Algorithmique avancée – TP

Compression bzip

Dernière modification: 3 octobre 2022

L'objectif de ce TP est de programmer la compression et la décompression d'assez gros fichiers en utilisant la méthode de l'utilitaire `bzip`. Celle-ci est basée sur la transformée de Burrows-Wheeler et le codage de Huffman, qui seront détaillés ci-dessous.

1 Préliminaires

On va acquérir le contenu du fichier à compresser comme une liste ou un tableau d'octets. Tous les traitements se feront donc sur des tableaux d'entiers compris entre 0 et 255 (ou un peu plus). Dans cet énoncé, le mot *texte* désigne ces tableaux d'entiers.

Q1. Écrire une fonction `contents` qui à partir d'un nom de fichier renvoie un tableau d'octets correspondant à son contenu. Il faut ouvrir le fichier en mode *binaire*.

Q2. Tester avec un petit fichier, et un gros : le fichier `pg5097.txt` contient une version de *Vingt-mille lieues sous les mers* de Jules Verne, qui fait 942 377 octets et nous permettra de vérifier que nos algorithmes passent (à peu près) à l'échelle. Il faudra faire ces tests tout au long du TP, en particulier dès qu'on a une nouvelle paire encodeur-décodeur.

2 Codage de Huffman

Normalement un symbole correspondant à octet est codé par la représentation binaire de celui-ci. Donc tous les symboles ont la même taille (8 bits).

Le codage de Huffman permet de coder les symboles fréquents par un nombre réduit de bits et les moins fréquents par un nombre plus élevé de bits. Le principe est de construire un arbre binaire dont les feuilles sont les symboles. Le chemin jusqu'à la feuille donne le code : quand on va à gauche, cela correspond à 0, et quand on va à droite cela correspond à 1 (cf. figure 1).

Par construction, le code correspondant à un tel arbre est un *code préfixe*, c'est-à-dire qu'il n'existe pas de symboles a et b tels que le code de a est un préfixe de b. On veut construire un code optimal au sens où la taille du tableau d'entiers encodés avec ce code est de taille minimale. On peut montrer qu'il existe toujours un code optimal qui est *préfixe* – et l'algorithme de Huffman en construit un.

Enfin, il est clair qu'un arbre correspondant à un code optimal est tel que chaque nœud qui n'est pas une feuille a exactement deux successeurs (essayez de le démontrer).

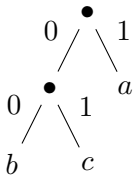

FIGURE 1 – Un arbre représentant un code préfixe. Le code de a est 1, celui de b est 00 et celui de c est 01.

2.1 Création du code

Pour construire l'arbre, on utilise donc le nombre d'occurrences de chaque symbole dans un algorithme glouton :

— on maintient une liste de nœud qui initialement contient les feuilles, c'est-à-dire les symboles, avec leurs nombres d'occurrences ;
— on trouve les deux nœuds les moins fréquents x et y et on en fait les successeurs d'un nouveau noeud de l'arbre. On associe à ce nouveau nœud la somme des nombres d'occurrences de x et y, et on le met dans la liste ;
— on recommence jusqu'à ce que la liste ne contienne plus qu'un seul élément qui sera la racine.

Q3. Écrire une classe `HuffmanTreeNode` qui représente un nœud de l'arbre, et qui contient un symbole, un nombre d'occurrences, et des pointeurs gauche (ou zéro) et droite (ou un) ;

Q4. Écrire une méthode `merge` qui permet de créer un nouveau nœud correspondant à la fusion de deux nœuds comme expliqué ci-dessus ;

Q5. Écrire une fonction (méthode statique) `less` qui prend deux nœuds et renvoie vrai si le nombre d'occurrences du premier est strictement plus petit que celui du deuxième, et faux sinon ;

Q6. Pour la liste des nœuds, on utilise une *file de priorité*. Écrire une classe `PQueue` qui implémente le tas binaire implicite vu en cours. Ajouter un membre qui est une fonction générique de comparaison des éléments et sera donné à la création de la file. Écrire l'insertion, l'extraction du minimum, et l'obtention de la taille de la file.

