RAPPORT TP Python 2 - Codes de Huffman

Structure du projet

- Les sources du projet se trouvent dans le dossier `src/`
- Les données (fichiers d'entrée / de sortie) se trouvent dans le dossier `data/`
- L'implémentation d'huffman sans fréquence se situe dans le fichier `huffman_nofreq.py`
- L'implémentation d'huffman avec fréquence se situe dans le fichier `huffman.py`
- Les fonctions / classes communes aux deux implémentations / génériques se situent dans le fichier `huffman_utils.py`
- Les tests se situent dans le fichier `tests.py`
  - Pour lancer les tests sur les fichiers présents dans `data/`, il faut se trouver dans le dossier `src/` et renseigner leurs noms dans la variable liste `files`, présente dans le main du fichier.

Organisation

Le travail en binôme a été effectué en pair programming.

En effet, les 2 implémentations ont des similitudes. Travailler ensemble permet de bien comprendre les enjeux de chaque implémentation et de ne pas avoir de différence majeure dans la mise en place de celles-ci (sur les similitudes avérées). Utilisation de GIT pour le partage des codes.

L'utilité du fichier huffman_utils.py

Comme des classes ou fonctions sont utilisées dans les 2 implémentations des codes d'Huffman, nous avons créé un fichier huffman_utils.py.

Il sert à éviter les réécritures de code et à réutiliser au maximum les fonctions communes aux 2 implémentations d'Huffman, ainsi que des fonctions génériques comme la conversion d'un caractère sur 16 bits.

Implémentation de la classe Arbre

L'implémentation de la classe Arbre dans le fichier huffman_utils.py est légèrement différente de celle proposée dans le TP, elle contient, en plus :

- La propriété **fréquence**
- Des fonctions de comparaison (`__gt__()` et `__lt__()`)
  - Ces fonctions comparent les arbres en se basant sur leur fréquence

Ces ajouts sont utiles notamment pour indiquer à heapq sur quoi effectuer la comparaison pour trier le tas. Nous avons choisi cette implémentation car nous trouvions cela moins redondant que de créer un triplet pour chaque élément du tas, simplement pour indiquer à heapq comment les comparer.

Méthode de prise en charge du binaire

Pour travailler avec des bits, afin de simplifier le code et la compréhension, nous travaillions avec une chaîne de caractères, composée de 0 et de 1, qui représentant notre code binaire, et ce n'est qu'au moment de l'écriture ou de la lecture le fichier binaire que nous convertissions cette chaîne en liste d'octets (bytearray), afin d'écriture/lire en binaire.

Cependant, la chaîne obtenue n'était pas toujours un multiple de 8, or, Python écrit et lit du binaire depuis un fichier octet par octet, cela résultait donc en des bits de padding ajoutés automatiquement à la fin, faussant le décodage de la fin du fichier.

Pour régler ce problème, nous avons ajouté, juste avant l'écriture dans un fichier, le nombre de bits qui seront ajoutés automatiquement pour compléter un octet. Ce nombre sera stocké sur 8 bits, au tout début du fichier encodé.

Pendant le décodage, après la conversion des données en chaîne représentant du binaire, et avant le décodage, on récupère cette valeur, la converti en entier, et on retire les bits en trop, ainsi que les 8 premiers bits de la chaîne, on résulte uniquement les données à décoder.

Huffman avec fréquence

NYT

Pour l'implémentation avec fréquence d'Huffman, nous avons choisi de prendre en compte les caractères n'étant pas dans le tableau de fréquences, donc de prendre en compte le NYT (Not Yet Transfered). Pour cela, nous avons dans un premier temps, ajouté une branche NYT à l'arbre, puis nous avons assigné tous les caractères non reconnus à cette branche, accompagnés de leur charcode en binaire, codé sur 16 bits.

