TP de Compression des données

Réalisé par : MULAPI TITA Ketsia

Compression et Décompression avec LZW

1. Concepts de bases

Lors des travaux précédents, nous avions vu le principe classique de Shannon, la méthode BPE et, le codage de Huffman, qui sont des méthodes de compression/décompression efficaces pour des symboles de probabilité $p=2^{-n}$.

L'une des raisons principales pour laquelle nous nous intéressons dans ce cours à **LZW** est qu'il s'agit d'une méthode sans perte de l'information numérique et à base de dictionnaire que nous traitons. Par ailleurs, la rapidité de sa mise en œuvre ne nous garantit nullement son efficacité, en effet d'après nos recherches, il s'avère même que celle-ci n'est pas plus efficace que le célèbre outil **ZIP** que nous utilisons dans nos ordinateurs.

Afin de répondre aux exigences de ce travail, nous avons programmé la méthode LZW proposée par Terry Welch en 1984. Cette méthode s'avère avantageuse et intéressante car elle permet par de faire la compression des séquences (chaînes) qui est de loin meilleure que la compression par symboles.

Dans la suite, nous allons tester les algorithmes de compression et décompression sur notre série de chapitres « OliverTwist » et, le fichier OliverTwist.txt également.

2. La méthode LZW

2.1. Le principe LZW sans remise à 0

On s'intéresse au fait de mettre à jour un dictionnaire de séquences de caractères, qu'on initialise à l'aide des 256 caractères ascii, qui ne sont rien d'autre que toutes les séquences possibles, il s'agit donc de notre dictionnaire de base qui a pour indexe de départ 0. Pour faire de la compression, il suffit de passer une chaine (notre texte) en paramètre, que nous utiliserons pour construire une table de traduction afin de traduire

les chaines de caractères à partir du texte à compresser et, pour la décompression, on se contentera que de reconstruire la même table de chaine de caractères de code identique.

Compression	Décompression
1- Initialiser la chaîne de travail I à nul ***2- On lit un caractère courant x*** ***3- Tant que la chaîne Ix existe dans le dictionnaire I=Ix x = caractère suivant Fin*** ***4- Écrire l'index de I dans le fichier compressé*** ***5- Ajouter Ix dans le dictionnaire I=x*** ***6- aller en 2***	**1- Initialiser le dico avec les caractères** **2- Lire l'index courant dans le fichier compressé** **3- Trouver la chaîne correspondante I dans le dico** **4- l'Écrire dans le fichier décompressé** **5- Lire l'index suivant dans le fichier d'entrée** **6- Trouver la chaîne correspondante J dans le dico** **7- l'écrire dans le fichier de sortie** **8- extraire x le premier caractère de J** **9- sauvegarder la chaîne Ix dans le dico** **10- I=J** 11- **aller en 5**

2.1.1 L'algorithme de compression

Afin d'illustrer, par un exemple simple la quintessence de ce travail, nous allons procéder par un exemple simple afin de rester dans des choses comparables, pour mieux cerner si on compresse beaucoup ou peu par exemple.

Utilisons la chaine de texte suivante, présenté dans le cours :
"**sir sid eastman easily teases sea sick seals**" qui veut dire français "**sir sid eastman taquine facilement les phoques malades de la mer**", mais ici nous ne nous intéressons pas à son sens littérale francophone.

Afin d'assurer une meilleure compression par une méthode à base de dictionnaire, nous pouvons par exemple nous rassurer que les séquences sont bien présente dans le dictionnaire, et que l'absence d'environ 70% des mots du fichier dans le dictionnaire est tolérable et donc à l'inverse au moins 30% devront être présents.

Enfin ici, nous savons que la façon la plus simple pour nous de distinguer une chaine et un index et que, si nous avons 12 bits par index, il nous faudra 2^{12}=4096 différents mots (caractères) et c'est grâce au $13^{ème}$ bit par exemple que nous saurions établir une différence entre un index et un mot. Enfin, notre dictionnaire grandit, au fur et à mesure de la compression.

