INF8770 – Technologies multimédias

Hiver 2018

TP No. 1

Groupe 1

Fabrice Charbonneau - 1798064
Félix Boulet - 1788287

Soumis à : Guillaume-Alexandre Bilodeau

2 février 2017

Question 1: Hypothèses

Nos hypothèses concernant la performance des deux méthodes (Huffman et LZW) sont les suivantes:

1. La méthode de Huffman est meilleure que LZW (en termes de longueur de code) lorsque la quantité de symboles uniques à encoder est petite, dans le message original. En ayant un nombre de symboles uniques largement inférieur au nombre de bits servant à les représenter. Par exemple, pour une image RGB comportant 1000 couleurs (10 bits) sur 16 777 216 possibles (24 bits), la compression sera plus efficace, car la représentation binaire des symboles sera plus courte.

2. La méthode de Huffman sera également efficace si il y a présence de symboles rares dans la séquence de symboles à coder, mais le nombre total de symboles doit tout de même rester relativement bas. Un message contenant un grand nombre de symboles étant tous uniques (par exemple, la séquence des 95 caractères ASCII imprimables) serait un cas où la méthode de Huffman ne serait pas vraiment efficace. En revanche, un message contenant du bruit aléatoire à travers des répétitions serait efficacement encodé par Huffman, car les codes correspondant aux symboles du bruit, plus longs, seront peu présents dans le message final.

3. La méthode LZW devrait être en général assez efficace lorsqu'un message à encoder contient une grande quantité de symboles, parce que cela augmente les chances que des séquences répétitives soient présentes dans le message à encoder.

4. Un cas où la méthode de Huffman ne serait pas efficace serait par exemple une image représentant un gradient horizontal de couleurs : grand nombre de symboles distribués uniformément. Ce cas serait par contre encodé efficacement en utilisant LZW : une séquence de couleurs retrouvée sur une ligne sera ensuite reconnue sur la ligne suivante (et une séquence plus longue ajoutée au dictionnaire), et ainsi de suite (jusqu'à coder une ligne entière de l'image à la fois, si l'image est carrée).

5. La méthode LZW ne sera pas très efficace pour encoder des images réelles (photographies), car de trop nombreuses variations de couleurs seront présentes et feront gonfler le dictionnaire de manière démesurée.

6. Pour du texte (message composé de caractères ASCII), Huffman sera largement meilleur que LZW dans le cas d’un message généré aléatoirement. En effet, puisqu’il risque d’y avoir peu de répétitions en raison de la variété de symboles utilisés, LZW ne pourra remplacer que peu de séquences dans la chaîne de caractères.

7. En revanche, dans le cas d'un texte écrit manuellement (un texte anglais par exemple), LZW risque de démontrer une meilleure performance relative. Les mots récurrents pourront être remplacés par des séquences de bits beaucoup plus courtes que la représentation normale de la chaîne de caractères correspondante (8 bits * longueur de la chaîne).

Question 2: Expériences

Afin de vérifier nos hypothèses, nous allons effectuer des tests en encodant différents types de données en utilisant les deux algorithmes, soit Huffman et LZW, implémentés en Python.

Données à encoder

Voici la liste des "messages" qui seront utilisés aux fins de test :

- Des images bitmap "idéales", de taille 100x100 : couleur unie, motif répété. et des images bitmap "pire cas", de taille 100x100 : gradient horizontal noir vers blanc, gradient 2D de couleur et de saturation (afin d'avoir ~ 10 000 couleurs différentes).

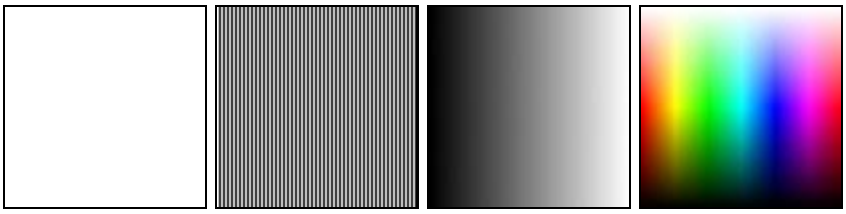


Figure 1 : De gauche à droite -
white.bmp, stripes.bmp, gradientBW.bmp, fullgradient.bmp

L'idée ici est que chaque image tentera d'aider ou de nuire à un ou deux algorithme(s). La première et la seconde devraient être encodées efficacement par les deux algorithmes (ou ou deux symboles répétés), alors que la 3e devrait nuire à Huffman (et bénéficier à LZW) et la dernière devrait nuire aux deux algorithmes (beaucoup de symboles, aucune répétition).

- Des images bitmap "réelles", de taille 100x100 : photographies variées, ramenées à la taille désirée pour réduire le temps de traitement.