Q7. Créer une classe `Huffman` avec un attribut `tree` qui sera (un pointeur vers) la racine de l'arbre du code ;

Q8. Ajouter une méthode `build_tree` à la classe `Huffman` qui à partir du texte construit l'arbre du code par l'algorithme glouton expliqué ci-dessus. Pour faciliter le décodage, on ajoute à la fin du texte un symbole dont la valeur sera 259 (créer une constante `huffman_marker`) ;

Q9. Ajouter une méthode récursive `build_codemap_rec` à la classe `HuffmanTreeNode` qui remplit un tableau dont les indices sont les symboles (représentés par des entiers donc) et les valeurs sont des tableaux de 0 et de 1 donnant le code associé au symbole dans l'arbre.

Q10. Ajouter un attribut `codes` à la classe `Huffman` et une méthode `build_codemap` qui le remplit en appelant la fonction de la classe précédente.

2.2 Codage et décodage

Q11. Ajouter une méthode `encode` à la classe `Huffman` qui associe à chaque symbole du texte son code binaire. Dès qu'on a 8 bits, on en fait un octet qu'on met dans le tableau de sortie. On pourra utiliser les opérateurs de décalage de bits `<<` et de *ou* bit à bit `|` plutôt que la division par deux et l'addition. Attention le dernier octet ne sera vraisemblablement pas complet et il faut faire comme si l'on avait encore des 0 pour compléter. C'est le marqueur de fin qui nous permettra d'ignorer ces 0 supplémentaires au décodage.

Q12. Ajouter une méthode `decode` à la classe `Huffman` qui prend un texte codé par la méthode `encode` et redonne le texte original, sans le marqueur de fin. On pourra utiliser les opérateurs de décalage de bits `>>` et de *et* bit à bit `&`.

3 Transformée de Burrows-Wheeler

L'objectif de la transformée de Burrows-Wheeler est de grouper les symboles identiques, de façon certes moins efficace qu'un tri, mais de façon réversible ! On exploitera ensuite ces groupes de symboles identiques dans les transformations présentées dans les deux sections suivantes en les représentant de manière compacte.

On écrit le texte et ses permutations successives ligne par ligne dans une table (qui sera donc carrée). Puis on trie les lignes et enfin on renvoie la dernière colonne. Comme pour le codage de Huffman, pour faciliter le décodage, on ajoute un marqueur de fin de valeur 256 (créer une constante `bwt_marker`). La figure 2 présente le processus de codage.

s	i	m	i	l	i	.		.	s	i	m	i	l	**i**
.	s	i	m	i	l	i		i	.	s	i	m	i	**l**
i	.	s	i	m	i	l		i	l	i	.	s	i	**m**
l	i	.	s	i	m	i	$\longrightarrow$	i	m	i	l	i	.	**s**
i	l	i	.	s	i	m		l	i	.	s	i	m	**i**
m	i	l	i	.	s	i		m	i	l	i	.	s	**i**
i	m	i	l	i	.	s		s	i	m	i	l	i	**.**

FIGURE 2 – Transformée de Burrows-Wheeler de la chaîne `simili.`. Le résultat est `ilmsii.`, en supposant que le caractère . est plus petit que les autres.

Sauf pour la première ligne, le dernier caractère est celui qui précède le premier caractère dans le texte original. Statistiquement les symboles précédant sont souvent les mêmes, donc en triant les lignes on a des chances de faire apparaître des symboles similaires côte-à-côte dans la dernière colonne.

3.1 Codage

Le codage est la partie critique de la transformée. On peut le faire en temps et espace linéaire avec un dérivé du tri par base (*radix sort*) qui produit un tableau de suffixes (*suffix*

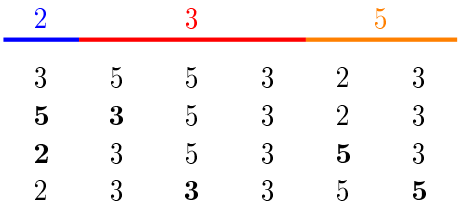

FIGURE 3 – Un tableau à trier (première ligne). Au dessus les places qu'occuperont les éléments dans le tableau trié. Les lignes suivantes montrent les étapes du tri par comptage sur ce tableau.

array). Ici, pour simplifier, on se contentera d'un tri par base classique, et la complexité pire cas sera quadratique.

