CONCEPTIONS ALGORITHMES

OLIVIER BODINI

Résumé. Nous continuons l'exploration des algorithmes de type Programmation Dynamique. Nous traiterons grâce à ce principe un problème d'ordonnancement (le problème du sac à dos). Nous introduisons en suite, la notion d'algorithmes gloutons avec pour exemple le problème de l'arbre couvrant optimal et le codage de Huffman.

1. Programmation Dynamique

1.1. **Le problème du sac à dos.** Nous allons ici illustrer le principe de la Programmation Dynamique par un algorithme qui résout le problème dit du sac à dos. Ce problème peut être décrit de la manière suivante. On dispose de n objets qui ont chacun une valeur de v_i euros et un poids de p_i kg et d'un sac à dos qui ne peut pas supporter un poids total supérieur à B kg. Quels objets doit-on prendre pour maximiser la valeur totale que l'on peut transporter dans le sac à dos ? En termes plus mathématiques, le problème peut donc être posé ainsi : Etant donnés B un nombre entier positif et v et p deux séquences de n nombres entiers positifs, trouver un sous-ensemble I de $\{1, ..., n\}$ tel que $\sum_{i\in I} p_i \leq B$ et pour lequel $\sum_{i\in I} v_i$ est maximal.

1.1.1. *Approche directe.* Une approche directe du problème pourrait consister à regarder tous les sous-ensembles I d'indices, de calculer pour chacun d'eux, $\sum_{i\in I} p_i$ et parmi ceux qui sont valides (ceux tels que $\sum_{i\in I} p_i \leq B$) prendre celui pour lequel $\sum_{i\in I} v_i$ est maximal. Mais cette démarche naïve n'est pas algorithmiquement convaincante car il y a 2^n sous-ensembles de $\{1, ..., n\}$ ce qui nous amènerait à faire un nombre exponentiel d'opérations pour trouver une solution.

1.1.2. *Sous-Structure optimale.* Encore une fois, une amélioration vient de la structure particulière que possède une solution optimale. Supposons que l'on ait une solution optimale I_{max}. On peut distinguer deux cas :

- soit $n \in I_{max}$ et dans ce cas, $I_{max}(C)\backslash\{n\}$ est une solution optimale pour le problème où le sac à dos a pour contenance maximale $C - p_n$ et où l'on cherche à le remplir avec les $n-1$ premiers objets (de poids respectifs $p_1, ..., p_{n-1}$ et de valeurs $v_1, ..., v_{n-1}$). En effet, si on avait une solution meilleure I' pour ce dernier problème, le sous-ensemble d'indices $I' \bigcup \{n\}$ serait une solution qui contredirait l'optimalité de la solution I_{max}.
- soit $n \notin I_{max}$ et dans ce cas, $I_{max}(C)$ est une solution optimale pour le problème où le sac à dos a pour contenance maximale C et où l'on cherche à le remplir avec les $n-1$ premiers objets. Car comme précédement, si on avait une solution meilleure I' pour ce dernier problème, le sous-ensemble d'indices I' serait une solution qui contredirait l'optimalité de la solution I_{max}.

On remarque donc que les solutions optimales du problème du sac à dos contiennent des solutions optimales pour le problème du sac à dos sur des instances plus petites.

1.1.3. *Une récurrence pour trouver la valeur optimale que l'on peut transporter.* Soit $p = (p_1, ..., p_n)$ et $v = (v_1, ..., v_n)$ deux séquences de n nombres entiers positifs. Nous notons pour tous $i \leq n$ et $C \geq 0$, $P_i(C)$ le problème de sac à dos dont les données initiales sont C pour la contenance maximale du sac à dos et $(p_1, ..., p_i)$ et $(v_1, ..., v_i)$ pour les deux suites respectivement des poids et des valeurs des i objets du problème. De plus, on note $I_i(C)$ un sous-ensemble d'indices qui permet d'obtenir la solution optimale du problème $P_i(C)$ et on note $M_i(C)$ la valeur maximale transportable $M_i(C) = \sum_{i\in I_i(C)} v_i$. On regarde maintenant la structure de $I_i(C)$: soit $i \in I_i(C)$ et dans ce cas,

$I_i(C)\backslash\{i\}$ est une solution optimale pour le problème $P_{i-1}(C-p_i)$, soit $i \notin I_i(C)$ et dans ce cas $I_i(C)\backslash\{i\}$ est une solution optimale pour $P_{i-1}(C)$. Comme on ne sait pas à l'avance si i appartient ou non à $I_i(C)$, on est obligé de regarder quelle est la meilleure des deux solutions. On a donc que :

$$M_i(C) = \max\{M_{i-1}(C), M_{i-1}(C-p_i) + v_i\}$$

1.1.4. *Pseudo-code pour la valeur optimale.* On va commencer par calculer le coût optimal par une approche itérative. L'entrée est deux tableaux $P[1..n]$ et $V[1..n]$ contenant respectivement les valeurs $p_1, ..., p_n$ et $v_1, ..., v_n$ et une valeur B représentant la contenance maximale du sac à dos. La procédure utilise un tableau auxiliaire $M[0..n, 0..B]$ pour mémoriser les coûts $M_i(C)$ et un tableau $X[1..n, 1..B]$ dont l'entrée (i, C) contient un booléen qui indique si i appartient ou non à une solution optimale du problème $P_i(C)$.