Voyons maintenant que représente la taille du code compressé, la taille du dictionnaire, le nombre de bits nécessaire pour coder un index dans le dictionnaire avec le nombre de bits déterminé, l'intérêt d'utiliser un index codé sur une plus petite quantité d'octet comme 2 et, sur une plus grande quantité d'octet que 2, à savoir 4 octets.

```
def compression(nom_fichier):
    # liste des 256 caractères ascii
    Dictionnaire = [chr(i) for i in range(256)]
    # liste vide des index entiers
    code = []
    #Charger le fichier source dans le tableau Chaine
    chaine = nom_fichier
    # au départ I est null et t=0
    I = ""
    t = 0
    #Tant que t < longueur(Chaine)
    while (t < len(chaine)):
        x = chaine[t]
        #Tantque Ix existe dans Dictionnaire
        while (I + x in Dictionnaire):
            I=I + x
            t+=1
            x = chaine[int(t)]
        # ajouter l'index de I dans Code
        code.append(Dictionnaire.index(I))
        # ajouter Ix au Dictionnaire
        Dictionnaire.append(I+x)
        I = x
        t = t + 1
    taille_dico = len(Dictionnaire)
    bit_per_index = math.ceil(np.log2(taille_dico))
    #taille_fic_compressed_en_bit = bit_per_index * np.log2(taille_dico)
    taille_fic_compressed_en_bit = len(code) * bit_per_index #bits
    taille_fic_compressed_en_octet = taille_fic_compressed_en_bit/8 #octet
    return code, taille_dico, taille_fic_compressed_en_bit, bit_per_index, taille_fic_compressed_en_octet
```

```
    return texte_decompresse
```

In [29]:
```
# CHARGEMENT du fichier

maChaineTest = "sir sid eastman easily teases sea sick seals"
compression(maChaineTest)
code_compresse, taille_dico, taille_fic_compressed, bit_per_index, taille_fic_compressed_en_octet = compression(maChaineTest)

print(code_compresse)
print("longueur du code compressé : " + str(len(code_compresse)))
print("la taille du dictionnaire : " +str(taille_dico))
print("le nombre de bit total pour tous les indexes du dictionnaire : " + str(bit_per_index))
#print("la taille du fichier compressé en bit : " + str(taille_fic_compressed_en_bit))
print("la taille du fichier compréssé en octet: " + str(taille_fic_compressed_en_octet))
```

```
[115, 105, 114, 32, 256, 100, 32, 101, 97, 115, 116, 109, 97, 110, 262, 264, 105, 108, 121, 32, 116, 263, 115, 101, 115, 259, 2
63, 259, 105, 99, 107, 281, 97, 108, 115]
longueur du code compressé : 35
la taille du dictionnaire : 290
le nombre de bit total pour tous les indexes du dictionnaire : 9
la taille du fichier compréssé en octet: 39.375
```

Et donc pour notre chaine de caractère nous disons que :

- La longueur du code compressé, c'est-à-dire, des différents index du dictionnaire qui sont associé à une valeur composé (mot ou caractères), ont été conservé, tant qu'il y'avait encore du contenu à explorer dans le texte et que cette composition était présente dans le dictionnaire. Ici, cette valeur est de 35.
- La taille du dictionnaire pour cette chaine est de 290 cela signifie que, en plus de notre dictionnaire de base, c'est en compressant qu'une quantité de 34 séquences ont été ajouté en mode append.

- Le nombre de bit total pour coder indexe qui est la quantité nécessaire de bit qu'il nous faut pour coder un index dans ce dictionnaire, est de 9, cela signifie que notre dictionnaire ici, aurait pu atteindre une capacité maximale de 2^9=512 mots (caractères) (soit -1 selon que le dictionnaire débute par 0 ou par 1).

- Enfin la taille du texte compressé en octet est de : 39.375 octets.

