Huffman coding

bit.ly/Projet_Huffman

Christopher Marshall-Breton

Présentation

L'objectif de ce projet est d'implémenter un algorithme de compression. Cela signifie que l'on cherche à réduire la place que prend une information, **sans perte de données**.
Nous travaillerons ici sur des fichiers textes.
Le codage de Huffman repose sur la traduction en un code court d'un caractère, en fonction de sa fréquence d'apparition. Plus un caractère apparaît souvent dans le texte à coder, plus sa traduction sera courte.
Vous savez peut-être qu'un caractère est codé sur un octet. Cela signifie que pour chaque caractère d'un texte, il nous faut 8 bits en mémoire. Un texte de 100 caractères va donc prendre 800 bits en mémoire. Comment faire alors pour réduire ce chiffre ?

Préambule

Travailler sur des bits directement serait contre productif ici, puisque nous nous intéressons surtout au fonctionnement du code de Huffman. Nous allons donc "visuellement" représenter les bits en caractères 0 et 1.

La première étape du projet va être de traduire un texte classique en une série de 0 et de 1, en codant chaque lettre sur un octet.
Exemple : la lettre **'a'**, sera traduite dans un premier temps par la chaîne **"01100001"**. Cela nous permettra de savoir en comptant les chiffres, combien de bits notre texte prend à l'origine.

Nous allons ensuite le traduire en utilisant le code de Huffman, pour comparer et estimer la place gagnée.

Principe

L'objectif est simple. Par défaut, on code chaque caractère sur un octet, donc 8 bits. Cela signifie que pour un texte de 1 000 caractères, il faut forcément 8 000 bits. Comment faire pour réduire ce nombre au maximum ?

On va diviser en 3 étapes:

Les occurrences

Si on doit coder tous les caractères de la table ascii, il y a des fortes chances qu'on ne puisse pas réduire à moins de 7 ou 8 bit par caractères. Mais dans la majorité des textes, on n'utilise que peu de symboles de cette table. On va donc pouvoir redéfinir une correspondance en fonction des caractères présents dans le texte.
Exemple : si on n'a que des **A** et de **L**, on peut coder **A** par **0**, **L** par **1**, et c'est fini.
On a ainsi gagné 7 bits sur chaque lettre, un texte de 100 A et L prendra donc 100 bits au lieu des 800 nécessaires auparavant.
Pour le texte "**AAALALALALLALA**", on passe de **112** bits (14 * 8) à **14** bits.

Bien sûr, avec plus de lettres, ça se complique.
Si l'on n'a que des **P** des **K** et des **D**, on peut coder les **P** par des **0**, les **K** par des **11** et les **D** par des **10**.

Note : observez qu'aucune lettre n'est codée par un 1. On y reviendra dans la partie arbre.

Avec le dictionnaire suivant,

P	**0**
K	**10**
D	**11**

On gagne 7 bits sur les P, 6 bits sur les K et 6 bits sur les D.
Cela dit, un texte avec des millions de K et D, et un seul P gagnera moins de place avec le même dictionnaire qu'un texte avec des millions de P et peu de K et de D.

Pour le texte "**PPKKDPKDPKDPKP**", on passe de **112** bits (14 * 8) à **22** bits (6*1 + 5*2 + 3*2).

En effet, la place que prendra le texte compressé sera :

$$\Sigma\ (o * b)$$

avec **o** le nombre de fois que le caractère apparaît, et **b** la place qu'il prend à représenter en bits.

En résumé, plus un caractère apparaît souvent, moins il faut que sa représentation ne prenne de place !

Il nous faut donc impérativement une liste contenant tous les **caractères** présents dans le texte, ainsi que leurs **occurrences**.

Exemple :

Alice.txt

```
Alice was beginning to
get very tired of
sitting by her sister on
the bank, and of having
nothing to do
```

Caractère	Occurrences
	20
n	10
i	9
t	9
e	8
o	7
g	6
r	4
s	4
a	4
h	4
b	3
d	3
f	2
v	2
y	2
k	1
A	1
l	1
w	1
c	1

On note que l'espace est le caractère apparaissant le plus souvent, il faudra donc le coder par une suite binaire la plus courte possible.

L'arbre

Maintenant que l'on a les caractères à représenter, et leurs occurrences, on peut enfin attaquer le principe du code de Huffman.
Tout repose sur comment assigner à chaque lettre une suite binaire, en fonction de sa **fréquence** et **sans préfixe** !

