COMPRESSION DE FICHIERS TEXTE PAR CODAGE DE HUFFMAN

1 Introduction

Il est facile de remarquer que, dans un texte, certains caractères (lettres de l'alphabet, espaces, ...) sont utilisés plus fréquemment que d'autres. En revanche, le codage ASCII d'un caractère dans un fichier utilise toujours le même espace mémoire : 1 octet. L'encodage d'un texte de n caractères nécessite toujours donc n octets. Une solution pour *compresser* un texte est d'utiliser un codage (dit entropique) basé sur l'utilisation de codes de longueur variable, en nombre de bits, en privilégiant des codes courts pour les caractères fréquents. Dans un texte en français, le gain de place est généralement de 40 à 60% [1].

L'objectif du TP est de coder un compresseur/décompresseur de fichiers texte. La génération d'un code de longueur variable sera faite grâce à l'algorithme du *codage de Huffman*.

2 Arbres de Huffman

2.1 Définition et utilisation

Un arbre de Huffman est un arbre binaire, généralement déséquilibré, dans lequel chaque feuille contient un des caractères à coder/décoder. Les noeuds internes ne portent pas d'information pertinente. Le code d'un caractère est obtenu en suivant le chemin de la racine jusqu'au noeud feuille qui le contient. Chaque fois que l'on descend par le fils gauche d'un noeud on ajoute un bit de valeur 0, et chaque fois que l'on descend par le fils droit on ajoute 1.

Considérons par exemple l'arbre de Huffman de la figure 1. Sur cet arbre, r est la racine, I_1 et I_2 sont deux noeuds internes et a, b, c, e les caractères que nous cherchons à coder. Le chemin de la racine vers c est : r, I_1, I_2, c. I_1 est le fils gauche de r, I_2 est le fils gauche de I_1 et c est le fils droit de I_2. gauche, gauche, droit $\Rightarrow$ le codage de c est donc 001. De manière similaire, on trouve les codes 01, 1 et 000 pour a, e et b.

Il est possible de coder un texte complet comme un flux binaire et inversement. Par exemple, le texte "acbeceea" est codé comme la concaténation des codes de chaque caractère, soit "0100100010011101". Seuls 16 bits sont ici nécessaires pour coder le texte, au lieu de 8 octets avec un codage ASCII

1. D'autres applications utilisent ce principe de codes de longueur variable, par exemple pour stocker les *codebook* (tables des codes) dans les formats `jpeg` ou `oog`.

Inversement, un flux comme "1100101" est décodé par un parcours de l'arbre. A chaque passage sur une feuille le caractère correspondant est récupéré. On se place initialement sur la racine puis on se déplace en fonction de la valeur de chaque bit : 1 ⇒ déplacement vers e. Puisque c'est une feuille on récupère le caractère e et on repart de la racine. Second bit : 1 ⇒ e encore une fois. Troisième bit : 0 ⇒ déplacement vers I_1. Quatrième bit : 0 ⇒ déplacement vers I_2. Cinquième bit : 1 ⇒ déplacement vers c. On récupère c puis on repart de la racine... Au final on décode ainsi : *eeca*.

Il est important de noter que l'arbre utilisé pour décoder doit nécessairement être le même que celui ayant servi au codage !

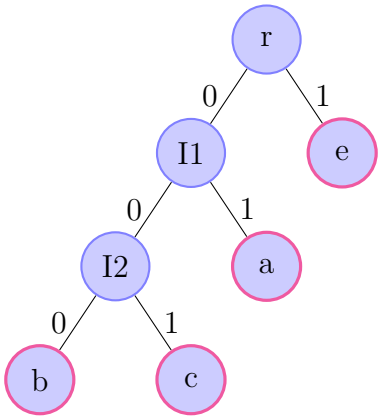

FIGURE 1 – Exemple d'arbre de Huffman

Un code de Huffman est dit *préfixé* car par construction aucun code de caractère, considéré ici comme une suite de bits, n'est le préfixe d'un autre code. Autrement dit, si on trouve une certaine séquence de bits dans un message, et que cette séquence correspond à un caractère, elle correspond forcément à ce caractère et ne peut pas être le début d'un autre code. Ainsi, il n'est pas nécessaire d'avoir des "séparateurs" entre les caractères même s'ils n'ont pas tous la même taille, ce qui est ingénieux. Par contre, le droit à l'erreur n'existe pas : si un bit est perdu en route, tout le flux de données est perdu et on décodera n'importe quoi.

2.2 Génération de l'arbre de Huffman

Un arbre de Huffman se construit en partant des feuilles, puis par fusions successives jusqu'à obtenir un arbre unique. Afin que les caractères les plus fréquents soient associés à un code court, ils sont insérés en dernier dans l'arbre. Ainsi, ils seront le plus près possible de la racine.

