Université du Québec
École de technologie supérieure
Département de génie logiciel et des T.I.

Rapport de Laboratoire

Numéro du laboratoire	02
Nom du laboratoire	Algorithme de Huffman
Étudiant(s)	• Jugurtha BANDOU • Gilles F. DJOKO KAMGAING • Nikola Perotic
Code(s) permanent(s)	• BANJ10049605 • DJOG15097506 • PERN06049400
Numéro d'équipe	204
Cours	LOG 320
Session	Automne 2021
Groupe	02
Chargé(s) de laboratoire	• Francis Cardinal • Hung Tam Nguyen
Date	05 Octobre 2021

Description des algorithmes en pseudo-codes et analyse asymptotique des algorithmes

- Algorithme de Huffman (Construction de l'arbre)

	buildHuffmanTree(Map<Character, Integer>characterFrequencyTable)
	queue ← nouveau FillPriorité qui dépend de la valeur de fréquence (plus petite au plus grand)
1	**pour** chaque entree entry ∈ characterFrequencyTable nouveauNoeud← nouveau noeud Huffman nouveauNoeud.character ← entry.cle nouveauNoeud.frequence ← entry.valeur queue ← ajouter(nouveauNoeud) **fin pour**
	rootNode ← null
	tant que queue.taille > 1
2	x ← queue.premierValeur queue ← queue.effacerPremiereValeurs y ← queue.premiereValeur queue ← queue.effacerPremiereValeurs f ← HuffmanNode((x.frequence + y.frequence) , x, y) f.frequence ← x.frequence + y.frequence f.gauch ← x f.droite ← y rootNode ← f queue ← ajouter(f) **fin tant que**
	retour rootNode

Annotation	Analyse de complexité	Notation Grand-O
1	O(n) : On parcourt chaque caractère de la table de fréquence	O(n)
2	O(n) : la longueur de queue descend de 1 après chaque itération (taille = taile -2 +1) => taille -1	O(n)

*Sachant que '**n**' correspond à la taille de **characterFrequencyTable**
Nous avons une complexité de O(2n) qui est équivalent à **O(n)**

- **Algorithme d'encodage (Encoder le fichier source)**

	buildHuffmanMapFromTree **(Map<Character, String> map, HuffmanNode: root, String s)**

	si root = null
	  retour
	sinon si root.isLeaf()
	  map ← hash (cle = root.character, valeur = s)
	  retour
	sinon
	  buildHuffmanMapFromTree(map, root.nodeAGauche, s + "0")
	  buildHuffmanMapFromTree(map, root.nodeADroite, s + "1")
	fin si

T(n) = { O(1), **si root == null**
O(1), **sinon si root.isLeaf**
2* (T(n/2) + O(1)) **sinon**

T(n) = 2 T(n/2) + 1 => T(n) = aT(n/b) + f(n) => a = 2, b = 2, f(n) = 1

Est-ce que $1 \in O(n^{\log_b(a) - \varepsilon})$ pour $\varepsilon > 0$? **Oui**

Alors $T(n) \in n^{\log_2(2)} = O(n)$ ce qui veut dire que **T(n) ∈ O(n)**

	compress **(text, Map<Character, String> huffmanMap)**
	encodedString ← “”
1	**pour** chaque character c ∈ text encodedString ← encodedString + map.valeurDeLaClee(c) **fin pour**
	retour encodedString

*Sachant que '**n**' correspond à la taille du texte

Annotation	Analyse de complexité	Notation Grand-O
1	O(n) : parcourt chaque caractère du texte	O(n)

Nous avons une complexité de **O(n)**

	encodage()
	map ← HashMap<Character, String> text ← textSrouce root ← PremierNœudDeLArbreHuffman stringDebut ← “” buildHuffmanMapFromTree(map, root, stringDebut) textEncoder ← compress(text, map) **retour** textEncoder

Annotations	Analyse de complexité	Notation Grand-O
1	O(n) : la fonction **buildHuffmanMapFromTree()** est de complexité O(n), calculée si haut.	O(n)
2	O(m) : La fonction **compress()** est de complexité O(n), calculée si haut.	O(m)

Alors, pour nombre de nœuds dans l’arbre ‘**n**’ et pour la longueur du texte ‘**m**’, la complexité de l’encodage est de **O(n + m).**

- **Algorithme de décompression (Décompresser un fichier)**

	decode()

	stringHuffman ← string huffman de 1 et 0s
	textOriginal ← string vide
	noeudCourrant ← noeud root de l'arbre huffman
	
	pour chaque character c dans stringHuffman
	  si c == '0'
	  noeudCourrant ← noeudCourrant.noeudGauche
	  sinon si c == '1'
	  noeudCourrant ← noeudCourrant.noeudDroit
	  fin si
	
	  si noeudCourrant.estUneFeuille
	  textOriginal = textOriginal + noeudCourrant.valeurCharacter
	  noeudCourrant ← noeud root de l'arbre huffman
	  fin si
	fin pour
	retour textOriginal

Annotations	Analyse de complexité	Notation Grand-O
1	O(n) : la fonction passe par chaque caractère de la chaîne de huffman.	O(n)
2	O(1) : on change la référence du nœud courant.	O(n)
3	O(1) : on change la référence du nœud courant.	O(n)
4	O(1) : on ajoute a la fin de la chaîne le character trouver au noeud.	O(n)
5	O(1) : on la référence du noeud courant au premier noeud de l'arbre.	O(n)

Pour "**n**" égal au nombre de caractères dans la chaîne de huffman la complexité de l'algorithme est de **O(n).**

Justification du choix de conception et d'implémentation.

Pour la construction de l'arbre, nous avons opté de parcourir les nœuds à partir du nœud racine en voulant que toutes les feuilles de l'arbre soient situées à gauche et que tous les prochains nœuds se situent à droite. Cependant, cette conception n'était pas l'idéal car, le niveau de compression était moins efficace dépendamment du nombre de caractères contenue dans la table de fréquence. C'est pour cette raison que nous avons décidé de créer l'arbre à partir des feuilles de l'arbre en remontant vers le nœud racine par ordre de fréquence de chaque nœud. Cela nous a permis d'obtenir un meilleur taux de compression ainsi que des codes binaires associés à chaque caractère de la table de fréquence qui sont plus optimaux.

Références

1. Cours de LOG-320
2. https://codes-sources.commentcamarche.net/source/view/21477/892751#browser
3. https://www.techiedelight.com/huffman-coding/
4. https://stackoverflow.com/questions/30013292/how-do-i-write-multiple-objects-to-the-serializable-file-and-read-them-when-the