République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Mohammed Seddik BenYahia -Jijel-

Faculté Des Sciences Exactes et Informatique

Département d'Informatique

Specialite : M1 ILM

Rapport de TP4 TNMI

Implémentation et comparaison de codeurs VLC (Huffman et Shannon-Fano)

- Birouk Mohammed Islam
- Chettab Mohcine

L'Encadreur :

- Zahir Mahrouk

1. Introduction

La compression de données joue un rôle essentiel dans la réduction de l'espace nécessaire pour stocker ou transmettre des données. Deux des algorithmes de compression sans perte les plus utilisés sont le codage Huffman et le codage Shannon-Fano. Les deux algorithmes ont pour objectif d'assigner des codes de longueur variable aux symboles en fonction de leurs fréquences, afin de minimiser la taille totale des données encodées. Bien que les deux méthodes partagent des similarités, leurs approches pour attribuer les longueurs des codes sont distinctes.

2. Aperçu des Méthodes

2.1 Codage Huffman

Le codage Huffman est une méthode populaire et efficace de compression sans perte. Il fonctionne en attribuant des codes plus courts aux symboles les plus fréquents et des codes plus longs aux symboles moins fréquents, avec pour objectif de minimiser la longueur totale du message encodé.

Le processus consiste à construire un arbre binaire appelé **arbre de Huffman** où les feuilles représentent les symboles. L'algorithme commence par attribuer des fréquences aux symboles et fusionne de manière répétée les deux symboles ou groupes de symboles les moins fréquents jusqu'à ce que tous les symboles soient représentés dans un seul arbre. Cette méthode garantit un code optimal sans préfixe, c'est-à-dire qu'aucun code n'est un préfixe d'un autre.

2.2 Codage Shannon-Fano

Le codage Shannon-Fano est une méthode de construction d'un code binaire sans préfixe basé sur les fréquences des symboles. Comme le codage Huffman, le codage Shannon-Fano attribue des codes plus courts aux symboles plus fréquents. Cependant, contrairement à Huffman, Shannon-Fano divise de manière récursive les symboles en deux groupes en fonction de leurs probabilités totales (ou fréquences) et

attribue "0" et "1" aux deux groupes. Le processus récursif se poursuit jusqu'à ce que tous les symboles aient un code.

Bien que le codage Shannon-Fano soit plus simple et plus intuitif que le codage Huffman, il n'est pas garanti d'offrir la solution optimale dans tous les cas, c'est pourquoi Huffman est généralement préféré en pratique.

3. Bibliothèques Utilisées

3.1 `math`

Le module `math` fournit des fonctions mathématiques utilisées pour calculer l'entropie dans les deux algorithmes. La fonction `math.log2()` est particulièrement utilisée pour calculer les logarithmes en base 2 nécessaires à cette mesure.

3.2 `collections.Counter`

La classe `Counter` du module `collections` permet de compter facilement la fréquence de chaque caractère dans un texte. Elle est utilisée pour obtenir les probabilités des symboles, essentielles pour les algorithmes de codage.

3.3 `heapq`

Le module `heapq` est utilisé dans le codage Huffman pour gérer une file de priorité (tas) afin de fusionner efficacement les nœuds les moins fréquents et construire l'arbre de Huffman.

4. Explication Détail des Méthodes dans les Codes

4.1 Explication des Méthodes du Codage Huffman

4.1.1 `calculate_frequencies(text)`

```python
def calculate_frequencies(text):  1 usage
    freq_count = Counter(text)
    total_chars = len(text)
    chars = []
    frequencies = []

    for char, count in freq_count.items():
        chars.append(char)
        frequencies.append(count / total_chars)

    return chars, frequencies
```

Cette méthode calcule la fréquence des symboles dans le texte d'entrée. Elle utilise `Counter` pour compter le nombre d'occurrences de chaque caractère et retourne les caractères et leurs fréquences relatives par rapport à la longueur totale du texte.

4.1.2 `calculate_entropy(frequencies)`

```python
def calculate_entropy(frequencies):  1 usage
    return -sum(p * math.log2(p) for p in frequencies if p > 0)
```

Cette méthode calcule l'entropie du texte. L'entropie est une mesure de l'incertitude ou de l'imprévisibilité du texte. Elle est calculée en appliquant la formule `-Σ(p * log2(p))`, où `p` est la probabilité (ou fréquence) d'un symbole.

4.1.3 `calculate_compression_metrics(text, huffman_codes, frequencies)`

```python
def calculate_compression_metrics(text, huffman_codes, frequencies):  1 usage
    original_size = len(text) * 8  # ASCII (8 bits per character)
    compressed_size = sum(len(code) * freq * len(text)
                          for (code), freq in zip(huffman_codes.items(), frequencies))
    table_size = sum(8 + len(code) for code in huffman_codes.values())
    compression_rate = original_size / compressed_size
    compression_rate_with_table = original_size / (compressed_size + table_size)
    avg_code_length = sum(len(code) * freq for (code), freq in zip(huffman_codes.items(), frequencies))

    return {
        'avg_code_length': avg_code_length,
        'original_size': original_size,
        'compressed_size': compressed_size,
        'table_size': table_size,
        'compression_rate': compression_rate,
        'compression_rate_with_table': compression_rate_with_table,

    }
```

Cette méthode calcule diverses métriques liées à la compression des données. Elle calcule la taille originale des données, la taille après compression, la taille de la table des codes, le taux de compression et la longueur moyenne des codes, en utilisant les codes de Huffman et les fréquences des symboles.

