IN 101 - TD 11
2 décembre 2011

Énoncé

ENSTA ParisTech

Exercice 1 : Chiffrement à flot

Le principe d'un chiffrement à flot est de générer une *suite chiffrante* à partir d'une clef et de simplement XORer cette suite au message à chiffrer. Avec un tel chiffrement, le déchiffrement est la même opération que le chiffrement : rechiffrer avec la même clef permet de retrouver le message d'origine. Dans cet exercice, la fonction `rand` sera utilisée pour générer la suite chiffrante et la clef sera un simple entier utilisé pour initialiser `srand`.

1.a] Écrivez une fonction `main` qui lit sur la ligne de commande une clef (un entier) et les noms d'un fichier d'entrée et d'un fichier de sortie. Cette fonction doit ouvrir en lecture le fichier d'entrée (et vérifier que l'opération a fonctionné) et en écriture le fichier de sortie (et aussi vérifier l'opération).
(Pour ceux qui vont vite) Vous pouvez programmer des options `-k` pour donner la clef, et `-i` et `-o` pour les fichiers d'entrée et sortie. Votre `main` devra pouvoir prendre ses arguments dans n'importe quel ordre et seule la clef est obligatoire. Si les fichiers d'entrée/sortie ne sont pas donnés, l'entrée/sortie standard sera utilisée à la place.

1.b] Écrivez une fonction de chiffrement ayant comme prototype *`void`* `encrypt(`*`int`* `key,` *`FILE*`* `input,` *`FILE*`* `output)` qui chiffre le contenu du fichier `input` dans un fichier `output`. Cette fonction commencera par initialiser `srand` avec `key` puis chiffrera `input` caractère par caractère. Le chiffrement consiste à XORer chaque octet du fichier d'entrée avec `rand() & 255` (qui permet d'extraire un octet généré par `rand`).

Utilisez cette fonction dans votre `main` et vérifiez que le chiffrement et le déchiffrement marchent. Avec ce programme, vous pouvez normalement échanger des fichiers chiffrés avec vos camarades (à condition de leur communiquer la clef pour déchiffrer).

Exercice 2 : Analyse de fréquence des lettres

Écrivez un programme qui prend en argument le nom d'un fichier d'entrée et d'un fichier de sortie (comme dans l'exercice précédent, vous pouvez programmer des options `-i` et `-o`) et qui écrit dans le fichier de sortie le nombre d'apparitions de chacun des caractères présents dans le fichier d'entrée.

Votre programme devra simplement parcourir le fichier d'entrée caractère par caractère et maintenir à jour 256 compteurs (un pour chaque caractère possible). Le fichier de sortie ne doit contenir que les caractères qui apparaissent dans le texte et doit ressembler à cela :

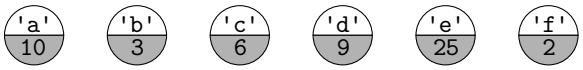
```
' ' 1593
'a' 600
'b' 61
'c' 249
'd' 346
...
```

Vous pouvez utiliser le fichier `~finiasz/huffman.txt` et vérifier les nombres ci-dessus.

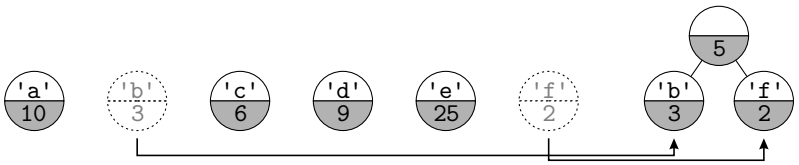
Exercice 3 : Codage de Huffman

Le codage de Huffman est un algorithme de compression simple qui consiste à coder chaque caractère d'un texte par une séquence binaire : les caractères les plus fréquents sont représentés par une séquence courte et les plus rares par une séquence longue. Pour déterminer quelle séquence correspond à quelle lettre, il faut partir des fréquences d'apparition des lettres et s'en servir pour construire un arbre.

