Programmation dynamique

INFORMATIQUE COMMUNE - TP nº 3.2 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ cerner les limitations des algorithmes gloutons dans certaines situations
- ☞ justifier l'optimalité d'une sous-structure d'un problème en programmation dynamique
- ☞ résoudre le problème en construisant un tableau de résolution (approche ascendante)
- ☞ utiliser la mémoïsation (approche descendante récursive)

A Le sac à dos est de retour

On cherche à remplir un sac à dos. Chaque objet que l'on peut insérer dans le sac est **insécable**[1] et possède une valeur et un poids connus. On cherche à maximiser la valeur totale emportée dans la sac à dos tout en limitant[2] le poids à π.

Soit un ensemble $\mathcal{O}_n = \{o_1, o_2, \ldots, o_n\}$ de n objets de valeurs $v_1, v_2, \ldots, v_n$ et de poids respectifs $p_1, p_2, \ldots, p_n$. Soit un sac à dos n'admettant pas un poids emporté supérieur à π. On note également qu'on peut mettre au plus n objets dans le sac.

Les objets sont rangés dans une liste et dans un ordre quelconque. Ils sont indicés par i variant de 1 à n. Un objet o_i possède une valeur v_i et un pèse p_i.

Avec ces notations, on peut formuler le problème du sac à dos comme suit.

■ **Définition 1 — Problème du sac à dos.** Comment remplir un sac à dos en maximisant la valeur totale emportée V tout en ne dépassant pas le poids maximal π admissible par le sac à dos.

Formellement, comment maximiser $V = \sum_{o_i \in B} v_i$ en respectant la contrainte $\sum_{o_i \in B} p_i \leqslant \pi$ où B est l'ensemble des objets emportés dans le sac ?

On note[a] le problème du sac à dos KP (n, π) et une solution optimale à ce problème $S(n, \pi)$.

a. en anglais, ce problème est nommé Knapsack Problem, d'où le KP.

On dispose d'une collection d'objets `o = (valeur, poids)` :

```
OBJETS = ((100, 40), (700, 15), (500, 2), (400, 9), (300, 18), (200, 2))
```

A1. Pour une poids maximal admissible de 11 kg, quel peut-être un chargement optimal du sac à dos ?

> **Solution :** On peut mettre au maximum pour une valeur de 900, en prenant le quatrième et le troisième objet (9+2 = 11).

1. Soit on le met dans le sac, soit on ne le met pas. Mais on ne peut pas en mettre qu'une partie.
2. On accepte un poids total inférieur ou égal à π.

A2. Pour le problème KP(n,π), implémenter un algorithme de résolution glouton. La signature de la fonction est `glouton_kp(objets, pmax)`. Quelle est la complexité de votre fonction ?

Solution : La complexité est en $O(n\log n+n)$ si on compte le tri de la liste en $O(n\log n)$ Sinon, la complexité est linéaire. Ce qui est bien, mais cette algorithme ne trouve pas toujours la solution optimale.

```python
OBJETS = ((100, 40), (700, 15), (500, 2), (400, 9), (300, 18), (200, 2))

def glouton_kp(objets, pmax):
    poids = 0  # poids total du sac
    valeur = 0  # valeur total du sac
    sac = []  # contenu du sac
    objets = sorted(list(objets))  # tri ascendant en fonction de la valeur
    while len(objets) > 0 and poids <= pmax:
        v, p = objets.pop()  # choix glouton : la valeur la plus grande
        if poids + p <= pmax:  # l'objet peut-il entrer dans le sac
            sac.append((v, p))
            poids += p
            valeur += v
    return sac, poids, valeur


def glouton_kp(objets, pmax):
    poids = 0  # poids total du sac
    valeur = 0  # valeur total du sac
    sac = []  # contenu du sac
    objets = sorted(list(objets))  # tri ascendant en fonction de la valeur
    i = 0  # Variable pour parcourir la liste objets
    while i < len(objets) and poids <= pmax:
        v, p = objets[len(objets) - i - 1]  # choix glouton : la valeur la
            plus grande
        if poids + p <= pmax:  # l'objet peut-il entrer dans le sac ?
            sac.append((v, p))  # on l'ajoute
            poids += p
            valeur += v
        i = i + 1
    return sac, poids, valeur

print(glouton_kp(OBJETS, 15))
print(glouton_kp(OBJETS, 11))
```

A3. Le problème du sac à dos est-il à sous-structure optimale ?

