Cyriac THIBAUDEAU
Aymeric GOUDOUT

Compte rendu de projet
IN407
Code d'Huffman

Table des matières:

I.

A.

L'objectif de notre projet consiste à implémenter le codage d'Huffman en python, pour nous permettre:
- D'afficher un arbre binaire résultant du codage d'un texte à l'aide de cet algorithme
- De prendre un texte en clair, en langue française, anglaise ou autre et de restituer un texte selon le code d'Huffman (cryptage)
- De prendre un texte codé selon le code de Huffman et de restituer le texte en clair correspondant (décryptage)

B.

Il existe plusieurs algorithmes de compression de données sans perte tel que le RLE (run-length encoding) et le codage d'Huffman. Ce dernier est un algorithme inventé par David Albert Huffman et publié en 1952. On parle d'un algorithme de compression de données sans perte car toute donnée encodée sera restituée. Cet algorithme permet donc de réduire le nombre de bit nécessaire au codage de symboles à partir d'un texte source. Chaque symbole obtient son codage associé par le nombre d'occurrences de celui-ci dans le texte source. Plus le codage sera court, plus le symbole est fréquent dans le texte. La structure des données est représentée sous forme d'arbre binaire dont chaque codage est constitué de 0 et/ou de 1. Chaque feuille de l'arbre binaire instancié à partir d'un texte correspond à un symbole contenu dans celui-ci avec un poids correspondant au pourcentage d'occurrence calculé. Tout autre sommet est le père de deux feuilles, d'une feuille et d'un sommet ou de deux sommets, avec comme donnée le poids de ses deux fils pour que la racine de l'arbre ai un poids égal à 1.

II.

A.

Notre logiciel est séparé en trois scripts python différents. Le premier, toolbox.py, contient toutes nos fonctions et ce que l'on pourrait considérer comme la partie « backend » du programme. Le second, interface.py, est la partie « frontend » du programme. Elle contient, comme son nom l'indique, tout ce qui se rapporte à l'interface graphique de notre application. Le dernier, main.py, est le fichier à exécuter pour lancer l'application.

B.

Durant tout ce projet, nous utiliserons la variante dite semi-adaptative comme méthode de construction de l'arbre. Il en existe deux autres ; statique et adaptatif. Le texte source est lu pour calculer chaque occurrence des symboles puis l'arbre est construit à l'aide de ces statistiques. Cet arbre est donc unique pour chaque texte source, et sera nécessaire lors du décryptage du texte. Son gain de compression est considéré comme plus performant ou égal à la variante statique, mais moins optimal que la variante adaptative.

C.

La première classe Sommet représente un sommet/nœud d'un arbre binaire. Elle possède deux attributs : poids et charactere. Le poids représente la fréquence d'apparition du caractère (ou du sous-arbre représenté par le nœud) dans un texte ou une chaîne de caractères. Le caractère est de type string (ou None si le sommet représente un sous-arbre plutôt qu'un caractère).
La classe possède plusieurs méthodes qui permettent de manipuler les attributs de l'objet Sommet.

- __init__(): le constructeur de la classe Sommet prend en argument le poids et le caractère du sommet. Si aucun caractère n'est spécifié, le caractère sera considéré comme None.

- set_poids(): cette méthode permet de modifier le poids du sommet avec la valeur new_poids,

- get_poids(): cette méthode permet de récupérer la valeur du poids du sommet,

- set_charactere(): cette méthode permet de modifier le caractère du sommet avec la nouvelle valeur charactere,

- get_charactere(): cette méthode permet de récupérer le caractère du sommet.

La seconde classe ArbreB_Huffman est une classe construite autour de la propriété de l'algorithme de Huffman, qui consiste à construire un arbre binaire de poids minimum à partir d'un ensemble de caractères avec leurs fréquences.
Chaque nœud de l'arbre binaire représente un caractère, et la profondeur du nœud correspond au nombre de bits nécessaire pour encoder ce caractère. Les caractères les plus fréquents sont placés à une profondeur inférieure, tandis que les caractères les moins fréquents sont placés à une profondeur supérieure. L'arbre est construit en fusionnant les arbres de poids minimum à chaque étape, jusqu'à ce qu'un arbre unique soit obtenu.
Voici les différentes méthodes de la classe :

- __init__(): est la méthode d'initialisation de la classe. Elle prend un seul argument sommet, qui est un objet de type Sommet et crée un arbre avec un seul nœud, qui est la racine de l'arbre.

- build_from_text(): prend un texte en entrée, utilise la fonction proportions(text) pour créer un dictionnaire des fréquences des caractères dans le texte, puis crée un arbre de Huffman en utilisant la méthode build_from_dico(dico) et renvoie cet arbre.
- get_poids(): renvoie le poids de l'arbre, qui est le poids du nœud racine.

