Programmation dynamique

Quentin Fortier

October 12, 2022

Sous-problèmes

Pour résoudre un problème, il est courant de le ramener à des sous-problèmes plus simples.

Deux grandes méthodes pour le faire :

1. **Diviser pour régner** : résoudre les sous-problèmes (récursivement) puis les combiner pour obtenir une solution du problème initial.
2. **Programmation dynamique / mémoïsation** : similaire, mais en conservant en mémoire tous les sous-problèmes pour éviter de les calculer plusieurs fois.

Sous-problèmes

Exemple pour le calcul des termes de la suite de Fibonacci :

$$u_0 = \emptyset \mathcal{A}$$

$$u_1 = 1$$

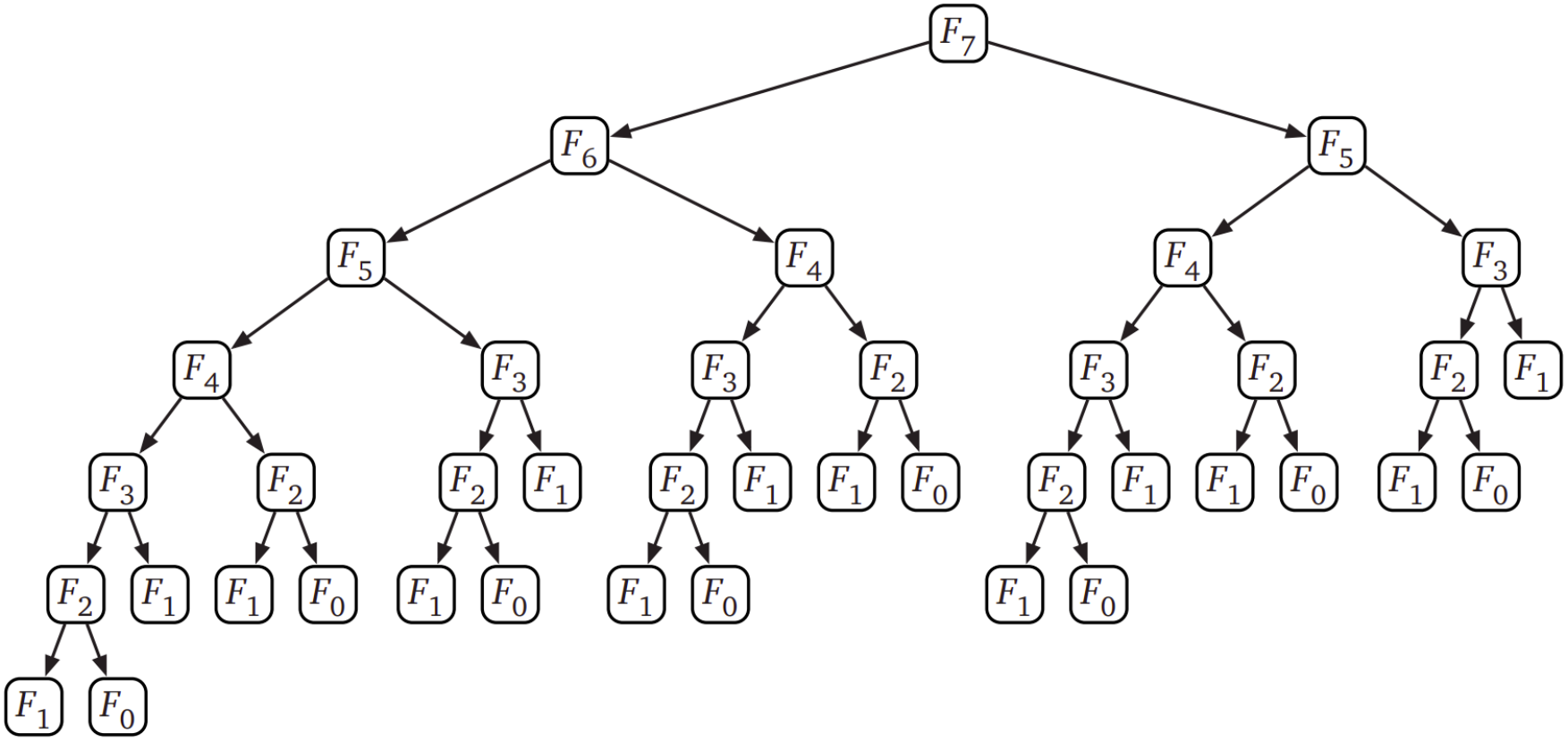
$$u_n = u_{n-1} + u_{n-2}$$

```
def fibo(n):
    if n <= 1:
        return 1
    return fibo(n - 1) + fibo(n - 2)
```

