université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

Projet IPF
LDD2 IM

Projet

Compression de texte

1 Objectifs

Le projet a plusieurs buts :

— Découvrir des algorithmes intéressants, assez subtils, et fondamentaux sur les textes ;
— Placer la programmation fonctionnelle dans le cadre plus général de la programmation et donc mélanger du code fonctionnel avec du code faisant des effets de bords (en particulier utilisation de tableaux et entrées-sorties dans des fichiers) ;
— Acquérir quelques compétences de *génie logiciel* en particulier de programmation modulaire et collaborative dans le cadre d'un projet écrit en OCaml ;

2 Sujet

On souhaite écrire un programme permettant de compresser des fichiers textes, en utilisant la méthode de Huffman (David A. Huffman, 1925-1999, informaticien Américain). Le programme `huff` que l'on souhaite écrire devra avoir les caractéristiques suivantes :

— `huff --help` : affiche un message d'aide sur les différentes options
— `huff fichier` : compresse le fichier donné en argument pour produire un fichier `fichier.hf`
— `huff fichier.hf` : décompresser le fichier donné en argument pour produire un fichier `fichier`
— `huff --stats fichier` : compresse le fichier mais affiche aussi des statistiques sur ce dernier

D'autres fonctionalités seront ajoutées en fin de projet dans un esprit *agile* (ajouter en peu de temps une fonctionalité sur un projet exsitant).

2.1 Présentation de l'algorithme de Huffman

L'algorithme de Huffman repose sur un constat simple. Traditionnellement, un caractère est stocké sur un nombre fixe de bits (typiquement 8) [1]. Cependant, si des caractères reviennent très souvent, on peut décider de leur donner un code plus court et en échange un code plus long aux caractères moins fréquents. De plus, un caractère non présent (par exemple « ê » dans un texte en anglais) n'aura même pas besoin d'avoir un code.

Par exemple, considérons le mot « satisfaisant ». Ce mot contient 12 caractères, donc codé en ASCII 8bits, il occupe $12 \times 8 = 96$ bits. Si on choisit un encodage plus malin, par exemple celui ci :

caractère	`a`	`s`	`f`	`n`	`i`	`t`
code binaire	`01`	`10`	`000`	`001`	`110`	`111`

Alors le mot « satisfaisant » est codé (en binaire) par `10 01 111 110 10 000 01 110 10 01 001 111`, soit 30 bits. Avant de comprendre comment sont attribués ces codes pour ce mot, on peut observer qu'avec les codes choisis, le décodage, en partant du début n'est pas ambigu. En effet, si on dispose de la table de correspondance précédente, alors :

— on commence par lire un `1`. Les seuls caractères possibles à ce stade sont `s`, `i` et `t` (les seuls dont le code commence par un `1`).
— on lit ensuite un `0`. Le seul caractère correspondant est `s` (car le `i` et le `t` se poursuivent par un `1`).

1. Tout ce qui est expliqué dans la suite s'applique aussi tel quel à l'encodage UTF-8 où chaque caractère est codé par un certain nombre d'octets

— **on peut donc afficher** `s`. On sait maintenant que le prochain bit sera le début d'un nouveau caractère.
— on lit un `0`. Les caractères possibles sont `a`, `f` et `n`.
— on lit ensuite un `1`, le seul caractère correspondant à `01` est `a`. **On peut afficher** `a`.

L'encodage choisi à une propriété très forte : aucun code complet n'est le *prefixe* d'un autre code. Par exemple, si on prend le code de `a` (`01`) alors on remarque qu'aucun autre code de caractère ne commence par `01`. De tels codes sont appelés des codes préfixes. Ils jouent un rôle important en télécommunication car ils permettent, comme nous l'avons fait ici, d'utiliser un encodage à taille variable sans avoir besoin de séparateur entre les codes.

Si on suppose un tableau de codes donné (D1), alors l'algorithme de Huffman est relativement simple.

Compression pour compresser un fichier, il suffit de le lire caractère par caractère, de trouver le code correspondant dans le tableau et d'écrire ce code dans la sortie (D2).

Décompression pour décompresser un fichier, il suffit de le lire bit par bit en accumulant les bits à la suite. Après chaque lecture, on peut regarder dans le tableau à quelle lettre correspond la suite de bits déjà lu. Il se peut qu'elle corresponde à un unique caractère (comme `01`) dans ce cas, on l'a décodé. Soit elle correspond à un début d'une autre séquence (par exemple `11` peut correspondre au début du code du `i` ou du `t`) et dans ce cas, on doit continuer de lire des bits (D3).

Évidemment, dans la description générale ci-dessus se cachent plusieurs difficultés (D1, D2, D3).