Figure 2 : De gauche à droite -
ae86.bmp, fuji.bmp, stallman.bmp, train.bmp, volcano.bmp

Ici, chaque image tente de représenter différents aspects que l'on peut retrouver sur des photographies ou des images qui ne sont pas "parfaites". La première est très sombre et possède peu de couleurs, la seconde contient beaucoup de bruit (les fleurs de cerisier et la neige sur le mont Fuji), la troisième est relativement simple (fond, cheveux, peau, etc.), la quatrième combine bruit (arbres, neige) et couleurs unies (sur le train) et la dernière est un mélange de tout (couleur unie du volcan, bruit des nuages, peu de couleurs).

- Des chaînes de caractères générées aléatoirement, de longueur et de nombre de types de caractères variables.
  Les chaînes de caractères sont générées aléatoirement avec la librairie `strgen` de Python. Les tests de chaînes de caractères aléatoires ont les caractéristiques suivantes:
  1 seul symbole : chaînes de longueurs 100, 1000 et 10000
  2 symboles : chaînes de longueurs 100, 1000 et 10000
  3 symboles : chaînes de longueurs 100, 1000 et 10000
  5 symboles : chaînes de longueurs 100, 1000 et 10000
  10 symboles : chaînes de longueurs 100, 1000 et 10000
  26 symboles : chaînes de longueurs 100, 1000 et 10000
  62 symboles : chaînes de longueurs 100, 1000 et 10000
  Le fait d'avoir des tests de 26 et 62 symboles est simplement dû au choix des caractères des chaînes générées aléatoirement : 26 symboles correspond à l'alphabet en minuscule *ou* majuscule, et 62 est l'alphabet minuscule *et* majuscule (26 + 26 = 52) plus tous les chiffres de 0 à 9 (52 + 10 = 62).

- Des chaînes de caractères issues de textes en anglais. Les textes qui seront encodés sont contenus dans les fichiers `largeText.txt`, `mediumText.txt`, `mediumRepetitive.txt`, `mediumShort.txt` et `shortText.txt`. Le premier est un texte tiré de Wikipédia faisant 7007 caractères, qui utilise des termes plutôt diversifiés. Le second est un texte de 1269 caractères avec quelques mots qui se répètent souvent. Le troisième texte consiste en les paroles de la fameuse chanson "Let it be" des Beatles, avec de nombreuses répétitions des mêmes mots, contenant 1221 caractères. Finalement, le quatrième texte possède 500 caractères et le dernier 69.
- Des chaînes de caractères servant spécifiquement à tester certaines hypothèses comportant des cas spéciaux. Le fichier `symbolesRares.txt` contient une chaîne de caractères comportant majoritairement les symboles 'A' et 'B', et quelques symboles n'apparaissant qu'une seule fois.

Critères d'évaluation

En raison de la variance sur le temps d'exécution amenée par l'implémentation en Python, nous n'utiliserons qu'un seul critère pour évaluer l'efficacité des deux algorithmes (nous négligerons le temps) :

- La taille du message encodé (incluant celle du dictionnaire nécessaire pour le décoder, soit le dictionnaire entier pour Huffman ou le dictionnaire initial pour LZW) et le taux de compression par rapport au message original (*space-saving*).

Code informatique

Les codes informatiques utilisés afin d'effectuer les tests sont des programmes en Python 3.6. Le code pour la méthode de Huffman est fortement inspiré d'une implémentation trouvée sur *www.techrepublic.com/article/huffman-coding-in-python*. Le code concernant LZW est un code Python suivant le même corps que le code Matlab `CodageLZW.m` présent sur le site Moodle du cours INF8770.

Question 3: Code informatique

Le code de la méthode de Huffman est contenu dans les deux fichiers `Huffman.py` et `Node.py`, alors que le contenu de la méthode LZW est dans le fichier `LZW.py`. Le code implémenté se retrouve dans l'annexe.

Huffman

Tout d'abord, la classe `Node` du fichier `Node.py` est utilisée afin de construire l'arbre des symboles du message à encoder. Cet arbre est standard et se balance de lui-même lorsqu'utilisé avec une file de priorité pour le remplir.

Dans le programme `Huffman.py`, une première lecture du message permet de créer une liste de `Node` comprenant tous les types de symboles (ligne 9) contenus dans le message à coder. L'utilisation d'un `heap` (avec la fonction `heapify(items)`) permet ensuite de trier naturellement les symboles par leur nombre d'occurrences, ce qui facilite la création de l'arbre qui doit en tenir compte (lignes 9 à 15). La fonction `encode` (ligne 29) est une fonction récursive qui assigne à chacune des feuilles de l'arbre (les symboles du message à encoder) une représentation binaire selon le chemin parcouru dans l'arbre (0 à gauche et 1 à droite).

Finalement, en ayant la représentation binaire de chacun des symboles, on parcourt le message à encoder une dernière fois et chacun des symboles est remplacé par sa représentation binaire (lignes 32-33). À la fin, on retourne le message codé et le dictionnaire des différents codes.

LZW

Le code implémentant l'encodage LZW est basé sur le fichier `CodageLZW.m` sur Moodle. Il est défini dans le fichier aptement nommé `lzw.py`.

En premier lieu, le message est transformé en liste de symboles, puis en ensemble distinct (un `set`). Le nombre de bits nécessaires `bitsNeeded` pour coder chacun des symboles dans un dictionnaire initial est déterminé par la fonction `findNumberOfBits(nSymbols)` (lignes 43-44).