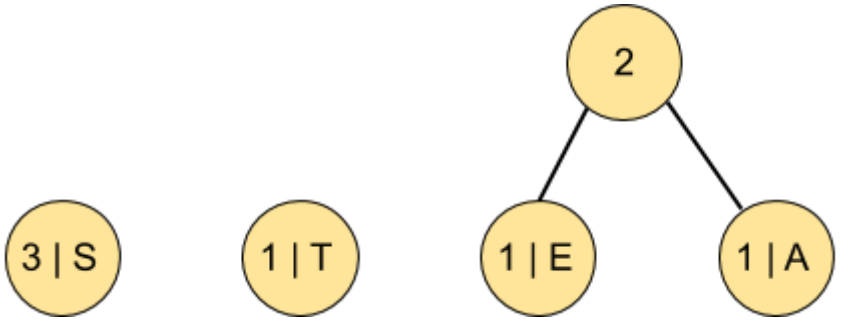
Pourquoi sans préfixe ?
Reprenons l'exemple d'un texte ne comportant que des **P**, **K** et **D**.
On ne peut pas coder les **P** par des **0**, **K** par des **1** et **D** par des **10**, car dans ce cas, on ne peut pas distinguer **KP** de **D** (tous deux représentés par **10**).
De même, si une lettre est codée par **XXX**, on ne peut pas coder une autre lettre par **XXX0** ou **XXX1**. Sinon, on n'a aucun moyen de différencier une lettre d'une combinaison de deux autres.
Un caractère ne peut donc pas être codé en tant que préfixe d'un autre.

Pourquoi par fréquence?
On a vu dans la partie précédente qu'une lettre apparaissant de nombreuses fois doit être codée par une suite binaire la plus courte possible.

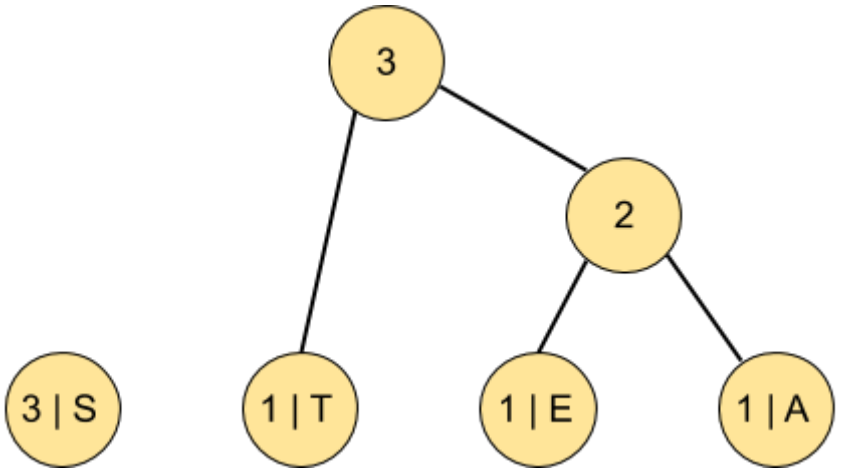
On va donc construire un arbre binaire en mettant au plus profond les caractères apparaissant le moins souvent.
En d'autres termes, il va falloir "remonter l'arbre" de son extrémité avec les occurrences les plus faibles, vers la racine avec ses occurrences les plus fortes.

Exemple : Avec le texte "**TASSES**". On a : S(3) T(1) E(1) A(1).

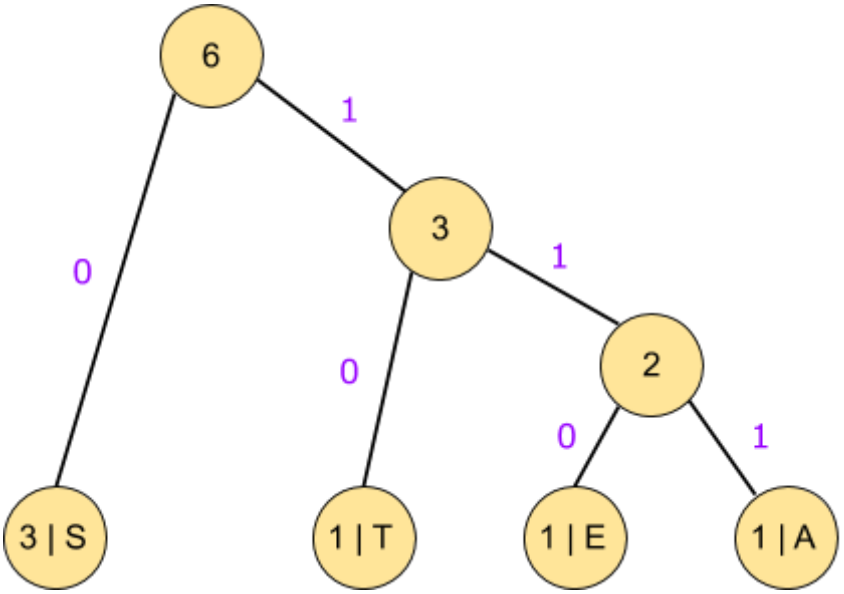
On va regrouper E et A dans un nœud.
Ce noeud va alors contenir deux enfants (E et A), et son poids va maintenant être 2 (1 pour le E + 1 pour le A)



