Structures de données et algorithmes

Projet 2: Structures de données

Pascal Fontaine – Jean-Michel Begon – Romain Mormont

30 mars 2020

L'objectif de ce projet est de vous familiariser avec la conception, l'implémentation et l'analyse de structures de données dans le contexte de la compression de fichiers textes. Le but sera de comparer les performances d'un point de vue pratique et théorique de différentes implémentations d'une même structure.

1 Enoncé

L'encodage fait partie intégrante des systèmes informatisés. Toute information, pour être stockée (on parle alors de donnée) et traitée par un ordinateur doit se ramener à une suite binaire. Par exemple, pour encoder des données textuelles, on peut utiliser le système ASCII qui a la particularité de représenter un caractère avec 7 bits[1]. On parle alors d'encodage à largeur fixe.

Lorsque la fréquence des caractères est variable (par exemple, la lettre "e" est plus fréquente que la lettre "z" en français), il est possible de représenter un même texte avec moins de bits qu'en utilisant un encodage à largeur fixe. Pour ce faire, on donne à un caractère fréquent un code avec moins de bits qu'un caractère rare. On parle d'encodage à largeur variable.

Afin d'éviter une ambiguïté lors du décodage d'un code à largeur variable, on doit utiliser des codes à préfixe unique (sinon, on ne sait pas où s'arrête un caractère et où commence le suivant). Nous allons nous intéresser à un tel codage: le codage optimal (et glouton) de Huffman.

1.1 Arbre de décodage

L'opération de décodage est rendue très simple par l'utilisation d'un arbre binaire.

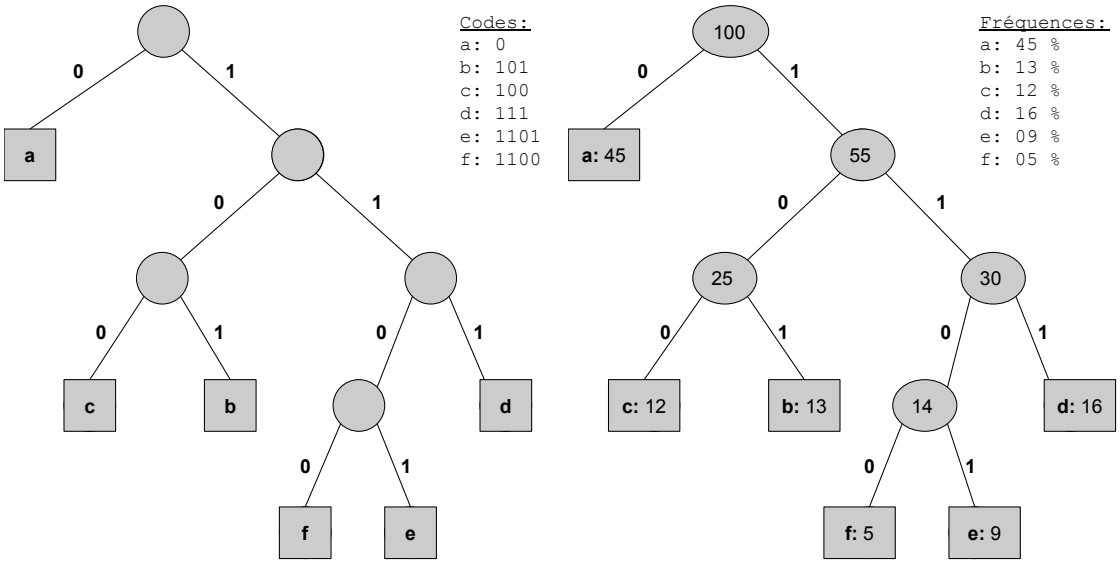
Soit un alphabet $\mathcal{A}$ contenant un nombre $|\mathcal{A}|$ de caractères utilisés dans un ensemble de textes de référence (*i.e.* un corpus). Un arbre de décodage est un arbre binaire possédant $|\mathcal{A}|$ feuilles. Chaque feuille est associée à un caractère distinct de l'alphabet.

Décoder une suite de bits revient à descendre dans l'arbre. Pour chaque 0 on passe au fils gauche. Pour chaque 1 on passe au fils droit (Figure 1a). Lorsqu'on arrive à une feuille, on a décodé notre caractère, et on peut poursuivre en redémarrant de la racine.

1.2 Encodage de Huffman

L'algorithme de Huffman procède de manière gloutonne: il attribue le code le plus court au caractère le plus fréquent, le second code le plus court au deuxième caractère le plus fréquent, et ainsi de suite.

