

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
- ☞ résoudre le problème en construisant un tableau de résolution (approche impérative)
- ☞ utiliser la mémorisation (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 limitant² le poids à π .

Soit un ensemble $\mathcal{O}_n = \{o_1, o_2, \dots, o_n\}$ de n objets de valeurs v_1, v_2, \dots, v_n et de poids respectifs p_1, p_2, \dots, 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 $\sum_{o_i \in B} p_i \leq \pi$ où B est l'ensemble des objets emportés dans le sac?

On note^a le problème du sac à dos $KP(n, \pi)$ 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 $o = (\text{valeur}, \text{poids})$:

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?

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

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

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 .

- A4. En utilisant construisant un tableau de résolution, coder la résolution du problème $KP(n, \pi)$ de manière impérative. La signature de la fonction est `dyn_kp(objets, pmax)`. Quelle est la complexité de votre fonction?
- A5. En utilisant la programmation dynamique récursive descendante, coder la résolution du problème $KP(n, \pi)$. La fonction a pour signature `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.

B Rendu de monnaie

Un commerçant doit à rendre la monnaie à un client³. La somme à rendre est une 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 $M = \{m_1, m_2, \dots, m_n\}$ qui possède n valeurs différentes.

On nomme ce problème le rendu de monnaie⁴ 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 monétaires $M_c = (5, 1, 2)$ et $M = (4, 1, 3)$. Rendre la monnaie de manière optimale sur 10 et 6 avec chaque système.
- 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?
- B3. Le problème du rendu de monnaie est-il à sous-structure optimale? Pourquoi?
- 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)
- 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.

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, \dots, 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.

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 définie par induction de la manière suivante :

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

On souhaite calculer la distance d'édition en programmant dynamiquement par le bas 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 . contenant les distances, il est utile d'exprimer le résultat d'une case en fonction de celles dont elle dépend dans le schéma dynamique :

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

- C1. La distance d'édition de "chien" à "niche" vaut 4. Expliquer pourquoi.
- C2. La distance d'édition représente-t-elle un problème à sous-structure optimale? Pourquoi?
- 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".
- 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 impérative). On pourra tester sur "AGTTC" et "AGCTC", sur "chien" et "niche" ou "sunday" et "saturday".
- C5. Écrire un code similaire en programmation dynamique avec memoïsation et comparer les résultats.

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-chaîne d'une chaîne de caractères $a = a_1 \dots a_n$ toute chaîne de caractères s extraite de a telle que : $s = a_{i_1} \dots a_{i_k}$ où (i_1, i_2, \dots, i_k) est un sous-ensemble ordonné de $\llbracket 1, n \rrbracket$ 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-chaine commune.** Soit $a = a_1 \dots a_q$ et $B = b_1 \dots b_p$ deux chaînes de caractères non vides. On appelle plus longue sous-chaine commune à a et b toute sous-chaine 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-chaine commune, la chaîne vide est alors l'unique plus longue sous-chaine commune à a et b . Elle a pour longueur 0.

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

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

- D1. On considère les chaînes $a = \text{"AATGCG"}$ et $b = \text{"TATTAGC"}$? Donner les solutions de $\mathcal{L}(a, b)$.
- 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-chaine de `ch` et `False` sinon.
- 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-chaine commune à a et b .
- D4. Formuler le problème $\mathcal{L}(a, b)$ récursivement afin de pouvoir justifier de sa sous-structure optimale.
- D5. Écrire un code qui résout $\mathcal{L}(a, b)$ avec la programmation dynamique, de manière impérative, en complétant un tableau de résolution.
- D6. Résoudre $\mathcal{L}(a, b)$ récursivement sans mémorisation.
- D7. Résoudre $\mathcal{L}(a, b)$ récursivement avec mémorisation.
- D8. Créer une fonction de type décorateur Python qui automatise la mémorisation d'une fonction récursive. --> HORS PROGRAMME

E Algorithme de Floyd-Warshall

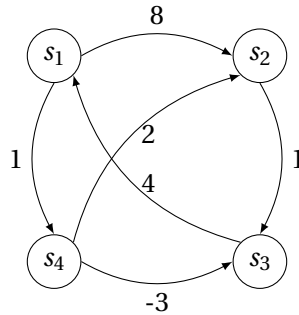
L'algorithme de Floyd-Warshall est l'application de la programmation dynamique à la recherche du plus court chemin entre deux sommets d'un graphe orienté et valué. Le plus court chemin n'est pas celui qui comporte le moins de sommets mais celui dont la somme des poids de chaque arc est la plus faible⁵. 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

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

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} .

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

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

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 $(M_p)_{p \in \llbracket 0, n \rrbracket}$ avec $M_0 = M$.

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 $\{0, 1, 2, \dots, p-1\}$, $p \leq n$. 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 $\{1, 2, \dots, p-1\}$: 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.
- E2. Coder une fonction qui implémente l'algorithme de Floyd-Warshall de manière impérative 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.
- E3. Quelle est la complexité temporelle de cet algorithme?
- E4. Quelle est la complexité spatiale de cet algorithme? Pourrait-on l'améliorer? Comment?