Par la suite, ce dictionnaire initial est construit en assignant chacun des symboles de l'ensemble distinct (qui sont désormais uniques) comme une clé du dictionnaire (lignes 12-13). La valeur liée à chaque clé est une version en format binaire sur `bitsNeeded` bits de l'index du dictionnaire (fourni par `enumerate`). Par exemple, un message "AABACB" serait initialement codé dans le dictionnaire comme `{'A': '00', 'B': '01', 'C': '10'}`, le `set` étant `{A, B, C}`.

Ensuite, le code se comporte de façon très similaire à l'exemple donné sur Moodle. Le message est parcouru par une boucle indexée selon `i`, qui se chargera d'effectuer l'algorithme LZW tel que vu en classe. On crée d'abord une sous-chaîne à partir du caractère indexé à `i` (qui sera soit ajouté à la sous-chaîne actuelle si la sous-chaîne précédente se trouvait déjà dans le dictionnaire, soit composée de ce seul caractère) (ligne 22). On tente ensuite d'aller chercher le codage correspondant à cette sous-chaîne dans le dictionnaire (la sous-chaîne agit comme clé) (ligne 23). Si l'entrée de dictionnaire correspondant à cette clé existe, on sauvegarde son codage et on incrémente `i` (lignes 34-39). Sinon (au caractère suivant, par exemple), on ajoute le codage précédemment sauvegardé à la chaîne de retour (`encoded_message`) et on ajoute une nouvelle entrée au dictionnaire se servant de la sous-chaîne nouvellement créée (lignes 24-27). On vérifie par le fait même s'il est nécessaire d'augmenter la longueur du codage correspondant à chaque symbole, et on le fait si c'est le cas (lignes 29-31). On réinitialise ensuite la sous-chaîne et on décrémente `i`, afin de rester sur le même caractère pour la prochaine sous-chaîne (les symboles individuels sont tous codés dans le dictionnaire initial, donc on passera au bloc `else` à la prochaine itération) (lignes 32-33).

À la toute fin, on obtient le message codé dans son ensemble, et on le retourne de même que le dictionnaire initial (nécessaire pour décoder LZW).

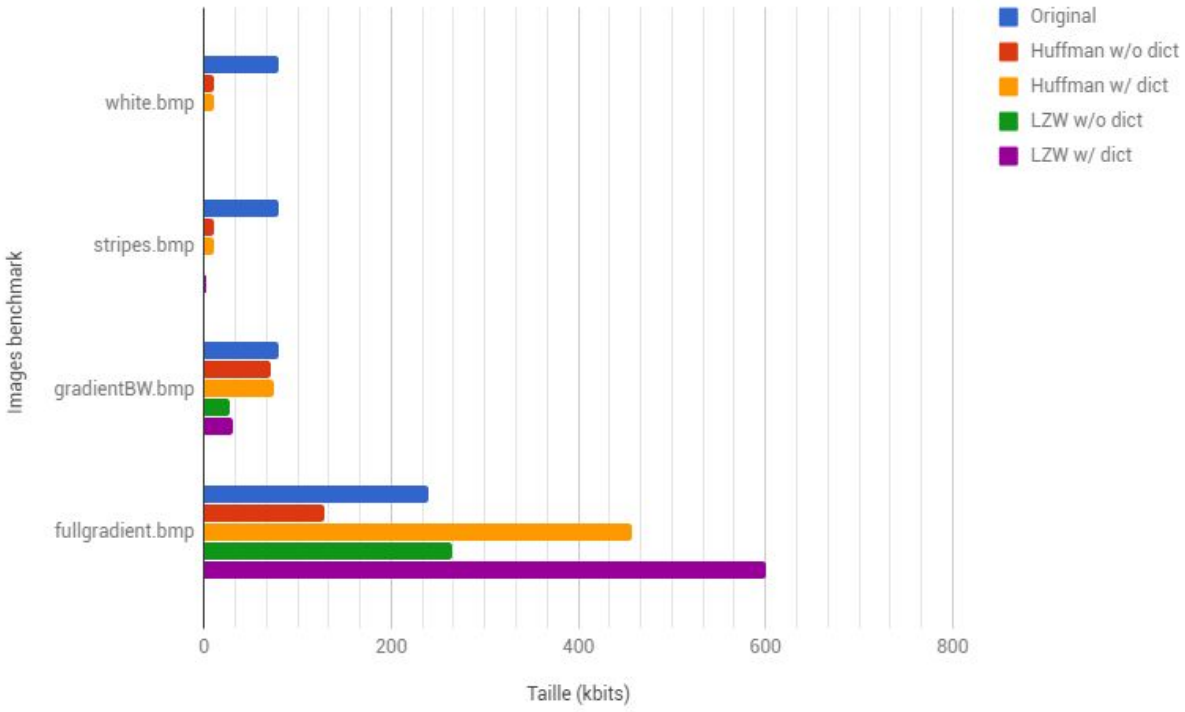
Génération de chaînes de caractères aléatoires

Le fichier `test.py` sert à générer des chaînes de caractères aléatoires et à afficher les résultats des tests dans les fichiers `Stats_from_random_string_generation.log` et `Full_verbose_results_from_random_string_generation.log`. Les chaînes de caractères sont générées à l'aide de la librairie `strgen` de python. La fonction `test_random_string` prend en paramètres la taille et les types de caractères que l'on veut dans une chaîne de caractères. Finalement, dans le bas du fichier, les fichiers `log` sont vidés, puis on appelle la fonction plusieurs fois dans des boucles `for` afin d'écrire les résultats des tailles, des taux de compression et du temps pris par LZW et Huffman pour chacune des chaînes.