- **L'intérêt d'utiliser ou non un index codé sur 2 octets et 4 octets est double, cela nous emmènerai à dire que si l'on avait T, la taille de notre dictionnaire qui coderait chaque index avec log2(T) bit(s), ente 2 et 4 octets nous aurions eu une capacité entre 4 et 16 caractères pour chaque index, ces valeurs vu d'un certain angle peuvent respectivement représenter la borne inférieur et supérieur possible pour coder un index, ainsi, (nous pourrions consommer les capacités de 2 octets pour les caractères les plus fréquents et, celle de 4 octets pour les moins fréquents), tout en sachant que, en plus d'avoir un dictionnaire dynamique, il y'aura accroissement de la taille des codes, notre dictionnaire aurait été flexible, car nous aurions eu des marges bien que cela prendrai un peu plus d'espace. A l'inverse, si on se décide de coder l'index sur une seule capacité fixe de 2 octets par exemple, il est clair que notre dictionnaire aurait eu une taille maximale de 4-1 octets (en partant de 0) et que à chaque, 3ème entrée dans le dictionnaire, lorsque notre dictionnaire grandit, les numéros des codes finiraient par dépasser la limite du nombre de bits pour permette leur écriture et, on sera donc contraint à augmenter le nombre de bits par code.**

Le tableau ci-dessous, renseigne pour chaque fichier Olivertwist les 4 valeurs recherchées, nous tenterons aussi d'établir une observation sur ces derniers. Système d'exploitation utilisé : Windows 10, encodage : utf-8.

Fichiers	Taille de la chaîne	Taille du Code	Taille du dictionnaire	Nombre de bits pour coder l'index	Taille du fichier compressé en octet
OliverTwist	919587	169338	169593	18	381010.5
OTChap1	6194	2548	2803	12	3822.0
OTChap2	23053	7420	7675	13	12057.5
OTChap3	18184	6085	6340	13	9888.125
OTChap4	14577	5050	5305	12	8206.25
OTChap5	23122	7498	7753	13	12184.25
OTChap6	9786	3679	3934	12	5518.5
OTChap7	13549	4798	5023	13	7796.75
OTChap8	17405	5953	6208	13	9673.625
OTChap9	13145	4641	4896	13	7541.625
OTChap10	10630	3892	4147	13	6324.5
OTChap11	15274	5276	5531	13	8573.5
OTChap12	16782	5728	5983	13	9308.0
OTChap13	16539	5745	6000	13	9335.625
OTChap14	21890	7028	7283	13	11420.5
OTChap15	13804	4984	5239	13	8099.0

Observation :

On constate que les chaines ou textes de grandes tailles, requière des grands dictionnaires et que la taille en octet du fichier compressé représente la taille de la chaine, réduit d'au moins 50%, ce qui confère à notre algorithme un caractère de bon compresseur. Enfin, lorsque nous observons le nombre de bits qu'il nous faut pour coder un index, on constate que pour des tailles de dictionnaire petit, on exigera moins de bit, et plus dans le cas contraire.

2.1.1 L’algorithme de décompression

Nous voulons à présent nous rassurer que tout s’est bien passé, pour se faire, essayons de décompresser le code obtenu en 2.1.1.

Notons que ce mécanisme consiste premièrement à cherche la chaîne correspondante dans le dictionnaire et à la concaténer au résultat déjà obtenu. Puis à concaténer la chaîne décodée juste avant avec la première lettre de la chaîne qui vient d'être décodée et on l'ajoute au dictionnaire si c'est un nouveau mot (Wikipédia).

Voici l’algorithme de décompression :

```
def decompression(nom_fichier): #nom_fichier = code_compressé !!!
    # tableau des 256 caractères ascii
    Dictionnaire = [chr(i) for i in range(256)]
    # Charger le fichier compressé dans la liste
    code = nom_fichier
    n_index = 0
    t=0
    # Dictionnaire(Index_courant)
    index_courant = code[n_index]
    #print(Dictionnaire.indexindex_courant))
    I = Dictionnaire.index(index_courant)
    # Ecrire I dans le fichier décompressé  a
    texte_decompresse = [I]
    while n_index < len(code):
        n_index = n_index + 1
        # Si J existe dans le dico
        if code[n_index]<len(Dictionnaire):
            J = Dictionnaire(code[n_index])
            # Ecrire J dans le fichier decompresse
            texte_decompresse.append(J)
            x = J[0]
        else :
        # J n'xiste pas encore dnas le dico car c'est I+I[0]
            x = I[0]
        # ajouter Ix au Dictionnaire
        Dictionnaire.append(I+x)
        I = J
    texte_decompresse.append(I)
    return texte_decompresse
```

```
decompression(code_compresse)
```

```
'sir sid eastman easily teases sea sick seals'
```

Nous avons donc pu retrouver la chaine de départ, et c’est ce qu’il fallait démontrer (Cqfd☺).

