Licence informatique - L3 Année 2012/2013

Conception d'algorithmes et applications (LI325)

COURS 6

Résumé. Dans cette sixième séance, nous continuons l'exploration des algorithmes de type Programmation Dynamique. Nous traiterons grâce à ce principe un problème de la théorie des graphes (recherche des plus courts chemins dans un graphe orienté sans circuit négatif) et un problème d'ordonnancement (le problème du sac à dos).

1. Programmation Dynamique en Algorithmique des Graphes

1.1. **Plus courts chemins dans un graphe orienté valué sans circuit négatif.** On considère un graphe $G = (X, A)$ où X de cardinal n est l'ensemble de ses sommets et A l'ensemble de ses arcs. On se donne une application v de A dans $\mathbb{Z}$, $v(a)$ est appelée abusivement la *longueur* de l'arc a. Par extension, la longueur d'un chemin est égale à la somme des longueurs des arcs qui le composent. Le problème consiste à déterminer pour chaque couple de sommets (x, x'), la longueur d'un plus court chemin, s'il existe, qui joint x à x'. On notera cette valeur $d(x, x')$ avec comme convention que $d(x, x') = \infty$ s'il n'existe pas de chemin entre x et x'. On peut de plus supposer qu'entre deux sommets il y a au plus un arc. En effet, s'il en existe plusieurs, il suffit de ne retenir que le plus court (c'est-à-dire celui de longueur minimale). On prendra garde de ne pas se laisser perturber par des notations et une terminologie déplacées. En effet, il n'y a rien de métrique dans ce problème. Cette terminologie est héritée du problème où les longueurs sont toutes positives. Dans ce cas, d est bien une distance au sens mathématique du terme. C'est alors la distance géodésique.

1.1.1. *Sous-Structure optimale.* Il est aisé de remarquer la propriété suivante concernant les plus courts chemins reliant x à x' : Si C est un chemin de longueur minimale joignant x à x' qui passe par z, alors il est en fait la concaténation d'un chemin de longueur minimale reliant x à z et d'un chemin de longueur minimale reliant z à x'. Si ce n'était le cas, en remplaçant l'un de ces chemins par un chemin de longueur inférieure, on contredirait la minimalité de C. On utilise ici le fait qu'il n'y a pas de circuit dont la somme des valeurs des arcs soit négative (sinon la distance entre deux sommets x et x' d'un circuit négatif serait $-\infty$ en faisant une infinité de tours autour du circuit et donc tout chemin reliant x à x' (non nécessairement minimal) pourrait être complété en un chemin de longueur minimale en lui rajoutant une infinité de tours autour du circuit).

Dans la suite, on notera $x_1, ..., x_n$ les sommets de G et pour tout $k > 0$ et tout chemin C, on notera $P_k(C)$ la propriété suivante :
Tous les sommets de C, autres que son origine et son extrémité, ont un indice strictement inférieur à k.

On peut remarquer qu'un chemin C vérifie $P_1(C)$ si et seulement s'il se compose d'un unique arc. D'autre part, la condition $P_{n+1}(C)$ est satisfaite par tous les chemins C du graphe. Notons $d_k(x_i, x_j)$ la longueur du plus court chemin C qui vérifie $P_k(C)$ et qui a pour origine x_i et pour extrémité x_j. Par convention, cette valeur est ∞ s'il n'existe pas de chemin ayant cette propriété.

1.1.2. *Une solution récursive pour le calcul des distances.* On va montrer qu'il y a une construction récursive pour les $d_k(x_i, x_j)$. Tout d'abord, on remarque que s'il y un arc a reliant x_i à x_j, on a $d_1(x_i, x_j) = v(a)$ et sinon $d_1(x_i, x_j) = \infty$. D'autre part on a le lemme suivant :

Lemme 1. *Pour tout $k > 0$, et tout couple de sommets (x_i, x_j), on a :*

$$d_{k+1}(x_i, x_j) = \min(d_k(x_i, x_j), d_k(x_i, x_k) + d_k(x_k, x_j))$$

Preuve. Soit un chemin de longueur minimale reliant x_i à x_j satisfaisant P_{k+1}. Deux cas se présentent : soit il ne passe pas par x_k et on a alors $d_{k+1}(x_i, x_j) = d_k(x_i, x_j)$, soit il passe par x_k et dans ce cas, en utilisant la propriété de sous-structure optimale, on a $d_{k+1}(x_i, x_j) = d_k(x_i, x_k) + d_k(x_k, x_j)$. □

Enfin, on remarque que le calcul des d_k permet de construire d. En effet, il est clair que $d(x_i, x_j) = d_{n+1}(x_i, x_j)$.

1.1.3. *Calcul de la distance minimale entre toute paire de sommets.* En se rapportant à la définition récursive des $d_k(i, j)$, on peut faire appel au principe de la Programmation Dynamique pour calculer les valeurs de manière itérative.

Pour la recherche du plus court chemin dans un graphe à N sommets $x_1, ..., x_N$, on va supposer un tableau $T[1..N, 1..N]$ initialisé de telle sorte que $T[i, j]$ vaut $v(a)$ s'il existe un arc a reliant x_i à x_j et ∞ sinon. En d'autres termes, $T[i, j] = d_1(x_i, x_j)$. On crée aussi un tableau $S[1..N, 1..N]$ qui va nous permettre de reconstruire pour tout couple de sommets (x, x') un chemin minimal reliant x à x'. Si $i \neq j$, $S[i, j]$ contiendra à la sortie l'indice du sommet qui suit i dans un chemin minimal qui relie x_i à x_j. Les valeurs $S[i, j]$ sont initialisées à j s'il existe un arc reliant x_i à x_j et à -1 sinon, $S[i, i]$ est lui initialisé à i.