On continue en regroupant les deux plus petites occurrences ensemble (ici 2 et T)



On finit enfin avec les deux noeuds de poid 3 pour compléter notre arbre :



Cet arbre nous permet d'obtenir une **séquence unique** et **sans préfixe** pour chaque caractère, avec un code plus **court** pour les caractères à **occurrences** plus élevées ! En effet, à chaque déplacement vers la gauche, on code par un **0**, pour chaque déplacement vers la droite, on code par un **1**. Plus on avance dans l'arbre (plus le code est long) plus on atteint les caractères à faible occurrence.

On obtient le dictionnaire suivant :

Caractère	bit
S	0
T	10
E	110
A	111

Le dictionnaire

Le dictionnaire est ce qui va nous permettre de coder/décoder le message.
Il suffit de convertir un caractère en sa représentation binaire.
Le mot **TASSES** donnait en binaire ASCII `01010100 01000001 01010011 01010011 01000101 01010011` soit **48** bits.
Grâce au code de Huffman, nous pouvons coder **TASSES** en `10 111 0 0 110 0` soit **11** bits.

Note : pour décoder, on a besoin du dictionnaire… Qui lui même prend de la place !

Partie 1 : De la lettre au bit

On veut ici pouvoir visualiser ce que donnerait un texte en bits. On va donc explicitement traduire chaque caractère en une chaîne binaire de 0 et de 1 correspondant à son octet ASCII.
Voici la table de correspondance, entre un caractère et sa représentation binaire.

Exemple: Le mot `Alice` donnera en binaire
`01000001011011000110100101100011011001010`

Attention... ne codez pas à la main chaque correspondance. Il y a des moyens plus ingénieux… De plus, n'oubliez pas les 0 du début s'il y en a !

A) ★★: Écrire une fonction qui lit un texte dans un fichier, et qui le traduit en son équivalent 0 et 1 dans un autre fichier.

Exemple: (les couleurs sont juste pour montrer la correspondance, et ne sont pas demandées) :

Alice.txt	Output.txt
`Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do`	`01000001011011000110100101100011011001010 01000000111011101100001011100110010000001 10001001100101011001110110100101101110011 01110011010010110111001100111001000000111 01000110111100100000011001110110010101110 10000100000011101100110010101110010011110 01001000000111010001101001011100100110010 10110010000100000011011110110011000100000 01110011011010010111010001110100011010010 11011100110011100100000011000100111100100 10000001101000011001010111001000100000011 10011011010010111001101110100011001010111 00100010000001101111011011100010000001110 10001101000011001010010000001100010011000 01011011100110101100101100001000000110000 10110111001100100001000000110111101100110 00100000011010000110000101110110011010010 11011100110011100100000011011100110111101 11010001101000011010010110111001100111001 00000011101000110111100100000011001000110 1111`

B) ★: Écrire une fonction qui affiche le nombre de caractères dans un fichier txt.

Exemple : pour le fichier `Alice.txt`, il affichera `103`, pour le fichier `Output.txt`, il affichera `824`.

Si tout a été fait correctement, on devrait avoir 8 fois plus de caractères dans le fichier de sortie, puisque chaque caractère est codé sur un octet (8 bit).

On a maintenant un moyen rapide de savoir exactement combien de bits va nécessiter un message en mémoire.

Partie 2 : Le code de Huffman version naïve

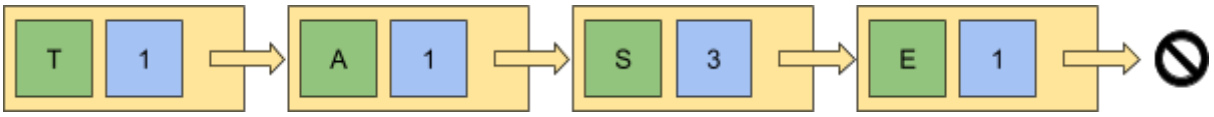
L'objectif de ce projet est de réduire au maximum le nombre de binaires nécessaires pour coder le texte en entrée. Pour le fichier **Alice.txt**, il faudra donc trouver un nombre plus petit que **824 bits**. Ici, on peut viser un output à **410 bits**.