L'idée du tri par base est simple. On a des tableaux de symboles à trier par exemple des nombres à n chiffres. On tout trie selon le premier chiffre, puis selon le deuxième, et ainsi de suite jusqu'au dernier. Si le tri utilisé est *stable*, c'est-à-dire que sur ses égalités il préserve l'ordre précédant, alors le résultat final est complètement trié. On peut aussi trier en partant du dernier chiffre, selon ce qu'on veut.

3.1.1 Tri par comptage

Le tri classiquement utilisé pour chaque itération du tri par base est le tri par comptage (*counting sort*). On compte le nombre d'occurrences de chaque élément puis l'utilise pour déterminer la partie du tableau qui contient ces occurrences (voir la figure 3). Enfin on parcours le tableau à trier et on met chaque élément à sa place dans un nouveau tableau.

Clairement le tri n'est pas en place et requiert $O(n+r)$ espace supplémentaire ou n est la taille du tableau et r le nombre de valeurs différentes des éléments à trier. C'est un tri stable et sa complexité temporelle pire cas est $O(n)$.

Comme nous allons potentiellement manipuler des gros tableaux d'entiers, on va implémenter une version non-stable mais en place qui ne requiert que $O(r)$ espace supplémentaire (on modifiera aussi légèrement le tri par base pour prendre l'instabilité en compte). Au lieu de mettre l'élément dans un nouveau tableau, on le permute avec celui qui est à sa place. Et on recommence : on n'avance à l'élément suivant que lorsque l'élément courant est à sa place (voir figure 3). Enfin, quand on arrive dans la zone d'un nouvel élément, il faut sauter par dessus les éléments déjà à leur place. Ainsi dans la dernière ligne de la figure 3, quand 2 est à sa place au début, on passe directement à l'indice 2 (qui contient un 5) sans passer par l'indice 1 qui contient un 3 qui a déjà été mis à sa place au premier échange (ligne 2).

Q13. Écrire une fonction `counting_sort` qui trie un tableau de symboles (de 0 à `bwt_marker`) par le tri par comptage en place. Il faut d'abord calculer les fréquence, puis pour chaque symbole, l'indice de début de la partie du tableau qui le contiendra.

Q14. Modifier la fonction précédente (ou en écrire une autre) pour trier un tableau de tableaux de symboles selon le n^{e} symbole du tableau. On pourra passer une fonction comme paramètre qui trouve l'élément à comparer en fonction.

Q15. Modifier la fonction précédente (ou en écrire une autre) pour trier les rotations du texte initial en fonction de leur n^{e} symbole. Le tableau à trier contient les indices

sur lesquelles débutent les rotations successives (ou de façon équivalente le nombre de symboles de décalage). La fonction de correspondance est plus complexe : à partir de l'indice dans le tableau à trier, on trouve la rotation et à partir de n et du texte original on trouve le caractère dans cette rotation.

3.1.2 Tri par base

Comme notre version du tri par comptage n'est pas stable, il faut être un peu plus précis dans le tri par base : on commence par trier tout le tableau selon le dernier symbole, puis on trie séparément, selon le symbole précédent, les sous-tableaux dans lesquels le dernier symbole est constant. Et ainsi de suite, avec les autres symboles : on trie ainsi des tableaux de plus en plus petits.

Q16. Modifier le tri par comptage pour qu'il renvoie les indices de début et fin des sous-tableaux dans lesquels se trouvent les égalités pour le critère de tri.

