Belhadj Walid

Master II Sicom

Travail pratique : Sécurité des cartes à puces

Sujet : codage Huffman (compression et décompression)

Explication et démarche :
Code huffman en un fichier **main.py**

Exécution : python main.py

Cette exécution retourne la compression et décompression du fichier **textfile.txt**

Résultat de sortie : un fichier binaire (**textfile.bin**) et un fichier **textfile_decompressed.txt** qui doit être comparé avec le fichier original le code est suffisamment commenté,

Blocks du code :

Bibliothèques :

```
import json # stocke le chemin vers le fichier source
from heapq import heappop, heappush # gestion de files
from bitstring import BitArray #conversion binaire/bytes/string
import os #pour s'adapter au OS
```

Ouverture des fichiers nécessaire :

```
# on ouvre et on charge le fichier à partir du fichier config.json
config = open('./config.json')
config_json = json.load(config)
# renvoie une liste contenant chaque ligne du fichier en tant qu'élément
# de liste
# utilise le paramètre d'encodage pour ouvrir le fichier et le lire
# dans n'importe quelle langue possible
file = open(config_json["filepath_text"], encoding="utf8").readlines()
# print(file)
```

Fonction de gestion de fréquence d'apparition des charactères :

```
def count_frequencies(file) -> dict:

    """
    Cette fonction ouvre un fichier texte en argument puis renvoie le
    fréquences chaque caractère apparaît dans le fichier texte. Les personnag
    sont ensuite triés par ordre en partant de la fréquence la plus basse jus
    le plus haut. Valeurs triées par ordre croissant.
    """
```

La sortie :

```
# stocker le dictionnaire des fréquences dans une variable
# appelée "d" pour une utilisation ultérieure
d = count_frequencies(file)
print(d)
```

Définition de l'arbre de Huffman :

```
def creation_of_huffmantree(d) -> list:

    """
    Cette fonction obtient le dictionnaire renvoyé par la fonction
    count_frequencies
     et renvoie une liste de listes qui représentent l'arbre de
     Huffman des personnages
     dans le fichier texte. Il est formaté de la manière suivante :
    [[character_frequency,[character,code]]
    Au début, le code sera une chaîne vide mais il aura du contenu plus tard.
    # affecte une variable aux listes de listes pour la représentation
    # de l'arbre de Huffman
    # utilise la méthode items() pour retourner un objet de vue
    # L'objet view contient les paires clé-valeur du dictionnaire,
    # sous forme de tuples dans une liste.
    huffman_tree = [[frequency, [char, ""]] for char, frequency in d.items()]
    # on utilise la boucle while pour créer un arbre huffman
    # tant que la longueur de la liste de l'arbre de Huffman est supérieure à
    while len(huffman_tree) > 1:···
    return huffman_list
print(creation_of_huffmantree(d))
```

Fonction d'encodage du textes (entré/sortie)

```
def encoded_texts(huffman_list: list, new_list: list) -> str:

    """
    Cette fonction obtient la liste renvoyée par la fonction
    creation_of_huffmantree
    et une nouvelle liste comme paramètre aussi. Cette fonction permet
    d'obtenir le code
    texte, il remplace les caractères par le contexte du fichier
    texte d'origine par
    codes attendus et renvoie donc un texte encodé.
    """
```

```
    encoded_text = string.translate(convert)
    # print(encoded_text)
    return dictionary, encoded_text
```

Fonction de remplissage de textes :

```
121   def padding_text(encoded_text: str):
122
123       """
124       Cette fonction obtient la chaîne encoded_text de la fonction
125       encoded_text ci-dessus et renvoie la version complétée du texte encodé.
126       Ajoute des caractères au format texte codé tel qu'il doit être affiché.
127       ajoute des bits à la chaîne encodée donc la longueur
128       est un multiple de 8 et peut donc être encodé efficacement.
129       """
```

Return :

Elle joint le texte rembourré et le texte encodé pour obtenir la version finale du texte encodé **encoded_text = padded_data + encoded_text**

return encoded_text

```
def compression():

    """
    Cette fonction est formatée pour être utilisée lors de la compression
     le fichier texte. Il utilise les valeurs renvoyées dans les
     fonctions pour obtenir un fichier correctement compressé :
     --> fonction count_frequencies
     --> fonction creation_of_huffmantree
     --> fonction encoded_text
     --> fonction padding_text
     Tout ce qui précède
    """
```

