Files de priorités, Arbres de Huffman

1 Introduction

Ce TP vise à coder un compresseur de fichiers texte par les arbres de Huffman. Alors que certaines lettres de l'alphabet sont utilisées plus fréquemment que d'autres, un codage ASCII des caractères implique que chaque lettre consomme le même espace : 1 octet. Il semble donc intéressant de remplacer le codage ASCII par un codage dont la taille varie en fonction de la fréquence d'utilisation de chaque caractère, de manière à ce que les caractères fréquents consomment peu de bits par rapport aux caractères plus rares. Les arbres de Huffman permettent justement de trouver un tel codage, qui est optimal pour un texte donné. [1] Le principe de ces arbres est détaillé section 2 et leur construction nécessite l'utilisation de files de priorités.

Le TP est donc l'occasion de manipuler des arbres, des files de priorité ainsi que des dictionnaires. La section 4 détaille les différentes fonctions que vous avez à programmer. Le programme final permettra de compresser / décompresser des fichiers texte. Un certain nombre de fichiers sont fournis afin d'éviter de passer trop de temps sur la lecture/écriture de fichiers. (voir section 3).

2 Arbres de Huffman

Dans toute la suite du document, on considère donc le codage de Huffman d'un texte donné. Par abus de langage, on appelle "*fréquence*" d'un caractère le nombre d'occurrences de ce caractère dans le texte (il suffit donc de diviser ce nombre par le nombre total de caractères pour retrouver la notion usuelle de "*fréquence*").

2.1 Utilisation

Un arbre de Huffman (voir figure 1) est un arbre de préfixes sur l'alphabet $\{0, 1\}$, pour lequel chaque feuille contient un des caractères à coder/décoder. Par définition, un arbre de préfixes sur $\{0, 1\}$ est un arbre tel que :

- chaque nœud représente un mot binaire (e.g. sur $\{0, 1\}$)
- la racine représente le mot vide (noté ici ϵ)
- le fils d'un nœud représente $p.c$ où p est le mot représenté par le père, et c un chiffre de $\{0, 1\}$. Ici, par convention, $c = 0$ pour un fils gauche et $c = 1$ pour un fils droit.

A titre d'exemple, on peut considérer l'arbre de Huffman de la figure 1 où on a représenté dans chaque nœud le mot binaire qu'il représente. Sur les feuilles, ce code apparaît à gauche du "⇒", tandis que le caractère associé apparaît à droite. Il est alors possible de coder un texte comme flux binaire et inversement. Par exemple, le texte "acbeceea" est codé comme la concaténation des codes de chaque lettre, soit "0100100010011101". Ici, il faudra prévoir une structure efficace de dictionnaire pour associer à chaque caractère son code dans l'arbre de Huffman (voir section 4).

Inversement, un flux comme "1100101" est décodé par des parcours successifs de l'arbre de Huffman. En effet, celui-ci correspond directement à un automate fini déterministe reconnaissant

1. Plus exactement, le codage d'une suite de caractères X par Huffman est un codage de taille minimale (en nombre de bits) de X parmi les codes associant à chaque caractère apparaissant dans X une suite fixée de bits. La preuve que le code de Huffman est optimal n'est pas traitée ici (mais se trouve dans les ouvrages de référence en Algorithmique).

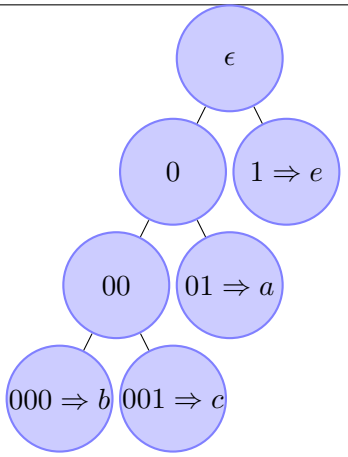

FIGURE 1 – Arbre de Huffman

les codes binaires des caractères : les nœuds sont les états de l'automate, les transitions sont les arêtes, l'état initial est la racine et les états finaux sont ici les feuilles. À chaque passage sur une feuille le caractère correspondant est affiché, puis on repart de la racine de l'arbre pour le caractère suivant. Au final, ce flux correspond au texte "*eeca*".