Complexité : exponentielle...

Sous-problèmes

Problème : le même sous-problème est résolu plusieurs fois, ce qui est inutile et inefficace.



Sous-problèmes

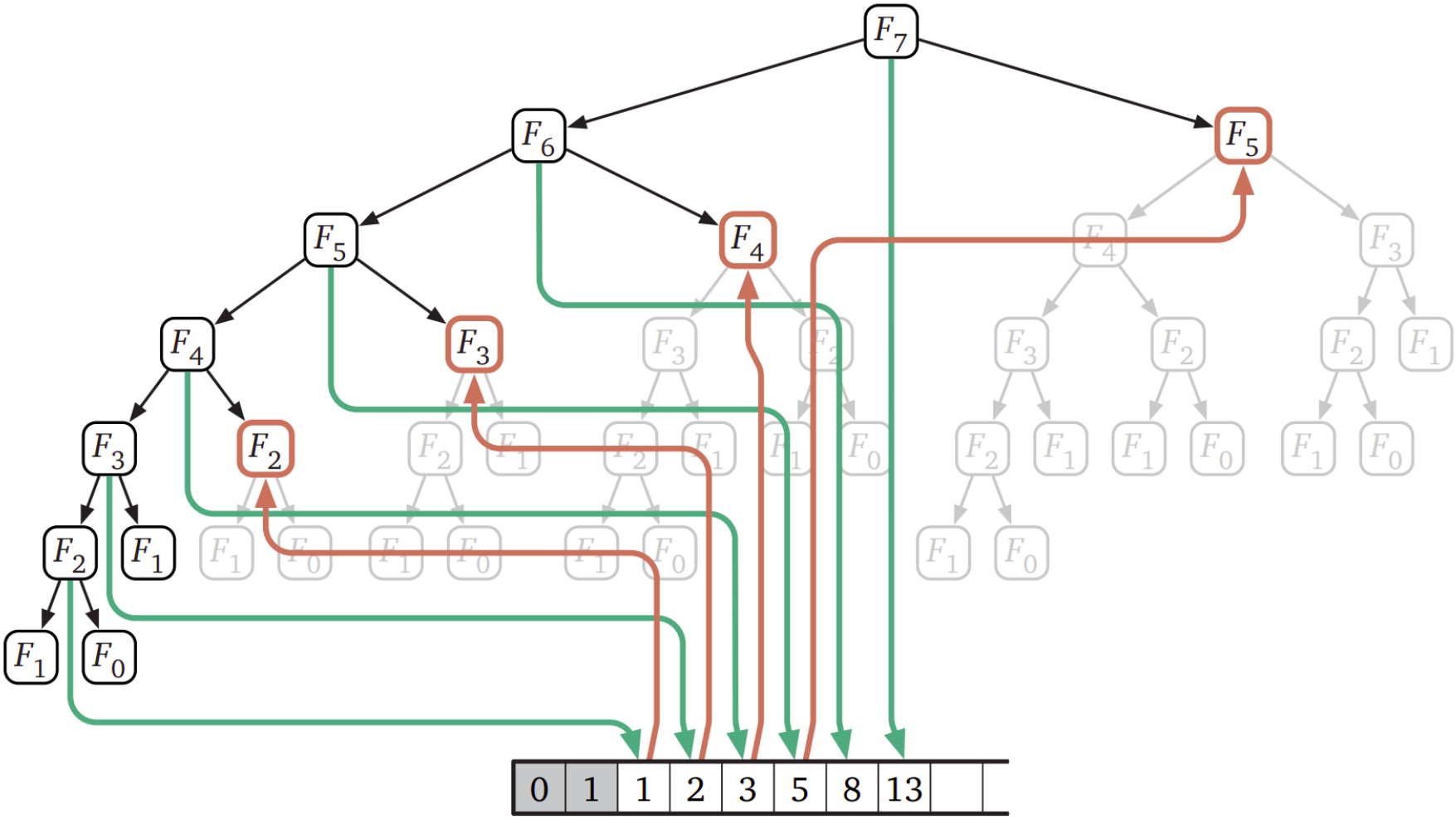
Idée : stocker les valeurs des sous-problèmes pour éviter de les calculer plusieurs fois.

```
def fibo(n):
    L = [0, 1] # L[i] contient le ième terme de Fibonacci
    for i in range(n - 2):
        L.append(L[-1] + L[-2])
    return L[n]
```