2.1 Les occurrences

Première étape : On veut obtenir la correspondance entre caractère et occurrences.

C) ★★: Ecrire une fonction qui renvoie une liste contenant chaque caractère présent dans le texte, ainsi que le nombre d'occurrences de ce caractère.

Exemple : pour le texte "**TASSES**", on veut la liste :



Indice : On cherche le maillon contenant la lettre correspondante pour y ajouter une occurrence. S'il n'y est pas, on ajoute un maillon contenant cette lettre. L'ordre n'a ici pas d'importance.

2.2 L'arbre

Deuxième étape : Il s'agit de la partie la plus compliquée. L'algorithme n'est pas très long, si l'on organise bien ses structures et les étapes que l'on cherche à coder.

D) ★★★: Ecrire une fonction qui renvoie un arbre de Huffman, à partir d'une liste d'occurrences.

Exemple : voir la partie arbre. L'ordre en cas d'égalité n'a pas d'importance.

Note : Il va falloir comparer des maillons de liste avec des nœuds.
Il peut être judicieux de regrouper les structures…Liste de Noeuds peut être ?

Indice : Une fonction qui isole et renvoie le plus petit élément d'une liste sera la bienvenue.

2.3 Le dictionnaire

Dernière étape : On a maintenant un arbre de Huffman. Il va donc falloir créer le dictionnaire correspondant à cet arbre. En d'autre termes, faire correspondre chaque caractère présent dans le texte à une chaîne de 0 et de 1.

Pour chaque feuille, il va falloir faire correspondre le chemin pour y arriver.

E) ★★: Écrire une fonction qui stocke dans un fichier txt le dictionnaire issu de l'arbre de Huffman.

Exemple : voir la partie arbre, surtout l'arbre final, avec les correspondances des trajets en violet.

On veut en sortie un fichier dico.txt contenant :

dico.txt

```
S:0
T:10
E:110
A:111
```

Le format précis et la séparation entre caractère et suite binaire est à votre discrétion.

Indice : Commencez par essayer d'afficher le chemin jusqu'à une lettre précise.

Indice : utilisez une chaîne de caractères (un tableau) pour retenir le chemin parcouru jusqu'à chaque nœud.

2.4 L'encodage

F) ★★: Écrire une fonction qui traduit un texte en une suite binaire basée sur un dictionnaire de Huffman.

Indice : Pour chaque caractère du fichier d'entrée, il va falloir chercher dans le fichier dictionnaire le code correspondant, et l'écrire dans le fichier de sortie.

Note : Le cas du caractère "retour à la ligne" peut vous poser problème !!

On n'a plus qu'à tout regrouper

G) ★: Écrire une fonction qui compresse un fichier texte. Le fichier d'entrée ne sera pas modifié, un autre fichier, contenant le texte compressé sera créé.

2.5 Le décodage (en option)

H) ★★: Écrire une fonction qui décompresse un fichier texte à partir d'un arbre de Huffman. Le fichier d'entrée ne sera pas modifié, un autre fichier, contenant le texte décompressé sera créé.

Note : On ne pourra pas décompresser un fichier hors de notre programme de compression, puisque l'on a besoin de l'arbre en mémoire.

Bon. On a un moyen de compresser notre fichier texte SANS PERTE !

Mais… C'est trèèèèès long. (environ 13 secondes pour 2 000 lignes)

Il est temps d'optimiser tout ça.

Partie 3 : Optimisation

On va ici retravailler les différentes parties pour voir ce qu'on peut optimiser.

3.1 Les occurrences

L'objectif est de récupérer le nombre de fois où chaque lettre apparaît dans un fichier. On était parti sur une idée : parcourir le fichier, et pour chaque caractère, le chercher dans une liste. S'il n'y est pas, on va l'ajouter.

On va donc faire un grand nombre de recherches, avec quelques ajouts seulement. Surtout si le texte est long.

Il faudrait alors optimiser la partie recherche, même si chaque ajout met un peu plus de temps.

Une recherche dichotomique peut être une bonne idée. Il nous faut donc un tableau trié, et conserver ce tri à chaque ajout d'éléments.

I) ★★: Écrire une fonction qui, par recherche dichotomique, ajoute à un tableau de nœuds une occurrence quand le caractère a déjà été trouvé, ou qui ajoute le nœud du caractère sinon.

I-bis (en option) ★★★: Un AVL ne serait pas mal non plus...

