COMPRESSION DE DONNÉES PAR CODAGE DE HUFFMAN

PROJ631 - PROJET ALGORITHMIQUE

RIBES MAËL
IDU3 - 2022

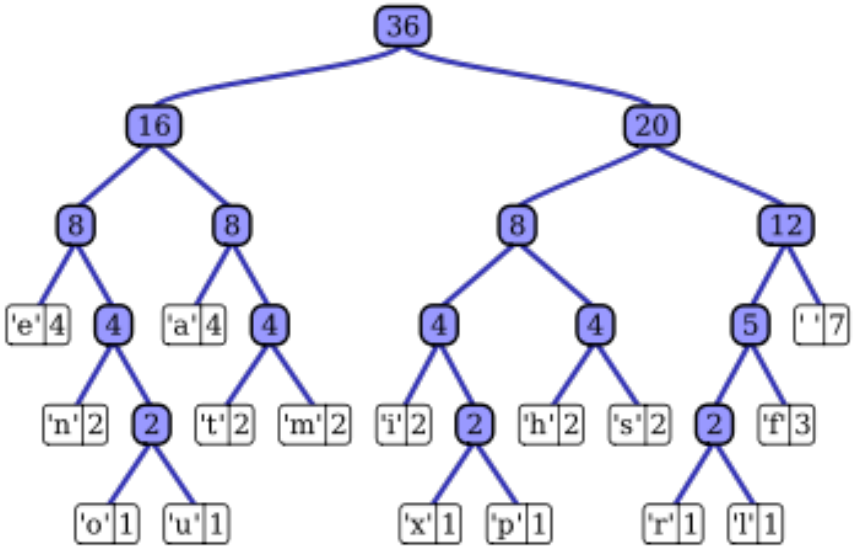
HTTPS://GITHUB.COM/MAELRIBES/PROJ631_HUFFMAN_CODING

INTRODUCTION

PRÉSENTATION DU PROJET

Le codage de Huffman est un algorithme de compression de données sans perte. Le codage utilise un code de longueur variable pour représenter un symbole source (par exemple, un caractère dans un fichier). Le code est déterminé à partir d'une estimation des probabilités d'occurrence des symboles sources, un code court étant associé aux symboles sources les plus fréquents.

Dans ce projet, j'utiliserais l'algorithme de Huffman pour créer un programme qui génère un ficher binaire compressé à partir d'un fichier texte.



Arbre de Huffman généré avec la phrase :
« *this is an example of a huffman tree* »

CHOIX DU LANGAGE

J'ai utilisé Python pour réaliser ce projet. Je n'ai eu à utiliser qu'une seule bibliothèque externe à Python : la bibliothèque os, simplement pour obtenir la taille des fichiers pour calculer le taux de compression.

ORGANISATION

J'ai commencé mon projet par créer un Trello pour déterminer l'ordre dans lequel je devais créer les différentes fonctionnalités. Cela m'a permis de séparer la création de mon code en plusieurs parties et de savoir par la suite quand créer les différentes branches Git.

STRUCTURE DU PROJET

Le projet est découpé en cinq fichiers : quatre de fonctions et un main. Chaque fichier gère une fonctionnalité et correspond plus ou moins à une question du sujet.

RÉCUPÉRATION DES FRÉQUENCES

La première étape pour utiliser l'algorithme de Huffman est de récupérer l'alphabet utilisé dans le texte à compresser et d'associer chaque caractère à son nombre d'occurence dans ledit texte.

```python
def get_frequencies(text):
    frequencies = {}
    for char in text:
        if char in frequencies:
            frequencies[char] += 1
        else:
            frequencies[char] = 1
    return alpha_sort(frequencies)
```

La fonction **get_frequencies**(*text*) retourne donc un dictionnaire contenant en clés les différents caractères du texte associés à leur nombre d'apparition. Cependant, pour créer un arbre de Huffman efficace, ce dictionnaire doit être d'abord trié par ordre croissant de fréquence puis par ordre alphabétique.

La fonction **alpha_sort**(*f*) se charge du tri du dictionnaire des fréquences. On utilise d'aborde la fonction native de python **sorted**() pour trier par ordre de fréquence puis on effectue un tri à bulle pour trier par ordre alphabétique les caractères ayant une fréquence égale.

```python
def alpha_sort(f):
    freq = sorted(f.items(), key=lambda x: x[1])
    for k in range(len(freq)):
        for index in range(len(freq) - 1):
            freq1 = freq[index]
            freq2 = freq[index + 1]
            if freq1[1] == freq2[1]:
                if ord(freq1[0]) > ord(freq2[0]):
                    freq[index], freq[index + 1] = freq[index + 1], freq[index]
    res = tuplelist_to_dict(freq)
    return res
```

Cependant **sorted**() ne retourne pas un dictionnaire mais une liste de tuples, ce qui n'est pas très pratique pour la suite du projet. On crée donc une fonction simple pour convertir notre liste en dictionnaire et on l'appelle à la fin de notre fonction de tri pour récupérer un dictionnaire.

```python
def tuplelist_to_dict(li):
    di = {}
    for tu in li:
        di[tu[0]] = tu[1]
    return di
```

CLASSE NOEUD

Maintenant que l'on peut obtenir le dictionnaire de fréquence, il nous reste à construire l'arbre qui lui est associé. On crée d'abord la classe **Node** permettant de créer les noeuds de l'arbre.