Solution : Oui, car on peut exprimer une solution optimale au problème en fonction des solutions optimales des sous-problèmes.

$$S(n,\pi)=\begin{cases}0 & \text{si } n=0 \text{ ou si } \pi=0\\ \max\big(v_n+S(n-1,\pi-p_n),S(n-1,\pi)\big) & \text{si } p_n\leqslant\pi\\ S(n-1,\pi) & \text{sinon}\end{cases}\qquad(1)$$

A4. En utilisant construisant un tableau de résolution, coder la résolution du problème KP(n,π) selon l'approche ascendante. La signature de la fonction est `dyn_kp(objets, pmax)`. Quelle est la complexité de votre fonction ?

Solution : La complexité est en $O(n \times pmax)$ car la première boucle effectue `n` itérations et la seconde `pmax`.

```
import numpy as np

OBJETS = ((100, 40), (700, 15), (500, 2), (400, 9), (300, 18), (200, 2))

def dyn_kp(objets, pmax):  # Programmation dynamique
    n = len(objets)
    s = np.zeros((n + 1, pmax + 1))
    for i in range(n + 1):
        for p in range(pmax + 1):
            if i == 0 or p == 0:
                s[i, p] = 0  # 0 kg, pas de valeur. 0 objets, pas de valeur
            else:
                vi, pi = objets[i - 1]  # on considère le ième objet
                if pi <= p:
                    s[i, p] = max(vi + s[i - 1, p - pi], s[i - 1, p])
                else:
                    s[i, p] = s[i - 1, p]
    return s[n, pmax]


def dyn_kp(objets, pmax):  # Programmation dynamique
    n = len(objets)
    s = [[0 for _ in range(pmax + 1)] for _ in range(n + 1)]
    for i in range(n + 1):
        for p in range(pmax + 1):
            if i == 0 or p == 0:
                s[i][p] = 0  # pas d'objet pas de solution
            else:
                vi, pi = objets[i - 1]  # on considère le ième objet
                if pi <= p:
                    s[i][p] = max(vi + s[i - 1][p - pi], s[i - 1][p])
                else:
                    s[i][p] = s[i - 1][p]
    return s[n][pmax]

print(dyn_kp(OBJETS, 15))
print(dyn_kp(OBJETS, 11))
```

A5. En utilisant la programmation dynamique récursive descendante, coder la résolution du problème KP(n,π). La fonction a pour signaturte `kp_mem(n, pmax, S)`. On cherchera à ne pas recalculer plusieurs fois les mêmes sous-problèmes en utilisant un dictionnaire. On considèrera la variable `OBJETS` comme une constante globale accessible depuis la fonction.

Solution :

```
OBJETS = ((100, 40), (700, 15), (500, 2), (400, 9), (300, 18), (200, 2))
```

```python
def kp_rec(n, pmax):
    if n == 0 or pmax == 0:
        return 0
    else:
        v, p = OBJETS[n - 1]
        if p > pmax:
            return kp_rec(n - 1, pmax)
        else:
            return max(v + kp_rec(n - 1, pmax - p), kp_rec(n - 1, pmax))


def kp_mem(n, pmax, S):  # memoïsation
    if (n, pmax) in S:  # déjà mémorisé
        return S[(n, pmax)]
    elif n == 0 or pmax == 0:
        return 0
    else:
        v, p = OBJETS[n - 1]
        if p > pmax:
            S[(n, pmax)] = kp_mem(n - 1, pmax, S)
            return S[(n, pmax)]
        else:
            S[(n, pmax)] = max(v + kp_mem(n - 1, pmax - p, S),
                               kp_mem(n - 1, pmax, S))
            return S[(n, pmax)]

print(kp_rec(6, 15))
print(kp_rec(6, 11))
print(kp_mem(6, 15, {}))
print(kp_mem(6, 11, {}))
```

B Rendu de monnaie

Un commerçant doit à rendre la monnaie à un client[3]. La somme à rendre est un somme entière P et le commerçant cherche à utiliser le moins de pièces possibles. On considère qu'il dispose d'autant de pièces qu'il le souhaite parmi un système monétaire $M = \{m_1, m_2, \ldots, m_n\}$ qui possède n valeurs différentes.

On nomme ce problème le rendu de monnaie[4] et on note CCP(M, i, P) le problème où il s'agit de rendre la monnaie P à l'aide des i premières pièces du système M. On note $S(i, P)$ une solution optimale au problème CCP(M, i, P), c'est à dire une solution qui nécessite **le moins de pièces possibles**.