— D1 : comment calculer un tel tableau ?
— D2 : sur n'importe quel OS raisonnable, un *fichier* est une collection d'octets (pas de bits). Ainsi si notre message compressé fait un nombre de bits qui n'est pas un multiple de 8, il faut trouver un moyen pour le stocker de façon à la fois compacte et correcte. En effet, on ne veut pas stocker les caractères 1 (8 bits, code ASCII 49) et 0 (8 bits, code ASCII 48) dans le fichier, car on multiplierait inutilement par 8 l'espace de stockage.
— D3 : en admettant que l'on soit capable de lire un fichier *bit à bit* (et pas octet par octet), alors rechercher, après chaque bit lu s'il existe un tel code dans le tableau (pour retrouver le caractère de départ) est inefficace.

Une autre difficulté (D4) est que le tableau de décodage est nécessaire pour décompresser le fichier, et qu'il dépend du texte initial (par exemple le mot « satisfaisant » et le mot « abracadabra » n'ont pas les même tableaux). Il faut donc en plus pouvoir stocker ce tableau, avec la version compressée du texte, de la façon la plus compacte possible.

2.2 Arbre de Huffman

L'ingéniosité de l'algorithme de Huffman consiste en l'utilisation d'une structure de donnée plus adaptée qu'un tableau pour représenter la correspondance entre caractère et code, *un arbre de Huffman*. Cette structure est une *arbre binaire* c'est à dire que chaque nœud de l'arbre possède exactement deux fils (nœud interne) ou aucun (feuille). L'arbre de Huffman du mot « satisfaisant » est donné à la figure 1. Dans cet arbre, les caractères du texte sont tous positionnés aux feuilles, les nœuds internes ne contenant pas de donnée. Le code d'un caractère est donné par son chemin depuis la racine, avec la convention que prendre le sous-arbre gauche correspond à un `0` et prendre le sous-arbre droit correspond à un `1`. Comme on le voit dans la figure, le chemin vers le caractère `i` est bien `110`, ce qui correspond au code que l'on souhaitait.

On peut se demander comment construire un tel arbre. L'algorithme procède en plusieurs étapes.

1. Calculer le nombre d'occurrences de chaque caractère du texte. Pour notre exemple, cela donne :

$$H = (3, \texttt{s})\ \ (3, \texttt{a})\ \ (2, \texttt{t})\ \ (2, \texttt{i})\ \ (1, \texttt{f})\ \ (1, \texttt{n})$$

2. Si la collection H est vide, c'est que le texte était vide, on n'a rien à faire (on ne peut pas compresser un tel texte).
3. Sinon, tant que la collection H possède au moins deux éléments :

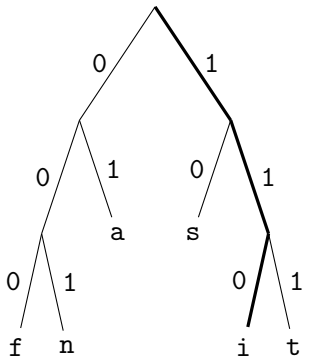


FIGURE 1 – Arbre de Huffman du mot satisfaisant avec le code de `'i'` en gras.

(a) prendre les deux plus petits (en considérant le nombre d'occurrences)
(b) les retirer de la collection
(c) rajouter un nœud binaire (interne au dessus) et faire la somme des occurrences
(d) replacer cet arbre et son nombre d'occurrence dans la collection

4. Lorsque la collection n'a qu'un seul élément, ce dernier est l'arbre recherché

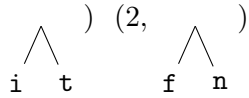
Ainsi, en partant de la collection

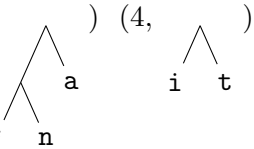
$$H = (3, \texttt{s})\ \ (3, \texttt{a})\ \ (2, \texttt{t})\ \ (2, \texttt{i})\ \ (1, \texttt{f})\ \ (1, \texttt{n})$$

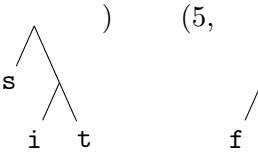
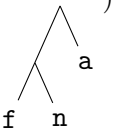
on effectue les opérations suivantes :

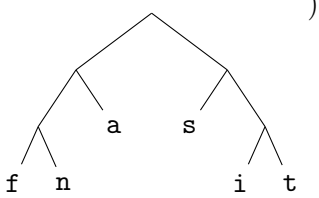
$(3, \texttt{s})\ \ (3, \texttt{a})\ \ (2, \texttt{t})\ \ (2, \texttt{i})\ \ (1, \texttt{f})\ \ (1, \texttt{n})$ (départ)

$(1+1,$  $)\ \ (3, \texttt{s})\ \ (3, \texttt{a})\ \ (2, \texttt{t})\ \ (2, \texttt{i})$ (regroupement des deux min.)

