Projet compression sans perte -- Ernesto, Alban

Objectif du projet:

Notre projet de NSI avait un but simple: la compression.
En effet, la compression est extrêmement importante car nos disques durs sont de taille limitée et la place en cloud peut nous coûter cher.
C'est pourquoi nous écrivons aujourd'hui un programme pour compresser les fichiers, images et fichiers textes. De plus, cette compression est sans perte, ce qui est extrêmement important pour les textes car, nous ne pouvons lire un livre qui ne contient que la moitié des mots. Pour les images, ceci peut être un peu moins nécessaire (si la qualité reste bonne) mais, c'est tout de même un objectif. Ceci permettra de compresser des images que nous désirons en bonne qualité, pour les retoucher après, par exemple. Un seul inconvénient de cette compression d'image : la taille du fichier compressé peut être plus importante que la taille du fichier que l'on souhaite compresser. Ceci est d'un grand désavantage car il rompt le principe même de la compression.

Description des algorithmes et de leurs fonctions:

huffman_image et huffman_texte

compare_size: cette fonction consiste à comparer la différence de taille entre deux fichiers, elle est utilisée pour completé le tableau
dico_de_proba: cette fonction consiste à renvoyer un dictionnaire avec le nombre d'occurence de chaque caractère, ceci est utile car cela permet de faire l'arbre par la suite
encodage: cette fonction permet de transformer un caractère en le nombre qui lui correspond pour la compression
compression_Huffman: cette fonction permet de créer l'arbre qui permet de trier les éléments
open_file: permet d'ouvrir un fichier et renvoie le contenu en str
wrtie_file: est utilisé dans la fonction décompresser pour créer un nouveau fichier avec les nom qui convient ainsi
compress: est la fonction principale, qui appelle la fonction **open_file**, **compression_Huffman**, **wrtie_file** et qui renvoie l'arbre Huffman de la compression
write_deco: permet de créer un nouveau fichier avec le nom adéquat à la décompression
huffman_decompression_avec_dico: cette fonction permet de faire la décompression avec un dictionnaire, il renvoie un texte. Ce dernier est décompressé.
import ast: permet de transformer un dictionnaire string en dictionnaire python
decompression_avec_dico: est la fonction principale de décompression avec dico, elle appelle **huffman_decompression_avec_dico** ainsi que **write_deco**.
huffman_decompression_avec_arbre: permet de faire la même chose que **huffman_decompression_avec_dico** mais doit recréer le dictionnaire

huffman_decompression_avec_arbre: la fonction principale de décompression avec arbre. Elle appelle **huffman_decompression_avec_arbre** et **write_deco**.

RLE

read_pgm_file: permet d'ouvrir un fichier .pgm ainsi que de lire son contenu. Ceci est important car le fichier PGM contient un 'header', chose que l'on ne peut faire sans
compression: permet de compresser l'image et de la renvoyer
compare_size: permet de comparer la différence de taille entre le fichier original et le fichier compressé. Elle imprime le nombre d'octets d'avant, d'après et le pourcentage de gain (ou perte, si en négatif)
run_compress: est la fonction principale de compress. Elle appelle la fonction **compression** ainsi que **compare_size**. Nous avons donc, en retour, un fichier compressé dans notre dossier ainsi que le texte de **compare_size**.
decompress: fonction qui s'occupe de la décompression. Elle parcourt les chiffres dans la liste par indice pair (soit 2 par 2) et puis ajoute chiffres[n] fois chiffres[n+1].
run_decompress : Fonction qui s'occupe de toute la décompression de l'image

Principales difficultés rencontrées:

Une des difficultés rencontrées est l'utilisation des arbres Huffman. En effet, la class Node et donc l'arbre lui-même ne peut pas être stockée. Donc même si on le crée on ne peut pas l'utiliser pour décompresser car le format pgm ne sauvegarde que du texte. Donc on a créé une fonction qui décompresse en utilisant l'arbre mais qui ne peut pas vraiment être utilisée en pratique. Donc nous avons utilisé une méthode avec le dictionnaire de valeur et un "sliding window" afin de décompresser le texte / image.
Nous avons aussi eu des problèmes lors de coder l'image RLE. En effet, nous avions, dans un premier temps eu une barre noire qui s'affichait en bas de l'écran lorsque nous compressions puis décompressions des fichier. Le seul problème est que ceci ne se passait pas sur tout les fichiers, donc détecter le problème a été assez compliqué. Pour cela, nous avons dût regarder chacune des fonctions, nous avons finalement trouvé le problème qui était que la compression/décompression ne se faisait pas sur tout le fichier. Nous avons donc réglé ce problème.
De plus, un autre problème que nous avons rencontré est le fait que nous n'avons pas les mêmes valeurs lors de la fonction taux de compression tandis que nous avions exactement les mêmes fonctions. Ce problème ne s'est pas réglé.