- build_from_sommets(): prend une liste de sommets distincts en entrée et construit un arbre binaire de Huffman en fusionnant les arbres de Huffman correspondant à chaque sommet. Cette méthode renvoie l'arbre binaire de Huffman résultant.

- build_from_dico(): prend un dictionnaire de proportions des caractères en entrée et construit un arbre binaire de Huffman en utilisant la méthode build_from_sommets(liste:list[Sommet]). Cette méthode renvoie l'arbre binaire de Huffman résultant.

- decomposition(): décompose l'arbre en deux de manière récursive, de sorte à récupérer l'arbre correspondant au contenu du fils gauche ainsi que l'arbre correspondant au contenu du fils droit.

- fusion(): prend un autre arbre binaire de Huffman en entrée et fusionne les deux arbres en un arbre binaire de Huffman résultant.

- __add__(): surcharge d'opérateur qui prend un autre arbre binaire de Huffman en entrée et renvoie un nouvel arbre binaire de Huffman résultant de la fusion des deux arbres.

- show(): affiche le contenu de l'arbre dans le terminal de manière récursive.

- draw(): dessine l'arbre sur un canevas Tkinter de manière récursive.

- search(): recherche récursive d'un élément dans l'arbre et renvoie son équivalent binaire selon le code de Huffman.

- get_encode_dict(): retourne un dictionnaire de conversion.

- get_profondeur(): algorithme récursif pour connaître la profondeur d'un arbre.

- get_largeur(): algorithme récursif pour connaître la largeur d'un arbre.

- supp_chr(): supprime un sommet choisi et retourne le nouvel arbre reconstruit.

- __isub__(): surcharge de l'opérateur -=

- __iadd__(): surcharge de l'opérateur +=

- add_sommet(): ajoute un sommet et retourne le nouvel arbre reconstruit.

- __str__(): surcharge de la fonction print() pour un afficher le contenu de l'arbre

D.

D'autres fonctions ont aussi été mises en place:

- proportions(): prends un texte en argument et retourne un dictionnaire clé : caractère, valeur : occurrence
- encoding(): prend un texte en argument et l'encode selon un dictionnaire de conversion
- decoding(): prend un texte en argument et le décode selon un dictionnaire de conversion
- somme_offsets(): permet de calculer un offset utilisé lors du dessin de l'arbre
- get_dico_from_huffman_save(): analyse un fichier .txt comme un dictionnaire de conversion et le retourne

- abr_path(): renvoie les chemins des sommets d'un objet de la classe ArbreB_Huffman à l'aide de get_encode_dict()

- file_dialog(): permet de lire ou d'écrire sur un fichier

- fletcher16(): prend un dictionnaire de proportions de caractères en entrée et renvoi un checksum en binaire

- pourcentage_compression(): prend en argument le texte original ainsi que son équivalent binaire compressé pour renvoyer le pourcentage de compression atteint.

III.

A.

Le calcul des fréquences d'apparitions des caractères se fait aisément à partir de la fonction "proportions()" : celle-ci prend en argument un texte quelconque puis renvoie un dictionnaire de la forme dictionnaire[caractère] = nombre d'occurrences dans le texte.
on stocke également un dictionnaire de cette même forme à l'intérieur de la classe ArbreB_Huffman dont les getters et setters ne sont pas définis car cette variable n'est pas utilisée directement par nos interfaces graphiques, celle-ci reste tout de même accessible via un "instance.proportions".

Les fréquences d'apparitions sont utilisées pour construire l'arbre binaire selon l'algorithme de Huffman mais aussi pour calculer un checksum fletcher 16 servant à authentifier chaque code de Huffman, pour plus de renseignement sur cette partie veuillez vous adresser aux sections suivantes.

B.

Le code commence par créer une liste d'arbres binaires de Huffman, un pour chaque caractère du dictionnaire. Chaque arbre est initialisé avec un sommet contenant le poids et le caractère correspondant. La classe ArbreB_Huffman est une classe qui implémente un arbre binaire de Huffman. Le poids est la proportion de l'apparition du caractère dans le texte.

Ensuite, le code exécute une boucle while tant que la longueur de la liste d'arbres est supérieure à 1. À chaque itération, la liste est triée selon le poids de chaque arbre. Les deux arbres ayant les poids les plus faibles sont fusionnés en un nouvel arbre, qui est ajouté à la liste. La fusion se fait en créant un nouveau sommet dont le poids est la somme des poids des deux sommets fusionnés, et en attachant les deux arbres en tant que fils de ce sommet. Ainsi, la liste d'arbres est progressivement réduite jusqu'à ce qu'il ne reste plus qu'un seul arbre, qui est l'arbre de Huffman final.