[1] Le codage ASCII utilise en réalité 8 bits, le dernier bit étant ce que l'on appelle un bit de parité. Sept bits sont néanmoins suffisant pour encoder les caractères les plus fréquents (26 lettres minuscules, majuscules, quelques signes de ponctuation, etc.). Les accents, notamment ne sont pas pris en compte par ce format et on doit faire appel à des encodages plus modernes pour cela.



(a) Arbre simple

(b) Arbre enrichi. Les fréquences sont exprimées en pourcent.

Figure 1: Arbre de décodage (adapté de CLRS 2009).

On peut l'implémentant simplement en enrichissant la structure d'arbre. Chaque feuille, en plus de la lettre, contient la fréquence d'apparition de celle-ci. Quant aux nœuds internes, ils stockent la fréquence cumulée des feuilles du sous-arbre dont ils sont la racine (Figure 1b).

Chaque étape de l'algorithme consiste à fusionner les deux arbres qui sont associés aux fréquences cumulées les plus faibles:

HUFFMAN(C)
1 $n = |C|$
2 $Q =$"create a min-priority queue from C"
3 **for** $i = 1$ **to** $n - 1$
4 Allocate a new node z
5 $z.left =$ EXTRACT-MIN(Q)
6 $z.right =$ EXTRACT-MIN(Q)
7 $z.freq = z.left.freq + z.right.freq$
8 INSERT(Q, z)
9 **return** EXTRACT-MIN(Q)

où C est une structure qui contient l'alphabet et les fréquences associées.

1.3 Implémentation

En ce qui concerne les structures de données, nous allons nous intéresser à l'arbre d'encodage, ainsi que la file à priorité.

Arbre d'encodage L'interface de la structure d'arbre d'encodage est définie dans le fichier `CodingTree.h`. Nous vous demandons de fournir un fichier `CodingTree.c` implémentant cette interface.

File à priorité L'interface de la file à priorité est définie dans le fichier `PriorityQueue.h`. Nous vous demandons de fournir deux implémentations de la file. Le fichier `ListPriorityQueue.c` devra

contenir l'implémentation d'une file à priorité sous forme de liste. Le fichier `HeapPriorityQueue.c` devra quant à lui contenir l'implémentation sous forme de tas-min.

Décodage Au niveau algorithmique, nous vous demandons d'implémenter dans un fichier `decoding.c` la fonction de décodage déclarée dans le header `coding.h`.

Afin de vous aider dans la réalisation de ce projet, nous vous fournissons les fichiers suivants:

`freq.csv` un fichier CSV valide représentant la fréquence des caractères ASCII pour la langue anglaise (établie sur un corpus de livres anglais).

`Alice_in_wonderland_Lewis_Carroll.ascii` le livre de Lewis Carroll au format ASCII afin d'effectuer vos tests.

`coding.c` un fichier implémentant l'algorithme d'encodage. **Note**: étant donnée la longueur variable résultant de l'encodage de Huffman, il se peut que le nombre de bits à écrire dans le fichier de sortie ne soit pas un multiple de 8. De ce fait, il est nécessaire d'ajouter un padding au dernier octet à écrire dans le fichier. Afin d'éviter le décodage de ces bits de padding, nous utiliserons un caractère ASCII spécial pour déterminer la fin de la séquence encodée (par exemple, *file separator*). Le décodage devra donc s'arrêter lorsque ce caractère est rencontré.

`BinarySequence.h` **et** `BinarySequence.c` une interface et son implementation permettant de manipuler efficacement une séquence de bits.

`CharVector.h` **et** `CharVector.c` une interface et son implementation permettant de manipuler un vecteur de caractères.

`main.c` un fichier permettant de (dé)coder un texte donné en entrée. Le programme produit en compilant ce fichier s'exécute en ligne de commande et possède plusieurs paramètres:

```
./main [-e] [-d] [-f <eof_char>] [-o <outptPath>] <textPath> <csvPath>
```

- `-e`: si présent (optionnel), le programme décode l'entrée. Sinon, la décode.
- `-d`: si présent (optionnel), lance le code en mode debug. Le contenu encodé/décodé est notamment affiché sur la sortie standard plutôt que dans le fichier.
- `-f`: la valeur correspond au code ASCII à utiliser pour indiquer la fin d'une séquence encodée (optionnel). Par défaut, on utilise le caractère 28 (file separator).
- `-o`: chemin du fichier où écrire le contenu encodé/décodé (optionnel)
- `textPath`: le chemin d'un fichier d'entrée
- `csvPath`: le chemin du fichier CSV contenant les fréquences