Q17. Écrire une version récursive `radix_sort_rec` du tri par base ;

Q18. Comme on peut avoir des fichiers très gros, la profondeur de récursion peut aussi être très grande, ce qui est incompatible avec la plupart des les langages non-fonctionnels (pour Python le maximum est 1000 par défaut). En utilisant une pile, modélisée par un tableau dans lequel on lit et on écrit à la fin, écrire une version dérécursifiée `radix_sort` du tri par base ;

Q19. Écrire la fonction `bwt_encode` qui réalise la transformée de Burrows-Wheeler. Comme le calcul est coûteux, on le réalise par blocs : on se donne une constante `block_size` qui vaut par exemple 500 000, on découpe le texte en tranches de `block_size` symboles (potentiellement moins pour le dernier bloc), et on réalise la transformée séparément sur chaque bloc.

3.2 Décodage

Pour retrouver le texte initial à partir de sa transformée, il faut se rappeler que, dans le tableau des rotations, sur une ligne donnée ne se terminant pas le marqueur spécial, la première colonne donne le symbole suivant, dans le texte original, celui de la dernière colonne. Pour la ligne se terminant par le marqueur spécial la première colonne donne le premier symbole du texte original.

Par ailleurs, on retrouve facilement la première colonne à partir du texte codé en triant simplement les symboles.

On retrouve donc ainsi de proche en proche tous les caractères du texte original mais attention, il peut y avoir plusieurs occurrences du même caractère, et donc il faut prendre la « bonne » occurrence !

Q20. Écrire une fonction `find_nth` qui trouve l'indice de la n^{e} occurrence d'un élément dans un texte. Cette fonction doit fonctionner en temps constant $O(1)$ amorti sur tout le décodage.

Q21. Écrire une fonction `find_rank` qui trouve le rang de l'élément donné par son indice dans un texte trié. Cette fonction doit avoir un pire cas (temporel) $O(\log_2 n)$.

Q22. Écrire une fonction `bwt_decode` qui décode le texte (tous les blocs) codé par la transformée de Burrows-Wheeler en utilisant le principe décrit ci-dessus. Cette fonction doit supprimer les marqueurs `bwt_marker`.

4 Move-to-front (MTF)

L'objectif de cette transformation est globalement de transformer des suites de symboles identiques en suites de 0. Pour se faire, on met nos symboles (des entiers entre 0 et 256) dans l'ordre dans une liste.

Chaque symbole est représenté par son indice dans la liste (commençant à 0) donc initialement chaque symbole est représenté par sa valeur. Quand on lit un symbole, on va chercher son indice, qu'on met dans le tableau de sortie, puis on déplace le symbole en tête de la liste, de sorte que s'il se répète, ses prochaines occurrences produiront des 0 (voir figure 4).

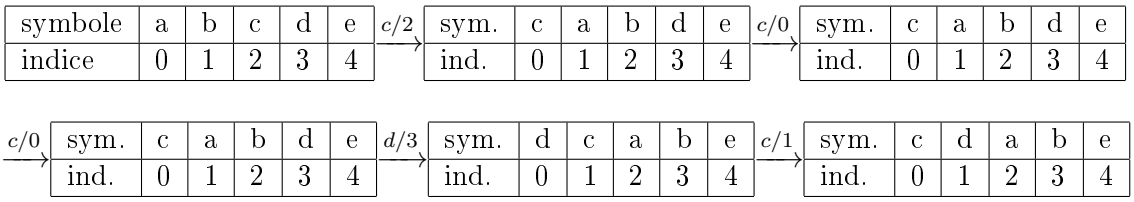

FIGURE 4 – *Move-to-font* pour la séquence `cccdc` pour un alphabet de symboles $\{a, b, c, d, e\}$. La notation $c/2$ indique qu'on lit un c et qu'on écrit un 2. La sortie est donc 20031.

Pour inverser la transformée, on cherche les symboles à partir de leur indice en utilisant la même liste initiale et la même façon de bouger les nœuds.

