PHOUPHETLINTHONG Settha - TD2
MARCHAL Clément - TD3

Projet Huffman

binTree.py

C'est un module python que nous avons développé pour ce projet, il contient :
- La class « `Node` », pour la structure de chaque nœud d'un arbre.
- La class « `BinaryTree` », pour un les arbres binaires.

Plusieurs fonctionnalités on été implémentées (dans `BinaryTree`) :
- insert : insertion d'une valeur (créé un objet nœud).
- delete : supprime le dernier nœud et le remplace par le plus profond dans l'arbre (s'il existe).
- delete_all : supprime tous les nœuds de l'arbre.
- depth_first et width_first : parcours en profondeur (récursif) et parcours en largeur à partir d'un nœud.
- display : afficher les nœuds avec un parcours en largeur (par défaut) ou avec un parcours en profondeur.
- get_coded_value : donne le code de Huffman pour un nœud donné
- size : la taille de l'arbre.

Nous avons aussi implémenté la fonction « merge_trees » (pour fusionner 2 arbres), mais n'est pas dans une class.

Huffman.py

Ce module implémente uniquement l'arbre de Huffman, avec la class « `HuffmanTree` », qui hérite de la class binTree.BinaryTree.

Voici les fonctions qui on été implémenté dans cette class :
- get_frequency : donne la frequence de chaque lettre dans un message, retourne un dictionnaire
- get_frequency_node : retourne la fréquence pour un nœud donné, ou la valeur du nœud si celui-ci n'est pas une feuille (à ne pas confondre avec la fonction précédente)
- get_characteur : même chose que get_frequency_node, mais pour le caractère (une des lettres du message).
- sort_list_nodes : trie une liste de nœud en fonction de leur fréquence
- merge_nodes : crée un nœud parents dont les les fils sont soit une feuille soit un sous-arbre.
- create_tree : crée l'arbre de Huffman, en fusionnant les deux ayant les plus petites valeurs.
- encode : retourne le message encodé et sa clé dans un tuple.
- decode : retourne le message décodé.

Interface utilisateur (UI), dans les grandes lignes

Le but est de pouvoir utiliser les fonctions des modules précédents à l'aide d'une interface graphique (GUI), nous l'avons fait avec le module Tkinter, ceci est séparé en 2 fichiers.

Des boutons permettent d'activer les fonctionnalités, tel quel l'ajout de nœud(s), encodage, de codage (...). Plus ou moins tous les boutons ouvrent une autre fenêtre, pour les saisis de texte ou valeurs. La fenêtre principale supprime tout ce qu'il y dans le tkinter.Canvas principal et redessine l'arbre à chaque actualisation. Il est possible de cliquer sur plusieurs boutons à la fois, nous avons essayé de minimiser les erreurs que cela pouvais causer.

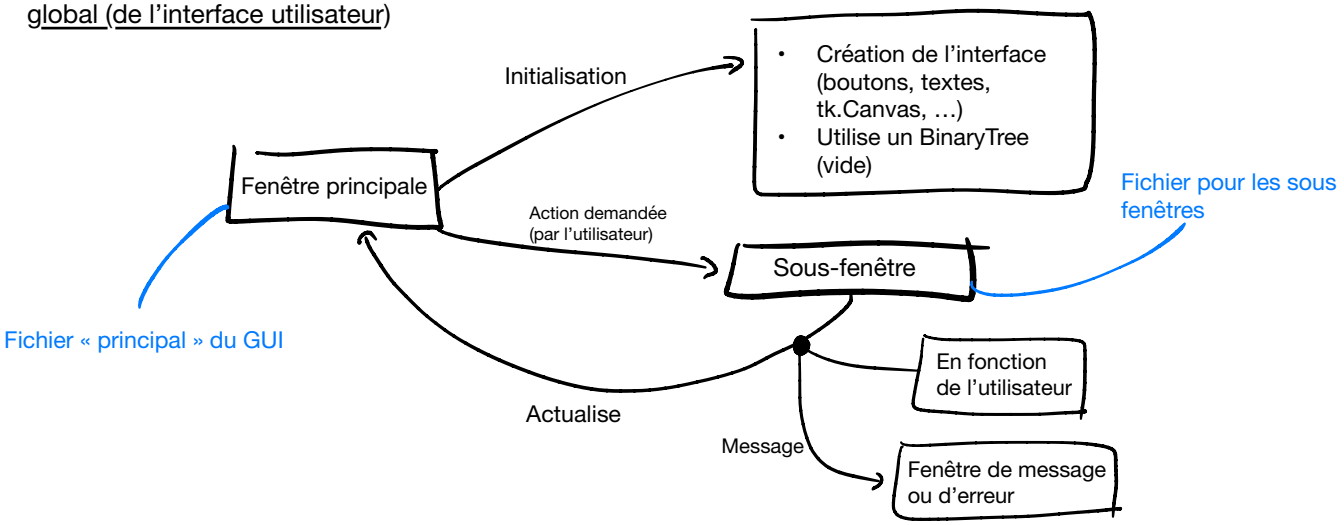
Une image vaut mieux que mille mots...

