PROGRAMMATION FONCTIONNELLE

Projet méthodes de compression sans perte

Professeur : Mme Inès ALAYA

24 MARS 2024

AÏT CHADI Anissa
COSTA Mathéo
KHALI Brayan
TOURE Matthieu
TRAN Alexandre

ING 2 GSI Groupe 1

Description succincte de la conception mise en place :

Pour mener à bien ce projet, nous avons tout d'abord établi un ordre de priorité pour chaque méthode de compression. Nous avons décidé de commencer par ce qui nous semblait le plus simple, c'est-à-dire RLE. Puis, nous avons continué avec les méthodes à dictionnaire LZ78 et LZW car nous voulions laisser les méthodes statistiques, qui nous semblaient les plus complexes, pour la fin.

Pour la conception, nous avons décidé de réaliser le pseudo-code de chaque méthode de compression, en essayant au mieux de respecter les contraintes. Cependant, étant donné que nous avions encore du mal à faire du code sans variables, nous en avons laissé dans le pseudo-code. Ainsi, nous avons commencé à coder sans prendre en compte toutes les contraintes, afin d'avoir quelque chose de fonctionnel. Ensuite, une fois que tout fonctionnait, nous avons alors modifié l'implémentation du code afin que ce dernier réponde à toutes les contraintes.

Nous avons, pour chaque méthode de compression, d'abord complété les modules nécessaires à la réalisation des méthodes (c'est-à-dire ***EncodingTree.hs*** *et* ***Source.hs***).
Nous avons donc implémenté *occurrences*, *entropy* et *orderedCounts*, puis *has*, *meanLength*, *encode*, *decodeOnce*, *decode*, *compress* et *uncompress* pour pouvoir ensuite compléter et implémenter les modules des méthodes de la manière décrite précédemment.

A présent, rentrons un peu plus dans le détail concernant la conception de chaque méthode avec notamment les éléments à inclure ou utiliser en amont de l'implémentation.

RLE.hs **:** Pour la fonction *compress*, nous avions pensé à utiliser la fonction *group* du paquet Data.List afin de regrouper les éléments identiques et pour la décompression : *concatMap* afin de décompresser la liste de tuples en une liste de répétitions. Finalement, nous avons continué avec une autre implémentation que nous détaillerons un peu plus dans la suite de ce rapport.

LZ78.hs **:** Pour la fonction *compress*, nous devions donc utiliser un dictionnaire vide (donc utiliser le module Dictionaries.hs) pour stocker les chaînes maximales rencontrées et les

associer à un numéro, puis ajouter dans le dictionnaire au fur et à mesure. Pour *decompress*, nous devions réutiliser le dictionnaire pour pouvoir reconstruire la séquence d'origine.

LZW.hs **:** Nous devions utiliser un dictionnaire initial non vide (la table ASCII de 0 à 255) pour compresser et réutiliser ce même dictionnaire pour décompresser la séquence.

Huffman.hs **:** Pour *tree*, nous devions trier la distribution par ordre décroissant et fusionner les deux derniers éléments de la liste jusqu'à obtenir un arbre.

ShanonFano.hs **:** Pour *tree*, au lieu de fusionner les deux derniers éléments comme pour Huffman, nous devions diviser la liste en deux sous-distributions les plus équilibrées possibles.

Description succincte de l'implémentation mise en place :

- ***RLE.hs***

Compress **:** Nous avons dû utiliser la récursivité non-terminale et nous avons utilisé *takeWhile* pour compter le nombre d'éléments consécutifs identiques et *dropWhile* pour passer à l'élément suivant après que la séquence ait été compressée.

Uncompress **:** Nous avons cette fois-ci utilisé la récursivité terminale pour décompresser la liste de tuples et utilisé *replicate* pour recréer les séquences originales à partir des tuples (symbole, compteur).