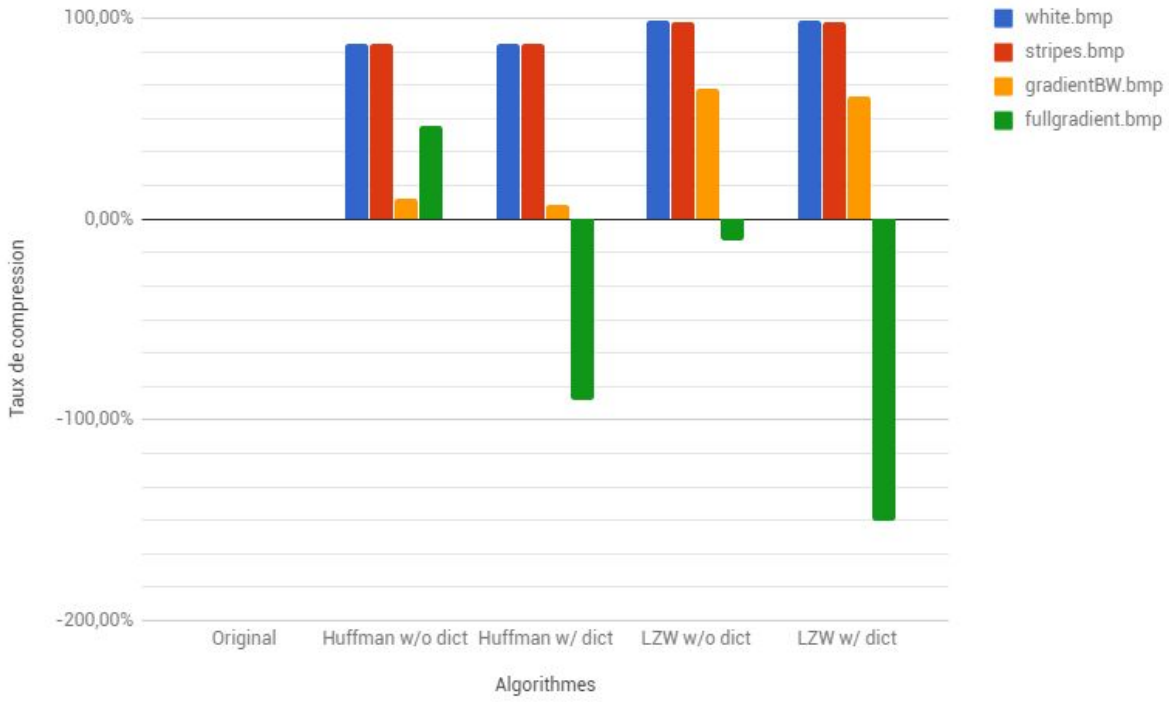
Encoder

Le fichier `encoder.py` permet d'exécuter le tout en spécifiant un fichier en argument sur la ligne de commande. Il peut prendre n'importe quel type de fichier non-binaire (.txt, code source, etc.) ou de type image (.png, .bmp, etc.). Pour décoder ces derniers, il utilise la librairie Python `imageio`. Il demande ensuite si l'on désire utiliser un encodage des symboles sur 8 bits ou 24 bits (triplets RGB), ce qui a été utilisé avec nos différentes images. Finalement, il lance le calcul des codes Huffman et LZW et affiche les résultats par rapport aux tailles avec et sans dictionnaire de chacun des algorithmes. Il sauvegarde en même temps le message codé ainsi que les dictionnaires produits par chacun des algorithmes dans des fichiers texte nommés `huffman.txt` et `lzw.txt`.

