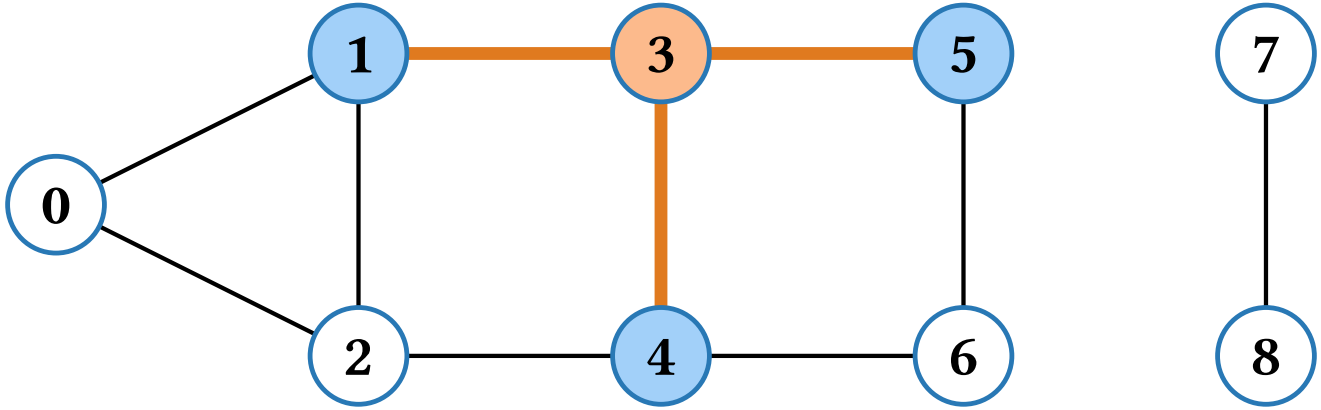


# Le vocabulaire sur le graphe du cours



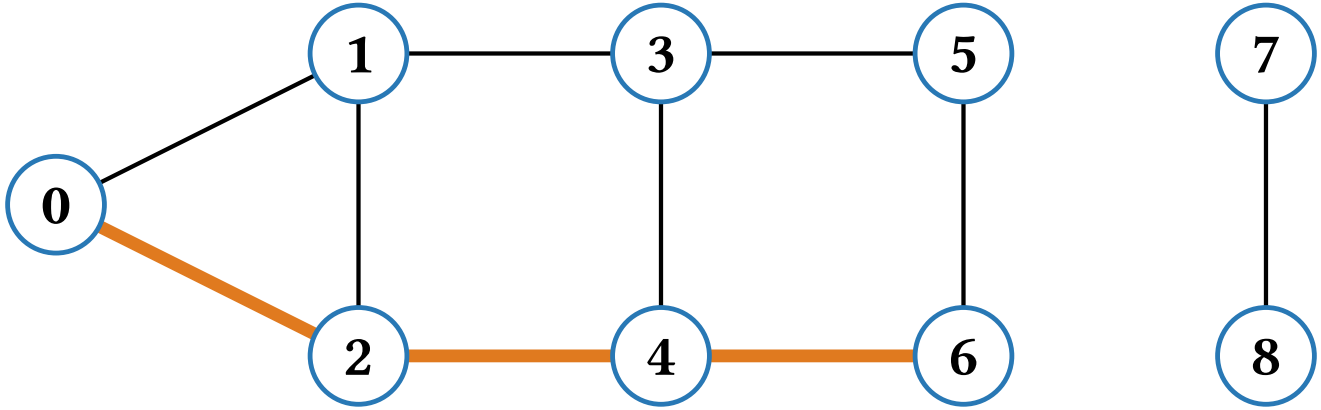

$n = 9$ sommets, $m = 10$ arêtes

# Le vocabulaire sur le graphe du cours



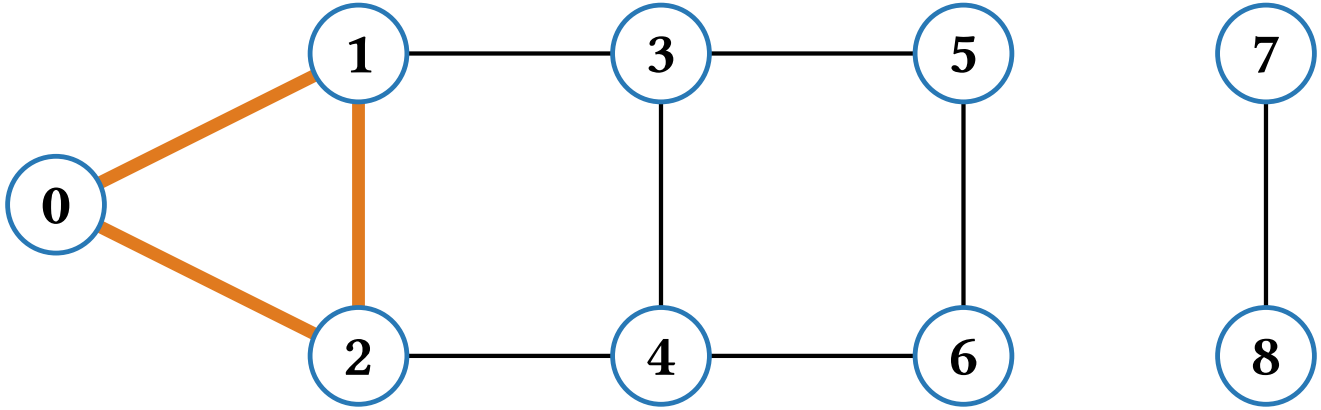

**voisins** de $3$ : $1, 4, 5$ **degré** $d(3) = 3$

# Le vocabulaire sur le graphe du cours



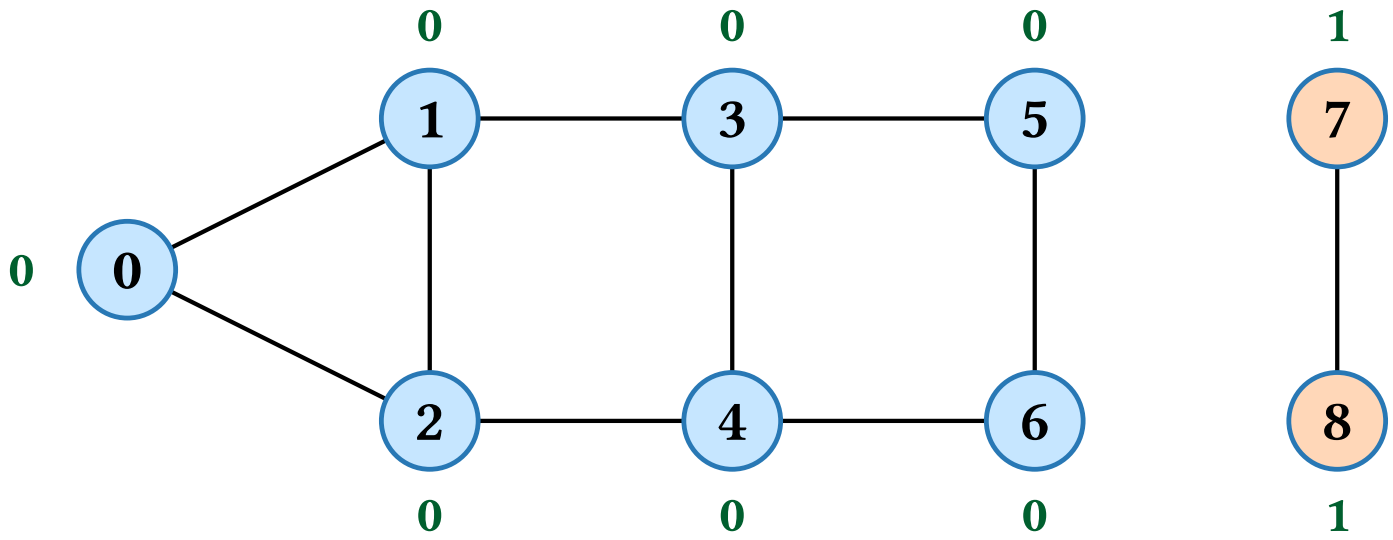

**chemin** 0, 2, 4, 6, de longueur 3

# Le vocabulaire sur le graphe du cours



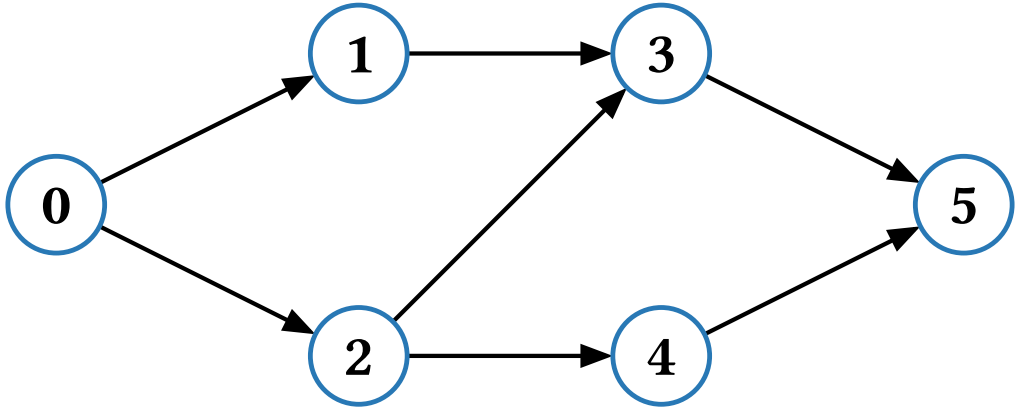

**cycle** $0, 1, 2, 0$

# Le vocabulaire sur le graphe du cours



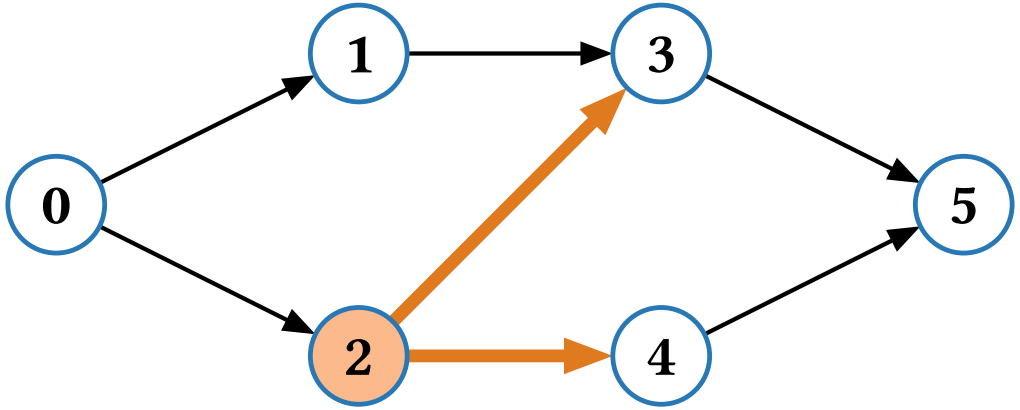

deux **composantes connexes** : le graphe n'est pas connexe

# Graphe orienté



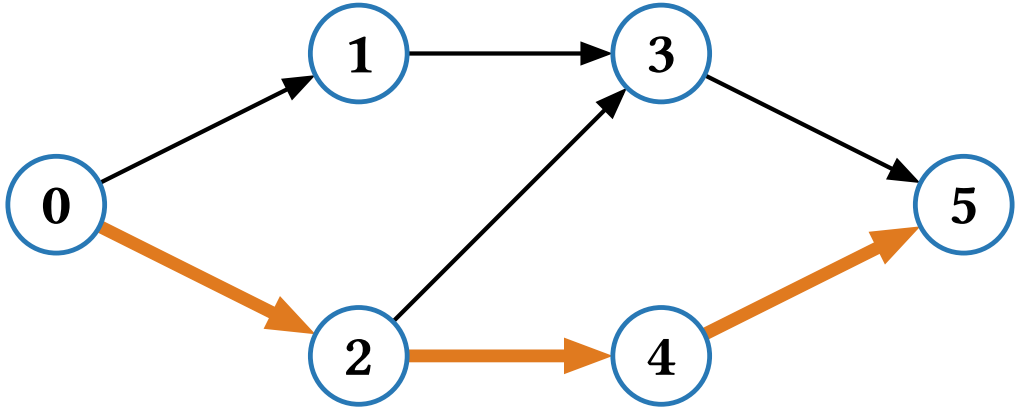

$n = 6, m = 7$

# Graphe orienté



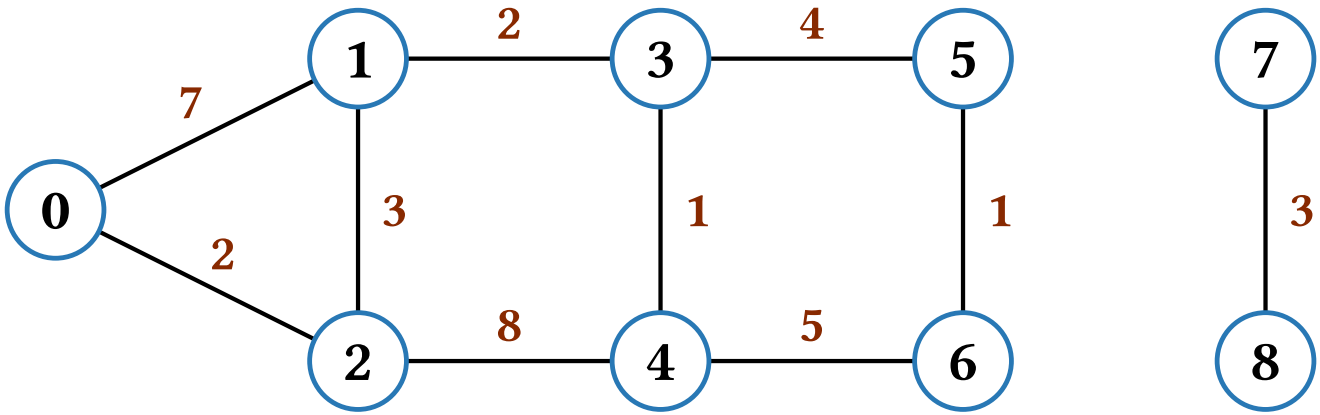

degré sortant $d^+(2) = 2$, degré entrant $d^-(2) = 1$

# Graphe orienté



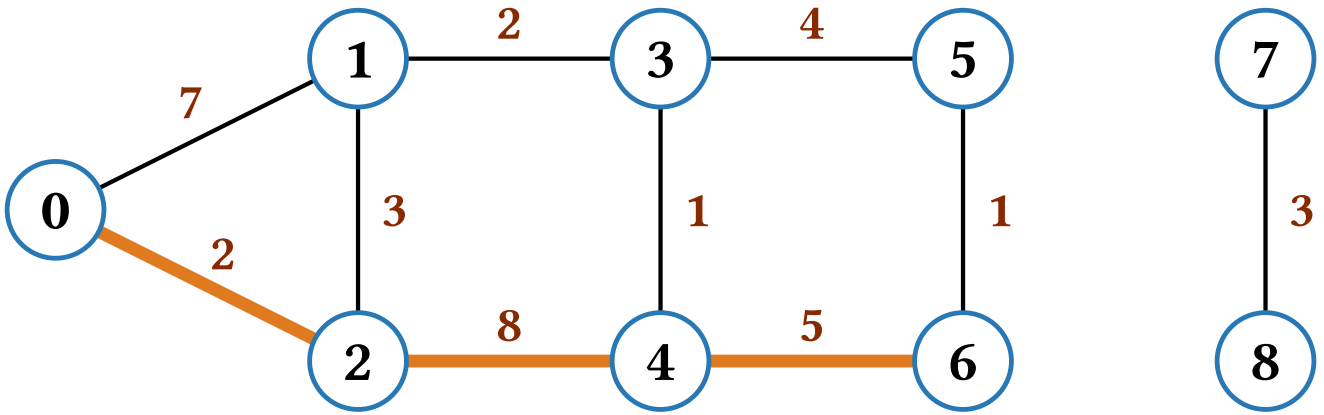

0, 2, 4, 5 est un chemin ; aucun chemin de 5 à 0, aucun cycle

# Graphe pondéré : poids d'un chemin, distance



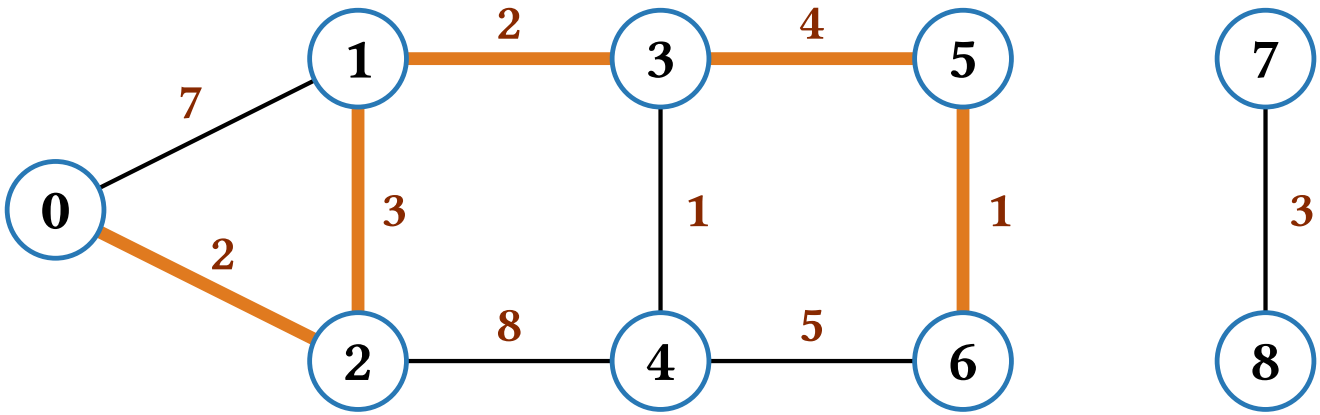

un poids $p(a)$ sur chaque arête

# Graphe pondéré : poids d'un chemin, distance



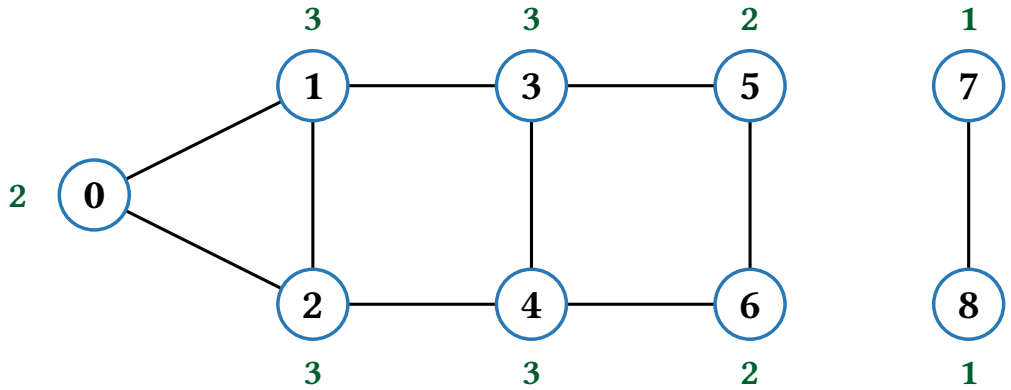

chemin $0, 2, 4, 6$ : poids $2 + 8 + 5 = 15$

# Graphe pondéré : poids d'un chemin, distance




chemin $0, 2, 1, 3, 5, 6$ : poids 12, c'est un **plus court chemin** : $\delta(0, 6) = 12$

# Le lemme des poignées de main

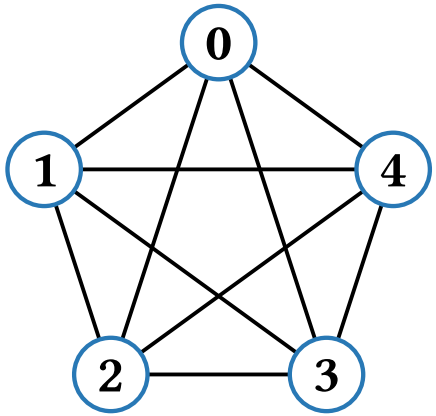



**Lemme des poignées de main.** non orienté : $\sum_{s \in S} d(s) = 2m$
orienté : $\sum_{s \in S} d^+(s) = \sum_{s \in S} d^-(s) = m$

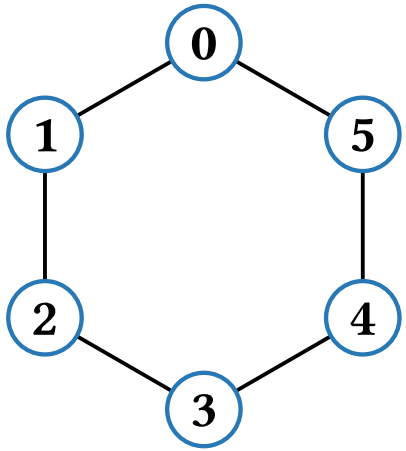
- chaque arête est comptée une fois par extrémité : ici la somme vaut $20 = 2 \times 10$
- parcourir les voisins de **tous** les sommets coûte $O(n + m)$, pas $O(n^2)$

# Quelques graphes particuliers

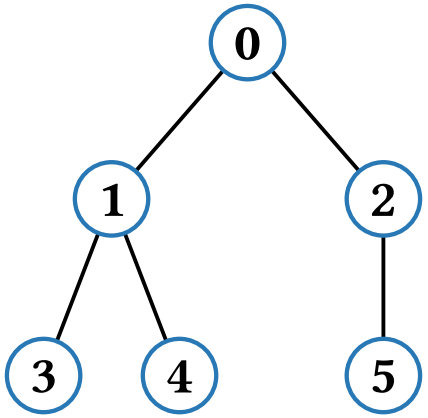
**$K_5$ : graphe complet**




**$C_6$ : cycle**

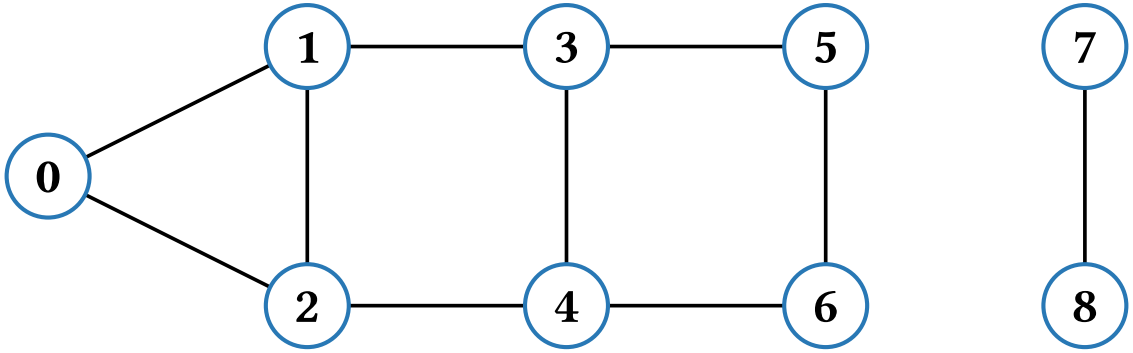



**un arbre**




$K_n : m = \frac{n(n-1)}{2}$ $\quad$ $C_n : m = n$ $\quad$ arbre (connexe, sans cycle) : $m = n - 1$

# Listes d'adjacence (Python)



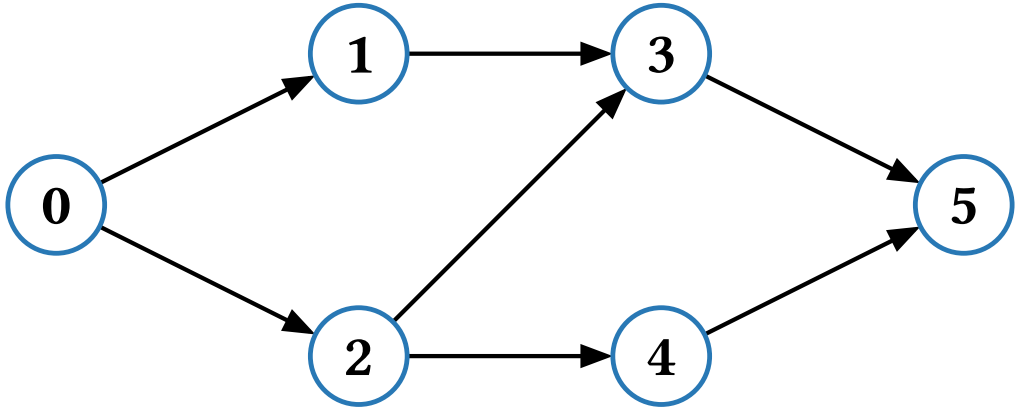

```python
G = [[1, 2],        # voisins de 0
     [0, 2, 3],     # voisins de 1
     [0, 1, 4],     # voisins de 2
     [1, 4, 5],     # voisins de 3
     [2, 3, 6],     # voisins de 4
     [3, 6],        # voisins de 5
     [4, 5],        # voisins de 6
     [8],           # voisins de 7
     [7]]           # voisins de 8
```

`G[s]` : liste des voisins de `s`

chaque arête apparaît **deux fois**

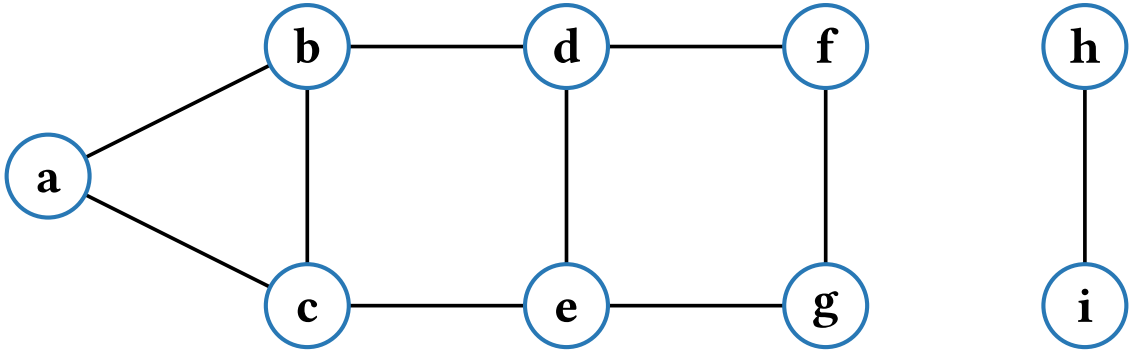
**la représentation de référence**

# Listes d'adjacence : graphe orienté (Python)




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

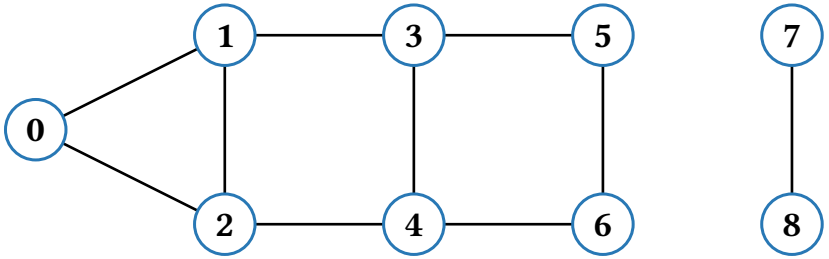
# Dictionnaire d'adjacence (Python)




```python
D = {"a": ["b", "c"],
     "b": ["a", "c", "d"],
     "c": ["a", "b", "e"],
     "d": ["b", "e", "f"],
     "e": ["c", "d", "g"],
     "f": ["d", "g"],
     "g": ["e", "f"],
     "h": ["i"],
     "i": ["h"]}
```

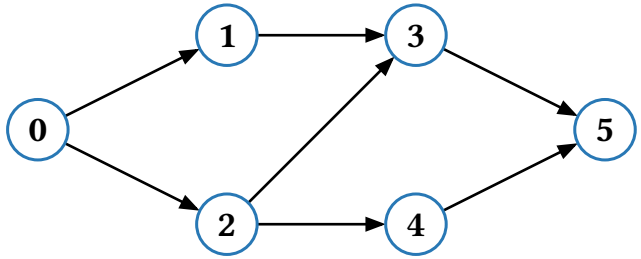
quand les sommets ne sont pas des entiers

# Matrice d'adjacence




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 4 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

`M[s][t]` vaut 1 s'il y a une arête de `s` vers `t`

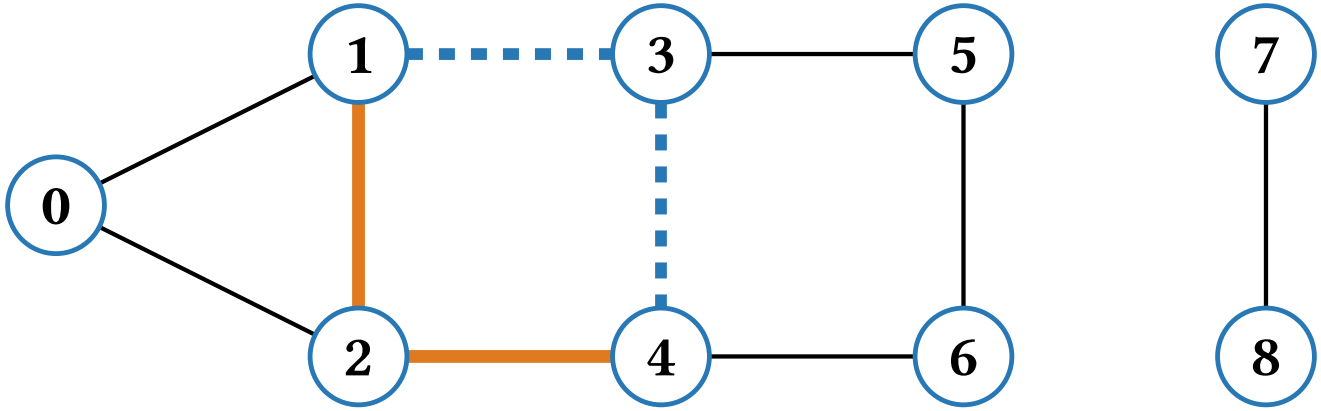



| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 1 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 | 0 | 1 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |

non orienté : matrice symétrique

## Les puissances de la matrice d'adjacence

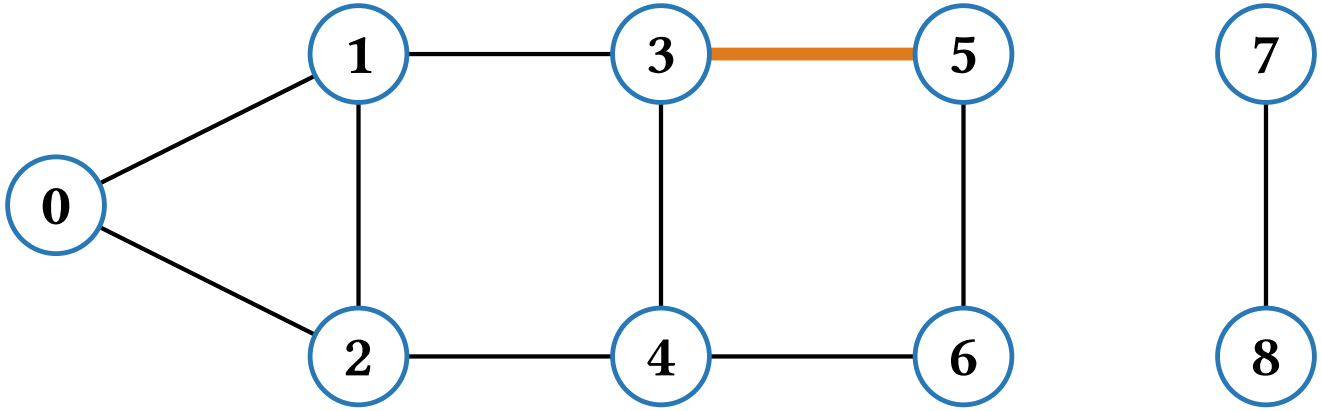
**Propriété.** $\left(M^k\right)_{s,t}$ est le nombre de chemins de longueur $k$ de $s$ à $t$ :
$\mathrm{tr}\left(M^k\right)$ compte les chemins **fermés** de longueur $k$.




