INF365 - Projet de Théorie de l'Information

17401761 - Mustafa Batuhan Ceylan

I. Structure de Fichier

Le projet a créé une nouvelle extension de fichier propriétaire appelée «.froz», qui est utilisée pour stocker les données encodées avec ses informations pertinentes pour le décodage. L'état initial du fichier est illisible jusqu'à ce que la phase de correction d'erreur se produise et révèle les octets comme :

- Taille du Dictionnaire [4 octets]
- Dictionnaire [(Taille du Dictionnaire) octets]
- Taille des Données Codées [4 octets]
- Remplissage des Données Codées [1 octet]
- Données Codées [(Taille des Données Codées) octets]

II. Compression Sans Perte

A. Encodage

J'ai décidé d'utiliser l'encodage Huffman comme méthode de choix pour la compression sans perte. Dans cette méthode, la fréquence de chaque caractère possible est analysée et leur fréquence est ensuite utilisée pour créer un arbre pour trouver les représentations binaires de chaque caractère. En raison de la métrique utilisée dans la construction de l'arbre, les caractères les plus fréquents ont les codes les plus courts. L'arbre de Huffman est généré de manière qu'un code donné n'est pas un préfixe d'un autre code. Cette caractéristique de l'arbre est dû au fait que les caractères ne peuvent être attribués à des noeuds qui ont pas de nœuds enfants de leur propre.
Après l'arbre de Huffman et le dictionnaire pertinent est créé, l'algorithme commence le codage. J'ai ajouté la possibilité d'encoder les données de manière asynchrone à l'aide de la bibliothèque de Python appelée **multiprocessing**, en particulier sa structure de données **Pool**:

```
def _encode_job(data: str, tree: Dict[str, str]) -> str:
    '''Worker function for multiproccessed encoding'''

    return ''.join([tree[c] for c in data])

#...

class FrozCompressor(object):
    '''Data compressor class'''
    #...
    def encode(self, jobs: int = 1) -> None:
        '''Encodes data on demand.
        Doesn't need to be called if plaintext data is read
through (read_string) or (read_file)'''
```

```
        # Split data
        n = len(self.__data) // jobs
        split_data = [self.__data[i: i + n]
                      for i in range(0, len(self.__data), n)]

        shards = [(shard, self.huffmanCode) for shard in
split_data]
        # MP
        with Pool(jobs) as p:
            res = p.starmap_async(_encode_job, shards)
            res = res.get()

        self.__encoded_data = res
    #...
```

À la fin du processus, **__encoded_data** est une liste de chaînes qui seront concaténées en données d'octets uniformes pendant la phase d'écriture.

Lors de l'écriture de ces données dans un fichier, la longueur de la chaîne de bits peut ne pas être un multiple de 8, donc pour écrire des octets exacts, le reste de la chaîne de bits est rempli de zéros et la quantité de remplissage est stockée dans la classe **FrozCompressor** qui écrira la taille de remplissage dans un fichier lorsque la sortie est enregistrée en tant que fichier.

B. Décodage

En raison de la complexité accrue et du manque de fiabilité causés par le stockage de morceaux de données séparés dans les fichiers «.froz», j'ai décidé de faire le décodage en un seul processus par opposition à l'encodage, même si le programme pouvait décoder des morceaux sans problème lorsqu'ils étaient représentés sous forme d'objets **bytes** de Python.

Le décodage est effectué sur les données après la suppression du remplissage et les données d'octet sont reconverties en une chaîne de bits. Il existe un moyen plus rapide de décoder directement sur les octets en utilisant la manipulation de bits, mais l'utilisation de chaînes de bits me donne un algorithme beaucoup plus simple à comprendre.

La chaîne de bits est introduite dans l'algorithme de décodage qui commence par une chaîne vide comme jeton et y ajoute la valeur de bit suivante, puis vérifie si la valeur de jeton actuelle se trouve dans le dictionnaire Huffman. Si c'est le cas, il écrit le caractère correspondant dans la chaîne de sortie et réinitialise la chaîne de jeton. Cette boucle se poursuit jusqu'à ce qu'il n'y ait plus de données à traiter et renvoie la sortie finale. Ce processus est très similaire à la façon dont les

compilateurs analysent le code source. Cependant, ce processus fonctionne sur une base de caractères au lieu de caractères d'échappement.

