Projet d'Algorithmique – codage de Huffman

1 Introduction

Considérer la chaîne de caractères `abracadabra`. Pour l'écrire sur un fichier en codage ASCII, on utilise un octet par caractère, soit 11 octets = 88 bits. Dans ce codage, la lettre `a` a le **mot de code** `01100001`.

Mais, la chaîne `abracadabra` ne contient que 5 caractères différents, donc on pourrait faire mieux en associant les mots de code `a = 000`, `b = 001`, `c = 010`, `d = 011` et `r = 100` en binaire, soit 3 bits par caractère. En binaire cela fait 33 bits :

```
000 001 100 000 010 000 011 000 001 100 000
```

On observe maintenant que la chaîne contient beaucoup d'occurrences de la lettre `a` et qu'il pourrait donc être avantageux d'encoder la lettre `a` avec moins de bits que la lettre `d` qui n'a qu'une occurrence dans la chaîne. Considérer l'arbre binaire suivant :

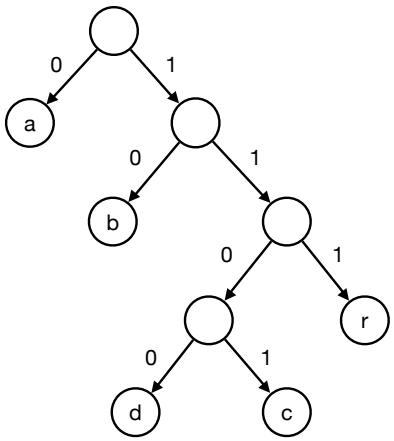


Fig. 1. un arbre de Huffman

Dans cet arbre binaire, les lettres `a`, `b`, `c`, `d` et `r` sont dans les feuilles, une arête vers un fils gauche est étiquetée par 0 et une arête vers un fils droit est étiquetée par 1. En suivant les arêtes de la racine vers les feuilles, on obtient les mots de code suivants : `a = 0`, `b = 10`, `c = 1101`, `d = 1100` et `r = 111`.

En binaire, la chaîne `abracadabra` est alors codée par 23 bits :

```
0 10 111 0 1101 0 1100 0 10 111 0
```

Exercice : vérifier qu'avec l'arbre binaire de la figure 1 et la chaîne codée (même sans les espaces, `01011101101011000101110`), vous pouvez facilement retrouver la chaîne d'origine.

L'arbre binaire de la figure 1 a été construit spécifiquement pour la chaîne `abracadabra` et ne convient évidemment ni pour des chaînes contenant d'autres caractères ni pour des chaînes ayant d'autres fréquences de lettres (plus de `'c'` ou `'d'` par exemple). En particulier, ce sont les fréquences des lettres de la chaîne d'origine qui déterminent le meilleur arbre binaire et codage. S'il y a beaucoup de lettres `e`, ce qui est souvent le cas, alors il est préférable de placer `e` plus près de la racine de l'arbre.

Ce type d'arbre binaire s'appelle un **arbre de Huffman**. L'objectif de ce projet est d'implanter **un codeur** et **un décodeur** en utilisant ce type de codage ainsi qu'un algorithme (inventé par David Huffman en 1952) pour construire l'arbre de Huffman donnant le meilleur codage (en nombre de bits utilisés) d'un texte donné. Chaque texte peut alors donner lieu à un arbre différent et il faut donc écrire l'arbre ainsi que le texte codé dans le même fichier. Cela a également pour conséquence que le codeur doit faire deux passages du fichier : d'abord un pour récupérer les fréquences des letters et ensuite, après que l'arbre est construit, un deuxième passage pour encoder le texte.

Les différentes parties du projet sont :

Encodage
- Lire un fichier de texte et construire l'arbre de Huffman correspondant.
- Écrire l'arbre sur le fichier de sortie.
- Relire le fichier de texte, l'encoder et écrire le texte codé sur le fichier de sortie.