$\left(M^2\right)_{1,4} = 2$ : les chemins $1, 2, 4$ et $1, 3, 4$

## Les puissances de la matrice d'adjacence

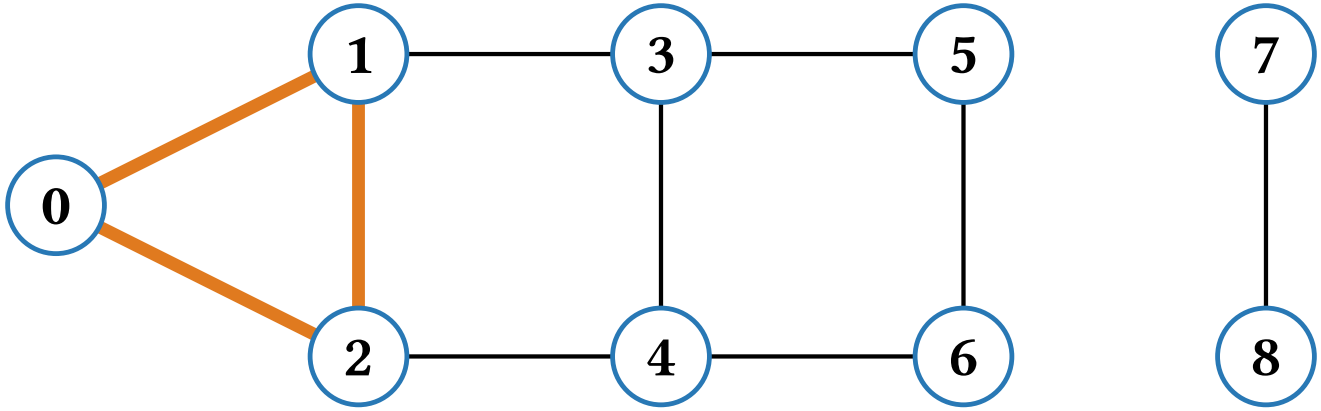
**Propriété.** $\left(M^k\right)_{s,t}$ est le nombre de chemins de longueur $k$ de $s$ à $t$ :
$\mathrm{tr}\left(M^k\right)$ compte les chemins **fermés** de longueur $k$.




$\mathrm{tr}\left(M^2\right) = 2m = 20$ : un aller-retour $s, t, s$ par arête et par sens

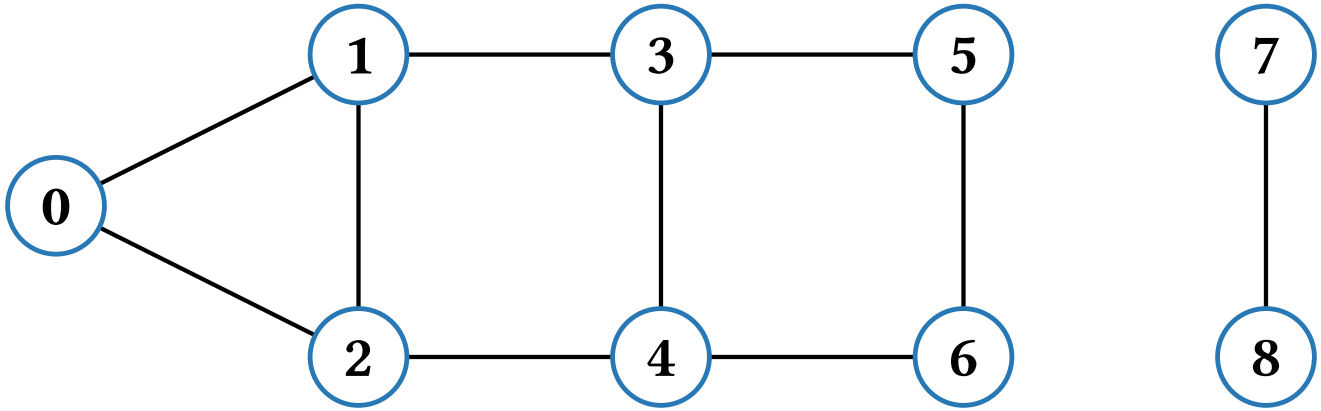
## Les puissances de la matrice d'adjacence

**Propriété.** $\left(M^k\right)_{s,t}$ est le nombre de chemins de longueur $k$ de $s$ à $t$ :
$\mathrm{tr}\left(M^k\right)$ compte les chemins **fermés** de longueur $k$.




$\mathrm{tr}\left(M^3\right) = 6 \times$ (nombre de triangles) $= 6$ : trois départs, deux sens de parcours

# Liste d'arêtes (Python)



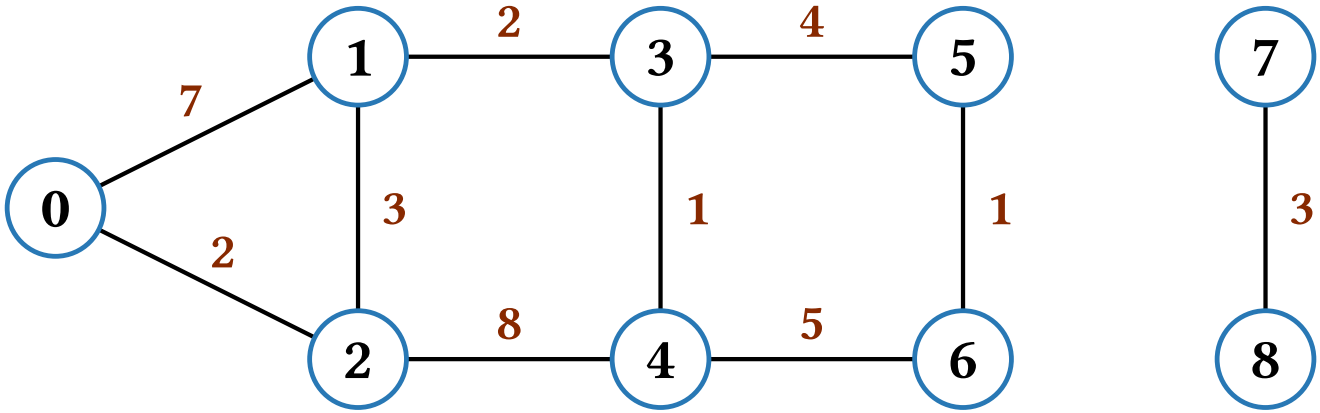

```
N = 9
ARETES = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 4),
          (3, 4), (3, 5), (4, 6), (5, 6), (7, 8)]
```

# Quatre représentations, quatre coûts

| | Listes d'adj. | Dictionnaire | Matrice | Liste d'arêtes |
|---|---|---|---|---|
| Mémoire | $O(n+m)$ | $O(n+m)$ | $O(n^2)$ | $O(m)$ |
| `s` et `t` voisins ? | $O(d(s))$ | $O(d(s))$ | $O(1)$ | $O(m)$ |
| Voisins de `s` | $O(d(s))$ | $O(d(s))$ | $O(n)$ | $O(m)$ |
| Toutes les arêtes | $O(n+m)$ | $O(n+m)$ | $O(n^2)$ | $O(m)$ |
| Degré de `s` | $O(1)$ | $O(1)$ | $O(n)$ | $O(m)$ |
| Ajouter une arête | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| Supprimer l'arête entre `s` et `t` | $O(d(s)+d(t))$ | $O(d(s)+d(t))$ | $O(1)$ | $O(m)$ |
| Ajouter un sommet | $O(1)$ | $O(1)$ | $O(n)$ | $O(1)$ |

un parcours énumère des voisins : **listes d'adjacence** matrice : tester une arête en $O(1)$, mais $O(n^2)$ ailleurs

# Graphes pondérés (Python)




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

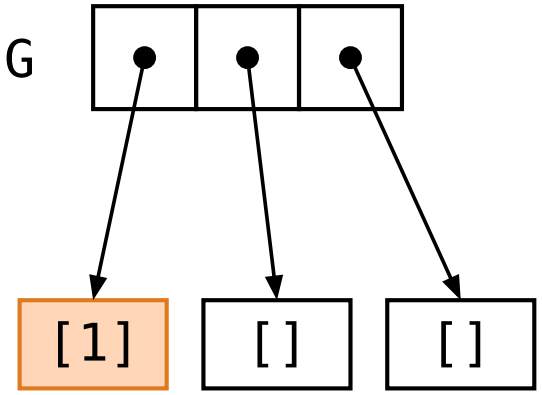
## D'une représentation à une autre (Python)

```python
def aretes_vers_liste(n, aretes):
    G = [[] for s in range(n)]
    for (s, t) in aretes:
        G[s].append(t)
        G[t].append(s)
    return G
```

toujours le même schéma : créer la structure vide, puis la remplir en parcourant l'autre

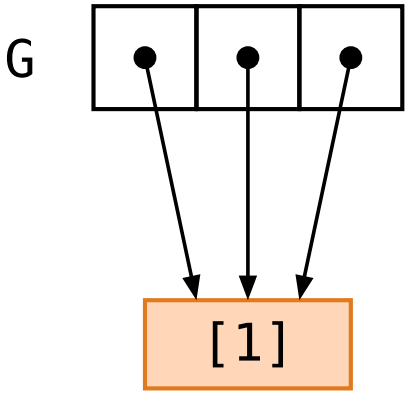
# Attention : `[[]] * n`

```
G = [[] for s in range(3)]
```




`G vaut [[1], [], []]`

```
G = [[]] * 3
```




`G vaut [[1], [1], [1]]`

après `G[0].append(1)` : trois listes distinctes, ou trois fois **la même**

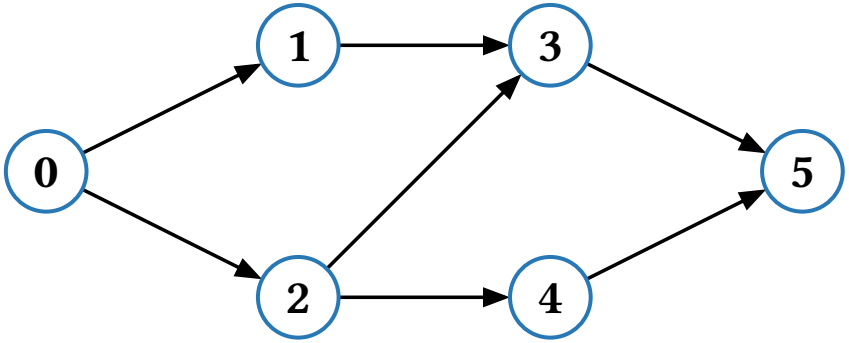
# Listes et matrice (Python)

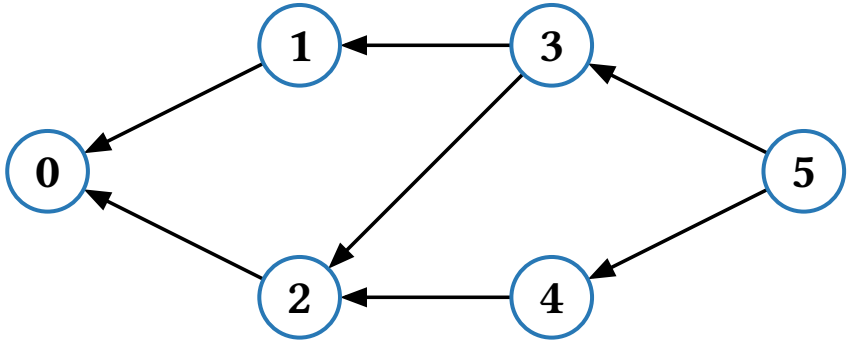
```python
def liste_vers_matrice(G):
    n = len(G)
    M = [[0 for t in range(n)] for s in range(n)]
    for s in range(n):
        for t in G[s]:
            M[s][t] = 1
    return M
```

```python
def matrice_vers_liste(M):
    n = len(M)
    G = [[] for s in range(n)]
    for s in range(n):
        for t in range(n):
            if M[s][t] == 1:
                G[s].append(t)
    return G
```

$O(n^2)$ dans les deux sens : avec une matrice, on ne descend jamais en dessous de $n^2$

# Graphe renversé (Python)



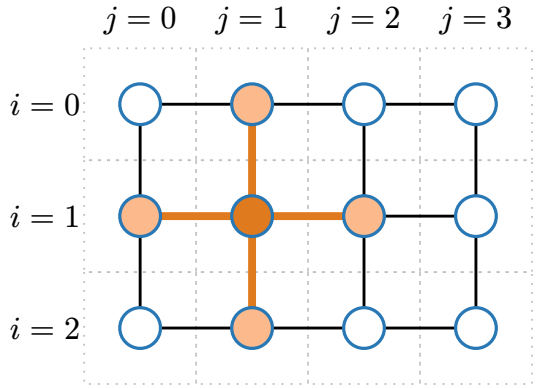




```python
def renverse(G):
    R = [[] for s in range(len(G))]
    for s in range(len(G)):
        for t in G[s]:
            R[t].append(s)
    return R
```

```
renverse(G_ORIENTE) :
[[], [0], [0], [1, 2], [2], [3, 4]]
```

# Les graphes qu'on ne construit pas : la grille (Python)

```python
def voisins_grille(nb_lignes, nb_colonnes, i, j):
    voisins = []
    for (di, dj) in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
        i_vois = i + di
        j_vois = j + dj
        if 0 <= i_vois and i_vois < nb_lignes:
            if 0 <= j_vois and j_vois < nb_colonnes:
                voisins.append((i_vois, j_vois))
    return voisins
```




$3 \times 4$ : 12 sommets, 17 arêtes

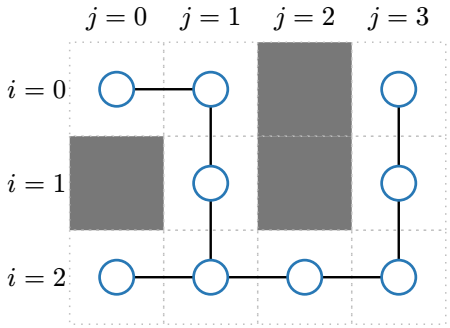
le graphe n'est stocké nulle part : on sait **calculer les voisins**

# La grille avec des cases interdites (Python)

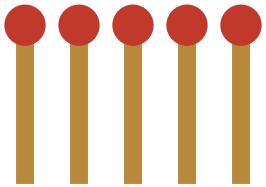
```python
def voisins_libres(grille, i, j):
    voisins = []
    for (i_vois, j_vois) in voisins_grille(len(grille), len(grille[0]), i, j):
        if grille[i_vois][j_vois] == 0:
            voisins.append((i_vois, j_vois))
    return voisins
```

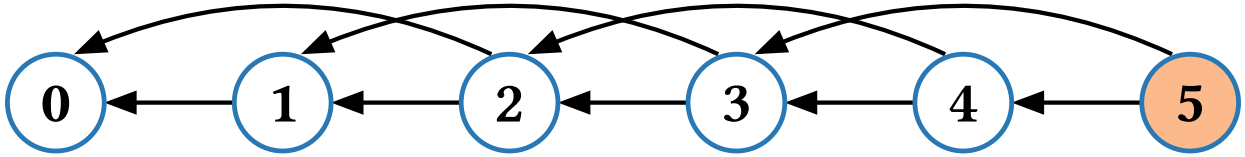
```python
LABYRINTHE = [[0, 0, 1, 0],
              [1, 0, 1, 0],
              [0, 0, 0, 0]]
```

