Rapport Python2 - Huffman Light

Mathys Rituper

Enguerrand Baudin

Cyberdef 3 - TP6

Objectif :

On cherche à implémenter en Python l'algorithme de compression de Huffman light : elle se base sur un arbre de Huffman prédéfini, connu et identique chez l'encodeur et le décodeur.

Construction de l'arbre :

Le module `heapq` permet de créer et d'utiliser des files en python.

- `heapify` : transformer une liste en file, on peut spécifier un second paramètre qui indique comment l'ordonner
- `heapop` : récupérer un élément de la file
- `heappush` : ajouter un élément à la file

Pour créer l'arbre, on utilise la méthode `arbre_huffman(frequences)` qui le construit récursivement selon la méthode vue en cours.

Note : le dictionnaire de fréquences étant incomplet, on réalise deux ajouts : un caractère "NEXIST" qui sera utilisé pour indiquer qu'on va encoder un caractère absent de l'arbre (plus d'explications dans la suite du rapport), et un caractère "EOF" utilisé pour indiquer quand on atteint la fin du fichier (cas où la version compressée du texte est d'une longueur qui n'est pas multiple de 8 bits).

Construction du dictionnaire

La fonction `parcours(arbre,prefixe,dico)` de construction du dictionnaire fonctionne de la manière suivante :

Pour suivre le cheminement dans l'arbre, on utilise une string de "préfixe" qui indique le chemin parcouru antérieurement depuis la racine pour arriver jusqu'à la node en cours : 0 quand on va à gauche, 1 quand on va à droite.

Ainsi, le préfixe 010 correspond à la node qu'on atteint quand depuis la racine, on va une fois à gauche, puis une fois à droite, puis une fois à gauche.

On va donc venir parcourir tout l'arbre récursivement jusqu'à arriver aux nodes sans éléments à gauche ni à droite, les feuilles qui correspondent aux caractères : à ce moment-là, le préfixe

représente l'encodage du caractère de la feuille, on peut donc ajouter le couple (caractère, encodage) au dictionnaire.

Codage du texte

La fonction `encodage(dico,fichier)` permet d'encoder un fichier selon l'arbre préalablement construit :

- On ouvre le fichier à compresser et on en récupère le contenu.
- On initialise une string `res` qui va venir contenir l'encodage du fichier sous forme de suite de caractères 0 et 1.
- Pour chaque caractère :
    - S'il est connu dans le dictionnaire de fréquences, on ajoute son encodage compressé à `res`.
    - Sinon, on ajoute l'encodage compressé du caractère `NEXIST` à `res`, puis à la suite la chaine de caractères correspondant à l'encodage en UTF-8 du caractère absent du dictionnaire au moyen de `utf8_to_binary(char)`.
- Une fois qu'il n'y a plus de caractères à encoder, on ajoute à `res` l'encodage compressé du caractère `EOF` pour signifier que le fichier est fini.
- Si `len(res) % 8 =! 0`, on vient ajouter des zéros jusqu'à avoir un multiple de 8.
- Pour chaque représentation d'un octet représentée dans `res` par une suite de 8 caractères, on vient calculer la valeur du nombre correspondant à cet octet.
- On ouvre le fichier de sortie et on y écrit cette suite de nombres, encodée en binaire.
- On renvoie le nom du fichier compressé.

Fonction `utf8_to_binary(char)`

Une des difficultés lors de l'encodage est de réussir à encoder les caractères non contenus dans l'arbre de Huffman d'une manière consistante pour permettre un décodage fiable. En effet, les fonctions standard d'encodage de Python encodant en UTF-8 vers la plus petite taille possible, cela produit des résultats de longueur qui ne sont pas des multiples de 8, ce qui complexifierait le décodage. Pour résoudre ce problème, on a donc opté pour la réalisation d'une fonction custom qui transforme un caractère en la chaine binaire (suite de caractères 1 et 0) de longueur multiple de 8 qui permet de le représenter.