Algorithme 1 : VALEURMAX

Entrées : Deux tableaux d'entiers positifs $P[1..n]$ et $V[1..n]$ et B un nombre entier positif.
Sorties : Rien

```
1  VALEURMAX(P, V, B)
2  n := longueur[P] ;
3  pour C allant de 1 à B faire
4      M[0, C] := 0
5  pour i allant de 1 à n faire
6      M[i, 0] := 0
7  pour C allant de 1 à B faire
8      pour i allant de 1 à n faire
9          q1 := M[i − 1, C];
10         si C − P[i] < 0 alors
11             q2 := −∞
12         sinon
13             q2 := M[i − 1, C − P[i]] + V[i]
14         si q1 < q2 alors
15             M[i, C] := q2; X[i, C] := vrai
16         sinon
17             M[i, C] := q1 ; X[i, C] := faux
```

Analyse de l'algorithme VALEURMAX :

Preuve de Terminaison :
Evident, il n'y a que des boucles prédéfinies.

Preuve de Validité :
L'algorithme commence aux lignes 3-6 par l'initialisation des cas de base. S'il n'y a pas d'objet à prendre ($i = 0$), la valeur maximale transportable est évidement 0 quelque soit la contenance du sac. Donc $M[0, C] := 0$, pour $C = 0, 1, ..., B$. De même si le sac a une contenance nulle, on ne peut prendre aucun ojet. Donc $M[i, 0] := 0$, pour $i = 1, ..., n$. On utilise ensuite lignes 7-17 la récurrence pour calculer $M[i, C]$ pour $i = 1, 2, ..., n$ et $C = 1, ..., B$. On remarque que, quand l'algorithme en est au calcul de $M[i, C]$, il a déjà calculé les valeurs $M[i-1, C]$ et $M[i-1, C-p_i]$ si $i > 0$ et $C - p_i \geq 0$. Si $C - p_i < 0$, cela veut dire que l'objet de poids p_i est trop lourd pour être rajouté dans le sac. Dans ce cas, on interdit alors la prise de cet objet ($M[i, C] = M[i-1, C]$ car $q_1 > q_2$ dans tous les cas). Quand on accède à la ligne 15, on a $M[i-1, C-p_i] > M[i-1, C]$. Ceci veut dire que l'on peut prendre le i-ième objet dans le sac pour construire une solution optimale.

Analyse de la Complexité en nombre de comparaisons :
Le nombre de comparaisons est nB (une comparaison à chaque passage dans la double boucle ligne 7-8). La procédure VALEURMAX fait donc $\Theta(nB)$ comparaisons. L'algorithme nécessite un espace de stockage $\Theta(nB)$ pour le tableau M. VALEURMAX est donc beaucoup plus efficace que

la méthode en temps exponentiel consistant à énumérer tous les sous-ensembles d'indice possibles et à tester chacun d'eux. Néanmoins l'algorithme du sac à dos est ce que l'on appelle un algorithme pseudo-polynomial. En fait, la valeur B peut être stockée en machine sur $\lceil \log_2 B \rceil$ bits et donc le nombre de comparaisons nB est exponentiel par rapport à la taille mémoire pour stocker B. On peut montrer de plus que le problème du sac à dos est un problème NP-complet.

1.1.5. *Construction d'une solution optimale.* Bien que VALEURMAX détermine la valeur maximale que peut contenir un sac à dos, elle ne rend pas le sous-ensemble d'indices qui permet d'obtenir la valeur maximale. Le tableau X construit par VALEURMAX peut servir à retrouver rapidement une solution optimale pour le problème $P_n(B)$. On commence tout simplement en $X[n, B]$ et on se déplace dans le tableau grâce aux booléens de X. Chaque fois que nous rencontrons $X[i, C] = faux$, on sait que i n'appartient pas à une solution optimale. Dans ce cas, on sait qu'une solution optimale pour le problème $P_{i-1}(C)$ est une solution optimale pour le problème $P_i(C)$. Si l'on rencontre $X[i, C] = vrai$, alors une solution optimale pour le problème $P_{i-1}(C - P[i])$ peut être étendue à une solution optimale pour le problème $P_i(C)$ en lui rajoutant i. Ceci permet d'écrire le pseudo-code récursif suivant :

Algorithme 2 : AFFICHER-INDICE

Entrées : les tableaux X et P et deux entiers i, C
Sorties : Rien mais affiche une solution optimale

```
1 AFFICHER-INDICE(X, P, i, C)
2 si i = 0 alors
3 |  Stop
4 si X[i, j] = faux alors
5 |  AFFICHER-INDICE(X, P, i − 1, C);
6 sinon
7 |  AFFICHER-INDICE(X, P, i − 1, C − P[i]);
8 |  afficher(i);
```

Nous proposons maintenant une version récursive de l'algorithme. Elle est composée de deux parties, une partie initialisation (MÉMORISATION-VALEURMAX) et une partie construction récursive (RÉCUPÉRER-VALEURMAX).