trois murs : 9 sommets, 8 arêtes



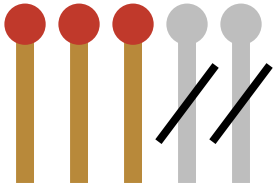
# Les états d'un jeu : les allumettes

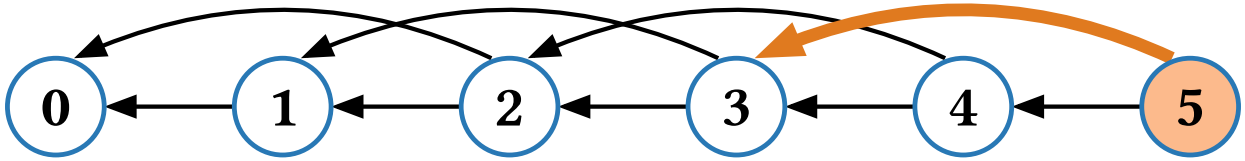





il reste 5 allumettes

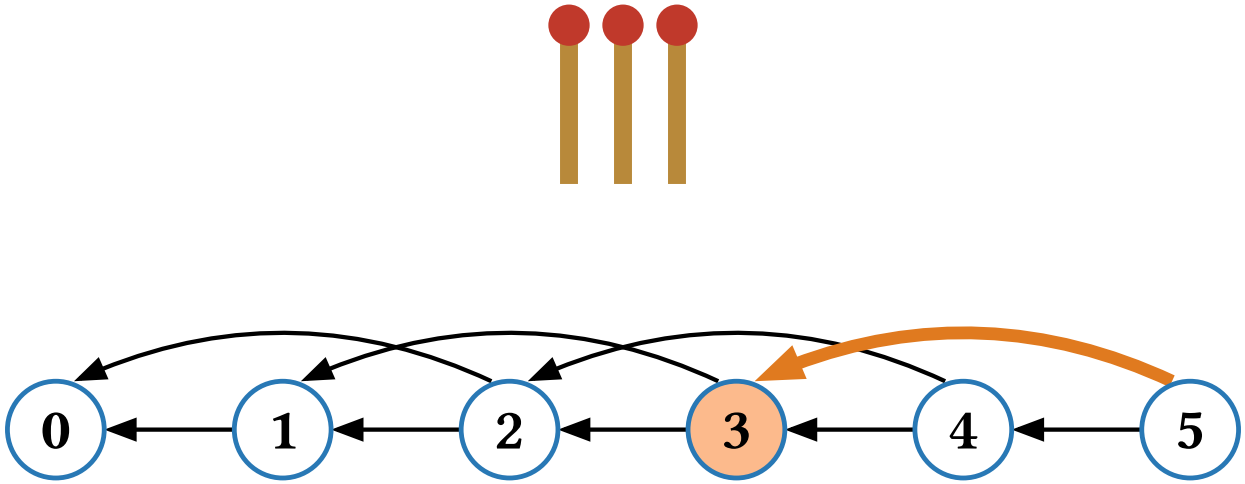
# Les états d'un jeu : les allumettes





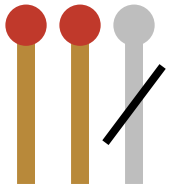

le premier joueur en retire deux

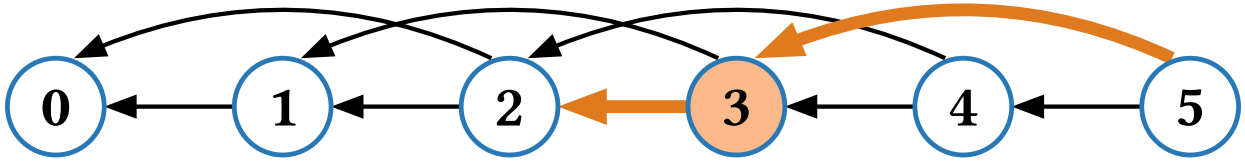
# Les états d'un jeu : les allumettes




il en reste 3

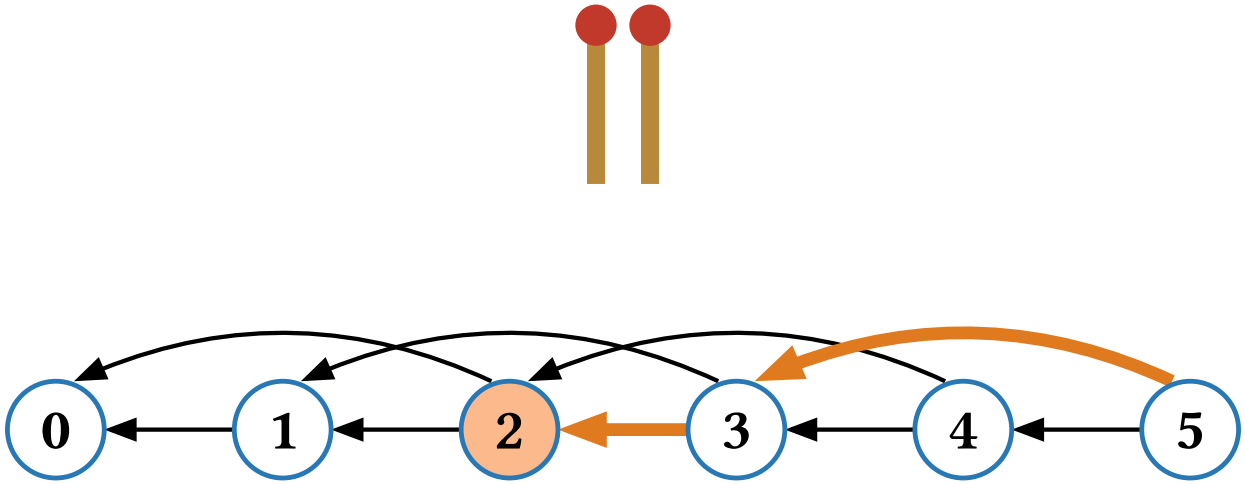
# Les états d'un jeu : les allumettes





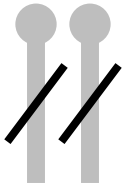

le second joueur en retire une

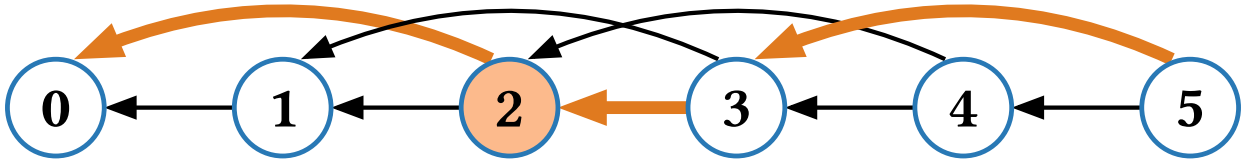
# Les états d'un jeu : les allumettes




il en reste 2

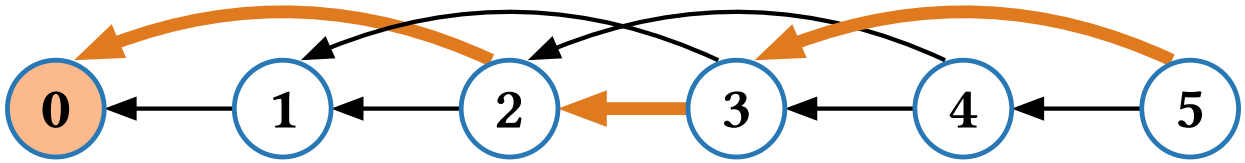
# Les états d'un jeu : les allumettes





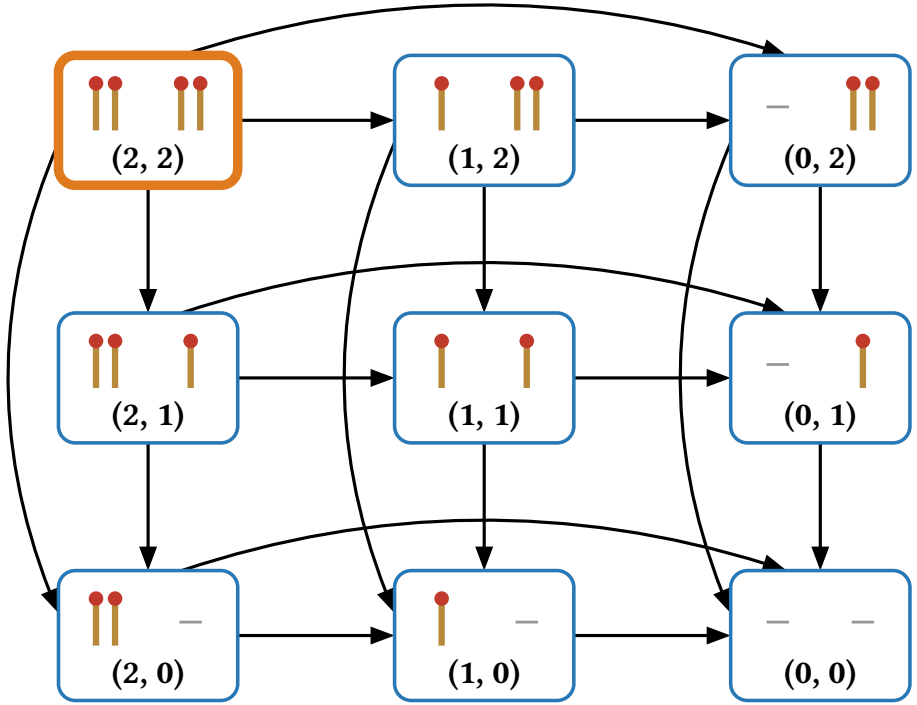

le premier joueur retire les deux dernières

# Les états d'un jeu : les allumettes



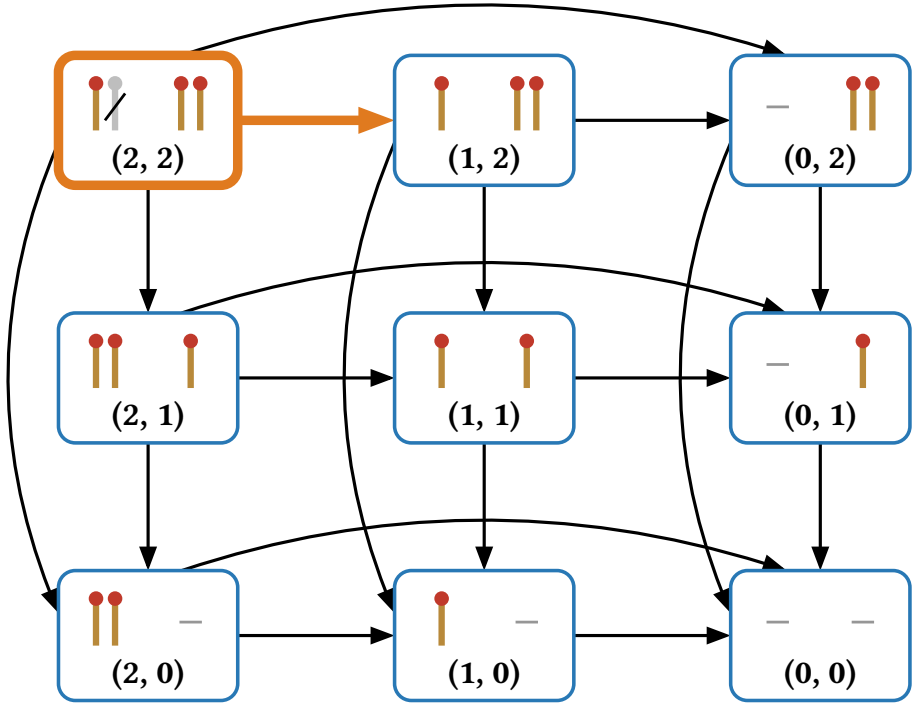

une partie = un chemin de 5 à 0

# Les états d'un jeu : Nim à deux tas



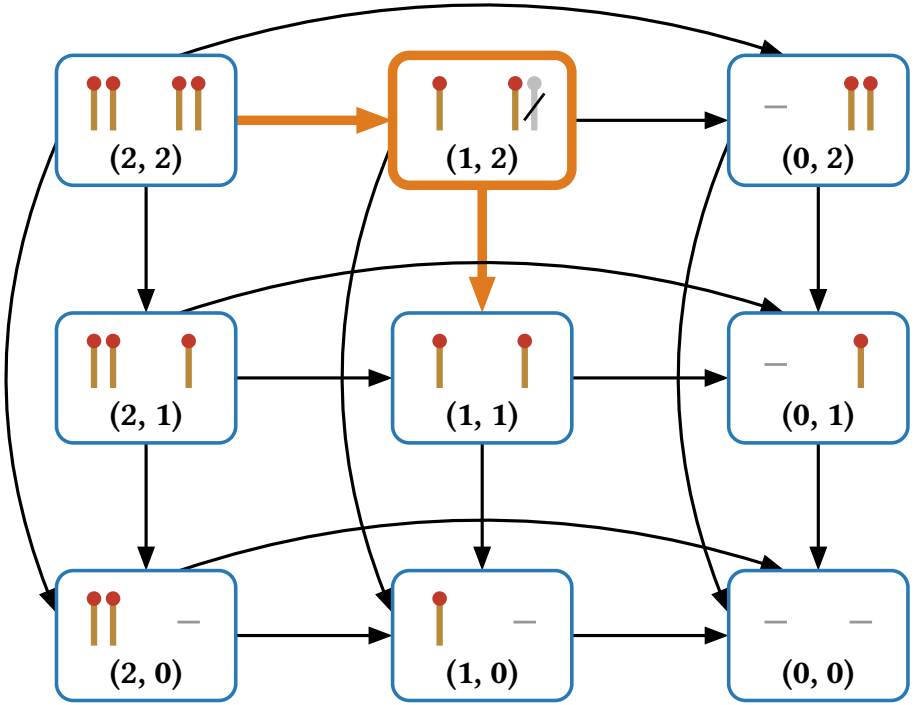

état $(a, b)$ : tailles des deux tas ; un coup vide en partie **un seul** tas

# Les états d'un jeu : Nim à deux tas



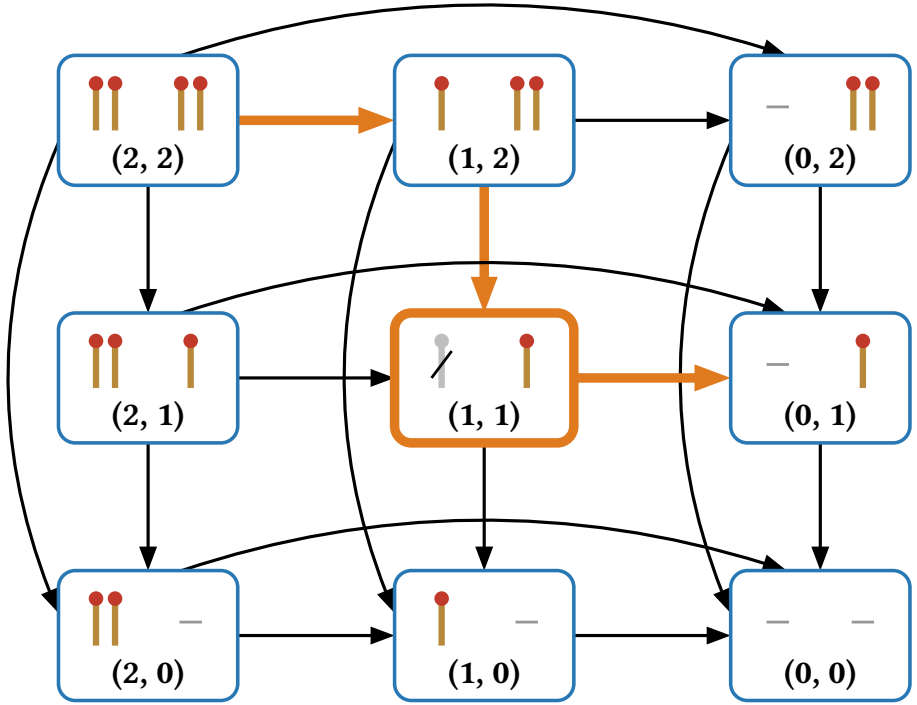

premier joueur : une allumette du premier tas

# Les états d'un jeu : Nim à deux tas



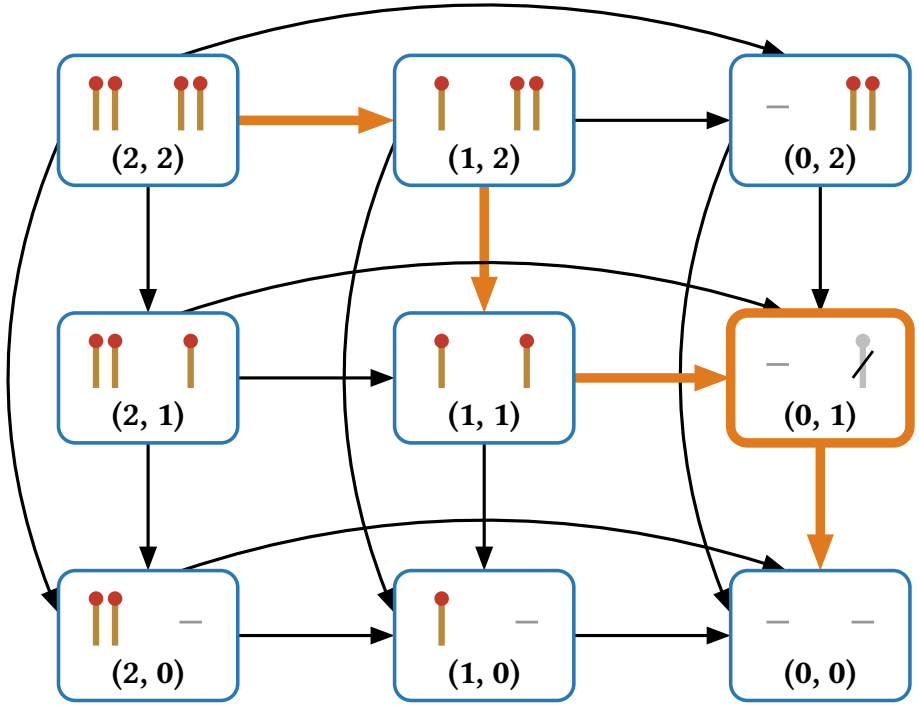

second joueur : une allumette du second tas

# Les états d'un jeu : Nim à deux tas



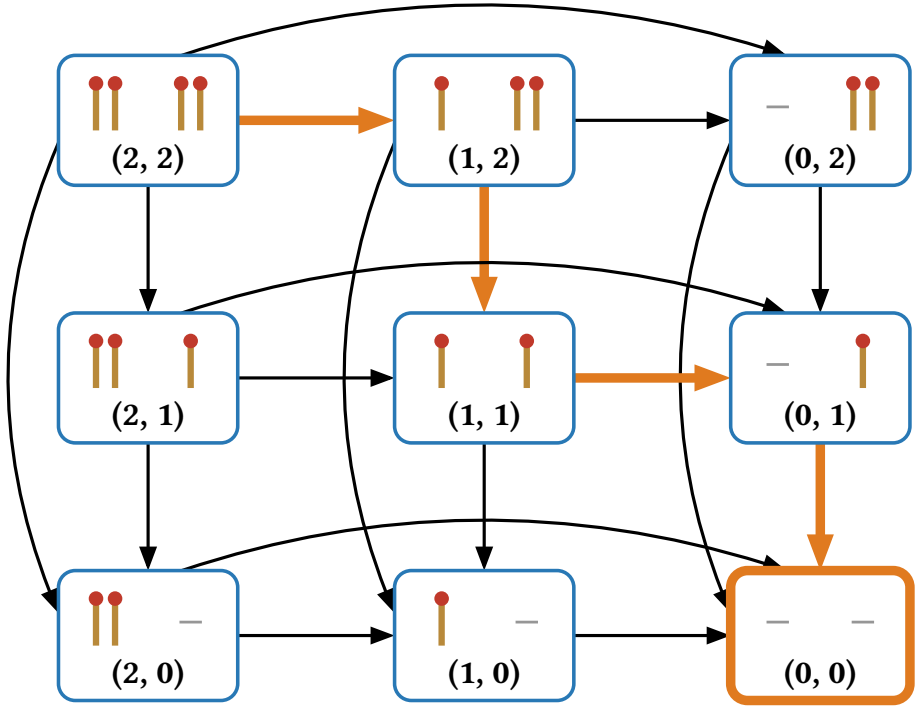

premier joueur : la dernière du premier tas

# Les états d'un jeu : Nim à deux tas



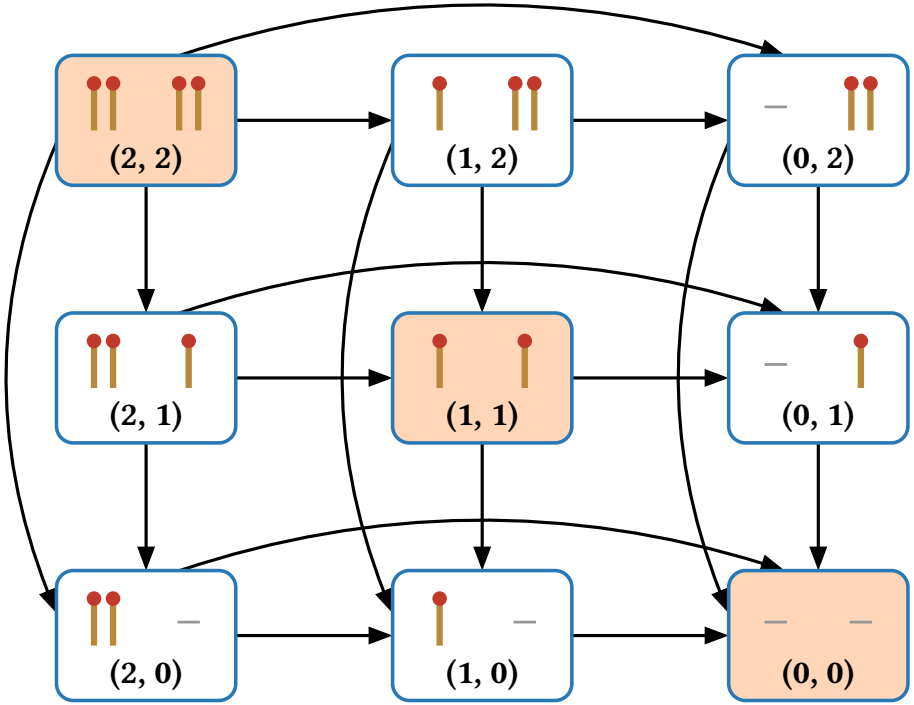

second joueur : la dernière allumette, il gagne

# Les états d'un jeu : Nim à deux tas



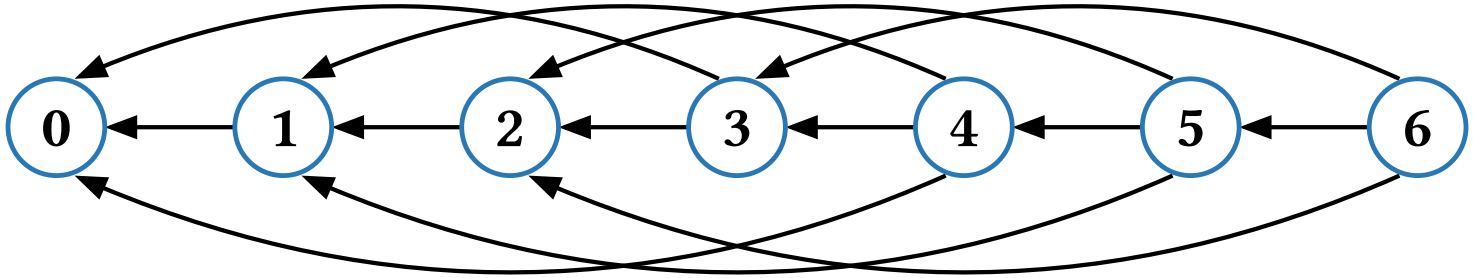

9 états, 18 arêtes

# Les états d'un jeu : Nim à deux tas




en orange, tas égaux : celui qui doit jouer **perd**, car l'adversaire peut toujours rétablir l'égalité

# Programmation dynamique : le rendu de monnaie

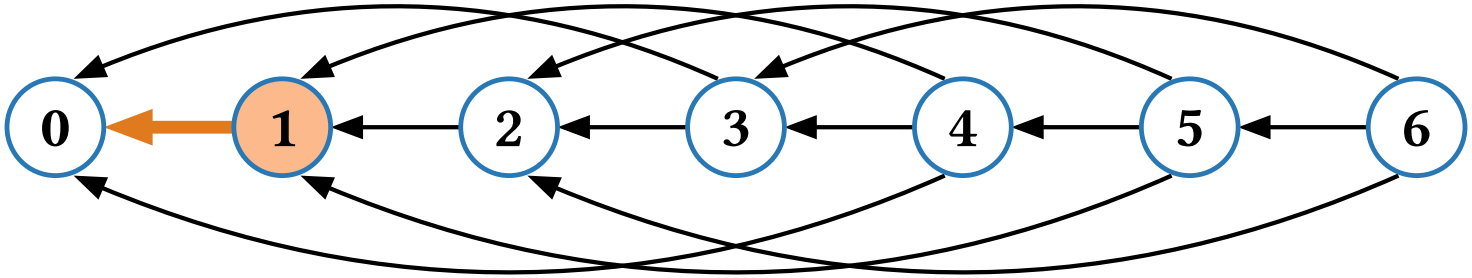



**0**

sommet $k$ : la somme à rendre  une arête de $k$ vers $k-1$, $k-3$, $k-4$ : les sommes dont elle dépend

sous chaque sommet : $f(k)$, le nombre minimal de pièces (pièces 1, 3, 4)

# Programmation dynamique : le rendu de monnaie

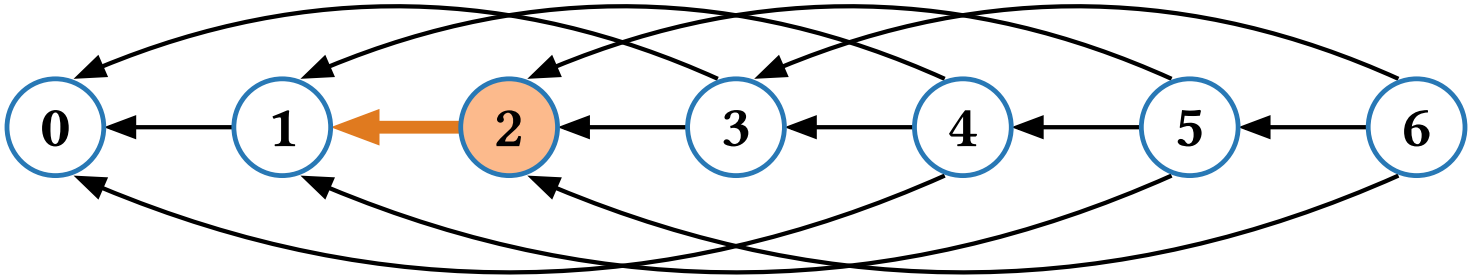



**0** **1**

$$f(1) = 1 + \min(f(0)) = 1$$

sous chaque sommet : $f(k)$, le nombre minimal de pièces (pièces 1, 3, 4)

# Programmation dynamique : le rendu de monnaie

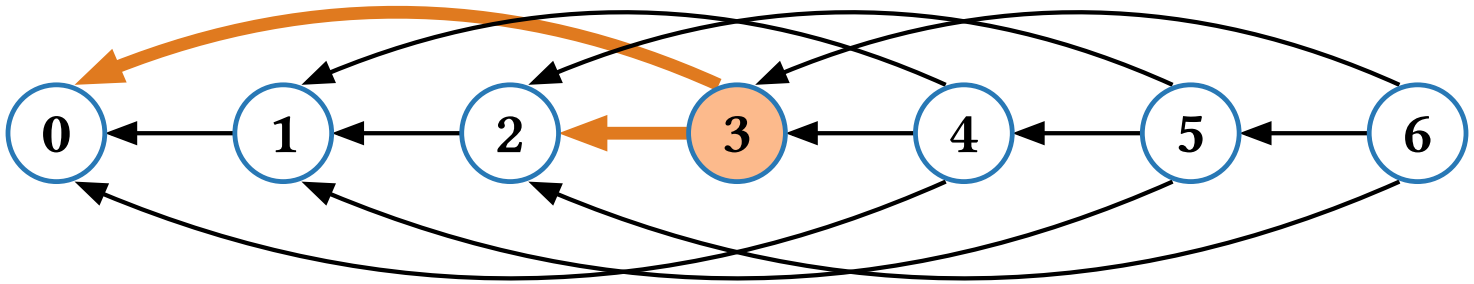



**0** **1** **2**

$$f(2) = 1 + \min(f(1)) = 2$$

sous chaque sommet : $f(k)$, le nombre minimal de pièces (pièces 1, 3, 4)

# Programmation dynamique : le rendu de monnaie

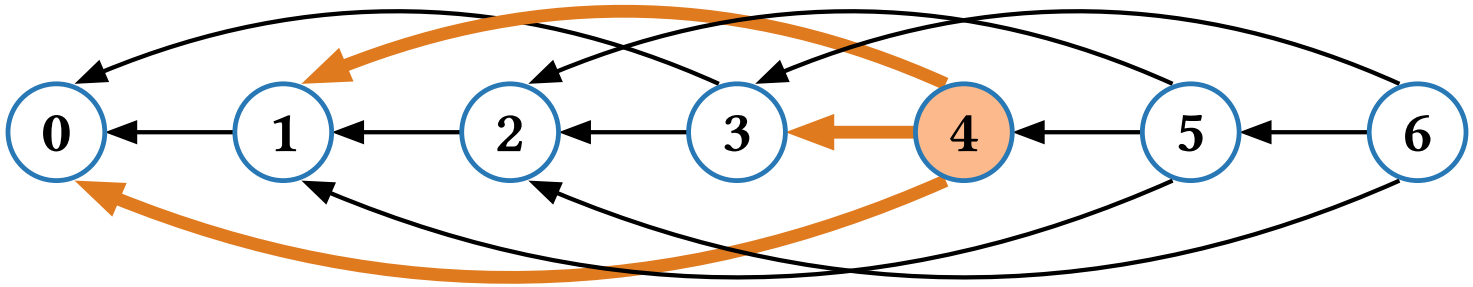



**0** **1** **2** **1**

$$f(3) = 1 + \min(f(2), f(0)) = 1$$

sous chaque sommet : $f(k)$, le nombre minimal de pièces (pièces 1, 3, 4)

# Programmation dynamique : le rendu de monnaie

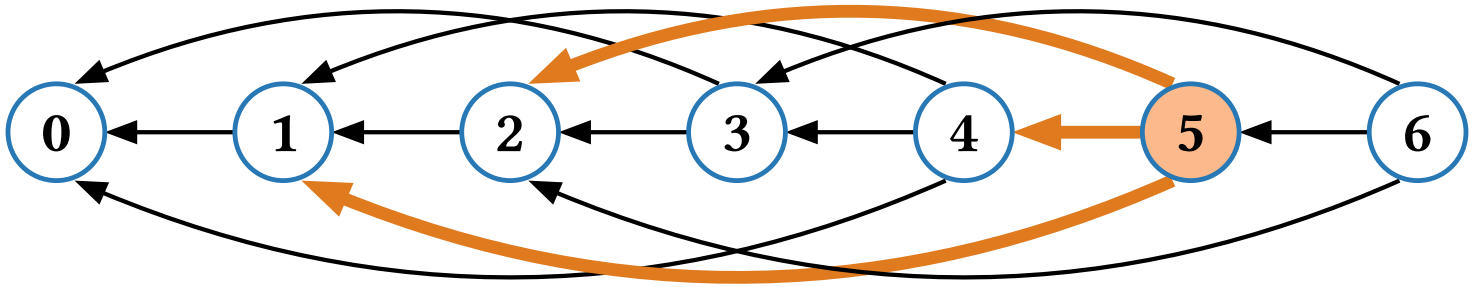



**0** **1** **2** **1** **1**

$$f(4) = 1 + \min(f(3), f(1), f(0)) = 1$$

sous chaque sommet : $f(k)$, le nombre minimal de pièces (pièces 1, 3, 4)

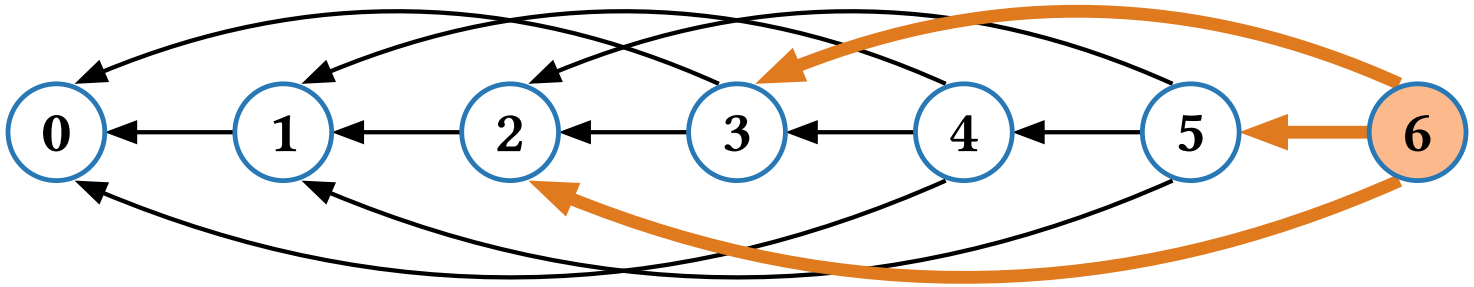
# Programmation dynamique : le rendu de monnaie




$$f(5) = 1 + \min(f(4), f(2), f(1)) = 2$$

sous chaque sommet : $f(k)$, le nombre minimal de pièces (pièces 1, 3, 4)

# Programmation dynamique : le rendu de monnaie

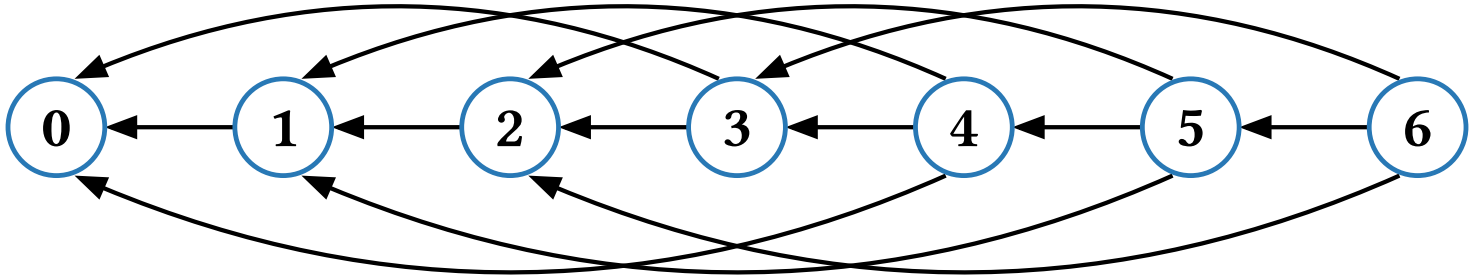



**0** **1** **2** **1** **1** **2** **2**

$$f(6) = 1 + \min(f(5), f(3), f(2)) = 2$$

sous chaque sommet : $f(k)$, le nombre minimal de pièces (pièces 1, 3, 4)

# Programmation dynamique : le rendu de monnaie

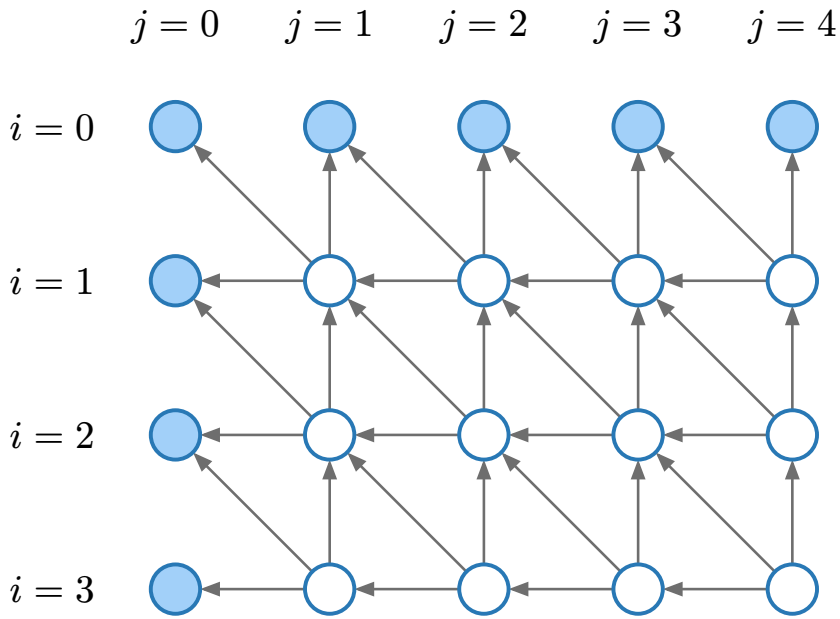



**0** **1** **2** **1** **1** **2** **2**

de gauche à droite, chaque sommet vient **après** ceux dont il dépend

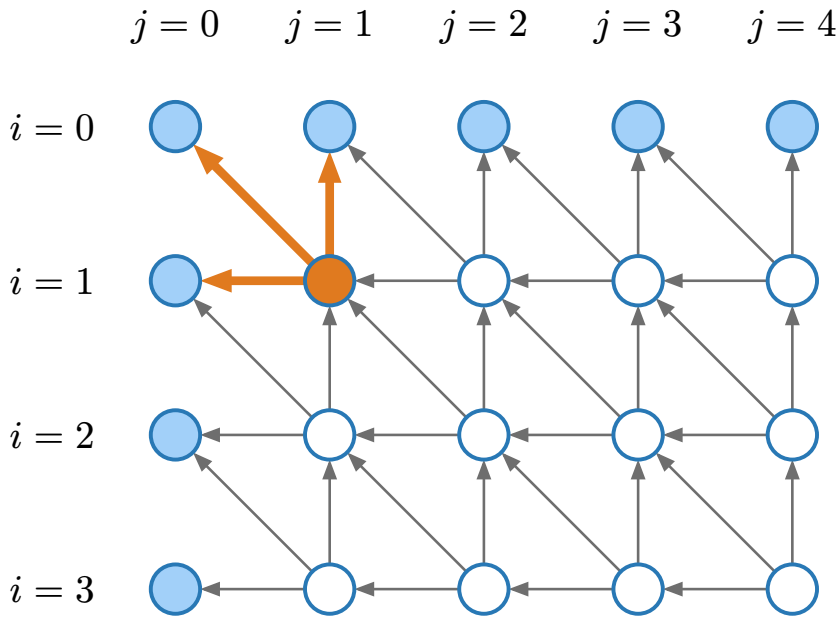
sous chaque sommet : $f(k)$, le nombre minimal de pièces (pièces 1, 3, 4)

# Programmation dynamique : la plus longue sous-suite commune



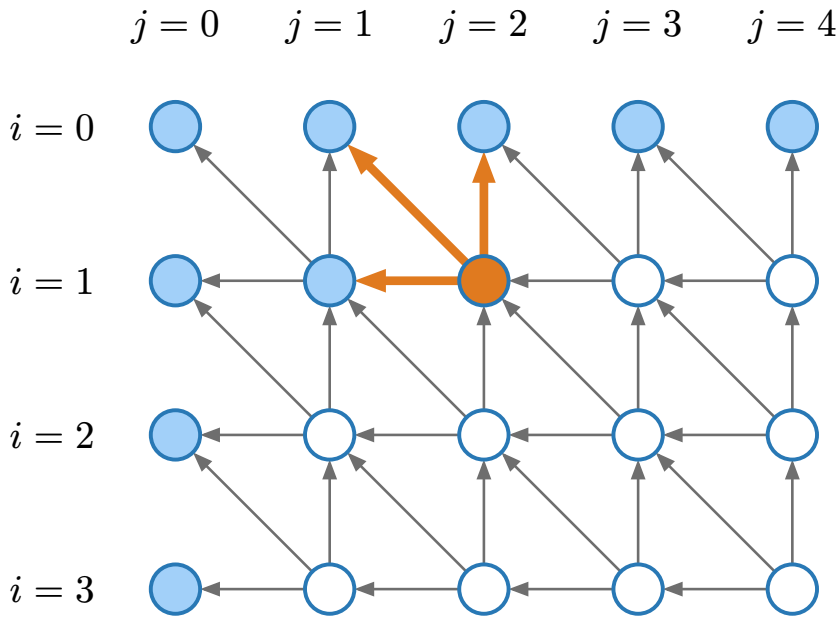

état $(i, j)$ : les préfixes de longueurs $i$ et $j$ en bleu, les cas de base

# Programmation dynamique : la plus longue sous-suite commune



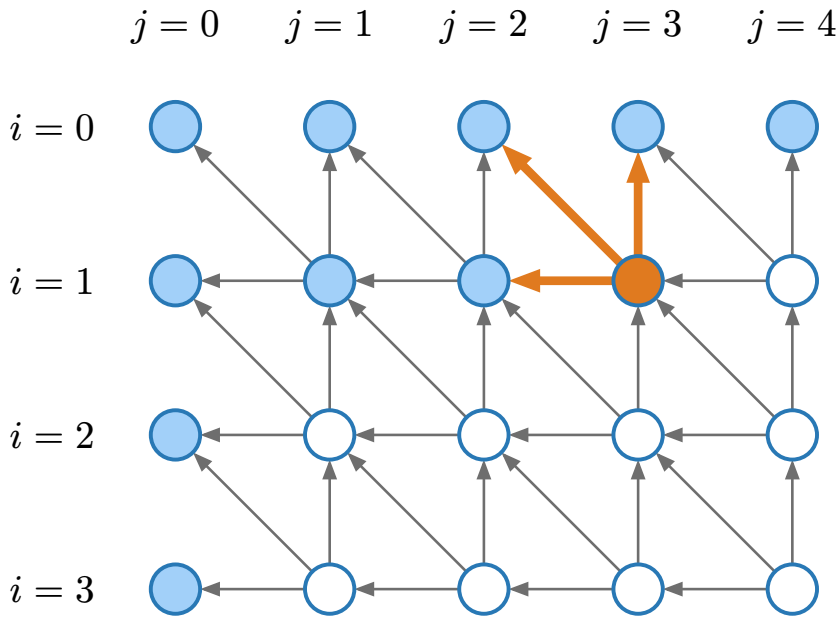

$L(1, 1)$ dépend de ses voisines du haut, de gauche et en diagonale, déjà calculées

# Programmation dynamique : la plus longue sous-suite commune



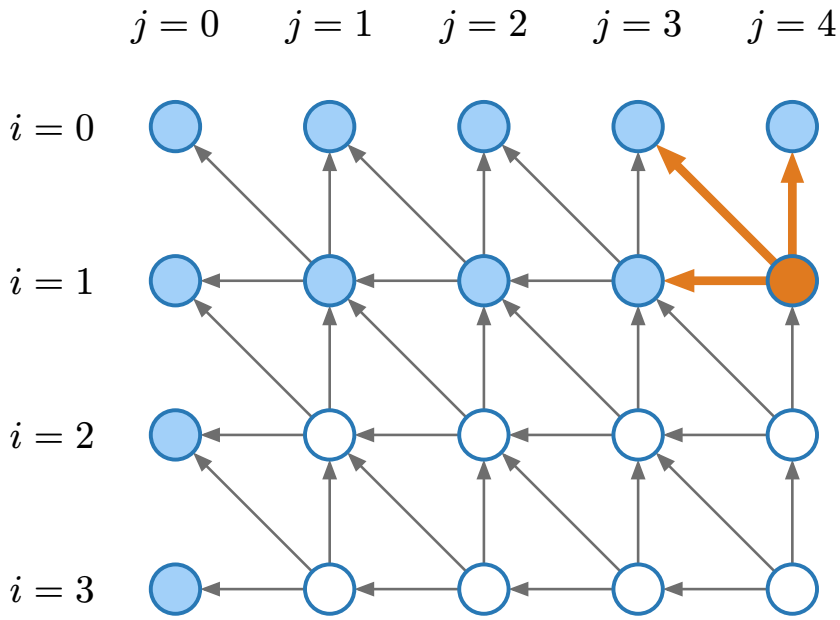

$L(1, 2)$ dépend de ses voisines du haut, de gauche et en diagonale, déjà calculées