4.1.4 `build_huffman_tree(chars, freq)`

```python
def build_huffman_tree(chars, freq):  1 usage
    priority_queue = [Node(char, f) for char, f in zip(chars, freq)]
    heapq.heapify(priority_queue)

    while len(priority_queue) > 1:
        left_child = heapq.heappop(priority_queue)
        right_child = heapq.heappop(priority_queue)
        merged_node = Node(frequency=left_child.frequency + right_child.frequency)
        merged_node.left = left_child
        merged_node.right = right_child
        heapq.heappush( *args: priority_queue, merged_node)

    return priority_queue[0]
```

Cette méthode construit l'arbre de Huffman à l'aide d'une file de priorité (`heapq`). Elle crée des nœuds pour chaque caractère avec sa fréquence, puis fusionne les deux nœuds les moins fréquents pour créer un nouvel arbre jusqu'à ce qu'il ne reste qu'un seul nœud, la racine de l'arbre de Huffman.

4.1.5 `generate_huffman_codes(node, code="", huffman_codes=None)`

```python
def generate_huffman_codes(node, code="", huffman_codes=None):  3 usages
    if huffman_codes is None:
        huffman_codes = {}

    if node is not None:
        if node.symbol is not None:
            huffman_codes[node.symbol] = code
        generate_huffman_codes(node.left, code + "1", huffman_codes)
        generate_huffman_codes(node.right, code + "0", huffman_codes)
    return huffman_codes
```

Cette méthode génère les codes de Huffman en parcourant l'arbre de Huffman. Elle attribue un code binaire à chaque caractère, en ajoutant "0" pour aller à gauche et "1" pour aller à droite dans l'arbre. Elle retourne un dictionnaire contenant les codes de Huffman pour chaque symbole.

4.1.6 `huffman_decode(encoded_text, huffman_codes)`

```python
def huffman_decode(encoded_text, huffman_codes):  1 usage
    reverse_mapping = {code: char for char, code in huffman_codes.items()}
    decoded = ""
    current_code = ""

    for bit in encoded_text:
        current_code += bit
        if current_code in reverse_mapping:
            decoded += reverse_mapping[current_code]
            current_code = ""

    return decoded
```

Cette méthode décode un texte encodé en Huffman. Elle utilise un dictionnaire inverse des codes de Huffman pour convertir les séquences de bits en leurs symboles d'origine.

4.1.7 `process_huffman_encoding(text)`

```python
def process_huffman_encoding(text):  2 usages
    chars, freq = calculate_frequencies(text)
    entropy = calculate_entropy(freq)
    root = build_huffman_tree(chars, freq)
    huffman_codes = generate_huffman_codes(root)
    metrics = calculate_compression_metrics(text, huffman_codes, freq)
    encoded_text = "".join(huffman_codes[char] for char in text)
    decoded_text = huffman_decode(encoded_text, huffman_codes)

    return {
        'coded_message': encoded_text,
        'decoded_message': decoded_text,
        'huffman_codes': huffman_codes,
        'entropy': round(entropy, 2),
        **metrics,
        'successful_decode': text == decoded_text
    }
```

Cette méthode est la fonction principale qui intègre toutes les étapes du codage Huffman. Elle calcule les fréquences, l'entropie, construit l'arbre de Huffman, génère les codes, encode et décode le texte, puis calcule les métriques de compression.

4.2 Explication des Méthodes du Codage Shannon-Fano

4.2.1 shannon(l, h, p)

```
def shannon(l, h, p):  3 usages
    if l >= h:
        return
    pack1 = sum(p[i].pro for i in range(l, h))
    pack2 = p[h].pro
    diff1 = abs(pack1 - pack2)

    j = h - 1
    while j > l:
        pack1 -= p[j].pro
        pack2 += p[j].pro
        diff2 = abs(pack1 - pack2)
        if diff2 >= diff1:
            break
        diff1 = diff2
        j -= 1

    for i in range(l, j + 1):
        p[i].top += 1
        p[i].arr[p[i].top] = 1

    for i in range(j + 1, h + 1):
        p[i].top += 1
        p[i].arr[p[i].top] = 0
```

Cette méthode est utilisée pour construire l'arbre de Shannon-Fano en divisant récursivement les symboles en deux groupes ayant des probabilités totales aussi équilibrées que possible. Elle attribue des "0" et "1" aux symboles en fonction de leur groupe, puis continue récursivement jusqu'à ce que chaque symbole ait un code unique.