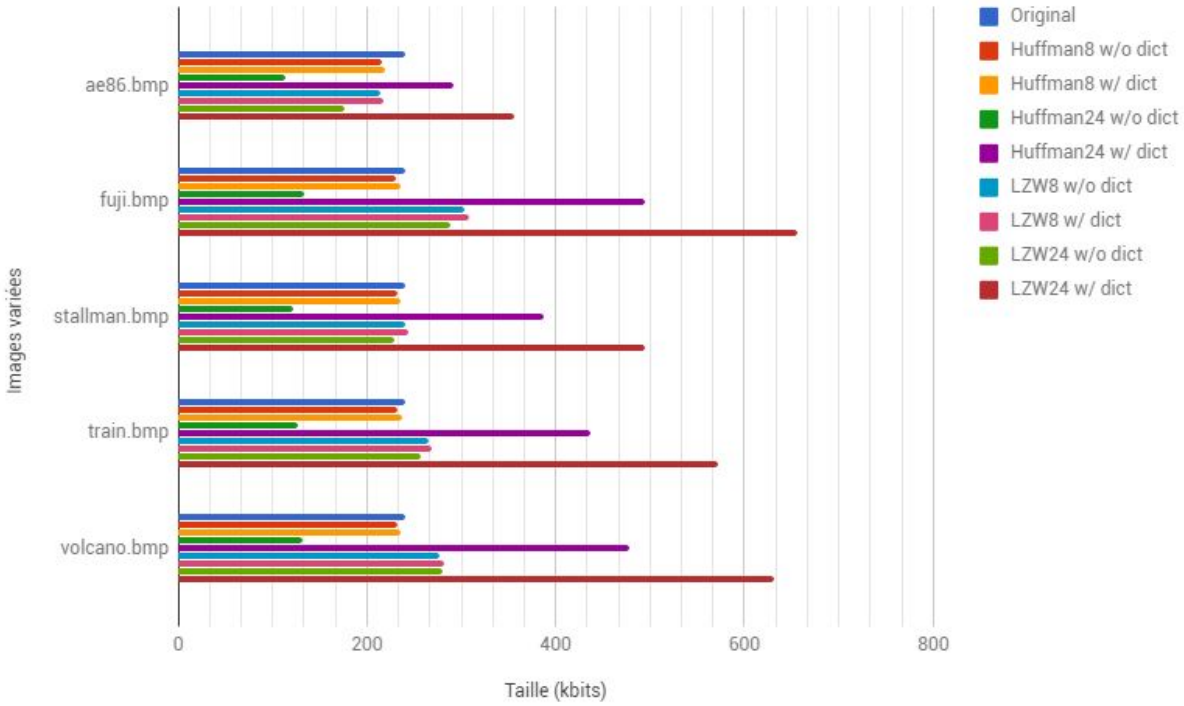
Voici les résultats de l'exécution de ce code, ainsi que de `test.py`, pour chacune de nos images et chacun de nos fichiers texte énoncés à la question 2 :



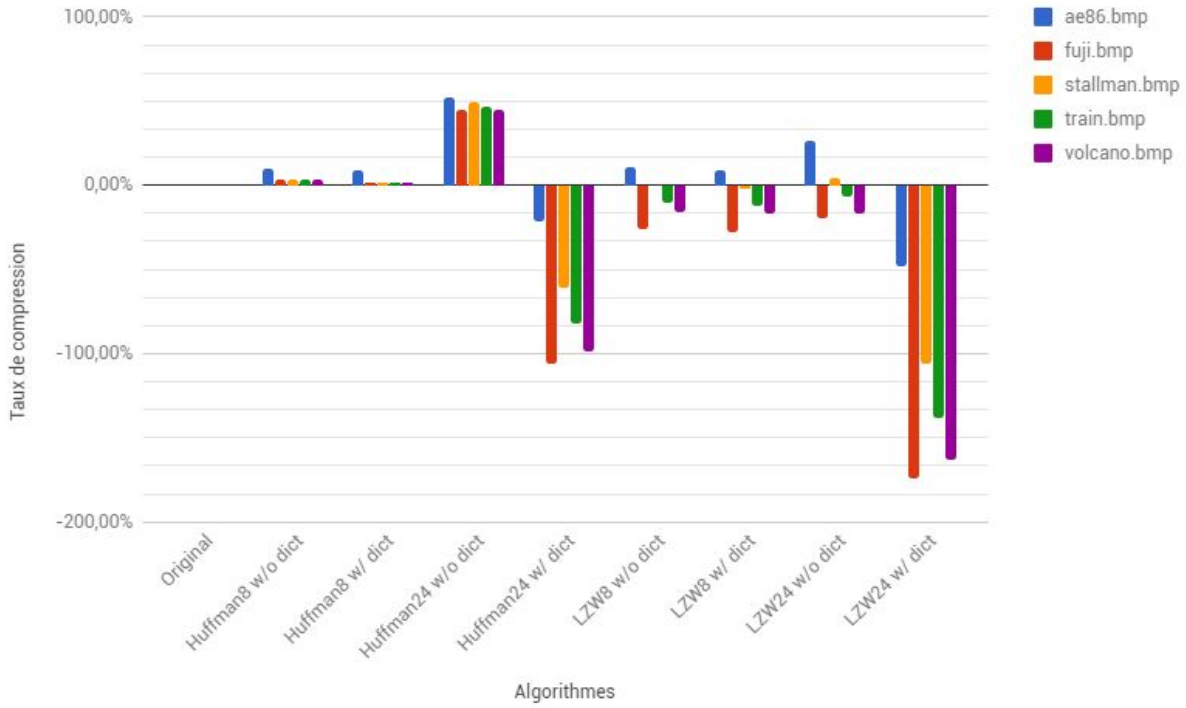
Graphique 1 : Taille des différents codes pour chaque image
(w/o dict : Sans dictionnaire, w/ dict : Avec dictionnaire)



