Programmation dynamique

INFORMATIQUE COMMUNE - TP n° 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
- ☞ coder de bas en haut une problème en programmation dynamique
- ☞ utiliser la mémoïsation et un dictionnaire pour pallier l'inefficacité de l'approche 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 $V = \sum_{i=1}^{n} v_i$ tout en ne dépassant pas le poids maximal admissible par le sac à dos, c'est à dire en respectant la contrainte $\sum_{i=1}^{n} p_i \leqslant \pi$? On note [a] le problème du sac à dos $\mathrm{KP}(n, \pi)$ 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 dont les valeurs et les poids sont les suivants :

- `valeurs = [100, 700, 500, 400, 300, 200]`,
- `poids = [40, 15, 2, 9, 18, 2]`.

Un objet `i` possède une valeur `valeurs[i]` et un poids `poids[i]`.

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

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.

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

A2. Pour le problème KP(n,π), implémenter un algorithme de résolution glouton.

Solution :

```
def greedy_kp(objects, W):
  total_weight = 0
  total_value = 0
  pack = []
  objects = sorted(objects)  # to choose max val, increasing order
  #print(objects)
  while len(objects) > 0 and total_weight <= W:
    o = objects.pop()  # choose an object, the last and the max val !
    # print(o)
    if total_weight + o[1] <= W:  # is it a solution ?
      pack.append(o)  # memorize
      total_weight += o[1]
      total_value += o[0]  # and keep on
  return pack, total_value, total_weight

values = [100, 700, 500, 400, 300, 200]
weights = [40, 15, 2, 9, 18, 2]
max_weight = 15
s = greedy_kp(zip(values, weights), max_weight)
```

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.

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

A4. En utilisant la programmation dynamique de bas en haut, coder la résolution du problème KP(n,π).

Solution :

```
import numpy as np

def kp_dp(v, w, W):
    n = len(w)
    s = np.zeros((n+1,W+1))
    for i in range(n + 1):
        for p in range(W + 1):
```

```
            if i == 0:
                s[i,p] = 0  # no objects, no solution, value is zero
            elif w == 0:  # 0 kg, one solution : take 0 object of 0 value
                s[i,p] = 0
            elif w[i - 1] <= p:
                s[i,p] = max(v[i - 1] + s[i - 1, p - w[i - 1]], s[i - 1,p])
            else:
                s[i, p] = s[i - 1, p]
    return s  # [n][W]


values = [100, 700, 500, 400, 300, 200]
weights = [40, 15, 2, 9, 18, 2]
max_weight = 15
s = kp_dp(values, weights, max_weight)
```

A5. En comparant les deux stratégies précédentes, identifier les cas pour lesquels l'algorithme glouton n'est pas optimal.

Solution :

```
values = [100, 700, 500, 400, 300, 200]
weights = [40, 15, 2, 9, 18, 2]
n = len(weights)
for max_weight in range(2, 87):
  s = kp_dp(values, weights, max_weight, n)
  sg = greedy_kp(zip(values, weights), max_weight)
  try:
    assert s[n][max_weight] == sg[1], f"{s[n][max_weight]} vs {sg[1]}"
  except Exception as e:
    print(f"Greedy not optimal ! for {n} objects, {max_weight} kg -->
        optimal : {s[n][max_weight]} vs {sg[1]}")

# Greedy not optimal ! for 6 objects, 15 kg --> optimal : 1100 vs 700
# Greedy not optimal ! for 6 objects, 16 kg --> optimal : 1100 vs 700
```

A6. Programmer une version gloutonne de KP(n,π) qui choisit le meilleur objet d'après le ratio valeur / poids.

Solution :

```
def ratios_greedy_kp(objects, W):
  total_weight = 0
  total_value = 0
  pack = []
  ratios = sorted([(v / w, v, w) for v, w in objects])  # to choose max val
  # print(f"Ratios {ratios}")
  while len(ratios) > 0 and total_weight <= W:
    o = ratios.pop()  # choose the last object and the greatest value !
    if total_weight + o[2] <= W:  # is it a solution ?
      pack.append((o[1], o[2]))  # memorize
```

```
            total_weight += o[2]
            total_value += o[1]
        return pack, total_value, total_weight
```

A7. En comparant les trois stratégies, identifier les cas pour lesquels les algorithmes gloutons ne sont pas optimaux.