Nous avons fait le choix de coder le caractère sur 16 bits pour prendre en compte tout l'UTF-8, qui est sur 16 bits, car des caractères du texte proposé en entrée, "leHorla.txt", n'étaient pas présent dans l'ASCII, comme le caractère `ï`, qui est encodé sur plus de 8 bits, rayant cette possibilité des choix qui nous avons à notre disposition.

Exercice 1

heapq a plusieurs méthodes pour comparer les éléments de la liste, il peut :

- Si les éléments sont des triplets, comparer les premiers éléments entre eux
- Sinon, il utilise les fonctions de comparaisons basiques, comme `__gt__` ou `__lt__`, pour comparer ses éléments

Comme dit auparavant, nous avons donc préféré l'approche utilisant les fonctions de comparaison, car elle indiquait un code moins redondant, plus compréhensible, et plus cohérent globalement.

Il est important de noter qu'avant de générer notre arbre, nous ajoutons artificiellement une entrée dans le dictionnaire des fréquences, cette entrée étant le NYT, avec une fréquence de 0, afin qu'il apparaisse au plus bas dans le tableau, et pour ne pas fausser les autres valeurs des fréquences (dont la somme doit faire 1).

Dans un premier temps, on génère une liste de feuilles, chaque élément de la liste étant une instance de la classe Arbre, avec lettre et fréquence indiquées, puis on transforme cette liste en tas avec heapq, ce qui nous permet d'en avoir une version triée sur la fréquence.

Dans un second temps, on génère un arbre d'Huffman à partir de ce tas, en retirant de ce tas les 2 plus petits éléments, puis en créant une nouvelle instance d'Arbre (un nouveau Noeud), dont la lettre et la fréquence sont la concaténation des 2 retirées auparavant.

Cette nouvelle instance d'Arbre ne sera pas une feuille, puisqu'elle aura comme enfants les 2 retirés précédemment, respectivement gauche et droit.

On ajoute ensuite ce nouveau noeud au tas, et on recommence l'itération jusqu'à ce que le tas ne fasse plus qu'un élément de longueur, l'élément résultant étant l'arbre complet d'Huffman.

Exercice 2

Comme demandé dans l'énoncé, pour générer le dictionnaire demandé, on effectue un parcours infixe de l'arbre, en commençant par le côté gauche.

La fonction de parcours (nommée `infix_traversal()` pour cohérence avec le nommage des autres fonctions) est une fonction récursive qui explore chaque branche jusqu'à ce que cette dernière soit une feuille, auquel cas elle attribue au caractère associé un code qui résulte du parcours effectué jusqu'à la branche, 0 = gauche et 1 = droite.

Exercice 3

L'encodage se fait donc en lisant le fichier source (en **UTF-8**),et en associant à chaque caractère, son code déduit avec le dictionnaire précédemment généré.

Il y a tout de même un traitement spécial lorsque l'on rencontre un caractère non reconnu. En effet, si le caractère n'est pas dans le dictionnaire, cela veut dire qu'il est NYT, on encode donc le chemin vers le NYT, mais on ajoute aussi le code du caractère sur 16 bits (pour prendre tout l'UTF-8) à l'encodage, cela permettra, au moment du décodage, de déduire que le caractère est un NYT, et ainsi de dire que les 16 prochains bits sont réservés au code du caractère, permettant son décodage.

Si le charcode du caractère ne fait pas exactement 16 bits, on ajoute des bit de padding jusqu'à ce que la longueur fasse 16. Pour ne pas affecter le décodage, nous avons choisi que les bits de padding seraient des 0 et qu'ils seraient ajoutés avant le charcode, et non après. Ainsi la valeur entière des 16 bits ne changera pas malgré le padding.

Nous avons également implémenté une fonction pour écrire directement le résultat dans un fichier. Le fichier résultant est le nom de l'ancien fichier, auquel on a retiré l'extension et ajouté "Encoded.bin". Pour écrire dans ce fichier binaire, on transforme notre chaîne binaire en bytearray, pour ce faire on transforme chaque chaîne d'octet en véritable octet (int(octet, 2)), puis on ajoute cet octet à notre bytearray.

