Rapport de travaux dirigés, Arbres de Huffman

TAMISIER Joshua - NADAUD Martin

January 9, 2026

---

1 Introduction

Dans ce TP, nous avons implémenté l'algorithme de Huffman dans le but d'encoder et de compresser des images ou du texte. Pour ce faire, nous avons en premier lieu implémenté l'algorithme de compréssion en python, puis nous l'avons adapté aux diverses utilisations possibles.

Contents

2 Algorithme de Huffman et codage source

L'algorithme de Huffman est un algorithme de **codage source**. L'objectif est de réduire la longueur moyenne des mots de code. Voici le fonctionnement de l'algorithme de Huffman étape par étape.
Ici, nous gardons l'implémentation la plus générale possible. On ne sait pas quel type de données on encode, mais l'algorithme fonctionne de la même manière pour tout type.

2.1 Arbre de Huffman

La première étape est de construire un arbre binaire tel que la profondeur d'un mot[1] dans l'arbre soit inversement correllée a sa fréquence d'apparition.
Nous définissons un arbre de manière récursive en créant une classe `Noeud` avec deux enfants (droite et gauche) ainsi qu'une étiquette et un indice. L'algorithme utilisé fonctionne comme suit :

```
Entrée : p une Liste de nombre d'occurences de chaque mot de code.
Sortie : Arbre binaire de Huffman
Algorithme :
arbres = liste_de_nombres_vers_liste_de_feuilles(p)
tant que arbres contient au moins 2 éléments
    trier(arbres)
    nouvel_arbre = Noeud(
      enfant_droit  : avant_dernier(arbres),
      enfant_gauche : dernier(arbres)
    )
    arbres = concatener( tous_sauf_n_derniers(arbres, 2), nouvel_arbre)

retourner premier_élément(arbres)
```

Cet algorithme nous donne bien un arbre binaire, dont les feuilles correspondent aux éléments de la liste de départ, et dont la profondeur des feuilles est inversement correllée à leur probabilité.

2.2 Nouveau code

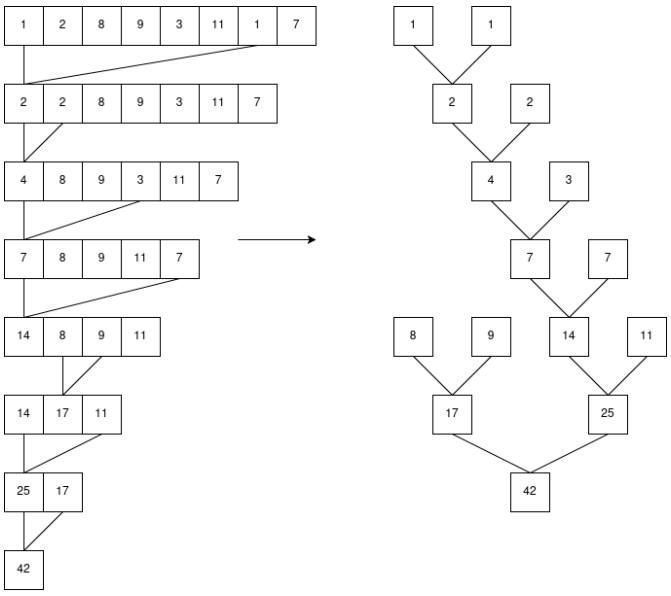
De cet arbre, on peut définir une table de conversion `mot` $\rightarrow$ `code` de la façon suivante:

- On crée une liste des mots encodés, on concatène 1 pour l'enfant de droite, 0 pour l'enfant de gauche.
- On associe chaque code à un mot source en fonction de sa fréquence. Mot très fréquent $\rightarrow$ Code court[2].

Grâce à cette table d'association, on peut alors encoder un à un les mots de la source, afin d'obtenir le message encodé.

[1] c'est-à-dire sa distance à la racine

[2] On remarque qu'il n'existe pas une seule et unique façon d'associer les mots et les codes. En effet, deux codes de même longueur peuvent être interchangés sans impacter la validité de l'algorithme.



