POLYTECHNIQUE
MONTRÉAL

LE GÉNIE
EN PREMIÈRE CLASSE

Département de génie informatique et génie logiciel

INF8770 – Technologies Multimédias

Travail pratique #1
Comparaison et caractérisation de méthode de codage

Luc Courbariaux 1670433
Renaud Picotin 1737483

Septembre 2017

Question 1

Suivant les directives données pour ce laboratoire et les restrictions imposées, nous avons développé quatre hypothèses concernant les différences entre les méthodes de codage de Huffman et de LZW.

Hypothèse 1: Selon la méthode de Huffman, nous devrions obtenir de meilleurs résultats qu'avec la méthode LZW lorsqu'il y a beaucoup d'inégalités entre les fréquences d'apparition des caractères. Nous supposons cette hypothèse vraie parce qu'avec le codage de Huffman, le code d'un caractère ayant une grosse fréquence d'apparition prend moins d'espace.

Hypothèse 2: Avec la méthode LZW, on obtient une meilleure performance que Huffman pour des textes ou il y a beaucoup de répétition de suites de caractères. Comme dans la méthode LZW on construit un dictionnaire à partir de chaines de caractères, si celles-ci reviennent avec une forte fréquence, on devrait obtenir une meilleure performance qu'avec Huffman qui, dans ce cas, doit toujours coder chaque caractère séparément.

Hypothèse 3: Pour des textes de très petite taille, la méthode de codage de Huffman devrait performer moins bien que LZW. Comme Huffman utilise un dictionnaire pour coder le contenu du texte, il faut ajouter ce dictionnaire à notre produit fini. On émet l'hypothèse que pour des textes de petite taille, lorsque le nombre de caractères du texte est plus petit ou équivaut à la taille de l'alphabet utilisé, la taille finale du dictionnaire nuira considérablement à la performance de l'algorithme.

Hypothèse 4: Lorsqu'on a une entropie de caractère élevé dans un texte, les deux méthodes auront une performance équivalente. Comme la fréquence de chaque caractère sera équivalente en moyenne, l'utilisation de la méthode d'Huffman ne pourra profiter de la fréquence élevée de certains caractères. Pour LZW, comme il n'y aura pas de répétition fréquente de chaine de caractère, on ne pourra tirer avantage de l'utilisation de son dictionnaire.

Question 2

Afin d'évaluer les hypothèses mentionnées précédemment, nous avons décidé d'évaluer la performance de nos deux méthodes suivant les modifications de deux paramètres, soit la taille de l'alphabet et la longueur des textes. Nous testerons d'abord des textes anglais de tailles 48, 465, 3145 et 83701 caractères. Ce test nous permet de tester les hypothèses 1 et 2, puisque la fréquence d'apparition des caractères dans des textes en anglais est très différente d'un caractère à l'autre et est caractérisé par de nombreuses répétition de syllabes et mots : les deux algorithme devraient éventuellement achever des taux dee compression honorables, meilleurs qu'avec de simples tests aléatoires.

Ensuite, notamment pour noter les différences théorisée plus tôt, nous générerons des textes constitués de caractères aléatoires (forte entropie) pour tester les hypothèses suivantes. Ces tests se feront pour des alphabets de 3, 40 et 98 caractères constituants des textes de 20, 400, 8000 et 160000 caractères aléatoires.

Les tests sur des échantillons de petite taille (anglais et texte aléatoire confondus) nous permettron s de tester l'hypothèse 3, tandis que ceux aléatoire avec un alphabet de taille 3 serons de bons examples pour l'hypothèse 2. Finalement, les tests sur des textes de grande taille à forte entropie, peu importe l'alphabet, nous permettent de tester l'hypothèse 4 en regardant si les performances des deux méthodes sont semblables.

Question 3

Pour réaliser la batterie de tests nécessaire, nous avons développé 3 petits modules en python utilisant la librairie bitstring, soit un pour la méthode de Huffman, un pour la méthode LZW et un dernier (benchmark.py) qui sert à tester les performances des deux dernières méthodes.

Pour s'assurer de la rigueur de nos algorithmes de compression, nous avons implémenté les aussi bien l'encodage que le décodage. À chaque expérience, en plus de noter la taille encodée de l'échantillon fourni, nous vérifions que le texte décodé est exactement celui original.