Algorithme 1 : PLUS-COURTS-CHEMINS(T, S)

Entrées : $T[1..N, 1..N]$ un tableau contenant les $d_1(x_i, x_j)$ et $S[1..N, 1..N]$ un tableau
Sorties : Rien mais on a modifié le tableau T de telle sorte qu'il contienne les $d(x_i, x_j)$

```
1  pour i allant de 1 à N faire
2      pour j allant de 1 à N faire
3          si i ≠ j alors
4              si T[i, j] < ∞ alors
5                  S[i, j] := j;
6              sinon
7                  S[i, j] := −1;
8          sinon
9              S[i, j] := i;
10 pour k allant de 1 à N faire
11     pour i allant de 1 à N faire
12         pour j allant de 1 à N faire
13             q := d[i, k] + d[k, j];
14             si d[i, j] > q alors
15                 d[i, j] := q ; S[i, j] := S[i, k];
```

Analyse de l'algorithme PLUS-COURTS-CHEMINS :

Preuve de Terminaison :
Immédiat.
Preuve de Validité :
Au départ, $T[i, j] = d_1(x_i, x_j)$. Si à l'entrée de la k-ième itération de la boucle externe ligne 10, on a $T[i, j] = d_k(x_i, x_j)$, alors, à la sortie de celle-ci, on a clairement $T[i, j] = d_{k+1}(x_i, x_j)$. On en déduit qu'à la fin de l'algorithme, on a $T[i, j] = d_{n+1}(x_i, x_j) = d(x_i, x_j)$.
Analyse de la Complexité en nombre de d'additions :
Le nombre d'addition faites est N^3, puisque l'on fait une addition à chaque itération des boucles imbriquées lignes 10-12. On a donc un algorihtme en $\Theta(N^3)$ avec un stockage en $\Theta(N^2)$.

1.1.4. *Construction d'un chemin de longueur minimale.* Le tableau S construit par PLUS-COURTS-CHEMINS peut servir à construire rapidement un chemin de longueur minimale entre deux sommets quelconques. En voici un pseudo-code.

Algorithme 2 : AFFICHER-PLUS-COURT-CHEMIN(S, i, j)

Entrées : Le tableau S et deux entiers i, j
Sorties : Rien mais imprime les sommets d'un plus court chemin entre x_i et x_j

```
afficher ("x"_i);
si i ≠ j alors
    AFFICHER-PLUS-COURT-CHEMIN(S, S[i, j], j);
```

Analyse de l'algorithme AFFICHER-PLUS-COURT-CHEMIN :

Preuve de Terminaison : Un chemin minimal reliant $S[i, j]$ à j contient un sommet de moins qu'un chemin minimal reliant i à j. Donc par appels récursifs successifs, on arrive à un appel sur une instance du type (S, k, k). L'algorithme se termine alors.

Preuve de Validité : AFFICHER-PLUS-COURT-CHEMIN(S, i, i) affiche bien un plus court chemin reliant x_i à x_i. Maintenant, si AFFICHER-PLUS-COURT-CHEMIN(S, i, j) affiche un plus court chemin reliant x_i à x_j pour des couples (x_i, x_j) tel qu'un chemin minimum les reliant contient k sommets. Alors pour des couples (x'_i, x'_j) tel qu'un chemin minimum les reliant contient $k+1$ sommets, comme AFFICHER-PLUS-COURT-CHEMIN(S, i', j') appelle AFFICHER-PLUS-COURT-CHEMIN$(S, S[i', j'], j')$, on en déduit aisément qu'il affiche un plus court chemin reliant x'_i à x'_j. Par induction, la validité s'ensuit.

Analyse de la Complexité en nombre d'affichages : Le nombre d'affichages est clairement le nombre de sommets d'un chemin minimal. Cette valeur est majorée par le nombre n de sommets du graphe.

2. Programmation Dynamique pour l'Ordonnancement

2.1. **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.

2.1.1. *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.

2.1.2. *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, $I_{max}(B) \backslash \{n\}$ est une solution optimale pour le problème où le sac à dos a pour contenance maximale $B - p_n$ 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, $I_{max}(B)$ est une solution optimale pour le problème où le sac à dos a pour contenance maximale B 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.

2.1.3. *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 :

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

2.1.4. *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 3 : VALEURMAX(P, V, B)

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

```
1  n := longueur[P] ;
2  pour C allant de 1 à B faire
3      M[0, C] := 0;
4  pour i allant de 1 à n faire
5      M[i, 0] := 0;
6  pour C allant de 1 à B faire
7      pour i allant de 1 à n faire
8          q1 := M[i − 1, C];
9          si C − P[i] < 0 alors
10             q2 := −∞;
11         sinon
12             q2 := M[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;
```

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 2-5 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 6-16 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 14, on a $M[i - 1, C - p_i] + v_i > M[i - 1, C]$. 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 6-7). 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.

2.1.5. *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 4 : AFFICHER-INDICE(X, P, i, C)

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

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

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

Algorithme 6 : RÉCUPÉRER-VALEURMAX(P, V, i, C)

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