POLYTECH ANNECY-CHAMBÉRY

UNIVERSITÉ SAVOIE MONT BLANC

PROJ631 – Projet algorithmique

DÉCOMPRESSION DE DONNÉES PAR CODAGE DE HUFFMAN

FASKA Rachid

IDU3

2021 – 2022

Table des matières

Introduction

Pour ce second projet algorithmique, j'ai choisi le sujet de la décompression de données par la méthode de Huffman. Le langage de programmation choisi est Java.

Le codage de Huffman permet une compression de données sans perte. Chaque caractère du texte à compresser est remplacé par une suite de bits de longueur variable, plus un caractère est présent dans le texte, plus la suite de bits sera petite.

Dans ce deuxième projet, la compression a déjà été réalisée et le fichier binaire ainsi que le fichier de description de l'alphabet utilisé nous sont fournis. L'objectif est alors de générer un fichier du texte décompressé.

Partie 1 : Organisation du projet

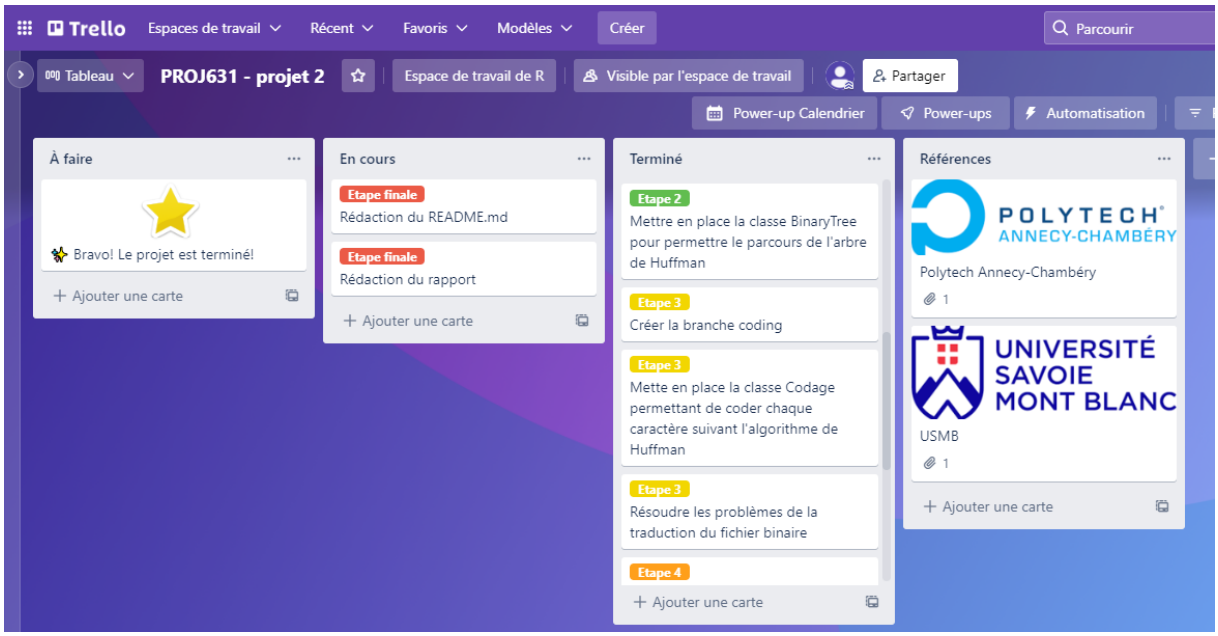
Pour répondre à la problématique, il faut décomposer le sujet en plusieurs étapes.

D'abord, il faut traduire le fichier de description de l'alphabet en HashMap pour faciliter la construction de l'arbre et son parcours.

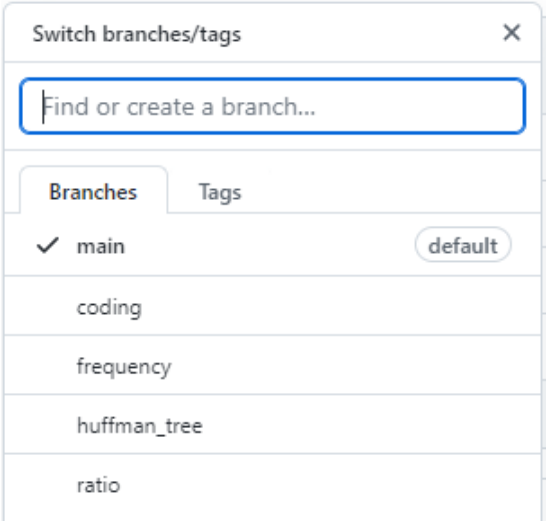
Après construction de l'arbre de Huffman, il faut déterminer la suite de bits de chaque caractère pour ensuite les faire correspondre avec le fichier binaire.