Remarque : `L[-k]` est le $k$ème élément de `L` en partant de la fin.

Complexité : linéaire en `n`.

Sous-problèmes



Sous-problèmes

Dans le cas de la suite de Fibonacci, on peut stocker seulement les 2 derniers termes :

```
def fibo(n):
    f0, f1 = 0, 1 # f0 et f1 contiennent les deux derniers termes
    for i in range(n - 2):
        f0, f1 = f1, f0 + f1
    return f1
```

Programmation dynamique

Pour résoudre un problème de programmation dynamique :

1. Chercher une équation de récurrence. Souvent, cela demande d'introduire un paramètre.
2. Stocker en mémoire les résultats des sous-problèmes pour éviter de les calculer plusieurs fois.

Programmation dynamique

Exemples d'utilisation de la programmation dynamique :

- Problème du sac à dos
- Plus courts chemins dans un graphe pondéré : Bellman-Ford et Floyd-Warshall

Programmation dynamique : Problème du sac à dos

Problème du sac à dos

Entrée : un sac à dos de capacité c, des objets $o_1, ..., o_n$ de poids $w_1, ..., w_n$ et valeurs v_1, ..., v_n.
Sortie : la valeur maximum que l'on peut mettre dans le sac.

Exemple

On considère un sac à dos de capacité 10kg et les objets suivants :

poids (kg)	2	2	2	3	5	5	8
valeur (€)	1	1	1	7	10	10	13

Quelle est la valeur maximum que l'on peut mettre dans le sac ?

Programmation dynamique : Problème du sac à dos

Problème (sac à dos)

Entrée : un sac à dos de capacité c, des objets $o_1, ..., o_n$ de poids $w_1, ..., w_n$ et valeurs v_1, ..., v_n.
Sortie : la valeur maximum que l'on peut mettre dans le sac.

Soit $dp[i][j]$ la valeur maximum que l'on peut mettre dans un sac de capacité i, en ne considérant que les objets o_1, ..., o_j.

$$dp[i][0] = 0$$

$$dp[i][j] = \max(\underbrace{dp[i][j-1]}_{\text{sans prendre } o_j}, \underbrace{dp[i-w_j][j-1] + v_j}_{\text{en prenant } o_j, \text{si } i-w_j \geq 0})$$

Programmation dynamique : Problème du sac à dos

Résolution du sac à dos par programmation dynamique

Pour $i = 0$ à c:
    $dp[i][0] \leftarrow 0$

Pour $i = 1$ à c:
    Pour $j = 1$ à n:
        Si $i - w_j \geq 0$:
            $dp[i][j] \leftarrow \max(dp[i][j-1], dp[i-w_j][i-1] + v_j)$
        Sinon:
            $dp[i][j] \leftarrow dp[i][j-1]$

Complexité : $O(nc)$ (exponentiel en $\lceil \log_2(c+1) \rceil$)

