Algorithmique TD6
L3 Informatique – ENS Cachan

Guillaume Bury

24 octobre 2016

Exercice 1

On considère le problème d'ordonnancement des tâches suivant. On dispose de n tâches $a_1, \dots, a_n$ dites unitaires (i.e chaque tâche prends exactement une unité de temps à effectuer). A chaque tâche est associé une date d'échéance $1 \leq d_i \leq n$, et une pénalité w_i qui survient si et seulement si la tâche a_i est exécuté à une date postérieur à d_i. On cherche à trouver un ordonnancement des tâches E (c'est à dire une permutation de la liste des tâches) qui minimise le total des pénalités.

1. On dit qu'une tâche est en retard si elle se termine après sa date d'échéance, et en avance dans le cas contraire. Montrer qu'on peut toujours mettre un ordonnancement sous forme en avance d'abord, i.e ou toutes les tâches en avance précèdent les tâches en retard.
2. Montrer que tout ordonnancement sous forme en avance d'abord, peut se mettre sous une forme canonique ou les tâches en avance sont triées par ordre croissant de date d'échéance

On dit qu'un ensemble de tâches est indépendant s'il existe un ordonnancement de ses tâches tel qu'aucune ne soit en retard. On note $\mathcal{I}$ l'ensemble de tous les ensemble indépendants de tâches.

Étant donné un ensemble F de tâches, pour $1 \leq t \leq n$, on note $N_t(F)$ le nombre de tâches de F dont la date d'échéance est inférieure à t.

3. Montrer l'équivalence des propositions suivantes :
   — F est indépendant
   — $\forall 1 \leq t \leq n, N_t(F) \leq t$
   — Si les tâches de F sont ordonnancées par ordre monotone croissant de dates d'échéance, alors aucune des tâches n'est en retard.
4. Montrer que $(E, \mathcal{I})$ est un matroïde
5. En déduire un algorithme permettant de trouver l'ordonnancement optimal d'un ensemble de tâches.

Exercice 2

Soit Σ un alphabet fini et de cardinal supérieur ou égal à deux. On appelle codage binaire une application injective α de l'alphabet Σ dans $\{0,1\}^*$. En utilisant l'opération concaténation, α s'étend de manière naturelle en $\alpha : \Sigma^* \to \{0,1\}^*$.

Un codage est dit préfixe si aucune lettre n'est codée par un mot de code qui est préfixe du codage d'une autre lettre.

1. Montrer que pour un codage préfixe, α est injective sur Σ^*.
2. Montrer qu'on peut représenter un codage préfixe par un arbre binaire dont les feuilles sont les lettres de l'alphabet.

On dit qu'un codage est de longueur fixe quand toutes les lettres sont codées par un mot de code de même longueur. Mais on obtient des codes plus efficaces en associant des codes plus courts aux lettres qui apparaissent le plus fréquemment, quitte à devoir rallonger les codes des lettres qui apparaissent peu fréquemment.

On associe à chaque lettre a de Σ une fréquence d'apparition $f(a)$. Cette fréquence est généralement estimée à partir d'un ensemble de textes représentatif de la langue considérée.

On définit alors le coût d'un codage préfixe par la somme :

$$\sum_{a \in \Sigma} f(a) \cdot |\alpha(a)|$$

et on cherche un codage qui minimise ce coût.

3. Montrer qu'à un codage préfixe optimal correspond un arbre binaire où tout nœud interne a deux fils.
4. Montrer qu'il existe un codage préfixe optimal pour lequel les deux lettres dont le nombre d'occurrences est le plus faible sont sœurs dans l'arbre.

Étant données x et y les deux lettres dont le nombre d'occurrences est le plus faible dans w, on considère l'alphabet $\Sigma' = (\Sigma \setminus \{x, y\}) \cup z$ où z est une nouvelle lettre à laquelle on associe $f(z) = f(x) + f(y)$.

5. Soit T' l'arbre d'un codage optimal pour Σ', montrer que l'arbre T obtenu à partir de T' en remplaçant la feuille associée à z par un nœud interne ayant x et y comme feuilles représente un codage optimal pour Σ.
6. En déduire un algorithme recherchant un codage optimal et donner sa complexité.

Exercice 3

On se propose d'étudier les arbres splay (« Splay trees »), qui sont des arbres binaires de recherche qui "s'autoéquilibrent", en ramenant les éléments fréquemment accédés près de la racine de l'arbre. Afin de faire cela on introduit une opération, appelée « splay », qui prend un arbre A et un nœud x, et fais remonter x pour qu'il devienne la racine de l'arbre. L'opération applique récursivement des transformations locales qui ont pour but de faire remonter x petit à petit.

1. Distinguer trois cas différents (modulo symétries), et proposer une transformation locale qui fait remonter x pour chacun de ces cas (on veut bien sûr que ces transformations locales conservent les propriétés d'un arbre binaire de recherche, et n'ajoutent ni ne retirent des éléments de l'arbre).
2. Expliquer comment implémenter les opérations suivantes en utilisant l'opération de « splay » :
   — Insérer un élément (à la fin, le nouvel élément doit être à la racine de l'arbre)
   — Supprimer un élément
   — Fusionner deux arbres splay
   — Séparer un arbre splay en deux : étant donné un nœud x de l'arbre, il faut rendre un arbre qui contient les éléments (strictement) plus petits que x, et un autre qui contient les éléments (strictement) plus grand que x.
3. On veut maintenant analyser la complexité amortie de opérations ci-dessus. On définit pour cela les notions suivantes :
   — La taille d'un nœud r : $\text{size}(r)$ est le nombre de nœuds du sous arbre qui a r pour racine.
   — Le rang d'un nœud r : $\text{rank}(r) = \log_2(\text{size}(r))$
   Trouver une fonction de potentiel adéquate, et calculer la complexité amortie de l'opération de « splay ».
4. En déduire la complexité amortie d'une suite d'opérations quelconques.