Synthèse du cours 5 : programmation dynamique (2)

11 octobre 2022

François Laroussinie

NB : Ces synthèses ont pour but de compléter les notes prises en cours. Elles ne les remplacent pas ! En particulier, la plupart des preuves n'y figurent pas. Rappel : il faut programmer les algorithmes vus en cours.

1 Rendre la monnaie

Ici l'objectif est de minimiser le nombre de pièces pur constituer une certaine somme S à partir d'un ensemble de n *types* de pièces de valeur $p_1,\ldots,p_n$. Notons que l'on fait une hypothèse importante : on suppose que l'on dispose d'autant de pièces que nécessaire pour chaque catégorie $p_1,\ldots,p_n$.

On définit alors $T[i,s]$ le nombre minimal de pièces de valeurs $p_1,\ldots,p_i$ pour constituer la somme s. On a alors $T[i,0]=0$ pour tout i, et par convention on prend $T[0,s]=\infty$ pour $s>0$. On a alors :

$$T[i,s] \;=\; \begin{cases} \min\Big(T[i-1,s]\;,\; 1+T[i,s-p_i]\Big) & \text{si } s \geq p_i \\ T[i-1,s] & \text{sinon} \end{cases}$$

Exemple. On reprend le cas $S=8$ et $p_1=1$, $p_2=4$ et $p_3=6$.

S	0	1	2	3	4	5	6	7	8
$p_1=1$	0	1	2	3	4	5	6	7	8
$p_2=4$	0	1	2	3	1	2	3	4	2
$p_3=6$	0	1	2	3	1	2	1	2	2

D'où l'algorithme suivant où on n'utilise pas les valeurs $T[s,0]$:

$T[0\ldots n, 0\ldots S]$: tableau d'entiers
$T[i,0]=0 \quad \forall i$
Pour $i=1,\ldots,n$:
... Pour $s=1,\ldots,S$:

$$\text{... ... } T[i,s] \;=\; \begin{cases} \infty & \text{si } i=1 \wedge s<p_i \\ 1+T[1,s-p_1] & \text{si } i=1 \wedge s\geq p_i \\ T[i-1,s] & \text{si } i>1 \wedge s<p_i \\ \min(T[i-1,s]\;,\; 1+T[i,s-p_i]) & \text{sinon} \end{cases}$$

Pour reconstituer un ensemble solution, on applique le schéma suivant :

— Si $T[i,s]=T[i-1,s]$, alors la pièce p_i n'est pas utile et la solution est celle pour $T[i-1,s]$,
— Si $T[i,s]=1+T[i,s-p_i]$, alors il faut une pièce de p_i ajoutée à une solution pour $T[i,s-p_i]$,
— (Et si $T[i-1,s]=1+T[i,s-p_i]$, alors les deux choix sont possibles !)

La complexité de la construction de $T[,]$ est $n \cdot (S+1)$, donc en $O(n \cdot S)$. Retourner un ensemble solution se fait en $O(n + T[n, S])$: le n vient du nombre maximal de « saut » de pièces (1 pour chaque catégorie) et $T[n, S]$ vient du nombre maximal d'opérations pour chaque pièce.

On peut là encore améliorer l'algorithme en ne prenant qu'un tableau à une dimension :

$T[0 \dots S]$: tableau d'entiers
$T[0] = 0 \quad \forall i$
Pour $i = 1, \dots, n$:
... Pour $s = 1, \dots, S$:

$$\text{... ... } T[s] \;=\; \begin{cases} \infty & \text{si } i = 1 \wedge s < p_i \\ 1 + T[s - p_1] & \text{si } i = 1 \wedge s \geq p_i \\ T[s] & \text{si } i > 1 \wedge s < p_i \\ \min(T[s]\ ,\ 1 + T[s - p_i]) & \text{sinon} \end{cases}$$

2 Plus courts chemins dans les graphes valués

Les algorithmes de Bellman-Ford et de Floyd-Warshall sont des algorithmes de programmation dynamique. Dans la suite on note $\delta_G(x, y)$ la longueur d'un plus court chemin (PCC) du sommet x au sommet y dans un graphe orienté valué $G = (S, A, w)$. A noter que la fonction w peut associer des poids négatifs (NB : en cas de circuits strictement négatifs, l'existence de PCC n'est plus garanti même si des chemins existent entre deux sommets).