4.2.2 `calculate_entropy(p)`

```
def calculate_entropy(p):  1 usage
    return -sum(node.pro * math.log2(node.pro) for node in p if node.pro > 0)
```

Cette méthode calcule l'entropie de la source de données à partir des probabilités des symboles. Elle applique la formule `-Σ(p * log2(p))`, où `p` est la probabilité d'un symbole.

4.2.3 `calculate_compressed_size(p, total_length)`

```
def calculate_compressed_size(p, total_length):  1 usage
    return sum((node.top + 1) * node.pro * total_length for node in p)
```

Cette méthode calcule la taille compressée des données en fonction des codes de Shannon-Fano et des probabilités des symboles. Elle multiplie la longueur du code pour chaque symbole par sa probabilité et la longueur totale des données.

4.2.4 `calculate_table_size(p)`

```
def calculate_table_size(p):  1 usage
    return sum(len(node.sym) * 8 + (node.top + 1) for node in p)
```

Cette méthode calcule la taille de la table des codes de Shannon-Fano. Elle prend en compte la longueur de chaque code et la longueur des symboles dans la table de correspondance.

4.2.5 `sortByProbability(p)`

```
def sortByProbability(p):  1 usage
    p.sort(key=lambda node: node.pro)
```

Cette méthode trie les symboles en fonction de leurs probabilités, de la plus faible à la plus élevée. Cela permet de structurer les symboles avant de commencer à les diviser en groupes dans l'algorithme de Shannon-Fano.

4.2.6 `decode_shannon_fano(coded_message, code_dict)`

```python
def decode_shannon_fano(coded_message, code_dict):  1 usage
    decoded_message = ""
    buffer = ""
    inverse_code_dict = {v: k for k, v in code_dict.items()}

    for bit in coded_message:
        buffer += bit
        if buffer in inverse_code_dict:
            decoded_message += inverse_code_dict[buffer]
            buffer = ""
    return decoded_message
```

Cette méthode décode un message codé selon Shannon-Fano. Elle parcourt le texte codé et utilise le dictionnaire des codes pour reconstruire le texte d'origine.

4.2.7 `process_shannon_fano(test_word)`

Cette méthode est l'intégration de tout le processus de codage Shannon-Fano. Elle calcule les fréquences, trie les symboles, effectue le codage et le décodage, puis calcule les métriques de compression.

5-Exécution et affichage des résultats :

Dans cette section, j'ai effectué l'exécution des algorithmes de codage Huffman et Shannon-Fano sur plusieurs exemples de mots afin d'analyser leur performance en termes d'efficacité de l'entropie, de taux de compression et de taille des données comprimées. Pour cette démonstration, j'ai choisi trois mots spécifiques : **"mohcine"**, **"message"**, et **"hello"**. Ces mots ont été soumis aux deux algorithmes, et les résultats obtenus pour chaque mot seront présentés ci-dessous. Les résultats incluent les valeurs d'efficacité de l'entropie, les taux de compression, ainsi que les tailles des données avant et après compression.

5.1. Résultats pour le mot "mohcine" (Shannon-Fano)

Shannon-Fano Results

Shannon-Fano Results

Character Frequencies

['m': 1], ['o': 1], ['h': 1], ['c': 1], ['i': 1], ['n': 1], ['e': 1]

coded_message	11111010110001101000
decoded_message	mohcine
shanon-fano-codes	{'m': '111', 'o': '110', 'h': '101', 'c': '100', 'i': '011', 'n': '010', 'e': '00'}
entropy	2.81
avg_code_length	2.86
original_size	56
compressed_size	20.0
table_size	76
compression_rate	2.8
compression_rate_with_table	0.58
successful_decode	True
theoretical_entropy	2.807354922057604
entropy_efficiency	1.0187525551289434