Enfin, le code renvoie le premier et unique arbre de la liste. Cet arbre représente le code de Huffman qui peut être utilisé pour compresser le texte.

C.

nom de la fonction : encoding(conversion:dict, texte:str)

Pour encoder les caractères du texte source il faut un dictionnaire de conversion (*.huffmann) qui a précédemment été créé et enregistré via l'application, ce dictionnaire doit à minima contenir tous les caractères présents dans le texte source sinon une erreur surviendra.

La fonction commence par définir une chaîne de caractères vide nommée "output", qui servira à stocker le texte encodé. Ensuite la fonction utilise une boucle "for" pour parcourir chaque caractère

de la chaîne "texte" passée en argument. Pour chaque caractère, la fonction recherche sa valeur correspondante dans le dictionnaire de conversion et l'ajoute à la fin de la chaîne "output".

Après avoir terminé la boucle "for", la fonction ajoute une valeur de contrôle appelée "checksum" à la fin de la chaîne "output". Cette valeur est également extraite du dictionnaire de conversion.

Si une exception se produit pendant la boucle "for", la fonction affiche un message d'erreur spécifique en utilisant la fonction "showerror" et lève une exception de type ValueError.

Enfin, la fonction retourne la chaîne "output" encodée si elle a réussi à encoder le texte.

D.

nom de la fonction : decoding(conversion:dict, texte:str)

La fonction commence par extraire la valeur de contrôle "checksum" de la fin de la chaîne "texte". Pour ce faire, elle utilise la fonction de découpage de chaînes en Python, en utilisant la longueur de "checksum" pour déterminer la position de découpe. La partie de la chaîne avant le "checksum" est stockée dans la variable "texte", tandis que le "checksum" lui-même est stocké dans la variable "checksum".

La fonction vérifie ensuite que le "checksum" extrait correspond à la valeur de contrôle du dictionnaire de conversion. Si c'est le cas, le décodage peut continuer. Sinon, la fonction affiche un message d'erreur en utilisant la fonction "showerror" (qui n'est pas définie dans le code donné) et lève une exception de type ValueError.

Si le "checksum" est correct, la fonction crée un dictionnaire de conversion inversé appelé "conversion_reversed". Ce dictionnaire a les mêmes clés que le dictionnaire de conversion, mais les valeurs sont les clés et vice versa. Cela permet de rechercher facilement les clés du dictionnaire de conversion à partir des valeurs correspondantes.

La fonction initialise ensuite une chaîne de caractères vide nommée "output", ainsi que deux variables entières "i" et "j" qui serviront à parcourir la chaîne "texte". La variable "keys" est également initialisée avec les clés du dictionnaire de conversion inversé.

La fonction utilise ensuite une boucle "while" pour parcourir chaque caractère de la chaîne "texte". Pour chaque caractère, la fonction extrait une sous-chaîne "texte[i:j]" et recherche cette sous-chaîne dans les clés du dictionnaire de conversion inversé. Si la sous-chaîne est trouvée, la fonction ajoute la clé correspondante à la chaîne "output". Sinon, la fonction augmente la variable "j" pour inclure le caractère suivant dans la sous-chaîne, jusqu'à ce qu'elle trouve une sous-chaîne correspondante ou qu'elle atteigne la fin de la chaîne "texte". Si la fonction ne parvient pas à trouver une sous-chaîne correspondante, elle affiche un message d'erreur en utilisant la fonction "showerror" et lève une exception de type ValueError.

La boucle "while" continue jusqu'à ce que la variable "i" atteigne la fin de la chaîne "texte", ce qui signifie que la chaîne "output" a été entièrement construite.

Enfin, la fonction retourne la chaîne "output" décryptée si elle a réussi à décoder le texte.

IV.

A.
Pour faciliter l'utilisation de notre programme, une interface graphique était demandée et surtout nécessaire. Elle nous permet de lier toutes nos fonctions à une démonstration visuelle. Pour cela,

nous avons fait le choix d'utiliser la librairie ttkbootstrap, soit un dérivé de tkinter, pour développer notre interface graphique. Cette librairie permet d'apporter à notre programme une interface qui est visuellement plus agréable et « neuve » comparé à tkinter. Il est cependant nécessaire de l'installer avant d'exécuter le programme (`python -m pip install ttkbootstrap`). Deux interfaces différentes sont accessibles lors du lancement de main.py : la première est le programme utilisable, et la seconde est une interface unittests.