Décompression

Les représentations de chaque caractère dans l'arbre de Huffmann étant de longueurs variables ne remplissant pas un octet ou dépassant de ce dernier, il n'est pas judicieux de travailler avec des données binaires brutes pour le décodage. Dans la fonction

`decodage(arbre,fichierCompresse)`, on utilise une approche "tête de lecture" qui permet de contourner ce problème :

- On initalise une string `res` qui contiendra la chaine décodée.
- On vient ouvrir le fichier compressé et on le transforme en une chaine binaire (une suite de '0' et de '1').
- A partir de cette chaîne, on vient faire un système de "tête de lecture" avec un buffer: on déplace la tête de lecture de gauche à droite en déplaçant dans le buffer les 0 et 1 un par un jusqu'à ce qu'on puisse traiter le contenu du buffer, ce qui a pour effet de "décaler à droite" la tête de lecture de la chaine binaire à chaque nouveau déplacement.

  Dès que le contenu du buffer correspond à un encodage valide d'un caractère dans le dictionnaire issu de l'arbre de Huffman, selon le type de caractère :

  - Si le caractère décodé n'est pas `EOF` ou `NEXIST`, on l'ajoute à `res`.
  - Si le caractère décodé est `EOF`, on arrête la lecture du fichier. Les éventuels bits restants sont du padding pour obtenir une longueur en multiple de 8 bits, et ne doivent donc pas être décodés.
  - Si le caractère décodé est `NEXIST`, cela signifie que les bits suivants de la chaîne binaire correspondent à l'encodage UTF-8 d'un caractère absent de l'arbre de Huffmann.

    On vient alors décoder le caractère en question à l'aide de `binary_to_utf8(binary_str)` qu'on ajoute à `res`, puis on "décale à droite" la tête de lecture du nombre de bits sur lesquels était encodé le caractère UTF-8 pour ne pas venir tenter d'interpréter en "mode Huffman" le caractère encodé en UTF-8.
- Une fois que la tête de lecture n'a plus rien à lire, on renvoie `res`, qui contient alors le texte décompressé.

Fonction `binary_to_utf8(binary_str)`

Ici encore, la majeure difficulté rencontrée a été dans le décodage des caractères absents de l'arbre de Huffman. En effet leur longueur dans la chaine binaire issue du fichier encodé est variable (de 1 à 4 octets), mais il est arrivé que certains outils proposés par Python utilisables pour transformer le tronçon de chaine binaire en caractère tronquent automatiquement les 0 "de gauche" inutiles, ce qui causait des décalages ou des erreurs de lecture. Ainsi, il a fallu programmer manuellement une fonction permettant d'extraire le caractère UTF-8 contenu en tête de la chaîne binaire, qui se divise de la manière suivante :

- On vient lire l'octet le plus à gauche de la chaîne binaire pour déterminer la taille d'encodage du caractère, de 1 à 4 octets.
- On isole de la chaine binaire les n d'octets les plus à gauche, n correspondant à la taille d'encodage obtenue à l'étape précédente

- On décode ces octets pour obtenir le caractère recherché
- On renvoie le caractère en question et la longueur de la représentation binaire de ce dernier (pour pouvoir décaler la tête de lecture)

Conclusion et retour d'expérience

Les parties "construction de l'arbre" et "construction du dictionnaire" ont été relativement simples à effectuer, en particulier avec l'aide de ChatGPT.

L'essentiel des difficultés s'est concentré sur les deux dernières parties, notamment sur la gestion des longueurs de chaines pour encoder et décoder les caractères hors arbre de Huffman.

On aurait possiblement pu travailler directement avec des données binaires au décodage sans faire la transposition données binaires brutes du fichier=> chaine binaire pour gagner en performances, mais on aurait perdu en lisibilité et maintenabilité du code, notamment pour la technique "tête de lecture" qui est plus facile à débugger en mode string qu'en données binaires brutes.