- ***LZ78.hs***

Compress **:** On construit un dictionnaire pour stocker les préfixes rencontrés et leur index et on utilise la fonction *elemIndex* pour vérifier si un préfixe existe déjà dans le dictionnaire et on utilise la fonction auxiliaire *compressHelper* pour gérer la compression récursive (qui prend en compte les préfixes et en mettant à jour le dictionnaire).

Uncompress **:** On utilise le dictionnaire et les tuples (index, caractère) pour reconstruire les séquences originales. Pour accéder à un élément spécifique dans une liste, on utilise la fonction

- ***LZW.hs :***

Compress **:** On utilise la récursivité pour parcourir la chaîne d'entrée. Le dictionnaire est mis à jour au fur et à mesure que de nouveaux codes sont générés grâce à la fonction auxiliaire updateDictionary. Nous avons rajouté la gestion des cas où une séquence est déjà présente dans le dictionnaire et où elle ne l'est pas.

Uncompress **:** On utilise la fonction auxiliaire Uncompress', qui reconstruit la chaîne originale en utilisant le dictionnaire.

- ***Huffman.hs :***

Nous devions implémenter la fonction *tree* qui permet de construire l'arbre d'Huffman à partir des feuilles et des nœuds internes. Pour cela, nous avons utilisé une logique conditionnelle pour gérer le cas où la liste d'entrée est vide.

Nous avons rajouté quelques fonctions auxiliaires :

- *buildLeafNodes*, qui construit les feuilles de l'arbre à partir de la liste d'entrée en regroupant et en comptant les fréquences de chaque symbole
- *buildHuffmanTree*, qui construit l'arbre d'Huffman en insérant les noeuds internes jusqu'à ce qu'il ne reste qu'un seul noeud
- *sortByFrequency*, pour trier les noeuds par fréquence
- *insertInternalNode*, pour insérer un noeud interne dans la liste des noeuds

- ***ShannonFano.hs :***

Pour implémenter la fonction *tree*, nous avons utilisé la fonction *orderedCounts* de **Source.hs** afin d'avoir la distribution ordonnée des symboles avec leurs fréquences. La fonction *tree* appelle la fonction auxiliaire *buildTree* pour construire l'arbre d'encodage à partir de la distribution des symboles.

Tout comme Huffman, nous avons ajouté des fonctions auxiliaires :

- *buildTree*, qui construit l'arbre d'encodage en divisant la distribution en deux parties à chaque niveau et en combinant les nœuds jusqu'à ce qu'il n'en reste plus qu'un seul.
- *splitDistribution*, qui permet de séparer la distribution des symboles en deux parties en trouvant le point de séparation basé sur la somme des fréquences.
- *findSplit* point, qui trouve donc le point de distribution, en parcourant la liste des fréquences, jusqu'à ce que la somme des fréquences atteigne la moitié de la somme totale.

- ***Spec.hs :***

Nous avons un fichier test, qui nous a énormément servi lors de l'implémentation des méthodes.
A chaque test, il y a 8 cas testés : compression et décompression pour RLE, LZ78 et LZW (donc 2*3) et un test pour vérifier l'arbre pour Huffman et Shannon-Fano. Lorsque tout fonctionne bien, nous obtenons ceci :

```
Cases: 8  Tried: 8  Errors: 0  Failures: 0
```

S'il y a une erreur, par exemple que le dictionnaire ne correspond pas à celui attendu, alors qu'on aura une Failure comme vous pouvez le voir dans l'exemple ci-dessous :

```
### Failure in: 1:0:LZ78 Compression
test/Spec.hs:22
expected: [(0,'e'),(0,'l'),(3,'e'),(0,' '),(2,'c'),(0,'h'),(2,'l'),(4,' '),(0,'!')]
 but got: [(0,'b'),(0,'e'),(0,'l'),(3,'e'),(0,' '),(2,'c'),(0,'h'),(2,'l'),(4,' '),(0,'!')]
Cases: 8  Tried: 8  Errors: 0  Failures: 1
```

Étude des méthodes de compression :

Pour étudier les différentes méthodes de compression, nous avons réalisé un benchmark. C'est-aà-dire une suite de tests sur différentes références afin de faire des comparaisons, qui nous donne le temps d'exécution pour tout le processus de chaque méthode (donc temps de compression ET décompression) ainsi que le taux de compression pour les méthodes RLE, LZ78 et LZW.