5.2. **Résultats pour le mot "mohcine" (huffman)**

5.3. **Résultats pour le mot "hello" (Shannon-Fano)**

Shannon-Fano Results

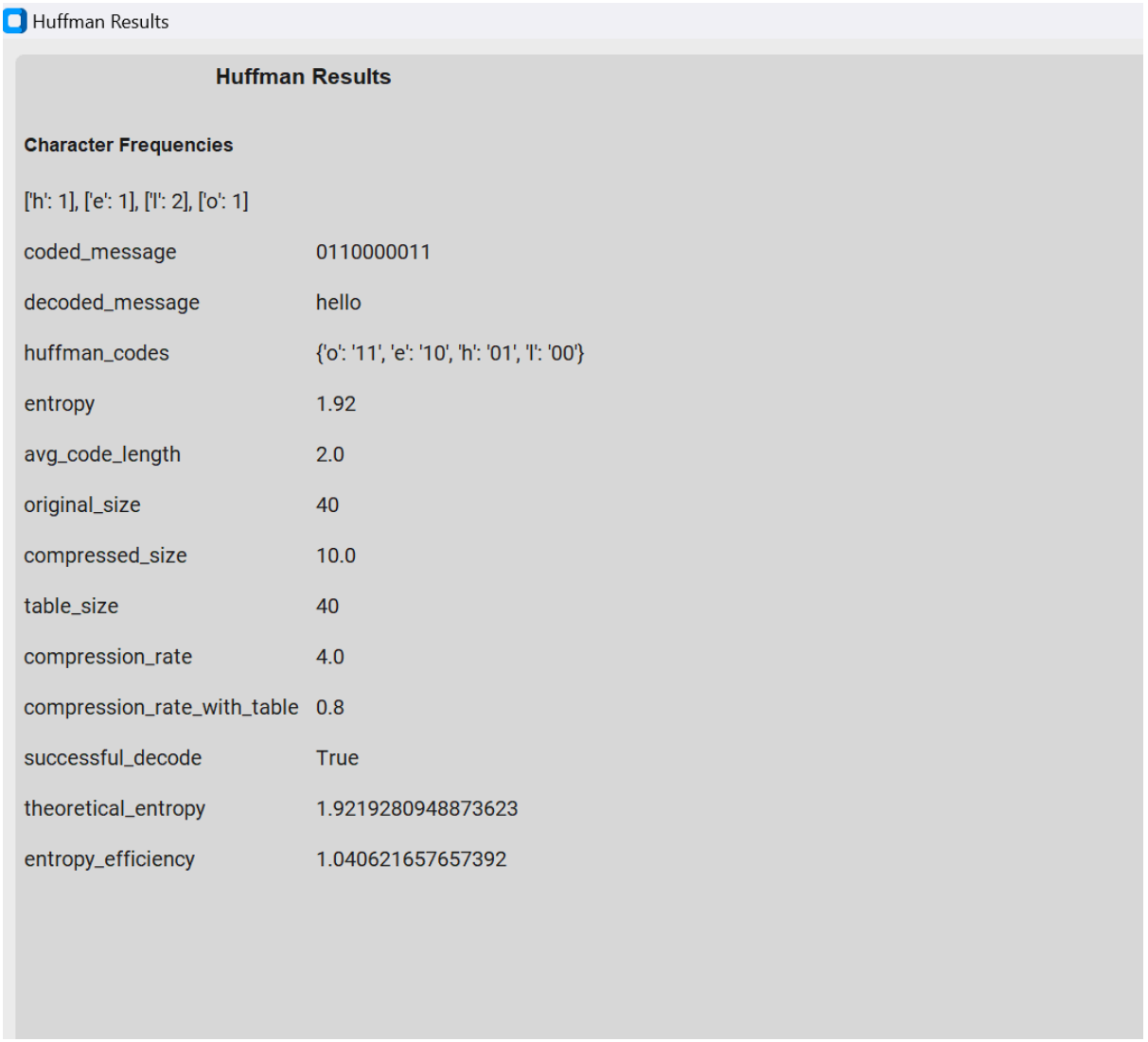
Shannon-Fano Results

Character Frequencies

['h': 1], ['e': 1], ['l': 2], ['o': 1]

coded_message	1110000001
decoded_message	hello
shanon-fano-codes	{'h': '11', 'e': '10', 'o': '01', 'l': '00'}
entropy	1.92
avg_code_length	2.0
original_size	40
compressed_size	10.0
table_size	40
compression_rate	4.0
compression_rate_with_table	0.8
successful_decode	True
theoretical_entropy	1.9219280948873623
entropy_efficiency	1.040621657657392