Le travail de décodage et la méthode de décodage de la classe **FrozCompressor**:

```
def _decode_job(encoded_data: str, tree: Dict[str, str]) -> str:
    '''Worker function for multiprocessed decoding'''

    token = ""
    decoded_data = ""

    for c in encoded_data:
        token += c

        # Lookup
        converted_value = tree.get(token)

        if converted_value:
            decoded_data += converted_value
            token = ""  # Reset token

    return decoded_data

#...

class FrozCompressor(object):
    '''Data compressor class'''

    #...

    def decode(self) -> None:
        '''Decodes data on demand.
        Doesn't need to be called if encoded data is read through
(read_string) or (read_file)'''

        inv_tree = {value: key for key, value in
self.huffmanCode.items()}

        shards = [(shard, inv_tree) for shard in
self.__encoded_data]

        with Pool(len(self.__encoded_data)) as p:
```

```
            res = p.starmap_async(_decode_job, shards)
            res = res.get()

        self.__decoded_data = res
    #...
```

III. Filtrage avec Perte

La compression avec perte de texte n'est pas aussi simple que dans d'autres domaines de compression. Étant donné que la perte de données a un impact accru sur le résultat final par rapport à l'audio ou aux graphiques, les méthodes de compression de texte avec perte sont presque impossibles à accepter dans un environnement réel. Les méthodes possibles de compression de texte avec perte peuvent varier, mais celles qui fournissent les meilleurs résultats s'attendent à ce que l'entrée donnée provienne d'un texte de langue prédéterminé, ce qui permet de supprimer les mots et les syllabes inutiles selon les règles de la langue donnée.

Cependant, je voulais créer un algorithme de compression universel afin que la meilleure façon possible de mettre en œuvre la perte de données pendant la compression était d'utiliser des filtres avec perte. Les filtres avec perte prennent l'entrée et suppriment certaines données selon un algorithme. Dans ce projet, j'ai décidé d'utiliser un filtre de suppression de voyelle pour supprimer chaque voyelle du texte d'entrée. Néanmoins, le code du programme peut accueillir plusieurs filtres personnalisés qui peuvent être ajoutés facilement :

```
def _vowel_filter(data: str) -> str:
    '''Removes vowels from input data'''

    vowels_dict = dict.fromkeys('aeıioöuüAEIİOÖUÜ')
    translation_table = str.maketrans(vowels_dict)
    return data.translate(translation_table)



SWITCH = {
    'vowel_removal': _vowel_filter,
}
```

Pendant la phase de lecture du fichier, si la méthode de filtrage avec perte est fournie, la fonction nécessaire est récupérée dans le dictionnaire **SWITCH** et utilisée avec les données d'entrée.

Malheureusement, il n'y a aucun moyen réel de récupérer les voyelles après leur suppression, donc le texte final perdra des informations.

IV. Codes de Correction d'Erreur

Au cours du processus de développement, j'avais deux options pour implémenter la correction d'erreur. Soit uniquement sur les données encodées, soit sur les données complètes elles-mêmes. L'ajout d'un code de correction d'erreurs uniquement aux données codées signifierait qu'une corruption à tout autre point du fichier, les informations de taille, par exemple, endommagerait les données. J'ai donc décidé d'ajouter des codes de correction d'erreur à toutes les données écrites dans le fichier. Il y avait encore deux choix à faire. Je pourrais traiter chaque partie comme la sienne et ajouter des codes en conséquence ou rassembler toutes les données d'octet dans un objet **bytes** et y ajouter les codes de correction d'erreurs. J'ai opté pour la deuxième option, ce qui signifiait que soit on corrigeait tout le fichier, soit on perdait toutes les informations. Mais il s'agissait plus d'une mise en œuvre simple.

Au moment de l'implémentation, j'ai d'abord écrit mon propre code de Hamming, mais la quantité de redondance dont j'avais besoin a rendu le code de Hamming tellement plus gros. Ce qui signifiait que les fichiers compressés étaient gonflés au lieu d'être plus petits que le fichier d'origine.

J'ai ensuite cherché une solution plus rapide et plus efficace et suis tombé sur des codes Reed-Solomon. Quelles sont la famille de codes de correction d'erreurs utilisés dans les codes QR pour corriger les erreurs possibles causées par l'impression et les angles de caméra, etc.
Cependant, l'énorme complexité et la taille de l'implémentation signifiaient que je n'avais plus le temps d'implémenter ce code par moi-même.
J'ai décidé d'utiliser une bibliothèque créée par Tomer Filiba pour Python. Son code a été écrit en Python pur pour éviter toute dépendance, mais je voulais plus de performances, j'ai donc ajouté des **tableaux NumPy** et leurs propres fonctions plus rapides au code source où les tests principaux sont effectués à l'aide de la méthode **Sieve of Eratosthenes**. Je voulais remplacer toutes les listes et les tableaux par des **tableaux NumPy**, mais l'implémentation d'origine utilisait une structure **bytearray**, ce qui signifiait que la conversion nécessiterait une réécriture de la majorité des calculs du **champ de Galois**.