Le programme à pour objectif de crypter ou décrypter des données en utilisant la méthode Huffman. Le code de Huffman est une méthode de compression de données qui permet après compression de données, de représenter des données avec moins de bits en utilisant les occurrences présentes dans ces données.

Oui, il est possible d'ouvrir beaucoup de fenêtres en même temps, et nous n'avons pas repérer de bug à ce sujet, mais l'utilisateur doit faire attention à ce qu'il fait (s'il a oublié de sauvegarder l'arbre par exemple, etc.)
Il y a aussi une latérale et horizontale permettant de mieux visualiser l'arbre s'il dépasse la taille de la fenêtre (du Canvas pour être précis).
Notez que la taille de la fenêtre est une taille fixe, donc nous avons fixé sa taille à environ 1300x800 pixels (si vous avez un écran trop petit, c'est à ligne 28 dans le fichier window_main.py pour le modifier).

Place aux explications

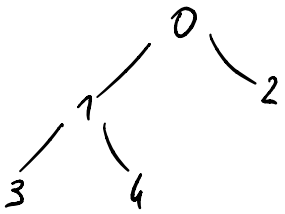
Schéma expliquant le fonctionnement global (de l'interface utilisateur)



Insertion et parcours en largeur

Parcours ne largeur : 0, 1, 2, 3, 4

Insérer ces valeurs dans le même ordre nous donne cette arbre :



Autrement dit :
Nous initialisons une liste, avec comme seul élément, la racine : queue = [Node]
Tant que le premier noeud de la liste a deux fils, nous l'ajoutons à la liste, autrement nous insérons un nouveau avec une certaine valeur souhaitée dans l'arbre, ce sera le fils droit au gauche du dernier noeud (ou plutôt l'ancien dernier noeud).

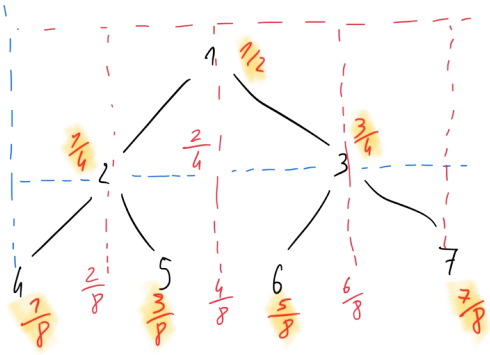
Suppression d'un noeud

Pour supprimer un noeud, nous le remplaçons par le noeud le plus profond de l'arbre, il s'avère que c'est le dernier noeud dans le parcours en largeur…

Lorsque certaines fonctions avaient besoins d'utiliser d'autres fonctions récursives, dans ce but, nous avions pu écrire des fonctions à l'intérieur de celles-ci (qui, bien souvent, étaient elles-mêmes dans une class), voici deux exemples :

Afficher un arbre

Pour afficher un arbre correctement, il était implicite de faire une fonction récursive, or nous avions besoin de quelques paramètres pour initialiser cette fonction (la hauteur de l'arbre, le rayon des cercles, …). Ainsi, « walk_tree_rec » (se trouvant dans « draw_tree »), permet de parcourir de l'arbre (parcours en profondeur) tout en dessinant les noeuds.
Plus en détail, à droite, vous pouvez voir un schéma dont les valeurs surlignées sont les proportion auxquels doivent être placés les noeuds, cela concerne uniquement l'axe des abscisses. Le dénominateur correspond à 2^h (2 exposant h), où h est la hauteur actuelle du noeud, en revanche le numérateur est un peu plus dur à avoir : lorsqu'il s'agit du fils gauche, il faut le multiplier par 2 et soustraire 1, mais quand il s'agit du fils droit il faut le multiplier par 2 et ajouter 1. Ici la récursivité est donc bien pratique.