5.4. **Résultats pour le mot "hello" (huffman)**



5.5. **Résultats pour le mot "message" (Shannon-Fano)**

Shannon-Fano Results

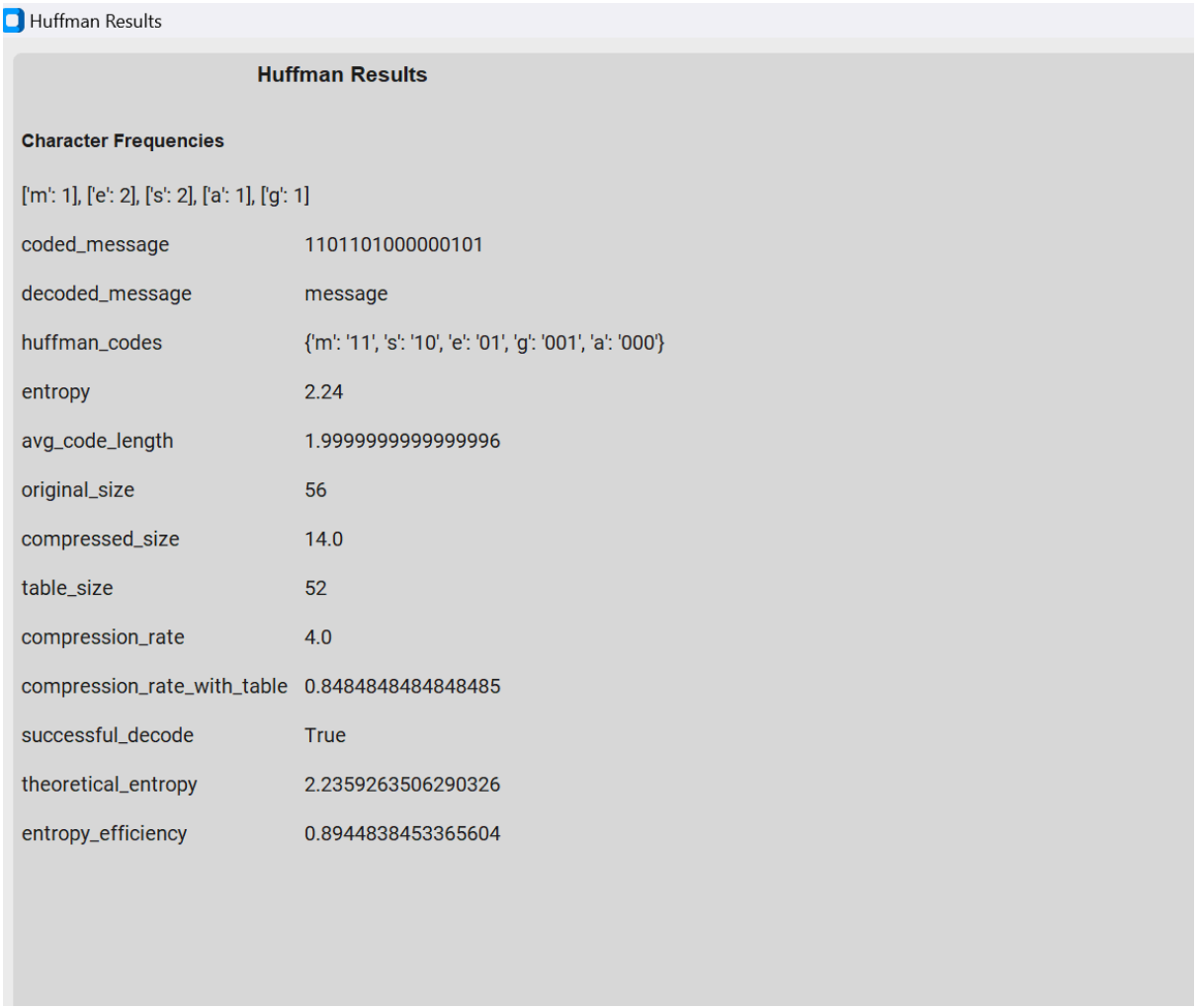
Shannon-Fano Results

Character Frequencies

['m': 1], ['e': 2], ['s': 2], ['a': 1], ['g': 1]

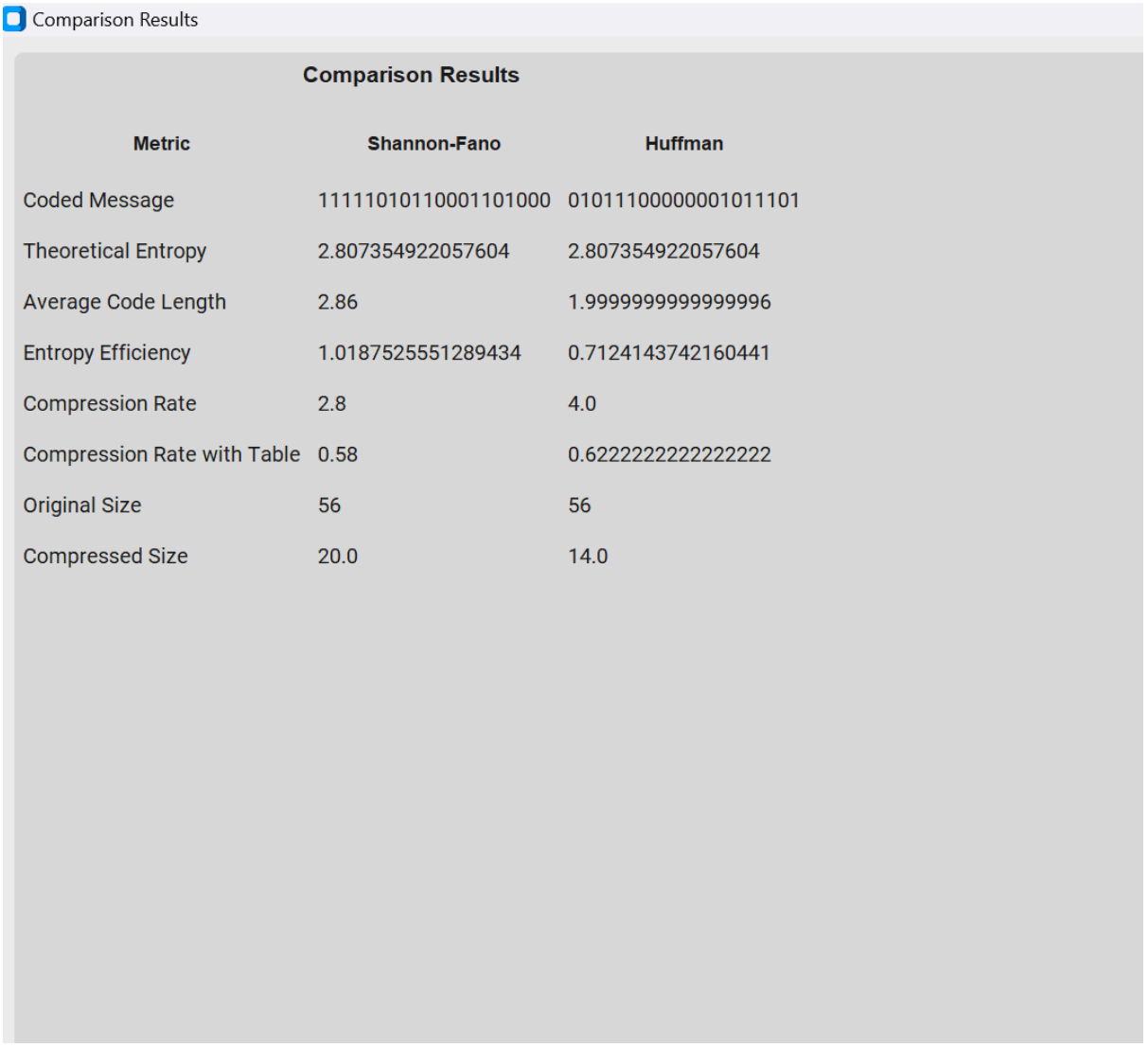
coded_message	1110100001101001
decoded_message	message
shanon-fano-codes	{'m': '111', 'a': '110', 'g': '10', 'e': '01', 's': '00'}
entropy	2.24
avg_code_length	2.29
original_size	56
compressed_size	16.0
table_size	52
compression_rate	3.5
compression_rate_with_table	0.82
successful_decode	True
theoretical_entropy	2.2359263506290326
entropy_efficiency	1.024184002910362

