TD - Programmation dynamique

Optimisation et Recherche Operationnelle

M1 Info - semestre d'automne 2020-2021

Université Claude Bernard Lyon 1

Christophe Crespelle
`christophe.crespelle@inria.fr`

Eric Duchêne
`eric.duchene@univ-lyon1.fr`

Aline Parreau
`aline.parreau@univ-lyon1.fr`

Exercice 1. *Sur l'autoroute du profit*

Vous implentez une chaine de restaurants sur les aires de repos d'une section d'autoroute. Les aires sont numerotees de 1 a n dans l'ordre dans lequel on les rencontre sur l'autoroute. Pour chaque aire i, on connait :

— sa position x_i, exprimee en km depuis le debut de la section autoroutiere $0 < x_1 < x_2 < \cdots < x_n$, et

— le revenu espere $r_i > 0$ en placant un de vos restaurants dans cette aire, qui depend de la frequentation de l'aire.

Les x_i et les r_i sont donnes dans deux tableaux separes indexes par i. La societe qui gere l'autoroute, l'AFN (Autoroutes de France et de Navarre), impose une contrainte sur l'implentation des restaurants : deux restaurants de la meme chaine (y compris la votre) doivent etre espaces d'au moins 50km. On veut faire un algorithme qui retourne un placement de vos restaurants qui ait un revenu escompte maximum, note OPT, compte tenu de la contrainte imposee par l'AFN.

a. On note $p(i)$ le numero de l'aire la plus proche de i qui est situee avant i ($x_{p(i)} < x_i$) et a au moins 50 km de l'aire i. Donnez un algorithme de complexite $O(n)$ qui calcule $p(i)$ pour tout $i \in [\![1, n]\!]$. On prendra pour convention $p(i) = 0$ lorsqu'il n'existe pas d'aire a au moins 50km de i avant i.

Solution.

Algorithme 1 : Algorithme pour le calcul de $p(i)$, $1 \leq i \leq n$.

1 $p \leftarrow 0$;
2 **pour** i *de* 1 *a* n **faire**
3   **tant que** $x[i] - x[p+1] \geq 50$ **faire**
4         $p \leftarrow p + 1$;
5     $p[i] \leftarrow p$;

Au cours de l'algorithme la valeur pretendante a etre $p[i]$ est stockee dans p. Cette valeur est initialisee a 0 (ligne 1) qui est la valeur par convention lorsqu'aucun site ne se trouve a au moins 50km avant le site i courant. Ensuite, la boucle "pour" sur

i (ligne 2) considere chacun des sites un par un. Pour chacun d'eux, la boucle "tant que" de la ligne 3 cherche le plus grand p (en l'incrementant a la ligne 4) tel que le site numero p se trouve au moins 50km avant le site i (condition d'arret de la boucle "tant que", ligne 3). Lorsque cette valeur de p est atteinte elle est affectee a $p[i]$ a la ligne 5.

La complexite de l'algorithme est bien $O(n)$ car la boucle "pour" de la ligne 2 s'execute exactement n fois et le nombre total d'iterations de la boucle interne "tant que" (ligne 3) au cours de l'algorithme n'excede pas n, car p augmente de 1 a chaque iteration de la boucle. Pour etre rigoureux, remarquez aussi que le nombre de fois ou le test de la condition de la boucle "tant que" est negatif, qui ne compte pas dans les iterations de la boucle, est aussi exactement n : une fois pour chaque valeur de i. Toutes les autres instructions sont elementaires et prennent un temps constant. La complexite totale de l'algorithme est donc $O(n)$.

b. Si on decide de placer un restaurant sur l'aire i, sur quelles aires j avant i, c.a.d. $j < i$, peut-on eventuellement placer un autre restaurant ?

Solution. Precisemment sur les aires numero j avec $j \leq p(i)$ car ce sont les aires se trouvant avant i et a au moins 50km de i. C'est la raison pour laquelle $p(i)$ a ete defini ainsi.

On s'interesse maintenant au sous-probleme $Restau(j)$ dans lequel on ne place des restaurants que sur les aires $i \in [\![1, j]\!]$, pour un $j \in [\![1, n]\!]$ fixe, et on note $OPT(j)$ le revenu maximum qu'on peut atteindre dans ce sous probleme (on etend cette notation en posant par convention $OPT(0) = 0$).

c. Soit S_j^* une solution de revenu maximum au sous-probleme $Restau(j)$ telle que $j \notin S_j^*$. Exprimez le revenu $r(S_j^*)$ de cette solution en fonction des $OPT(j')$ pour $j' < j$.