2 Analyse théorique et empirique

Répondez aux questions suivantes dans votre rapport:

1. Illustrez et expliquez la structure des arbres construits avec la fonction HUFFMAN sur base des distributions de fréquences suivantes:
   (a) équiprobable: $f_i = \frac{1}{k}, i \in \{1, \ldots, k\}$
   (b) puissance de deux: $f_i = \frac{1}{2^i}, i \in \{1, \ldots, k-1\}$ et $f_k = f_{k-1}$
   (c) epsilon: $f_i = \frac{1}{k} + \epsilon_i$ avec

$$\epsilon_i = \begin{cases} \frac{-1}{3ki}, & i \in \{1, \ldots, \frac{k}{2}\} \\ \frac{1}{3k(k-i+1)}, & i \in \{\frac{k}{2}+1, \ldots, k\} \end{cases}$$

   où f_i $(0 < f_i < 1)$ dénote la fréquence du caractère $i \in \{1, \ldots, k\}$. On supposera un alphabet de taille $k = 2^q$, $k > 1$. Prenez $k = 8$ pour l'illustration.

2. Analysez la complexité de la fonction HUFFMAN(C) en fonction de k, la taille de l'alphabet:
   (a) dans le pire et meilleur cas pour l'implémentation de la file à priorité *avec liste*
   (b) dans le pire cas pour l'implémentation de la file à priorité *avec tas*

   Suggestion: faites le lien avec les distributions de fréquences développées dans la question 1.

3. Algorithme de décodage:
   (a) Donnez le pseudo-code de la fonction DECODE(B, T) qui, grâce à l'arbre de décodage T, décode une suite binaire représentée par un tableau B.
   (b) Analysez la complexité au meilleur et au pire cas de la fonction DECODE(B, T) par rapport à n, la taille du texte original (avant encodage), et k, la taille de l'alphabet.
   (c) Effectuez la même analyse pour le décodage d'un texte encodé sur base d'un encodage à largeur fixe. *Suggestion:* déterminez d'abord la structure de l'arbre résultant d'un encodage à largeur fixe.
   (d) Discutez de la différence moyenne de taille entre les fichiers encodés avec le codage à largeur fixe et celui de Huffman en fonction de n.
   (e) Reportez dans un graphique l'évolution des temps de décodage moyens dans le cas d'un codage à largeur fixe et dans le cas du codage de Huffman en fonction de n, la taille du texte original.
   (f) Discutez de l'adéquation entre l'analyse théorique et les résultats empiriques.

Pour trouver de long textes, vous pouvez vous référer au projet Gutenberg (`https://www.gutenberg.org/`). Attention toutefois aux caractères qui ne seraient dans l'alphabet.

Remarque: lorsqu'on parle d'évolution des temps moyens, il faut mesurer les temps de calculs pour une valeur de n (respectivement k) fixée mais pour des entrées différentes et faire la moyenne de ceux-ci. Il faut alors répéter le processus pour différentes valeurs de n (respectivement k). Vous pouvez adapter la fichier `main.c` à votre convenance pour réaliser ces mesures.

3 Deadline et soumission

Le projet est à réaliser **par groupe de 2** pour le **23 avril 2020** à **23h59** au plus tard. Le projet est à remettre via la plateforme de soumission de Montefiore: `http://submit.montefiore.ulg.ac.be/`. Il doit être rendu sous la forme d'une archive `tar.gz` contenant :

1. Votre rapport (5 pages maximum) au format PDF. Soyez bref mais précis et respectez bien la numération des (sous-)questions.
2. Les fichiers `CodingTree.c`, `ListPriorityQueue.c`, `HeapPriorityQueue.c` et `decoding.c`

Respectez bien les extensions de fichiers ainsi que leur nom pour les fichier `*.c` (en ce compris la casse). Seule la dernière archive soumise sera prise en compte. Vos fichiers seront évalués avec la commande:

```
gcc main.c CodingTree.c BinarySequence.c CharVector.c coding.c decoding.c
   ListPriorityQueue.c --std=c99 --pedantic -Wall -Wextra -Wmissing-prototypes
   -o huffman
```

en substituant adéquatement l'implémentation de la file à priorité.

Quelques remarques:

- Le projet doit être réalisé dans le standard C99.
- La présence de *warnings* impactera négativement la cote finale.
- Un projet qui ne compile pas avec cette commande recevra une cote nulle.

Un projet non rendu à temps recevra également une cote nulle. En cas de plagiat avéré, l'étudiant se verra affecter une cote nulle à l'ensemble des projets.

Les critères de correction seront précisés sur la page web du cours.