Programmation dynamique : Problème du sac à dos

```python
def knapsack(c, w, v):
    """
    Renvoie la valeur maximum que l'on peut mettre
    dans un sac à dos de capacité c.
    Le ième objet a pour poids w[i] et valeur v[i].
    """
    n = len(w) # nombre d'objets
    dp = [[0]*(n + 1) for i in range(c + 1)]
    # dp[i][j] = valeur max dans un sac de capacité i
    # où j est le nombre d'objets autorisés
    for i in range(1, c + 1):
        for j in range(1, n + 1):
            if w[j - 1] <= i:
                x = v[j - 1] + dp[i - w[j - 1]][j - 1]
                dp[i][j] = max(dp[i][j - 1], x)
            else:
                dp[i][j] = dp[i][j - 1]
    return dp[c][n]
```

Programmation dynamique : Problème du sac à dos

Comme on a juste besoin de stocker `dp[...][k - 1]` pour calculer `dp[...][k]` :

```
def knapsack2(c, w, v):
    n = len(w)
    dp = [0]*(c + 1)
    for j in range(n):
        dp_ = dp[:] # copie de dp
        for i in range(c + 1):
            if w[j] <= i:
                dp[i] = max(dp_[i], v[j] + dp_[i - w[j]])
    return dp[-1]
```

Mémoïsation

La programmation dynamique est une stratégie **bottom-up** : on résout les problèmes du plus petit au plus grand. On utilise des boucles for.

La **mémoïsation** est similaire mais avec une stratégie **top-down** : on part du problème initial pour le décomposer. On utilise des appels récursifs.

Pour éviter de résoudre plusieurs fois le même problème (comme pour Fibonacci), on mémorise (dans un tableau ou un dictionnaire) les arguments pour lesquelles la fonction récursive a déjà été calculée.

Mémoïsation

Version mémoïsée du calcul naïf de la suite de Fibonacci :

```
def fibo(n):
    f = {} # f[k] contiendra le kème terme de la suite
    def aux(k):
        if k < 2:
            return k
        if k not in d:
            f[k] = aux(k - 1) + aux(k - 2)
        return f[k]
    return aux(n)
```

Mémoïsation

Il est possible de « mémoïser » automatiquement une fonction.
En Python 3.10 :

```
from functools import cache
@cache
def f(n):
    if n <= 1:
        return n
    return f(n-1) + f(n-2)
```

Mémoïsation

Version récursive et mémoïsée pour le sac à dos :

```
def knapsack_memo(c, w, v):
    dp = {}
    def aux(i, j):
        if i == 0 or j == 0:
            return 0
        if (i, j) not in dp:
            dp[(i, j)] = aux(i, j - 1)
            if w[j - 1] <= i:
                x = v[j - 1] + aux(i - w[j - 1], j - 1)
                dp[(i, j)] = max(dp[(i, j)], x)
        return dp[(i, j)]
    return aux(c, len(w))
```

Plus courts chemins par programmation dynamique

On considère ici seulement des graphes orientés.