5.6. **Résultats pour le mot "message" (huffman)**



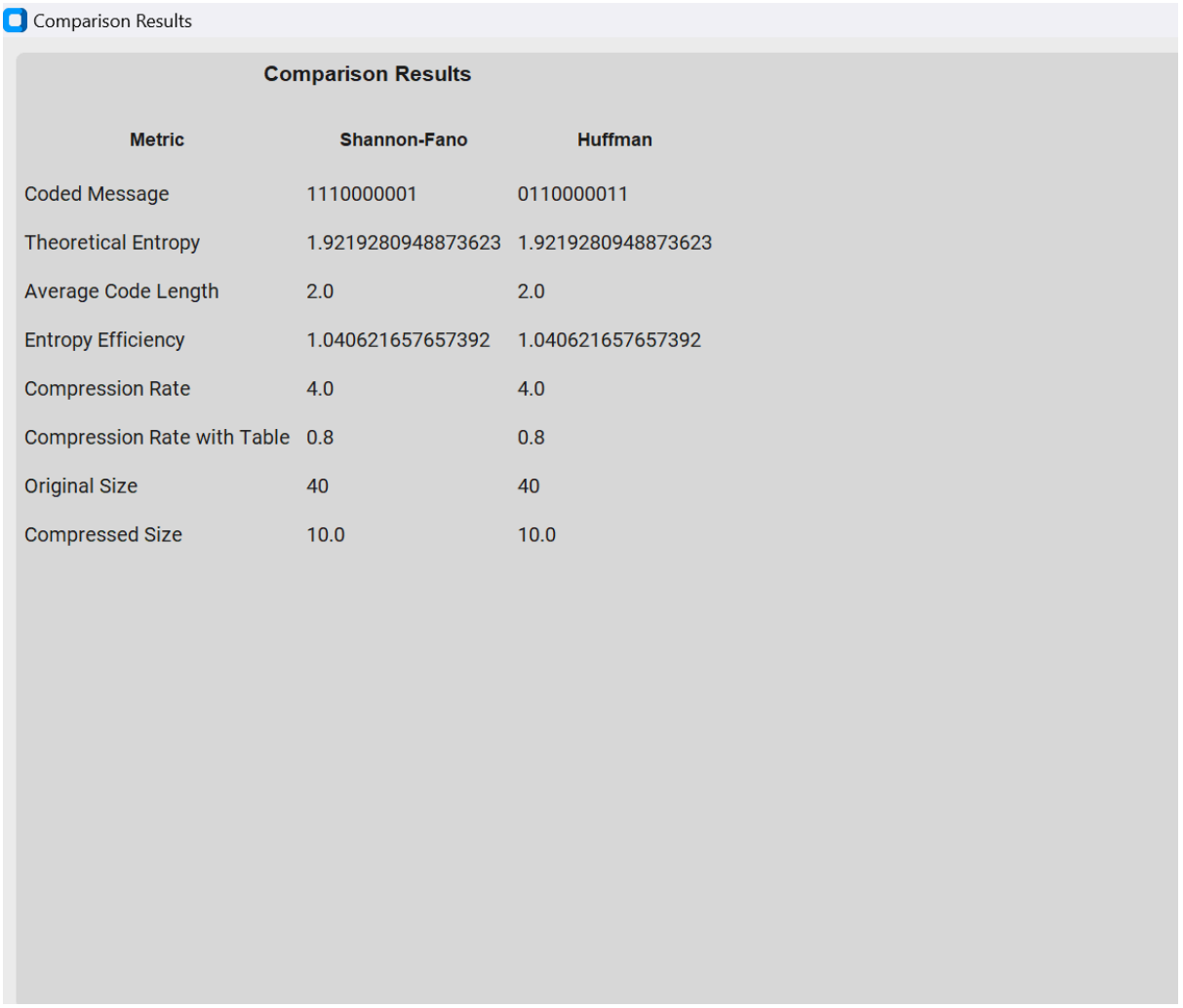
6- Discussion sur les résultats

1. **Résultats de comparaison pour le mot "mohcine"**



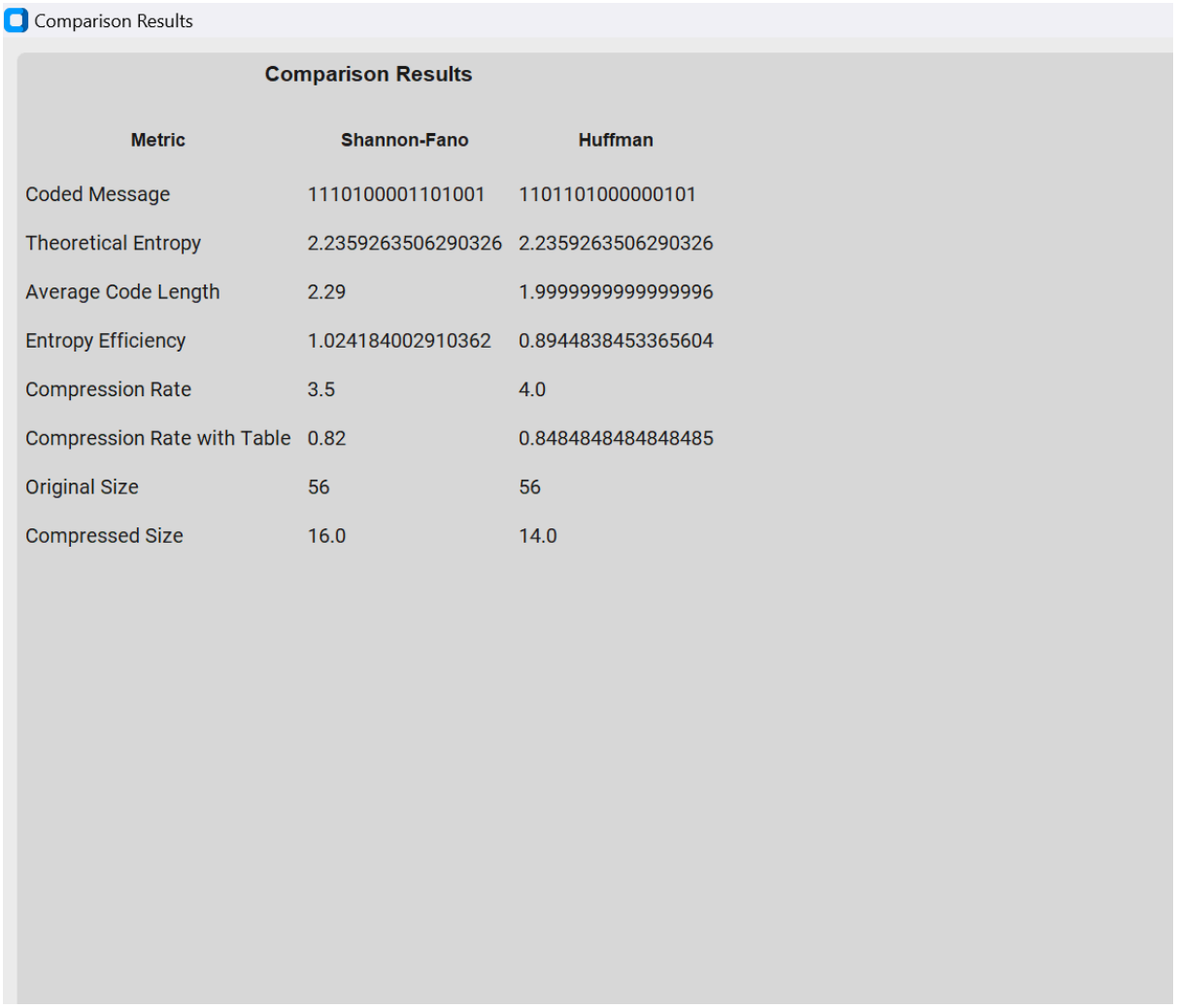
Metric	Shannon-Fano	Huffman
Coded Message	11111010110001101000	01011100000001011101
Theoretical Entropy	2.807354922057604	2.807354922057604
Average Code Length	2.86	1.9999999999999996
Entropy Efficiency	1.0187525551289434	0.7124143742160441
Compression Rate	2.8	4.0
Compression Rate with Table	0.58	0.6222222222222222
Original Size	56	56
Compressed Size	20.0	14.0