B1. On considère les systèmes monétaire $M_c = (5, 1, 2)$ et $M = (4, 1, 3)$. Rendre la monnaie de manière optimale sur 10 et 6 avec chaque système.

3. Mais on pourrait considérer d'autres problèmes qui se résoudraient de la même manière. Par exemple, le remplissage d'un conteneur dont le volume total est V à l'aide d'objets de volume $v_1, v_2, \ldots, v_n$. On dispose d'autant d'objets que l'on veut pour compléter le conteneur mais on souhaite en charger le moins possible.

4. En anglais Coin Change Problem.

Solution : On note les résultats sous la forme d'une liste de couple représentant la valeur de la pièce et le nombre d'occurrences : `[(valeur, nombre)]`.

- 10 : M_c : `[(5, 2)]` M : `[(3,2),(2,1)]`
- 6 : M_c : `[(2, 2), (1,1)]` M : `[(3,2)]`

B2. Résoudre le problème CCP(M, n, P) en implémentant un algorithme glouton. Tester l'algorithme sur les systèmes M_c et M. Que constatez-vous ?

Solution : Cet algorithme glouton est optimal avec M_c mais pas avec M. Par exemple, pour 6 avec M, l'algorithme glouton trouve `[(4,2), (1,2)]`, ce qui n'est pas l'optimal puisqu'il faut trois pièces alors qu'on peut le faire avec deux seulement.

```python
# PIECES = (5, 1, 2)
PIECES = (4, 1, 3)

def greedy_ccp(prix):
    solution = {}
    nb_de_pieces = 0
    pieces = sorted(list(PIECES))
    while len(pieces) > 0 and not prix == 0:
        m = pieces.pop()  # on prend la valeur la plus grande
        n = prix // m  # combien de fois ?
        if n > 0:
            solution[m] = n
            prix = prix - n * m  # continue...
            nb_de_pieces += n
    if prix == 0:  # success
        return [nb_de_pieces, solution]
    else:
        return None  # no solution

def greedy_rec_ccp(pieces, prix, solution):
    if prix == 0:
        return solution
    elif len(pieces) == 0:
        return solution
    else:
        m = pieces.pop()  # la plus grande valeur
        n = prix // m  #
        if n > 0:
            solution[m] = n
        return greedy_rec_ccp(pieces, prix - n * m, solution)  # continue...
```

B3. Le problème du rendu de monnaie est-il à sous-structure optimale ? Pourquoi ?

Solution : Oui, car on peut exprimer une solution optimale du problème en fonction des solutions optimales des sous-problèmes.

$$S(i,\pi) = \begin{cases} 0 & \text{si } \pi = 0 \quad \text{Il suffit de rendre zéro pièces.} \\ \infty & \text{si } i = 0 \quad \text{Il n'y pas de solution, coût max.} \\ \min(1 + S(i,\pi - m_i), S(i-1,\pi)) & \text{si } m_i \leqslant \pi \\ S(i-1,\pi) & \text{sinon} \end{cases} \tag{2}$$

B4. En utilisant la programmation dynamique, coder la résolution du problème CCP(M,n,P) en construisant un tableau de résolution. On utilisera un dictionnaire pour mémoriser la liste des pièces nécessaires. (Question difficile)

Solution :

```python
import math

# PIECES = (5, 1, 2)
PIECES = (4, 1, 3)


def dyn_ite_ccp(prix):
    n = len(PIECES)
    S = [[[math.inf, {}] for i in range(prix + 1)] for i in range(n + 1)]
    # On cherche à connaître la solution optimale ET le détail de la
        répartition des pièces
    # math.inf est utilisable avec la fonction min
    for p in range(1, prix + 1):
        for i in range(1, n + 1):
            mi = PIECES[i - 1]  # on considère la ième pièce
            if mi <= p:
                a = 1 + S[i][p - mi][0]  # avec
                b = S[i - 1][p][0]  # sans
                S[i][p][0] = min(a, b)
                if a <= b:
                    S[i][p][1] = S[i][p][1] | S[i][p - mi][1]
                    if mi in S[i][p][1]:
                        S[i][p][1][mi] += 1
                    else:
                        S[i][p][1][mi] = 1
                else:
                    S[i][p][1] = S[i][p][1] | S[i - 1][p][1]
            else:
                S[i][p] = S[i - 1][p]
    return S[n][prix]

print(dyn_ite_ccp(6))
```

B5. En comparant les deux stratégies précédentes, identifier les cas pour lesquels l'algorithme glouton n'est pas optimal pour les systèmes monétaires M et M_c définis plus haut.