Dans le module de Huffman, on commence par parcourir le texte complet et compter le nombre d'occurrences de chaque symbole, puis on construit un arbre avec ceux-ci qui sera notre dictionnaire. On terminera par l'encodage du texte à l'aide du dictionnaire ainsi créé et ajoute ce dernier au texte encodé.
Le décodage extrait d'abord l'arbre puis interprète les suites de bits pour naviguer entre ses branches et trouver le caractère correspondant.

Le code pour LZW semble beaucoup plus court et simple que pour Huffman. Dans la partie encodage, nous commençons par assigner l'espace nécessaire à la création du dictionnaire, puis on lit le texte caractère par caractère. On construit ensuite la version codée du texte en regardant si la chaine qu'on construit existe déjà dans notre dictionnaire, et on ajoute une nouvelle chaine à celui-ci en insérant le nouveau caractère manquant à la fin. La section décodage procède à l'inverse afin de vérifier si le contenu de la version codée correspond au code original.

Finalement, le 3e module sert à faire les tests de performance. Il permet de générer les textes aléatoires pour les différents tests ainsi que de mesurer la performance des algorithmes (taille finale et ratio de compression) en lançant les algorithmes de Huffman et LZW.

Suite à la réalisation de l'expérience, nous avons obtenu les résultats suivants :

Tableau de résultat de compression pour un textes en anglais

Taille(Char)/ Taille (bits)	Taille avec LZW (bits)	Ratio de compression (LZW)	Taille avec Huffman (bits)	Ratio de compression (Huffman)
48 / 384	504	0,76	1196	0,32
465 / 3720	3660	1,02	3706	1
3145 / 25160	16752	1,5	15905	1,58
83701 / 669608	309672	2,16	364609	1,84

Tableau de résultat de compression pour un alphabet de 3 caractères

Taille(Char)/ Taille (bits)	Taille avec LZW (bits)	Ratio de compression (LZW)	Taille avec Huffman (bits)	Ratio de compression (Huffman)
20 / 160	144	1,11	159	1,01
400 / 3200	1584	2,02	782	4,09
8000 / 64000	18708	3,42	13444	4,76
160000 / 1280000	284784	4,49	266747	4,8

Tableau de résultat de compression pour un alphabet de 40 caractères

Taille(Char)/ Taille (bits)	Taille avec LZW (bits)	Ratio de compression (LZW)	Taille avec Huffman (bits)	Ratio de compression (Huffman)
20 / 160	240	0,67	835	0,19
400 / 3200	4380	0,73	4015	0,8
8000 / 64000	56640	1,13	44942	1,42
160000 / 1280000	989148	1,29	864935	1,48

Tableau de résultat de compression pour un alphabet de 98 caractères

Taille(Char)/ Taille (bits)	Taille avec LZW (bits)	Ratio de compression (LZW)	Taille avec Huffman (bits)	Ratio de compression (Huffman)
20 / 160	240	0,67	986	0,16
400 / 3200	4776	0,67	7146	0,45
8000 / 64000	77400	0,83	57995	1,1
160000 / 1280000	1442088	0,89	1074207	1,19

Une analyse plus poussée du contenu des tableaux sera présentée à la section suivante du rapport.

Question 4

Analyse des résultats : hypothèse 1

Les résultats du tableau de compression pour des textes en anglais sont ceux nécéssaires à l'évaluation de l'hypothèse 1. On observera ici principalement les ratios de compression.

En regardant un graphique des ratios de compression en fonction de la taille du texte pour des textes anglais, on remarque que LZW semble avoir une performance plus élevée que Huffman. En effet, on obtient de meilleurs résultats avec LZW pour de petits et pour de grands textes. Pour les deux tailles de texte intermédiaire, il semblerait que les deux méthodes offrent des résultats très semblables. Il nous est donc possible ici d'écarter l'hypothèse 1 et d'affirmer qu'elle n'est pas nécessairement vraie. Il est possible d'expliquer le faible rendement de la méthode d'Huffman pour de petits textes par la présence d'un dictionnaire, ce qui augmente la taille du fichier compressé. De plus, pour des textes de grandes tailles, il y a beaucoup de répétitions de chaines de caractères, ce que ne tient pas en compte la méthode de Huffman et dont profite la méthode LZW.

