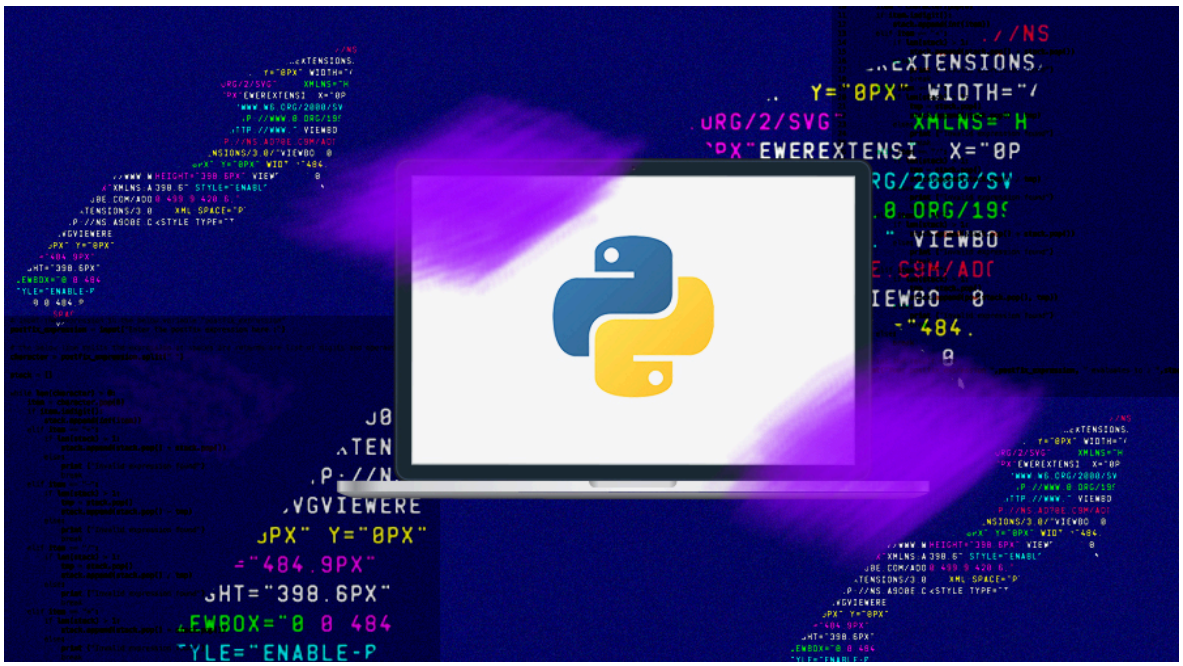
Université Bretagne Sud École d'ingénieurs ensibs Université Bretagne Sud	Python
	JONCOUR Tristan **ANDRE Romain** **CDA 3A, TD2, TP4**

Compression de texte sans perte : Codes de Huffman



Sommaire

Introduction

Dans le cadre de ce projet, nous avons implémenté trois variantes de l'algorithme de Huffman : une version statique, une version classique (ou dynamique), et une version streaming (ou adaptative). L'objectif était de comprendre les différences entre ces approches, de comparer leurs performances, et de documenter nos choix de conception.

Huffman statique

Pour commencer, on devait implémenter la version statique de l'algorithme de Huffman. Un dictionnaire de fréquences était déjà fourni, donc on n'avait pas à analyser le texte pour calculer quoi que ce soit. Nous avons juste utilisé ce dictionnaire tel quel pour construire l'arbre.

Nous avons transformé chaque caractère du dictionnaire en un nœud, avec sa fréquence associée. Ensuite, nous avons trié tous les nœuds du plus petit au plus grand et fusionné les deux plus faibles à chaque étape. À chaque fusion, nous créions un nœud parent avec la somme des fréquences, et nous recommençons, jusqu'à ce qu'il ne reste plus qu'un seul nœud, la racine de l'arbre. À la fin, nous avions un arbre de Huffman complet, construit manuellement à partir des fréquences fixes.

Un point un peu piégeux au début était le traitement de l'espace. Dans le dictionnaire, l'espace n'apparaît pas sous forme de " ", mais sous forme de "<sp>". Du coup, avant d'encoder le texte, nous avons remplacé tous les espaces par ce token. Sinon, ça générait des erreurs, car l'arbre ne reconnaissait pas le caractère.

Ensuite, pour générer les codes binaires, nous avons fait un parcours récursif de l'arbre. Chaque fois que nous descendions à gauche, nous ajoutions un "0" ; à droite, un "1" ; et quand nous arrivions sur une feuille, nous avions le code complet du caractère. Ce code-là, nous l'avons utilisé pour transformer tout le texte en une longue chaîne de bits.

