

Synthèse du cours 5 : programmation dynamique (2)

17 octobre 2023

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 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 garantie même si des chemins existent entre deux sommets).

1.1 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 $(|S| - 1)$ -ième 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 $D^i[q]$ correspondant à $\delta^{\leq i}(s, q)$, c'est-à-dire la longueur d'un PCC d'au plus i arcs. On a ainsi :

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

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

Le calcul effectué par l'algorithme de Bellman-Ford est un peu différent. Il utilise un tableau unidimensionnel et la mise à jour de ces coefficients permet juste de s'assurer que l'on a $D[q] \leq \delta^{\leq i}(s, q)$ après la i -ième itération, ce qui nous suffit pour obtenir *in fine* $D[q] = \delta(s, q)$.

1.2 Algorithme de Floyd-Warshall

Maintenant 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 $\langle x, y \rangle$.

On va utiliser une représentation matricielle d'un graphe valué $G = (S, A, w)$ avec $S = \{x_1, \dots, x_n\}$ et $w : A \rightarrow \mathbb{R}$. 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}^k la distance $\delta(x_i, x_j)$. L'idée clé est de calculer des coefficients δ_{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

Procédure PCC-Bellman-Ford (G, s)

// $G = (S, A, w)$: un graphe orienté, valué avec $w : A \rightarrow \mathbb{R}$.

// $s \in S$: un sommet origine.

begin

pour chaque $u \in S$ **faire**

$\Pi[u] := \text{nil}$

$d[u] := \begin{cases} 0 & \text{si } u = s \\ \infty & \text{sinon} \end{cases}$

pour $i = 1$ à $|S| - 1$ **faire**

pour chaque $(u, v) \in A$ **faire**

si $d[v] > d[u] + w(u, v)$ **alors**

$d[v] := d[u] + w(u, v)$

$\Pi[v] := u$

pour chaque $(u, v) \in A$ **faire**

si $d[v] > d[u] + w(u, v)$ **alors**

return $(\perp, -, -)$

return (\top, d, Π)

Algorithme 1 : algorithme de Bellman-Ford

d'un chemin est l'ensemble des sommets intermédiaires). La solution recherchée est donc les coefficients d_{ij}^n pour tout i, j .

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 inclu dans $\{1, \dots, k\}$ est :

- soit d'intérieur inclu dans $\{1, \dots, k - 1\}$ (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 (sinon il y aurait un cycle strictement négatif!) et c'est donc un PCC d'intérieur inclu dans $\{1, \dots, k - 1\}$, et on a de même pour la portion entre k et j .

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 en $O(n^3)$ (et utilise un espace mémoire en $O(n^2)$).

2 Le problème du sac à dos

Input : Un poids maximal $W \in \mathbb{N}$, un ensemble de n objets ayant chacun un poids w_i et une valeur v_i pour $i = 1, \dots, n$. On suppose $w_i > 0$ et $v_i > 0$.

Output : La valeur maximale pouvant être stockée dans le sac sans dépasser sa capacité W .

On peut distinguer deux variantes selon que l'on autorise à mettre plusieurs fois un même élément dans le sac (on a, dans ce cas, plutôt une liste de *types* d'objets) ou pas. La variante la plus classique est celle sans répétition.

Procédure PCC-Floyd (G)//avec $M = (\alpha_{ij})_{1 \leq i, j \leq n}$ la matrice corresp. à G **begin**

```

  pour  $i = 1 \dots n$  faire
  | pour  $j = 1 \dots n$  faire  $d_{ij} := \alpha_{ij}$ 
  pour  $k = 1 \dots n$  faire
  | pour  $i = 1 \dots n$  faire
  | | pour  $j = 1 \dots n$  faire
  | | | si  $d_{ij} > d_{ik} + d_{kj}$  alors
  | | |    $d_{ij} := d_{ik} + d_{kj}$ 
  return  $D, \Pi$ 

```

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

Par exemple avec $W = 10$ et l'ensemble $\{(5, 20), (5, 2), (6, 3)\}$, la valeur maximale pour un sac de capacité 10 est 22 pour la version sans répétition (avec les objets 1 et 2) et 40 si on autorise des répétitions (avec deux objets 1).

2.1 Sans répétition

Chacun des n objets peut être mis au plus une fois dans le sac.

On va calculer le tableau $K[i, w]$, avec $0 \leq i \leq n$ et $0 \leq w \leq W$, défini comme étant la **valeur** maximale pouvant être stockée dans un sac de **capacité** w avec les objets $1, 2, 3, \dots, i$. Le résultat du problème est alors la valeur $K[n, W]$.