L'écriture se fait sur un fichier binaire **(.bin)**

```
return output_path
```

Fonction de suppression de bourrage :

```
def remove_padding(bit_string):

    """
    Cette fonction est utilisée dans la fonction de décompression pour
    décompresser le fichier texte. Sa fonctionnalité est de supprimer
    le remplissage ajouté à la fonction de remplissage de texte pour
    obtenir une décompression correcte du texte.
    Elle renvoie le texte encodé sans le remplissage.
    """

    padded_data = bit_string[:8]
    extra_padding = int(padded_data, 2)

    bit_string = bit_string[8:]
    encoded_text = bit_string[:-1*extra_padding]

    return encoded_text
```

Fonction de décompression :

```
def decompress(input_path):

    """
    Cette fonctionnalité de fonctions est de décompresser le déjà compressé
    fichier texte d'avant. Il utilise les valeurs renvoyées dans les
    fonctions pour obtenir un fichier correctement compressé :
    --> fonction creation_of_huffmantree
    --> fonction encoded_text
    --> fonction remove_padding
    Tout ce qui précède
    """
```

Fonction de décodage de bytes :

```python
def decode_text(encoded_text: str, reverse_mapping: dict):

    """
    Cette fonction renvoie le texte décodé du fichier texte.
    Utilise les boucles for et l'instruction if pour obtenir le texte décodé.
    """
```

Partie opérations :

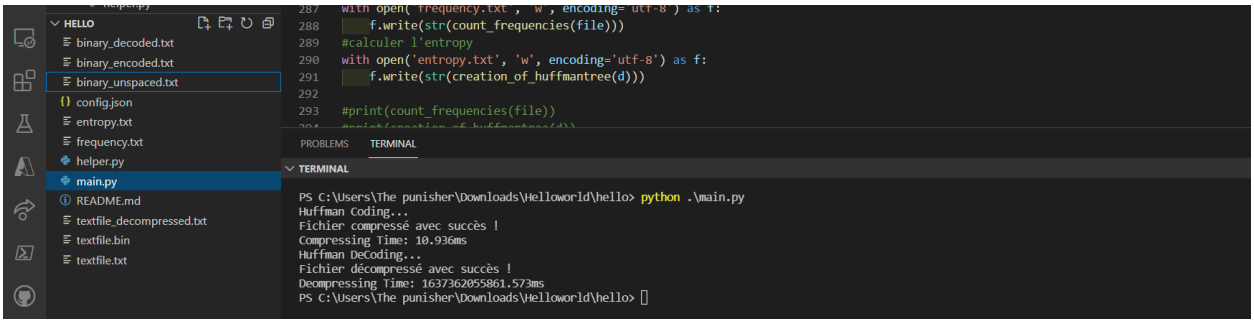
```python
print("Huffman Coding...")
timeofcompress = time.time()
# Compter l'apparition de chaque charactère
with open('frequency.txt', 'w', encoding='utf-8') as f:
    f.write(str(count_frequencies(file)))
#calculer l'entropy
with open('entropy.txt', 'w', encoding='utf-8') as f:
    f.write(str(creation_of_huffmantree(d)))

#print(count_frequencies(file))
#print(creation_of_huffmantree(d))
#stock binary text
with open('binary_encoded.txt', 'w', encoding='utf-8') as f:···
#print(encoded_texts(file, creation_of_huffmantree(d))[1])

with open('binary_decoded.txt', 'w', encoding='utf-8') as f:···

#print(padding_text(encoded_texts(file, creation_of_huffmantree(d))[1]))
with open('binary_unspaced.txt', 'w', encoding='utf-8') as f:···
print("Fichier compressé avec succès ! ")
timeofcompress = time.time()-timeofcompress
print('Compressing Time: ', end='')
print(str(round(timeofcompress * 1000, 3)) + 'ms')
#print(compression())
print("Huffman DeCoding...")
timeofdecompress = time.time()
decompress(compression())
#print(decompress(compression()))
print('Deompressing Time: ', end='')
print(str(round(timeofdecompress * 1000, 3)) + 'ms')
```

Résultat et fichiers crées :



Amélioration :

Comparaison des deux fichiers :

```
#comparaison
print("Phase de comparaison")
with open('textfile.txt', 'r') as file1:···

check=False
same.discard('\n')
if same != check:

    print("les fichiers sont identiques")
else:
    print("les fichiers ne sont pas identiques")

with open('save_difference.txt', 'w') as file_out:
    for line in same:
        file_out.write(line)
```