Enfin, on calcule le taux de compression et le nombre moyen de bits de stockage d'un caractère.

Pour faciliter l'organisation, j'ai donc créé un Trello pour bien répartir les tâches avec des étiquettes de couleur pour bien distinguer les grandes étapes du projet, ce qui m'a permis de bien suivre l'évolution du projet.

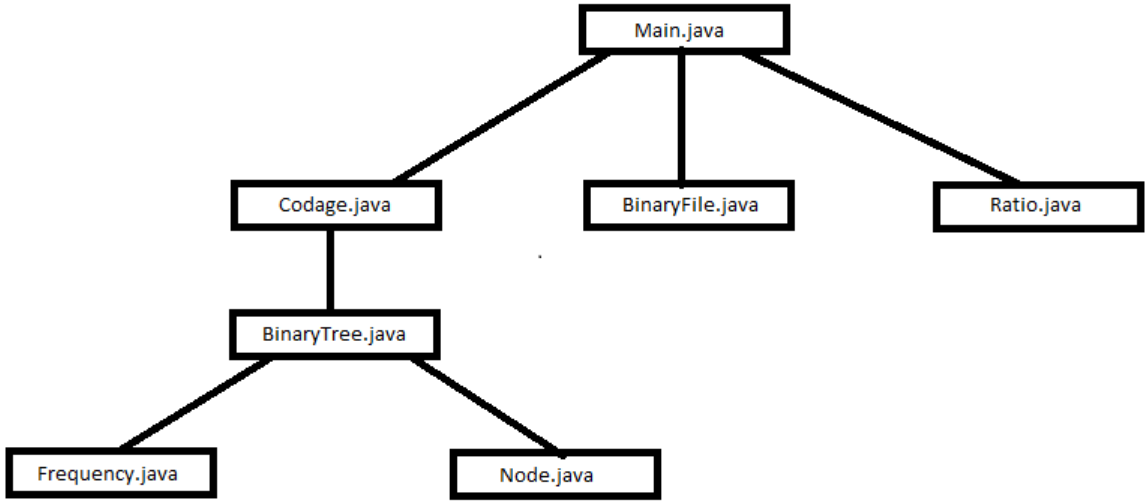


Par ailleurs, j'ai également fait un dépôt sur GitHub avec une branche pour chaque grande étape. Cela est très utile pour réduire les problèmes et les erreurs et avancer plus facilement au cours du projet.



La branche *frequency* correspond à l'étape 1, *huffman_tree* à l'étape 2, *coding* à l'étape 3 et la branche *ratio* contient les calculs de taux de compression et le nombre moyen de bits de stockage d'un caractère.

Décomposition fonctionnelle :



Pour créer la classe BinaryTree, il nous faut obligatoirement les classes Frequency et Node. On peut ensuite déterminer la classe Codage qui permet un parcours en profondeur de l'arbre et donne le codage de chaque caractère. La classe BinaryFile permet de traduire le fichier binaire et la classe Ratio permet le calcul du taux de compression ainsi que le nombre moyen de bits de stockage.

En exécutant le fichier Main.java, un fichier du texte décompressé est généré.

Partie 2 : Les étapes du projet

Étape 1 :

L'étape 1 est une étape primordiale, elle consiste à traduire le fichier contenant la description de l'alphabet en *HashMap<String, Integer>*.

Cela facilitera l'utilisation des caractères du texte avec leur fréquence pour la suite du projet.

Étape 2 :

Pour faciliter la construction de l'arbre j'ai créé une classe *Node*. D'abord on convertit le dictionnaire des fréquences *HashMap* en *ArrayList* pour que le tri selon les fréquences soit plus simple.

L'arbre est créé à l'aide de l'algorithme de Huffman :

1. On créé une liste de nœuds, *ArrayList<Node>*, avec chaque caractère et sa fréquence et on la trie à chaque itération selon les fréquences croissantes pour pouvoir récupérer plus facilement les 2 nœuds de fréquence minimale (indice 0 de la liste)
2. On créé un nœud parent avec comme fréquence la somme des fréquences des 2 nœuds et on l'ajoute à la liste que l'on retrie
3. On réitère jusqu'à ce qu'il ne reste plus qu'un nœud dans la liste : c'est le nœud racine

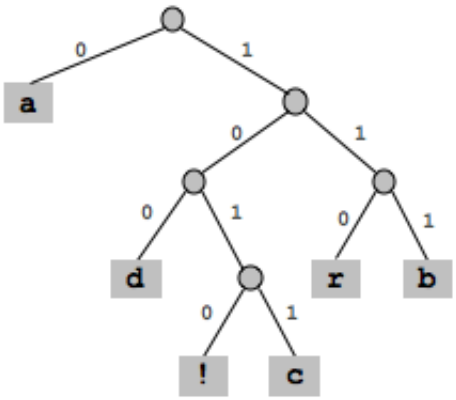
Pour cela, dans la classe *BinaryTree*, on fait une boucle *while* jusqu'à ce qu'il ne reste plus qu'un nœud dans la liste. Le premier nœud de la liste de nœuds correspondra donc au nœud racine de l'arbre de Huffman.