On a clairement $K[0, w] = 0$ et $K[j, 0] = 0$. Et le cas général s'exprime ainsi :

$$K[j, w] \stackrel{\text{def}}{=} \max \left(K[j-1, w], v_j + K[j-1, w - w_j] \right)$$

D'où l'algo :

Procédure SaDssRepet (O, W)**begin**

```

  //O correspond à la liste des  $n$  objets de poids  $w_i$  et de valeur  $v_i$ 
  K : tableau d'entiers de taille  $(n+1) \times (W+1)$ 
  pour  $w = 1 \dots W$  faire  $K[0, w] := 0$ 
  pour  $i = 0 \dots n$  faire  $K[i, 0] := 0$ 
  pour  $i = 1 \dots n$  faire
  | pour  $w = 1 \dots W$  faire
  | | si  $w_i \leq w$  alors  $K[i, w] := \max(K[i-1, w], v_i + K[i-1, w - w_i])$ 
  | | else  $K[i, w] := K[i-1, w]$ 
  return  $K[n, W]$ 

```

Exemple : avec $W = 10$ et l'ensemble d'objets $\{(2, 5), (8, 22), (3, 8), (4, 5)\}$ (NB : ce sont des paires (w, v)). On obtient le tableau K ci-dessous :

	0	1	2	3	4	5	6	7	8	9	10	11	12
–	0	0	0	0	0	0	0	0	0	0	0	0	0
(2, 5)	0	0	5	5	5	5	5	5	5	5	5	5	5
(8, 22)	0	0	5	5	5	5	5	5	22	22	27	27	27
(3, 8)	0	0	5	8	8	13	13	13	22	22	27	30	30
(4, 5)	0	0	5	8	8	13	13	13	22	22	27	30	30

On peut améliorer l'algorithme précédent en utilisant une seule ligne du tableau (en veillant à énumérer correctement les indices!) :

Procédure SaDssRepet2 (O, W)

```

begin
  //O correspond à la liste des n objets de poids w_i et de valeur v_i
  K : tableau d'entiers de taille W + 1
  pour w = 0...W faire K[w] := 0
  pour i = 1...n faire
    pour w = W...w_i faire
      K[w] := max(K[w], v_i + K[w - w_i])
  return K[n, W]

```

Et l'on peut adapter l'algorithme pour calculer aussi un ensemble d'objets optimal (c-à-d. dont la valeur totale est $K[n, W]$ et dont le poids est inférieur ou égal à W).

Procédure SaDssRepet3 (O, W)

```

begin
  //O correspond à la liste des n objets de poids w_i et de valeur v_i
  K : tableau d'entiers de taille W + 1
  Sol : tableau d'ensembles (listes) de taille W + 1
  pour w = 0...W faire
    K[w] := 0
    Sol[w] := ∅
  pour i = 1...n faire
    pour w = W...w_i faire
      si K[i - 1, w] < v_i + K[i - 1, w - w_i] alors
        K[i, w] := v_i + K[i - 1, w - w_i]
        Sol[i, w] = Sol[i - 1, w - w_i] ∪ {(w_i, v_i)}
      else
        K[i, w] := K[i - 1, w]
        Sol[i, w] = Sol[i - 1, w]
  return K[n, W]

```

2.2 Avec répétition

Là encore, on peut calculer des coefficients $[i, w]$. On a toujours $K[0, w] = 0$ et $K[j, 0] = 0$ et le cas général s'énonce ainsi :

$$K[j, w] \stackrel{\text{def}}{=} \max \left(K[j-1, w], v_j + K[j, w-w_j] \right)$$

NB : c'est de prendre $K[j, w-w_j]$ (et non $K[j-1, w-w_j]$) qui permet de mettre plusieurs occurrences du même objet j !

D'où l'algo :

Procédure SaDavecRepet (O, W)

begin

```

//O correspond à la liste des n objets de poids w_i et de valeur v_i
K : tableau d'entiers de taille (n+1) × (W+1)
pour w = 1...W faire K[0, w] := 0
pour i = 0...n faire K[i, 0] := 0
pour i = 1...n faire
    pour w = 1...W faire
        si w_i ≤ w alors K[i, w] := max(K[i-1, w], v_i + K[i, w-w_i])
        else K[i, w] := K[i-1, w]
return K[n, W]

```

Exemple : avec $W = 10$ et l'ensemble d'objets $\{(2, 5), (8, 22), (3, 8), (4, 5)\}$. On obtient le tableau K ci-dessous :

	0	1	2	3	4	5	6	7	8	9	10	11	12
–	0	0	0	0	0	0	0	0	0	0	0	0	0
(2, 5)	0	0	5	5	10	10	15	15	20	20	25	25	30
(8, 22)	0	0	5	5	10	10	15	15	22	22	27	27	32
(3, 8)	0	0	5	8	10	13	16	18	22	24	27	30	32
(4, 5)	0	0	5	8	10	13	16	18	22	24	27	30	32

Et son amélioration (en veillant à énumérer correctement les indices!) :

Procédure SaDavecRepet2 (O, W)

begin

```

//O correspond à la liste des n objets de poids w_i et de valeur v_i
K : tableau d'entiers de taille W+1
pour w = 0...W faire K[w] := 0
pour i = 1...n faire
    pour w = w_i...W faire
        K[w] := max(K[w], v_i + K[w-w_i])
return K[n, W]

```