Enregistrements de la table de fréquences, binary files …

```
print("Huffman Coding...")
timeofcompress = time.time()
# Compter l'apparition de chaque charactère
with open('frequency.txt', 'w', encoding='utf-8') as f:
    f.write(str(count_frequencies(file)))
#calculer l'entropy
with open('entropy.txt', 'w', encoding='utf-8') as f:···
#stock binary text
with open('binary_encoded.txt', 'w', encoding='utf-8') as f:···
#print(encoded_texts(file, creation_of_huffmantree(d))[1])
with open('binary_decoded.txt', 'w', encoding='utf-8') as f:···
#print(padding_text(encoded_texts(file, creation_of_huffmantree(d))[1]))
with open('binary_unspaced.txt', 'w', encoding='utf-8') as f:···
```

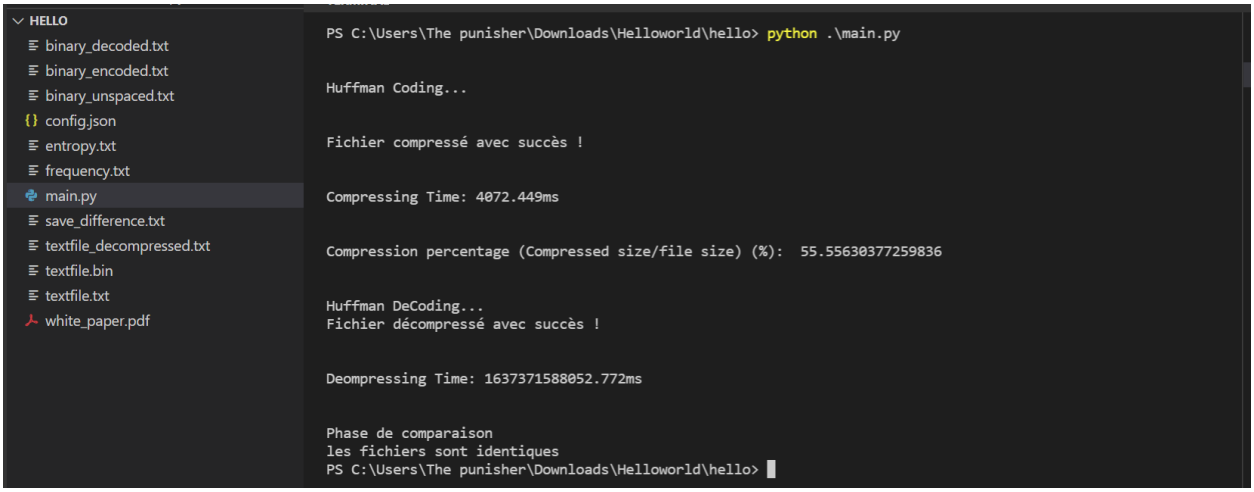
Compter le temps de compression et décompression

```
print('Compressing Time: ', end='')
print(str(round(timeofcompress * 1000, 3)) + 'ms')
inputsize=os.path.getsize("textfile.txt")
outputsize=os.path.getsize("textfile.bin")
compression_ratio=float(outputsize/inputsize)*100
```

```
print("Huffman DeCoding...")
timeofdecompress = time.time()
decompress(compression())
#print(decompress(compression()))
print("\n")
print('Deompressing Time: ', end='')
print(str(round(timeofdecompress * 1000, 3)) + 'ms')

```

Résultat final avec le fichier original


HELLO
binary_decoded.txt
binary_encoded.txt
binary_unspaced.txt
config.json
entropy.txt
frequency.txt
main.py
save_difference.txt
textfile_decompressed.txt
textfile.bin
textfile.txt
white_paper.pdf
PS C:\Users\The punisher\Downloads\Helloworld\hello> python .\main.py
Huffman Coding...
Fichier compressé avec succès !
Compressing Time: 4072.449ms
Compression percentage (Compressed size/file size) (%): 55.55630377259836
Huffman DeCoding...
Fichier décompressé avec succès !
Deompressing Time: 1637371588052.772ms
Phase de comparaison
les fichiers sont identiques
PS C:\Users\The punisher\Downloads\Helloworld\hello>