2.1 Algorithme de Bellman-Ford

On part d'un sommet initial s et on souhaite calculer $\delta_G(s, x)$ pour tout $x \in S$. L'idée de l'algorithme de Bellman-Ford est de procéder en $|S| - 1$ itérations afin de calculer des coefficients d_x^i correspondant à la longueur (le poids) d'un PCC d'au plus i arcs entre s et x. Si un PCC existe, alors il a forcément au plus $|S|-1$ arcs, et après la $(|S|-1)$-ème itération on a le résultat voulu. L'algorithme se termine par un test de la présence de circuits strictement négatifs.

On peut calculer un tableau $\mathsf{D}^i[q]$ correspondant à $\delta^{\leq i}(s, q)$, c'est-à-dire le poids d'un PCC d'au plus i arcs. On ainsi :

$$\mathsf{D}^0[q] = \begin{cases} 0 & \text{si } q = s \\ \infty & \text{sinon} \end{cases} \qquad \mathsf{D}^{i+1}[q] = \min(\mathsf{D}^i[q], \min_{(q',q) \in A} \{\mathsf{D}^i[q'] + w(q', q)\})$$

On a bien $\mathsf{D}^i[q] = \delta^{\leq i}(s, q)$ et $\mathsf{D}^{n-1}[q] = \delta(s, q)$.

Ensuite, le calcul peut se faire sur un seul tableau D, on a alors l'invariant $\mathsf{D}[q] \leq \delta^{\leq i}(s, q)$ après la i-ème itération, ce qui nous suffit pour obtenir *in fine* $\mathsf{D}[q] = \delta(s, q)$.

2.2 Algorithme de Algorithme de Floyd-Warshall

Maintenant nous considérons la recherche des plus courts chemins entre **tous** les sommets d'un graphe : nous voulons une procédure qui calcule la distance $\delta(x, y)$ – *i.e.* la longueur d'un PCC entre x et y – pour toute paire de sommet $\langle x, y \rangle$.

Procédure `PCC-Bellman-Ford` (G, s)
`//`$G = (S, A, w)$ `: un graphe orienté, valué avec` $w : A \to \mathbb{R}$.
`//`$s \in S$ `: un sommet origine.`
begin
  pour chaque $u \in S$ **faire**
    $\Pi[u] := \mathsf{nil}$
    $\mathsf{d}[u] := \begin{cases} 0 & \text{si } u = s \\ \infty & \text{sinon} \end{cases}$
  pour $i = 1$ *à* $|S| - 1$ **faire**
    pour chaque $(u, v) \in A$ **faire**
      si $d[v] > d[u] + w(u, v)$ **alors**
        $\mathsf{d}[v] := \mathsf{d}[u] + w(u, v)$
        $\Pi[v] := u$
  pour chaque $(u, v) \in A$ **faire**
    si $d[v] > d[u] + w(u, v)$ **alors**
      return $(\bot, -, -)$
  return $(\top, d, \Pi)$

Algorithme 1 : algorithme de Bellman-Ford

On va utiliser une représentation matricielle d'un graphe valué $G = (S, A, w)$ avec $S = \{x_1, \dots x_n\}$ et $w : A \to \mathbb{R}$. On note $M = (\alpha_{ij})_{1 \leq i,j \leq n}$ la matrice représentant G où α_{ij} décrit l'arc entre x_i et x_j : α_{ij} vaut (1) 0 si $i = j$, (2) $w(x_i, x_j)$ si $i \neq j$ et si $(x_i, x_j) \in A$ et (3) ∞ sinon.