$(2+2,$  $)\ \ (2,$ $)\ \ (3, \texttt{s})\ \ (3, \texttt{a})$ (regroupement des deux min.)

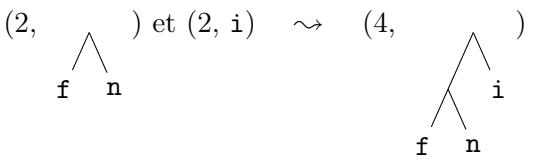
$(2+3,$  $)\ \ (4,$ $)\ \ (3, \texttt{s})$ (regroupement des deux min.)

$(3+4,$  $)\ \ (5,$  $)$ (regroupement des deux min.)

$(5+7,$  $)$ (regroupement des deux min.)

(fin)

S'il y a plus que deux éléments de valeurs minimales, on peut en choisir deux quelconques parmis les arbres minimaux. Par exemple à l'étape 2 précédente, on aurait pu choisir de de créer :



L'arbre final aurait été différent, les codes de chaque caractères auraient été différents, mais *la taille du texte compressé final aurait été la même* (30 bits dans le cas de « satisfaisant »).

Remarque un sous-problème (D5) est de penser à une structure de données adaptée pour faire les opérations précédentes :

— garder une collection d'objet ayant un « poid »
— pouvoir retirer efficacement le minimum d'une telle collection
— pouvoir ajouter un élément à la collection
— avoir la taille ou tester si la collection est vide ou singleton

3 Organisation du travail

Le but du projet est de vous donner une grande autonomie tout en ne laissant personne de côté. Ainsi, on ne donne pas plus d'explication technique pour l'instant, mais chaque semaine des suggestions seront proposées (sous forme d'un petit énoncé de TP, facultatif). Nous suggérons de progresser de la façon suivante :

1. Mise en place des fichiers du projet, lecture du sujet. Dans cette phase il faut apprivoiser quelques aspects impératifs d'OCaml (tableaux, lecture de fichiers). Un premier objectif peut être d'arriver à compter pour chaque caractère d'un fichier texte son nombre d'occurrences (et réfléchir dans quelle structure stocker ce résultat).
2. À partir de la collection des caractères et fréquences, construire l'arbre de Huffman.
3. Compresser le texte et *sauvegarder dans un fichier* l'arbre et le texte compressé (donc réfléchir à une façon de sauvegarder l'arbre).
4. Décompresser un texte compressé : lire l'arbre puis, cet arbre étant chargé, lire la suite de bits et la décompresser.
5. Une fois les algorithmes de base mis en place, travailler sur le programme principal (gestion de la ligne de commande en particulier).

4 Fichiers fournis

L'archive proposée sur le site du cours est composée comme suit :

— Un fichier `huff.ml` qui contiendra le fichier principal.
— Deux fichiers `heap.mli` et `heap.ml` contenant l'interface de la structure de donnée pour le sous-problème (D5). Évidemment le fichier `heap.ml` doit être complété par vos soins.
— Deux fichiers `bs.ml` et `bs.mli`. Ces derniers contiennent une petite bibliothèque permettant de lire et écrire un fichier bit à bit (ou n bits par n bits). La lecture du fichier `bs.mli` est vivement recommandée quand vous en serez à la phase de lecture/écriture des fichiers compressés.
— Un fichier `huffman.ml`. Ce dernier contient deux fonctions, `compress` et `decompress` (pour l'exemple, ce fichier est évidemment à enrichir, compléter)
— Des fichiers `dune`, `dune-project` et `dune-workspace` permettant de construire votre projet avec la commande `dune`. Il ne faut pas modifier ces fichiers.

Votre projet final doit comporter **au moins** ces fichiers, mais vous pouvez évidemment l'enrichir d'autres fichiers auxiliaires et organiser votre code comme vous le souhaitez.

5 Modalités

Le projet est à faire en **binôme**. Les rendus se feront obligatoirement via `git`. Il est demandé d'utiliser exclusivement le `gitlab` FramaGit (cf. Feuille de TP 1) :

`https://framagit.org/`

Il vous est demandé d'ajouter à vos projets les trois personnes suivantes :

- `Citodas`
- `RVilmart`
- `knguyen`

En leur donnant le rôle de développeur. Le barême précis sera donné prochainement, mais les points pris en compte seront :

- la qualité du code
- la qualité des commits et de l'organisation
- le rapport
- les tests