A chacun des caractères du texte à coder est donc associé un *poids*, correspondant au nombre d'occurences de ce caractère dans le texte complet. L'ordre d'insertion des noeuds est ensuite défini par une *file à priorités*.

2.2.1 Le type "file à priorités"

Le type "file à priorités" est une généralisation des files et piles déjà rencontrées. Il permet de stocker un ensemble de données, auxquelles est associée une priorité. C'est cette priorité, et non pas la date d'insertion dans la file, qui déterminera l'ordre de sortie des données. Ainsi, le prochain élément à sortir est celui de priorité maximale parmi les éléments présents dans la file à priorités.

Exemples :
- file à priorités générale : on entre `'a'` avec une priorité de 4, puis `'b'` avec une priorité de 6, `'c'` avec une priorité de 1, et `'d'` avec une priorité de 3. Les éléments sont retirés successivement : `'b'` sort d'abord, puis `'a'`, `'d'` et `'c'`
- si on veut utiliser une file à priorités pour réaliser une pile (modèle `LIFO`), on va rentrer par exemple `'a'` avec une priorité de 1, puis `'b'` de priorité 2, `'c'` de priorité 3 et `'d'` de priorité 4. Ainsi les éléments seront dépilés dans l'ordre `'d'`, `'c'`, `'b'`, `'a'`.
- si on veut utiliser une file à priorités pour réaliser une file (modèle `FIFO`), on va rentrer par exemple `'a'` avec une priorité de 4, `'b'` de priorité 3, `'c'` de priorité 2 et `'d'` de priorité 1. Ainsi les éléments seront défilés dans l'ordre `'a'`, `'b'`, `'c'`, `'d'`

2.2.2 Construction de l'arbre

L'initialisation de l'algorithme consiste à créer les feuilles portant les caractères (il s'agit en fait d'arbres à un seul noeud). A chaque feuille est associée un poids entier p, égal au nombre d'occurences du caractère dans le texte. Toutes ces feuilles sont ajoutées dans une file de priorité. Dans ce contexte, l'élément le plus prioritaire est celui dont la valeur p est minimale (les caractères rarement utilisés sont insérés en premier, et seront portés par les feuilles les plus profondes).

Le principe est ensuite le suivant :
- les deux arbres a_1 et a_2 les plus prioritaires sont sortis de la file.
- un nouvel arbre est crée ayant comme fils a_1 et a_2. Sa racine ne porte pas de caractère significatif.
- cet arbre est inséré dans la file avec comme poids la somme des poids de ses deux fils.

Le nombre d'éléments dans la file diminue exactement de 1 à chaque étape. L'opération est réitérée jusqu'à ce qu'il ne reste plus qu'un seul arbre dans la file : il s'agit alors de l'arbre de Huffman recherché.

La figure 2 illustre la construction d'un arbre correspondant au texte `"veni vidi vici"`.

Il est important de noter que plusieurs arbres pourraient être générés à partir d'un même ensemble de caractères. Par exemple lorsque plusieurs noeuds ont même priorité, dans quel ordre sont-ils sortis de la file ? Les fils droit et gauche peuvent aussi être inversés. Certains