On note δ_{ij} la distance $\delta(x_i, x_j)$. L'idée clé est de calculer des coefficients d_{ij}^k correspondant à la distance d'un PCC entre x_i et x_j d'*intérieur* inclus dans $\{x_1, \dots, x_k\}$ (l'intérieur d'un chemin est l'ensemble des sommets intermédiaires). La solution recherchée est donc les coefficients d_{ij}^n pour tout i, j (on calcule aussi une matrice des prédécesseurs Π pour conserver la description des chemins : le coefficient π_{ij} vaut s si et seulement si le prédécesseur immédiat de x_j le long d'un PCC de x_i à x_j est s).

Le point clé sur lequel est basé l'algorithme de Floyd-Warshall est la relation suivante liant les durées minimales pour aller de i à j par des chemins d'intérieur $\{1, \dots, k\}$ et celles basées sur les chemins d'intérieur $\{1, \dots, k-1\}$:

$$\alpha_{ij}^k \stackrel{\text{def}}{=} \min(\alpha_{ij}^{k-1}\ ,\ \alpha_{ik}^{k-1} + \alpha_{kj}^{k-1})$$

Cette relation est correcte car un PCC entre i et j d'intérieur $\ll \leq k \gg$ est :

— soit d'intérieur $\ll \leq k - 1 \gg$ (et ne passe pas par k),

— soit il passe par k et alors la portion entre i et k n'a pas besoin de passer par k (hypothèse d'absence de cycle strictement négatif !) et c'est donc un PCC d'intérieur $\ll \leq k - 1 \gg$, et on a de même pour la portion entre k et j.

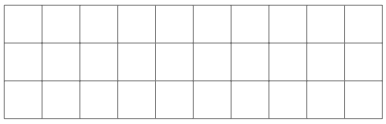
On peut utiliser une famille de matrices $D^{(k)}$ $(k = 1, \dots n)$ contenant les calculs intermédiaires, et la dernière $D^{(n)}$ contient les coefficients δ_{ij}. Mais comme pour les autres problèmes de programmation dynamique étudiés ici, on peut améliorer la complexité en espace mémoire, et n'utiliser qu'une seule matrice. C'est ce qui est fait dans l'algorithme 2. La complexité (en temps) de cet algorithme est clairement en $O(n^3)$. Il utilise un espace mémoire en $O(n^2)$.

Procédure `PCC-Floyd` (G)
`//`$G = (S, A, w)$ `: un graphe orienté, valué avec` $w : A \to \mathbb{R}$.
`//avec` $S = \{x_1, \ldots, x_n\}$
`//avec` $M = (\alpha_{ij})_{1 \leq i,j \leq n}$ `la matrice corresp. à` A
begin
  `//On initialise` D `avec` M`:`
  pour $i = 1 \ldots n$ **faire**
    pour $j = 1 \ldots n$ **faire**
      $d_{ij} := \alpha_{ij}$
      si $\alpha_{ij} \neq \infty$ **alors** $\pi_{ij} := i$
  pour $k = 1 \ldots n$ **faire**
    pour $i = 1 \ldots n$ **faire**
      pour $j = 1 \ldots n$ **faire**
        si $d_{ij} > d_{ik} + d_{kj}$ **alors**
          $d_{ij} := d_{ik} + d_{kj}$
          $\pi_{ij} := \pi_{kj}$
  return D, Π

Algorithme 2 : algorithme de Floyd-Warshall (avec économie de mémoire)

3 Un problème de pavage

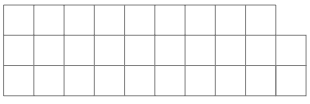
L'objectif ici est de calculer le nombre de pavages différents d'une zone $3 \times n$ par des dominos 2×1. La zone à paver correspond à la zone Z_n de la figure 2.



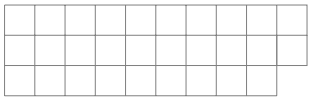
Z_n avec $n = 10$

FIGURE 1 – Des zones à paver.