Étape 3 :

À partir du nœud racine on peut récupérer les fils gauches, droits et petit-fils, etc.… et donc, on va utiliser ce nœud racine pour faire un parcours en profondeur jusqu'à chaque feuille car les feuilles représentent les nœuds ayant pour label, le caractère.

Ce parcours en profondeur nous permet donc de définir le codage de chaque caractère.

Si on se déplace vers le fils gauche, on rajoute un "0" à la suite de bits du caractère et si on se déplace à droite, on rajoute un "1".



char	encoding
a	0
b	111
c	1011
d	100
r	110
!	1010

La fonction est alors récursive et va visiter chaque nœud de l'arbre.

Pour cette étape, j'ai utilisé une HashMap qui associe une suite de bits pour chaque caractère.

Par ailleurs, dans cette étape, j'ai également mis en place la classe *BinaryFile* permettant de traduire le fichier binaire en *String* représentant une suite de bits du texte en entier.

Étape 4 :

Cette étape est la dernière étape, il suffisait juste de calculer le taux de compression atteint pour ensuite déterminer le nombre moyen de bits de stockage d'un caractère.

La formule utilisée pour le calcul du taux de compression est la suivante :

$$\text{Taux de compression} = \text{Gain en volume} / \text{Volume initial} = 1 - \text{Volume final} / \text{Volume initial}$$

Le volume initiale correspond au nombre total de caractères dans le texte et le volume final correspond au nombre d'apparitions du caractère multiplié par la longueur de sa suite de bits et on additionne à chaque fois pour chaque caractère.

Il suffit ensuite de faire une fonction qui renvoie le taux de compression et une fonction qui renvoie le nombre moyen de bits de stockage :

```
public double compression_ratio() {
    double taux = 1 - (volume_finale() / volume_initial());
    return (double) Math.round(taux * 10000d) / 10000d;
}

public double nb_bits_moyen() {
    double sm = 0;
    int t = Frequency.taille(this.nom);
    Codage cd = new Codage(new BinaryTree(this.nom));
    HashMap<String, String> d = cd.dictCodage(cd.arbre.getRoot(), "");
    for (String cle : d.keySet()) {
        sm += d.get(cle).length();
    }
    double bit_moyen = sm / t;
    return (double) Math.round(bit_moyen * 10000d) / 10000d;
}
```

Étape finale :

Pour finir le projet, il faut mettre en commun les différentes branches sur GitHub et mettre au propre la classe Main pour que, lors de l'exécution de celle-ci, un fichier décompressé est généré.

Il faut également compléter le *README.md* pour expliquer aux utilisateurs le fonctionnement du répertoire et la manière de l'utiliser.

Partie 3 : Résultats obtenus

Pour cet exemple, le but est d'obtenir un fichier décompressé *exemple.txt* à partir des fichiers donnés : *exemple_freq.txt* et *exemple_comp.bin*.

Le fichier *exemple_freq.txt* contient la description de l'alphabet comme suit :

```
7
b 1
j 1
n 1
r 1
u 1
! 2
o 2
```

Et le fichier *exemple_comp.bin* représente le texte compressé :

Après traduction du fichier binaire et création de l'arbre de Huffman, en exécutant le fichier *Main.java*, on obtient le fichier décompressé *exemple.txt* suivant :

```
bonjour!!
```

Sur le terminal, le taux de compression atteint et le nombre moyen de bits de stockage sont affichés:

```
----BIENVENUE DANS LE DECOMPRESSEUR DE FICHIER ----

Quel texte voulez-vous décompresser ?
exemple

Taux de compression atteint: 0.5714
Nombre moyen de bits de stockage d?un caractère: 2.8571

Décompression terminée
```

Conclusion

Pour conclure, je trouve que ce projet était très intéressant car il est dans la continuité du premier projet algorithmique que j'ai réalisé.

Le projet s'est assez bien déroulé car j'avais acquis des connaissances sur la méthode de Huffman qui m'ont permis d'avancer plus efficacement au cours du projet. Mais la partie sur la traduction du fichier binaire était compliqué car il fallait transformer une succession d'octet en une collection de bits à valeur 0 ou 1.

Les apprentissages du premier semestre en Graphes et Langages m'ont également beaucoup apporté au cours de ce projet.

Liens Trello et GitHub

1. *Trello :* https://trello.com/invite/b/r0seFsTi/8deb5c2e2222d8001886a8c7c5896554/proj631-projet-2
2. *GitHub :* https://github.com/playerRC/Data_decompression_Huffman_coding