Fonctionnement de l'algorithme de Huffman sur une liste d'entiers

2.3 Décodage

Une fois encodées, on peut décoder les données en utilisant le même arbre. Les codes de Huffman étant préfixes, le décode peut se faire de manière "gloutonne". Le décodage se fait en prenant chaque bit l'un après l'autre, en descendant l'arbre jusqu'à arriver à une feuille. Par construction, on a la certitude que cette feuille correspond bien au mot encodé.
Ainsi, notre algorithme de décodage fonctionne ainsi :

```
Entrée : seq une séquence de bits représentant les mots codés,
         tree l'arbre utilisé pour les encoder.
Sortie : une séquence de mots décodés
Algorithme :
offset := 0
noeud_curseur := tree
décodé := []
tant que seq n'est pas vide:
  si est_feuille(noeud_curseur) :
    seq = enlever_n_premier_bits(seq, longueure)
    décodé = concatener(décodé, mot(noeud_curseur))
    longueur = 0
    noeud_curseur = tree
  sinon
    si seq[0] = 0 :
      noeud_curseur = noeud_curseur.enfant_gauche
    sinon :
      noeud_curseur = neoud_curseur.enfant_droit
    longueur = longueur + 1

retourner décodé
```

3 Application

3.1 Compression d'images

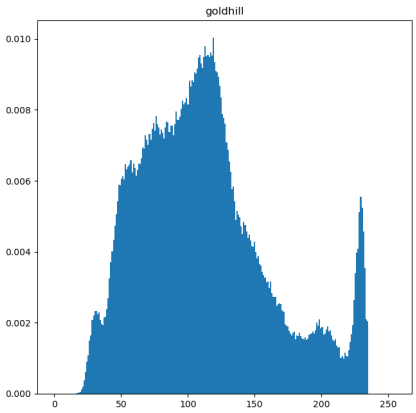
3.1.1 Principe

On se propose d'appliquer l'algorithme de Huffman en compressant des images en niveaux de gris, qui seront représentées, à l'aide de la bibliothèque Image de Python, par des tableaux d'entiers compris entre 0 et 255.
On définit donc une fonction `encoder_decoder(image)` qui encode puis décode une image avec les fonctions définies précedemment, teste si l'image décodée est égale à l'originale, puis affiche à l'écran les valeurs de l'entropie empirique, de la longueur moyenne et du ratio de compression.
Pour une image donnée, on commence par afficher l'histogramme des valeurs prises, que l'on utilisera ensuite comme tableau de probabilités. On génère aussi un tableau d'entiers de 0 à 255 comme tableau des symboles. Pour respecter la précondition des fonctions de `huffman`, on doit trier préalablement le tableau des probabilités par ordre décroissant, en triant dans l'ordre correspondant le tableau de symboles.
Pour encoder l'image on joint le tableau des "mots" encodés en binaire en une seule chaîne de caractères, et pour la décoder, on utilise la fonction `decode` et la liste des mots de code correspondants aux symboles, générée grâce à la fonction `huffman_code`. On affiche enfin à l'écran les valeurs demandées.

3.1.2 Expérimentation

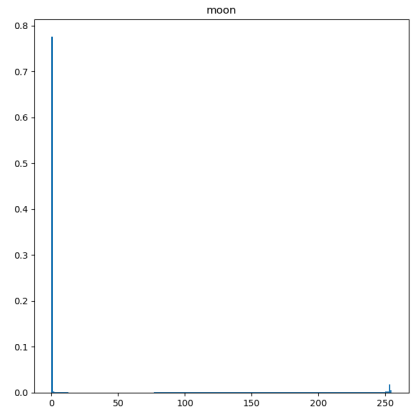
Ci-après figurent les images fournies ainsi que leurs histogrammes respectifs:

(a) `goldhill.png`



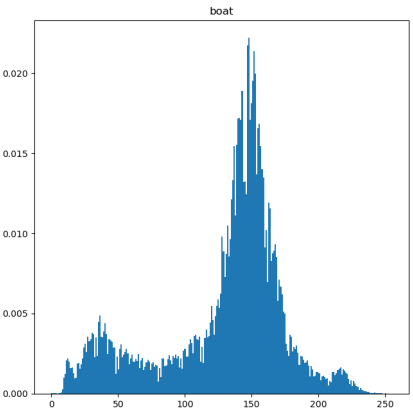
(b) Histogramme de `goldhill.png`

(a) `moon.png`



(b) Histogramme de `moon.png`

(a) `boat.png`



(b) Histogramme de `boat.png`

Les valeurs expérimentales:

```
#goldhill.png

Entropie empirique: 7.472173225084263
Longueur moyenne du code: 7.526447610294115
Ratio de compression: 0.9371309280395508

============================================

#moon.png

Entropie empirique: 2.465213838156748
Longueur moyenne du code: 2.714325138504536
Ratio de compression: 0.3379652882415313
```

```
==========================================

#boat.png

Entropie empirique: 7.185723654928107
Longueur moyenne du code: 7.246993719362746
Ratio de compression: 0.9023356437683105
```

On remarque deux choses:

- Les longueurs moyennes sont bien comprises entre H et H+1, donc le code respecte bien le théorème de Shannon.
- Le ratio de compression et la longueur moyenne associés à `moon.png` sont largement inférieurs à ceux des deux autres images, ce qui s'accorde avec le résultat attendu, car comme on peut le voir sur l'image et sur son histogramme, elle est composée principalement de pixels noirs, donc l'entropie de l'image est moindre (elle porte peu d'information) et il est possible de la compresser davantage.

3.2 Compression de texte

3.2.1 Principe

Pour la compression de texte, le principe général est très similaire, à quelques différences près. Au lieu d'utiliser comme alphabet source les entiers de 0 a 255, on utilise ici la table ascii (donc des entiers de 0 a 127) réduite aux caractères presents dans le texte.
Aussi, en lieu d'histogramme, on compte le nombre d'occurences de chaque caractère manuellement pour en obtenir la fréquence d'apparition. Ce sont ensuite les mêmes fonctions qui permettent d'encoder et de décoder le texte.
On définit donc une fonction `encoder_décoder_texte()` qui comme son nom l'indique, encode puis décode un texte, en comparant que le texte décodé est égal à l'entrée. Cette fonction retourne aussi l'entropie empirique du texte donné, la longueur moyenne des mots encodés et le taux de compression.

3.2.2 Expérimentation

Voici les résultats de la compression des textes :

texte source	entropie	longueur moyenne	ratio de compression
buscon	4.41	4.45	0.55
candide	4.57	4.59	0.57
dorian	4.48	4.52	0.56
clair de lune	4.50	4.53	0.56

On remarque que la taille finale des textes est environ réduite de moitié. On peut attribuer cela à la loi de Zipf qui stipule que dans une langue naturelle, le caractère le plus présent est environ deux fois plus commun que le second plus présent (et ainsi de suite). On peut alors supposer que quelques caractères sont représentés par des mots très courts, mais que ceux-ci représentent une grosse partie du texte source.

4 Pour aller plus loin

La question est, peut-on améliorer cet algorithme? On sait que les codes de Huffman sont optimaux, et donc le seul moyen d'améliorer le ratio de compression serait de réduire l'entropie de la source. Voici donc une proposition d'algorithme pour réduire l'entropie de la source avant de la compresser par Huffman.

4.1 Proposition d'algorithme

L'idée de cet algorithme consiste a remplacer la valeur de chaque pixel par la différence entre lui et le pixel précédent. C'est ce qu'on appelle du `Delta Encoding`. Voici un exemple de code C pour encoder un tableau ainsi :

```
void encode(static char* input, int input_len, char* output)
{
  //On suppose que output est initialisé a 0
  output[0] = input[0];
  for(int i = 1; i < input_len; i++)
  {
    output[i] = input[i] - input[i-1];
  }
}
```

Si une image n'est pas aléatoire, c'est-à-dire qu'elle contient des patterns (comme par exemple une photographie), on peut s'attendre à des différences similaires à plusieurs endroits de la photo. Par exemple, avec cet encodage préalable, tous les aplats de couleurs seront représentés par des 0. On peut donc s'attendre à une entropie réduite, et donc à un code de Huffman plus efficace. Cette méthode est notamment utilisée dans les images aux format `png` [3].

[3] https://www.w3.org/TR/png-3/#7Filtering