Cette première interface est divisée en deux grandes parties, une partie affichant l'arbre binaire et une seconde partie pour l'encodage et le décodage de texte. Sur la première page, 5 widgets sont positionnés. Un widgets texte permet à l'utilisateur d'insérer son texte sur lequel il veut exécuter l'algorithme d'Huffman. Pour cela, après l'avoir écrit/collé, il appuie sur un autre widget, le bouton "Afficher l'arbre de Huffman de votre texte" relié à la fonction affiche_arbre(), et celui-ci se dessine dans le canva, notre 3ème widget situé au bas de la fenêtre. Il est aussi possible d'ouvrir un fichier .txt et d'insérer tout texte qui s'y trouve dans le widget texte en appuyant sur le menu "Fichier" puis "Ouvrir" grâce à la fonction import_text(). Il peut ensuite parcourir l'arbre à l'aide de deux scrollbars en largeur et en profondeur. Les symboles sont affichés sous chaque feuille de cet arbre. De plus, il peut aussi connaître le chemin de chaque symbole présent dans le texte avec le dernier widgets présent, la listbox. Dans celle-ci, chaque symbole a son code et son poids en occurrence correspondant à côté de lui. Elle permet de faciliter le repérage de ceux-ci.
Sur la seconde page, 6 autres widgets sont positionnés. Une entrée identique à la page précédente permettant à l'utilisateur d'écrire/coller son texte sur lequel il veut appliquer l'algorithme. Il est nécessaire de créer et d'enregistrer un dictionnaire de cryptage avec le bouton "Enregistrer un dictionnaire de cryptage Huffman", relié à la fonction ExportCodes(), avant d'encoder ou de décoder un texte. Après avoir enregistré le dictionnaire de cryptage, en laissant son texte dans le widget, l'utilisateur peut appuyer sur le bouton "Encoder votre texte", relié à la fonction cryptag(), ou "Décoder votre texte binaire", relié à la fonction decryptage(), selon ce qu'il veut faire. Qu'importe son choix, une fenêtre lui demandera de choisir le dictionnaire correspondant à son texte. Le programme affichera le résultat dans le widget de texte du bas. Il peut copier ce résultat dans ce dernier widget en le sélectionnant.
Il est aussi possible d'accéder à la page github de ce projet en cliquant sur le menu "Aide" puis sur "Page GitHub" qui ouvrira un onglet internet sur votre navigateur grâce à la fonction webgithub().

La seconde interface est plus simple puisqu'elle à pour objectif de démontrer les possibilités de travail sur notre arbre créé dans le programme, à savoir l'ajout et la délétion de sommet, mais aussi la fusion et la décomposition d'arbre de la classe ArbreB_Huffman. Tous les boutons sont situés dans la barre de menu. Le premier menu "Arbre" permet de créer un arbre et de l'afficher avec le bouton "Créer à partir d'un texte", relié à la fonctions create_from_text(), de fusionner l'arbre affiché avec un second arbre en ajoutant du texte avec le bouton "Rajout de texte", relié à la fonction add_from_text(), et de décomposer l'arbre affiché en choisissant de garder le sous-arbre gauche ou droit en partant de la racine avec le bouton "Décomposition", relié à la fonction decomposer(). Le second menu concerne les sommets. Il permet d'ajouter un sommet et son poids avec le bouton "Ajouter", relié à la fonction add_sommet(), et de supprimer un sommet avec le bouton "Retirer", relié à la fonction rm_sommet().
L'arbre sera affiché dans la partie gauche de la fenêtre sous forme de liste de ses sommets, et à droite de la fenêtre sous forme de dessin.

V.

En conclusion,notre programme implémente un algorithme basé sur la propriété de l'algorithme de compression de Huffman. Cette application permet la construction d'un arbre binaire de Huffman à partir d'un texte ou d'un dictionnaire de proportions, la décomposition d'un arbre binaire en deux sous-arbres, la fusion de deux arbres en un, l'affichage de l'arbre dans le terminal et le dessin de l'arbre avec Tkinter.

La construction de l'arbre binaire de Huffman est l'étape cruciale de l'algorithme de Huffman qui permet de compresser les données en utilisant un code préfixe. La classe ArbreB_Huffman facilite cette étape en fournissant une méthode de construction efficace et facile à utiliser. En outre, la décomposition, la fusion et les méthodes d'affichage et de dessin permettent d'analyser l'arbre de Huffman et de mieux comprendre son fonctionnement.

En résumé, notre interface graphique est un outil pratique pour travailler avec l'algorithme de Huffman et pour comprendre les concepts fondamentaux des algorithmes de compression.