Les noeud ont cinq attributs : un caractère, une fréquence (probe), un fils gauche, un fils droit, et un code binaire : 0 si le noeud en question est un fils droit, 1 sinon).

```
class Node:
    def __init__(self, char, proba, left=None, right=None):
        self.char = char
        self.proba = proba
        self.left = left
        self.right = right
        self.code = None
```

On initialise le code à **None** car on ne sait pas à la création si un noeud sera à droite ou à gauche de son parent puisque l'on commence la création de l'arbre par les feuilles. On place également la valeur nulle par défaut dans les attributs gauche et droite si ceux-ci ne sont pas renseignés pour avoir un constructeur flexible qui nous permet d'instancier à la fois des feuilles et des noeuds complets.

```
def is_leaf(self):
    if(self.left is None)and(self.right is None):
        return True
    return False
```

La seule méthode de la classe sert simplement à vérifier si un noeud est une feuille ou non.

CRÉATION DE L'ARBRE

```
def create_list_leaf(dict):
    list_node = []
    for k in dict.keys():
        proba = dict[k]
        list_node.append(Node(k, proba))
    return list_node
```

La première étape de la création de l'arbre est la création de la liste de toutes les feuilles grâce au dictionnaire. Comme celui-ci est déjà classé correctement, la liste est elle aussi ordonnée.

```
def create_huffman_tree(dict_freq):
    nodes = create_list_leaf(dict_freq)
    while len(nodes) > 1:
        l_child = nodes[0]
        l_child.code = 0
        nodes.pop(0)
        r_child = nodes[0]
        r_child.code = 1
        nodes.pop(0)
        father = Node('\0', r_child.proba + l_child.proba,
                      l_child, r_child)
        insert_node(nodes, father)
    return nodes[0]
```

L'idée maintenant est de prendre les deux premiers noeuds de cette liste (ceux qui ont donc les plus petites fréquences) et de créer un nouveau noeud parent de ces deux fils. Ce nouveau noeud prend en attribut le caractère nul et la somme des fréquences de ses fils. On met ensuite à jour le

code des fils en fonction de leur position droite ou gauche. On peut ensuite supprimer les noeuds fils de la liste des noeuds et insérer le nouveau noeud à la bonne place. On réitère le processus jusqu'à ce qu'il ne reste plus qu'un noeud dans la liste. On retourne alors ce noeud qui correspond à la racine de l'arbre de Huffman.

```python
def insert_node(l_nodes, node):
    if len(l_nodes) == 0:
        l_nodes.append(node)
        return l_nodes
    for i in range(len(l_nodes)):
        if node.proba <= l_nodes[i].proba:
            l_nodes.insert(i, node)
            return l_nodes
        else:
            l_nodes.append(node)
            return l_nodes
```

Afin d'éviter de devoir re-trier la liste des noeuds à chaque création de noeud, j'ai préféré créer un fonction pour insérer directement au bon endroit le noeud pour alléger le code.

insert_node(*l_nodes, node*) insère donc le noeud au premier indice où la probabilité/fréquence du noeud est inférieure ou égale à celle du noeud suivant. Comme tous les nouveaux noeuds ont le caractère nul en attribut, la liste des noeuds est directement classée par ordre croissant de fréquence et par ordre alphabétique.

PARCOURS DE L'ARBRE

Maintenant que l'on a créé l'arbre associé au texte, il faut à présent parcourir en profondeur celui-ci afin de trouver le code binaire de remplacement associé à chaque caractère.

```python
def coding_char(node, p, cod):
    if node.is_leaf():
        str_path = ""
        for i in p:
            str_path += str(i)
        cod[node.char] = str_path
        return
    if node.left is not None:
        p.append(node.left.code)
        coding_char(node.left, p, cod)
        p.pop(len(p) - 1)
    if node.right is not None:
        p.append(node.right.code)
        coding_char(node.right, p, cod)
        p.pop(len(p) - 1)
```

coding_char(*node, p, cod*) est donc une fonction récursive qui prend en paramètre un noeud, un dictionnaire et une liste. Le dictionnaire servira à stocker le codage des différents caractères et la liste à mémoriser le code des noeuds déjà parcourus pour n'avoir à parcourir qu'une fois l'arbre pour obtenir tout les codages.

Si le fils gauche n'est pas nul, on ajoute son code (qui est normalement 1 si l'arbre est créé correctement) à la liste. Puis on rappelle la fonction sur le fils gauche. On applique le même processus pour les fils droits.