Décodage
- Lire l'arbre de Huffman à partir d'un fichier codé.
- Lire et décoder le texte codé et écrire le texte décodé sur un fichier de sortie.

Points importants

- Vous devez faire du code propre, c'est-à-dire,
  - lisible (commentaires, identificateurs bien nommés, etc),
  - structuré (fichiers séparés pour les différentes composantes),
  - correct, vérifié (pas de `segmentation fault` – utiliser valgrind),
  - sans fuites (libérer la mémoire, vérifier avec valgrind)
- Votre programme est censé fonctionner
  - dans un premier temps avec tous les caractères ASCII $0, \ldots, 127$ (https://en.wikipedia.org/wiki/ASCII), les caractères `'0'`, `'1'`, `'\n'`, `'␣'` (espace), `'\0'` et `'\a'` (BEL) inclus et
  - dans un deuxième temps avec tous les octets possibles $(0, \ldots, 255)$ – vous devez être capable d'encoder et décoder un fichier exécutable de façon que l'exécutable fonctionne après le décodage.

2 Abracadabra

2.1 Lecture d'un arbre

▷ Écrire une fonction qui permet de lire un arbre binaire à partir d'un fichier.

Un arbre binaire est représenté dans un fichier par une chaîne de caractères correspondant à un parcours en profondeur en ordre préfixe de l'arbre en indiquant si chaque nœud visité est un nœud interne ou une feuille (et dans ce cas, quel caractère lui est associé) :

- un `1` indique que le nœud est un nœud interne avec deux fils ; on écrit d'abord le fils gauche et ensuite le fils droit ;
- un `0` indique que le nœud est une feuille. Le `0` est suivi par le caractère de la feuille.

Par exemple, l'arbre de la figure 1 est représenté par la chaîne :

```
10a10b110d0c0r
```

Explication de la représentation :

- `1` : le premier nœud (la racine) est un nœud interne ;
- `0` : le fils gauche de la racine est une feuille ;
- `a` : cette feuille contient le caractère `a` ;
- `1` : en remontant l'arbre, on continue avec le fils droit de la racine : le fils droit est un nœud interne ;
- `0` : le fils gauche du fils droit de la racine est une feuille ;
- `b` : cette feuille contient le caractère `b` ;
- ...

2.2 Construction des mots de code

▷ Écrire une fonction qui permet de traduire l'arbre à un tableau de mots de code.

Pour l'encodage du texte, on veut traduire chaque caractère au mot de code donné par le chemin de la racine de l'arbre jusqu'à la feuille qui contient le caractère. C'est utile de construire un tableau indexé par des caractères et qui dans chaque case contient une chaîne de caractères correspondant au mot du code du caractère. Un exemple d'une telle structure :

```
char *code_table[256];
```

Dans ce tableau, à la position 99 (le code ASCII de `'c'`) on veut stocker la chaîne de caractères `"100"`. L'encodage devient alors un simple parcours des caractères du texte qui à chaque caractère `ch` associe le mot de code `code_table[(unsigned char)ch]`.

Pour la construction du tableau, il suffit de faire un parcours en profondeur de l'arbre et pour chaque feuille rencontrée portant un caractère `ch`, stocker le chemin de la racine à cette feuille comme une chaîne de caractères dans `code_table[(unsigned char)ch]`.

Par exemple, en faisant un parcours en profondeur dans l'arbre de la figure 1, on stocke

```
code_table[97] = "0"
code_table[98] = "10"
code_table[99] = "1100"
code_table[100] = "1101"
code_table[114] = "111"
```

Ensuite, on pourrait encoder la chaîne `char str[] = "abracadabra"` par la boucle

```
int i, len = strlen(str);
for (i = 0; i < len; i++)
    printf("%s␣", code_table[(unsigned char)str[i]]);
```

2.3 Encodage

▷ Écrire un programme qui encode un fichier d'entrée et écrit le texte codé dans un fichier de sortie.

