1 Introduction

Question 1 : Montrer que l'algorithme de modification de l'arbre n'échange jamais un nœud avec un de ses ancêtres.

La seule situation où l'on peut échanger deux nœuds est pendant le traitement d'un des nœuds si l'autre est son fin de bloc.

Cela implique que les deux nœuds ont le même poids.

Cependant le poids de chaque nœud interne est la somme des poids de ses deux fils.

Comme chaque nœud interne possède deux fils, la seule situation où le père a le même poids que l'un de ses fils est quand son deuxième fils est le caractère spécial car c'est la seule feuille de l'arbre ayant un poids nul.

Dans cette situation, quand on modifie ce nœud, on vérifie si sa fin de bloc est son père. Si c'est le cas, on va incrémenter son poids et traiter son père, qui lui à son tour n'aura pas un ancêtre comme fin de bloc car chaque nœud à partir de là a un frère avec un poids non nul.

Si son père n'est pas sa fin de bloc, alors il ne sera pas échangé avec ce dernier, ni avec un ancêtre pour la même raison que précédemment.

Question 2 : Quel est le nombre maximal d'échanges réalisés lors d'une modification de l'arbre (en fonction de la taille de l'alphabet des symboles) ?

En notant n la taille de l'alphabet des symboles. Le nombre d'échange maximal est log2(n)-1.

3 Compression et Décompression

Choix du langage : Python

Nous avons choisi d'implémenter cet algorithme en python en vue de la lisibilité de son code, qui se rapproche du pseudo-code donnée dans le cours. Il était donc simple à implémenter pour nous. En plus, comme c'est un langage très utilisé, il nous a été facile de se documenter sur les fonctions des bibliothèque python qui nous ont été utiles.

Malgré qu'il ne soit pas optimal niveau temps d'exécution, nous n'avons pas pris la rapidité de la compression et de la décompression comme étant un critère important dans cet algorithme.

Les structures de données utilisées : Classes

-Sdd : Notre structure principale. Elle stock la racine de l'arbre (une instance de Arbre, un pointeur vers la feuille spéciale ainsi qu'un dictionnaire symbole - code (le chemin de chaque symbole dans l'arbre).

-Arbre : Graphe

Logique d'un arbre, elle stock les nœuds (valeur, poids, enfants), sauf que nous avons ajouté un pointeur parent, donc la structure est un graphe.

L'ajout de ce pointeur parent facilite la modification de l'arbre, le parcours gamma dans la fonction traitement et le changement du chemin. Cependant, il plus couteux lors de l'initialisation et la modification du nœud.

Les fonctions :

Pour l'implantation des algorithmes de compression et décompression, nous avons suivi ce qu'il nous a été donné dans le cours.

La fonction compression parcourt le texte lettre par lettre et applique la fonction modification pour chaque lettre.

La fonction modification vérifie si la lettre a déjà été croisée, si ce n'est pas le cas alors elle crée le nœud correspondant à la lettre en tant que frère du caractère spécial en utilisant le pointeur vers le chemin spécial ainsi que le pointeur vers le père de ce dernier, puis elle le traite. Sinon elle prend le nœud correspondant à la lettre avec la fonction feuille, et vérifie s'il est frère du caractère spécial et si son père est son finbloc (toujours en utilisant le pointeur vers le père), si c'est le cas alors elle traite le père, sinon elle traite ce nœud.

Notre première version de la fonction de traitement appelait à chaque fois la fonction suivantsList qui renvoie le parcours GDBH de l'arbre, cette fonction étant couteuse car parcourant tout l'arbre (O(n)), n étant la taille maximum de l'arbre, et comme ce parcours ne change pas forcement à chaque étape (quand il n'y a pas d'échange de nœud), nous avons décidé de mettre ce parcours en paramètre de la fonction et de le changer uniquement en cas de changement de l'arbre, juste après la fonction échanger (O(n)). Cette modification nous a permis de diviser par presque 2 la vitesse de compression et décompression des textes dont on parle dans la partie suivante.

Notre fonction traitement/modification n'atteignent pas la complexité O(n), mais plutôt $O(n^2*nb\_echange\_max)$

Notre fonction de compression est donc de complexité $O(n^2*N*nb\_echange\_max)$ N étant la longueur du texte à compresser.

La fonction décompression fonctionne similairement, avec la même complexité.

Une manière de diminuer cette complexité serait d'optimiser la fonction de parcours GDBH. Notre fonction effectue un parcours en profondeur à chaque modification de l'arbre et à chaque échange de deux nœuds, il serait plus efficace de garder dans la structure de donnée le parcours en largeur de l'arbre en stockant les différents sous arbres.

Nous n'avons malheureusement pas eu le temps d'explorer cette solution.

4 Etude Expérimentale

Textes

Nous avons compressé des textes en plusieurs langues : français, anglais, arabe, tchèque, chinois et japonais.

Tout ça afin de tester l'impact de la langue/l'alphabet sur le taux de compression, et la durée de la compression/décompression. En général, nous obtenons des taux de compression entre 52% et 56% environ, sauf pour le texte chinois qui ne fait que 42% de compression.