Grâce à l'arbre de Huffman, on retrouve le caractère associé à un code avec un coût en temps linéaire en fonction du nombre de bits de son code. Il faut noter que l'implémentation de cet arbre peut être assez compacte en mémoire puisqu'il est inutile de stocker le mot représenté par les nœuds : on recalcule ce mot au fur et à mesure du parcours.

La construction des arbres de Huffman (voir section 2.2) garantit que les caractères les plus fréquents se retrouvent sur une feuille de faible profondeur et donc avec un code binaire court. De plus, un arbre de Huffman est en fait un arbre binaire localement complet.

2.2 Construction

Un arbre de Huffman se construit en partant des feuilles. On travaille par fusions successives jusqu'à obtenir la racine. Afin que les éléments fréquents soient liés à un code court, ces éléments sont insérés en dernier dans l'arbre. Pour ce faire, nous utilisons une file de priorités : les données dans la file sont des arbres, et la priorité de chaque arbre correspond à la somme des nombres d'occurrences (dans le texte à compresser) des caractères apparaissant sur ses feuilles.

Initialement la file contient toutes les feuilles (avec une priorité égale à la fréquence de leur caractère). Ensuite, on récupère grâce à la file les 2 arbres de priorité minimale, on crée un nouveau sommet intermédiaire et on attache chacun des deux arbres comme fils du sommet créé. Les 2 arbres sont enlevés de la file et le nouvel arbre obtenu est inséré dans la file (avec une priorité égale à la somme des priorités de ses 2 fils). Par cette opération, le nombre d'arbres de la file diminue exactement de 1. On réitère l'opération jusqu'à ce qu'il ne reste plus qu'un arbre dans la file : c'est l'arbre de Huffman cherché.

Prenons par exemple la construction de l'arbre de Huffman correspondant au mot "hello". On a ici 4 lettres différentes : h, e, l, o. Nous créons donc 4 sommets, chacun associé à une lettre et nous les insérons dans la file de priorité avec comme priorités : 1,1,2,1. La lettre l apparaissant 2 fois, sa priorité est donc de 2. La figure 2 détaille les différentes étapes suivantes. Dans un premier temps nous sélectionnons deux arbres de priorité minimale, ici h et e de priorité 1 (notons au passage que différents choix sont possibles). Ces arbres sont alors rattachés à un nouveau sommet interne, ce qui laisse 3 arbres de priorité 2, 2 et 1 dans la file. A l'étape

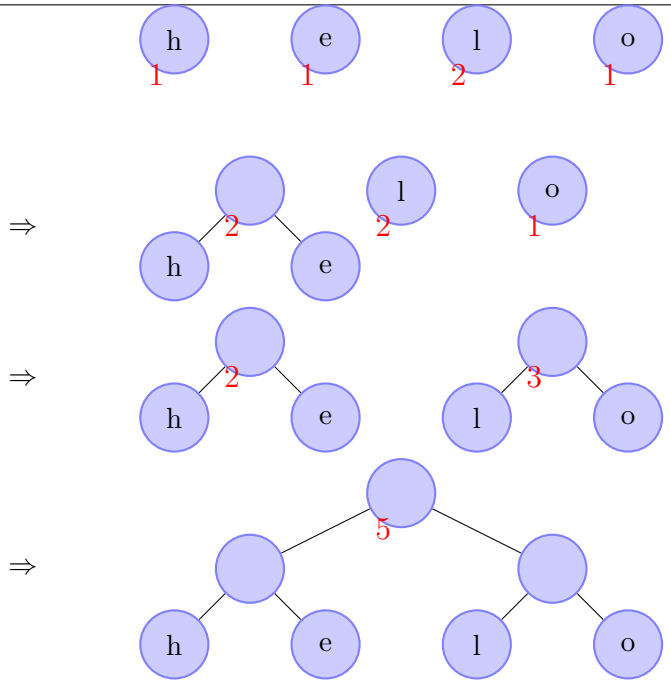

FIGURE 2 – Construction d'un arbre de Huffman

suivante, l'arbre de priorité 1 fusionne avec un des arbres de priorité 2, et le processus continue jusqu'à ce qu'il ne reste plus qu'un seul arbre dans la file.