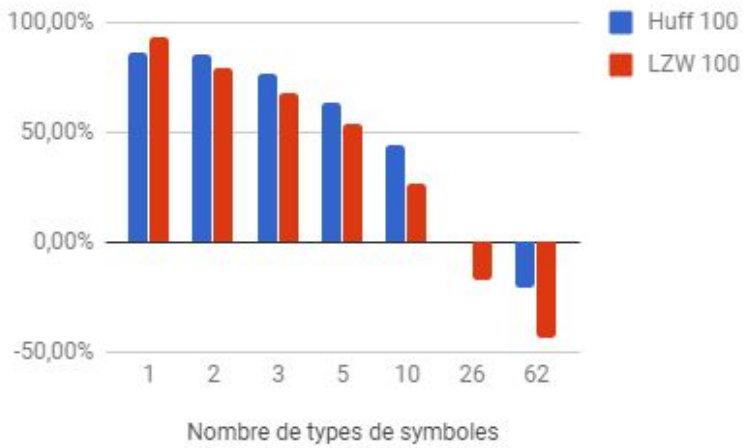
Graphique 2 : Taux de compression de chaque algorithme de codage en fonction de l'image



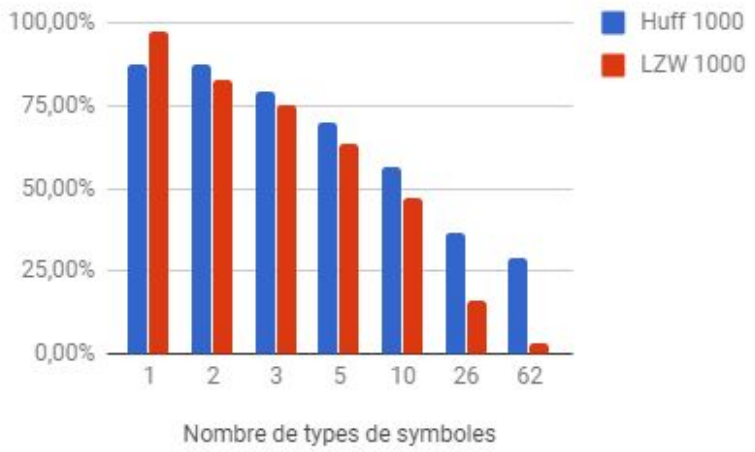
Graphique 3 : Taille des différents codes pour chaque image
(X8 : symboles sur 8 bits, X24 : symboles par triplets RGB 24 bits)



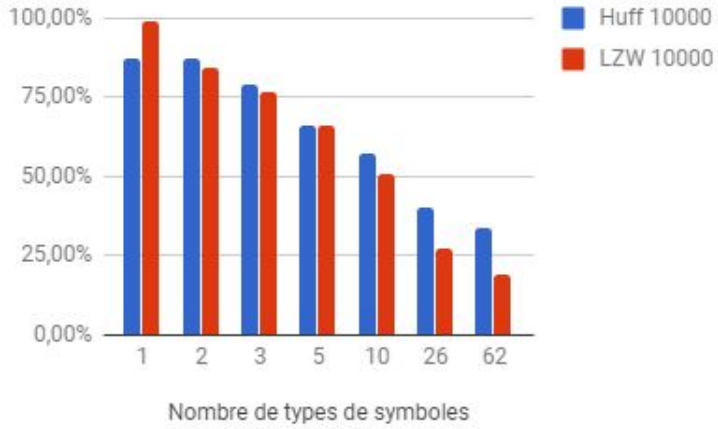
Graphique 4 : Taux de compression de chaque algorithme de codage en fonction de l'image