2.2. Le principe LZW avec remise à 0

On va à présent s'intéresser à l'impact de la remise à zéro, sur la taille du code compressé et le nombre de dictionnaire utilisés ainsi que sur la taille du code compressé si l'on venait à coder un index sur 12 bits et, avant de faire le constat sur les expériences « with or without reset dictionary ».

2.2.1. Préciser les algorithmes de compression et décompression

Comme vu précédemment, la taille de notre dictionnaire ne peut que augmenter, imaginons donc que nous avions que notre dictionnaire se met à croître de façon exponentielle, il est claire que nous perdrons en qualité, l'efficacité de notre compresseur.

Afin d'éviter cela, nous allons introduire le concept de remise à zéro du dictionnaire :

- Pour l'algorithme de compression, nous allons essayer de limiter la taille du dictionnaire à tmax=4096 soit 2^{12}, afin de ne pas aller au-delà et donc à chaque fois que nous itérons nous allons vérifier si la tmax est atteint, dans le cas où celle-ci et atteinte, nous pouvons utiliser une variable compteur qui incrémente le nombre de dictionnaire et, étant donné que nous ne voulons pas perdre ou détruire le dictionnaire, nous pouvons considérer un méga dico (dictionnaire de dictionnaires), à partir duquel et à l'aide de sa taille que nous pourrions aussi retrouver le nombre de dictionnaire qu'il nous faut. Notons aussi que au moment de la compression, chaque nouveau dictionnaire sera d'abord un dictionnaire de base (avec 256 car).
- Pour la décompression, il suffira alors de désempiler ce méga dictionnaire tout en appliquant, comme vu plus haut, le mécanisme de décodage.

2.2.2. Mise en œuvre de ces 2 algorithmes sur les fichiers OliverTwist

Fichiers	Taille du Code	Nombre de dictionnaire	Taille du fichier en octet avec nombre de bit par index = 12
OliverTwist	339339	89	509008.5
OTChap1	2533	1	3799.5
OTChap2	8703	3	13054.5
OTChap3	6816	2	10224.0
OTChap4	5587	2	8380.5
OTChap5	8847	3	13270.5
OTChap6	3655	1	5482.5
OTChap7	5255	2	7882.5
OTChap8	6612	2	9918.0
OTChap9	5038	2	7557.0
OTChap10	3873	2	5809.5
OTChap11	5874	2	8811.0
OTChap12	6376	2	9564.0
OTChap13	6391	2	9586.5
OTChap14	8124	3	12186.0
OTChap15	5483	2	8224.5

2.2.3. Comparaison entre les valeurs obtenu avec et sans remise à zero

Premièrement, il est trivial de dire qu'il nous faudrait plus d'un dictionnaire lors de la remise à zéro que sans remise à zéro, à partir du moment où la taille du code est supérieur à 4096. Ensuite, le fait d'avoir plusieurs dictionnaire rend la tâche de décompression moins facile et, alors que en majorité la taille du code dans le mécanisme de remise à zéro parais plus grand que sans remise à zéro, proportionnellement, lorsque nous codons sur 12 bits, la taille du fichier compressé est aussi supérieur à la taille du fichier compressé sans remise à zéro, d'où, la remise à zéro avec ou sans destruction du dictionnaire est plus couteuse que la compression et décompression sans remise à zéro.

3. Conclusion

Somme toute, le présent travail consistait à implémenter, utiliser et examiner l'algorithme de compression sans perte LZW, nous avons ainsi évalué la taille d'un code compressé, la taille de la chaine, nous avons vu la différence et/ou les conséquences entre coder un index avec une petite quantité de bit et une grande quantité et, après avoir effectué le mécanisme de remise à zéro, nous avons eu, de façon brève et claire, une idée des différentes liens qui pourraient exister entre les différentes caractéristiques des chaines analysés.