1. **Résultats de comparaison pour le mot "hello"**



Metric	Shannon-Fano	Huffman
Coded Message	1110000001	0110000011
Theoretical Entropy	1.9219280948873623	1.9219280948873623
Average Code Length	2.0	2.0
Entropy Efficiency	1.040621657657392	1.040621657657392
Compression Rate	4.0	4.0
Compression Rate with Table	0.8	0.8
Original Size	40	40
Compressed Size	10.0	10.0

1. **Résultats de comparaison pour le mot "message"**


Comparison Results

Comparison Results

Metric	Shannon-Fano	Huffman
Coded Message	1110100001101001	1101101000000101
Theoretical Entropy	2.2359263506290326	2.2359263506290326
Average Code Length	2.29	1.9999999999999996
Entropy Efficiency	1.024184002910362	0.8944838453365604
Compression Rate	3.5	4.0
Compression Rate with Table	0.82	0.8484848484848485
Original Size	56	56
Compressed Size	16.0	14.0

Dans les résultats obtenus pour les différentes méthodes de codage, il est intéressant de noter plusieurs différences significatives entre le codage Huffman et le codage Shannon-Fano, en particulier en ce qui concerne l'efficacité de l'entropie, le taux de compression et la taille des données comprimées.

1. Efficacité de l'entropie :

L'efficacité de l'entropie mesure à quel point le codage est proche de l'entropie théorique du message. En général, **Shannon-Fano** présente une efficacité d'entropie supérieure par rapport à **Huffman**. Cela signifie que Shannon-Fano utilise des représentations des symboles plus proches de l'optimal théorique.

Exemple :

- Pour le mot **"message"**, l'efficacité de l'entropie pour **Shannon-Fano** est de **1.024**, tandis que pour **Huffman**, elle est de **0.894**. Cela montre que Shannon-Fano est plus proche de l'entropie théorique, ce qui indique une meilleure utilisation des symboles par rapport à Huffman, qui est moins optimal sur ce point.

2. Taux de compression :

Le taux de compression représente l'efficacité avec laquelle un message est réduit en taille après codage. Ici, **Huffman** surpasse généralement **Shannon-Fano**. Cela signifie que, pour des données spécifiques, Huffman parvient à obtenir une meilleure compression.

Exemple :

- Pour **"message"**, le taux de compression pour **Shannon-Fano** est de **3.5**, tandis que pour **Huffman**, il est de **4**. Cela suggère que Huffman réduit plus efficacement la taille du message que Shannon-Fano. Le même constat peut être fait pour le mot **"mohcine"**, où **Shannon-Fano** a un taux de compression de **2.8** contre **4** pour **Huffman**. Ainsi, Huffman réussit toujours à réduire davantage la taille des données.

3. Taille des données comprimées :

La taille des données comprimées est un facteur clé pour évaluer la performance des algorithmes de compression. **Huffman** obtient généralement des tailles comprimées plus petites, ce qui est un avantage lorsque l'objectif est de minimiser l'espace de stockage ou de transmission des données.

Exemple :

- Pour le mot **"message"**, la taille originale est de **56**, et après compression, Shannon-Fano réduit la taille à **16**, tandis que Huffman atteint **14**.

- De même, pour **"mohcine"**, la taille originale est également de **56**, et Shannon-Fano réduit la taille à **20**, tandis que Huffman la compresse à **14**. Ces résultats montrent que Huffman est plus performant dans la réduction de la taille des données, ce qui peut être essentiel dans des applications où la minimisation de l'espace est cruciale.

4. Complexité et performance :

Une différence notable entre les deux méthodes réside dans leur complexité. **Shannon-Fano** est plus simple à implémenter que **Huffman**, mais cette simplicité a un coût en termes de performance de compression. En effet, bien que Shannon-Fano soit plus intuitif et plus facile à coder, il n'atteint pas les performances de compression de Huffman.

Exemple :

- En prenant l'exemple du mot **"message"**, même si Shannon-Fano atteint une efficacité théorique plus élevée, **Huffman** parvient à une compression plus efficace (avec une taille comprimée de 14 contre 16 pour Shannon-Fano). Cela montre que la complexité supplémentaire de Huffman vaut souvent la peine lorsque l'objectif est une compression maximale, même si la mise en œuvre de Huffman peut être plus complexe.

Conclusion générale :

En conclusion, **Shannon-Fano** et **Huffman** présentent chacun des avantages et des inconvénients en fonction des critères d'application. **Shannon-Fano** est plus proche de l'efficacité théorique de l'entropie, ce qui le rend plus adapté pour des situations où l'optimisation théorique de la représentation des symboles est importante. Cependant, **Huffman** est plus efficace en termes de taux de compression et de taille des données comprimées, ce qui le rend préférable lorsque la compression réelle des données est une priorité. En somme, le choix entre les deux méthodes dépendra des objectifs spécifiques de l'application : si l'on recherche une compression maximale des données, **Huffman** sera généralement le

choix préféré, tandis que si l'on privilégie la simplicité ou l'efficacité théorique, **Shannon-Fano** peut être plus adapté.