C Distance d'édition

Les séquences de caractères peuvent encoder de nombreuses informations de nature différente, par exemple du texte, de la voix ou des séquences ADN. L'alignement de deux chaînes des caractères consiste à comparer deux séquences de caractères afin d'évaluer la similarité entre les deux.

La distance d'édition ou distance de Levenshtein est une mesure de la similarité entre deux chaînes de caractères. Cette distance est le nombre minimal de caractères qu'il faut supprimer, insérer ou substituer pour passer d'une chaîne à l'autre.

■ **Définition 2 — Distance d'édition.** Soit a et b deux chaînes de caractères. On note $|a|$ le cardinal de a, c'est-à-dire le nombre de caractères de la chaîne. $a[-1]$ désigne le dernier caractère de la chaîne a. On dénote par $a[:-1]$ la chaîne a tronquée de son dernier caractère.

On suppose que :

- supprimer un caractère,
- insérer un caractère,
- substituer un caractère,

sont des opérations qui ont toute un coût **unitaire (1)**. Si le caractère est identique, la substitution ne coûte rien (0).

La distance d'édition est définition par induction de la manière suivante :

$$d_e(a,b) = \begin{cases} \max(|a|,|b|) & \text{si } \min(|a|,|b|) = 0 \\ d(a[:-1], b[:-1]) & \text{si } a[-1] = b[-1] \\ 1 + \min \begin{cases} d(a[:-1], b) \\ d(a, b[:-1]) \\ d(a[:-1], b[:-1]) \end{cases} & \text{sinon} \end{cases} \tag{3}$$

On souhaite calculer la distance d'édition en programmant dynamiquement de manière ascendante en utilisant un tableau S. Chaque case du tableau S contient la distance entre la chaîne constituée des i premiers caractères de a et la chaîne constituée des j premiers caractères de b. Il est nécessaire d'exprimer le résultat d'une case en fonction de celles dont elle dépend dans le schéma dynamique :

$$S()i,j) = \begin{cases} \max(i,j) & \text{si } \min(i,j) = 0 \\ S(i-1, j-1) & \text{si a[i-1] = b[j-1], les caractères sont les mêmes} \\ 1 + \min(S(i-1,j), S(i,j-1), S(i-1,j-1)) & \text{sinon} \end{cases} \tag{4}$$

C1. La distance d'édition de `"chien"` à `"niche"` vaut 4. Expliquer pourquoi.

> **Solution :** On écrit chien et niche l'un au dessous de l'autre. On opère sur `"niche"` : deux suppressions (n et i), deux substitutions (c et h), une insertion (i), une substitution (e) et une insertion (n). Comme les deux premières substitutions sont des correspondances (ce sont les mêmes lettres), elles ne coûtent rien. Donc la distance d'édition de ces deux mots vaut 4.

C2. La distance d'édition représente-t-elle un problème à sous-structure optimale ? Pourquoi ?

Solution : Oui car on arrive à exprimer une solution optimale en fonction des solutions optimales des sous-problèmes.

C3. On souhaite utiliser la programmation dynamique. Compléter à la main un tableau de résolution associé à la distance d'édition de `"chien"` à `"niche"`.

Solution : On note :

- x pour suppression, déplacement horizontal,
- s pour substitution, déplacement en diagonal,
- i pour insertion, déplacement vertical.

i \ j	0	1,n	2,i	3,c	4,h	5,e
5,n	5	4	4	4	4	4i
4,e	4	4	3	3	4	3s
3,i	3	3	2	3	3i	3
2,h	2	2	2	3	2s	3
1,c	1	1	2	2s	3	4
0	0	1x	2x	3	4	5

C4. Écrire un code qui calcule la distance d'édition de deux chaînes de caractères par programmation dynamique en construisant un tableau de résolution (approche ascendante). On pourra tester sur `"AGTTC"` et `"AGCTC"`, sur `"chien"` et `"niche"` ou `"sunday"` et `"saturday"`.