# Programmation dynamique : la plus longue sous-suite commune



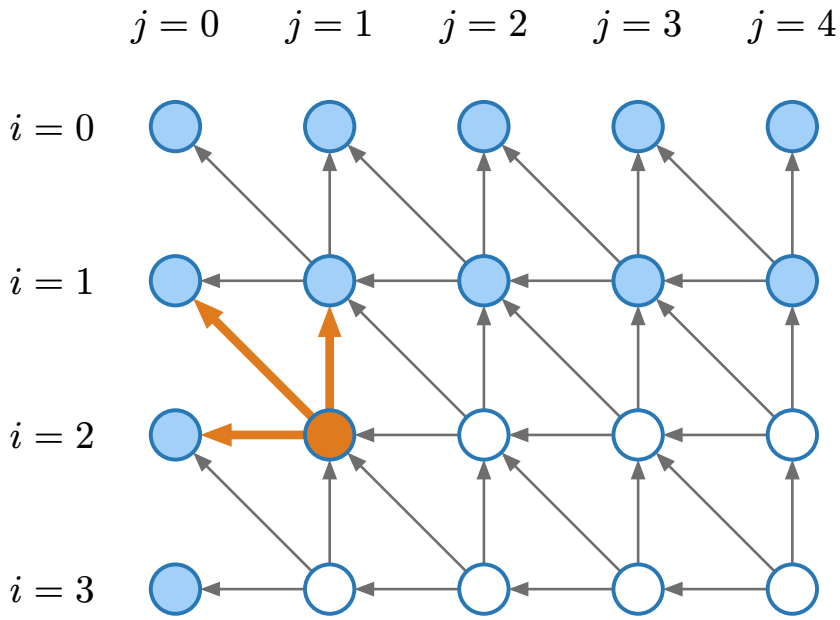

$L(1, 3)$ dépend de ses voisines du haut, de gauche et en diagonale, déjà calculées

# Programmation dynamique : la plus longue sous-suite commune



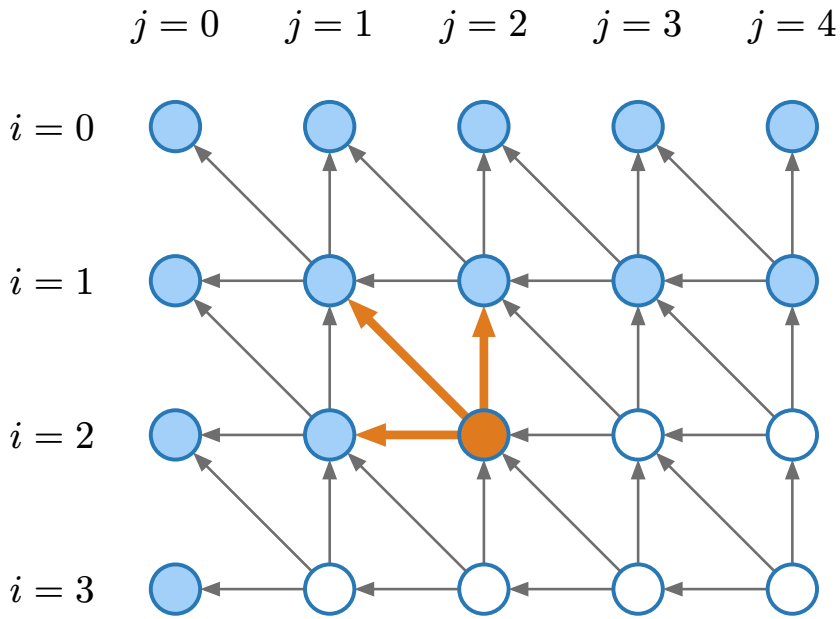

$L(1, 4)$ dépend de ses voisines du haut, de gauche et en diagonale, déjà calculées

# Programmation dynamique : la plus longue sous-suite commune



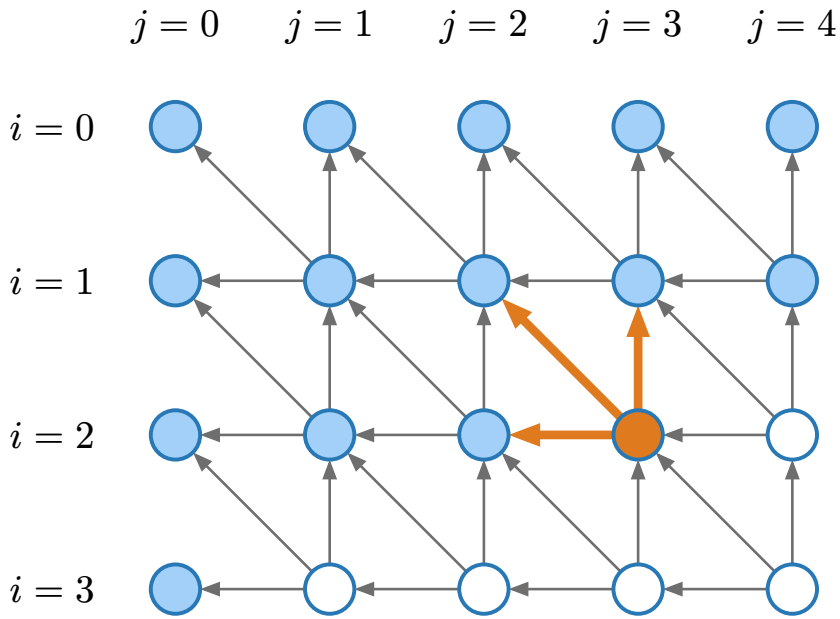

$L(2, 1)$ dépend de ses voisines du haut, de gauche et en diagonale, déjà calculées

# Programmation dynamique : la plus longue sous-suite commune



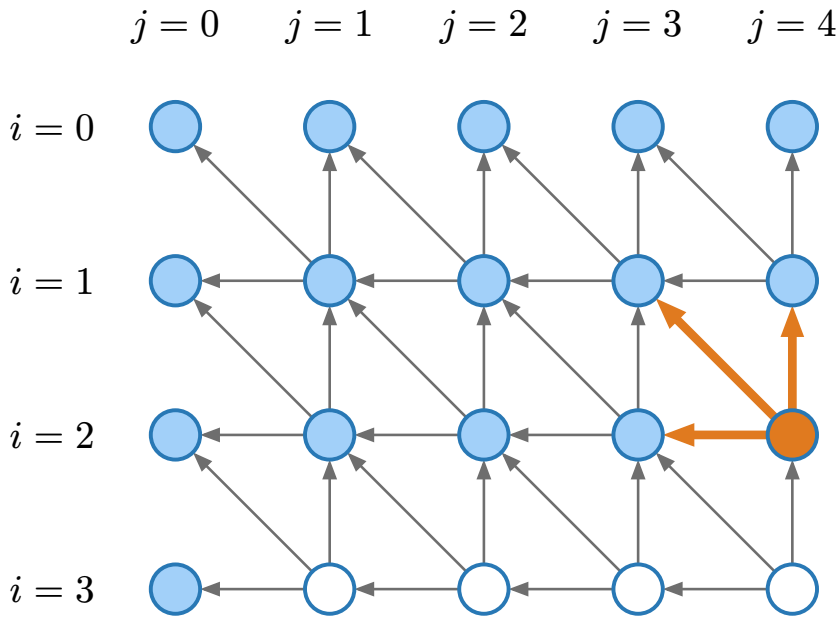

$L(2, 2)$ dépend de ses voisines du haut, de gauche et en diagonale, déjà calculées

# Programmation dynamique : la plus longue sous-suite commune



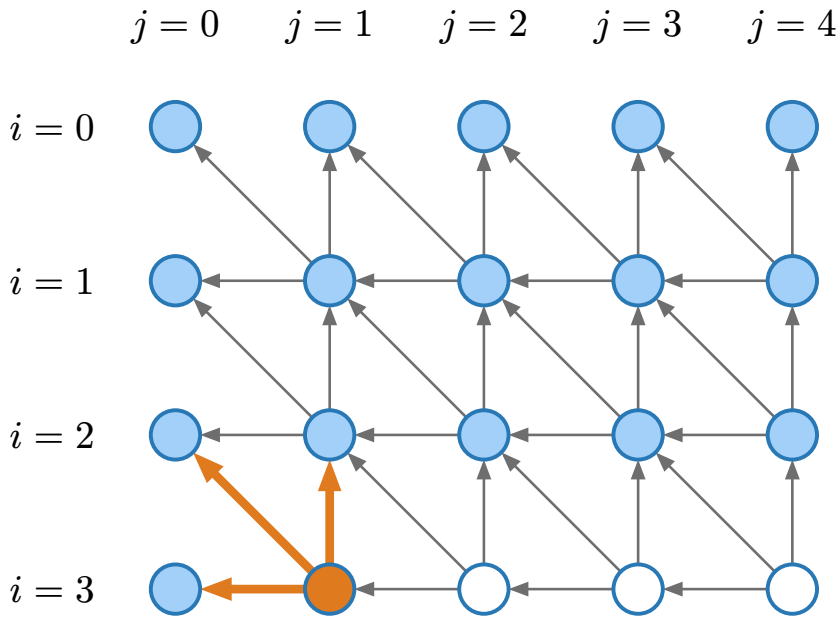

$L(2, 3)$ dépend de ses voisines du haut, de gauche et en diagonale, déjà calculées

# Programmation dynamique : la plus longue sous-suite commune



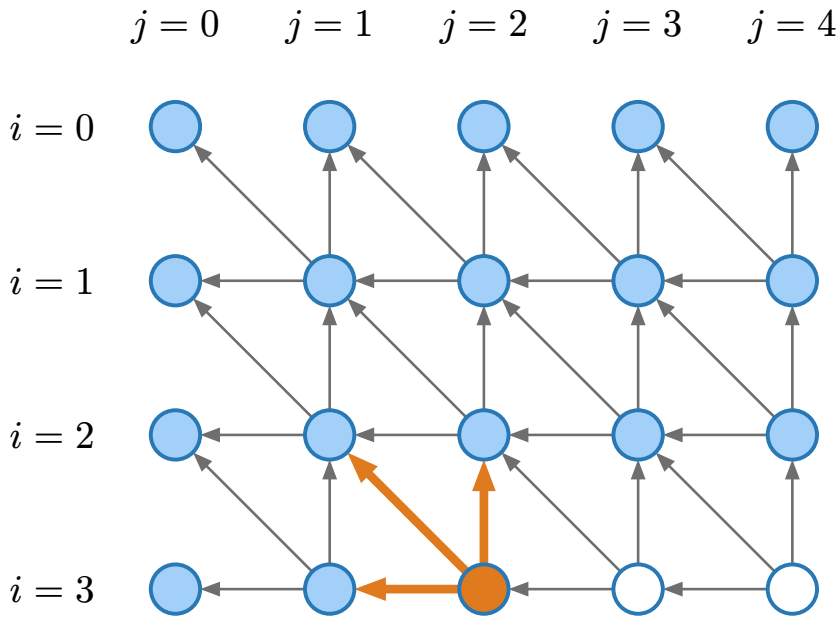

$L(2, 4)$ dépend de ses voisines du haut, de gauche et en diagonale, déjà calculées

# Programmation dynamique : la plus longue sous-suite commune



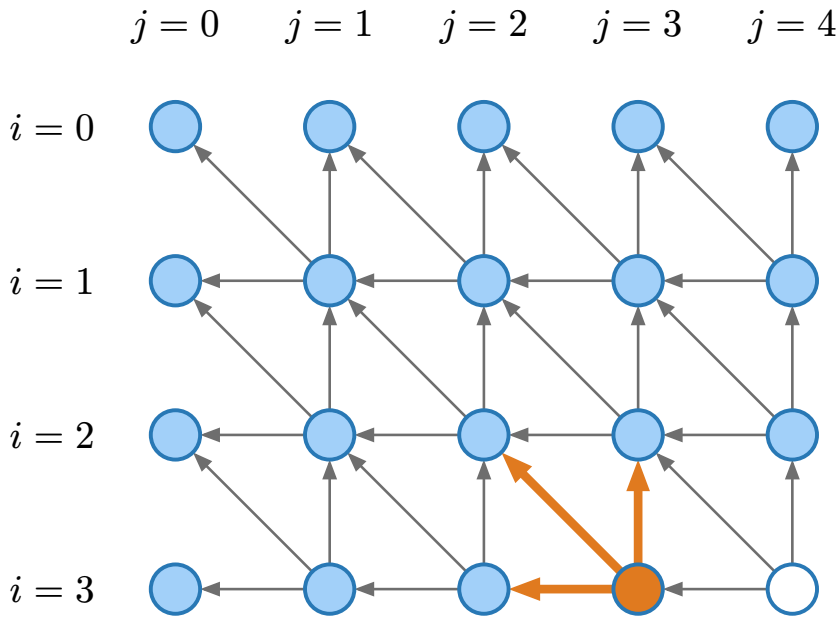

$L(3, 1)$ dépend de ses voisines du haut, de gauche et en diagonale, déjà calculées

# Programmation dynamique : la plus longue sous-suite commune



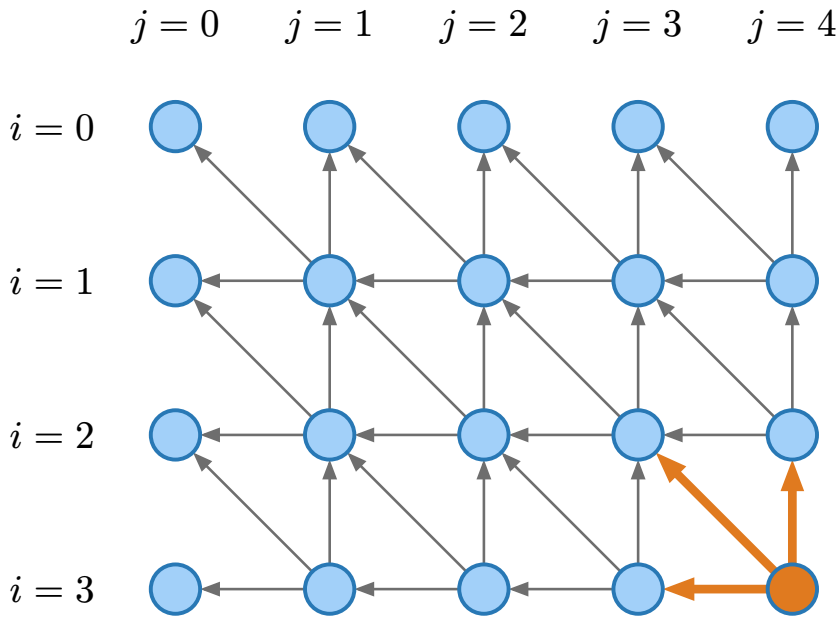

$L(3, 2)$ dépend de ses voisines du haut, de gauche et en diagonale, déjà calculées

# Programmation dynamique : la plus longue sous-suite commune




$L(3, 3)$ dépend de ses voisines du haut, de gauche et en diagonale, déjà calculées

# Programmation dynamique : la plus longue sous-suite commune




ordre lexicographique : chaque case est calculée **après** celles dont elle dépend

## La première question devant un énoncé

**Quels sont les sommets ? Quelles sont les arêtes ?**
**Comment obtient-on les voisins d'un sommet ?**

# Parcourir un graphe

# Un seul schéma (Python)

```python
def parcours_sac(G, depart):
    vus = [False for s in range(len(G))]
    ordre = []
    sac = sac_vide()
    ajouter(sac, depart)
    while len(sac) > 0:
        s = retirer(sac)
        if not vus[s]:
            vus[s] = True
            ordre.append(s)
            for t in G[s]:
                ajouter(sac, t)
    return ordre
```

**avec une pile**

```python
def sac_vide():
    return []

def ajouter(sac, s):
    sac.append(s)

def retirer(sac):
    return sac.pop()
```

seule liberté : **quel sommet** sort du sac

un sommet peut y entrer plusieurs fois : test **à la sortie**

# Un seul schéma (Python)

```python
def parcours_sac(G, depart):
    vus = [False for s in range(len(G))]
    ordre = []
    sac = sac_vide()
    ajouter(sac, depart)
    while len(sac) > 0:
        s = retirer(sac)
        if not vus[s]:
            vus[s] = True
            ordre.append(s)
            for t in G[s]:
                ajouter(sac, t)
    return ordre
```

**avec une file** (deque importé)

```python
def sac_vide():
    return deque()

def ajouter(sac, s):
    sac.append(s)

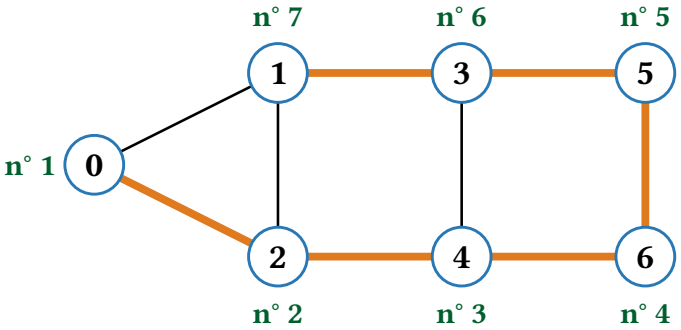
def retirer(sac):
    return sac.popleft()
```

seule liberté : **quel sommet** sort du sac

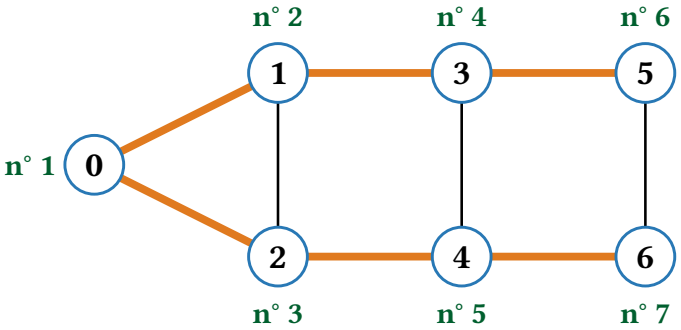
un sommet peut y entrer plusieurs fois : test **à la sortie**

# Trois sacs, trois parcours

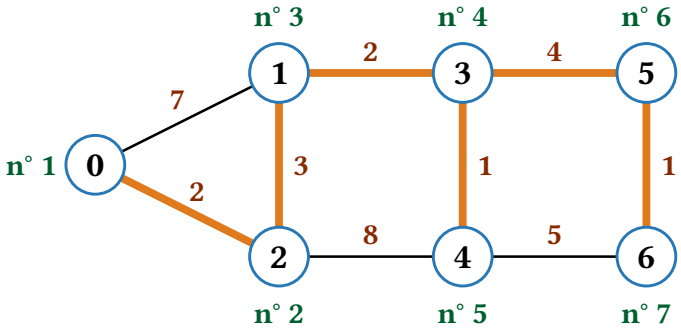
**pile : en profondeur**




**file : en largeur**




**file de priorité : Dijkstra**




numéro de visite de chaque sommet ; en gras, l'arbre du parcours

## Ce que calcule un parcours

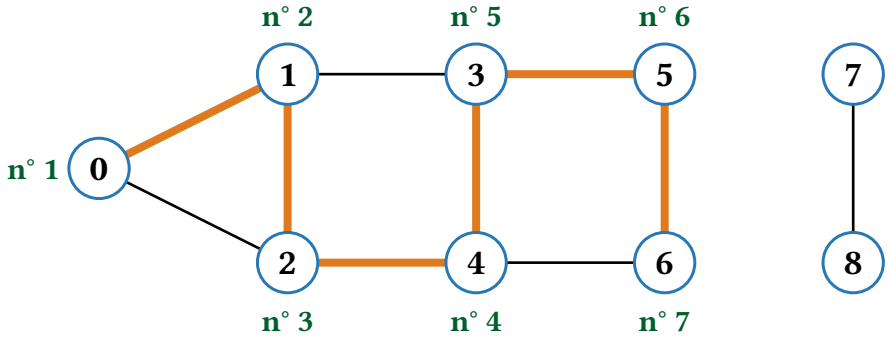
**Propriété.** Quel que soit le sac, à la fin du parcours les sommets marqués sont exactement les sommets **accessibles** depuis le départ.

- un sommet n'entre dans le sac que comme voisin d'un sommet marqué : **marqué $\Longrightarrow$ accessible**
- sur un chemin vers un sommet accessible non marqué, le premier non marqué a un prédécesseur marqué : il est entré dans le sac, donc en est sorti marqué, **contradiction**
- grâce à `vus`, chaque sommet n'est traité qu'**une seule fois** : le parcours termine
- l'ensemble visité ne dépend pas du sac ; l'**ordre** de visite, si : c'est lui qui porte l'information

# Le parcours en profondeur (Python)

```python
def profondeur_aux(G, s, vus, ordre):
    vus[s] = True
    ordre.append(s)
    for t in G[s]:
        if not vus[t]:
            profondeur_aux(G, t, vus, ordre)

def parcours_profondeur(G, depart):
    vus = [False for s in range(len(G))]
    ordre = []
    profondeur_aux(G, depart, vus, ordre)
    return ordre
```

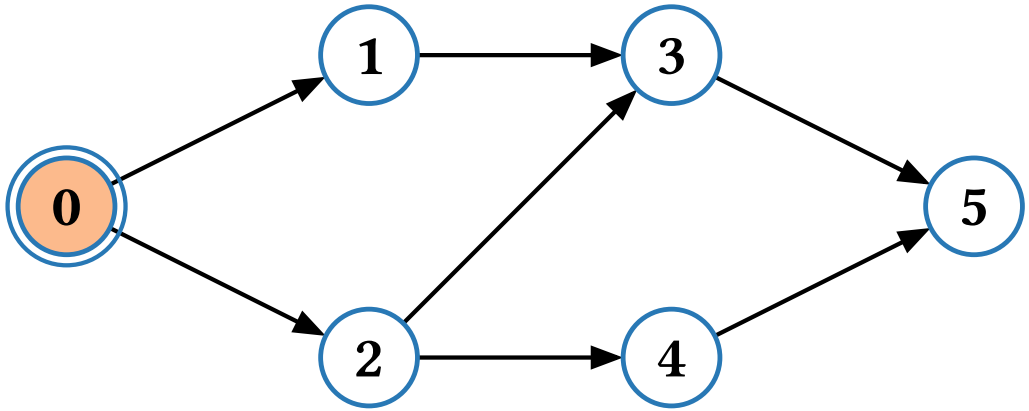



la pile est celle des **appels récursifs**

un appel par sommet, de coût $O(1 + d(s))$ :