Solution. Puisque j n'est pas dans la solution optimale S_j^* au probleme $Restau(j)$, alors cette solution n'utilise que des sites $j' \leq j-1$ (remarquez que lorsque $j \notin S_j^*$ alors necessairement $j > 1$). Ainsi, S_j^* est aussi une solution au probleme $Restau(j-1)$. Et comme S_j^* est la solution optimale de $Restau(j)$ alors c'est aussi la solution optimale de $Restau(j-1)$: $r(S_j^*) = OPT(j-1)$.

d. Meme question lorsque $j \in S_j^*$.

Solution. Lorsque $j \in S_j^*$, les autres sites j' impliques dans S_j^*, c'est a dire $j' \in S_j^*$ et $j' \neq j$, verifient necessairement $j' \leq p(j)$, car S_j^* satisfait les contraintes de distanciation imposees par l'AFN. Ainsi, comme S_j^* est la solution optimale a $Restau(j)$, les sites $j' \in S_j^*$ avec $j' \neq j$ forment une solution optimale a $Restau(p(j))$. On a donc $r(S_j^*) = r_j + OPT(p(j))$.

e. Donnez une formule de recurrence qui exprime $OPT(j)$ en fonction des $OPT(j'), j' < j$.

Solution. Comme toute solution optimale a $restau(j)$ contient ou ne contient pas j, d'apres les deux questions precedentes, on a $OPT(j) = \max\{OPT(j-1), r_j + OPT(p(j))\}$.

f. Donnez un algorithme de complexite $O(n)$ pour calculer OPT, le revenu escompte maximum, et un placement de vos restaurants correspondant.

Solution.

Pour calculer OPT, l'algorithme 2 suit une approche de programmation dynamique dans laquelle on calcule $OPT[j]$ pour tout $0 \leq j \leq n$, en posant comme convenu $OPT[0] = 0$ et en utilisant le tableau p calcule a la question a. A la fin de l'algorithme, on obtient alors la valeur de OPT comme $OPT = OPT[n]$.

Le calcul des $OPT[j]$, $1 \leq j \leq n$, se fait dans la boucle "pour" de la ligne 2 en utilisant la formule de recurrence de la question e (disjonction de cas des lignes 3 a 8). Afin d'obtenir non seulement la valeur de OPT mais egalement une solution qui realise cette valeur, on utilise un tableau *prem* qui pour chaque valeur de $j \geq 1$ donne le site de plus grand indice utilise dans la solution de valeur optimale $OPT[j]$ au probleme intermediaire $restau(j)$. Grace au tableau *prem*, a la fin de l'algorithme (ligne 10 a 14), on peut construire une solution optimale en remarquant que si le site x_i est utilise dans la solution optimale que l'on construit, alors le prochain site x_j, avec $j < i$, utilise dans cette solution est celui d'indice $j = prem[p[i]]$, car lorsque x_i participe a la solution optimale a $restau(i)$ (ligne 8), cette derniere est construite en prenant le site x_i et une solution optimale a $restau(p[i])$ (ligne 7). Dans l'algorithme, les listes sont notees entre parentheses et le . designe la concatenation de deux listes. Dans la liste S que l'on construit, les sites apparaissent dans l'ordre decroissant de leurs indices.

Algorithme 2 : Algorithme pour le calcul de OPT et d'une solution S realisant un revenu escompte de OPT.

```
1  OPT[0] ← 0;
2  pour j de 1 a n faire
3      si OPT[j − 1] ≥ r[j] + OPT[p[j]] alors
4          OPT[j] ← OPT[j − 1];
5          prem[j] ← prem[j − 1];
6      sinon
7          OPT[j] ← r[j] + OPT[p[j]];
8          prem[j] ← j;
9  OPT ← OPT[n];
10 k ← prem[n];
11 S ← (k);
12 tant que p[k] > 0 faire
13     k ← prem[p[k]];
14     S ← S.(k);
15 retourner (OPT, S);
```

Il est aise de verifier que la complexite de l'algorithme 2 est $O(n)$. Toute les instructions sont elementaires et prennent un temps constant, y compris la concatenation de deux listes ligne 14 avec une structure de donnee adequate (dans laquelle les listes sont representees avec un pointeur sur leur premier et sur leur dernier element). La boucle "pour" de la ligne 2 s'execute exactement n fois et la boucle "tant que" de la ligne 12 au plus n fois, car comme $p[k] < k$ pour tout k, $p[k]$ decroit strictement (ligne 13) a chaque iteration de la boucle.