Solution :

```
import numpy as np

def de(a, b):
    S = np.zeros((len(a) + 1, len(b) + 1), dtype="int")
    for i in range(len(a) + 1):
        sa = a[:i]
        for j in range(len(b) + 1):
            sb = b[:j]
            if i == 0 or j == 0:
                S[i, j] = max(i, j)
            elif sa[-1] == sb[-1]:
                S[i, j] = S[i - 1, j - 1]
            else:
                S[i, j] = 1 + min(S[i - 1, j],S[i - 1, j - 1],S[i, j - 1])
    return S[len(a), len(b)]
```

C5. Écrire un code similaire en programmation dynamique avec memoïsation et comparer les résultats.

Solution :

```
import numpy as np

def mem_de(a, b, mem):
    if (a, b) in mem:
        return mem[(a, b)]  #  already computed !
    else:
        if len(a) == 0 or len(b) == 0:
            mem[(a, b)] = max(len(a), len(b))
        elif a[-1] == b[-1]:
            mem[(a, b)] = mem_de(a[:-1], b[:-1], mem)
        else:
            mem[(a, b)] = 1 + min(mem_de(a[:-1], b, mem), mem_de(a[:-1], b
                [:-1], mem), mem_de(a, b[:-1], mem))
        return mem[(a, b)]
```

D Plus longue sous-chaîne commune

La distance d'édition permet de mesurer le degré de similarité de deux chaînes. Elle ne donne pas d'information quant aux séquences maximales communes aux deux chaînes. Hors, en génétique par exemple, il peut s'avérer très important de savoir quels sont les points communs de deux génomes. Le problème de la plus longue sous-chaîne permet d'apporter une réponse à cette question.

■ **Définition 3 — Sous-chaîne.** On appelle sous-chaine d'une chaîne de caractères $a = a_1...a_n$ toute chaine de caractères s extraite de a telle que : $s = a_{i_1}...a_{i_k}$ où $(i_1, i_2, \dots i_k)$ est un sous-ensemble ordonné de $[\![1, n]\!]$ tel que $i_1 < i_2 < \dots < i_k$. Les caractères de s n'apparaissent pas nécessairement de manière consécutive dans la chaîne a.

■ **Définition 4 — Plus longue sous-chaîne commune.** Soit $a = a_1 \dots a_q$ et $B = b_1 \dots b_p$ deux chaines de caractères non vides. On appelle plus longue sous-chaine commune à a et b toute sous-chaîne commune à a et b de longueur maximale.

Si l'une des chaînes a ou b est vide ou si a et b n'ont aucune sous-chaîne commune, la chaine vide est alors l'unique plus longue sous-chaîne commune à a et b. Elle a pour longueur 0.

■ **Exemple 1 — Plus longue sous-chaîne commune.** Par exemple, les chaines de caractères `"AAA"` et `"TAA"` sont les plus longues sous-chaînes communes aux chaines de caractères `"ATAGA"` et `"TAACA"`.

Le problème de la plus longue sous-chaîne commune entre a et b est noté $\mathcal{L}(a, b)$, son résultat est la longueur maximale d'une sous-chaîne commune à a et b.

D1. On considère les chaines `a="AATGCG"` et `b="TATTAGC"` ? Donner les solutions de $\mathcal{L}(a, b)$.

Solution : ATGC et AAGC.

D2. Écrire une fonction de prototype `is_ss(ch,sch)` où les paramètres sont deux chaînes de caractères et qui renvoie `True` si `sch` est une sous-chaîne de `ch` et `False` sinon.

Solution :

```
def is_ss(ch, sch):
    i = 0 # pour parcourir ch
    j = 0 # pour parcourir sch à un autre rythme
    while i < len(ch) and j < len(sch):
        if ch[i] == sch[j]:
            j += 1  # un caractère commun
        i += 1
    return len(sch) == j
```

D3. Écrire une fonction de prototype `is_common_ss(a,b,sch)` où les paramètres sont des chaînes de caractères et qui renvoie `True` si `sch` est une sous-chaîne commune à `a` et `b`.

Solution :

```
def is_common_ss(a, b, sch):
  return is_ss(a, sch) and is_ss(b, sch)
```

D4. Formuler le problème $\mathcal{L}(a,b)$ récursivement afin de pourvoir justifier de sa sous-structure optimale.