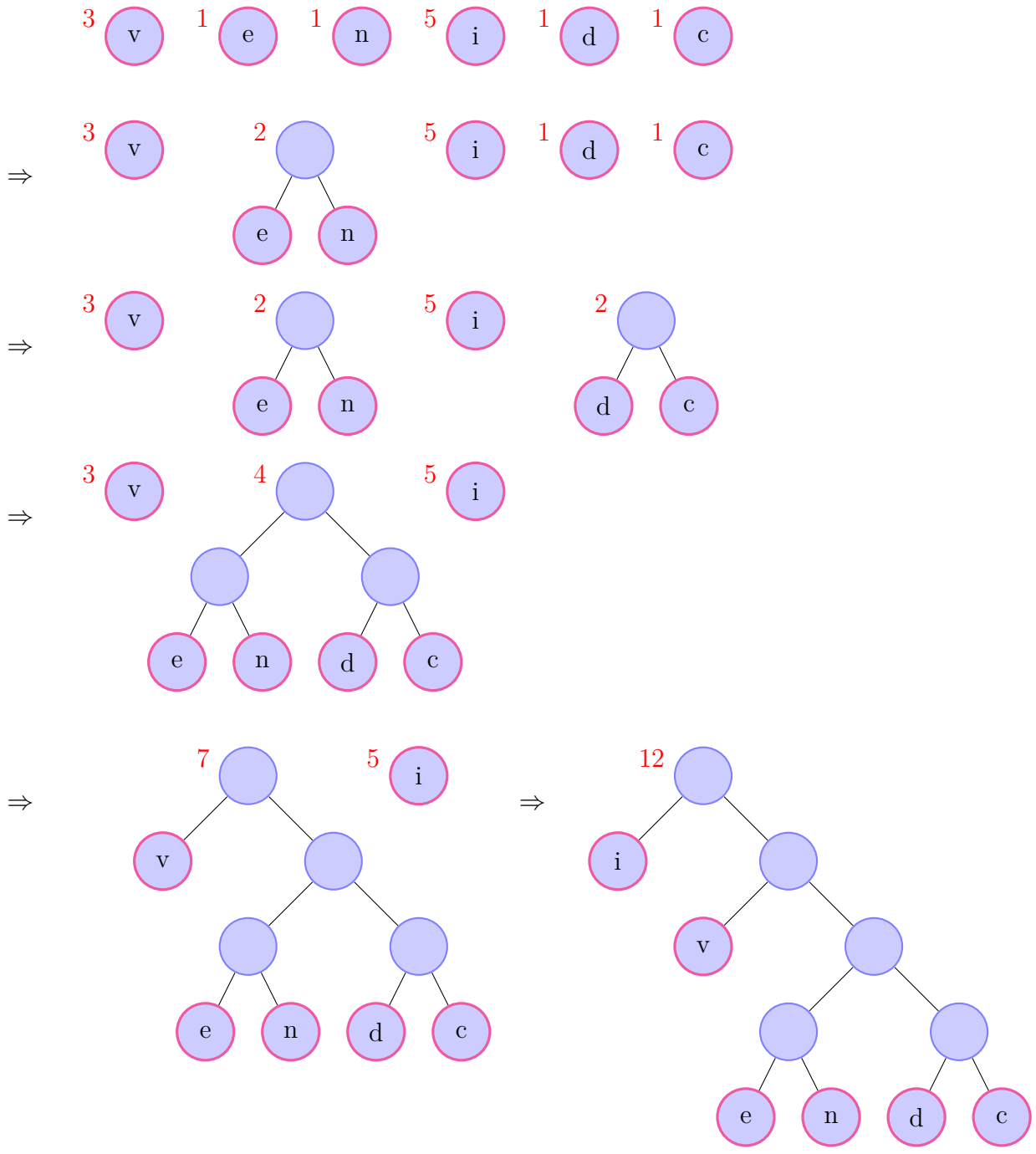

FIGURE 2 – Exemple de construction d'un arbre de Huffman pour le texte `veni vidi vici`.

arbres sont plus déséquilibrés que d'autres : y a-t-il une meilleure stratégie pour minimiser la longeur totale du texte codé ?

Toujours retenir qu'un code de huffman est *indissociable* de l'arbre ayant servi à le générer.

2.3 Stockage d'un arbre

Pour pouvoir décompresser un fichier, l'arbre ayant servi à le compresser doit être connu. Il doit donc être stocké dans l'entête du fichier compressé (ou en tout cas les données permettant de le reconstruire).

Solution simple Une solution triviale est de stocker les différents caractères avec leur fréquence d'apparition. Si le même algorithme de génération d'arbre est utilisé à la compression et à la décompression, l'arbre de Huffman sera bien identique.

Solution avancée Un autre format possible est celui utilisé dans les formats de compression *jpeg* ou *ogg-vorbis*, qui utilisent des codes de Huffman pour stocker non pas des caractères mais les dictionnaires des valeurs numériques de codage de l'information (*codebooks*).

Les codes de Huffman sont des codes préfixés. Une fois qu'un mot de code a été utilisé, il ne peut plus servir comme début à un autre mot. Si on fixe comme règle que les codes sont assignés avec d'abord le 0 (arbre gauche) puis le 1 (arbre droit), et que l'on donne **les symboles** et **leur longueur** dans l'ordre où ils sont rangés dans l'arbre on peut reconstruire l'arbre assez facilement. Prenons l'exemple de la figure 3, tiré de la norme *Vorbis*.

Symbole	Longueur
0	2
1	4
2	4
3	4
4	4
5	2
6	3
7	3

FIGURE 3 – Données stockées : les symboles à coder et la longueur de leur codage.

Pour construire les mots de code associés, on procède symbole par symbole en créant les feuilles d'un arbre binaire. On cherche toujours à créer la feuille le plus à gauche possible, à une profondeur égale à la longueur du mot de code.

On commence donc par le symbole 0, qui a un code de longueur 2. Comme on assigne toujours le bit 0 d'abord, le mot de code associé à 0 est `00`. L'entrée 1 a un mot de code de longueur 4. On ne peut plus commencer par `00`, le mot commence donc par 01, puis on

assigne les 0 d'abord. On obtient donc `0100`. En continuant on obtient les codes et l'arbre de la figure 4.