Ensuite on ouvre le fichier cible en écriture binaire (bw), et on y écrit notre bytearray.

Concernant le calcul du taux de compression, il est effectué dans la partie tests unitaires

Exercice 4

Pour décoder, il faut d'abord lire depuis un fichier binaire. Pour cela, on effectue l'étape inverse de l'exercice 3, en lisant le contenu du fichier en mode byte (rb).

Ensuite, pour chaque octet, on le converti en chaîne de caractère, puis en ajoutant du padding si nécessaire pour que la chaîne fasse bien 8 caractères.

Puis, pour chaque bit de la chaîne de caractère, on va effectuer le parcours associé dans l'arbre.

On commence à la racine de l'arbre.

Si le bit est un 1, on va dans l'enfant droit, si c'est un 0, on va dans l'enfant gauche.

Si la branche actuelle est un enfant, alors on récupère la lettre associé et on retourne à la racine, puis on recommence avec le bit suivant.

Cas particulier si la "lettre" associée est NYT, alors on sait que les 16 prochains bits seront réservés au code du caractère inconnu. Ainsi, on récupère directement ces 16 prochains bits, et on les décode, puis on saute 16 itérations de la boucle pour ne pas prendre en compte ces bits lors du parcours de l'arbre.

Huffman sans fréquence

Concernant la version sans fréquence de l'implémentation de l'algorithme de Huffman, les grands principes restent les mêmes que pour la version avec fréquence, à quelques exceptions près :

- La fréquence d'apparition des caractères est calculée pour chaque texte que l'on veut encoder
- Il n'y a donc pas besoin d'implémenter de gestion de NYT
- En revanche, on n'a pas l'arbre à notre disposition pour décoder le texte, il faut donc trouver un moyen de l'insérer dans le fichier encodé avec une méthode pour le décoder

Dans notre implémentation, nous avons eu plusieurs choix à faire :

- Dans un premier temps : quelles données garder pour pouvoir décoder, tout en prenant un minimum de place pour l'encodage
  - Pour répondre à cette question, nous avons d'abord pensé à la librairie pickle, qui sérialise des données, mais pour gagner de la place, nous avons voulu nous-même sérialiser. Pour cela, nous avons hésité entre sérialiser un dictionnaire caractère -> code (parcours à faire dans huffman) ou bien le dictionnaire de fréquences.

    Nous avons finalement opté pour le dictionnaire de fréquences, car la taille nécessaire pour stocker la fréquence va évoluer moins vite. En effet, si l'on prend une valeur de 16 bits, on peut stocker une fréquence allant jusqu'à 65535, alors que l'on ne pourra stocker un code de parcours d'un arbre n'ayant que 16 niveaux. De plus, le choix du code de parcours aurait entraîné une difficulté supplémentaire au niveau du padding (si on met 0 en padding et que le parcours commence à 0, les 2 vont se confondre et on ne saura pas où commencer)

    Nous avons donc choisi de sérialiser le dictionnaire de fréquences, avec des fréquences sur 16 bits, et des caractères sur 16 bits (pour prendre tout l'UTF-8).

    Reste ensuite à déterminer le signal d'arrêt de lecture du dictionnaire, marquant la limite entre le dictionnaire et les données encodées. Pour cela, nous avons réservé les premiers 16 bits du fichier à la valeur de la taille du dictionnaire sérialisé. Ainsi, une fois la taille atteinte, on saura que les données lues appartiennent au message encodé et non au dictionnaire sérialisé.

Méthode de test unitaire

Afin de vérifier que les algorithmes d'huffman que nous avons employés fonctionnent correctement, nous avons réalisé dans le fichier *tests.py* une batterie de tests :

1. Taux de compression selon différentes tailles de fichiers.
   - Taux avec l'algorithme d'huffman sans fréquence.
   - Taux avec l'algorithme d'huffman avec fréquences.
2. Vérification de l'équivalence entre le contenu du fichier initial et le contenu décodé du fichier encodé avec chaque algorithme.