Là où nous avons vraiment dû réfléchir, c'est quand nous sommes tombés sur des caractères qui ne sont pas dans le dictionnaire. Par exemple, des majuscules, des lettres accentuées ou des symboles qu'on ne voit pas souvent. Plutôt que de bloquer ou d'ignorer ces caractères, nous avons décidé d'ajouter un nœud spécial dans l'arbre, appelé "<inconnu>". Quand un caractère est inconnu, nous commençons par écrire le code de "<inconnu>", puis nous ajoutons la longueur UTF-8 du caractère (en binaire sur 8 bits), puis les octets UTF-8 eux-mêmes, chacun encodé sur 8 bits aussi.

Grâce à ça, la décompression fonctionne même si le texte contient des caractères un peu bizarres ou non prévus. C'est plus lourd côté encodage, mais ça garantit qu'on peut toujours récupérer le texte original sans perte.

Un autre point technique important : le padding. Comme les bits ne tombent pas toujours pile sur des octets, nous avons dû ajouter du remplissage à la fin. Nous avons compté combien de bits nous avons ajouté et nous stockons cette info dans le premier octet du fichier. Comme ça, au moment de la décompression, nous savons exactement combien de bits il faut retirer.

Ce que nous avons trouvé le plus difficile, ce n'était pas l'algorithme en soi, mais la gestion des cas particuliers : le traitement des espaces, le codage des caractères inconnus et le respect du format binaire du fichier.

Huffman classique

Pour cette deuxième version, nous avons travaillé sur l'algorithme de Huffman classique, c'est-à-dire qu'on ne part pas d'un dictionnaire imposé, mais qu'on construit tout dynamiquement à partir du contenu réel du fichier à compresser. L'idée, c'est de faire une compression adaptée au texte lui-même, ce qui permet souvent d'avoir un meilleur taux de compression que la version statique.

Dès le début, on lit le fichier texte, et on compte combien de fois chaque caractère apparaît. Ce comptage nous donne un dictionnaire de fréquences. Ensuite, on construit un arbre de Huffman exactement comme dans la version statique, sauf que cette fois, l'arbre change en fonction du fichier. Donc deux fichiers différents auront deux arbres différents.

Une grosse différence ici, c'est que, comme l'arbre n'est pas connu à l'avance, on est obligé de le stocker dans le fichier compressé. Sinon, on ne pourrait pas décompresser correctement. Pour ça, nous avons mis en place un système de sérialisation de l'arbre : quand on tombe sur une feuille, on encode un "1", puis la longueur du caractère en UTF-8 (sur 8 bits), et ensuite les octets UTF-8 du caractère. Quand on tombe sur un nœud interne, on met juste un "0" et on continue à encoder ses deux enfants dans l'ordre. Tout l'arbre est donc transformé en une longue chaîne de bits.

Côté décompression, c'est l'inverse. On commence par lire cette chaîne de bits et on reconstruit l'arbre depuis zéro. Ça demande un peu d'attention, surtout pour bien gérer les index dans la chaîne de bits, mais une fois qu'on a bien compris l'ordre (préfixe, récursif), ça fonctionne bien.

Une fois l'arbre reconstruit, on lit le reste du fichier bit par bit. On descend dans l'arbre selon les 0 et les 1 jusqu'à tomber sur une feuille. À ce moment-là, on récupère le caractère associé, on l'ajoute au résultat, et on repart de la racine pour lire le caractère suivant.
Une difficulté qu'on a rencontrée, c'est la gestion du padding à la fin du fichier. Comme les bits n'arrivent pas toujours à un multiple de 8, il faut compléter avec des 0, sinon on ne peut pas convertir ça proprement en octets. Du coup, j'ai rajouté les 8 premiers bits du fichier pour stocker la longueur du padding. Ça permet de le retirer proprement au moment de la lecture.

Ce qui est intéressant avec cette version, c'est qu'on est obligé de penser le format du fichier compressé, il ne suffit pas de stocker des bits, il faut aussi structurer le contenu (arbre + données). Et contrairement à la version statique, on peut ici compresser n'importe quel caractère, même ceux qu'on n'a jamais vus, puisqu'ils seront tous pris en compte dès le début dans le comptage des fréquences.

Huffman streaming

Pour le Huffman en streaming, on ne stocke pas l'arbre dans le fichier. L'arbre est construit et mis à jour au fur et à mesure qu'on lit les caractères du fichier, aussi bien pour la compression que pour la décompression.