Analyse des résultats : hypothèse 2

Pour le test de notre 2e hypothèse, regardons le tableau de résultats de compression pour un alphabet de 3 caractères. Encore une fois, nous ne regarderons que le ratio de compression des textes codés.

Suite à l'observation des résultats pour ce test, on remarque que la méthode de Huffman performe beaucoup mieux que la méthode LZW. En effet, on obtient des ratios de compression de 4 à presque 5 aussitôt que le texte augmente de taille. Instinctivement, nous devrions infirmer notre hypothèse 2. Cela dit, il semble qu'avec Huffman on atteint un plateau de performance un peu en dessous de 5, alors qu'avec LZW la tendance donne l'impression de s'améliorer avec l'augmentation du nombre de caractères dans le texte. Nous pouvons donc supposé de Huffman nous donne un avantage pour les textes de court à long, mais qu'avec des documents de très grandes tailles (plus de 160 000 caractères) LZW deviendraient beaucoup plus avantageux, car de longues chaines de caractères sont déjà présentes dans le dictionnaire et permette de remplacer de longues parties de texte. À la lumière de ces résultats, il n'est pas possible d'infirmer ni de confirmer l'hypothèse. Il serait nécessaire de reproduire ces tests avec des textes de plus grandes tailles pour effectuer des vérifications plus poussées.

Analyse des résultats : hypothèse 3

Pour vérifier notre 3e hypothèse, il nous est nécessaire de regarder les résultats des 4 tableaux pour des textes de très petite taille et de comparer LZW et Huffman.

On remarque dans le tableau précédent que peu importe le cas, pour des textes de petite taille, LZW a toujours une meilleure performance de Huffman. Cela est explicable par la présence du dictionnaire qui prend une place trop grande dans le fichier final. Il nous est donc possible de confirmer l'hypothèse 3. On remarquera une montée des performances pour les deux méthodes lorsqu'il n'y a que 3 caractères dans l'alphabet, principalement pour Huffman. Cela s'explique par la présence de très peu de bits pour coder chaque caractère dans le cas de Huffman, et par une grande répétition des mêmes chaines de symboles dans le cas de LZW.

Analyse des résultats : hypothèse 4

Pour la vérification de notre 4e hypothèse, nous regarderons les textes les plus longs (160000 symboles) de chacune des catégories de test afin d'observer s'il est vrai que nous obtiendrons des résultats semblables pour les deux tests lorsque nous sommes en présence d'une forte entropie dans les caractères.

Au premier regard, il semblerait qu'il n'y ait en effet aucune différence significative entre les deux méthodes. Cela dit, on remarquera aussi que dans les 3 cas d'entropie élevés nous obtenons de meilleurs résultats avec Huffman qu'avec LZW. Dans le cas du texte anglais à faible entropie, LZW offre une meilleure performance, explicable par la répétition de chaines de caractères. La différence de performance entre la faible entropie et la forte entropie ne permet pas de confirmer l'hypothèse 4, puisque les résultats ne diffèrent pas suffisamment pour être significatifs. Similaire au cas de l'hypothèse 2, nous croyons qu'il faudrait effectuer des tests sur des textes beaucoup plus longs que 160000 symboles afin de vérifier s'il y a une différence entre les textes anglais et les textes aléatoires pour les méthodes LZW et de Huffman.

Analyse générale des résultats

En retrospective, bien que la taille de l'alphabet et des échantillons de test que nous avons choisi semblent toujours raisonnables, on pourrait faire la critique que la variance de la taille prise par la version encodée augmente plus le texte est court, et qu'il aurait été plus précis de prendre la moyenne de plusieurs expéirence similaires par catégorie (par exemple pour la catégorie “texte anglais de 48 charactères” on aurait pu utiliser une vingtaine d'échantillons plutôt qu'un seul).

Remarquons d'autre part que la performance des algorithme est liée à l'efficacité de leur implémentation, c'est-à-dire qu'une optimisation pourrait faire rendre un algorithme plus performant ; par exemple LZW aurait pu avoir de meilleurs résultats si on avait utilisé une taille de code dynamique (en tirant parti du fait que le dictionnaire a un nombre croissant de d'entrées et que les codes de peuvent être plus grand que celui des entrées à un instant t : on commencerait en utilisant le nombre de bits d'encodage minimal pour l'alphabet initial et ajouterait des bits au fûr et à mesure), cela permettrait selon nos estimations de gagner entre 10% et 30% de performance.