Finalement la condition d'arrêt ne se déclenche que si le noeud est une feuille. On récupère alors dans la liste la chaine de 0 et de 1 correspondant au codage du caractère de la feuille puis on l'ajoute au dictionnaire. Lors du retour de pile, l'instruction `p.pop(len(p) - 1)` permet de retirer le dernier élément de la liste du codage pour que celle-ci ne garde pas en mémoire les noeuds par lesquels on ne passe plus pour déterminer le codage.

```python
def get_coding(root):
    path = []
    coding = {}
    coding_char(root, path, coding)
    return coding
```

Cette fonction initialise simplement le dictionnaire de codage et la liste utilisés dans **coding_char**(*node, p, cod*) et appelle cette même fonction pour retourner les dictionnaire rempli.

```python
def huffman_coding(text, codage):
    compressed_text = ""
    for char in text:
        compressed_text += codage[char]
    return compressed_text
```

Il ne reste plus qu'à convertir chaque caractère du texte initial par son code pour obtenir le texte binaire compressé.

Cette dernière fonction permet de calculer le nombre moyen de bits utilisés pour coder un caractère du texte.

```python
def bits_avarage(coding, freq):
    sum = 0
    n = 0
    for char in freq:
        n += freq[char]
    for char in coding:
        sum += len(coding[char])*freq[char]
    return sum / n
```

ÉCRITURE DES FICHIERS DE RÉSULTAT

Cette dernière partie du code contient les deux fonctions qui permettent la création des fichiers de résultats.

```python
def bin_writer(file_name, text_bin):
    compressed_file = open('results/' + file_name + '_comp.bin', 'wb')
    compressed_file.write(text_bin)
    compressed_file.close()
```

bin_writer(*file_name, text_bin*) créée et remplie le fichier binaire avec le texte compressé. À noter que la ligne du main :

```python
text_bin = int(compressed_text, base=2).to_bytes((len(compressed_text) + 7) // 8, byteorder='big')
```

est nécessaire avant l'appel de la fonction pour transformer la chaine de caractère binaire en véritable texte binaire rangé en octets.

```python
def freq_writer(file_name, freq):
    frequencies_file = open('results/' + file_name + '_freq.txt', 'w')
    frequencies_file.write('Number of different characters : '+str(len(freq))+'\n\n')
    frequencies_file.write('Number of occurrences of each character : '+'\n')
    for char in freq:
        if char == '\n':
            frequencies_file.write("\\n " + str(freq[char]) + '\n')
        else:
            frequencies_file.write(char + " " + str(freq[char]) + '\n')
    frequencies_file.close()
```

freq_writer(*file_name, freq*) créée et remplie et le fichier des fréquences avec les caractères et leur fréquence ainsi que le nombre total de caractères.

MAIN ET RÉSULTATS

On construit finalement un fichier main qui fait appel successivement à toutes les fonctions que nous avons pu créer jusque là. Pour que la compression s'effectue correctement, le fichier cible doit être placé au préalable dans le dossier ***assets.*** Le résultats de la compression va directement dans le dossier ***results***.

```
import file_managment
import frequences
import huffman
from node import Node

print('================ Huffman compression ================\n')
file_name = input('Input file name : ')
file = open('assets/'+file_name+'.txt', 'r')
text = file.read()
file.close()

freq = frequences.get_frequencies(text)
root = huffman.create_huffman_tree(freq)
coding = huffman.get_coding(root)

compressed_text = huffman.huffman_coding(text, coding)
text_bin = int(compressed_text, base=2).to_bytes((len(compressed_text) + 7) // 8, byteorder='big')

file_managment.bin_writer(file_name, text_bin)
file_managment.freq_writer(file_name, freq)

initial_size = file_managment.size('assets/'+file_name+'.txt')
final_size = file_managment.size('results/' + file_name + '_comp.bin')
compression_rate = 1 - (final_size/initial_size)

bits_avarage = huffman.bits_avarage(coding, freq)

print('Compression rate : ', compression_rate)
print('Avarage bit per char : ', bits_avarage)
```

Finalement, on obtient les résultats ci-contre pour la compression du texte « alice ».

```
================ Huffman compression ================

Input file name : alice
Compression rate :  0.404401370250314
Avarage bit per char :  4.880557108316889

Process finished with exit code 0
```

CONCLUSION

Je suis satisfait du résultat final que j'ai pu obtenir avec ce projet. J'arrive à obtenir un taux de compression plutôt acceptable pour une méthode de Huffman. Je sais qu'il est possible d'obtenir un meilleur taux de compression mais en cherchant dans mon code, je n'ai pas trouvé où je pouvais encore optimiser la compression.

Contrairement au précédent projet que j'ai réalisé en PROJ631, je me suis beaucoup mieux organisé avec Git et Trello et j'ai pu économiser énormément de temps. Je pense aussi que le choix d'utiliser Python a aussi pu contribuer à la réalisation plus rapide de ce projet de par la plus grande liberté que le langage offre.