Exercice 2. *Un sac de valeur*

Dans le probleme du sac a dos, on donne une collection d'objets numerotes de 1 a n et chaque objet a une valeur $v_i \in \mathbb{R}^+$ et un poids $w_i \in \mathbb{R}^+$, pour $i \in [\![1,n]\!]$. Le probleme est a valeurs entieres si de plus les valeurs v_i sont des entiers, c.a.d. $\forall i \in [\![1,n]\!], v_i \in \mathbb{N}$. Pour une collection d'objets $S \subseteq [\![1,n]\!]$, on note $v(S) = \sum_{i \in S} v_i$ la valeur de la collection S et $w(S) = \sum_{i \in S} w_i$ son poids (avec par convention $v(\varnothing) = 0$ et $w(\varnothing) = 0$). On donne aussi un poids limite $W \geq 0$ pour le sac a dos et on demande la valeur maximum OPT d'une collection d'objets S telle que $w(S) \leq W$. En clair, on veut maximiser la valeur de ce que l'on prend en ayant une limite ferme sur le poids total. Ce probleme est un grand classique de l'optimisation combinatoire, utilise pour modeliser de nombreux problemes pratiques. Il est NP-difficile et on se propose de faire un algorithme pseudopolynomial pour le resoudre de maniere exacte, en utilisant l'approche de la programmation dynamique. Cet algorithme a une complexite theorique exponentielle mais est tres efficace en pratique lorsque les valeurs entieres restent relativement petites (c'est a dire du meme ordre de grandeur que le nombre d'objets).
On considere le sous-probleme $PoidsSac(i,V)$ suivant : quel est le poids limite minimum d'un sac qui peut recevoir une collection d'objets de valeur au moins V, avec $V \leq \sum_{j=1}^{i} v_j$, qui sont choisis uniquement parmi les objets d'indice $j \leq i$? Ce poids minimum est note $\overline{OPT}(i,V)$. Soit O une solution qui atteint le poids minimum $\overline{OPT}(i,V)$.

a. Que vaut $\overline{OPT}(i,V)$ si $i \in O$ et i est l'unique objet de O ?

Solution. $\overline{OPT}(i,V) = w(O) = w_i$.

b. Que vaut $\overline{OPT}(i,V)$ si $i \in O$ et i n'est pas l'unique objet de O ?

Solution. $\overline{OPT}(i,V) = w(O) = w_i + \overline{OPT}(i-1, V - v_i)$.

c. Montrer que dans le cas ou $i \in O$, on a toujours $\overline{OPT}(i,V) = w_i + \overline{OPT}(i-1, \max\{0, V - v_i\})$.

Solution. Si i est le seul objet de O, alors $v_i \geq V$ et la formule donne $\overline{OPT}(i,V) = w_i + \overline{OPT}(i-1, \max\{0, V - v_i\}) = w_i + \overline{OPT}(i-1, 0) = w_i$, ce qui est correct d'apres la question a. Si i n'est pas le seul objet de O, alors $v_i < V$ et la formule donne $\overline{OPT}(i,V) = w_i + \overline{OPT}(i-1, \max\{0, V - v_i\}) = w_i + \overline{OPT}(i-1, V - v_i)$, ce qui est correct d'apres la question b.

d. Que vaut $\overline{OPT}(i,V)$ si $i \notin O$?

Solution. Si $i \notin O$ alors il existe une solution optimale a $PoidsSac(i,V)$ qui n'utilise que les objets $j < i$. Cette solution est donc aussi une solution optimale a $PoidsSac(i-1,V)$. Dans ce cas, on a donc $\overline{OPT}(i,V) = \overline{OPT}(i-1,V)$.

e. Donnez une formule de recurrence pour $\overline{OPT}(i,V)$ dans le cas ou $V > \sum_{j=1}^{i-1} v_j$?

Solution. Lorsque $V > \sum_{j=1}^{i-1} v_j$, necessairement i appartient a toute solution optimale a $PoidsSac(i,V)$. D'apres la question c, on a donc $\overline{OPT}(i,V) = w_i + \overline{OPT}(i-1, \max\{0, V - v_i\})$.

f. Donnez une formule de recurrence pour $\overline{OPT}(i,V)$ dans le cas ou $V \leq \sum_{j=1}^{i-1} v_j$?

Solution. Dans ce cas, il est possible qu'il existe une solution optimale qui ne contienne pas i. La valeur de $\overline{OPT}(i,V)$ est donc le min entre l'optimum des solutions qui ne contiennent pas i, c'est a dire $\overline{OPT}(i-1,V)$, et l'optimum des solutions qui contiennent i, qui vaut $w_i + \overline{OPT}(i-1, \max\{0, V - v_i\})$ comme on l'a deja montre a la question c. On obtient donc, dans le cas ou $V \leq \sum_{j=1}^{i-1} v_j$, $\overline{OPT}(i,V) = \min\{\overline{OPT}(i-1,V), w_i + \overline{OPT}(i-1, \max\{0, V - v_i\})\}$.

g. En utilisant les formules des deux questions precedentes, ecrivez un algorithme qui calcule $\overline{OPT}(i,V)$ pour toutes les valeurs possibles et pertinentes de i et V et qui retourne OPT, la valeur de la solution optimale au probleme du sac a dos a valeurs entieres.