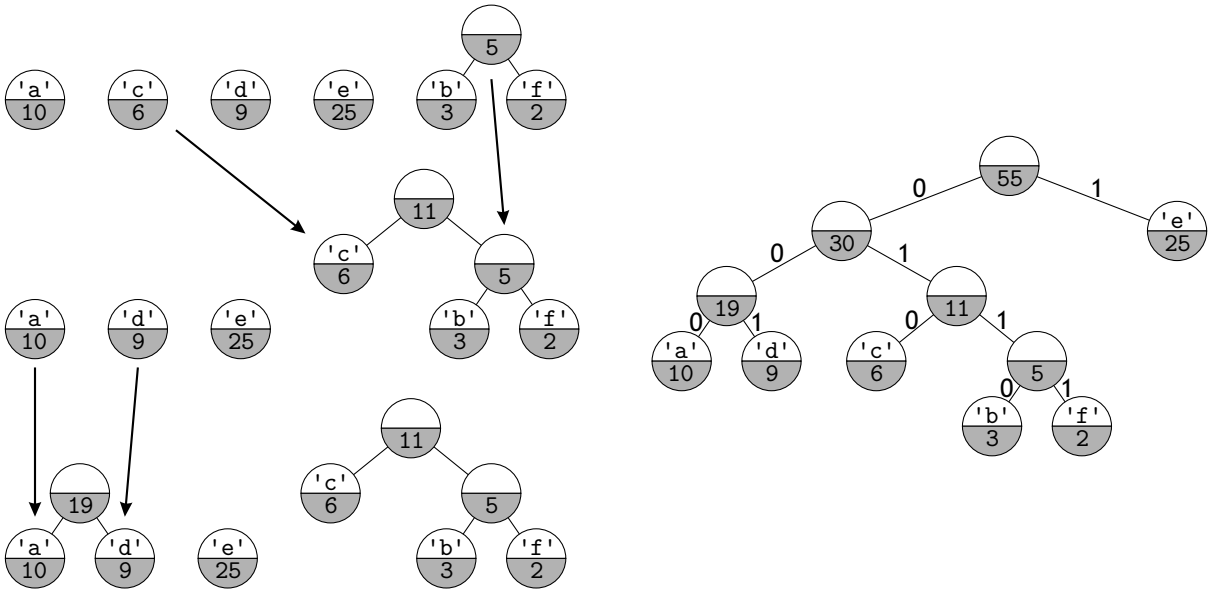
La construction de l'arbre de Huffman commence par construire toutes les feuilles de l'arbre. Chaque feuille contient un caractère (issu du fichier de fréquences) et sa fréquence d'apparition.



Ensuite, la construction de l'arbre se fait de façon itérative. À chaque étape on choisit les deux arbres les plus rares et on les fusionne en un nouvel arbre auquel on attribue comme fréquence la somme de ses deux fils.



On fusionne ainsi tous les arbres jusqu'à n'en avoir plus qu'un : c'est l'arbre de Huffman.



Le codage correspondant à chaque caractère correspond au parcours qu'il faut effectuer dans l'arbre pour atteindre la feuille contenant ce caractère : descendre à gauche correspond à un 0, descendre à droite à un 1. Dans l'arbre ci-dessus on obtient donc : `a=000`, `d=001`, `c=010`, `b=0110`, `f=0111`, `e=1`.

3.a] Écrivez une fonction de prototype ***node*** `build_tree(`***FILE***` freq_file)` qui lit un fichier de fréquences de lettres (au format de sortie de l'exercice précédent) et construit l'arbre de Huffman associé. Vous devrez utiliser la fonction `fscanf` pour lire le contenu du fichier. Il faut aussi définir une structure d'arbre : chaque nœud contient un ***char***, une fréquence (***int*** ou ***double***, comme vous préférez) et deux pointeurs vers les fils gauche et droit. Vous pouvez stocker tous les arbres que vous créez dans un tableau (de taille 256 par exemple) qu'il suffit de parcourir pour trouver l'arbre le plus rare.

3.b] Écrivez une fonction de prototype *`char**` `build_table(`node* `H)`* qui construit la table (de 256 cases) de correspondance caractère/séquence binaire. La case `'a'` du tableau renvoyé doit contenir une chaîne de caractères correspondant au codage de `a`. Cette fonction pourra faire appel à une autre fonction récursive qui fera le parcours de l'arbre de Huffman en construisant les chaînes de catactères.

Dans le `main`, lisez un fichiers de fréquences de lettres, construisez l'arbre de Huffman associé et la table de correspondance et faite afficher la table pour vérifier que les lettres les plus fréquentes ont bien une représentation plus courte que les autres.

3.c] Programmez une fonction `main` qui permet de coder/décoder un texte à l'aide du codage de Huffman. Ce programme prend en argument un fichier de fréquences de lettres, un fichier d'entrée, un fichier de sortie et éventuellement un argument de plus pour savoir s'il faut coder ou décoder. Le codage se fait en lisant le fichier d'entrée caractère par caractère et en écrivant la séquence binaire associée dans le fichier de sortie (le fichier de sortie doit contenir des caractères `'0'` et `'1'`). Le décodage se fait en lisant le fichier d'entrée "bit par bit" et en descendant dans l'arbre au fur et à mesure : quand une feuille est atteinte on écrit le caractère correspondant dans le fichier décodé et on repart du haut de l'arbre. Vérifiez qu'un fichier codé puis décodé n'est pas modifié.

Pour vérifier que votre algorithme compresse effectivement, vous pouvez vérifier que le fichier codé a bien une taille plus petite que le fichier d'origie. Attention, dans le fichier codé chaque bit est écrit sur un caractère (donc un octet), en l'écrivant en binaire directement il serait 8 fois plus petit. Il faut donc vérifier que le fichier codé est moins de 8 fois plus grand que le fichier d'origine.