Définition

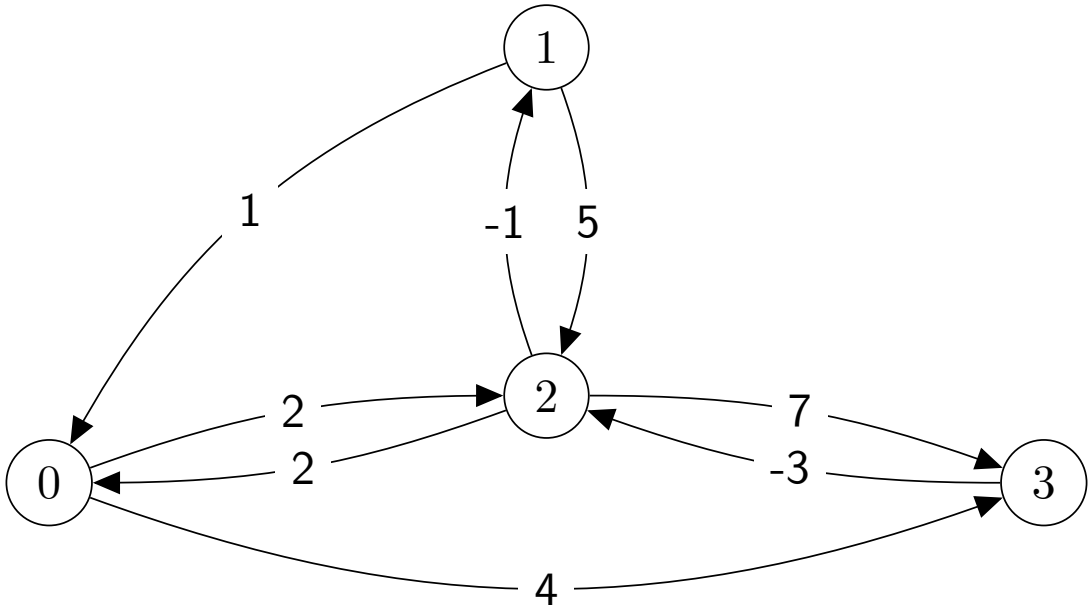
Un graphe **pondéré** est un graphe $\vec{G} = (V, \vec{E})$ muni d'une fonction de poids $w : \vec{E} \longrightarrow \mathbb{R}$.
$w(u, v)$ est le **poids** de l'arête de u vers v.

Il est pratique de définir $w(u, v) = \infty$ s'il n'y a pas d'arête entre u et v.
En Python, on peut utiliser `float("inf")` pour représenter $+\infty$.

Pour représenter un graphe pondéré, on utilisera une **matrice d'adjacence pondéré**, contenant $w(u, v)$ sur la ligne u, colonne v.

Plus courts chemins par programmation dynamique

Exemple de graphe représenté par matrice d'adjacence pondérée :



$$\begin{pmatrix} 0 & \infty & 2 & 4 \\ 1 & 0 & 5 & \infty \\ 2 & -1 & 0 & 7 \\ \infty & \infty & -3 & 0 \end{pmatrix}$$

```
G = [
    [0, float("inf"), 2, 4],
    [1, 0, 5, float("inf")],
    [2, -1, 0, 7],
    [float("inf"), float("inf"), -3, 0]
]
```

Plus courts chemins par programmation dynamique

Vous avez déjà vu plusieurs algorithmes pour trouver des plus courts chemins :

- **Parcours en largeur** (BFS), si tous les poids sont égaux (distance = nombre d'arêtes).
- **Dijkstra**, si tous les poids sont positifs.

Nous allons voir deux algorithmes supplémentaires, par programmation dynamique :

- **Bellman-Ford**, avec des poids quelconques.
- **Floyd-Warshall**, avec des poids quelconques.

Floyd-Warshall trouve toutes les distances entre deux sommets quelconques, contrairement aux autres algorithmes (qui calculent les distances depuis un sommet de départ).

Plus courts chemins par programmation dynamique

Algorithme	**Précondition**	**Complexité**
Parcours en largeur	Tous les poids égaux (distance = nombre d'arêtes)	$O(n+p)$
Dijkstra	Poids positifs	$O(p \log(n))$
Bellman-Ford	Poids quelconque	$O(np)$
Floyd-Warshall	Poids quelconque	$O(n^3)$

Floyd-Warshall trouve toutes les distances entre deux sommets quelconques, contrairement aux autres algorithmes (qui calculent les distances depuis un sommet de départ).

Notre problème est P = « trouver $d(u, v)$, pour des sommets u, v ».