$O(n + m)$ au total

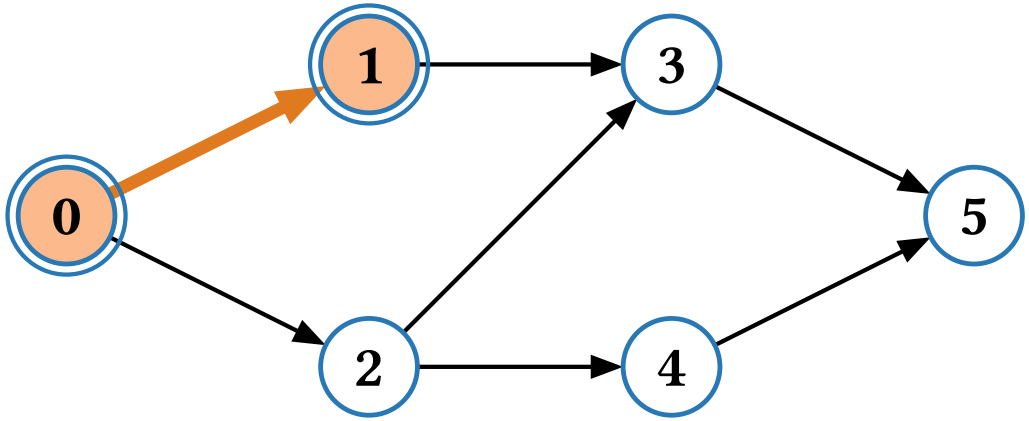
# Les appels récursifs pas à pas




appels en cours : `[0]`
ordre : `[0]`

appel sur 0

# Les appels récursifs pas à pas

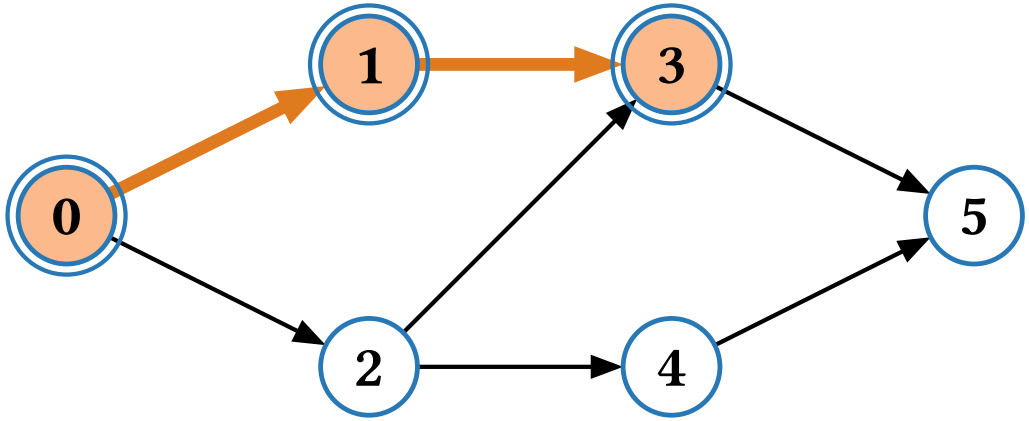



appels en cours : `[0, 1]`

ordre : `[0, 1]`

appel sur 1

# Les appels récursifs pas à pas

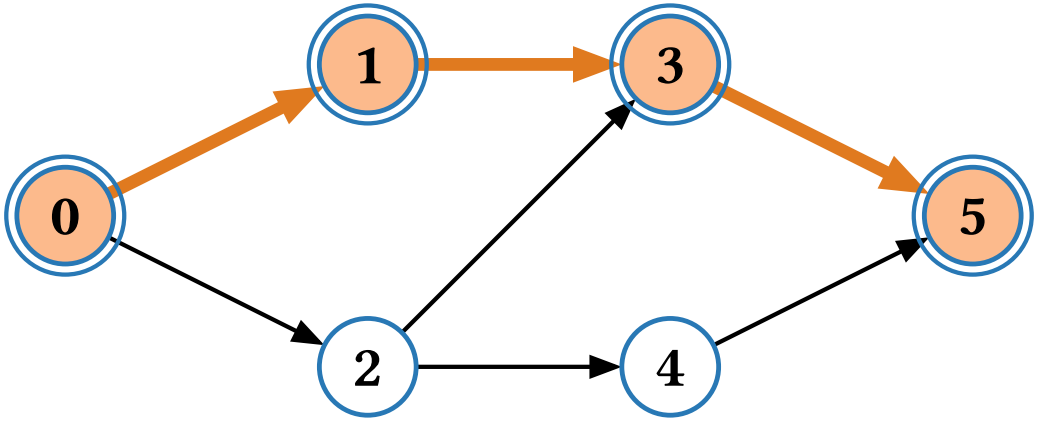



appels en cours : `[0, 1, 3]`

ordre : `[0, 1, 3]`

appel sur 3

# Les appels récursifs pas à pas

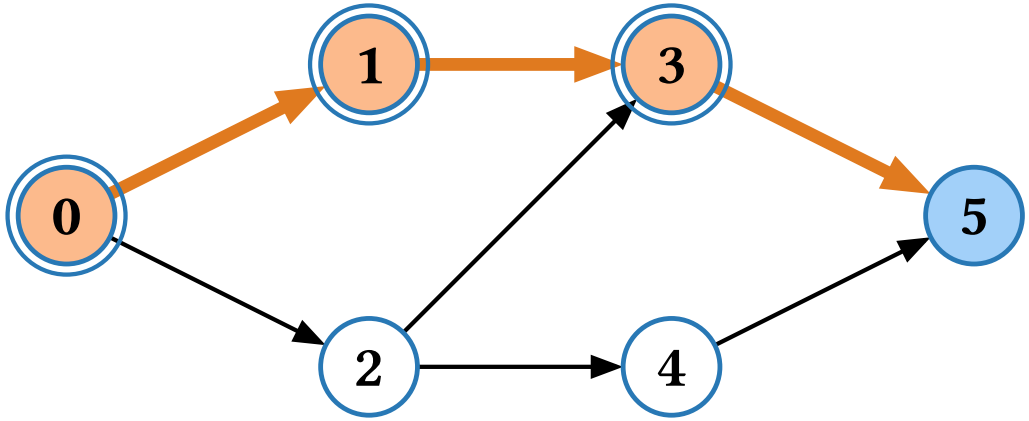



appels en cours : `[0, 1, 3, 5]`

ordre : `[0, 1, 3, 5]`

appel sur 5

# Les appels récursifs pas à pas

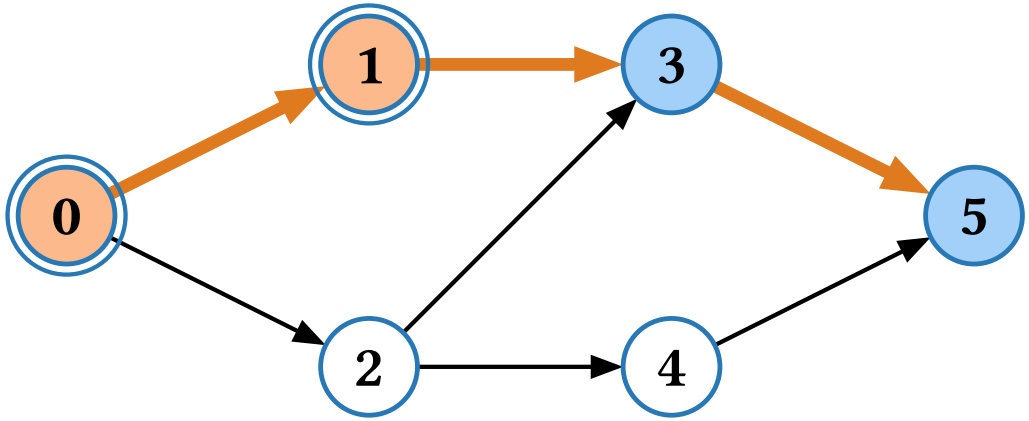



appels en cours : `[0, 1, 3]`

ordre : `[0, 1, 3, 5]`

fin de l'appel sur 5 : on revient en 3

# Les appels récursifs pas à pas

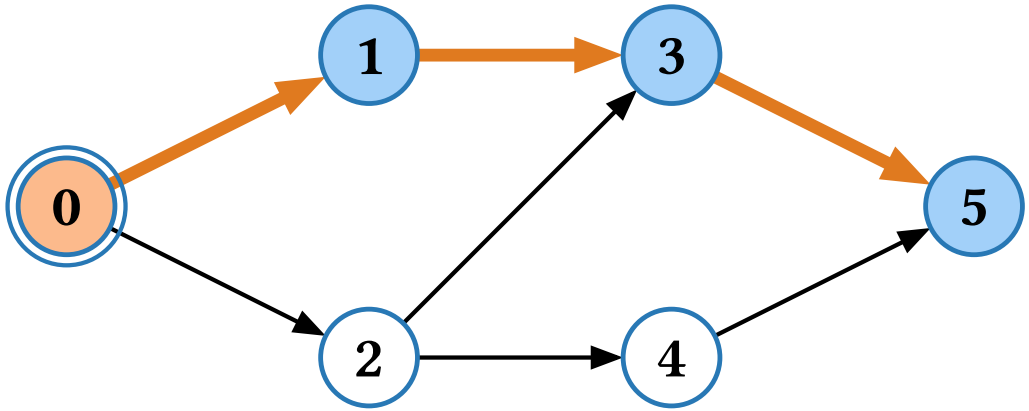



appels en cours : `[0, 1]`

ordre : `[0, 1, 3, 5]`

fin de l'appel sur 3 : on revient en 1

# Les appels récursifs pas à pas

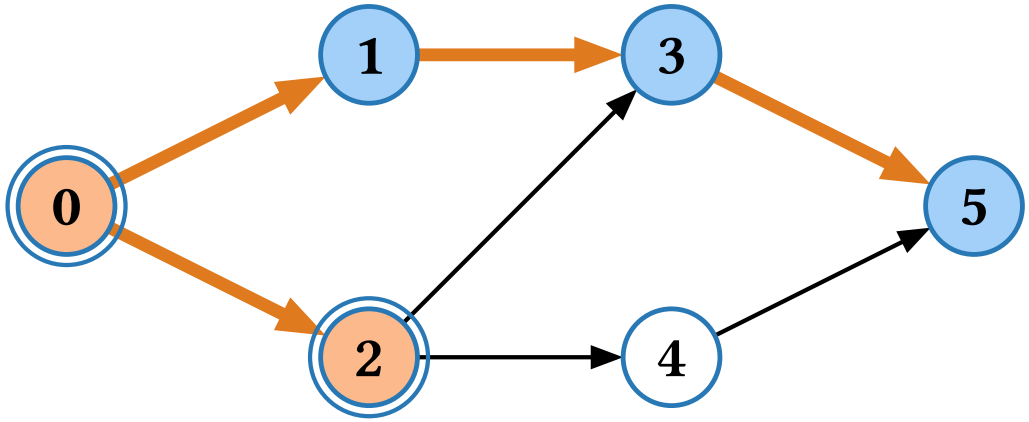



appels en cours : `[0]`

ordre : `[0, 1, 3, 5]`

fin de l'appel sur 1 : on revient en 0

# Les appels récursifs pas à pas

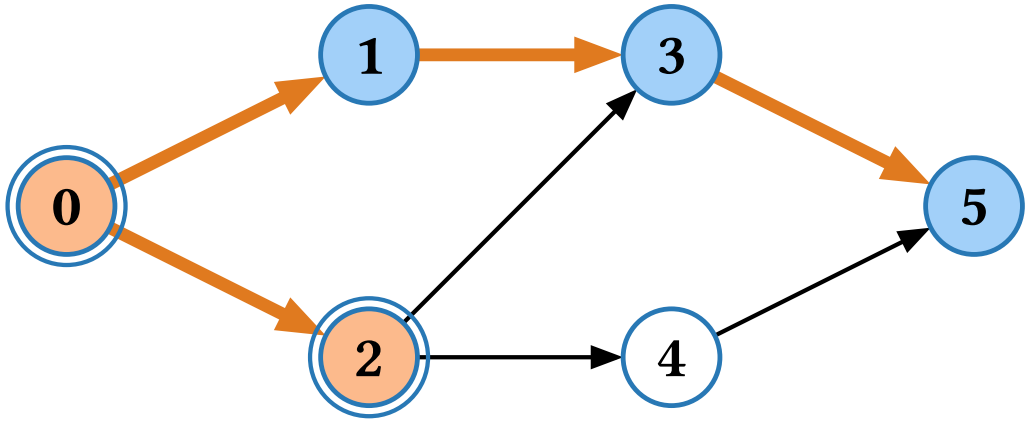



appels en cours : `[0, 2]`

ordre : `[0, 1, 3, 5, 2]`

appel sur 2

# Les appels récursifs pas à pas

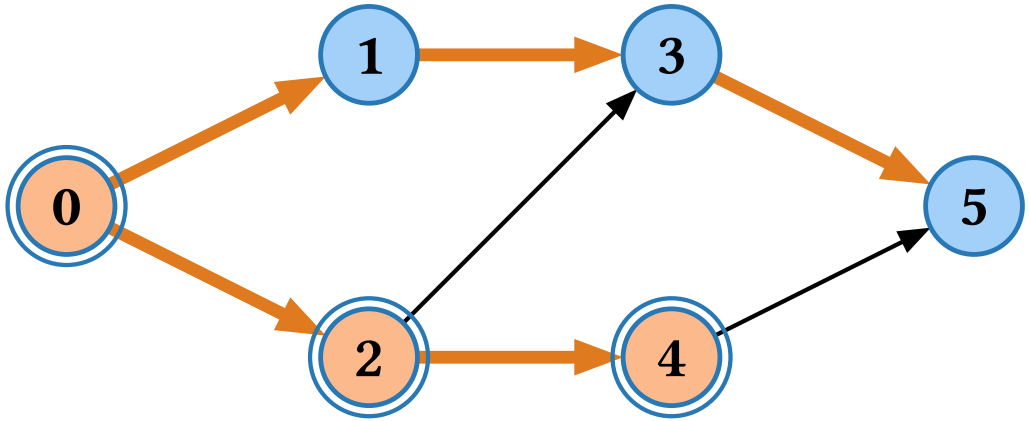



appels en cours : `[0, 2]`

ordre : `[0, 1, 3, 5, 2]`

2 examine son voisin 3 : déjà vu

# Les appels récursifs pas à pas

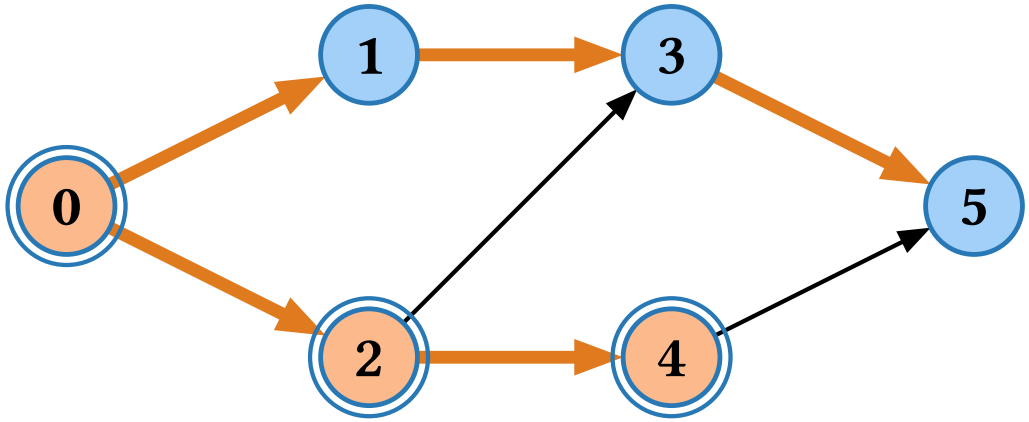



appels en cours : `[0, 2, 4]`

ordre : `[0, 1, 3, 5, 2, 4]`

appel sur 4

# Les appels récursifs pas à pas

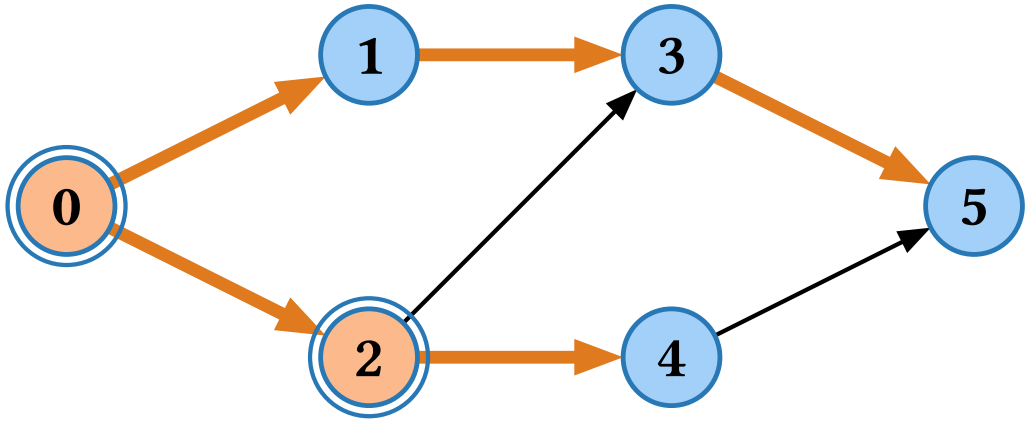



appels en cours : `[0, 2, 4]`

ordre : `[0, 1, 3, 5, 2, 4]`

4 examine son voisin 5 : déjà vu

# Les appels récursifs pas à pas

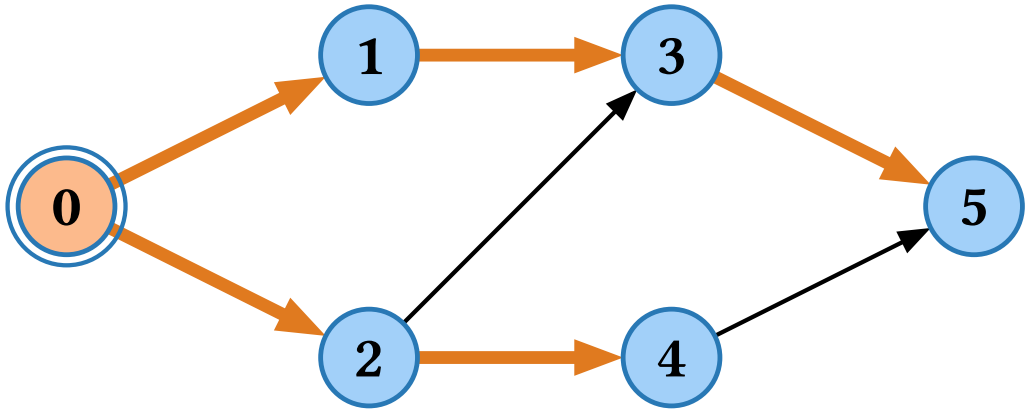



appels en cours : `[0, 2]`

ordre : `[0, 1, 3, 5, 2, 4]`

fin de l'appel sur 4 : on revient en 2

# Les appels récursifs pas à pas

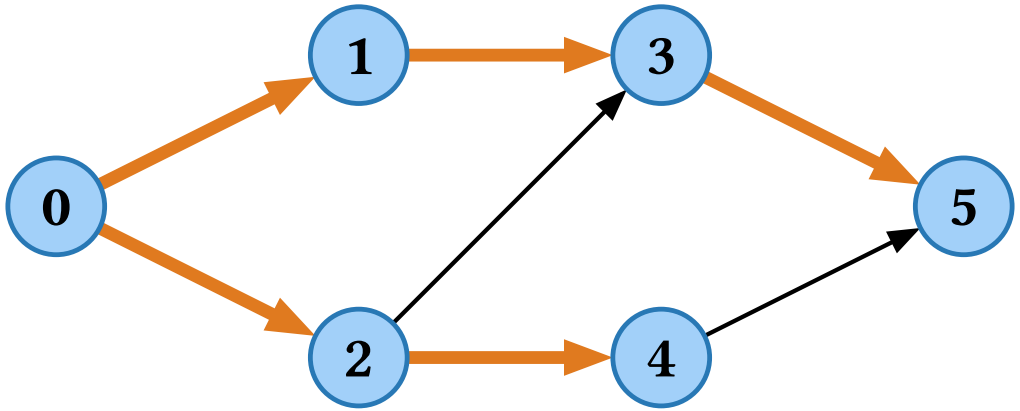



appels en cours : `[0]`

ordre : `[0, 1, 3, 5, 2, 4]`

fin de l'appel sur 2 : on revient en 0

# Les appels récursifs pas à pas



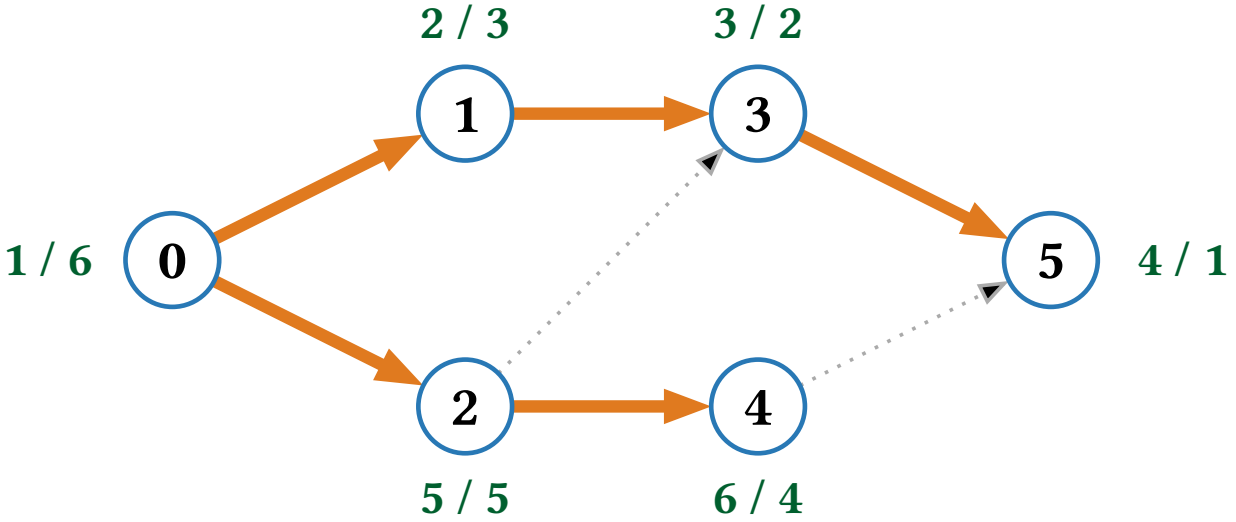

appels en cours : `[]`

ordre : `[0, 1, 3, 5, 2, 4]`

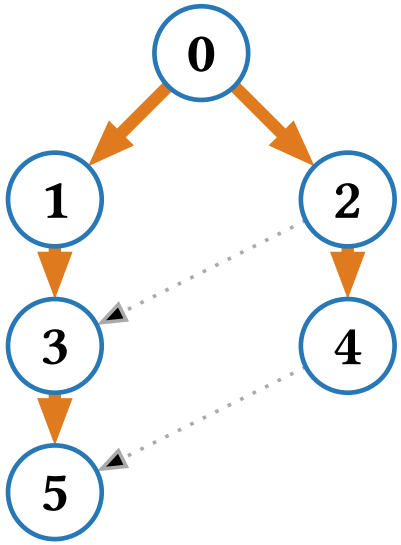
fin de l'appel sur 0 : terminé

# Revenir en arrière

## le graphe : entrée / sortie




## l'arbre des appels



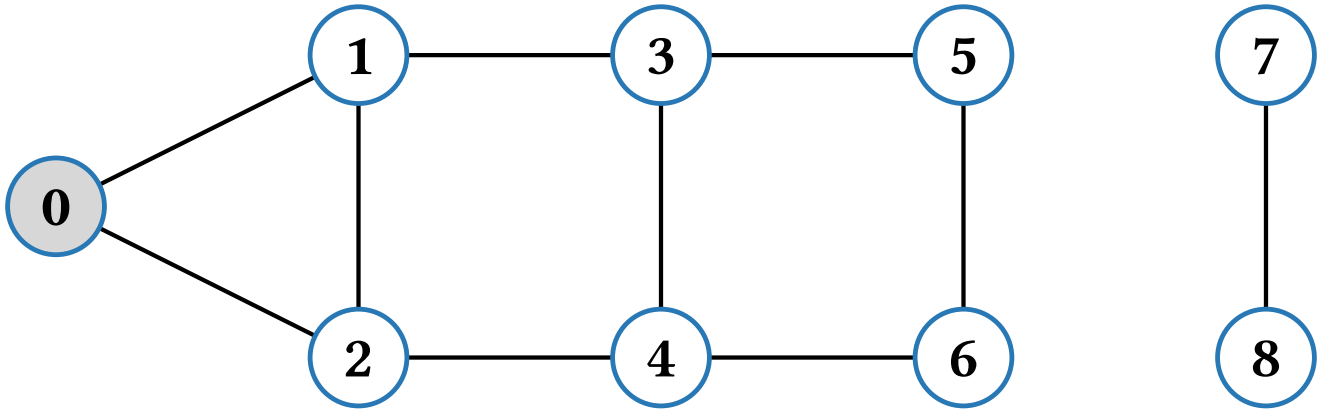

rang d'entrée / rang de sortie ; à droite, l'arbre des appels

# Le parcours en profondeur avec une pile (Python)

```python
def parcours_pile(G, depart):
    vus = [False for s in range(len(G))]
    ordre = []
    pile = [depart]
    while len(pile) > 0:
        s = pile.pop()
        if not vus[s]:
            vus[s] = True
            ordre.append(s)
            for t in G[s]:
                if not vus[t]:
                    pile.append(t)
    return ordre
```

un sommet peut être **plusieurs fois** dans la pile, d'où le test à la sortie ; au plus $1 + 2m$ éléments : $O(n + m)$