Solution :

```
values = [100, 700, 500, 400, 300, 200]
weights = [40, 15, 2, 9, 18, 2]
n = len(weights)
for max_weight in range(2, 87):
    t = kp_dp_list(values, weights, max_weight)
    s = kp_dp(values, weights, max_weight)
    sg = greedy_kp(zip(values, weights), max_weight)
    srg = ratios_greedy_kp(zip(values, weights), max_weight)
    try:
        assert s[n][max_weight] == sg[1], f"Optimal : {s[n][max_weight]} vs
            {sg[1]} (greedy -> not optimal)"
        assert s[n][max_weight] == srg[1], f"Optimal : {s[n][max_weight]} vs
             {srg[1]} (ratio greedy -> not optimal)"
    except Exception as e:
        print(e)

# Optimal : 900 vs 700 (ratio greedy -> not optimal)
# Optimal : 900 vs 700 (ratio greedy -> not optimal)
# Optimal : 1100 vs 700 (greedy -> not optimal)
# Optimal : 1100 vs 700 (greedy -> not optimal)
# Optimal : 1200 vs 1100 (ratio greedy -> not optimal)
# Optimal : 1200 vs 1100 (ratio greedy -> not optimal)
# Optimal : 1600 vs 1400 (ratio greedy -> not optimal)
# Optimal : 1600 vs 1400 (ratio greedy -> not optimal)
# Optimal : 1900 vs 1800 (ratio greedy -> not optimal)
# Optimal : 1900 vs 1800 (ratio greedy -> not optimal)
```

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 $\text{CCP}(M, i, P)$ 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

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.

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.

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 $M_c = [5,1,2]$ et $M = [4,1,3]$. 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.

```
def greedy_ccp(M, price):
    solution = {}
    coins_nb = 0
    M = sorted(M, reverse=True)
    for m in M:  # greatest value first
        nb = price // m  # how many times ?
        if nb > 0:  # if 0, no solution with m
            solution[m] = nb
            price = price - nb * m  # continue...
            coins_nb += nb
    if price == 0:  # success
        return [coins_nb, solution]
    else:
        return None  # no solution
```

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.

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

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

Solution :

```
def dp_ite_ccp(M, price):
    S = [[[0, {}] for i in range(price + 1)] for i in range(len(M) + 1)]
    # print(S)
    for p in range(0, price + 1):  # no coins, no solution
        S[0][p][0] = math.inf  # !!!! useful for min function

    for p in range(1, price + 1):
        for i in range(1, len(M) + 1):
            mi = M[i - 1]
            if mi <= p:
                a = 1 + S[i][p - mi][0]  # with
                b = S[i - 1][p][0]   # without
                S[i][p][0] = min(a, b)
                if a <= b:  # Update dictionary
                    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  # init dictionnary key
                else:
                    S[i][p][1] = S[i][p][1] | S[i - 1][p][1]
            else:
                S[i][p] = S[i - 1][p]
    return S[len(M)][price]
```

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.

Solution :

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

for P in range(49):
    s = dp_ite_ccp(M, P)
    sg = greedy_ccp(M, P)
    try:
        assert sg[0] == s[0], f"M={M} - Price --> {P} - Optimal : {s} vs  {
            sg} - Greedy NOT optimal"
    except Exception as e:
        print(e)
```

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. $a[0]$ désigne le premier caractère de la chaîne a. On dénote par $a[:-1]$ la chaîne a tronquée de son premier 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 en coûte rien (0).

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

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

Si on souhaite programmer dynamiquement par le bas en utilisant un tableau S contenant les distances, il est utile d'exprimer le résultat d'une case en fonction de celles dont elle dépend dans le schéma dynamique :

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

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 et de bas en haut le tableau 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 de bas en haut. 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[0] == b[0]:
      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 $s = a_i...a_k$ où $i \leqslant ... \leqslant k < n$ et $\forall p \in \{i,...,k\}, a_p \in a$. 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.

■ **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):
  j = 0
  for i in range(len(ch)):
    if ch[i] == sch[j]:
      j += 1
    if j == len(sch):
      return True
```

```
    return False
```

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 $a_1 a_2 \dots a_i$ et $b_1 b_2 \dots b_j$ 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 du haut vers le bas.