On peut écrire l'équation :

$$d(u, v) = \min_{v'} d(u, v') + w(v', u)$$

On ne se ramène pas à des sous-problèmes plus petits !
Il faut introduire un paramètre :

1. Bellman : nombre d'arêtes
2. Floyd-Warshall : numéros des sommets que l'on peut utiliser

Plus courts chemins par programmation dynamique : Bellman-Ford

Soit $d_k(v)$ le poids minimum d'un chemin de r à v **utilisant au plus** k **arêtes**.

$$d_{k+1}(v) = \min_{(u,v)\in E} d_k(u) + w(u,v)$$

Preuve : soit C un plus court chemin de r à v utilisant au plus $k+1$ arêtes.
Soit u le prédecesseur de v dans C.
Alors le sous-chemin de C de r à u est un plus court chemin utilisant au plus k arêtes (s'il y avait un chemin plus court que C', on pourrait le remplacer dans C ce qui contredirait la minimalité de C).

Remarque : c'est une propriété de **sous-structure optimale** (un sous-chemin d'un plus court chemin est aussi un plus court chemin).

Plus courts chemins par programmation dynamique : Bellman-Ford

On va utiliser un tableau `d[v][k]` pour stocker $d_k(v)$.

Algorithme de Bellman-Ford

```
d[r] = 0
Pour v ≠ r :
    Pour k = 0 à n − 2 :
        d[v][k] = ∞

Pour k = 0 à n − 2 :
    Pour tout sommet v :
        Pour tout arc (u, v) entrant dans v :
            Si d[u][k] + g[u][v] < d[v][k + 1] :
                d[v][k + 1] = d[u][k] + g[u][v]
```

Complexité : $O(np)$

Plus courts chemins par programmation dynamique : Bellman-Ford

Parcourir tous les sommets puis tous les arcs (u, v) entrants dans v revient à parcourir tous les arcs du graphe :

Algorithme de Bellman-Ford

```
d[r] = 0
Pour v ≠ r :
    Pour k = 0 à n − 2 :
        d[v][k] = ∞

Pour k = 0 à n − 2 :
    Pour tout arc (u, v) :
        Si d[u][k] + g[u][v] < d[v][k + 1] :
            d[v][k + 1] = d[u][k] + g[u][v]
```

Plus courts chemins par programmation dynamique : Bellman-Ford

Comme on a juste besoin de stocker $d[...][k-1]$ pour calculer $d[...][k]$:

Algorithme de Bellman-Ford

```
d[r] = copie de 0
Pour v ≠ r :
    d[v] = ∞

Pour k = 0 à n − 2 :
    d' ← copie de d
    Pour tout arc (u, v) :
        Si d[u] + g[u][v] < d[v] :
            d[v] = d[u] + g[u][v]
```

Plus courts chemins par programmation dynamique : Bellman-Ford

matrice d'adj

```
def bellman(g, r):
    n = len(g)
    d = [float("inf")]*n
    d[r] = 0
    for k in range(n - 2):
        for u in range(n):
            for v in range(len(g[u])):
                if g[u][v] != ∅:
                    d[v] = min(d[v], d[u] + g[u][v])
    return d
```

↳ distances depuis r.

float("inf")

Plus courts chemins par programmation dynamique : Floyd-Warshall

Soit $d_k(u, v)$ la longueur d'un plus court chemin de u à v n'utilisant que des sommets intermédiaires de numéro $< k$ (∞ s'il n'existe pas).
On va résoudre P(k) = « trouver $d_k(u, v)$, pour tous sommets u, v ».

Soit C un plus court chemin de u à v n'utilisant que des sommets intermédiaires de numéro $< k + 1$.

1. Si C n'utilise pas k comme sommet intermédiaire :

$$d_{k+1}(u, v) = d_k(u, v)$$

2. Si C utilise k comme sommet intermédiaire :

$$d_{k+1}(u, v) = d_k(u, k) + d_k(k, v)$$

Équation de récurrence :

$$\boxed{d_{k+1}(u, v) = \min(d_k(u, v), d_k(u, k) + d_k(k, v))}$$

Plus courts chemins par programmation dynamique : Floyd-Warshall

Initialiser $d_0(u, v) \leftarrow g[u][v]$ si $(u, v) \in \vec{E}$.
∞ sinon.