# La pile pas à pas

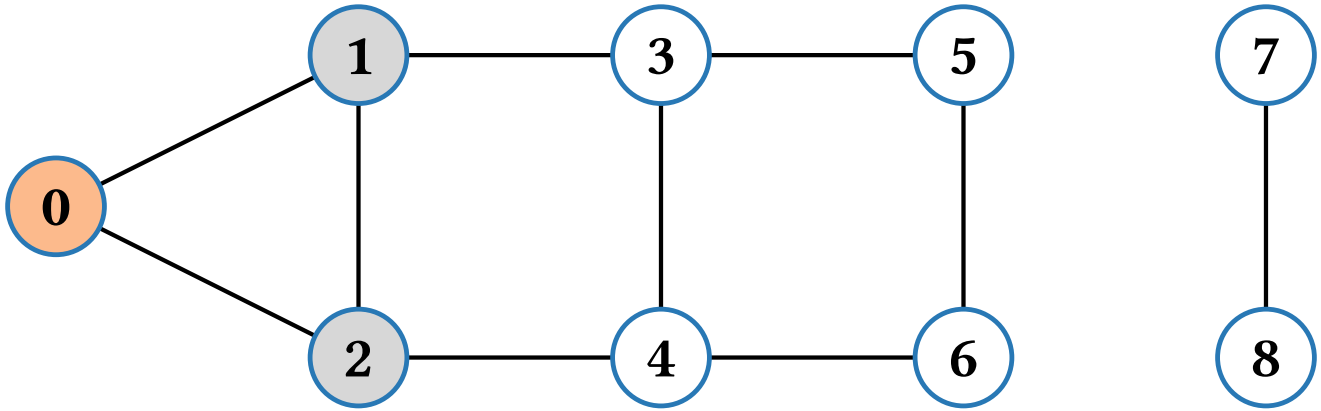



pile : `[0]`

ordre : `[]`

au début

# La pile pas à pas

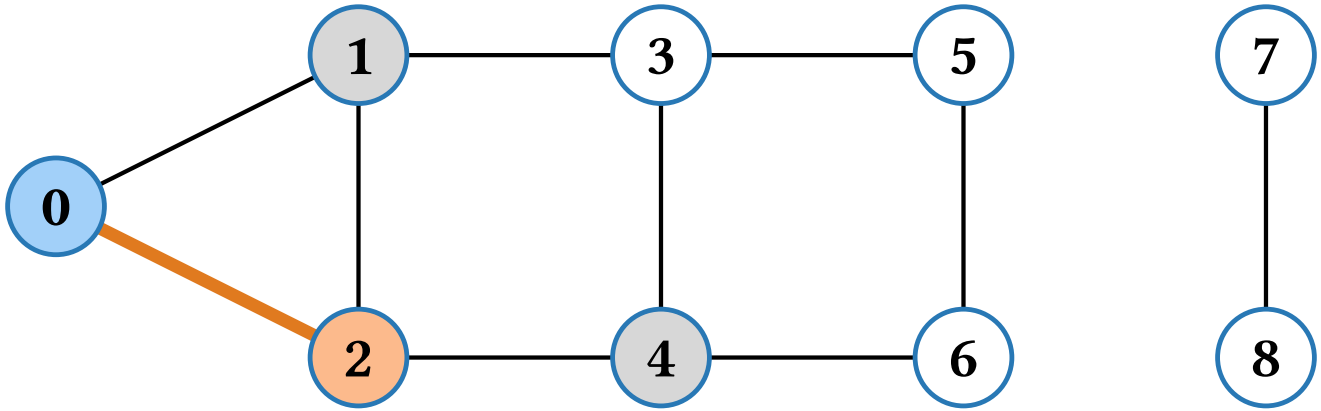



pile : `[1, 2]`

ordre : `[0]`

on sort 0

# La pile pas à pas

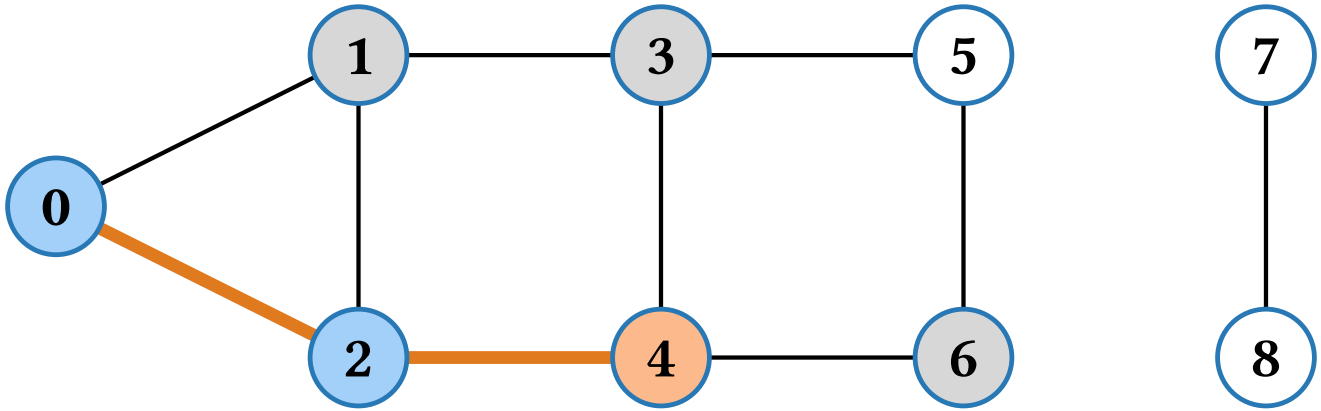



pile : `[1, 1, 4]`

ordre : `[0, 2]`

on sort 2

# La pile pas à pas

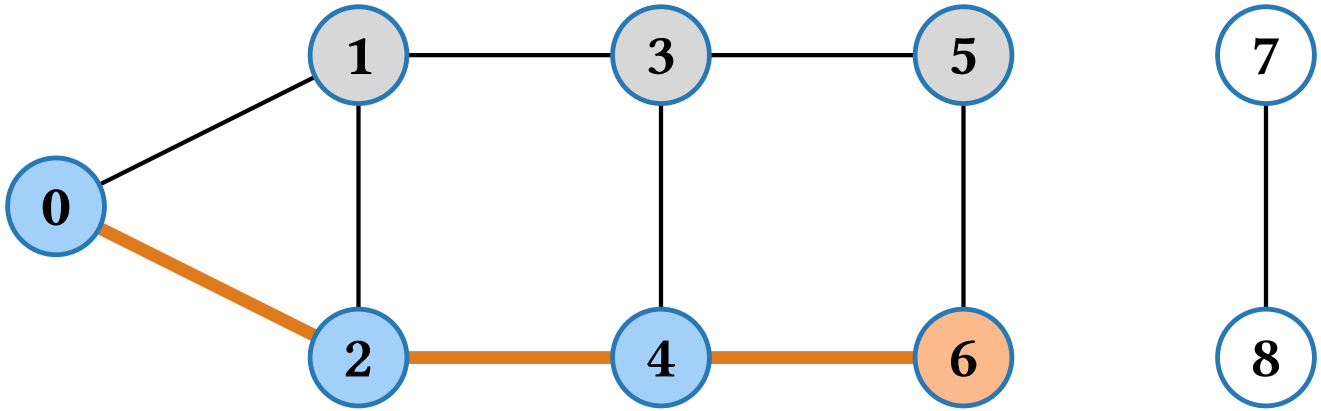



pile : `[1, 1, 3, 6]`

ordre : `[0, 2, 4]`

on sort 4

# La pile pas à pas

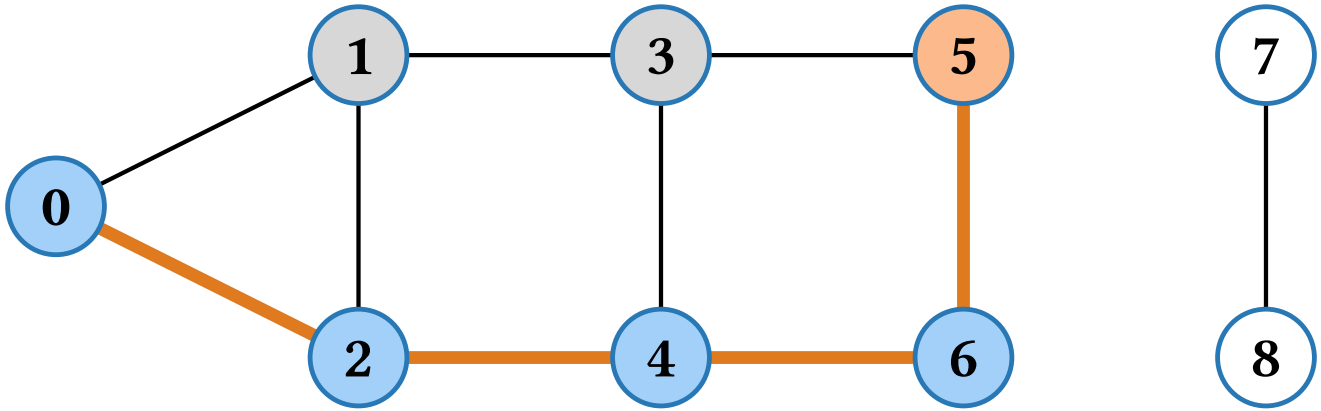



pile : `[1, 1, 3, 5]`

ordre : `[0, 2, 4, 6]`

on sort 6

# La pile pas à pas

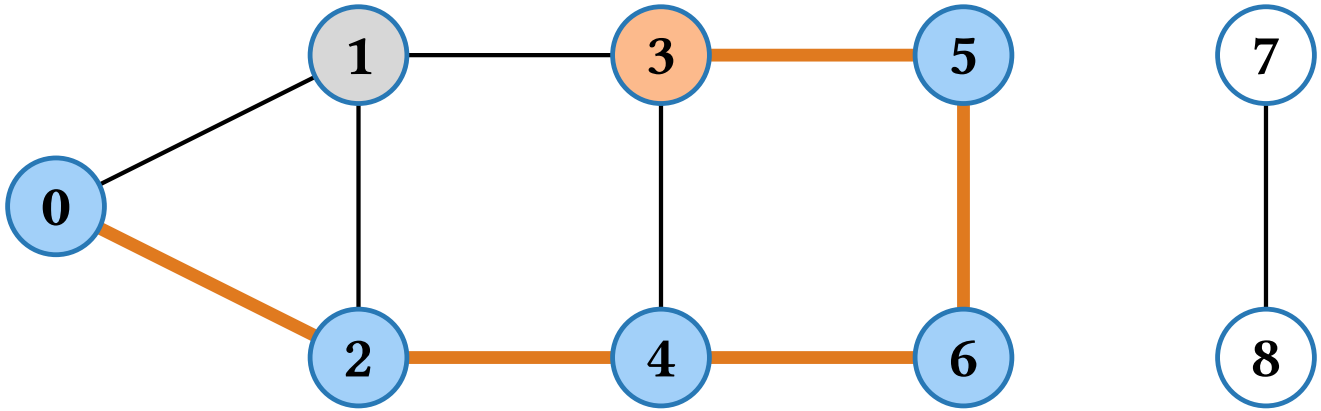



pile : `[1, 1, 3, 3]`

ordre : `[0, 2, 4, 6, 5]`

on sort 5

# La pile pas à pas

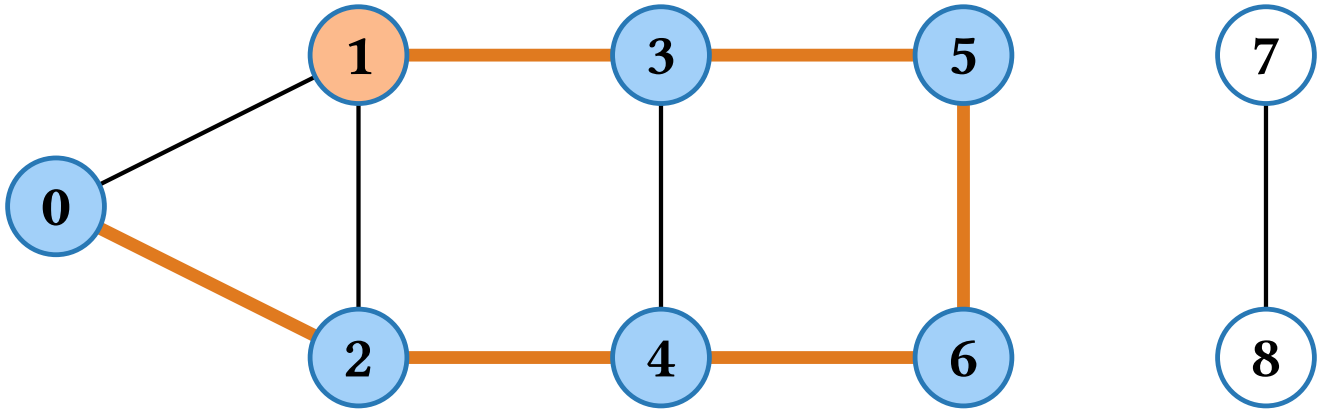



pile : `[1, 1, 3, 1]`

ordre : `[0, 2, 4, 6, 5, 3]`

on sort 3

# La pile pas à pas

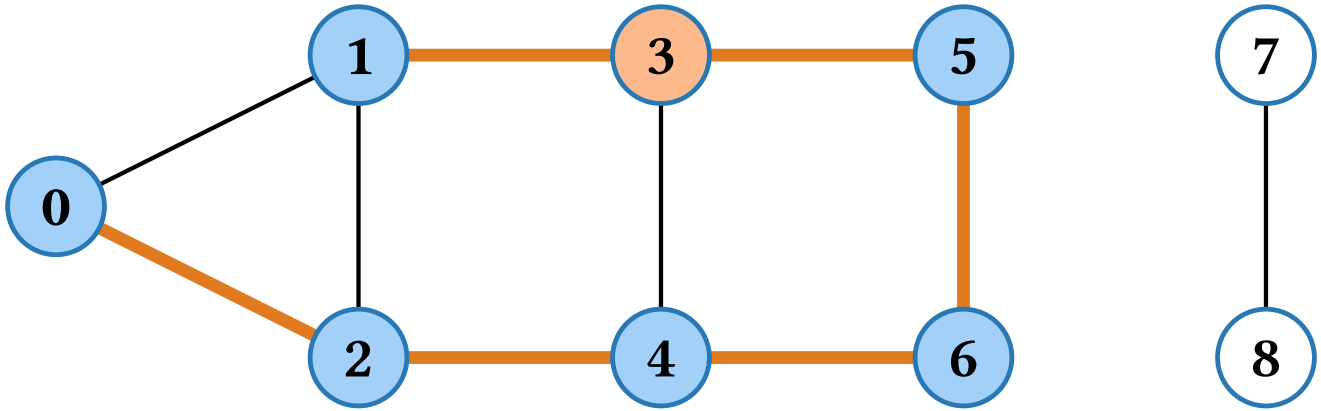



pile : `[1, 1, 3]`

ordre : `[0, 2, 4, 6, 5, 3, 1]`

on sort 1

# La pile pas à pas

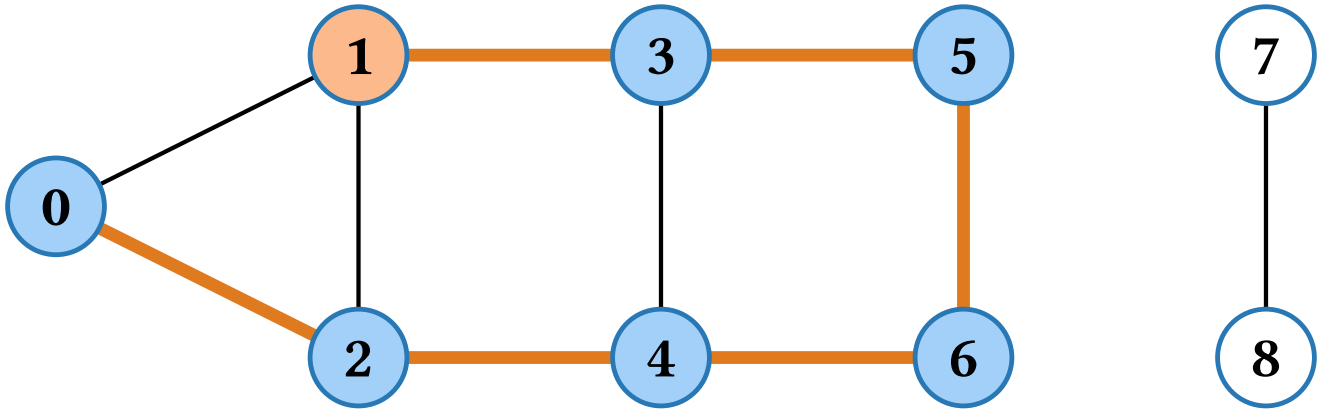



pile : `[1, 1]`

ordre : `[0, 2, 4, 6, 5, 3, 1]`

on sort 3 : déjà vu, on ne fait rien

# La pile pas à pas

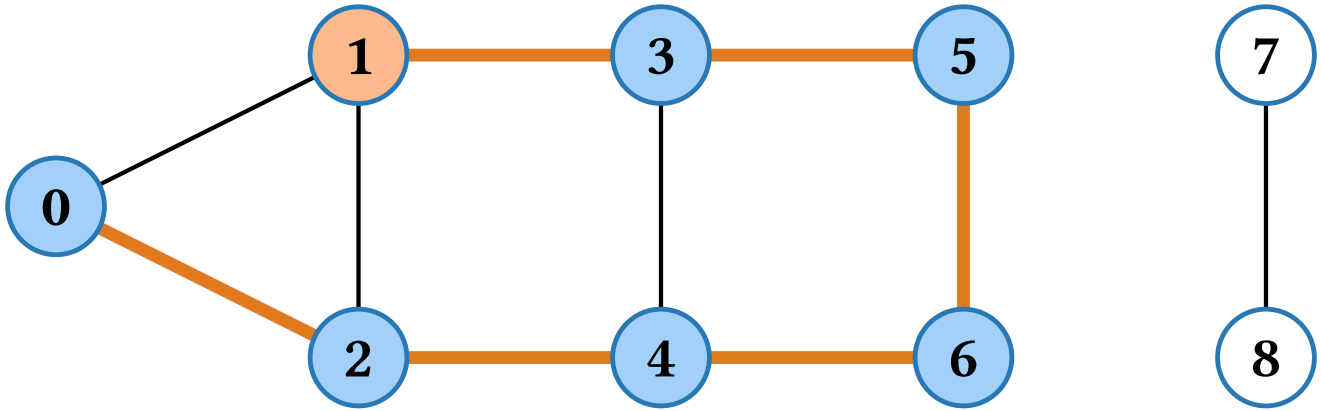



pile : `[1]`

ordre : `[0, 2, 4, 6, 5, 3, 1]`

on sort 1 : déjà vu, on ne fait rien

# La pile pas à pas




pile : `[]`

ordre : `[0, 2, 4, 6, 5, 3, 1]`

on sort 1 : déjà vu, on ne fait rien

## Attention : une liste Python est une mauvaise file

| `collections.deque` | |
|---|---|
| `from collections import deque` | en tête de programme |
| `file = deque()` | file vide |
| `file.append(x)` | ajoute `x` à la fin, en $O(1)$ |
| `file.popleft()` | retire et renvoie la tête, en $O(1)$ |
| `len(file)` | nombre d'éléments |

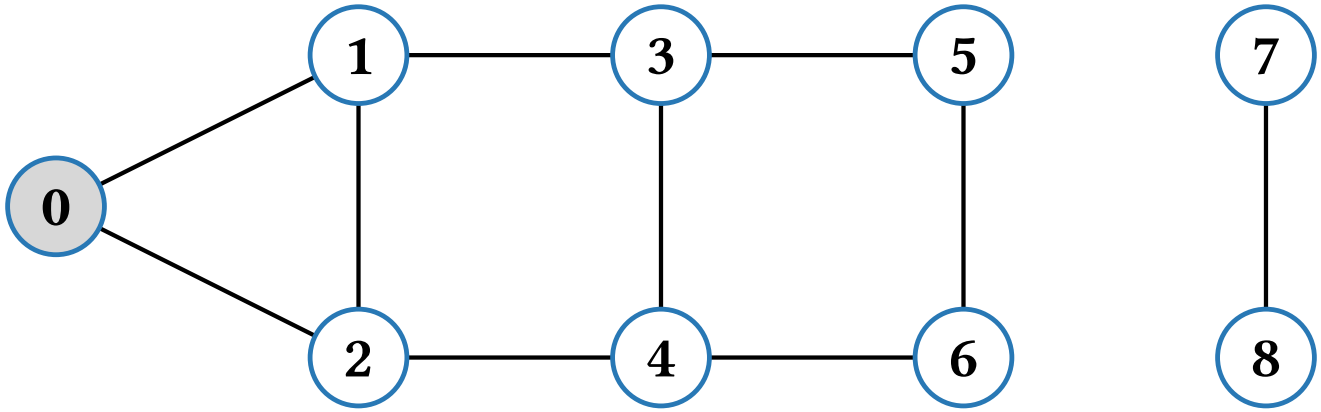
retirer le premier élément d'une liste décale tous les autres : coût proportionnel à sa longueur

# Le parcours en largeur (Python)

```
def parcours_largeur(G, depart):
    vus = [False for s in range(len(G))]
    ordre = []
    file = deque()
    file.append(depart)
    vus[depart] = True
    while len(file) > 0:
        s = file.popleft()
        ordre.append(s)
        for t in G[s]:
            if not vus[t]:
                vus[t] = True
                file.append(t)
    return ordre
```

marquage **à l'entrée** : chaque sommet entre une seule fois dans la file $O(n+m)$

# La file pas à pas

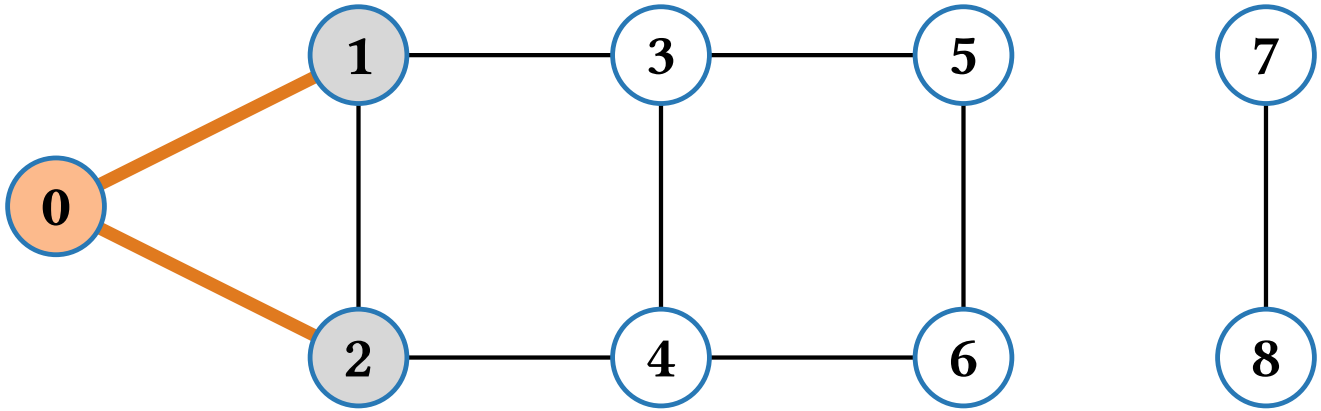



file : `[0]`

ordre : `[]`

au début

# La file pas à pas

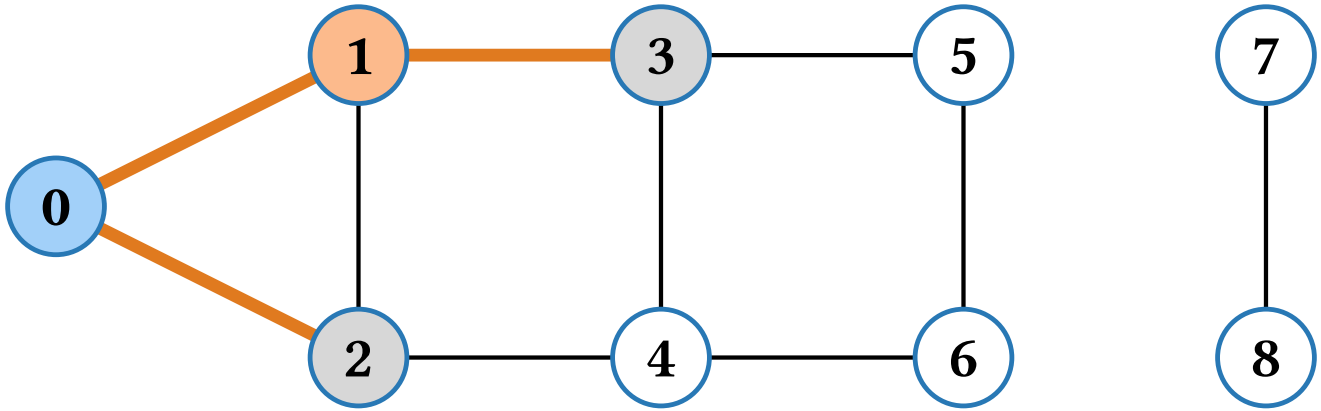



file : `[1, 2]`

ordre : `[0]`

on sort 0

# La file pas à pas

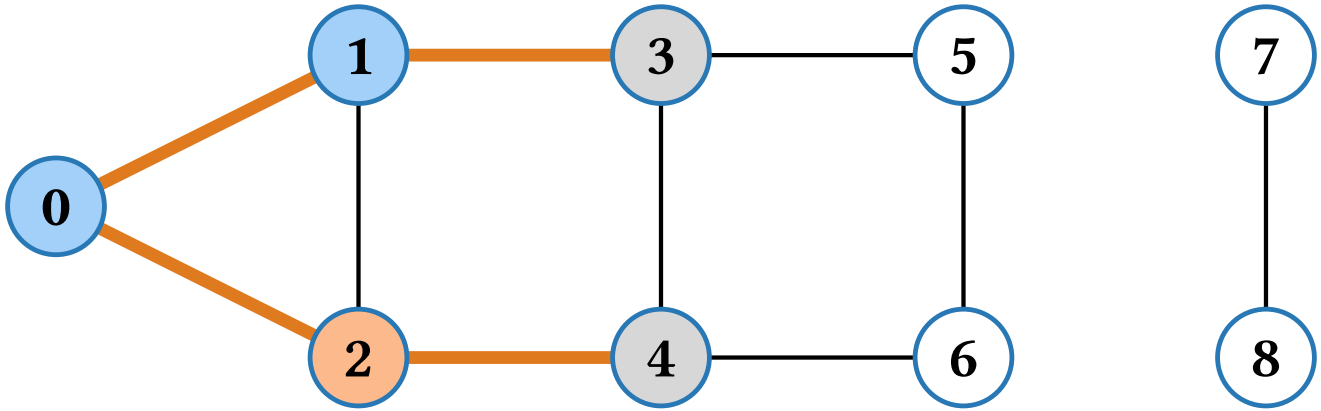



file : `[2, 3]`

ordre : `[0, 1]`

on sort 1

# La file pas à pas

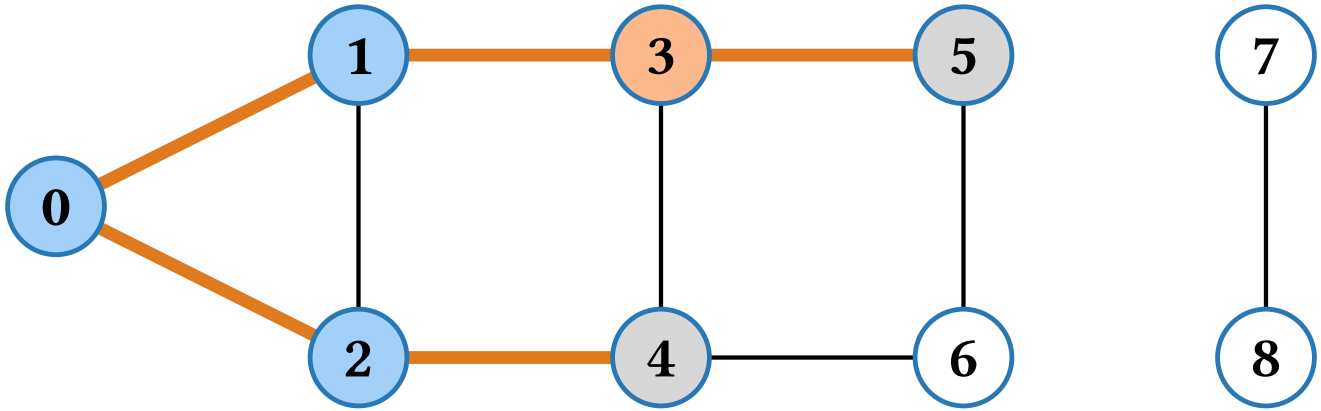



file : `[3, 4]`

ordre : `[0, 1, 2]`

on sort 2

# La file pas à pas

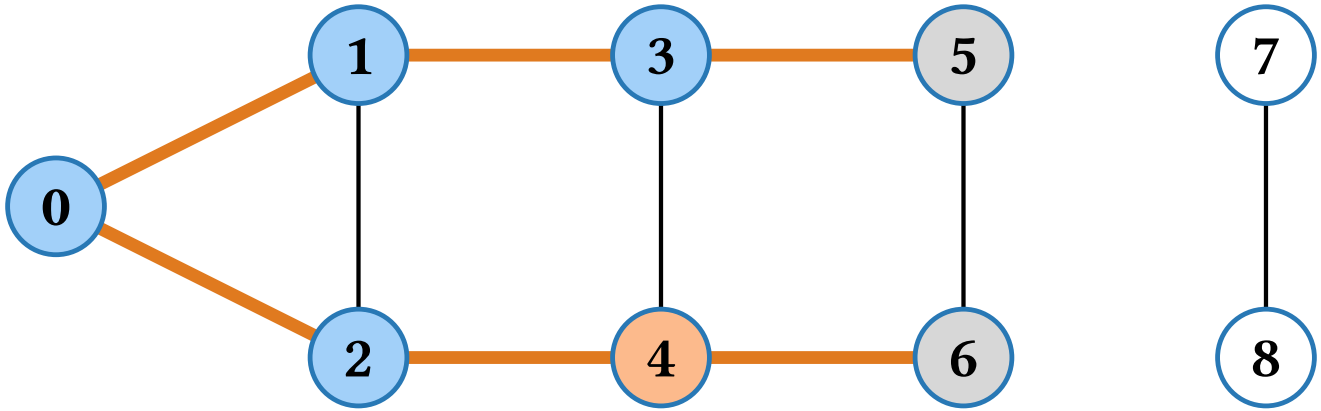



file : `[4, 5]`

ordre : `[0, 1, 2, 3]`

on sort 3

# La file pas à pas

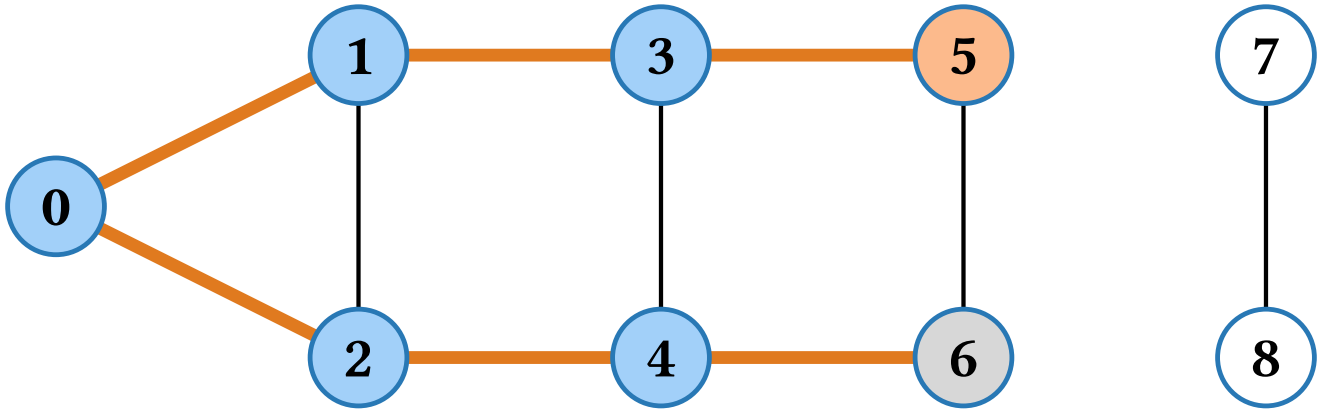



file : `[5, 6]`

ordre : `[0, 1, 2, 3, 4]`

on sort 4

# La file pas à pas

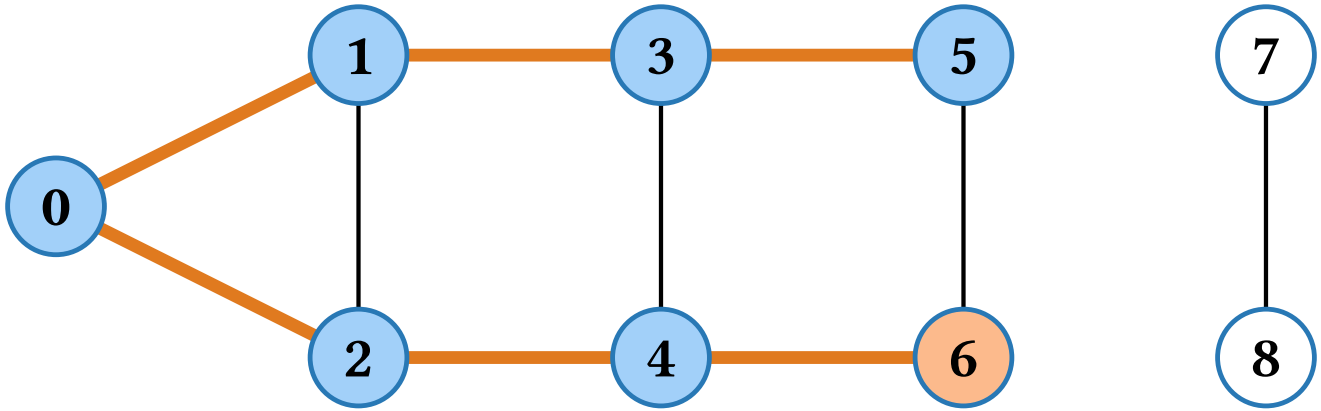



file : `[6]`

ordre : `[0, 1, 2, 3, 4, 5]`

on sort 5

# La file pas à pas



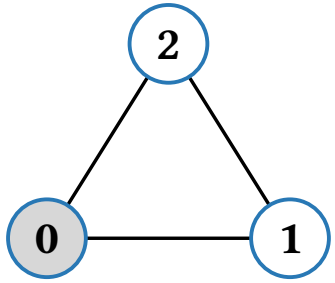

file : `[]`

ordre : `[0, 1, 2, 3, 4, 5, 6]`

on sort 6

# Attention : marquer à l'entrée

```
def parcours_largeur_maladroit(G, depart):
    vus = [False for s in range(len(G))]
    ordre = []
    file = deque()
    file.append(depart)
    while len(file) > 0:
        s = file.popleft()
        vus[s] = True
        ordre.append(s)
        for t in G[s]:
            if not vus[t]:
                file.append(t)
    return ordre
```



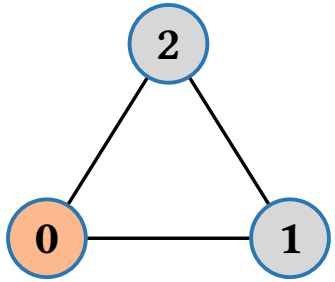

file : `[0]`
ordre : `[]`

au début

**On marque un sommet quand il entre dans la file, pas quand il en sort.**

# Attention : marquer à l'entrée

```
def parcours_largeur_maladroit(G, depart):
    vus = [False for s in range(len(G))]
    ordre = []
    file = deque()
    file.append(depart)
    while len(file) > 0:
        s = file.popleft()
        vus[s] = True
        ordre.append(s)
        for t in G[s]:
            if not vus[t]:
                file.append(t)
    return ordre
```



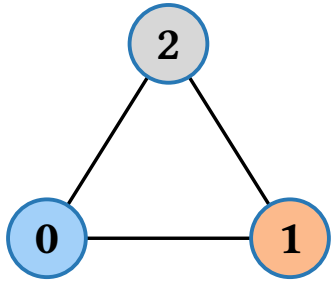

file : `[1, 2]`
ordre : `[0]`

on sort 0 : ses voisins 1 et 2 entrent

**On marque un sommet quand il entre dans la file, pas quand il en sort.**

# Attention : marquer à l'entrée

```
def parcours_largeur_maladroit(G, depart):
    vus = [False for s in range(len(G))]
    ordre = []
    file = deque()
    file.append(depart)
    while len(file) > 0:
        s = file.popleft()
        vus[s] = True
        ordre.append(s)
        for t in G[s]:
            if not vus[t]:
                file.append(t)
    return ordre
```



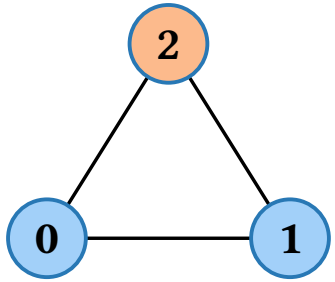

file : `[2, 2]`
ordre : `[0, 1]`

on sort 1 : 2 n'est pas encore **marqué**, il entre une seconde fois

**On marque un sommet quand il entre dans la file, pas quand il en sort.**

# Attention : marquer à l'entrée

```
def parcours_largeur_maladroit(G, depart):
    vus = [False for s in range(len(G))]
    ordre = []
    file = deque()
    file.append(depart)
    while len(file) > 0:
        s = file.popleft()
        vus[s] = True
        ordre.append(s)
        for t in G[s]:
            if not vus[t]:
                file.append(t)
    return ordre
```



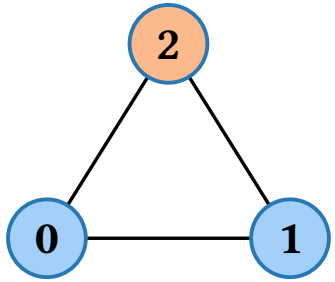

file : `[2]`
ordre : `[0, 1, 2]`

on sort 2

**On marque un sommet quand il entre dans la file, pas quand il en sort.**

# Attention : marquer à l'entrée

```
def parcours_largeur_maladroit(G, depart):
    vus = [False for s in range(len(G))]
    ordre = []
    file = deque()
    file.append(depart)
    while len(file) > 0:
        s = file.popleft()
        vus[s] = True
        ordre.append(s)
        for t in G[s]:
            if not vus[t]:
                file.append(t)
    return ordre
```



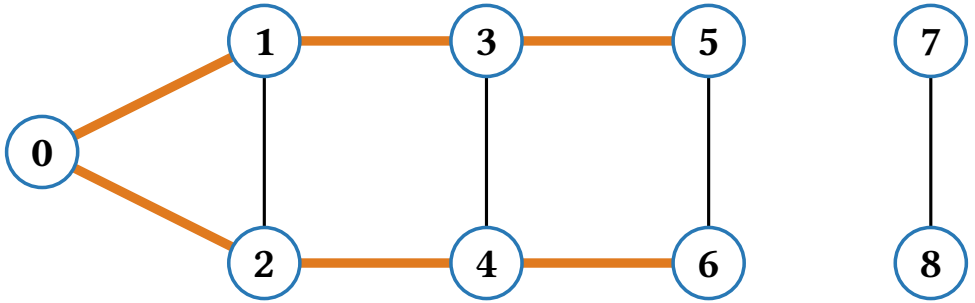

file : `[]`
ordre : `[0, 1, 2, 2]`

on sort 2 **une seconde fois** : il est visité deux fois

**On marque un sommet quand il entre dans la file, pas quand il en sort.**

# L'arbre de parcours (Python)

```python
def arbre_largeur(G, depart):
    pred = [-1 for s in range(len(G))]
    pred[depart] = depart
    file = deque()
    file.append(depart)
    while len(file) > 0:
        s = file.popleft()
        for t in G[s]:
            if pred[t] == -1:
                pred[t] = s
                file.append(t)
    return pred
```




`pred` : `[0, 0, 0, 1, 2, 3, 4, -1, -1]`

il sert aussi de marquage

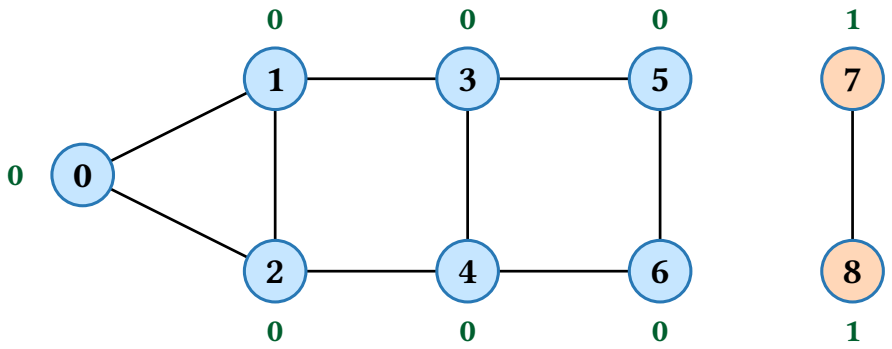
`pred` contient, pour chaque sommet atteint, un chemin depuis le départ : on remonte de prédécesseur en prédécesseur