Pour implémenter cette version, nous avons, en plus des nœuds, créé une nouvelle classe ArbreHuffman qui nous permet de centraliser la gestion de l'arbre, notamment la navigation, la mise à jour dynamique, et les opérations spécifiques comme le swap ou la recherche de chemins binaires.

Pour la compression, dès qu'un caractère est lu, on vérifie s'il a déjà été rencontré. Si c'est le cas, on utilise son code binaire actuel pour l'écrire dans le flux compressé. Sinon, on encode un symbole NYT (Not Yet Transmitted), suivi du code binaire du caractère en UTF-8. Ensuite, l'arbre est mis à jour pour inclure ce nouveau caractère.

Dans notre code, la mise à jour (quand on gère un nouveau caractère) se fait en créant un nouvel objet Noeud pour le caractère, en ajustant les liens parent-enfant, et en attribuant un numéro unique à chaque nouveau nœud pour respecter les règles de l'algorithme de Vitter.

Lorsque la fréquence d'un nœud augmente (lui-même ou un de ses sous-nœuds a augmenté sa fréquence), il peut ne plus respecter l'ordre de priorité dans l'arbre. Pour corriger cela, on identifie un nœud leader, c'est-à-dire un nœud ayant la même fréquence mais une priorité plus élevée (numéro plus élevé). Ensuite, on effectue un swap entre le nœud courant et le meneur pour rétablir l'ordre.

Pour trouver le code binaire des caractères, on remonte jusqu'à la racine à partir du nœud (si on connaît le nœud du caractère). Si on ne connaît pas le nœud du caractère (donc qu'on gère un nouveau caractère), on encode avec le chemin vers le NYT, puis la longueur du code UTF-8 du caractère sur un octet, puis le code UTF-8 en lui-même.

Pour la décompression, on reconstruit l'arbre à partir des bits lus. D'abord, les 8 premiers bits indiquent la longueur du padding ajouté à la fin (même chose que pour huffman classique), qui est ensuite retiré pour obtenir les bits effectifs. Aussi, on initialise un arbre basique (un seul nœud NYT) que l'on va enrichir durant notre lecture des bits. Ensuite, les bits sont parcourus un par un pour naviguer dans l'arbre : un "0" descend à gauche et un "1" descend à droite. Lorsqu'une feuille est atteinte, le caractère correspondant est récupéré. Si le nœud NYT est atteint, cela signifie qu'un nouveau caractère est encodé juste après. Sa longueur en UTF-8 est lue, suivie des octets du caractère, qui sont décodés et ajoutés à l'arbre. Ce processus continue jusqu'à ce que tous les bits soient traités. A la fin, nous avons reconstruit l'arbre.

Le point le plus complexe était de comprendre l'algorithme dans un premier temps. Dans un second temps, le plus compliqué était d'implémenter la logique de recherche de nœud leader, puis de l'utiliser pour mettre à jour l'arbre.

Analyse comparative des performances

Critère	Huffman statique	Huffman classique	Huffman Streaming
Taille fichier original	55 445 octets	55 445 octets	55 445 octets
Taille fichier compressé	38 126 octets	35 882 octets	33 293 octets
Taille après décompression	55 718 octets	55 718 octets	55 718 octets
Taux de compression	31,2 %	35,3 %	39,9 %
Arbre inclus dans le fichier	Non	Oui	Non
Vitesse d'exécution	0.036 secondes (compression) 0.082 secondes (décompression)	0.016 secondes (compression) 0.034 secondes (décompression)	1.675 secondes (compression) 1.657 secondes (décompression)
Complexité de l'algo	Moyenne	Moyenne+ (arbre à sauvegarder)	Complexe
Idéal pour	Textes simples ou normalisés (avec beaucoup de caractères similaires qui reviennent, ce qui fait qu'on profiterait de l'arbre de fréquence connu) On pourrait penser à un format d'écriture standardisé par exemple, où il n'y a qu'une certaine sélection de caractères qui peuvent être utilisés	Textes complexes (nombreux caractères différents) et longs Il s'agit de la meilleure version pour compresser de gros fichiers On peut aussi voir une utilisation pour le stockage de gros fichiers, du fait que l'arbre est directement dans le fichier compressé.	Transmission sur le réseau, de messages ou petits fichiers, là où la bande passante est importante (car il s'agit du meilleur taux de compression) On peut aussi directement penser à la compression de flux en temps réel, mais la durée de décompression plus longue que les autres huffman pose problème.