3 Code fourni

Un certain nombre de fichiers sont fournis. Certains forment des paquetages complets :

- *comparaisons.ads*, *comparaisons.adb* : définit des fonctions de comparaisons pour quelques types de base (ne pas éditer)
- *file_priorite.ads* : interface avec une file de priorité (générique) (à implémenter)
- *test_file.ads* : vérification que les files de priorité fonctionnent
- *arbre_huffman.adb* : le fichier principal à modifier : contient les arbres et différentes structures annexes
- *huffman.adb* : le fichier principal (à modifier éventuellement pour extensions)

Commençons par détailler les procédures fournies par le fichier *file_priorite.ads* listées ci-dessous. La fonction `Nouvelle_File` crée une nouvelle file de priorité d'une taille maximale donnée en argument. La procédure `Insertion` insère une donnée D de priorité P dans la file. La procédure `Meilleur` retourne le meilleur élément (priorité + donnée) ainsi qu'un booléen valant faux si la file est vide. La procédure `Suppression` supprime le meilleur élément de la file. La programmation du fichier *.adb* correspondant est à votre charge.

```
function Nouvelle_File(Taille: Positive) return File;
procedure Insertion(F:in out File; P:Priorite; D:Donnee);
procedure Meilleur(F: in File; P: out Priorite;
                   D: out Donnee; Statut: out Boolean);
procedure Suppression(F: in out File);
```

Le paquetage *arbre_huffman* est plus complexe. Nous y trouvons tout d'abord les différent types de données ci-dessous :

```
type Noeud;
type Arbre is access Noeud;

subtype ChiffreBinaire is Integer range 0..1 ;
type TabFils is array(ChiffreBinaire) of Arbre ;

type Noeud(EstFeuille: Boolean) is record
   case EstFeuille is
      when True => Char : Character;
      when False =>
         Fils: TabFils;
         -- on a: Fils(0) /= null and Fils(1) /= null
   end case ;
end record;

type Tableau_Ascii is array(Character) of Natural;

type TabBits is array(Positive range <>) of ChiffreBinaire ;
type Code is access TabBits;
```

Ces définitions commencent avec le type `Arbre` des arbres de Huffman. Ces arbres étant des arbres binaires localement complets, un nœud est donc soit un nœud interne avec 2 fils, soit une feuille qui contient un caractère. On utilise ici un type enregistrement `Noeud` avec discriminant pour distinguer ces 2 types de nœuds.

Le type `Tableau_Ascii` sert à représenter la fréquence (e.g. le nombre d'occurrences) de chaque caractère : la $i^{\text{ème}}$ case du tableau contient le nombre d'apparitions du caractère i. Enfin, le type `Code` représente les mots binaires associés aux caractères dans le codage de Huffman.

Enfin, le fichier *huffman.adb* contient les procédures de compression et de décompression. Le programme principal de compression est le suivant :

```
  Lecture_Frequences(Fichier_Entree, Frequences, Taille);
  Affiche_Frequences(Frequences);
  Arbre_Huffman := Calcul_Arbre(Frequences);
  Affiche_Arbre(Arbre_Huffman);
  D := Calcul_Dictionnaire(Arbre_Huffman);
  Create(Sortie, Out_File, Fichier_Sortie);
  SAcces := Stream( Sortie );
  Natural'Output(Sacces, Taille);
  Tableau_Ascii'Output(Sacces,Frequences) ;
  Open(Entree, In_File, Fichier_Entree);
  EAcces := Stream(Entree);
  Reste := null;
  while (not End_Of_File(Entree)) or Reste /= null loop
   Recuperation_Caractere(Reste, Entree, EAcces, Caractere_Sortie, D);
   Character'Output(SAcces, Caractere_Sortie);
  end loop;
  Close(Entree);
  Close(Sortie);
```

Ce programme commence par lire les fréquences d'apparition de chaque caractère dans le fichier d'entrée. Il affiche les fréquences trouvées puis calcule l'arbre de Huffman correspondant. Chaque code de chaque lettre est ensuite stocké dans un dictionnaire. L'arbre ne nous est alors plus utile. Nous créons alors le fichier de sortie. Ici, nous allons remplir ce fichier en utilisant la bibliothèque `Ada.Streams` qui est présentée sur

`http://ensiwiki.ensimag.fr/index.php/Streams_en_Ada`

Ce mécanisme permet de lire et d'écrire n'importe quelle type de valeur Ada dans un fichier binaire. Nous commençons par écrire la taille du fichier d'entrée ainsi que les fréquences d'apparition de chaque caractère. En effet la décompression nécessitera de connaître le code utilisé lors de la compression et donc l'arbre de Huffman. Plutôt que de stocker l'arbre nous stockons ici directement les fréquences. L'arbre sera alors reconstruit lors de la décompression. [2]