Le taux de compression est calculé ainsi :

taille du fichier compressé (oct) / taille du fichier non compressé (oct).

Plus le taux de compression est proche de 0, plus le texte a été compressé.

Nous remarquons que la durée de compression pour texte_chinois.txt qui, même si plus petit en taille que le fichier texte_tchec.txt, est plus grande. Cela est dû au fait que l'alphabet chinois est codé sur plus d'octet que l'alphabet tchèque en UTF-8. Alors lors de sa lecture et de son écriture octet par octet, son codage est plus couteux en temps.

On conclut donc que le temps de compression n'a pas toujours un effet sur le taux de compression.

Nous remarquerons que tous les textes du projet Gutenberg contiennent une partie importante de texte en anglais, peu importe la langue étudiée. Cela engendre la compression de deux langues différentes, souvent avec deux alphabets différents, comme par exemple notre fichier texte_japonais.txt, ce qui explique son taux de compression assez élevé de 57%.

Fichiers aléatoires

La fonction pour générer les fichiers aléatoires avec une distribution uniforme a été générée en utilisant chat GPT, on a considéré que c'était une fonction assez générique.

La fonction pour générer les fichiers aléatoires avec des distributions normales et exponentielles a été empruntée à un camarade. On ne l'a donc utilisé que pour générer les fichiers et pas mis dans le projet.

Avec les fichiers aléatoires, quand on génère les caractères avec une distribution uniforme, comme il y'a environ 150000 caractères (dont une partie considérable sont les emojis), en testant avec des textes de longueur 1000 caractères ou 10000, les résultats ont été similaires, c-à-d 1.4 de taux de compression.

Les fichiers n'ont alors pas du tout été compressé même avec une taille conséquente. Ce qui les distinguent des autres types de fichiers qui eux, se compressent de plus en plus que leur taille est grande.

On explique cela par la génération de caractères différents à chaque fois, puisqu'ils ont tous une probabilité égale d'apparaitre.

Il faudrait des fichiers à partir de 1m de caractères pour remarquer une réelle compression.

On conclut que lors de la compression des fichiers générés (longs ou courts) avec des caractères qui ne se répètent pas. Les fichiers compressés sont plus lourds que les fichiers originaux.

Dans le tableau ci-dessous, on a généré des fichiers en utilisant différentes distributions de probabilité.

Tous les fichiers contiennent 10000 caractères.

Type de distribution	Nb caractères distincts	Taux de compression
Uniforme	9111	1.46902
Normale	5959	1.14864
Exponentielle	477	0.25458

Ici nous remarquons que la compression du fichier généré avec une distribution de probabilité normale est quand même meilleure que la distribution uniforme. Nous n'avons pas pu tester les deux premières distributions avec des fichiers plus lourds car prenants trop de temps pour la compression.

Cependant, intuitivement, nous sommes amenés à penser que les fichiers plus lourds auront des taux de compression différents. Une distribution normale sera meilleure qu'une probabilité uniforme car la probabilité d'apparition des symboles est plus concentrée vers le centre. Cependant, les symboles ayants des chances d'apparitions moins probables dans la distribution normale rendront la compression moins efficace et donc comparable à la compression d'un fichier avec une distribution uniforme.

Enfin, avec une distribution exponentielle la compression se fait beaucoup mieux en raison de la génération d'un petit nombre de caractères différents dans notre exemple, avec 39% de probabilité d'apparition pour le caractère le plus fréquent.

Fichiers de code

Les fichiers de code (par ex notre fichier exo3.py qui contient les fonctions de compression et décompression et toutes les fonctions dépendantes) ne se compressent pas forcément mieux que les fichiers de langage naturel. Notre intuition était le contraire, vu que dans le code, beaucoup de mots se répètent (while, if, {,…). Cependant l'algorithme de Huffman dynamique ne prend pas en compte la répétition de mot ou de bout de texte, alors il le compresse comme un texte avec une distribution normale des symboles.

Nous pensons donc qu'il serait plus efficace de compresser ce type de fichier avec des algorithmes de compression par dictionnaire, entre autres, l'algorithme Lempel-Ziv-Welch.

Comme les fichiers de texte naturelle, il serait plus intéressant de compresser des fichiers de code plus lourds que celui que nous avons testé

Ci-dessous, un tableau démonstratif de nos résultats :

Fichier	Taille non-compressée (octets)	Taille compressée (octets)	Temps de compression (ms)	Taux de compression
Texte_anglais	441189	244487	45460	0.55415
Texte_arabe	40562	21351	4929	0.52638
Texte_japonais	37921	21473	25460	0.56626
Texte_tcheque	832259	463033	89462	0.55636
Texte_chinois	287589	123184	613376	0.42833
Blaise_Pascal_	118026	66246	13536	0.56128
Exo3.py	8338	5495	817	0.65903
Texte_exo4	2962	4167	3095	1.40682

Nous n'avons pas gardé les liens vers ces fichiers, ils sont tous situés dans le répertoire data/inputFile.

Pour la décompression nous avons les même remarques que pour la compression.

En conclusion, l'algorithme de Huffman dynamique compresse mieux les textes «utiles» composés d'un alphabet limité et qui sont de taille importante.