# Ce qu’on fait avec un parcours

# Composantes connexes (Python)

```python
def marquer_aux(G, s, numero, c):
    numero[s] = c
    for t in G[s]:
        if numero[t] == -1:
            marquer_aux(G, t, numero, c)

def composantes(G):
    numero = [-1 for s in range(len(G))]
    nb = 0
    for s in range(len(G)):
        if numero[s] == -1:
            marquer_aux(G, s, numero, nb)
            nb = nb + 1
    return nb, numero
```




information ajoutée : le **numéro**

$O(n+m)$ **au total** : chaque sommet n'est marqué qu'une fois

# Composantes sur une grille (Python)

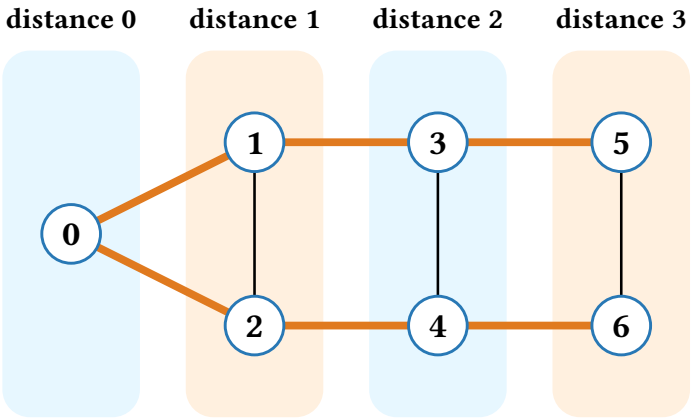
```python
def marquer_domaine(grille, vus, i_depart, j_depart):
    pile = [(i_depart, j_depart)]
    while len(pile) > 0:
        (i, j) = pile.pop()
        if not vus[i][j]:
            vus[i][j] = True
            voisins = voisins_grille(len(grille), len(grille[0]), i, j)
            for (i_vois, j_vois) in voisins:
                if grille[i_vois][j_vois] == 1 and not vus[i_vois][j_vois]:
                    pile.append((i_vois, j_vois))
```

| | | | | |
|---|---|---|---|---|
| 1 | 1 | | | 2 |
| | 1 | | 2 | 2 |
| | | | | |
| 3 | | 4 | 4 | |

quatre domaines $L$ lignes, $C$ colonnes : $O(LC)$

# Distances en nombre d'arêtes (Python)

```python
def distances(G, depart):
    dist = [-1 for s in range(len(G))]
    pred = [-1 for s in range(len(G))]
    dist[depart] = 0
    pred[depart] = depart
    file = deque()
    file.append(depart)
    while len(file) > 0:
        s = file.popleft()
        for t in G[s]:
            if dist[t] == -1:
                dist[t] = dist[s] + 1
                pred[t] = s
                file.append(t)
    return dist, pred
```

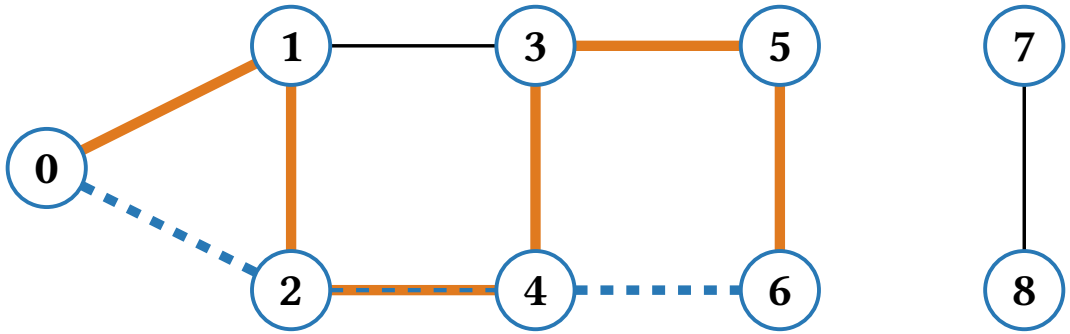



```
[0, 1, 1, 2, 2, 3, 3, -1, -1]
```

la file contient des sommets à distance $k$, puis $k+1$ : ils sortent **par distance croissante**

# Retrouver le chemin (Python)

```python
def chemin(pred, depart, arrivee):
    sommets = [arrivee]
    s = arrivee
    while s != depart:
        s = pred[s]
        sommets.append(s)
    return sommets[::-1]
```




`chemin(pred, 0, 6)` : `[0, 2, 4, 6]`

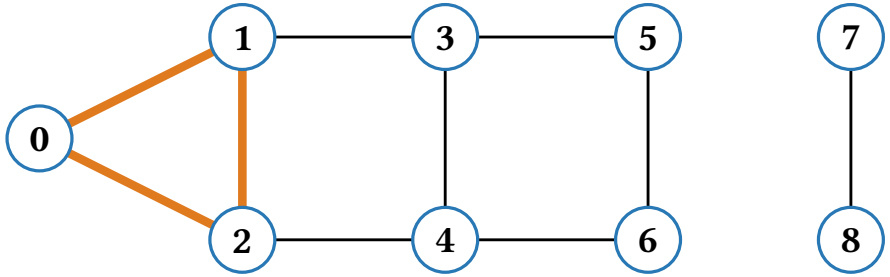
trait plein : en profondeur
tirets : en largeur

le parcours en profondeur trouve **un** chemin, pas un plus court : seule la largeur donne les distances

# Cycle dans un graphe non orienté (Python)

```python
def cycle_aux(G, s, parent, vus):
    vus[s] = True
    for t in G[s]:
        if not vus[t]:
            if cycle_aux(G, t, s, vus):
                return True
        elif t != parent:
            return True
    return False

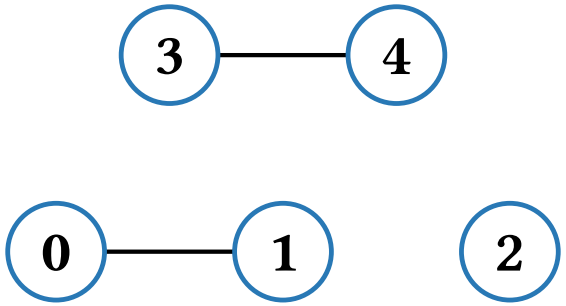
def a_un_cycle(G):
    vus = [False for s in range(len(G))]
    for s in range(len(G)):
        if not vus[s]:
            if cycle_aux(G, s, -1, vus):
                return True
    return False
```




voisin **déjà vu** : un cycle,
sauf si c'est le **parent**

## Cycle : il suffit de compter

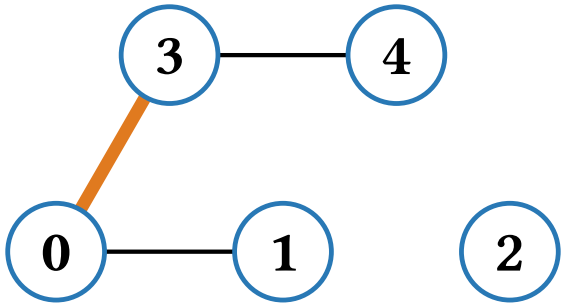
**Propriété.** $n$ sommets, $m$ arêtes, $c$ composantes : $m \geq n - c$, avec égalité si et seulement si le graphe est sans cycle.




$n = 5, m = 2, c = 3$

## Cycle : il suffit de compter

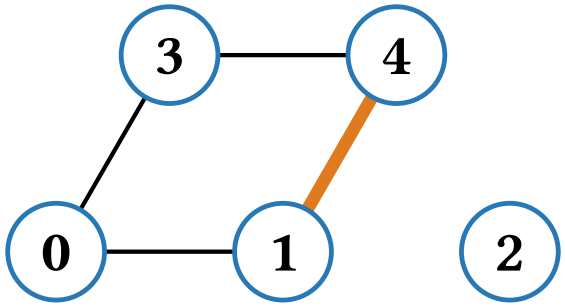
**Propriété.** $n$ sommets, $m$ arêtes, $c$ composantes : $m \geq n - c$, avec égalité si et seulement si le graphe est sans cycle.




l'arête relie deux composantes : $c = 2$, pas de cycle

## Cycle : il suffit de compter

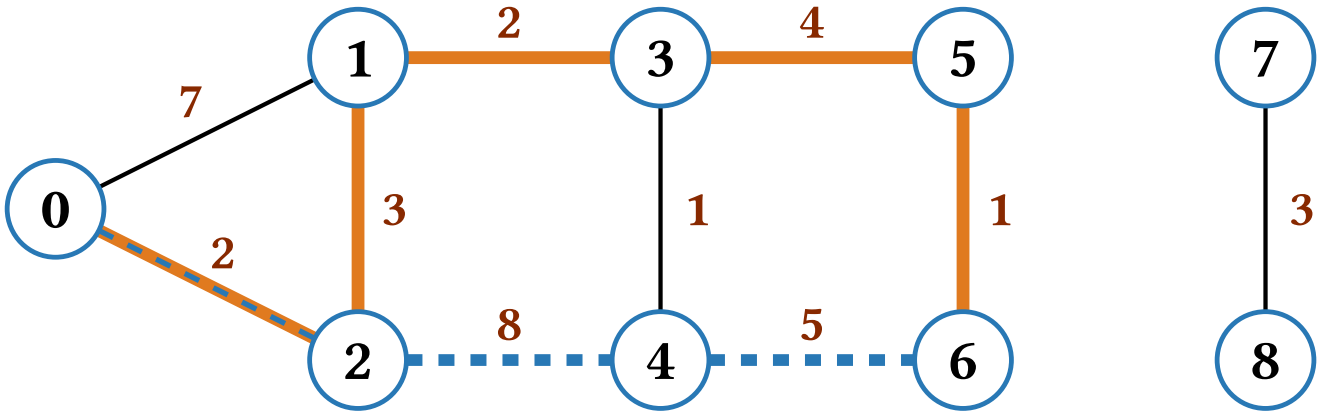
**Propriété.** $n$ sommets, $m$ arêtes, $c$ composantes : $m \geq n - c$, avec égalité si et seulement si le graphe est sans cycle.




l'arête reste dans une composante : $c = 2$, un cycle

# Plus courts chemins pondérés : l'algorithme de Dijkstra

# Le problème




poids **positifs ou nuls**

tirets : le moins d'arêtes, poids 15

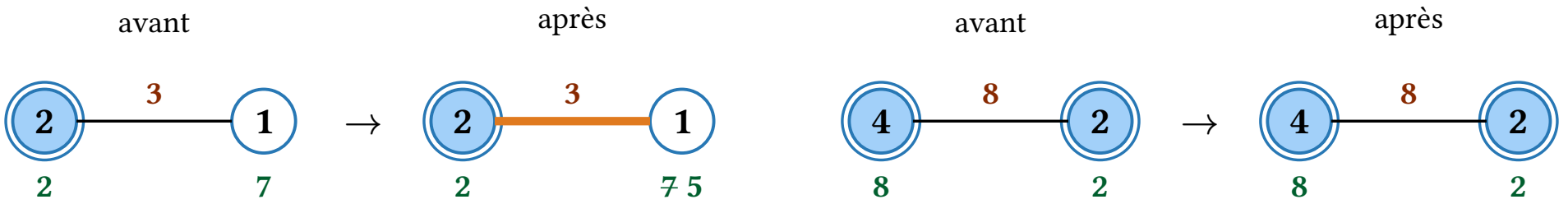
trait plein : poids 12

# Une première idée : découper les arêtes




une **vague** qui atteint les sommets par distance croissante

# L'idée : choisir, relâcher



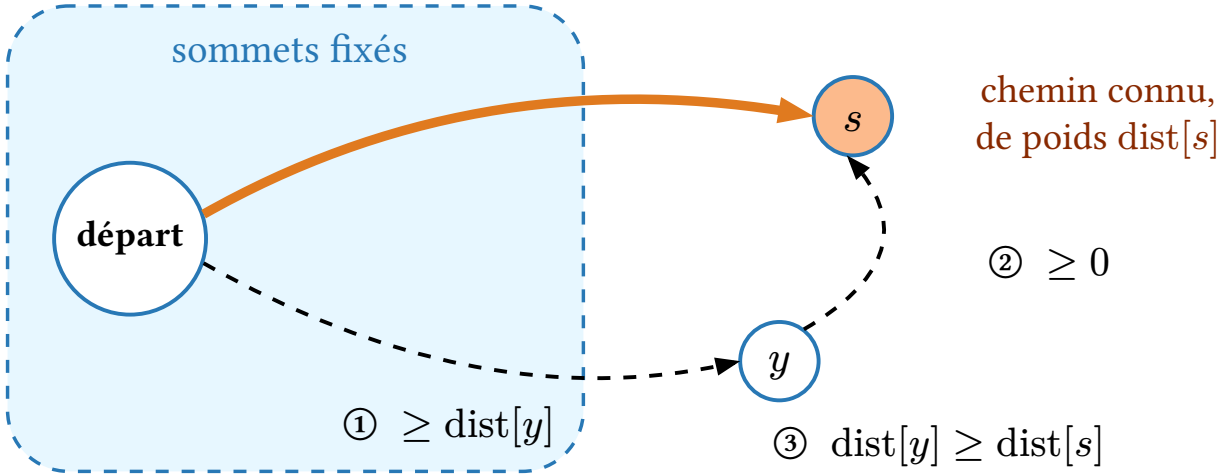

$2 + 3 < 7$ : l'estimation de 1 devient 5

$8 + 8 \geq 2$ : rien ne change

1. **Choisir** le sommet non fixé d'estimation minimale, et le fixer
2. **Relâcher** ses arêtes : si `dist[s] + poids < dist[t]`, améliorer `dist[t]`

**Algorithme glouton : le choix n'est jamais remis en cause.**

# Pourquoi le choix est sûr




poids du chemin en tirets $=$ ① $+$ ② $\geq$ dist$[y] \geq$ dist$[s]$

- tout chemin vers $s$ sort de la zone fixée en un sommet $y$ : son poids est au moins `dist[y]` $\geq$ `dist[s]`
- ② est le **seul** endroit où servent les poids positifs

## L'invariant

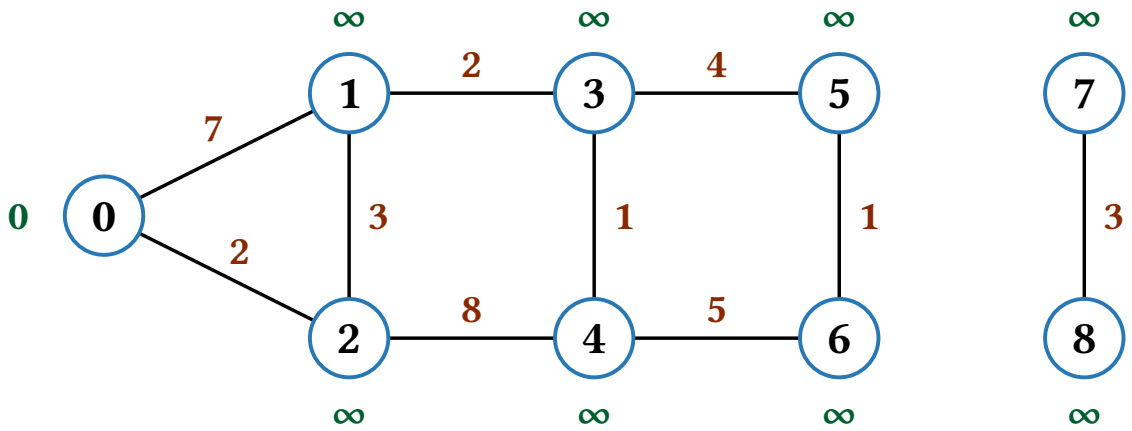
**Invariant.** Avant chaque choix :

1. pour tout sommet fixé $u$, `dist[u]` $= \delta(u)$,
2. pour tout sommet non fixé $t$, `dist[t]` est le plus petit poids d'un chemin du départ à $t$ dont tous les sommets, sauf $t$, sont fixés.

le **choix** maintient le point 1, le **relâchement** rétablit le point 2

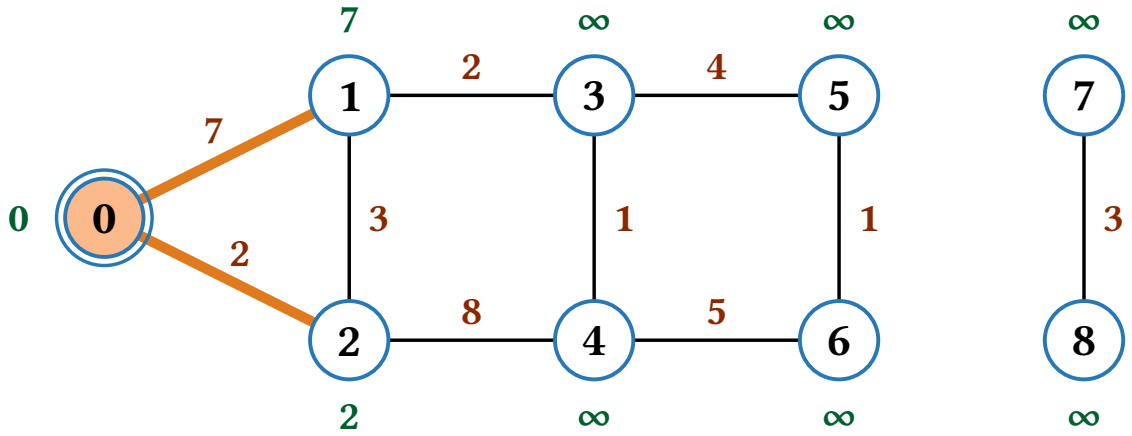
les sommets sont fixés **par distance croissante**

# Exécution à la main



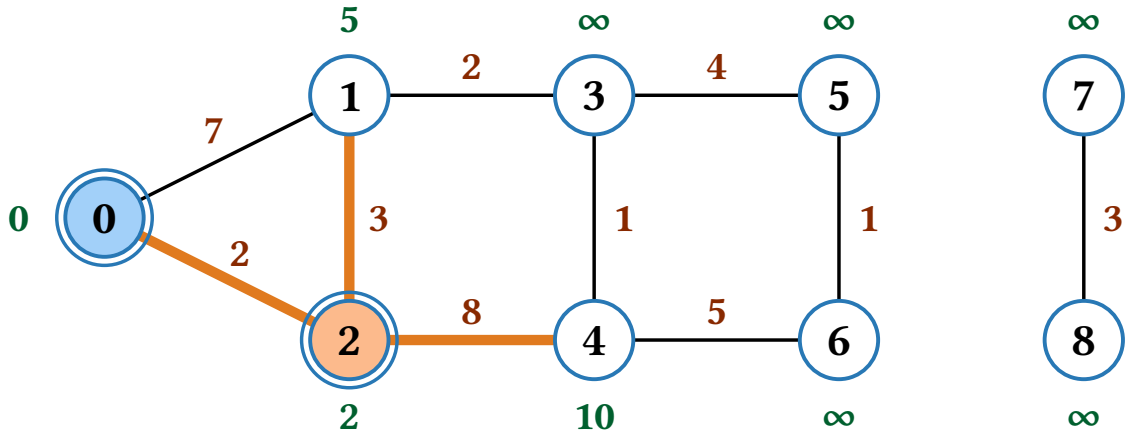

| Sommet fixé | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| (début) | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |

# Exécution à la main



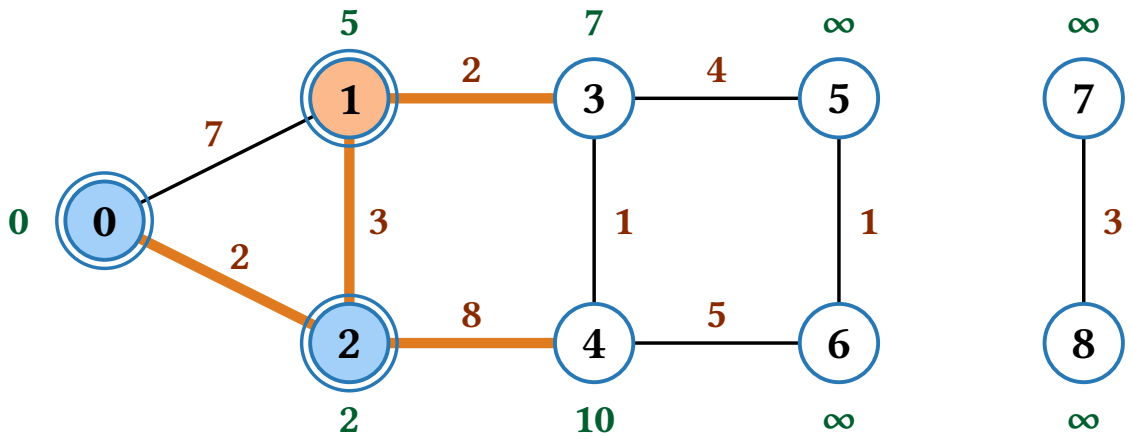

| Sommet fixé | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| (début) | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 0 | 0 | 7 | 2 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |

# Exécution à la main



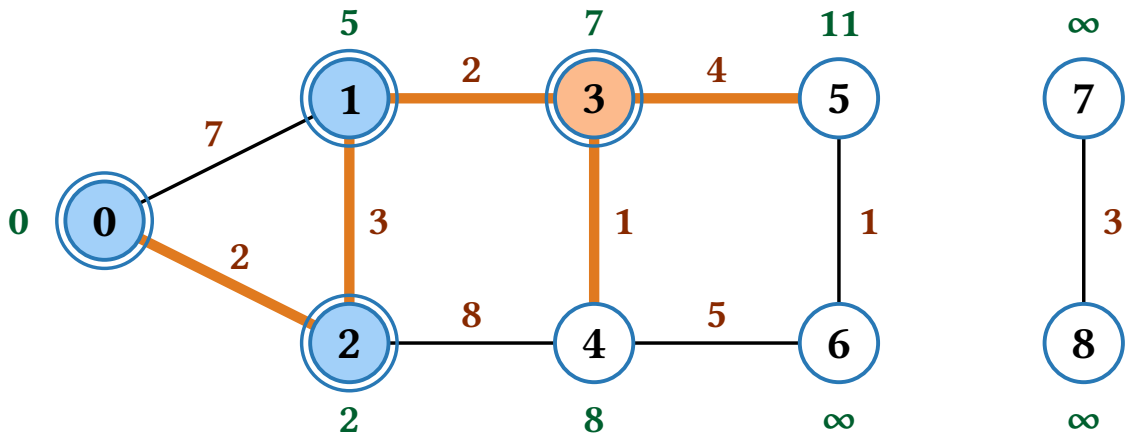

| Sommet fixé | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| (début) | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 0 | [0] | 7 | 2 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 2 | [0] | 5 | [2] | ∞ | 10 | ∞ | ∞ | ∞ | ∞ |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |

# Exécution à la main



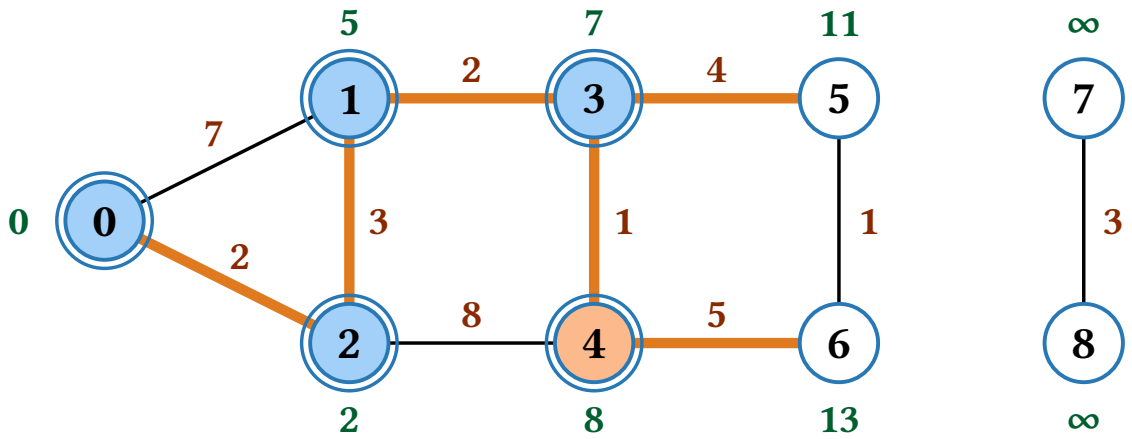

| Sommet fixé | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| (début) | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 0 | [0] | 7 | 2 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 2 | [0] | 5 | [2] | ∞ | 10 | ∞ | ∞ | ∞ | ∞ |
| 1 | [0] | [5] | [2] | 7 | 10 | ∞ | ∞ | ∞ | ∞ |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |

# Exécution à la main



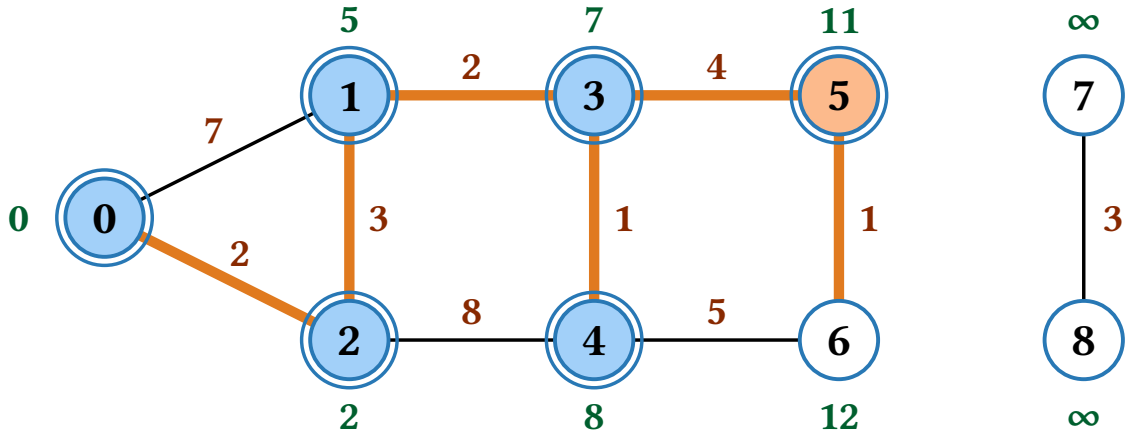

| Sommet fixé | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| (début) | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 0 | [0] | 7 | 2 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 2 | [0] | 5 | [2] | ∞ | 10 | ∞ | ∞ | ∞ | ∞ |
| 1 | [0] | [5] | [2] | 7 | 10 | ∞ | ∞ | ∞ | ∞ |
| 3 | [0] | [5] | [2] | [7] | 8 | 11 | ∞ | ∞ | ∞ |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |

# Exécution à la main



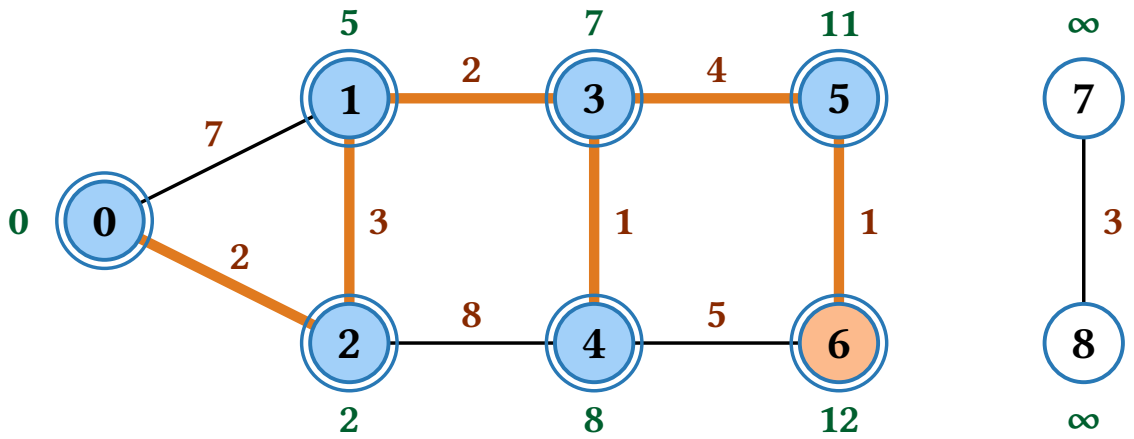

| Sommet fixé | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| (début) | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 0 | [0] | 7 | 2 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 2 | [0] | 5 | [2] | ∞ | 10 | ∞ | ∞ | ∞ | ∞ |
| 1 | [0] | [5] | [2] | 7 | 10 | ∞ | ∞ | ∞ | ∞ |
| 3 | [0] | [5] | [2] | [7] | 8 | 11 | ∞ | ∞ | ∞ |
| 4 | [0] | [5] | [2] | [7] | [8] | 11 | 13 | ∞ | ∞ |
| | | | | | | | | | |
| | | | | | | | | | |

# Exécution à la main




| Sommet fixé | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| (début) | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 0 | [0] | 7 | 2 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 2 | [0] | 5 | [2] | ∞ | 10 | ∞ | ∞ | ∞ | ∞ |
| 1 | [0] | [5] | [2] | 7 | 10 | ∞ | ∞ | ∞ | ∞ |
| 3 | [0] | [5] | [2] | [7] | 8 | 11 | ∞ | ∞ | ∞ |
| 4 | [0] | [5] | [2] | [7] | [8] | 11 | 13 | ∞ | ∞ |
| 5 | [0] | [5] | [2] | [7] | [8] | [11] | 12 | ∞ | ∞ |
| | | | | | | | | | |

# Exécution à la main




| Sommet fixé | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| (début) | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 0 | [0] | 7 | 2 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 2 | [0] | 5 | [2] | ∞ | 10 | ∞ | ∞ | ∞ | ∞ |
| 1 | [0] | [5] | [2] | 7 | 10 | ∞ | ∞ | ∞ | ∞ |
| 3 | [0] | [5] | [2] | [7] | 8 | 11 | ∞ | ∞ | ∞ |
| 4 | [0] | [5] | [2] | [7] | [8] | 11 | 13 | ∞ | ∞ |
| 5 | [0] | [5] | [2] | [7] | [8] | [11] | 12 | ∞ | ∞ |
| 6 | [0] | [5] | [2] | [7] | [8] | [11] | [12] | ∞ | ∞ |

# L'algorithme de Dijkstra (Python)

```
def dijkstra(G, depart):
    n = len(G)
    dist = [INFINI for s in range(n)]
    fixe = [False for s in range(n)]
    dist[depart] = 0
    s = plus_proche(dist, fixe)
    while s != -1:
        fixe[s] = True                          # choisir : s est définitif
        for (t, poids) in G[s]:                 # relâcher les arêtes issues de s
            if dist[s] + poids < dist[t]:
                dist[t] = dist[s] + poids
        s = plus_proche(dist, fixe)
    return dist
```