Pour l'instant vous utilisez l'arbre de la figure 1. Notez que cet arbre vous permet seulement d'encoder des textes qui contiennent les lettres `a`, `b`, `c`, `d` et `r`.

Il peut être utile (mais pour l'instant pas nécessaire) pour le décodeur de connaître le nombre de caractères du fichier d'origine. Une représentation possible est de commencer par écrire le nombre de caractères codés (ici 11) et ensuite les mots de code séparés par des espaces :

```
11
0 10 111 0 1101 0 1100 0 10 111 0
```

3 Construction de l'arbre de Huffman

3.1 Fréquences des caractères

▷ Écrire une fonction qui calcule les fréquences des caractères du fichier d'entrée et les stocke dans un tableau indexé par les caractères.

3.2 Construction de l'arbre

▷ Écrire une fonction qui, à partir d'un tableau de fréquences, construit l'arbre de Huffman.

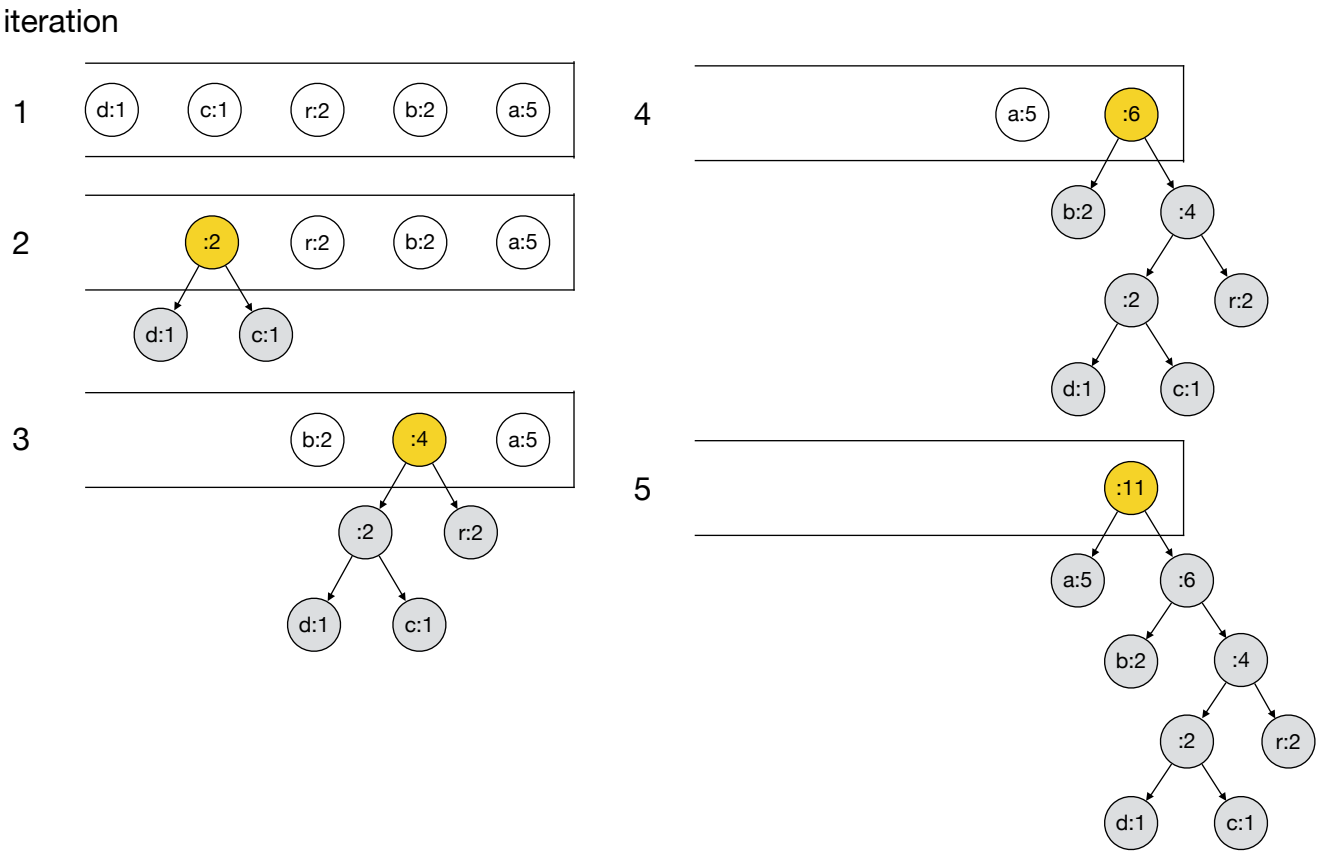
Pour construire l'arbre de Huffman, il faut ajouter à la définition d'un nœud un champs qui permet de stocker la fréquence (un entier). (Ce nombre indiquera la somme des fréquences de toutes les feuilles dans le sous-arbre ayant ce nœud comme racine.)

