Laboratoire 1

Comparaison et Caractérisation de Méthodes de Codage

Gabriel-Andrew Pollo-Guilbert, *1837776*

Question 1

Hypothèse 1. La méthode de compression arithmétique sera considérablement plus lente que la méthode de compression LZW.

Cette première méthode demande une précision arbitraire de calculs avec des nombres flottants. Les unités de calculs à virgules flottantes des processeurs modernes utilisent la représentation IEEE 754. Hors, cette représentation offre une précision limitée et les calculs contient une petite erreur. Pour remédier au problème, il faut utiliser une librairie d'arithmetique large. Cela dit, ces librairies doivent contourner les limitations de l'unité de calculs IEEE 754. Il devrait donc y avoir une perte considérable de performances.

Hypothèse 2. La méthode de compression par dictionnaire produira une meilleure compression pour les sources textuelles.

Une source textuelle comme un texte anglais ou francais peu contenir beaucoup de patrons et de répétitions. Par exemple, les déterminants «les» ou «de» risquent d'être présent à plusieurs reprises dans un texte francais. Dans le cas du codage par dictionnaire LZW, celui-ci devrait éventuellement construire ces déterminants dans sont dictionnaire et les réutiliser par la suite.

Hypothèse 3. La méthode de compression arithmétique produira une meilleure compression pour les sources ayant une haute entropie.

Lorsque la source est très entropique/aléatoire, il est très peu probable que le codage LZW puisse construire des mots qui vont être réutilisés à plusieurs reprises. Le codage arithmétique ne cherche pas à construire des patrons et de les réutiliser. Il va éventuellement encoder les symbols les plus utilisés avec moins de bits. L'avantage de cette propriété par rapport au dictionnaire est qu'elle est moins dépendante de l'ordre des symbols en entrée.

Question 2

Pour tester la première hypothèse, les types de données ne sont pas importantes, car on assume que les opérations à virgule flottante qui doivent être effectuées ne change pas en fonction des symbols que l'on recoit. Par conséquent, on teste tout simplement les deux algorithmes avec des entrées de taille croissante afin de comparer les résultats.

Afin de tester les deux autres hypothèses, les algorithmes seront testé sur 3 entrées différentes à des tailles différentes. On teste plusieurs entrées binaires et textuelles afin de tester l'ensemble des hypothèses ci-dessus.

Les premières entrées, situées dans le répertoire `test/binary/` sont des fichiers binaires générés aléatoirement avec le générateur de nombre aléatoire sécuritaire du système d'exploitation. La commande suivante génère un fichier aléatoire de 100 octets :

```
$ cat /dev/urandom | head -c 100 > test/binary/100
```

Les deuxièmes entrées, situées dans le répertoire `tests/text/`, sont des fichiers textes latins générés semi-aléatoirement à l'aide du site suivant. Contrairement à une source de lettre totalement aléatoire, cette source contient des mots et des patrons qui sont répétés à plusieurs reprises, ce qui diminue l'entropie des données.

Les troisièmes entrées, situées dans le répertoire `test/letters/`, sont des fichiers textes aléatoires contenant 500 caractères dans un ensemble de lettres restreint. La commande suivante générère 500 caractères aléatoires parmis un alphabet de 10 lettres :

```
$ cat /dev/urandom | tr -cd '[:alnum:]._-' | tr -cd '[abcdefghij]' | head -c 500 >
10.decoded
```

La table 1 suivante présente les fichiers de test, leur taille en octet ainsi que l'entropie en bits par octet.

Table 1. Liste des Fichiers de Tests

Fichier	Taille (octets)	Entropie (bits/octet)
test/binary/1000.decoded	1000	7.804207
test/binary/100.decoded	100	6.236307
test/binary/10.decoded	10	3.321928
test/binary/4000.decoded	4000	7.962127
test/letters/10.decoded	500	3.306265
test/letters/20.decoded	500	4.305818
test/letters/2.decoded	500	0.999815
test/letters/3.decoded	500	1.584437
test/letters/4.decoded	500	1.993291
test/letters/5.decoded	500	2.318703
test/other/pattern.decoded	351	4.447380
test/text/average.decoded	1010	4.191556
test/text/long.decoded	4240	4.184810
test/text/short.decoded	124	4.062426

Question 3

Afin de tester les différentes techniques de codage, deux logiciels furent écris basés sur les exemples suivants du cours : codage arithmétique et codage LZW. Les deux encodeurs ont été écrits en Go en raison de la simplicité du language, de ses bonnes performances et des librairies offertes.

Il important de noter que chacun des programmes opères sur des symboles de la taille d'un octet, donc il y au maximum 256 symboles dans l'alphabet utilisé.

Code Arithmetique

Comme mentionné plus haut, l'encodeur arithmétique doit être en mesure d'effectuer des opérations à virgule flottante d'une précision arbitraire. Pour ce faire, Go offre la librairie `math/big` qui permet d'effectuer principalement des opérations de taille arbitraire sur des entiers `big.Int` et de précision arbitraire sur des nombres à virgule flottante `big.Float`.