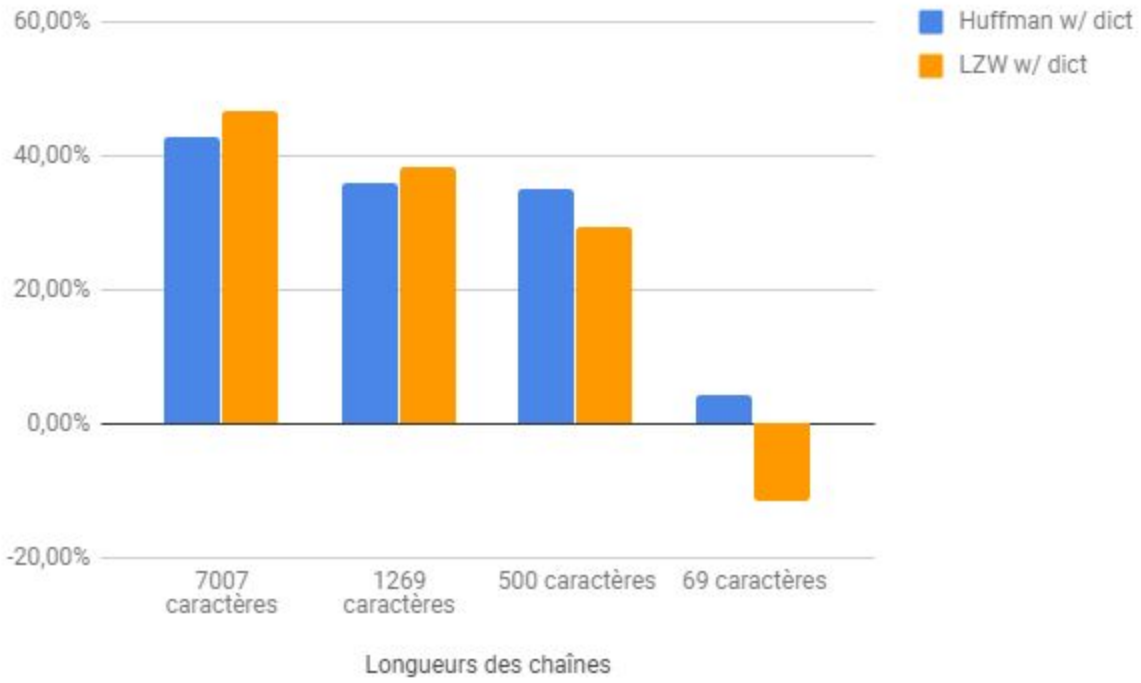
Graphique 5: Taux de compression de Huffman et LZW pour des chaînes de caractères aléatoires de taille 100 selon le nombre de types de symboles



Graphique 6: Taux de compression de Huffman et LZW pour des chaînes de caractères aléatoires de taille 1000 selon le nombre de types de symboles



Graphique 7: Taux de compression de Huffman et LZW pour des chaînes de caractères aléatoires de taille 10000 selon le nombre de types de symboles



Graphique 8: Taux de compression de Huffman et LZW sur des chaînes de caractères en anglais de différentes longueurs

Question 4: Analyse des résultats et retour sur les hypothèses

Afin de procéder à l'analyse des résultats, nous allons passer par chacune des hypothèses que nous avions énoncées :

1. La première hypothèse était que le codage Huffman serait meilleur que LZW lorsque la quantité de types de symboles à coder est petite. Du point de vue des images, on observe que c'est généralement faux : une quantité de couleurs réduite amène généralement (et logiquement) davantage de répétitions, ce qui favorise grandement LZW (taux de compression de ~99% en meilleur cas contre ~90% pour Huffman, avec `white.bmp` et `stripes.bmp`, voir **graphique 2**). Pour ce qui est du texte, le même comportement est observé que dans les images. Lorsque la quantité de symboles est petite, LZW a généralement une meilleure performance que Huffman, encore une fois dû au fait que peu de types de caractères augmente en moyenne la de rencontrer des séquences répétitives lors du codage du message. Le taux de compression des chaînes de caractères générées aléatoirement montre que LZW est plus efficace lorsqu'il y a peu de types de symboles, et que Huffman devient meilleur lorsque le nombre de types de symboles est grand. On peut observer ces résultats dans les **graphiques 5, 6 et 7**.

On peut visuellement constater que plus le nombre de types de symboles augmente, meilleur est Huffman en comparaison avec LZW. Il est à noter que les chaînes de caractères ayant été générées aléatoirement comportent probablement moins de répétitions que des chaînes de caractères issues de certains contextes réels.

2. La seconde hypothèse était que le codage Huffman serait efficace lorsque le message contient des symboles rares, par exemple une image avec peu de bruit parmi des répétitions. On observe effectivement ce comportement sur les images concernées, en particulier ae86.bmp et stallman.bmp. Ces deux images, parmi notre sélection "photographies", sont celles qui ont l'encodage le plus efficace avec Huffman (taux de compression approximativement nul en mode 8 bits avec dictionnaire). Cela s'explique par le nombre réduit de couleurs sur ces images, et l'absence relative de grandes zones de bruit comme l'on retrouve par exemple sur fuji.bmp et volcano.bmp (où le taux de compression est fortement négatif). Dans le cas du texte, l'exécution des algorithmes sur la chaîne de caractères présente dans le fichier `symbolesRares.txt` vient confirmer l'hypothèse. La méthode de **Huffman** donne un taux de compression de **74,66%**, alors que celui de **LZW** est de **38,53%**. La différence des deux taux en question est plus grande que celles observées avec les taux de compression des expériences pour les chaînes générées aléatoirement.

3. La troisième hypothèse était qu'un long message serait plus efficacement encodé avec LZW, car il amène nécessairement plus de répétitions. Puisque nos images sont toutes de la même taille, il faudra se fier uniquement à nos expériences avec les messages sous forme de texte. Si l'on remonte aux **graphiques 5, 6 et 7**, on peut constater que les ratios de LZW semblent augmenter avec la longueur des chaînes. Cela indique qu'il y a une certaine tendance en faveur de notre hypothèse. Par contre, les expériences effectuées avec les textes en anglais donnent des résultats plus concluants. Le **graphique 8** présente les taux de compression des deux algorithmes appliqués sur nos différents textes anglais.