Le cas de « get_coded_value »

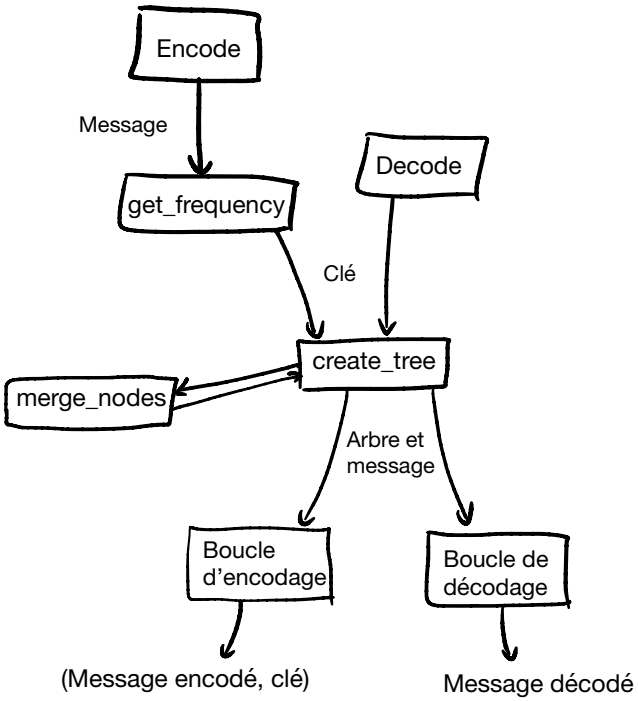
Cette fonction se trouvant dans la class BinaryTree permet d'obtenir le fameux code de Huffman pour un noeud, il s'avère que cette fonction utilise aussi un parcours en profondeur modifié. Chaque appel récursif ajoute un 0 ou un 1 à une chaîne de caractère en fonction du fils vers lequel en effectue cette recursion.

On se demande bien pourquoi on a écrit cette fonction, n'est-ce pas ? …

Le code de Huffman

La class « HuffmanTree » hérite de la class « BinaryTree », cela évite certaines redondances dans le code. Nous avons gardé la même structure pour les noeuds, ce qui nous a contraint à donner donner des types à nos sommets, dans un arbre de Huffman, les feuilles seront donc des chaînes de caractères tel que "t:12" veut dire qu'il y a 12 « t » dans le texte à encoder (ou décoder), les autres seront des entiers.

Schéma du fonctionnement

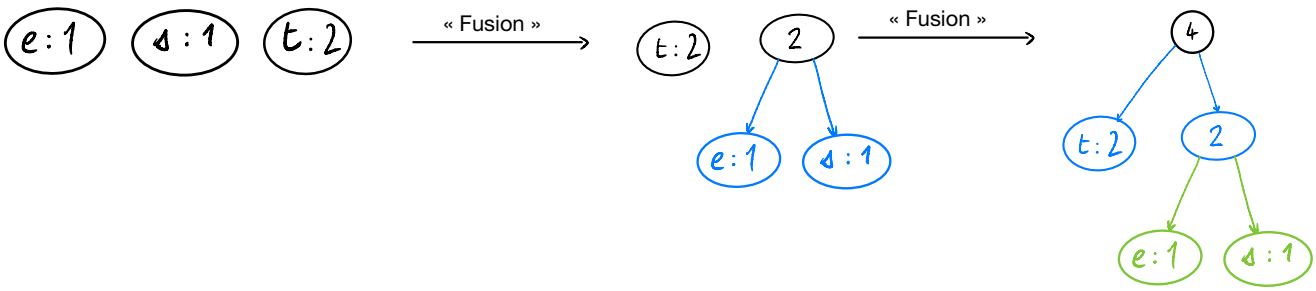


Explications

Pour encorder un message, on obtenir la fréquence de chaque lettre, on obtient un dictionnaire (dict()) qui est la clé, elle-même secret à la création de l'arbre, une fois cette arbre obtenu, nous pouvons enfin encoder le message. La fonction de décodage est similaire, elle prend pour arguments la clé (donc le dictionnaire) et bien sûr le message a décoder, le décodage est donc légèrement plus rapide que l'encodage.
« create_tree » appelle la fonction « merge_nodes » jusqu'à obtenir un arbre, cela fonctionne de la manière suivante :
Soit une liste noeud non relié, appelons la l_nodes, talent que cette liste contient plus d'un seul élément, nous fusionnons les deux noeuds ayant la plus petite valeur pour former petit arbre, on rajoute la racine de celui-ci dans l_nodes, ainsi de suite. (Voir le dernier schéma *).
Afin de d'encoder un message, nous bouclons sur la fonction « get_coded_value », vue précédemment.
Pour décoder un message, c'est similaire, cette fois on suit les branches de l'arbre que le code nous indique, jusqu'à tomber sur une feuille de l'arbre, puis on boucle jusqu'à la fin de la suite de 0 et de 1.

* Schéma de la « fusion » de noeuds, prenons comme exemple le mot « test »

Notre liste est donc composée de ces noeuds :



À la dernière itération, il ne reste plus qu'un noeud, donc l'arbre est retourné, afficher, ainsi il est possible de crypter ou décrypter un message.