Cela dit, cette librairie offre aussi des nombres rationels de taille arbitraire `big.Rat`. Puisque les seuls opérations demandées par l'encodeur sont des additions/soustractions et des multiplications/divisions entières, ce type est plus adapté que `big.Float`, car il ne doit pas gérer les nombres irrationels.

L'encodeur génère une sortie avec une entête afin de pouvoir décoder les données par la suite avec le décodeur. L'entête contient la liste des symbols étant dans le fichier original ainsi que leur occurence. La table 2 suivante montre la disposition du fichier binaire produit par l'encodeur arithmétique.

Table 2. Fichier Binaire Produit (Codage Arithmétique)

Entête		**Données**
Nombre de Symboles	Symboles et Occurences	Nombre à Virgule Flottante
`uvarint`	`[(byte, uvarint), ..]`	`[byte, ..]`

Code LZW

Le codage par dictionnaire génère aussi une sortie avec une entête afin de pouvoir décoder les données générées. Son entête contient la liste des symbols individuels en ordre de première occurence. La table 3 suivante montre la disposition du fichier binaire produict par l'encodeur par dictionnaire.

Table 3. Fichier Binaire Produit (Codage LZW)

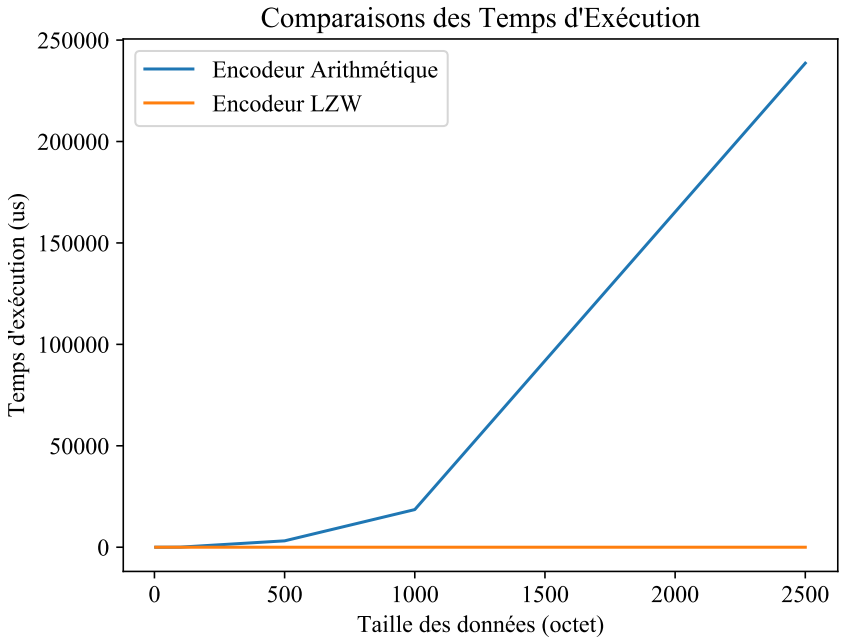
Entête		**Données**
Nombre de Symboles	Symboles	Numéro dans le Dictionnaire
`uvarint`	`[byte, ..]`	`[byte, ..]`

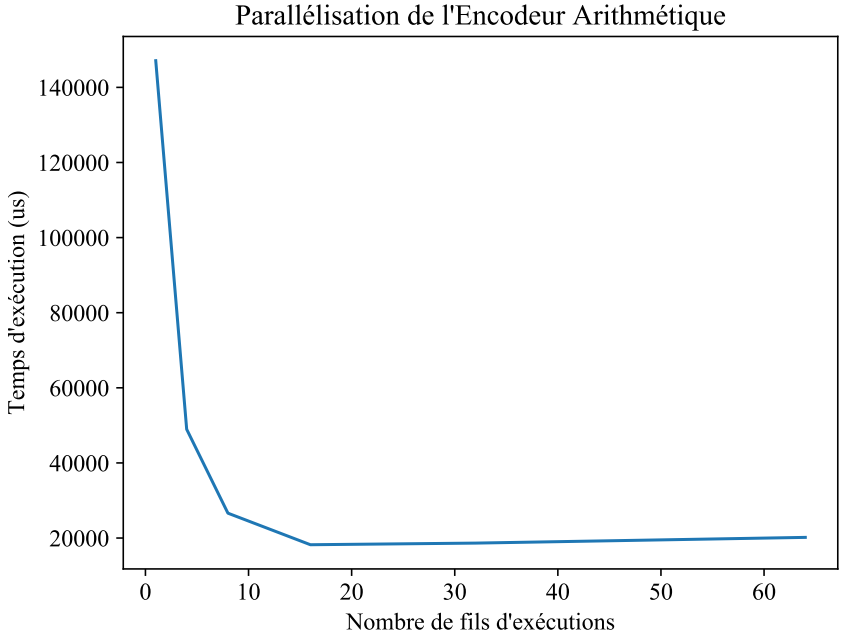
Même si le language Go offre des dictionnaires dans les types de base, le code écrit dans ce laboratoire effectut les recherches des symbols dans un arbre en avancant d'un symbol (octet) à la fois dans le fichier. Ceci est techniquement plus efficace que d'effectuer une recherche d'une chaîne

de caractère (qui doit être hacher au complet à chaque fois) dans un dictionnaire. Cela dit, l'utilisation de l'arbre fut plus un exercise à l'auteur que une optimisation visée.