Nous avons essayé de le faire pour les méthodes statistiques mais nous n'y sommes pas parvenus car cela était beaucoup trop complexe et que le temps nous manquait.

Le benchmark est assez simple : On écrit une chaîne de caractères à tester dans le code et ce sera cette même chaîne qui sera utilisée pour toutes les méthodes, ce qui permettra une meilleure comparaison.

Nous avons réalisé 5 benchmarks avec 5 chaînes de caractères différentes.

1) "AAAABBBCCCCDDDD"

```
Benchmarking RLE...
RLE Compression/Decompression Time: 4.482e-6s, Compression Ratio: 26.666666666666668%
Original Size: 15 bytes, Compressed Size: 4 bytes
Benchmarking LZ78...
LZ78 Compression/Decompression Time: 1.1466e-5s, Compression Ratio: 66.66666666666666%
Original Size: 15 bytes, Compressed Size: 10 bytes
Benchmarking LZW...
LZW Compression/Decompression Time: 2.62e-7s, Compression Ratio: 73.33333333333333%
Original Size: 15 bytes, Compressed Size: 11 bytes
Benchmarking Huffman...
Huffman Compression/Decompression Time: 1.3547e-5s
Benchmarking Shannon-Fano...
Shannon-Fano Compression/Decompression Time: 1.1661e-5s
```

2) "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

```
Benchmarking RLE...
RLE Compression/Decompression Time: 9.049e-6s, Compression Ratio: 100.0%
Original Size: 26 bytes, Compressed Size: 26 bytes
Benchmarking LZ78...
LZ78 Compression/Decompression Time: 1.8247e-5s, Compression Ratio: 100.0%
Original Size: 26 bytes, Compressed Size: 26 bytes
Benchmarking LZW...
LZW Compression/Decompression Time: 2.87e-7s, Compression Ratio: 100.0%
Original Size: 26 bytes, Compressed Size: 26 bytes
Benchmarking Huffman...
Huffman Compression/Decompression Time: 5.444e-5s
Benchmarking Shannon-Fano...
Shannon-Fano Compression/Decompression Time: 3.1899e-5s
```

3) **“ABCDABCDXYZXYZ”**

```
Benchmarking RLE...
RLE Compression/Decompression Time: 4.362e-6s, Compression Ratio: 100.0%
Original Size: 14 bytes, Compressed Size: 14 bytes
Benchmarking LZ78...
LZ78 Compression/Decompression Time: 1.0459e-5s, Compression Ratio: 78.57142857142857%
Original Size: 14 bytes, Compressed Size: 11 bytes
Benchmarking LZW...
LZW Compression/Decompression Time: 2.33e-7s, Compression Ratio: 78.57142857142857%
Original Size: 14 bytes, Compressed Size: 11 bytes
Benchmarking Huffman...
Huffman Compression/Decompression Time: 1.8995e-5s
Benchmarking Shannon-Fano...
Shannon-Fano Compression/Decompression Time: 1.5583e-5s
```

4) **“abcdeffedcbazzzzzzzzz”**

```
Benchmarking RLE...
RLE Compression/Decompression Time: 4.436e-6s, Compression Ratio: 57.14285714285714%
Original Size: 21 bytes, Compressed Size: 12 bytes
Benchmarking LZ78...
LZ78 Compression/Decompression Time: 1.6631e-5s, Compression Ratio: 61.904761904761905%
Original Size: 21 bytes, Compressed Size: 13 bytes
Benchmarking LZW...
LZW Compression/Decompression Time: 3.11e-7s, Compression Ratio: 76.19047619047619%
Original Size: 21 bytes, Compressed Size: 16 bytes
Benchmarking Huffman...
Huffman Compression/Decompression Time: 2.8375e-5s
Benchmarking Shannon-Fano...
Shannon-Fano Compression/Decompression Time: 2.0241e-5s
```