L'algorithme de Huffman

1. Créer une file de priorité pour stocker des nœuds priorisés par leur fréquence.
2. Pour chaque caractère X ayant une fréquence > 0,
   (a) créer un nœud portant le caractère X, sa fréquence et deux fils vides ;
   (b) ajouter le nœud à la file de priorité.
3. Tant que la file de priorité contient au moins deux arbres,
   (a) Extraire les deux arbres A_1 et A_2 avec les plus petites fréquences ;
   (b) Insérer un nœud A avec A_1 et A_2 comme fils gauche et droit et une fréquence égale à la somme des fréquences de A_1 et A_2.
4. Le seul arbre restant dans la file de priorité est l'arbre de Huffman.

Exemple

Le diagramme ci-dessous illustre l'exécution de l'étape 3 de l'algorithme pour le texte d'entrée `abracadabra` qui a les fréquences `a` : 5, `b` : 2, `c` : 1, `d` : 1 et `r` : 2. Dans chaque itération, les deux nœuds les plus à gauche sont extraits et un nouveau nœud (en jaune) est créé et réinséré.



4 Écriture de l'arbre

4.1 Encodage

▷ Modifier votre encodeur pour qu'il écrive également la représentation de l'arbre.

Votre encodeur devrait maintenant : lire le fichier une fois pour récupérer les fréquences, construire l'arbre de Huffman, écrire l'arbre de Huffman sur le fichier de sortie, ensuite écrire le nombre de caractères du fichier d'entrée et finalement encoder le fichier d'entrée et écrire les mots de code sur le fichier de sortie.

Par exemple, la sortie pour la chaîne de caractères `abracadabra` sera

```
10a10b110d0c0r
11
0 10 111 0 1101 0 1100 0 10 111 0
```

Notez que si votre fichier d'entrée contient un caractère `'\n'`, alors la représentation de l'arbre ne sera pas écrite sur une seule ligne.

4.2 Décodage

▷ Écrire un décodeur qui marche avec le nouvel encodeur.

5 Extensions possibles

Suppression des espaces

Les espaces entre les mots de code ne sont pas nécessaires pour le décodage. En arrivant à une feuille, on sait qu'un nouveau mot commence immédiatement après. Donc, vous pouvez supprimer ces espaces.

Amélioration de la file de priorité

Utiliser une implantation de la file de priorité basée sur **un tas**. Cette structure de données sera décrite ultérieurement dans le cours.

Écriture et lecture binaire et de bits

Remplacer la lecture et l'écriture des fichiers de textes par la lecture et l'écriture des fichiers binaires. Vous lisez toujours des octets du fichier d'entrée mais vous écrivez les mots de code sur un nombre de bits variable. Les `'0'` et les `'1'` dans la représentation de l'arbre seront également remplacés par des bits. Le nombre de caractères du fichier sera écrit par sa représentation binaire (normalement 4 octets pour un entier de type `int`).