1. Taux de compression selon différentes tailles de fichiers

Afin de réaliser cette partie, nous avons créé les fichiers encodés à partir des fichiers initaux. Ensuite, on compare la taille du fichier initial avec celle du fichier compilé et on calcule le taux de compression. Voici nos résultats

- Taux avec l'algorithme d'huffman sans fréquence

  i. Pour un fichier de très petite taille, la compression n'est pas efficace, en effet, devoir enregistrer l'arbre et les données encodées n'est pas optimal et fait même perdre de la place comme on le voit avec le résultat ci-dessous :

  - Le fichier initial (*littleFile.txt*) a une taille de : 3 octets.
    Le fichier compressé a une taille de : 16 octets.
    Le taux de compression est de : -433%.

  ii. Pour un fichier de très grosse taille, la compression est très efficace, en effet, comme on le voit avec le résultat ci-dessous, on obtient un taux de compression de 46% :

  - Le fichier initial (*bigFile.txt*) a une taille de : 58812 octets.
    Le fichier compressé a une taille de : 32273 octets.
    Le taux de compression est de : 46%.

  iii. Pour le fichier témoin *leHorla.txt*, on obtient un bon taux de compression (19%) :

  - Le fichier initial (*leHorla.txt*) a une taille de : 571 octets.
    Le fichier compressé a une taille de : 460 octets.
    Le taux de compression est de : 19%.

- Taux avec l'algorithme d'huffman avec fréquences

  i. Pour un fichier de très petite taille, la compression n'a pas d'effet. On garde la même taille de fichier. Le taux de compression l'illustre bien en étant à 0% :

  - Le fichier initial (*littleFile.txt*) a une taille de : 3 octets.
    Le fichier compressé a une taille de : 3 octets.
    Le taux de compression est de : 0%.

  ii. Pour un fichier de grosse taille, la compression est efficace avec un taux de compression s'élevant à 32% :

  - Le fichier initial (*bigFile.txt*) a une taille de : 58812 octets.
    Le fichier compressé a une taille de : 40312 octets.
    Le taux de compression est de : 32%.

  iii. Pour le fichier témoin *leHorla.txt*, on obtient un bon taux de compression qui est de 19% :

  - Le fichier initial (*leHorla.txt*) a une taille de : 571 octets.
    Le fichier compressé a une taille de : 457 octets.
    Le taux de compression est de : 19%.

2. Vérification de l'équivalence entre le contenu du fichier initial et le contenu décodé du fichier encodé avec chaque algorithme

Pour chaque fichier, on va aller décompresser le fichier encodé (*nameEncoded.bin*) et récupérer le contenu ainsi décodé. On va comparer ce dernier avec le texte clair initial pour vérifier que les deux sont bien égaux.

- Vérification avec l'algorithme d'huffman sans fréquence

  i. Test avec le fichier *bigFile* :
  Le contenu du fichier encodé (une fois décodé) est bien le même que celui du fichier clair.

  ii. Test avec le fichier *littleFile* :
  Le contenu du fichier encodé (une fois décodé) est bien le même que celui du fichier clair.

  iii. Test avec le fichier témoin *leHorla* :
  Le contenu du fichier encodé (une fois décodé) est bien le même que celui du fichier clair.

- Vérification avec l'algorithme d'huffman avec fréquences

  i. Test avec le fichier *bigFile* :
  Le contenu du fichier encodé (une fois décodé) est bien le même que celui du fichier clair.

  ii. Test avec le fichier *littleFile* :
  Le contenu du fichier encodé (une fois décodé) est bien le même que celui du fichier clair.

  iii. Test avec le fichier témoin *leHorla* :
  Le contenu du fichier encodé (une fois décodé) est bien le même que celui du fichier clair.