Lorsqu'on essaie de paver la zone Z_n, on peut générer d'autres formes de zones à paver, en particulier, on définit :



Z'_n avec $n = 10$



Z''_n avec $n = 10$

FIGURE 2 – Des zones à paver.

Pour paver Z_n, on peut donc se ramener à la zone $Z_n - 2$ (si on pose trois dominos empilés), à la zone Z'_n (si on place un domino vertical et un domino horizontal en dessus, NB : le second domino est imposé) ou à Z''_n (si on place un domino vertical et un domino horizontal en dessous). Ces trois possibilités sont illustrées sur la figure 4.

On note A_n le nombre de pavages distincts de la zone Z_n, B_n le nombre de pavages pour

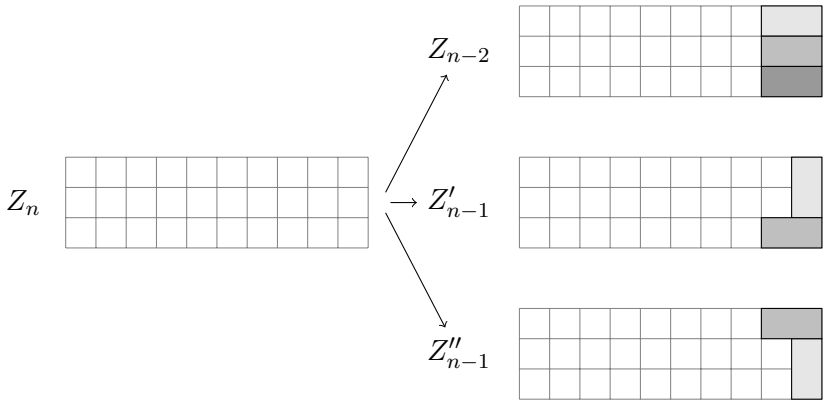


FIGURE 3 – Evolution de la zone à paver Z_n.

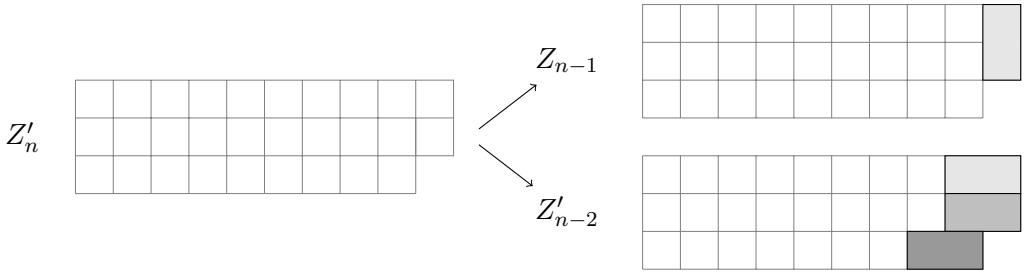


FIGURE 4 – Evolution de la zone à paver Z'_n.

Z'_n et C_n pour Z''_n. On a bien sûr $B_n = C_n$. On peut ensuite définir les récurrences :

$$A_1 = 0 \quad A_2 = 3 \quad B_1 = 1 \quad B_2 = 0$$

et pour $n > 2$: $A_n = A_{n-2} + 2 \cdot B_{n-1}$ $\qquad$ $B_n = A_{n-1} + B_{n-2}$

NB : on peut aussi prendre $A_0 = 1$ et $B_0 = 0$ par convention, les deux suites restent inchangées. On en déduit le calcul de A_n par l'algorithme suivant.

Procédure `NbPavages` (n)
begin
  `A` $= 1$, `B` $= 1$
  si n *est impair* **alors return** 0
  pour $i = 2 \ldots n$ **faire**
    si i *est pair* **alors**
      `A` $:=$ `A` $+ 2 \cdot$ `B`
    else
      `B` $:=$ `A` $+$ `B`
  return `A`