Solution.
L'algorithme 3 fait un simple parcours de tous les couples (i,V) valides, c'est a dire avec $V \leq \sum_{j=1}^{i} v_j$, et affecte pour chacun d'eux la valeur de $\overline{OPT}(i,V)$ dans une table, en suivant les formules de recurrence trouvees aux questions e et f (disjonction de cas des lignes 4 a 7). L'initialisation de la recurrence se fait par les valeurs de $\overline{OPT}(i,0)$ qui sont 0 pour tous les i (ligne 2). Le calcul de OPT se fait a la fin en parcourant la derniere ligne de la table, $\overline{OPT}(n,V)$ pour $1 \leq V \leq \sum_{j=1}^{n} v_j$, et en y selectionnant la valeur maximale de V telle que $\overline{OPT}(n,V) \leq W$. Notez que sur cette derniere ligne, le probleme $PoidsSac(n,V)$ n'est pas contraint sur le choix des objets $i \in [\![1,n]\!]$ qui peuvent etre utilises dans la solution. Il est donc identique au probleme initial du sac a dos.

Algorithme 3 : Algorithme pour le calcul de $\overline{OPT}(i,V)$ et de la valeur OPT de la solution optimum au probleme du sac a dos a valeurs entieres.

1 **pour** i *de* 1 *a* n **faire**

2 $\quad\overline{OPT}[i,0] \leftarrow 0$;

3 $\quad$**pour** V *de* 1 *a* $\sum_{j=1}^{i} v_j$ **faire**

4 $\quad\quad$**si** $V > \sum_{j=1}^{i-1} v_j$ **alors**

5 $\quad\quad\quad\overline{OPT}(i,V) \leftarrow w_i + \overline{OPT}(i-1, \max\{0, V - v_i\})$;

6 $\quad\quad$**sinon**

7 $\quad\quad\quad\overline{OPT}(i,V) \leftarrow \min\{\overline{OPT}(i-1,V), w_i + \overline{OPT}(i-1, \max\{0, V - v_i\})\}$;

8 $OPT \leftarrow 0$;

9 **pour** V *de* 1 *a* $\sum_{i=1}^{n} v_i$ **faire**

10 $\quad$**si** $\overline{OPT}(n,V) \leq W$ **alors**

11 $\quad\quad OPT \leftarrow V$;

12 **retourner** OPT;

On note $v^* = \max_{1 \leq i \leq n} \{v_i\}$.

h. Donnez la complexite de votre algorithme en fonction de n et v^*.

Solution. Remarquez que le calcul de la somme $\sum_{j=1}^{i} v_j$ pour tous les $i \in [\![1, n]\!]$ peut etre fait preliminairement et prend seulement un temps $O(n)$. Le calcul de la formule de recurrence (lignes 4 a 7) se fait en temps constant grace a la table $\overline{OPT}(.,.)$. En plus de la boucle "pour" principale (ligne 1) qui s'execute n fois et de la boucle "pour" de la ligne 9 qui s'execute $\sum_{j=1}^{n} v_j = O(nv^*)$, le temps d'execution de l'algorithme depend du nombre d'execution de la boucle "pour" interne (ligne 3) qui s'execute exactement $\sum_{i=1}^{n} \sum_{j=1}^{i} v_j = O(n^2 v^*)$. Au total on obtient donc une complexite de $O(n + nv^* + n^2 v^*) = O(n^2 v^*)$.

On note aussi $w^* = \max_{1 \leq i \leq n} \{w_i\}$.

i. Exprimez la taille t, en nombre de bits, de l'entree de l'algorithme en fonction de n, v^* et w^*.

Solution. Comme chaque valeur v_i est code en binaire sur $\log v_i$ bits et chaque poids w_i est code sur $\log w_i$ bits, cela prend au total un espace $t = \sum_{i=1}^{n} (\log v_i + \log w_i) = O(n(\log v^* + \log w^*))$.

j. La valeur de v^* est-elle polynomiale en fonction de t ?

Solution. Comme t depend logarithmiquement de v^*, on ne peut borner v^* qu'exponentiellement en fonction de t. C'est pour ca que la complexite de l'algorithme 3 telle que nous l'avons exprimee depend en fait exponentiellement de la taille t de l'entree, malgre son aspect a premiere vue polynomial : $O(n^2 v^*)$. Remarquez que si dans l'entree les nombres etaient codes en unaire (a ne pas faire !), alors cette complexite serait bien polynomiale en la taille de l'entree car on aurait alors $t = O(n(v^* + w^*))$, et surtout $t \geq v^*$ et $t \geq n$ (ainsi $O(n^2 v^*) = O(t^3)$). Dans ce cas, on dit que la complexite de l'algorithme est pseudo-polynomiale. C'est a dire polynomiale avec un codage de l'entree en unaire (qui est mauvais), et exponentielle avec un codage naturel en binaire (qui est le bon codage a utiliser).