Q23. On encode la liste des symboles par une liste simplement chaînée. Écrire une classe `ListNode` dont le seul attribut est un symbole ;

Q24. Ajouter une méthode `insert_front_node` qui ajoute un nœud en tête de la liste représentée par le nœud courant ;

Q25. Ajouter une méthode `insert_front_symbol` qui ajoute un symbole (il faut donc créer le nœud) en tête de la liste représentée par le nœud courant ;

Q26. Ajouter une méthode `delete_next` qui efface le nœud suivant le nœud courant ;

Q27. Ajouter une méthode `find_npi` qui étant donné un symbole, donne le nœud le contenant, le parent de ce nœud, et l'indice de ce nœud ;

Q28. Écrire la fonction `mtf_encode` qui réalise la transformée MTF ;

Q29. Ajouter une méthode `find_np` qui étant donné un indice, donne le nœud le contenant et le parent de ce nœud ;

Q30. Écrire la fonction `mtf_decode` qui réalise la transformée MTF inverse.

5 Encodage des longueurs de chaînes de zéros (*Zero run-length encoding (zRLE)*

Grâce à MTF, on a (statistiquement) de longues séquences de 0 consécutifs, mais pas de telles séquences pour les autres symboles. On va encoder ces séquences en les remplaçant par leur longueur, qui est exponentiellement plus petite. Plutôt que d'utiliser un codage en

binaire, on va utiliser la numération bijective en base 2, qui est plus courte et permet d'exploiter naturellement le fait que la longueur n'est jamais 0.

Dans le codage binaire positionnel usuel, les chiffres sont 0 et 1 et la séquence de chiffres $a_0a_1...a_n$ représente le nombre entier $\sum_{i=0}^{n} a_i 2^i$. Notons qu'un entier donné a une infinité de telles représentations en ajoutant des 0 à gauche.

Dans le codage binaire bijectif, chaque entier a une unique représentation. On ne s'intéresse ici qu'aux entiers positifs. On utilise les chiffres 1 et 2 et la séquence de chiffres $a_0a_1...a_n$ représente aussi le nombre entier $\sum_{i=0}^{n} a_i 2^i$. Par exemple la représentation de 10 est 122 $(2*1+2*2+1*4)$. En binaire positionnel, le plus court serait 1010. Notons au passage que 0 est encodé par la liste de chiffres vide.

Pour trouver les chiffres a_i en binaire positionnel de l'entier x, on a la récurrence :

$$q_0 = x \text{ et } q_{n+1} = \lfloor q_n/2 \rfloor \text{ et } a_n = q_n - 2q_{n+1}$$

On peut continuer à l'infini, ce qui ajoute des 0 à gauche, mais il suffit de s'arrêter quand q_{n+1} vaut 0.

Dans la représentation bijective la récurrence est :

$$q_0 = x \text{ et } q_{n+1} = \lceil q_n/2 \rceil - 1 \text{ et } a_n = q_n - 2q_{n+1}$$

Et le dernier chiffre a_k est bien sûr tel que $q_{k+1} = 0$.

Q31. Écrire la fonction `zrle_encode` qui élimine les séquences de 0 dans le texte et les remplace par leur longueur exprimée avec la numération bijective en base 2. On définira deux constantes `zrle_one` et `zrle_two`, valant respectivement 257 et 258, qui représenteront les chiffres de la représentation bijective.

Q32. Écrire la fonction `zrle_decode` qui réalise la transformation inverse.

6 Codes de Huffman canoniques

Avec les sections précédentes, on a obtenu une représentation compressée et réversible de notre texte original. La transformée de Burrows-Wheeler, MTF, et zRLE sont réversibles directement en utilisant la chaîne codée.

Pour le codage de Huffman, par contre, il faut connaître l'arbre du code. Il faut donc ajouter une représentation du code à la chaîne compressée. On peut, par exemple, simplement ajouter le nombre de symboles puis pour chaque symbole un triplet constitué du symbole de la longueur de son code et du code lui-même. Cette représentation est malheureusement très volumineuse et on peut faire mieux.

On peut en fait reconstruire un nouveau code de Huffman dit canonique à partir seulement de la longueur des codes de chaque symbole. On trie les symboles selon la longueur de leur code.