5) **“But I must explain to you how all this mistaken idea of denouncing pleasure and praising pain was born and I will give you a complete account of the system, and expound the actual teachings of the great explorer of the truth, the master-builder of human happiness. No one rejects, dislikes, or avoids pleasure itself, because it is pleasure, but because those who do not know how to pursue pleasure rationally encounter consequences that are extremely painful. Nor again is there anyone who loves or pursues or desires to obtain pain of itself, because it is pain, but because occasionally circumstances occur in which toil and pain can procure him some great pleasure. To take a trivial example, which of us ever undertakes laborious physical exercise, except to obtain some advantage from it? But who has any right to find fault with a man who chooses to enjoy a pleasure that has no annoying consequences, or one who avoids a pain that produces no resultant pleasure?”**

```
Benchmarking RLE...
RLE Compression/Decompression Time: 1.51243e-4s, Compression Ratio: 98.86597938144331%
Original Size: 970 bytes, Compressed Size: 959 bytes
Benchmarking LZ78...
LZ78 Compression/Decompression Time: 3.432983e-3s, Compression Ratio: 38.76288659793815%
Original Size: 970 bytes, Compressed Size: 376 bytes
Benchmarking LZW...
LZW Compression/Decompression Time: 4.3e-7s, Compression Ratio: 55.05154639175258%
Original Size: 970 bytes, Compressed Size: 534 bytes
Benchmarking Huffman...
Huffman Compression/Decompression Time: 6.58209e-4s
Benchmarking Shannon-Fano...
Shannon-Fano Compression/Decompression Time: 6.83323e-4s
```

Analyse :

Pour la méthode RLE, nous pouvons voir que les configurations donnant les meilleures performances sont les chaînes de caractères qui comportent de nombreuses répétitions. En effet, le taux de compression est plus bas ce qui indique une bonne compression (par exemple dans le premier exemple, la taille de la chaîne était de 15 bytes contre 4 après compression). Nous avons également testé avec une chaîne de a répétés 35x et le taux de compression avoisine les 2% ce qui est très optimal et cohérent.

Concernant les méthodes à dictionnaire, nous avons remarqué que les configurations donnant les meilleurs résultats sont les longues chaînes de caractères comme un paragraphe. Dans l'exemple 5, on peut voir que le message original était de 970 bytes et qu'il est passé en compressé à 376 pour LZ78 et 534 bytes pour LZW.
De manière générale, on constate que LZ78 reste plus performant que LZW en termes de données compressées, mais c'est LZW qui a le temps d'exécution le plus rapide, ce qui n'est pas négligeable non plus dans un programme.

Enfin, pour les deux méthodes statistiques, nous pouvons seulement comparer les temps d'exécution ainsi que les entropies et longueurs moyennes.
Globalement, nous pouvons voir que les deux méthodes sont à peu près équivalentes en temps d'exécution, bien que Shannon-Fano soit tout de même parfois un peu plus rapide que Huffman.
Nous pouvons comparer leur entropie et longueurs moyennes. Très souvent, elles sont identiques pour les deux méthodes. Il arrive parfois que la longueur moyenne diffère comme c'est le cas pour la chaîne **"abcdeffedcbazzzzzzzzz"** où les deux donnent la même entropie mais une mean length différente pour les deux (2.52 pour Huffman contre 2.66 pour Shannon-Fano) ou encore pour le paragraphe (4.27 pour Huffman contre 4.48 pour Shannon-Fano).

Voici le lien de notre github si vous souhaitez voir comment nous avons travaillé dessus 🙂 :

https://github.com/Le-7/Compress-HaskHell.git