INF4100
Laboratoire no. 10

09/04/08

(Algorithmes voraces)

1. Encodage de Huffman (exercice 24 du manuel)

À l'aide de l'algorithme de Huffman, construisez un code binaire préfixe pour les caractères et nombres d'occurences suivants :

A	B	I	M	S	X	Z
12	7	18	10	9	5	2

2. Code de Huffman

Supposons un ensemble de huit (8) caractères tels que tous les caractères ont exactement la même fréquence d'apparition (12,5 %) — c'est-à-dire exactement le même nombre d'occurrences. En utilisant l'algorithme d'encodage de Huffman vu en classe, quelle forme aurait alors le code de Huffman résultant ?

3. Code de Huffman

Nous avons vu en cours une version MPD de l'algorithme de Huffman. Plus précisément, nous avons vu la fonction `huffman` qui, à partir d'un tableau de caractères et d'un tableau de leurs nombres d'occurrences, construit un arbre binaire représentant un encodage de Huffman (et ce en utilisant une file de priorité).

Écrivez, en MPD, une procédure `emettreEncodage` qui émet, sur `stdout`, l'encodage résultant, et ce en parcourant l'arbre binaire retourné par l'appel à `huffman`. Un exemple de sortie serait comme suit :

```
B => 0
C => 10
A => 11
```

Rappel : Voici la déclaration du type `Arbre`, retourné par `huffman` :

```
type Arbre    = ptr ArbreRec;
type ArbreRec = rec( char car; int nbOccs; Arbre gauche, droite;  );
```

Indice : En MPD, l'opérateur «`||`» permet de concaténer deux chaînes, de type `string[]`. Par exemple — mais voir le site web pour un exemple (`strings.mpd`) avec des procédures (récursives) :

```
"abc" || "def" => "abcdef"
""    || "x"   => "x"
```

4. Code de Huffman

Dans l'algorithme de Huffman vu en cours, lorsque deux sous-arbres sont associés au même nombre d'occurrences (même niveau de priorité), c'est le choix fait par la mise en oeuvre des files de priorités qui déterminera lequel des sous-arbres sera sélectionné comme sous-arbre gauche (`a1`, résultat du premier appel à `retirerMin`) ou comme sous-arbre droit (`a2`, résultat du deuxième appel à `retirerMin`).

Supposons qu'on désire plutôt que, lorsque deux sous-arbres ont le même niveau de priorité (même nombre d'occurrences), ce soit le sous-arbre *le moins profond (le moins haut)* qui soit utilisé comme sous-arbre gauche. De quelle façon faudrait-il modifier la procédure `huffman` pour mettre en oeuvre cette façon de procéder? Est-ce que cela a un effet sur la complexité asymptotique de l'algorithme?

5. Algorithmes pour arbre couvrant minimum

Soit G un graphe non-connexe, c'est-à-dire qu'il existe (au moins) un sommet v_i et un sommet v_j ($i \neq j$) tel qu'il n'y a aucun chemin entre v_i et v_j dans G.

a. Soit l'algorithme de Prim, version pseudocode révisée (Algorithme 7) dans le cahier de notes de cours (p. 90). Que se passe-t-il si on exécute cet algorithme sur un graphe G qui n'est pas connexe? Serait-il possible de corriger ce problème?

b. Même question que la précédente, mais pour l'algorithme de Kruskal (Algorithme 8) dans le cahier de notes de cours (p. 93).

6. Algorithme de Prim

En vous basant sur l'algorithme 7 du cahier de notes de cours (p. 90), écrivez une version MPD de l'algorithme de Prim.

Note : Tel qu'indiqué en classe, il faut ajouter, au début de l'algorithme, l'initialisation de `distance[1]` avec `DEJA_DANS_Y`.

7. Algorithmes pour arbre couvrant minimum

Supposons qu'on ait un graphe G, avec n sommets, pour lequel on a déjà trouvé un arbre couvrant minimal T.

Soit alors un $n+1^{ième}$ sommet qu'on désire ajouter au graphe G pour obtenir un nouveau graphe G'. Soit $D[1:n]$ un tableau qui indique, pour chacun des n sommets de G, le poids, dans G', de l'arête reliant le sommet v_i (de G) au nouveau sommet v_{n+1}.

Comment devrait-on alors procéder pour déterminer, de façon aussi efficace que possible, un arbre de recouvrement pour G', possiblement en utilisant T? Quelle serait la complexité asymptotique?

Indice : Proposition et figure 4.1, pp. 86–87 du cahier de notes de cours.

8. Files de priorité et algorithme de Kruskal

La mise en oeuvre des files de priorité avec monceau (`fp3.mpd`) utilise la procédure auxiliaire suivante :

```
# Procedure auxiliaire: transporte un element vers le haut a sa bonne position.
procedure deplacerVersLeHaut( int pos )
{
  while ( parent(pos) >= 1
          & elements[parent(pos)].prio > elements[pos].prio ) {
    elements[pos]  :=: elements[parent(pos)];
    pos = parent(pos);
  }
}
```

Supposons que dans la comparaison des priorités, on remplace «>» par «>=».

a. De façon informelle, expliquez la différence de comportement entre les deux mises en oeuvre.

b. Soit le graphe présenté à la figure 4.3 du cahier de notes de cours (p. 94). Supposons que dans l'algorithme 9 (p. 95), plutôt que de trier explicitement les diverses arêtes du graphe, on crée une file de priorité, en les insérant en fonction des paires de numéros de sommet composant l'arête, plus spécifiquement, dans l'ordre suivant :

```
v1-v2
v1-v3
v2-v4
v2-v5
v3-v4
v3-v6
v4-v5
v4-v6
```

Pour chacune des mises en oeuvre de `deplacerVersLeHaut` (avec «>» vs. avec «>=»), quel serait l'arbre couvrant minimum produit par l'algorithme de Kruskal en utilisant une file de priorité définie avec cette mise en oeuvre de la procédure?

Exercices tirés des notes de cours :

- Série 5 (si pas faits?) (p. 226) : 1, 2

Exercices tirés des notes de cours :

- Série 5 (p. 229) : Exercices traduits du manuel de Neapolitan & Naimipour (Chap. 4) : 2, 3, 5