Solution : Soit $S(i,j)$ la longueur de la plus longue sous-chaîne des chaînes extraites $a_1a_2\dots a_i$ et $b_1b_2\dots b_j$ des chaînes a et b. On a la récurrence :

$$\forall i \in [\![0,p]\!], \forall j \in [\![0,q]\!], S(i,j) = \begin{cases} 0 & \text{si } i=0 \text{ ou } j=0 \\ \max(S(i-1,j), S(i,j-1)) & \text{si } a_i \neq b_j \\ 1+S(i-1,j-1) & \text{si } a_i = b_j \end{cases} \tag{5}$$

D5. Écrire un code qui résout $\mathcal{L}(a,b)$ avec la programmation dynamique (approche ascendante), en complétant un tableau de résolution.

Solution :

```
import numpy as np

def dp_lss(a, b):
  S = np.zeros((len(a) + 1, len(b) + 1), dtype="int")
  for i in range(len(a) + 1):
    for j in range(len(b) + 1):
      if i == 0 or j == 0:
        S[i, j] = 0
      elif a[i - 1] == b[j - 1]:
        S[i, j] = 1 + S[i - 1, j - 1]
      else:
        S[i, j] = max(S[i - 1, j], S[i, j - 1])
```

```python
    return S[len(a), len(b)]
```

D6. Résoudre $\mathcal{L}(a,b)$ récursivement sans mémoïsation.

Solution :

```python
def rec_lss(a, b):
    if len(a) == 0 or len(b) == 0:
        return 0
    elif a[-1] == b[-1]:
        return 1+ rec_lss(a[:-1], b[:-1])
    else:
        return max(rec_lss(a[:-1], b), rec_lss(a, b[:-1]))
```

D7. Résoudre $\mathcal{L}(a,b)$ récursivement avec mémoïsation.

Solution :

```python
def mem_lss(a, b, m):
    global c  # pour compter les appels récursifs
    if (a, b) in m:
        return m[(a, b)]
    else:
        if len(a) == 0 or len(b) == 0:
            return 0
        elif a[-1] == b[-1]:
            c = c + 1
            m[(a, b)] = 1 + mem_lss(a[:-1], b[:-1], m)
            return m[(a, b)]
        else:
            c = c + 2
            m[(a, b)] = max(mem_lss(a[:-1], b, m), mem_lss(a, b[:-1], m))
            return m[(a, b)]
```

D8. Créer une fonction de type décorateur Python qui automatise la mémoïsation d'une fonction récursive. ⇢ HORS PROGRAMME

Solution :

```python
def memoize(f):
    memo = {}

    def aux(*x):
        if x not in memo:
            memo[x] = f(*x)
        return memo[x]

    return aux
```

```python
@memoize
def rec_lss(a, b):
    global c  # pour compter les appels récursifs
    if len(a) == 0 or len(b) == 0:
        return 0
    elif a[-1] == b[-1]:
        c = c + 1
        return 1 + rec_lss(a[:-1], b[:-1])
    else:
        c = c + 2
        return max(rec_lss(a[:-1], b), rec_lss(a, b[:-1]))
```

E Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall est l'application de la programmation dynamique à la recherche du plus court chemin entre deux sommets d'un graphe orienté et valué. Le plus court chemin n'est pas celui qui comporte le moins de sommets mais celui dont la somme des poids de chaque arc est la plus faible [5]. Les valuations peuvent être négatives mais on exclue tout circuit de poids strictement négatif.

R On ne considère que des graphes orientés et la raison est simple : si le graphe n'était pas orienté et possédait une pondération négative, cela signifierait qu'il existe un cycle de poids négatif : l'arête qui est pondérée négativement peut être parcourue dans les deux sens, c'est donc un cycle. Et donc, on ne pourrait pas justifier de l'existence d'un plus court chemin dans le graphe.

Soit un graphe orienté et pondéré $G = (S, A, w)$. G peut être modélisé par une matrice d'adjacence M

$$\forall i, j \in [\![0, |S|-1]\!], M = \begin{cases} w(i,j) & \text{si } (i,j) \in A \\ +\infty & \text{si } (i,j) \notin A \\ 0 & \text{si } i = j \end{cases} \tag{6}$$

Un exemple de graphe associé à la matrice d'adjacence M_{init} est donné sur la figure 1 :

$$M_{\text{init}} = \begin{pmatrix} 0 & 8 & +\infty & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & +\infty & 0 & +\infty \\ +\infty & 2 & -3 & 0 \end{pmatrix} \tag{7}$$

Pour trouver le plus court chemin entre deux sommets, on essaye tous les chemins de toutes les longueurs possibles et on ne garde que les plus courts. Chaque étape p de l'algorithme de Floyd-Warshall est donc constitué d'un allongement **éventuel** du chemin par le sommet. À l'étape p, on associe une matrice M_p qui contient la longueur des chemins les plus courts d'un sommet à un autre passant par des sommets de l'ensemble $\{0, 1, 2, \dots, p-1\}$. On construit ainsi une suite de matrice finie $\left(M_p\right)_{p \in [\![0,n]\!]}$ avec $M_0 = M$.

