Laboratoire 1

Algorithmes de compression

Numéro d'équipe	208
Étudiants	Kevin Chénier Jérémy Lavoie-Lagueux Hadrien Renault
Codes permanents	CHEK27129505 LAVJ15049709 RENH23089909
Cours	LOG320 - Structure de données et algorithmes
Session	Été 2020
Groupe	04
Chargé de laboratoire	Francis Cardinal
Date	14 juin 2020

1. Introduction

Il existe plusieurs types d'algorithmes de compression. Les algorithmes de compression avec perte compressent des fichiers et la décompression de ces fichiers génèrent une perte de données, c'est-à-dire que le fichier décompressé diffère du fichier original. Pour ce laboratoire, nous avons travaillé sur les algorithmes de compression sans perte, tels que LZW et Huffman. Le but du laboratoire était d'implémenter les algorithmes LZW et Huffman, apporter une optimisation à ces algorithmes sur leur temps d'exécution, analyser asymptotiquement les algorithmes et justifier les choix de conception.

2. Description des algorithmes en pseudocode

2.1 LZW

Compression :

Premièrement, on construit le dictionnaire de base:

```
11. dictSize = 256
12. Hashmap dictionnary
13. pour i = 0 à dictSize
14.     dictionnary.add( (char) i, i)
```

On lit ensuite le fichier et on ajoute les symboles dans le dictionnaire, dépendamment si un certain "pattern" se répète ou non:

```
23. tant que singleCharInt = bufferedInputStream et que le fichier peut encore être lu
24.     singleChar = (char) singleCharInt
25.     wc = w + singleChar
26.     si dictionnary contient wc
27.         w = wc
28.     sinon
29.         result.add(dictionnary.get(w))
30.         dictionnary.put(wc, dictSize++)
31.         w = "" + singleChar
```

On ajoute ensuite la valeur contenu dans "w" (qui provient du dictionnaire), dans "result":

```
41. si w n'est pas vide
```

```
42.     result.add(dictionnary.get(w))
```

On écrit ensuite le résultat de notre compression dans un fichier à l'aide de DataOutputStream:

```
47. pour tous les i de result
48.     os = DataOutputStream(bufferedOutputStream)
49.     os.writeInt(i)
```

Décompression:

Premièrement, on construit le dictionnaire de base, tout comme la compression:

```
58. dictSize = 256
59. Hashmap dictionnary
60. pour i = 0 à dictSize
61.     dictionnary.add( (char) i, i)
```

Nous utilisons ensuite un DataInputStream et un BufferedInputStream pour lire ce que le fichier compressé contient. Les valeurs contenues dans le fichier compressé sont en binaires:

```
71. k = 0
72. tant que is.available() > 0
73.     k = is.readInt()
74.     entry = dictionnary.get(k)
75.     si entry est null
76.             entry = w + w.charAt(0)
```

On écrit ensuite la valeur de l'entrée dans le fichier de sortie:

```
77.     os.writeBytes(entry)
```

Si la valeur de "w" n'est pas nulle, on ajoute w + entry.charAt(0) dans le dictionnaire et on incrémente la grosseur du dictionnaire:

```
78.     si w n'est pas null
79.             dictionnary.put(dictSize++, w + entry.charAt(0))
```

80. w = entry

2.2 Huffman

Compression :

On commence par créer notre table de fréquences en stockant chaque caractère qui apparaît dans le fichier et le nombre de fois qu'il apparaît dans une HashMap.

```
64. HashMap freqCodes
65. Pour tous les caractères du fichier
66.     Si la key de freqCodes ne contient pas le caractère
67.         freqCodes.put(fileString.charAt(i), 1)
68.     Sinon
69.         freqCodes.put(fileString.charAt(i), freqCodes.get(fileString.charAt(i)) + 1)
```

On crée ensuite notre arbre de Huffman en utilisant notre table de fréquence, une PriorityQueue et notre classe HuffmanNode

```
19. PriorityQueue prioQueue
20. Set<Character> keySet = freqCodes.keySet()
21. Pour tous les éléments du keyset
22.     HuffmanNode huffmanNode = new HuffmanNode()
23.     huffmanNode.data = c
24.     huffmanNode.frequency = freqCodes.get(c)
25.     huffmanNode.left = null
26.     huffmanNode.right = null
27.     prioQueue.offer(huffmanNode)

30. Tant que la prioQueue n'est pas vide
31.     HuffmanNode x = prioQueue.peek()
32.     prioQueue.poll()
33.     HuffmanNode y = prioQueue.peek()
34.     prioQueue.poll()
35.     HuffmanNode sum = new HuffmanNode()
36.     sum.frequency = x.frequency + y.frequency
37.     sum.data = '-'
38.     sum.left = x
39.     sum.right = y
40.     root = sum
41.     prioQueue.offer(sum)
```

