Rapport

Paul Guennec e1900144 - Nathan Dunand e2101774

Contexte

Durant le semestre 6 nous avons eu un projet consistant à implémenter l'algorithme de compression basé sur l'arbre de Huffman en python. Ce projet avait pour but de nous initier à la compression des fichiers, aux arbres, à des notions plus avancées sur le langage python et plus largement à des notions avancées d'algorithmique. Dans la présent rapport, nous expliquerons notre code, notre approche, les difficultés que nous avons rencontrées ainsi que les tests.

Explication technique et approche

Nous avons réalisés deux versions de l'algorithme, un qui code un arbre à partir du fichier (sans perte) et l'autre qui a un jeu de caractère ainsi que des fréquences pré-définies (avec perte). Nous allons détailler le fonctionnement de notre algorithme ainsi que notre approche.

Nous avons réalisé deux versions de l'algorithme, la première construit l'arbre d'Huffman avec la méthode vue dans le cours, à partir du fichier en entrée. Cette version est sans pertes ; lors du décodage, elle restitue strictement le même fichier que celui encodé. Durant l'encodage, l'arbre est construit à partir des caractères présents dans le texte et le texte compressé est écrit à la volée dans le fichier de sortie. Il est important que l'écriture se fasse à la volée et non pas en écrivant tout le contenu d'un coup, au quel cas, pour les fichiers imposants, on satture la RAM. Nous avons observé que python n'était pas adapté à l'écriture en binaire dans un fichier. Nous avons usé de plusieurs stratégies, plus ou moins propres, pour écrire le binaire correctement formaté. C'est à ce moment là que nous avons compris que l'écriture à la volée était une absolue nécessité.

Cet algorithme nécessite que l'arbre soit intégré au fichier compressé, sans cela, impossible pour le receveur de décompresser le fichier. Nous avons donc défini un format de fichier pour standardiser l'encodage mais également pour formaliser le décodage. La structure de notre fichier est la suivante :

Table 1: Description de notre format de fichier

Padding texte 1 o	Padding dictionnaire 1 o	Longueur dictionnaire 2 o	Nombre caractères 1 o	Unicode de la lettre 2 o	Longueur du code 1 o	Codes de toutes les lettres X bits	données Y octets + padding

Le premier octet est donc le padding pour le texte. Il représente le nombre de bits qu'il manque à la fin du fichier pour faire un octet complet. Le padding pour le dictionnaire représente la même chose

pour le dictionnaire. Les deux octets suivants représentent la longueur du dictionnaire, en nombre d'octet. L'octet suivant nous informe sur le nombre de lettre dans le dictionnaire, nous assumons le fait qu'il ne puisse pas y avoir plus de 256 caractères différents. Les deux octets suivants représentent l'encodage en unicode de la lettre, puis un octet, qui code le nombre de bit sur lequel sera écrit le code associé. Cette séquence de trois octets se répète autant de fois qu'il y a de lettres dans le dictionnaire. Lors du décodage, on enregistre ces données pour savoir où l'on se trouve dans la suite de X bits qui vient ; le code de toutes les lettres. Grâce au padding dictionnaire, on rajoute un nombre de bits pour terminer l'octet. Enfin, viennent les données compressées et on rajoute à la toute fin le nombre de bits manquants pour terminer l'octet, se nombre est décrit au tout début du fichier, dans padding texte. Nous avons essayé d'inclure le minimum d'informations par soucis de simplicité. Le décodage se fait en parcourant l'entête du fichier et en faisant correspondre les lettres avec leur code. Ce parcours se fait en une fois ; on avance séquentiellement dans le fichier jusqu'à arriver aux données. Ce faisant, nous n'avons pas besoin d'inclure un offset qui donnerait la taille du header.

La création de ce format de fichier était intéressante, mais assez complexe. Le décodage l'était davantage puisqu'il nécessite un algorithme subtil qui gère les paddings et le nombre variable de bits sur lequel se trouve le code de la lettre. Cette variabilité écarte de fait les cas limites où le code de la lettre est très grand.

Le second algorithme est davantage souple. Il s'appuie sur un jeu de caractère préalablement défini ainsi que les fréquences associées à chacun d'eux. Les caractères que nous avons retenus sont :

- les lettres majuscules non accentuées

- les lettres minuscules non accentuées

- le caractère d'espacement classique (hors tabulation)

- les caractères de ponctuation suivants : . ; : , ‘ « (double quote) ? !

- les caractères spéciaux : \n, -

- les chiffres de 0 à 9 inclus

- les lettres minuscules accentuées : ù, é, è, à, ï, ö, ë, ü, û, ô, ê, â, î, ç

Plus formellement, voici la liste exhaustive :`"A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z", " ", "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z", "\n", "ù", "é", "è", "à", "ï", "ö", "ë", "ü", "û", "ô", "ê", "â", "î", "ç", "." ";", ":", ",", "'", '"', "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "-", "!", "?".`

Pour le calcul des fréquences associées à ce jeu de caractère, nous nous sommes basés sur plusieurs points et avons fait plusieurs hypohèses. Premièrement nous sommes allés chercher la fréquence des lettres minuscules en français sur ce tableau : https://fr.wikipedia.org/wiki/Fr%C3%A9quence_d

%27apparition_des_lettres_en_fran%C3%A7ais. Deuxièmement, nous avons calculé les fréquences pour les lettres majuscules. Nous avons fait plusieurs postulats :

- les lettres majuscules ne sont pas accentuées

- elles sont pour la plupart en début de phrase (calcul de la fréquence)

- elles ont toutes la même probabilité d'apparition (calcul de la fréquence)

Nous sommes conscients que ce sont des approximations grossières mais nous avons jugé que cela n'impactera pas de façon significative le taux de compression pour un texte générique. Selon cette source : https://stephanearnier.com/2017/06/09/focus-nombre-de-mots-par-phrase/, le nombre moyen de mot actuel par phrase est de 15 en français, et le nombre moyen de lettre par mot est entre 4 et 6 lettres ; 4,65 pour un discours journalistique (https://books.google.fr/books?id=gpjWd0pfFxsC&pg=PA40&lpg=PA40&dq=malrieu+rastier+nombre+lettres+moyen+par+mots&source=bl&ots=8kWMzwuoXr&sig=Cnnr0DoIMXRk175Rvihz0YMqYzs&hl=fr&sa=X&ei=865hT63JK9S5hAenu7i9CA#v=onepage&q=malrieu%20rastier%20nombre%20lettres%20moyen%20par%20mots&f=false). Donc une lettre majuscule qui n'apparaîtrait qu'une fois en début de phrase aurait une fréquence de : 1/(15*4,65)=0,01. Nous avons arbitrairement divisé par deux cette fréquence afin que les autres caractères puissent également avoir une fréquence significative. Nous avons supposé que la ponctuation de fin de phrase avait la même probabilité qu'une majuscule, et que les autres caractères de ponctuation (, : ;) était un peu plus fréquents. D'une manière générale, pour les autres caractères, nous avons attribué des fréquences arbitraires.

Par manque de temps, nous n'avons pas implémenté le « Not Known Character » (NYC). Nous avons préféré nous focaliser sur les caractères utilisés dans la langue française ainsi qu'en anglais. La raison à cela est que l'ajout d'un caractère est simple, alors que le NYC, bien que passe-partout, demande davantage de temps. Si un caractère inconnu est rencontré, comme un caractère cyrillique par exemple, il sera remplacé par un espace. Cela induit de la compression avec pertes. Nous sommes conscients de cette limitation importante, la cause principale est le manque de temps dont nous avons soufferts sur la fin du projet.

Tests