Le programme boucle alors sur le fichier d'entrée et encode chaque caractère dans le fichier de sortie. La difficulté principale vient ici du fait qu'on ne peut pas écrire bit-à-bit dans le fichier. On pourrait éventuellement écrire directement des valeurs de type `TabBits`, mais ce serait assez inefficace, car chaque "bit" de type `ChiffreBinaire` est en fait un `Integer` donc un entier sur 4 octets (et même si on utilisait le type `Boolean` au lieu `ChiffreBinaire`, ça prendrait encore un octet). On est donc en fait obligé d'écrire octet par octet dans le fichier. Comme chaque code ne tient pas forcément sur un multiple de 8 bits, on introduit la variable `Reste` qui contient ici le morceau de code plus petit que 8 bit que n'a pas pu être écrit jusqu'à présent. Ici on utilise le type `Character` pour représenter les octets à écrire dans le fichier.

4 Objectifs du TP

Votre objectif principal est d'obtenir un programme final fonctionnel qui permette de compresser et de décompresser n'importe quel fichier texte. Le code rendu devra être lisible et commenté (éventuellement mieux que le code fourni). On détaille ci-dessous les différentes procédures à modifier ou à implémenter. On vous suggère également des optimisations à réaliser : il ne faut les traiter que lorsque votre programme est déjà fonctionnel.

- la file de priorité : vous devez écrire le fichier *file_priorite.adb* en implantant un tas.
- `Affiche_Arbre` : procédure d'affichage de l'arbre sur la sortie standard
- `Calcul_Arbre` : procédure de calcul de l'arbre de Huffman à partir des différentes fréquences
- `Calcul_Dictionnaire` : insertion de tous les codes de l'arbre dans un dictionnaire
- `Decodage_Code` : cette procédure est utilisée pour la décompression et lit le flux de bits en entrée jusqu'à avoir réussi à décoder un caractère. Il vous est demandé de l'optimiser en évitant d'allouer et désallouer des données en permanence.
- `Compression/Decompression` : au lieu de stocker la table des fréquences dans le fichier compressé, on peut réfléchir comme optimisation à une façon compacte d'encoder directement l'arbre de Huffman. [3]
- enfin, une dernière optimisation plus complexe, serait de couper le fichier en tampons et d'utiliser des arbres différents pour chacun d'entre eux.

2. Lorsque votre programme fonctionnera vous pourrez revenir sur ce choix, afin de gagner un peu de place (voir section 4).

3. Typiquement, en utilisant un parcours en profondeur préfixe des nœuds, on peut coder chaque arbre de Huffman par un mot binaire unique. On code une feuille sur 9 bits, sous la forme d'un 1 suivi du code ASCII du caractère de cette feuille. On code un nœud interne par le bit 0 suivi du codage de chacun de ses fils. Par exemple, l'arbre de la figure 1 est codé par le mot "0001b1c1a1e" où il faut remplacer a, b, c et e par leur code ASCII sur 8 bits. Comme dans un arbre localement complet à f feuilles, il y a exactement $f-1$ nœuds internes, il faudra au total $10f-1$ bits pour coder l'arbre via cette technique (où f est le nombre de caractères qui ont une fréquence non-nulle) contre 32×256 via la table des fréquences (si on compte qu'il y a 256 caractères dans le code ASCII et qu'un entier prend 32 bits).

Chaque équipe doit déposer sur Teide **avant le jeudi 14 mai 2015 à 23h59** une archive `tar.gz` contenant :

- Les sources complètes du TP (y compris les sources fournies et les éventuels pilotes de tests développés pour l'occasion).
- Un rapport au format **pdf** de 4 pages maximum présentant :
  1. brièvement, l'état d'avancement du TP et le contenu de l'archive ;
  2. éventuellement, quelques explications sur l'implémentation, à condition qu'elles ne figurent pas déjà dans le sujet, et qu'elles semblent nécessaire pour comprendre les sources commentées ;
  3. une présentation détaillée de la démarche de validation (tests de correction) et d'expérimentation (tests de performances : rapidité d'exécution et gains/pertes de place due à la compression de fichier).