On crée ensuite nos codes binaires pour chaque caractère (codes plus courts pour caractères qui apparaissent plus fréquemment)

```
HuffmanNode nodePrefix
```

```
HashMap huffmanPrefix
StringBuilder prefixCode
48. Tant que l'arbre n'est pas vide
49.     Si l'arbre de gauche est vide et l'arbre de droite est vide
50.         huffmanPrefix.put(nodePrefix.data, prefixCode.toString())
51.     Sinon
52.         prefixCode.append('0')
53.         setPrefixCodes(nodePrefix.left, prefixCode)
54.         prefixCode.deleteCharAt(prefixCode.length() - 1)
55.         prefixCode.append('1')
56.         setPrefixCodes(nodePrefix.right, prefixCode)
57.         prefixCode.deleteCharAt(prefixCode.length() - 1)
```

On transforme le fichier original en fichier binaire avec les codes créés.

```
91. Pour tous les caractères du fichier
92.     char c = fileString.charAt(i)
93.     s.append(huffmanPrefix.get(c))
```

Finalement, on crée des bits avec le fichier binaire obtenu pour permettre la compression (un fichier plus petit que l'original)

```
39. BitOutputStream bos
40. Pour tous les caractères du fichier
41.     bos.writeBit(Character.getNumericValue(binaryString.charAt(j)))
```

Décompression :

Convertir les bits en binaire pour pouvoir reconvertir le fichier à sa forme originale

```
104. StringBuilder sBuilder
121. Int bit
122. Tant que le bit lu n'est pas -1
123.     bit = bis.readBit()
124.     Si le bit est 0 ou 1if (bit == 0 || bit == 1)
125.         sBuilder.append(bit)
```

Le même algorithme pour créer l'arbre que dans la compression (pseudo-code 2 et 3 de la compression)

```
131. HuffmanNode decodeRoot = buildTree(freqCodes)
```

Traverse le fichier binaire et utilise l'arbre d'huffman créé pour convertir les string binaires et recomposer le fichier original

```
133.Pour tous les caractères du fichier
134.    int j = Integer.parseInt(String.valueOf(s.charAt(i)));
135.    Si j est égal à 0
136.            decodeRoot = decodeRoot.left;
137.            Si decodeRoot gauche et droite sont vides
138.                    compressedFile.append(decodeRoot.data);
139.                    decodeRoot = root;
140.    Si j est égal à 1
141.            decodeRoot = decodeRoot.right;
142.            Si decodeRoot gauche et droite sont vides
143.                    compressedFile.append(decodeRoot.data);
144.                    decodeRoot = root;
```

3. Ordre Asymptotique

3.1 LZW

Tout d'abord, nous allons analyser la fonction compress().

L'algorithme de compression crée le dictionnaire avec la boucle "for" aux lignes 13-14. La grosseur de cette boucle dépend de la variable dictionnarySize, qui est habituellement initialisé à 256. Cette boucle est donc négligeable pour le calcul de l'ordre asymptotique de cette fonction. La boucle 23-32 lit le fichier et reconstruit le dictionnaire dépendamment des valeurs trouvées dans le fichier. Il n'y a aucune boucle imbriquée dans cette boucle, donc celle-ci est d'ordre O(n). Pour la prochaine boucle 47-49, sa grosseur dépend du résultat construit dans le deuxième boucle 23-32. On peut donc dire qu'au pire des cas, cette boucle a comme ordre asymptotique O(n). L'addition de ces deux ordres est O(2n). Nous pouvons enlever la constante, donc l'ordre asymptotique de la fonction compress() est O(n).

Ensuite, nous allons analyser la fonction decompress().

L'algorithme de décompression crée le dictionnaire avec la boucle "for" aux lignes 60-62. Tout comme la fonction de compression, la grosseur de cette boucle dépend de la variable dictionnarySize, qui est habituellement initialisé à 256. On dit alors que cette boucle est négligeable. La boucle 72-83 lit un fichier et le décompresse. Cette boucle dépend donc de la grosseur du fichier, ce qui correspond au pire des cas à l'ordre O(n). Étant donné que c'est la seule boucle qui compte, l'ordre asymptotique de la fonction est O(n) aussi.

3.2 Huffman

Tout d'abord, nous allons analyser la fonction compress().