J'avais créé un objet **FrozFileHandle** pour gérer les lectures et les écritures dans les fichiers «.froz» et «.txt» et j'y avais ajouté la classe Reed-Solomon. J'ai utilisé la valeur de 16 octets en initialisation dans l'implémentation principale, ce qui signifiait que pour chaque 255 octets de données, je pouvais corriger 8 erreurs ou 16 effacements.

V. Bilan

A. Sans perte

```
File Name: test_huge.txt
```

```
Original File Size (bytes): 59875
Shannon Entropy: 4.620224401040314
Theoretical Minimum Compression (bytes): 34579.4920015361
Theoretical Minimum Compression (bytes&ceil): 37421.875
Compressed Data Size (bytes): 35288
Compression Ratio: 1.6967524370890954
Compression Efficiency: 71.10283748213476%

File Name: test_big.txt
Original File Size (bytes): 15188
Shannon Entropy: 4.298228365271916
Theoretical Minimum Compression (bytes): 8160.186551468732
Theoretical Minimum Compression (bytes&ceil): 9492.5
Compressed Data Size (bytes): 9787
Compression Ratio: 1.551854500868499
Compression Efficiency: 66.18721233803028%

File Name: test_small.txt
Original File Size (bytes): 2964
Shannon Entropy: 4.227078893029531
Theoretical Minimum Compression (bytes): 1566.1327298674412
Theoretical Minimum Compression (bytes&ceil): 1852.5
Compressed Data Size (bytes): 2321
Compression Ratio: 1.2770357604480826
Compression Efficiency: 41.05654570251042%

File Name: test_tiny.txt
Original File Size (bytes): 12
Shannon Entropy: 3.0220552088742005
Theoretical Minimum Compression (bytes): 4.533082813311301
Theoretical Minimum Compression (bytes&ceil): 6.0
Compressed Data Size (bytes): 140
Compression Ratio: 0.08571428571428572
Compression Efficiency: -2823.6854536195706%
```

B. Perte

Dans les tests de compression avec perte, j'ai gardé les médiums théoriques pour la compression sans perte pour voir la différence.

```
File Name: test_huge.txt
Original File Size (bytes): 59875
Shannon Entropy: 4.620224401040314
```

```
Theoretical Minimum Compression (bytes): 34579.4920015361
Theoretical Minimum Compression (bytes&ceil): 37421.875
Compressed Data Size (bytes): 24063
Compression Ratio: 2.4882599842081206
Compression Efficiency: 103.56427445032782%

File Name: test_big.txt
Original File Size (bytes): 15188
Shannon Entropy: 4.298228365271916
Theoretical Minimum Compression (bytes): 8160.186551468732
Theoretical Minimum Compression (bytes&ceil): 9492.5
Compressed Data Size (bytes): 6405
Compression Ratio: 2.371272443403591
Compression Efficiency: 107.63234326326973%

File Name: test_small.txt
Original File Size (bytes): 2964
Shannon Entropy: 4.227078893029531
Theoretical Minimum Compression (bytes): 1566.1327298674412
Theoretical Minimum Compression (bytes&ceil): 1852.5
Compressed Data Size (bytes): 1435
Compression Ratio: 2.0655052264808362
Compression Efficiency: 97.62901769694933%

File Name: test_tiny.txt
Original File Size (bytes): 12
Shannon Entropy: 3.0220552088742005
Theoretical Minimum Compression (bytes): 4.533082813311301
Theoretical Minimum Compression (bytes&ceil): 6.0
Compressed Data Size (bytes): 113
Compression Ratio: 0.10619469026548672
Compression Efficiency: -2228.0643032466924%
```

VI. Références Sources

- https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/
- https://www.programiz.com/dsa/huffman-coding
- https://en.wikipedia.org/wiki/Huffman_coding
- https://github.com/tomerfiliba/reedsolomon
- https://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html

- https://en.wikiversity.org/wiki/Reed%E2%80%93Solomon_codes_for_coders
- https://stackoverflow.com/questions/32675679/convert-binary-string-to-bytearray-in-python-3
- https://stackoverflow.com/questions/8815592/convert-bytes-to-bits-in-python