Soit n la longueur de code minimale, on associe au symbole associé le nouveau code constitué de n zéros. Puis pour chaque code de même taille on associe les codes obtenus en incrémentant simplement la valeur (on sait qu'il n'y aura pas dépassement par construction). Enfin, quand on a une augmentation de taille, on ajoute, *après* avoir incrémenté le code, autant de zéros à droite dans le nouveau code pour obtenir la même taille. Par exemple, supposons qu'on avait le code suivant, déjà trié par longueurs :

d	b	e	a	g	f	e	j	i	h
10	110	111	000	0010	0110	0100	0101	0111	0011

On obtient le code canonique suivant :

d	b	e	a	g	f	e	j	i	h
00	010	011	100	1010	1011	1100	1101	1110	1111

On peut donc s'épargner la transmission du code lui même, seuls le nombre de symboles, les symboles eux-mêmes, et les longueurs suffisent. On pourrait ne pas transmettre explicitement les symboles en transmettant tout l'alphabet mais c'est en général moins avantageux.

Enfin, plutôt que transmettre les longueurs, on peut ne transmettre que les différences qui sont plus petites (et en particulier tiennent sur un octet). De même plutôt que transmettre le nombre total de symboles, puisqu'on sait qu'on a forcément au moins les 4 symboles spéciaux ajoutés lors des transformations, on peut n'écrire que le nombre de symboles moins 4 qui lui aussi est assuré de tenir sur un seul octet.

Q33. Ajouter à la classe `Huffman` une méthode `canonical_diffs` qui à partir de l'attribut `codes` renvoie une représentation du code canonique sous la forme d'une liste dont les indices pairs contiennent les symboles utilisés et les indices impaires les incréments de la longueur du code associé. Pour le premier symbole, c'est sa vraie longueur plutôt qu'un incrément (ou un incrément par rapport à 0...) ;

Q34. Écrire une fonction `binary_list` qui donne une liste de n chiffres de la représentation binaire positionnelle du nombre x ;

Q35. Ajouter à la classe `Huffman` une méthode `build_canonical_codemap` qui remplace l'attribut `codes` par sa version canonique ;

Q36. Ajouter à la classe `Huffman` une méthode récursive `rebuild_tree_rec` qui à partir du code canonique calcul par la fonction précédente reconstruit et renvoie l'arbre du code correspondant ;

Q37. Ajouter à la classe `Huffman` une méthode `rebuild_tree` qui modifie l'attribut `tree` en appelant la méthode de la question précédente.

7 Finalisation

On a maintenant tous les éléments nécessaires, il suffit de tout mettre ensemble dans le bon ordre (attention ce n'est pas l'ordre dans lequel on les a développés) :

Q38. Écrire une fonction `encode` qui encode un texte en appliquant les transformation successives développées ci-dessus. On ajoutera au tout début le nombre de symboles (-4 comme expliqué ci-dessus) puis le code sous la forme donnée par la méthode `canonical_diffs`. Ajouter éventuellement l'écriture du résultat dans un fichier ;

Q39. Écrire une fonction `decode` qui décode un texte codé par la fonction précédente, éventuellement récupéré du fichier écrit.

Q40. Vérifier qu'en encodant puis décodant, on obtient bien la même chose, en en temps raisonnable ($<$30s) y compris pour le texte de Jules Verne. Comparer éventuellement la taille et la durée avec ce que produit `bzip2`.

On obtient une compression très similaire à ce que produit le vrai outil `bzip2`.
Les différences principales sont vraisemblablement :

— `bzip2` utilise plusieurs codes de Huffman et peut en changer tous les 50 symboles pour coller au mieux aux variations de fréquences dans le texte ;
— Son implémentation de la transformée de Burrows-Wheeler, et en particulier du tri, est vraisemblablement linéaire ;
— La représentation de quels symboles sont utilisés dans le code est plus sophistiquée, avec un tableau de bits à deux niveaux ;
— On a optimisé plein de choses au niveau algorithmique mais on n'a pas cherché à optimiser plus finement (l'encadrant a fait ça en Python par exemple, ha ha).