La boucle 85-89 remplit la table de fréquence. Cette boucle dépend de la grosseur du fichier, donc au pire des cas, cette boucle sera d'ordre O(n). La boucle aux lignes 112-115 s'occupe de construire le résultat entre le tableau de fréquences et le texte à compresser. Cette boucle dépend de la grosseur du texte à compresser, donc au pire des cas, cette

boucle sera d'ordre O(n). L'addition des ordres asymptotiques sera O(2n). On peut enlever la constante, donc l'ordre asymptotique sera O(n).

Ensuite, nous allons analyser la fonction decompress().

La boucle se situant aux lignes 143-148 s'occupe de lire tous les bits du fichier passés en BitInputStream. Le temps d'exécution dépend du nombre de bits présents dans le fichier, donc on peut assumer que l'ordre asymptotique sera O(n). Ensuite, la boucle aux lignes 156-172 s'occupe de reconstruire le fichier original. Il n'y a pas de boucle imbriqué, ni une condition qui décrémente, donc l'ordre asymptotique sera O(n). L'addition des deux ordres asymptotiques donne O(2n), on peut enlever la constante, donc l'ordre asymptotique final sera O(n).

4. Conception et implémentation

4.1 LZW

Nous n'avons pas réalisé d'améliorations majeures à l'algorithme fourni dans l'énoncé du laboratoire. La principale différence est que nous avons intégré la lecture et l'écriture des fichiers directement dans l'algorithme pour éviter d'avoir à emmagasiner l'entrée et la sortie dans des variables en mémoire. On optimise ainsi l'algorithme et notre gestion de la mémoire.

Un autre aspect important est que nous avons décidé d'avoir l'entier complet dans le fichier compressé. Bien que nous aurions pu déterminer la taille d'encodage en fonction du dictionnaire, nous avons manqué de temps pour avoir une optimisation parfaite. Dans ce sens, nous avons préféré y aller avec la taille complète de l'entier, soit 4 octets, par pure simplicité et s'assurer ainsi d'avoir un algorithme toujours fonctionnel. (Au détriment d'une compression plus efficace)

Finalement, nous avons utilisé des BufferedInputStream, BufferedOutputStream, DataInputStream et DataOutputStream pour la gestion des fichiers. Nous avons utilisé les BufferedStream puisqu'ils sont efficaces et permettent la lecture binaire de fichier. Ils nous permettaient donc de respecter la contrainte que notre algorithme doit être en mesure de compresser n'importe quel type de fichier. De plus, les DataStream utilisent les BufferedStream, mais rajoute une couche de méthodes par dessus. Nous avons donc pu utilisés les méthodes writeInt(), readInt() et writeBytes() pour manipuler efficacement nos fichiers.

4.2 Huffman

Pour l'algorithme de la table de fréquence, nous avons utilisé la même logique que le pseudo-code donné dans l'énoncé, mais évidemment adapté à une HashMap. Pour la création de l'arbre de Huffman, nous avons fait le choix logique d'utiliser la PriorityQueue ainsi qu'une classe HuffmanNode. La PriorityQueue permet d'implémenter de façon logique notre arbre : les éléments plus fréquents sont plus hauts dans la queue (qui représente l'arbre huffman).

Le principal ajout que nous avons fait est l'utilisation d'un fichier supplémentaire avec la table de fréquence pour permettre la décompression. En effet, nous avons d'abord essayé d'ajouter la table de fréquences dans l'en-tête du fichier compressé, mais le fichier que nous obtenions était impossible à parse pour pouvoir récupérer notre table.

Pour permettre la compression du fichier, nous avons utilisé les librairies disponibles sur moodle BitInputStream et BitOutputStream et nous avons converti notre fichier binaire en fichier de bits, ce qui nous a permi de passer d'un fichier de 1 478 621 bytes à un fichier compressé de 862 134 bytes. Pour la gestion des fichiers, nous avons appliqué la même logique que celle que nous avons expliqué en LZW.

5. Conclusion

En conclusion, le laboratoire nous demandait d'implémenter deux algorithmes de compression sans perte. Nous avons utilisé les pseudo codes proposés pour nous aider à développer les algorithmes. Nos difficultés ont surtout tourné autour de la lecture des fichiers et de l'écriture. Nous avons donc perdu énormément de temps à faire fonctionner cette portion, et avons conséquemment moins travaillé sur l'optimisation de nos algorithmes.

Au final, notre programme offre d'utiliser les algorithmes LZW et Huffman en passant en argument -lzw ou -huff en paramètre. À noter que vous devez également spécifier -c ou -d pour compresser ou décompresser et mettre par la suite le fichier intrant et le fichier extrant.