5. Dans un réseau de télécommunications, il s'agit bien du chemin le plus court si les poids des arcs sont les débits en Gbits/s des liens.

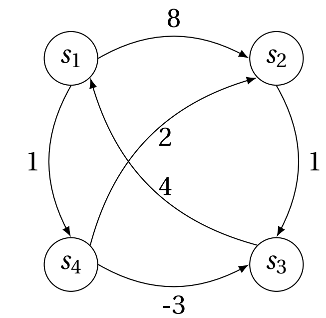

FIGURE 1 – Exemple de graphe orienté et valué associé à M_{init}.

Supposons qu'on dispose de M_p. Considérons un chemin $\mathcal{C}$ entre i et j dont la longueur est minimale et dont les sommets intermédiaires sont dans $\{0,1,2,\ldots,p-1\}$, $p \leqslant n$. Pour un tel chemin :

- soit le chemin le plus court passe par $p-1$. Dans ce cas, $\mathcal{C}$ est la réunion de deux chemins dont les sommets sont dans $\{1,2,\ldots,p-1\}$: celui de i à $p-1$ et celui de $p-1$ à j.
- soit $\mathcal{C}$ ne passe pas par $p-1$.

Entre ces deux chemins, on choisira le chemin le plus court.

Disposer d'une formule de récurrence entre M_p et M_{p-1} permettrait de montrer que le problème du plus court chemin entre deux sommets d'un graphe orienté et valué est à sous-structure optimale. On pourrait alors utiliser la programmation dynamique pour résoudre le problème.

E1. Formuler le problème du plus court chemin entre deux sommets d'un graphe orienté afin de montrer que ce problème est à sous-structure optimale.

Solution :

$$\forall p \in [\![1, n]\!], \forall i, j \in [\![0, n-1]\!], M_p(i,j) = \min\left(M_{p-1}(i,j), M_{p-1}(i,p-1) + M_{p-1}(p-1,j)\right) \quad (8)$$

Pour $p = 0$, on pose $M_0 = M_{\text{init}}$.

E2. Coder une fonction qui implémente l'algorithme de Floyd-Warshall selon l'approche ascendante en calculant la matrice à chaque étape. Tester ce code sur l'exemple de la figure 1. On pourra utiliser un tableau numpy à trois dimensions.

Solution :

```python
import numpy as np

def floyd_marshall(M):
  C = np.zeros((M.shape[0] + 1, M.shape[0], M.shape[0]))
  C[0, :, :] = M
  for p in range(1, M.shape[0]+1):
    for i in range(M.shape[0]):
      for j in range(M.shape[1]):
        C[p, i, j] = min(C[p-1, i, j], C[p-1, i, p-1] + C[p-1, p-1, j])
  return C[M.shape[0]]
```

Solution : `[[ 0. 3. -2. 1.] [ 5. 0. 1. 6.] [ 4. 7. 0. 5.] [ 1. 2. -3. 0.]]`

E3. Quelle est la complexité temporelle de cet algorithme ?

Solution : La complexité temporelle de cet algorithme est en $O(n^3)$, si n est le nombre de sommets du graphe.

E4. Quelle est la complexité spatiale de cet algorithme ? Pourrait-on l'améliorer ? Comment ?

Solution : La complexité spatiale de cet algorithme est en $O(n^3)$ si on procède ainsi. Par contre, on n'est pas obligé de conserver les matrices de toutes les étapes. Seule la dernière étape est utilisée dans le calcul suivant. On peut donc ainsi faire le calcul en place et atteindre une complexité spatiale en $O(n^2)$.

```python
def in_place_floyd_marshall(M):
    for p in range(M.shape[0]):
        #print(p, M)
        for i in range(M.shape[0]):
            for j in range(M.shape[1]):
                M[i, j] = min(M[i, j], M[i, p] + M[p, j])
```

Comme on travaille en place directement sur `M` est que `M` est un type muable, on n'a pas besoin de renvoyer `M` à la fin de la fonction.