Symbole	Longueur	Code
0	2	`00`
1	4	`0100`
2	4	`0101`
3	4	`0110`
4	4	`0111`
5	2	`10`
6	3	`110`
7	3	`111`

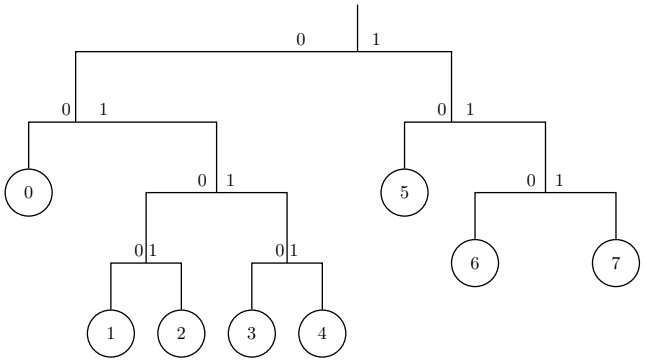

FIGURE 4 – Codes et arbre de Huffman correspondant.

3 Compression d'un fichier texte

3.1 Lecture du fichier source

La première étape consiste à lire l'ensemble des caractères du fichier à compresser et à calculer leur nombre d'occurences. L'arbre de Huffman peut alors être généré.

3.2 Codage des caractères

Le code des caractères présents dans l'arbre est déterminé en parcourant l'arbre à partir de la racine et en descendant jusqu'à chacune des feuilles. Le chemin parcouru depuis la racine donne directement le code du caractère porté par une feuille.

Pour éviter des parcours multiples, le plus simple est de traiter l'arbre une seule fois et de stocker le code associé à chaque caractère dans un dictionnaire. L'arbre lui-même n'est ensuite plus utile, les codes associés aux caractères sont directement lus dans le dictionnaire.

3.3 Encodage du fichier compressé

Entête Le fichier compressé contient tout d'abord un entête avec toutes les informations qui seront nécessaires pour la décompression. L'arbre est en particulier stocké, comme décrit section 2.3 ; vous êtes libre de choisir le format de stockage retenu (la version simplifiée ou celle utilisée en pratique).

Données compressées La seconde étape consiste à écrire la suite de bits codant l'ensemble des caractères du fichier d'origine. La difficulté principale vient du fait que les codes des différents caractères sont de longueur variable, alors qu'il n'est possible d'écrire que des octets (8 bits) dans le fichier de sortie.

4 Décompression d'un fichier

De manière opposée au codage, les principales étapes du décodage sont :

1. Lire les informations de l'entête du fichier compressé, pour reconstruire l'arbre de Huffman utilisé pour le codage ;
2. Décoder le flux de bits, et regénérer le texte initial.

5 Structures de données

Une structuration du projet est proposée, spécifiée dans plusieurs fichiers `.ads` :

`huffman.ads` une représentation de l'arbre de Huffman lui-même, basée sur structure arborescente. Ce package regroupe notamment les opérations de création d'un arbre partir à d'un fichier à compresser, de lecture/écriture de l'arbre dans l'entête d'un fichier compressé, et de décodage d'un caractère à partir d'une suite de bits.

`dico.ads` un *dictionnaire* des caractères, permettant de stocker les codes binaires associés à chaque caractère.

`code.ads` représentation d'un code binaire de longueur variable ;

`file_priorite.ads` une file de priorité générique, utilisée pour la construction de l'arbre de Huffman ;

`tp_huffman.adb` contient les deux fonctions principales `Compresse` et `Decompresse`, et gère le programme principal (ligne de commande).

Un fichier supplémentaire `exemple_io.adb` donne une méthode en Ada pour réaliser des lectures/écritures, en textuel et en binaire, dans des fichiers.

Ces packages ne sont qu'*une proposition*, vous êtes libres de les modifier selon vos besoins, d'ajouter de nouvelles structures ou de ne pas du tout les utiliser !

6 Objectifs & Travail à rendre

Les objectifs de ce TP sont :
— fournir une implémentation fonctionnelle des packages.
— le programme final devra être fonctionnel et permettre de compresser et de décompresser n'importe quel fichier texte.

Le TP est à réaliser en binôme. Devront être rendu sur `Teide` :
— L'ensemble de vos fichiers sources, lisibles et commentés.
— Eventuellement vos fichiers tests (selon leur taille, vous ne pourrez peut-être pas les poser sur `Teide`...)
— Un court rapport (4 pages maximum) pour expliquer et justifier vos structures de données et principaux algorithmes, leurs coûts, et présenter les résultats obtenus.

La date de rendu du TP est fixée au **Venderdi 8 décembre 2017**.

Tout y est. Bon travail à vous tous !