`dijkstra(G_POIDS, 0)` renvoie `[0, 5, 2, 7, 8, 11, 12, inf, inf]`

# Le choix : une recherche de minimum (Python)

```python
def plus_proche(dist, fixe):
    s = -1
    mini = INFINI
    for t in range(len(dist)):
        if not fixe[t] and dist[t] < mini:
            s = t
            mini = dist[t]
    return s
```

la « file de priorité naïve » du programme ; `-1` s'il ne reste que des sommets à l'infini

## Terminaison et complexité

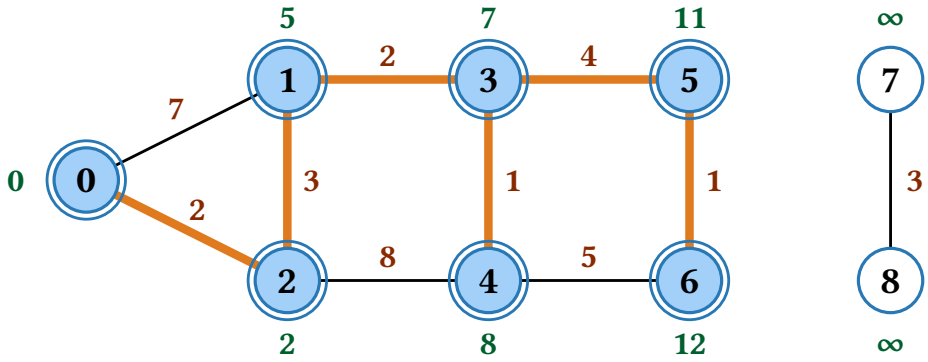
- chaque tour fixe un nouveau sommet : **au plus $n$ tours**
- `plus_proche` : $O(n)$ par tour, donc $O(n^2)$
- relâchements : $O(d(s))$ pour le sommet `s`, donc $O(m)$ en tout

$$O(n^2 + m) = O(n^2)$$

avec une vraie file de priorité : $O((n + m)\ln(n))$

# Variante : retrouver les chemins (Python)

```
def dijkstra_pred(G, depart):
    n = len(G)
    dist = [INFINI for s in range(n)]
    pred = [-1 for s in range(n)]
    fixe = [False for s in range(n)]
    dist[depart] = 0
    pred[depart] = depart
    s = plus_proche(dist, fixe)
    while s != -1:
        fixe[s] = True
        for (t, poids) in G[s]:
            if dist[s] + poids < dist[t]:
                dist[t] = dist[s] + poids
                pred[t] = s
        s = plus_proche(dist, fixe)
    return dist, pred
```




l'arbre des plus courts chemins

`chemin(pred, 0, 6)` : `[0, 2, 1, 3, 5, 6]`

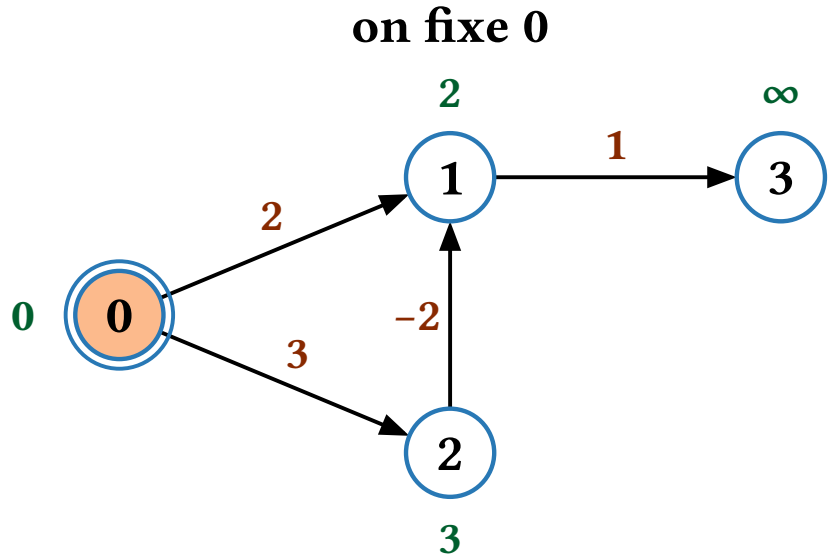
# Variante : sur une grille (Python)

```python
def dijkstra_grille(grille, depart, arrivee):
    dist = {depart: 0}      # estimations des cases déjà découvertes
    fixe = {}               # fixe[s] = True pour les cases fixées
    s = plus_proche_dico(dist, fixe)
    while s != AUCUNE and s != arrivee:
        fixe[s] = True
        for (t, poids) in successeurs(grille, s):
            if not (t in dist) or dist[s] + poids < dist[t]:
                dist[t] = dist[s] + poids
        s = plus_proche_dico(dist, fixe)
    if arrivee in dist:
        return dist[arrivee]
    return INFINI
```

| | | | |
|---|---|---|---|
| **1** | 9 | **1** | **1** |
| **1** | 9 | **1** | 9 |
| **1** | **1** | **1** | 1 |

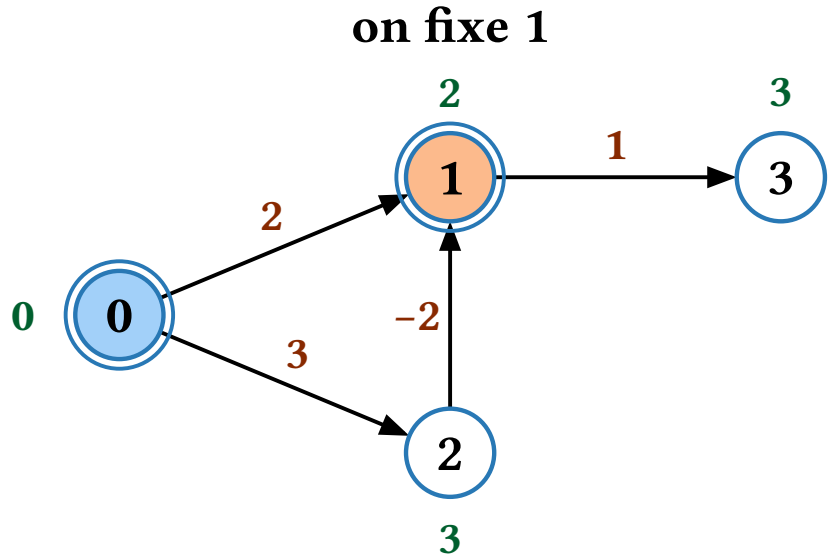
`dist`, `fixe` : des **dictionnaires**  meilleur chemin : coût 7

# Attention : pas de poids négatif



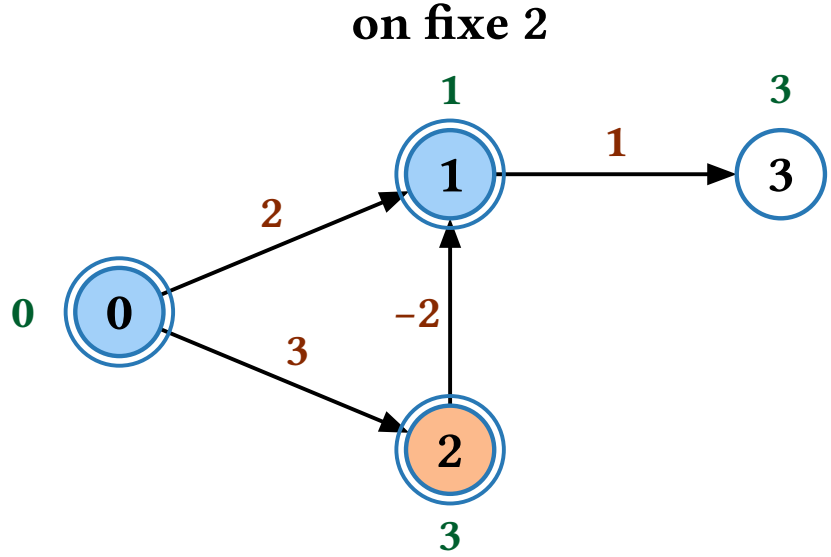

le sommet 1 est fixé trop tôt : on annonce 3 pour le sommet 3 au lieu de 2, **sans message d'erreur**

# Attention : pas de poids négatif



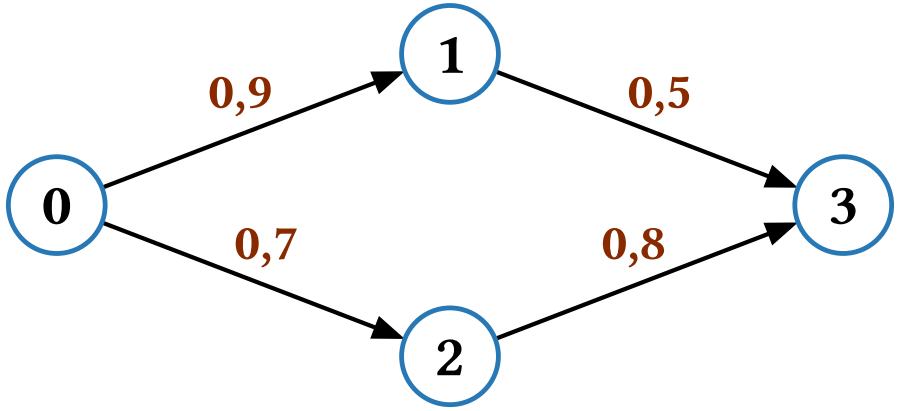

le sommet 1 est fixé trop tôt : on annonce 3 pour le sommet 3 au lieu de 2, **sans message d'erreur**

# Attention : pas de poids négatif




le sommet 1 est fixé trop tôt : on annonce 3 pour le sommet 3 au lieu de 2, **sans message d'erreur**

# Se ramener à des poids positifs

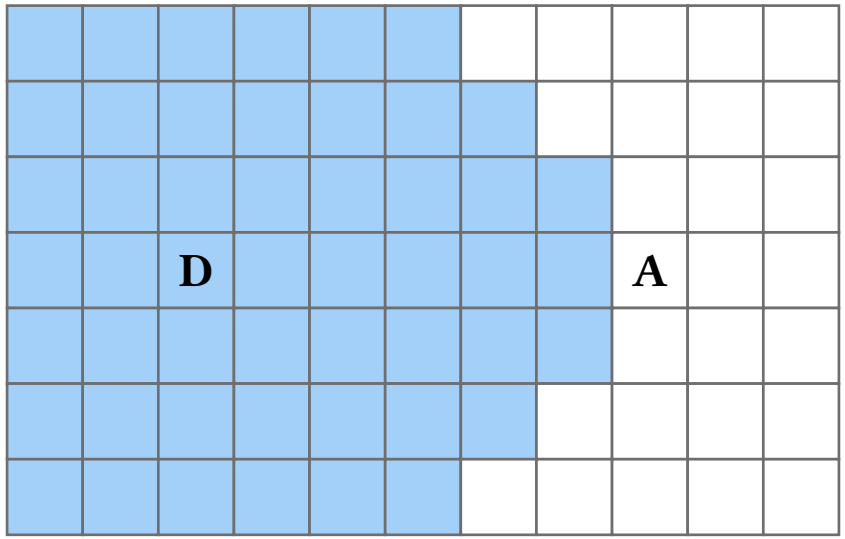



chemin de **produit maximal** ? poids $-\ln q \geq 0$

$$-\ln(q_1 q_2 \cdots q_k) = (-\ln q_1) + (-\ln q_2) + \cdots + (-\ln q_k)$$

# Une seule cible : ce que fait Dijkstra (HP)

**Dijkstra : 50 cases fixées**




il fixe tous les sommets plus proches du départ que la cible, dans toutes les directions

# L'algorithme A* : l'idée (HP)

**On choisit le sommet non fixé qui minimise $\mathrm{dist}[t] + h(t)$.**

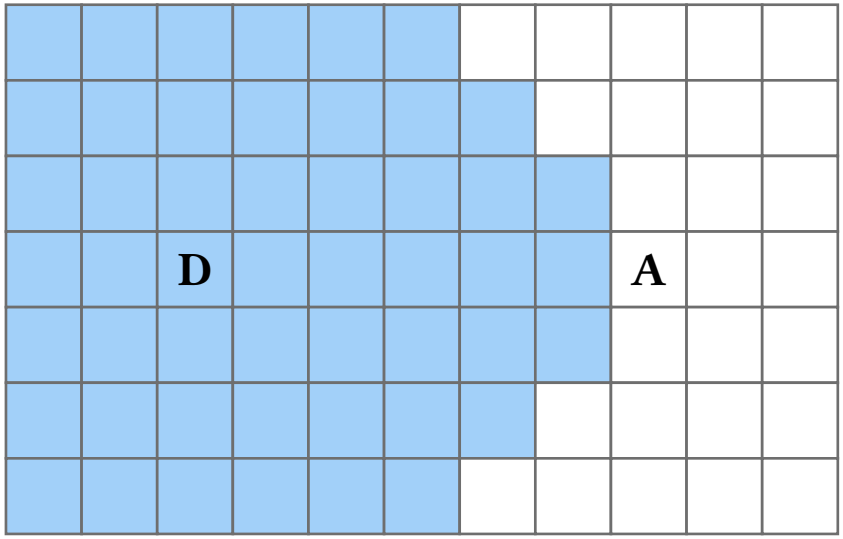
**heuristique** $h(t)$ : estimation de ce qu'il reste à parcourir de $t$ à l'arrivée

rien d'autre ne change : ni `dist`, ni le relâchement, ni l'arrêt

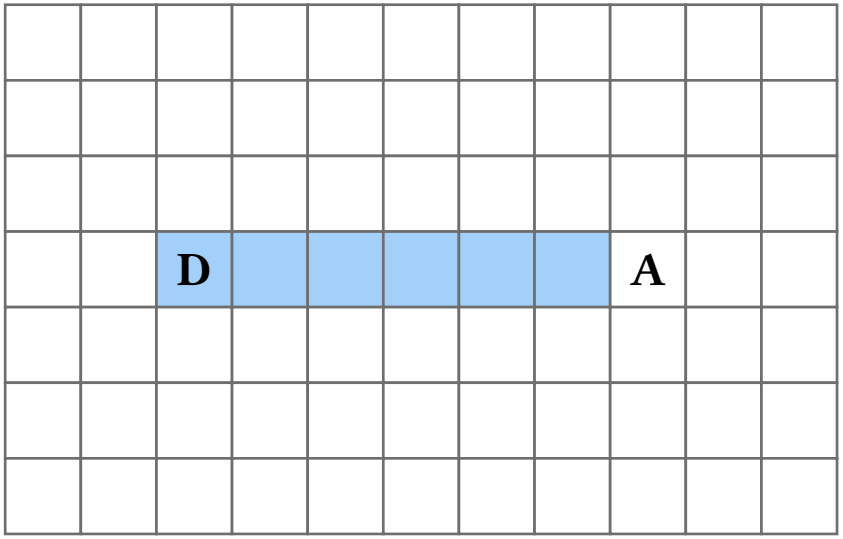
```
def plus_prometteur(dist, fixe, arrivee):
    s = AUCUNE
    mini = INFINI
    for t in dist:
        if not (t in fixe) and dist[t] + manhattan(t, arrivee) < mini:
            s = t
            mini = dist[t] + manhattan(t, arrivee)
    return s
```

# Dijkstra et A* : les cases fixées (HP)

**Dijkstra : 50 cases fixées**



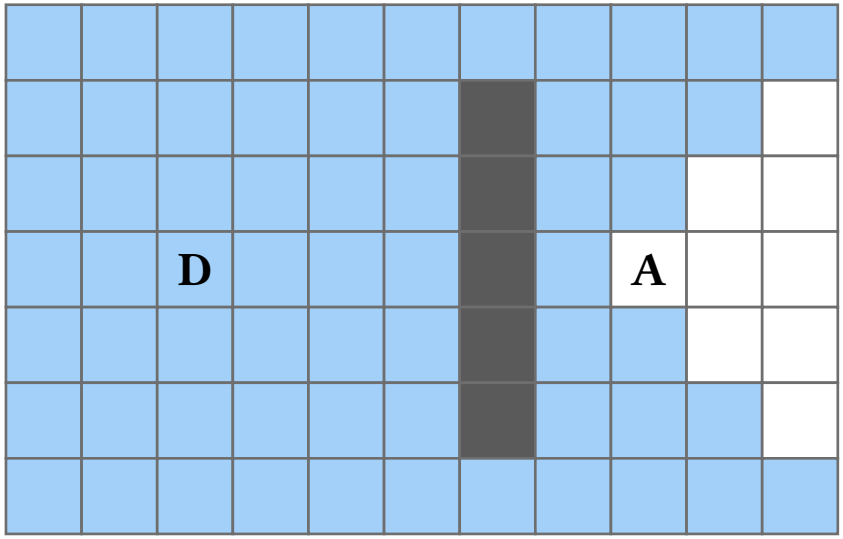

**A* : 6 cases fixées**



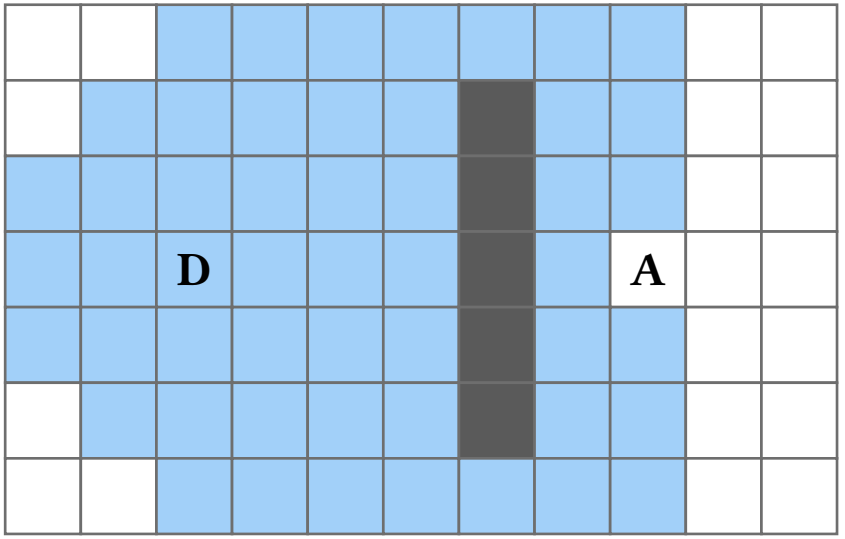

même coût, 6 : ici la distance de Manhattan **est** la vraie distance

# Dijkstra et A* : avec un mur (HP)

**Dijkstra : 63 cases fixées**




**A* : 51 cases fixées**




même coût, 12 : l'heuristique se trompe de direction, le gain est moindre

## Quelle heuristique choisir ? (HP)

**Admissible.** $h(t) \leq \text{reste}(t)$ pour tout sommet $t$ : $h$ ne surestime jamais.

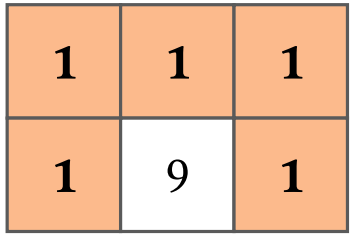
**Cohérente.** $h(\text{arrivée}) = 0$ et, pour toute arête de $s$ vers $t$ de poids $p$ : $h(s) \leq p + h(t)$.

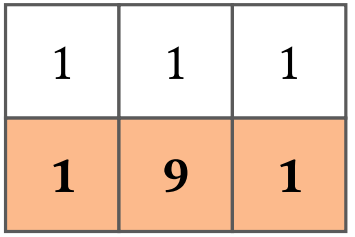
**Si $h$ est cohérente, A* renvoie la distance.**

preuve : c'est Dijkstra avec les poids $\tilde{p} = p + h(t) - h(s) \geq 0$

cohérente $\Longrightarrow$ admissible  Manhattan est cohérente si chaque case coûte au moins 1

# Attention : une heuristique qui surestime (HP)



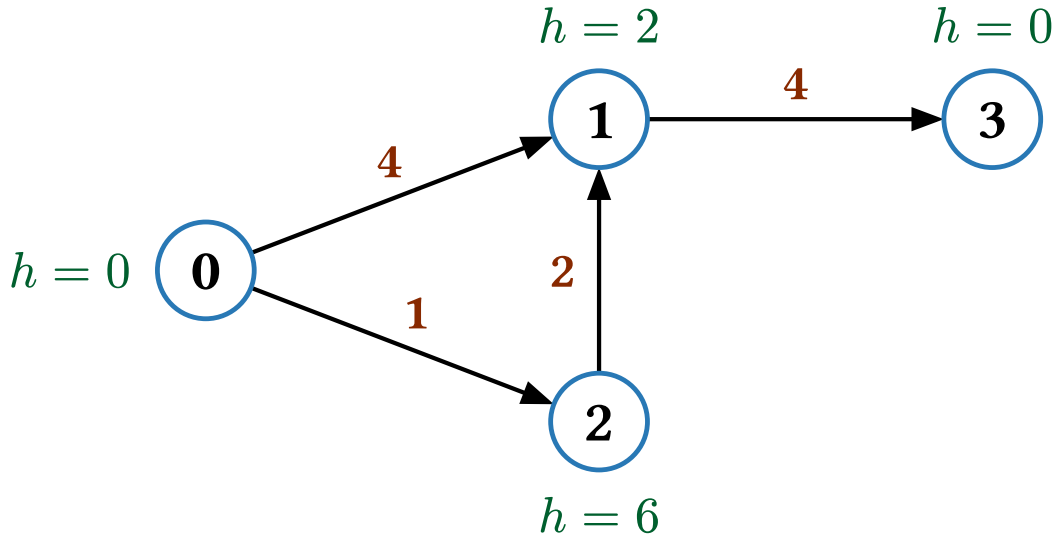

le meilleur chemin : coût 4

avec $5 \times$ Manhattan : coût 10

$h = 0$ : c'est Dijkstra

si $h$ surestime, le résultat peut être **faux**

# Admissible ne suffit pas (HP)



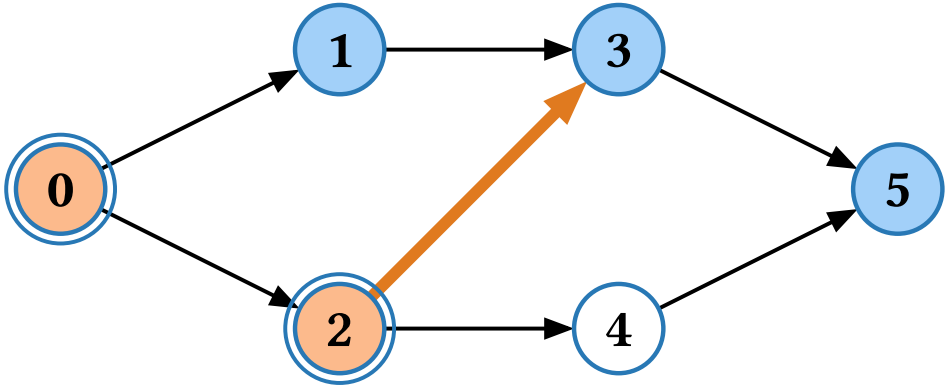

$h(2) = 6 > 2 + h(1)$ : on fixe 1 avant 2, et l'on obtient 8 au lieu de 7

# Quel algorithme pour quel problème ?

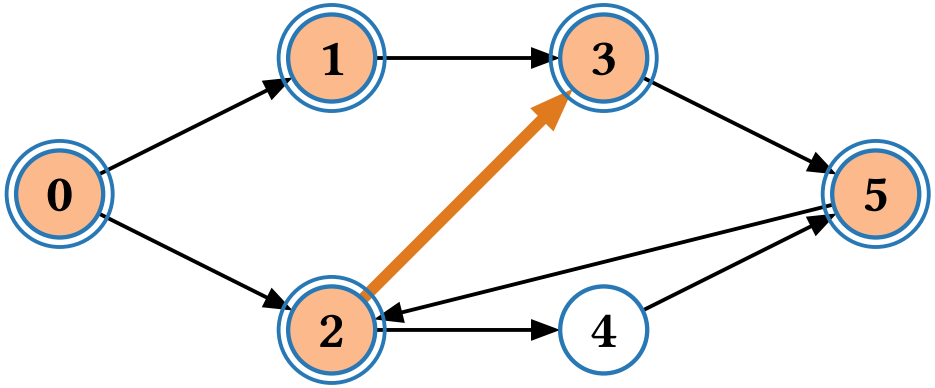
| Algorithme | Hypothèses | Complexité | Ce qu'il calcule |
|---|---|---|---|
| Parcours en largeur | non pondéré | $O(n+m)$ | distances en nombre d'arêtes depuis un sommet |
| Dijkstra | poids $\geq 0$ | $O(n^2)$ ou $O((n+m)\ln(n))$ | distances depuis un sommet |
| A* | poids $\geq 0$, $h$ cohérente | comme Dijkstra (pire cas) | distance à une cible, en fixant moins de sommets |
| Prog. dynamique | orienté sans cycle | $O(n+m)$ | distances depuis ou vers un sommet |

# Graphes orientés sans cycle

# Détecter un cycle : trois états (HP)




2 examine 3, qui est **fini** : pas de cycle




2 examine 3, qui est **en cours** : cycle 3, 5, 2

blanc : **non vu** orange : **en cours** (dans la pile des appels) bleu : **fini**

retomber sur un sommet déjà vu ne prouve rien ; sur un sommet **en cours**, si

# Détecter un cycle (Python) (HP)

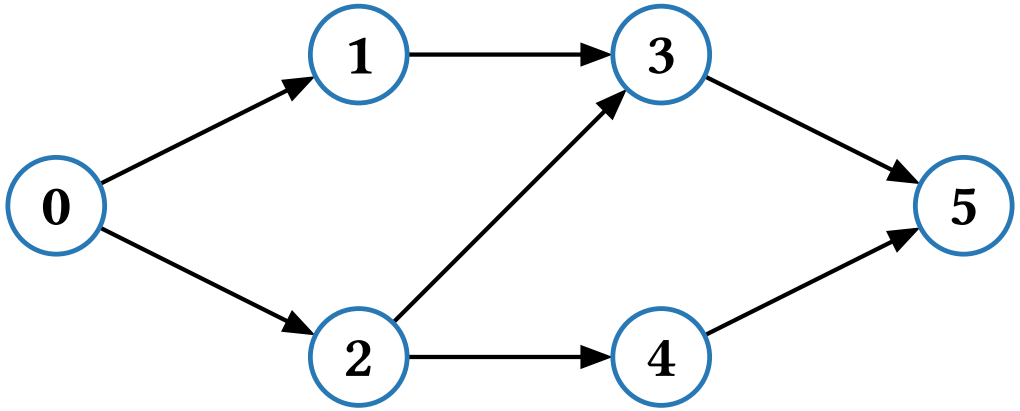
```python
def cycle_oriente_aux(G, s, etat):
    etat[s] = EN_COURS
    for t in G[s]:
        if etat[t] == EN_COURS:
            return True
        if etat[t] == NON_VU:
            if cycle_oriente_aux(G, t, etat):
                return True
    etat[s] = FINI
    return False

def a_un_cycle_oriente(G):
    etat = [NON_VU for s in range(len(G))]
    for s in range(len(G)):
        if etat[s] == NON_VU:
            if cycle_oriente_aux(G, s, etat):
                return True
    return False
```

```python
NON_VU = 0
EN_COURS = 1
FINI = 2
```

le parcours en profondeur,
avec un état au lieu d'un booléen

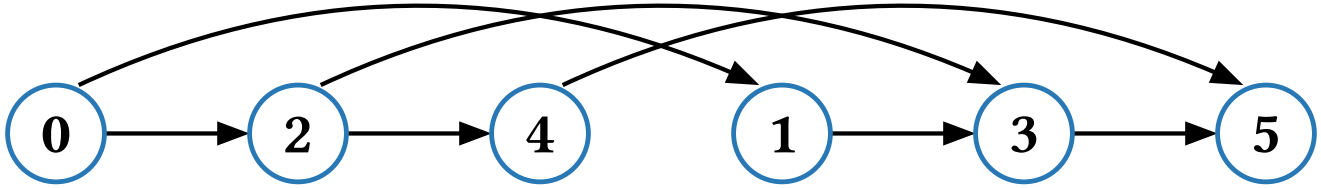
# Ordre topologique (HP)




ranger les sommets sur une ligne…

il en existe un si et seulement si le graphe n'a pas de cycle

# Ordre topologique (HP)




… de sorte que toutes les arêtes aillent de la gauche vers la droite

il en existe un si et seulement si le graphe n'a pas de cycle

# Ordre topologique (Python) (HP)

```python
def sortie_aux(G, s, vus, sortis):
    vus[s] = True
    for t in G[s]:
        if not vus[t]:
            sortie_aux(G, t, vus, sortis)
    sortis.append(s)

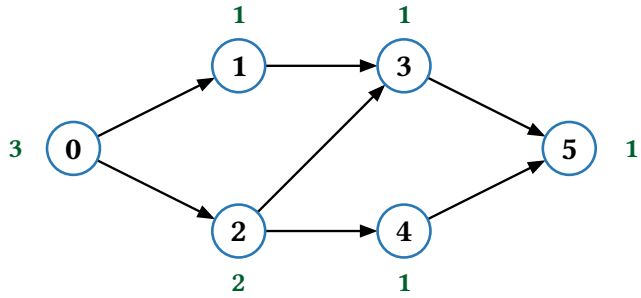
def ordre_topologique(G):
    vus = [False for s in range(len(G))]
    sortis = []
    for s in range(len(G)):
        if not vus[s]:
            sortie_aux(G, s, vus, sortis)
    return sortis[::-1]
```

on note chaque sommet quand son exploration **se termine**, puis on retourne la liste

# Le lien avec la programmation dynamique (Python)

```python
def nb_chemins_aux(G, s, arrivee, memo):
    if s == arrivee:
        return 1
    if s in memo:
        return memo[s]
    total = 0
    for t in G[s]:
        total = total + nb_chemins_aux(G, t, arrivee, memo)
    memo[s] = total
    return total

def nb_chemins(G, depart, arrivee):
    return nb_chemins_aux(G, depart, arrivee, {})
```




nombre de chemins vers 5

**mémoïser** = parcours en profondeur **remplir un tableau** = suivre un ordre topologique

sans `memo` : exponentiel ; avec : $O(n+m)$