Notions apprises durant le projet:

Nous avons appris durant ce projet beaucoup de choses. La première est le fait d'utiliser 'os' pour renommer des fichiers, en créer de nouveaux etc. Ceci a été particulièrement utile lors de coder huffman_image car, nous devions simplement traduire le fichier .pgm en .txt lors de la compression, puis inversement lors de la décompression. Nous avons utilisé la fonction *os.path* pour trouver le chemin jusqu'au fichier. Nous avons ensuite utilisé *.splitext* pour décomposer le texte. Nous avons ensuite changé la dernière partie du texte (l'extension) en .txt au lieu de .pgm.
Nous avons aussi appris comment manier des fichier .pgm, chose qui est importante, même si les formats JPEG et PNG sont plus utilisés.

Tableau:

Codage Huffman : fichiers txt

Nom du fichier	Taux de compression	Temps d'éxectition
adn.txt	67,9% (de 56 bits à 18 bits)	2,6 s
carol.txt	43,9% (de 1 260 768 bits à 707 491 bits)	0,1 s
molly_flanders.txt	45,7% (de 5 542 792 bits à 3 006 977 bits)	0,2 s
tom_jones.txt	44,9% (de 15 406 024 bits à 8 487 025 bits)	0,6 s
tristram_shandy.txt	43% (de 8 289 880 bits à 4 727 410 bits)	0,3 s

Nous remarquons que, plus le temps d'execution est grand, plus le temps d'execution est long (peut etre seulement dût au fonctionnement du fichier adn).
Nous remarquons surtout que la compression est beaucoup plus importante lorsque les caractères se ressemblent

Codage Huffman : fichiers images

Nom du fichier	Taux de compression	Temps d'éxectition
Logo_Python.pgm	64,1% (de 1 729 336 bits à 621 353 bits)	0,1s
kenna.pgm	60,1% (de 1 585 928 bits à 632 167 bits)	0,1s
franck.pgm	60,6% (de 9 706 184 bits à 3 827 293 bits)	0,4s

Nous remarquons que le codage Huffman est beaucoup plus efficace que le codage RLE. En effet, ceci peut être dût au fait que, pour le codage RLE, les pixels pareils doivent être à la suite tandis que pour le codage Huffman, ils ne doivent pas. Ceci montre donc la disparité entre les deux types de codage. Le codage de Huffman est beaucoup plus efficace lorsqu'il n'y a que de petites variations. Nous remarquons aussi que le codage RLE ne fait pas gagner beaucoup de temps (voir pas du tout) en ayant un rendement moins bon. Le codage de Huffman est donc celui a préférer pour les images .pgm

Codage RLE : fichiers images

Nom du fichier	Taux de compression	Temps d'éxectition
Logos_Python.pgm	44, 7% (de 270 926 bits à 149 997)	0,1 s

kenna.pgm	- 5,8% (de 198 241 bits à 209 673 bits)	0,1 s
franck.pgm	- 70% (de 1 213 273 bits à 2 083 250 bits)	0,4 s

Nous remarquons que les petites variations de couleur changent grandement le taux de compression. En effet, dans la photo franck, nous voyons de micro changements dans le fond, par exemple. Ceci fait en sorte que le la compression prenne 70% de place en plus. Car à chaque nouvelle unique valeurs on ajoute 2 chiffres à la place de 1. Ce qui veut dire qu'une image ayant beaucoup de couleurs individuelles peut aller jusqu'à doubler de taille! Nous voyons donc que, le plus une image est simple (en nombre de couleurs), le mieux elle se compresse et le plus il y a de variations (grandes ou petites, même simplement alternance) rends la taille du fichier compressé beaucoup plus grands que l'original