Pour `k` $= 0$ à $n - 1$:
    Pour tout sommet u :
        Pour tout sommet v :
            $d_{k+1}(u, v) \leftarrow \min(d_k(u, v), d_k(u, k) + d_k(k, v))$

On peut utiliser un tableau `d` à 3 dimensions pour stocker $d_k(u, v)$ dans `d[u][v][k]`.

On a en fait juste besoin de d_k pour calculer d_{k+1} : on peut donc utiliser une matrice `d` telle que `d[u][v]` contient le dernier $d_k(u, v)$ calculé (ça marche car $d_{k+1}(u, k) = d_k(u, k)$).

Plus courts chemins par programmation dynamique : Floyd-Warshall

On utilise une matrice `d` telle que `d[u][v]` contienne le dernier $d_k(u, v)$ calculé :

Algorithme de Floyd-Warshall

```
d = copie de g

Pour k = 0 à n − 1 :
    Pour tout sommet u :
        Pour tout sommet v :
            d[u][v] = min(d[u][v], d[u][k] + d[k][v])
```

Complexité : $\boxed{O(n^3)}$

Plus courts chemins par programmation dynamique : Floyd-Warshall

Initialiser `d[u][v]` $\leftarrow g[u][v]$ si $(u, v) \in \vec{E}$, ∞ sinon.

Pour `k` $= 0$ à $n - 1$:
    Pour tout sommet u :
        Pour tout sommet v :

```
d[u][v] = min(d[u][v], d[u][k] + d[k][v])
```

Question

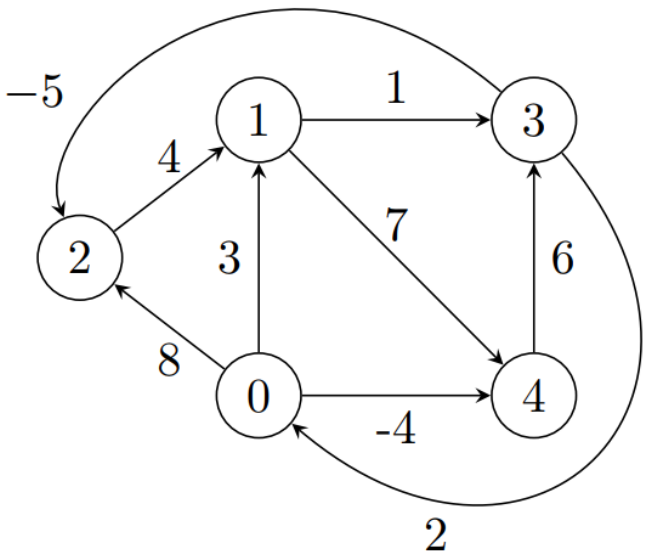
Comment détecter un cycle de poids négatif?

Il y a un cycle de poids négatif $\iff \exists$ `u`, `d[u][u] < 0`.

Plus courts chemins par programmation dynamique : Floyd-Warshall

On utilise souvent une matrice d'adjacence $A = (a_{i,j})$ modifée pour représenter un graphe $\vec{G} = (V, \vec{E})$ pondéré par w :

- $a_{i,j} = w(i,j)$, si $(i,j) \in \vec{E}$
- $a_{i,j} = 0$, si $i = j$
- $a_{i,j} = \infty$, si $(i,j) \notin \vec{E}$