Algorithme 3 : MÉMORISATION-VALEURMAX

Entrées : Deux tableaux d'entiers positifs $P[1..n]$ et $V[1..n]$ et B un nombre entier positif.
Sorties : La valeur optimale que l'on peut transporter.

```
1 MÉMORISATION-VALEURMAX(P, V, B)
2 n := longueur[P]
3 Créer un tableau d'entiers M[0..n, 0..B];
4 Créer un tableau de booléens X[1..n, 1..B];
5 pour i allant de 1 à n faire
6 |  pour C allant de 1 à B faire
7 |  |  M[i, C] := −1 ; X[i, C] := faux
8 retourner RÉCUPÉRER-VALEURMAX(P, V, n, B)
```

Algorithme 4 : RÉCUPÉRER-VALEURMAX

Entrées : Deux tableaux d'entiers positifs $P[1..n]$ et $V[1..n]$ et deux nombres entiers positifs i et C.

Sorties : Un entier indiquant la valeur maximale que peut transporter le sac de contenance C pour des objets pris parmis les i premiers

```
1  RÉCUPÉRER-VALEURMAX(P, V, i, C)
2  si M[i, C] ≥ 0 alors
3  |  retourner M[i, C]
4  sinon
5  |  si i = 0 ou C = 0 alors
6  |  |  M[i, j] := 0
7  |  sinon
8  |  |  q1 := RÉCUPÉRER-VALEURMAX(P, V, i − 1, C);
9  |  |  si C − P[i] < 0 alors
10 |  |  |  q2 := −∞
11 |  |  sinon
12 |  |  |  q2 :=RÉCUPÉRER-VALEURMAX(P, V, i − 1, C − P[i]) + V[i]
13 |  |  si q1 < q2 alors
14 |  |  |  M[i, C] := q2; X[i, C] := vrai
15 |  |  sinon
16 |  |  |  M[i, C] := q1 ; X[i, C] := faux
17 retourner M[i, C]
```

Analyse de l'algorithme :

Preuve de Terminaison :
Il suffit de remarquer que l'on appelle récursivement RÉCUPÉRER-VALEURMAX sur des instances où i a diminué de 1 et que l'algorithme se termine quand $i = 0$.

Preuve de Validité :
Immédiat par définition récursive de $M[i, C]$.

Analyse de la Complexité en nombre de comparaisons :
Il est facile de montrer que l'on fait moins de nB comparaisons car on remplit au plus une fois chaque case du tableau M. Mais dans cette version récursive le tableau M n'est pas nécessairement entièrement calculé. Le calcul de la complexité en moyenne est très ardu et sort du cadre de ce cours.

2. Algorithmes Gloutons

En termes imagés, un algorithme glouton est un algorithme qui pour trouver une solution optimale globale, met bout à bout des choix optimaux locaux. Comme par exemple, pour un examen, répondre aux questions que l'on sait le mieux faire en premier (choix locaux optimaux), permet d'avoir la meilleur note possible (optimum global). Evidemment, ce type de démarche est en général voué à l'echec. Néanmoins, il existe des exemples et même un cadre théorique pour des problèmes qui peuvent être résolus de cette manière. Les algorithmes gloutons sont alors souvent utilisés pour résoudre ces problèmes car ils sont simples à implémenter et efficaces.

2.1. Le problème de l'arbre couvrant minimum.

On dispose d'un graphe non-orienté connexe $G = (S, A)$ et d'une fonction longueur l définie sur les arêtes de G telle que $l(e)$ soit la longueur positive de l'arête e. On demande de trouver un sous-ensemble d'arêtes de longueur totale minimale, qui relie tous les sommets.

L'algorithme Glouton que l'on propose est le suivant : On classe des arêtes de la plus courte à la plus longue, ce qui nous donne une séquence $a_1, ..., a_n$. On construit l'arbre en prenant à chaque fois dans cette séquence, la première arête qui ne crée pas de cycle.

Cela donne le pseudo-code suivant :

Algorithme 5 : GLOUTON

Entrées : Un tableau $A[1..n]$ contenant les arêtes du graphes $e_1, ..., e_n$, $L[1..n]$ un tableau contenant les longueurs ($L[i] = l(e_i)$).

Sorties : Un arbre couvrant minimum

```
1 GLOUTON(A, L)
2 F := ∅ ;
3 trier le tableau A par ordre de longueurs croissantes l.
4 pour i allant de 1 à n faire
5     si F ∪ {A[i]} n'a pas de cycle alors
6         F := F ∪ {A[i]}
7 retourner F
```

Pour une implémentation réaliste, voir Introduction à l'algorithmique (Cormen, Leiserson, Rivest, Stein) à la page 494 pour la 1ère édition ou à la page 551 pour la 2ème édition.

2.2. **Le codage d'Huffman.** Voir Introduction à l'algorithmique (Cormen, Leiserson, Rivest, Stein) à la page 331 pour la 1ère édition ou à la page 376 pour la 2ème édition.