Question 4

Le prochain graphique montre le temps d'exécution des deux algorithmes en fonction de la taille des données à coder. On peut facilement voir que le codage arithmétique est par ordres de magnitude plus lent que le codage par dictionnaire.





Selon notre hypothèse 1, non seulement le codage arithétique est beaucoup plus lent, mais plus le nombre de données à encoder est grand, de plus en plus le prochain symbole va prendre du temps à encoder. Ceci est dû au fait que l'implémentation des `bit.Rat` est basé sur un tableau d'entier. Plus la précision demandée à `big.Rat` augmente, plus se tableau augmente et plus ses opérations, qui

traversent se tableau, sont lente.

Cela dit, l'opération qui est demandante est celle où les intervals doivent être redimensionnés. Le redimensionnement de chacune des sous-intervals est indépendent l'un de l'autre, alors ce problème est triviallement parralélisable. En effet, le graphique après montre des sérieux gains de performance lorsque le problème est parraléliser sur plusieurs noeuds d'exécution sur un ordinateur 16 coeurs. Malgré cette parallélisation, l'algorithme reste très lent par rapport au codage LZW.

La table suivante montre les résultats de chaque algorithme sur les fichiers tests.

Table 4. Résultats des Fichiers de Tests

Fichier	Originale (octets)	Arithmétique (octets)	Dictionnaire (octets)
`test/binary/1000`	1000	**1484**	1655
`test/binary/100`	100	239	**181**
`test/binary/10`	10	26	**21**
`test/binary/4000`	4000	**4495**	6118
`test/letters/10`	500	**228**	338
`test/letters/20`	500	**311**	447
`test/letters/2`	500	**70**	118
`test/letters/3`	500	**110**	165
`test/letters/4`	500	**137**	206
`test/letters/5`	500	**156**	234
`test/other/pattern`	351	249	**78**
`test/text/average`	1010	**606**	782
`test/text/long`	4240	**2321**	2823
`test/text/short`	124	**108**	120

On remarque que dans certain cas, principalement les sources qui étaient très entropiques, les codes générés sont plus grand que le fichier d'origine. Cela est dû au fait qu'il est très difficile de compresser ces sources là d'avantage et qu'on ajoute un entête qui prend considérablement d'espace pour sauver peu d'espace en premier lieu.

Cela dit, on remarque que le codage arithématique bat dans la majorité des cas le codage par dictionnaire. Dans les cas des fichiers binaires `10` et `100`, il est très probable que les différences proviennent de l'entête du codage arithmétique qui est plus lourde que celle du codage par dictionnaire. Avec l'addition que les fichiers sont petits et qu'ils n'ont déjà pas beaucoup de données à compresser de plus.

À la lumière de ce constant, ces données contredisent notre hypothee 2. En effet, le codage aritmétique semble battre en compression le codage par dictionnaire dans toute les situations, même le texte généré. Il est possible que le texté généré soit encore ici trop entropique pour le codage par dictionnaire. Peut-être qu'une source écrite à la main comme un court blog ou un

chapitre de livre donnerait de meilleurs résultats pour le codage LZW.

Finalement, ses données semblent appuyer notre hypothèse 3 où les sources sont très entropiques. L'exemple le plus évident est le fichier binaire `4000` où le codage arithmétique a sauvé plus de 1500 octets par rapport au codage par dictionnaire.

Avec tous ces résultats, il est difficile de trouver une situation où le codage LZW donnerait des meilleurs résultats que le codage arithmétique. Malheureusement, le codage arithmétique est beaucoup trop lent pour pouvoir être utilisé sérieusement sur des gros fichiers. En connaissant le fonctionnement de l'algorithme LZW, il est facile de générer un entré qui sera facilement compressé. C'est l'exemple du fichier `other/pattern` où le codage LZW bat le codage arithmétique d'une grande marge. Par contre, ce type de fichier est peu représentatif de ce qu'un algorithme de compression devra faire.

Cela dit, l'expérience avec les données textuelles devraient probablement être réassayée avec des textes plus représentatifs d'une langue. Si les performances sont bonne, je crois que les deux algorithmes pourraient être combinées ensemble pour aller chercher quelques performances de plus: le codage LZW reste utilisé pour des données textuelles et le codage arithmétique pour des courtes données binaires. De cette manière, on pourrait parfois obtenir les bonnes performances de compression du codage arithmétique sans toutefois être ralentit sur les plus gros fichiers.