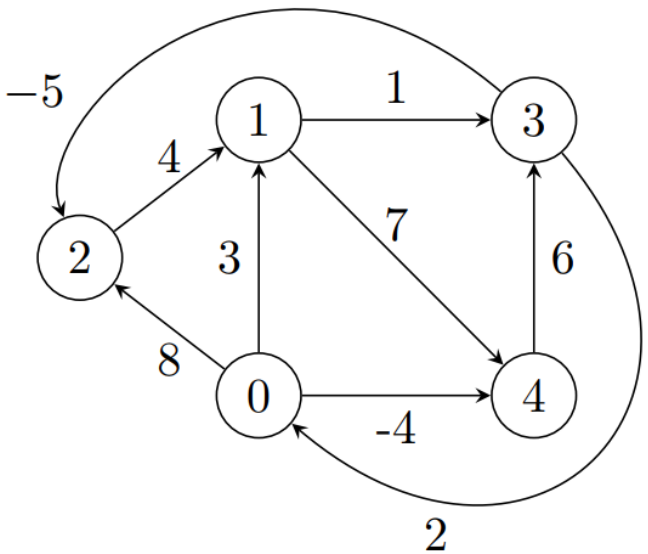
$$A = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Matrice d'adjacence

Plus courts chemins par programmation dynamique : Floyd-Warshall

On utilise souvent une matrice d'adjacence $A = (a_{i,j})$ modifée pour représenter un graphe $\vec{G} = (V, \vec{E})$ pondéré par w :

- $a_{i,j} = w(i,j)$, si $(i,j) \in \vec{E}$
- $a_{i,j} = 0$, si $i = j$
- $a_{i,j} = \infty$, si $(i,j) \notin \vec{E}$



$$\begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Matrice des distances renvoyée par Floyd-Warshall

Plus courts chemins par programmation dynamique : Floyd-Warshall

On suppose que le graphe d est représenté par matrice d'adjacence pondérée :

```
import copy

def floydwarshall(g):
    n = len(g)
    d = copy.deepcopy(g) # pour éviter de modifier g
    for k in range(n):
        for u in range(n):
            for v in range(n):
                d[u][v] = min(d[u][v], d[u][k] + d[k][v])
    return d
```

Plus courts chemins par programmation dynamique : Floyd-Warshall

Initialiser `d[u][v]` $\leftarrow g[u][v]$ si $(u, v) \in \vec{E}$, ∞ sinon.

Pour `k` $= 0$ à $n - 1$:
    Pour tout sommet u :
        Pour tout sommet v :
            `d[u][v] = min d[u][v] (d[u][k] + d[k][v])`

Question

Comment connaître un plus court chemin de n'importe quel sommet u à n'importe quel un autre v?

Utiliser une matrice `pere` telle que `pere.(u).(v)` est le prédécesseur de `v` dans un plus court chemin de `u` à `v`.

Plus courts chemins par programmation dynamique : Floyd-Warshall

Soit $d_k(u, v)$ la longueur d'un plus court chemin de u à v n'utilisant que des sommets intermédiaires de numéro $< k$ (∞ s'il n'existe pas).
Soit $p_k(u, v)$ un prédécesseur de v dans un plus court chemin de u à v n'utilisant que des sommets intermédiaires de numéro $< k$.
Soit C un plus court chemin de u à v n'utilisant que des sommets intermédiaires de numéro $< k + 1$.

1. Si C n'utilise pas k comme sommet intermédiaire :

$$d_{k+1}(u, v) = d_k(u, v)$$

$$p_{k+1}(u, v) = p_k(u, v)$$

2. Si C utilise k comme sommet intermédiaire :

$$d_{k+1}(u, v) = d_k(u, k) + d_k(k, v)$$

$$p_{k+1}(u, v) = p_k(k, v)$$

Plus courts chemins par programmation dynamique : Floyd-Warshall

Initialiser `d[u][v]` $\leftarrow w($`u, v`$)$ si $($`u, v`$) \in \vec{E}$, ∞ sinon.
Initialiser `pere.(u).(v)` = `u`, $\forall$ `u, v` $\in V$.

Pour `k` $= 0$ à $n - 1$:
    Pour tout sommet u :
        Pour tout sommet v :
            Si `d[u][v] > d[u][k] + d[k][v]` :
                `d[u][v] = d[u][k] + d[k][v]`
                `pere.(u).(v) = pere.(k).(v)`

On obtient une matrice `pere` telle que `pere.(u).(v)` est le prédécesseur de `v` dans un plus court chemin de `u` à `v`.