3.2 L'arbre

Il nous faut à présent trier notre tableau non pas par caractère, mais par occurrences.

J) ★★: Écrire une fonction qui trie un tableau de noeuds en fonction des occurrences.

Indice : Certains tris sont plus rapides que d'autres...

Maintenant que l'on a un set trié par occurrences, on peut utiliser cet ordre pour rapidement créer notre arbre. Plus besoin de chercher le minimum à chaque fois. Sauf que si on crée un nouveau noeud ayant la somme des occurrences de ses enfants, on risque de casser cet ordre. L'idée est ici d'utiliser deux files pour construire notre arbre.

La première va contenir tous nos noeuds du tableau trié, la seconde servira de stockage lorsque l'on crée un nouveau noeud.
On comparera à chaque étape les premiers de chaque file pour savoir quels noeuds prendre.

K) ★★★: Écrire une fonction qui, en utilisant deux files, crée l'arbre de Huffman à partir d'un tableau de noeuds trié par occurrences.

3.3 Le dictionnaire

Pour le dictionnaire, le fait de le stocker sur disque pour le lire à chaque fois n'est pas très efficace. On veut juste savoir à quel code correspond la lettre lue dans le fichier à compresser. Passer par l'arbre de Huffman ne nous aide pas, puisqu'on ne sait pas où se trouve la lettre.

En revanche, comme pour les occurrences, on peut retenir les lettres et leurs occurrences en mémoire dans un AVL.

Ainsi, on minimise le nombre d'opérations nécessaires pour obtenir le code correspondant à chaque lettre.

L) ★★★★: Écrire une fonction qui organise les nœuds dans un AVL en fonction de l'ordre des caractères présents.

Indice : Pour chaque lettre(feuille) de l'arbre de Huffman, on veut ajouter à notre AVL un noeud contenant cette lettre et son code.

Note : Le dictionnaire est donc maintenant un arbre, non plus un fichier.
Si vous voulez conserver votre fichier dico.txt (très utile pour les tests) il faudra stocker cet arbre sur disque dans un fichier, au même format que la question E.

3.4 L'encodage

Il ne reste plus qu'à parcourir cet AVL, de façon intelligente, pour trouver la lettre à coder, et du coup le code qui lui est associé !

M) ★★: Écrire une fonction qui compresse un fichier texte de façon optimisée. Le fichier d'entrée ne sera pas modifié, un autre fichier, contenant le texte compressé sera créé.

Note : On passe ici sur du 0.04 secondes pour 2 000 lignes.

3.5 Le décodage

Pour optimiser le décodage, il faut pouvoir recréer l'arbre de Huffman. On ne peut pas le garder en mémoire vive, il va falloir trouver un moyen de le transporter.

On peut donc soit passer par un fichier séparé, soit par stoquer le dictionnaire avant le texte (dans le même fichier).

N) ★★: Écrire une fonction qui décompresse un fichier texte à partir d'un fichier dictionnaire d'Huffman. Le fichier texte d'entrée ne sera pas modifié, un autre fichier, contenant le texte décompressé sera créé.

Indice : Il va falloir recréer l'arbre de Huffman à partir du fichier dico.

Note : A chaque fois que l'on va compresser un fichier, il nous faudra donc le dictionnaire, qui sera la clé de déchiffrage. Ce qui signifie qu'il faut transmettre le dictionnaire également. On peut tout à fait stocker le dictionnaire dans le même fichier de sortie, rendant notre compression plus autonome.

Next Steps (pour aller plus loin)

- reformater le dictionnaire pour prendre le moins de place possible.
- stocker le dico dans le fichier compressé.
- Adapter l'ensemble du programme pour travailler avec un dictionnaire stocké dans le fichier compressé.

Pour les fous

- Refactorer le tout avec des VRAI bits ! => pouvoir réellement compresser un fichier.

Annexes

Si vous voulez tester vos programmes, vous trouverez sur ce drive les fichiers suivants :

- **input.txt** : Un fichier source, contenant le texte à compresser. Il ne contient pas de caractères spéciaux hors table ascii simple.
- **binary.txt** : Une traduction du fichier input.txt en sa représentation binaire.
- **huffman.txt** : Une traduction du fichier input.txt en sa représentation codée par l'arbre de Huffman.
- **dico.txt** : le dictionnaire de Huffman obtenu du fichier input.txt

Note : Le même fichier peut donner des arbres de Huffman différents suivant la priorité donnée en cas d'égalité de fréquence. De même, le dico peut être obtenu en inversant 1 et 0. Vous obtiendrez peut être des choses différentes, qui ne sont pas forcément fausses.