   Les taux de compression de LZW sont supérieurs à ceux de Huffman dans les cas des textes relativement longs de 7007 et 1269 caractères. Graduellement, on remarque que les taux continuent de diminuer avec la taille des chaînes, et que celui de LZW passe sous celui de Huffman entre 1269 et 500 caractères. Évidement, cela dépend de la chaîne elle-même et si elle comporte plus ou moins de répétitions. En effet, l'exécution de LZW sur le texte comportant des répétitions (qui est de longueur semblable au texte de 1269 caractères) donne un taux de compression de 54,07%, ce qui est meilleur que le 46,82% du texte de 7007 caractères. Cela dépend donc vraiment des répétitions contenues dans un message donné. Néanmoins, on peut généraliser et affirmer que selon cette expérience, l'algorithme LZW devient effectivement meilleur si le message à encoder est d'une plus grande taille.

4. La quatrième hypothèse était que LZW serait plus efficace que Huffman pour coder une image de type gradient horizontal (comme `gradientBW.bmp`). C'est effectivement le cas, avec un taux de compression de 50-60% pour LZW contre environ 10% pour Huffman (**graphique 2**). Cette hypothèse étant plutôt spécifique, il n'y a pas vraiment d'autre analyse qui peut s'y appliquer.

5. La cinquième hypothèse était que LZW ne serait pas très efficace pour encoder des images "réelles", en raison de la variation de couleurs et de la taille exagérée du dictionnaire initial.

   Comme on peut le voir sur les **graphiques 3 et 4**, c'est en effet le cas : un codage LZW sur les triplets RGB d'une image, avec son dictionnaire, peut occuper jusqu'à 2,7 fois plus d'espace que l'image originale! Ce cas extrême est observé sur l'image `fuji.bmp`, qui contient énormément de gradients de couleur (le ciel bleu, la montagne, la neige...) mélangés à du bruit (les fleurs de cerisier, la ville au bas de l'image).

   Donc, en plus d'avoir un très grand nombre de symboles distincts, il y a peu de répétitions dans l'image, ce qui provoque une grande inefficacité de l'algorithme LZW. Toutefois, Huffman suit aussi le pas, la taille importante de son dictionnaire amenant jusqu'à -105% comme taux de compression. Sans compter le dictionnaire, on se retrouve avec quelque chose de bien plus raisonnable toutefois (50-60%). LZW appliqué sur 8 bits plutôt que par triplet est beaucoup moins pire, mais produit tout de même un taux de compression négatif pour toutes les images "réelles" à l'exception de `ae86.bmp`, qui contient beaucoup de sections relativement unies et répétées.

   On en déduit donc que la variété de symboles encodé fait grandement varier le taux de compression, mais qu'il est généralement plus efficace d'utiliser un traitement par octet plutôt que par triplet, en raison de la taille importante du dictionnaire initial avec cette dernière méthode.

   Il est par contre intéressant de comparer l'efficacité des algorithmes sans se soucier du dictionnaire : on voit que LZW est légèrement plus efficace pour coder par triplet que par octet, mais que Huffman est environ 2 fois plus efficace de toute façon. Les deux souffrent toutefois de la taille de leur dictionnaire (étonnamment, moins Huffman que LZW).

6. La sixième hypothèse était que pour du texte (message composé de caractères ASCII), Huffman sera largement meilleur que LZW dans le cas d'un message généré aléatoirement. Cela s'avère juste : les résultats obtenus pour Huffman appliqué sur des chaînes de caractères générées aléatoirement sont meilleurs que LZW. Encore une fois, sur les **graphiques 5, 6 et 7**, on remarque que Huffman a un taux de compression supérieur à LZW dans tous les cas, sauf lorsqu'il n'y a qu'un seul type de symbole. En

fait, ce cas n'est pas vraiment important, puisque l'effet de hasard n'est plus présent (il s'agit de 100, 1000 et 10000 fois le même caractère répété).

7. La dernière hypothèse était que dans le cas d'un texte écrit, ce serait plutôt LZW qui aurait de meilleures performances que Huffman. Encore une fois, cela s'avère être le cas, mais seulement lorsque le texte est assez long. Si l'on se fie au **graphique 8**, LZW donne une compression légèrement meilleure que Huffman dans les cas des textes de 7007 et 1269 caractères. Pour des textes plus petits, par exemple d'ordre de 500 et 69 caractères, c'est Huffman qui prend le dessus. Tout comme l'hypothèse 3, il faut y aller cas par cas, puisque certains textes peuvent contenir plus de répétitions que d'autres, ce qui irait alors en faveur de LZW.

Références

- Bilodeau, G.-A. (2018). CodageLZW.m [Logiciel]. Repéré à https://moodle.polymtl.ca/mod/resource/view.php?id=93858

- Gibson, N. (2007). Huffman coding in Python. Repéré à https://www.techrepublic.com/article/huffman-coding-in-python/

- Images variées tirées de Google Images (ae86, fuji, stallman, train et volcano). Tous droits réservés, si applicable. Les images ont été modifiées pour les besoins du travail.