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

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

Solution :

```
def mem_lss(a, b, m):
  if (a, b) in m:
```

```
      return m[(a, b)]
    else:
      if len(a) == 0 or len(b) == 0:
        return 0
      else:
        if a[0] == b[0]:
          r = 1 + mem_lss(a[1:], b[1:], m)
          m[(a, b)] = r
          return r
        else:
          r1 = mem_lss(a[1:], b, m)
          r2 = mem_lss(a, b[1:], m)
          m[(a, b)] = max(r1, r2)
          return m[(a, b)]
```

D7. Faire en sorte que les codes qui résolvent $\mathcal{L}(a,b)$ produisent également les sous-chaînes solutions.

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.

Soit un graphe orienté et valué $G = (S, A, C)$ décrit par ses $n = |\,S\,|$ sommets $S = \{s_1, s_2, \ldots, s_n\}$, ses arcs A et les coûts associés aux arcs C.

G peut être modélisé par une matrice d'adjacence M :

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

Un exemple de graphe associé à sa matrice d'adjacence est donné sur la figure 1.

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 s_p. 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 $\{s_1, s_2, \ldots, s_p\}$. On construit ainsi une suite de matrice finie $(M_p)_{p \in [\![0,n]\!]}$ avec $M_0 = M$.

Supposons qu'on dispose de M_p. Considérons un chemin $\mathcal{C}$ entre s_i et s_j dont la longueur est minimale et dont les sommets intermédiaires sont dans $\{s_1, s_2, \ldots, s_{p+1}\}$, $p \leqslant n$. Pour un tel chemin :

- soit $\mathcal{C}$ passe par s_{p+1}. Dans ce cas, $\mathcal{C}$ est la réunion de deux chemins dont les sommets sont dans $\{s_1, s_2, \ldots, s_{p+1}\}$: celui de s_i à s_{p+1} et celui de s_{p+1} à s_j.
- soit $\mathcal{C}$ ne passe pas par s_{p+1}.

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.

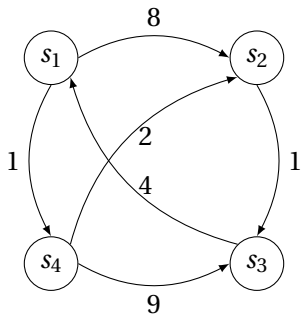

FIGURE 1 – Exemple de graphe orienté et valué. La matrice d'adjacence de ce graphe $G = (S, A, V)$ vaut : $M = \begin{pmatrix} 0 & 8 & +\infty & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & +\infty & 0 & +\infty \\ +\infty & 2 & 9 & 0 \end{pmatrix}$. Sur cet exemple, le chemin le plus court de s_4 à s_3 vaut 3 et passe par s_2.

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

Disposer d'une formule de récurrence entre M_{p+1} et M_p 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 :

$$\forall p \in [\![1,n]\!], \forall i,j \in [\![1,n]\!], M_{p+1}(i,j) = \min\left(M_p(i,j), M_p(i,p+1) + M_p(p+1,j)\right) \tag{7}$$

E2. Coder une fonction qui implémente l'algorithme de Floyd-Warshall et la programmation dynamique de bas en haut. Tester ce code sur l'exemple de la figure 1. On pourra utiliser un tableau numpy à trois dimensions.

Solution :

```
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. 4. 1.] [ 5. 0. 1. 6.] [ 4. 7. 0. 5.] [ 7. 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 $O(n^2)$.

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

(R) Pour ceux qui font l'option informatique, cet algorithme est celui qui permet de trouver la forme d'une expression régulière dénotant le langage rationnel défini par un automate fini (algorithme de McNaughton et Yamada).