Codage de Huffman

Anas SEGHROUCHNI, Emile JEANDON

Équipe EF11

Ce rapport a pour objectif de synthétiser la démarche utilisée pour aborder le projet de PIM du premier semestre, traitant du codage de huffman et de son application direct sur la compression/décompression de fichiers. Il permettra de présenter nos choix effectués : l'architecture et l'application de modules, les principaux algorithmes et données manipulées, les démarches adoptées pour tester le programme, les difficultés rencontrées et les solutions trouvées ainsi que l'organisation générale de travail.

INTRODUCTION

L'objectif de ce projet est l'obtention de programmes permettant de compresser et décompresser des fichiers selon le codage de huffman, ce dernier étant un codage statistique. L'objectif est donc d'obtenir de nouveaux codages d'octets à écrire dans le fichier compressé, codés sur un nombre réduit de bits selon leurs fréquence d'apparition dans le fichier d'origine. Ce codage permettra alors de passer d'un fichier compressé à un fichier décompressé et vice-versa.

PLAN DU DOCUMENT

I) *Architecture et présentations*

1) Architecture de l'application en modules

Le module Suite : Permet de manipuler les objets de type T_Suite. Il sera utile pour l'affichage de l'arbre.

Le module Table : Permet de manipuler les objets de type T_Table (pointeur) correspondant à une LCA. Il sera utile pour passer du fichier compressé au fichier décompressé et dans l'autre sens.

Le module Arbre : Permet de manipuler les objets de type T_Arbre (pointeur) correspondant soit à des nœuds, soit à des feuilles. Il permet également de construire l'arbre de Huffman à partir d'un fichier donné, ainsi que la table de huffman correspondante en utilisant le module table.

Le module Piles : Permet de manipuler des piles.

Le module Compression : Utilise les modules "arbre" et "table", il permet de passer d'un fichier à un fichier compressé. Il consiste en la construction de l'arbre à partir du fichier, puis à la construction de la table à partir de l'arbre construit précédemment. Ensuite on écrit dans un nouveau fichier (le fichier "compressé") les octets selon le parcours infixe de l'arbre puis le code de l'arbre (ces derniers seront utile lors de la décompression), on parcourt ensuite le fichier d'origine et convertit les octets en leur version réduite selon la table construite. On obtient alors le fichier compressé.

Le module Decompression : Utilise les modules "arbre" et "table", il permet de passer d'un fichier compressé à son fichier d'origine. Il consiste en un traitement du fichier compressé: on lit le fichier afin d'obtenir 3 objets distincts : un tableau contenant les caractères présent dans l'arbre dans l'ordre du parcours infixe, le code de l'arbre, ainsi que les octets compressés (qu'on manipule en Unbounded_String) du fichier. À partir de ces derniers, on reconstruit alors la table de Huffman et on réécrit le fichier d'origine en transformant les octets compressés en octet.

2) Présentation des principaux types de données

Le type T_Octet : (mod 2**8) stocke des octets

Le type T_Table : pointeur sur un enregistrement contenant une Clé de type T_Octet, une Donnee de type Unbounded_String correspondant au codage sur un nombre de bits réduit de la Clé (exemple *Clé = 01001100, Donnee = "010"*), ainsi qu'un suivant de type T_Table.

Le type T_Arbre : pointeur sur un cellule contenant une clé de type T_Octet, une donnée de type integer correspondant à une fréquence, ainsi qu'un fils droit et un fils gauche de type T_Arbre. On utilise ce type à la fois pour les noeuds et pour les feuilles : dans le cas des feuilles on aura les fils droit et gauche = null et la donnée égale à la fréquence d'apparition de la Clé dans le fichier, dans le cas des noeuds, aucune Clé ne sera affectée et la donnée sera égale à la somme des données des fils droit et gauche.

Le type T_Chaine : pointeur sur une cellule contenant un arbre de type T_Arbre et un suivant de type T_Chaine. On l'utilise dans la construction de l'arbre : on parcourt le fichier et on enregistre dans la chaîne des feuilles de type T_Arbre contenant les octets présents dans un fichier ainsi que leur fréquence d'apparition. On appliquera un algorithme à cette chaîne afin d'en obtenir l'arbre de Huffman correspondant au fichier d'origine.

3) Présentation des principaux algorithmes

La procédure Fréquence : prend un fichier en argument et crée un objet de type T_Chaine décrit ci-dessus correspondant au fichier.

La procédure Deux_Min : prend la chaîne construite précedemment par la procédure Fréquence et renvoie les deux arbres dont les fréquences (Donnees) sont minimum, supprime les arbres correspondant de la chaîne.

La procédure Construction_Arbre : prend la chaîne construite par la procédure Fréquence et construit l'arbre de Huffman correspondant au fichier traité. Pour cela on utilise la procédure Deux_Min et on modifie la chaîne : on insère en début de chaîne un nœud avec pour fils gauche le premier minimum et fils droit le deuxieme minimum. On répète ces instructions jusqu'à ce que la table ne contienne plus qu'un arbre : il s'agit de l'arbre de Huffman.

La procédure Construction_Table : construit la table de Huffman correspondant à l'arbre de Huffman pris en argument.

La procédure Mise_a_jour : modifie la table de Huffman pris en argument : on place le caractère de fin de fichier au début et on obtient sa valeur (=sa position de base dans la table) ainsi que son codage.

La procédure Compresse : utilise les procédures précédentes et construit un fichier compressé à partir d'un fichier.

La procédure Traitement : prend en paramètre un fichier compressé et renvoie un tableau d'octets correspondant aux octets du début du fichier compressé, puis une chaine de caractère correspondant au reste du fichier compressé, c'est-à-dire le code de l'arbre et les octets compressés, étant de base sous la forme d'octets. Lorsqu'un bit est égal à 1 on ajoute "1" à la chaîne de caractère, et "0" pour 0.

La procédure Reconstruction_Arbre : reconstruit l'arbre de Huffman à partir de la chaîne de caractère obtenue avec la procédure Traitement, renvoie également la position dans la chaîne de caractère du début des octets compressés, c'est-à-dire la position après le code de l'arbre.

La procédure Decompresse : utilise les procédures précédentes et reconstruit le fichier d'origine à partir d'un fichier compressé.

II) *Démarche et déroulement*

1) Démarche adoptée pour tester le programme

Afin de tester le programme, nous avons écrit des programmes de test pour chacun des programmes principaux cités ci-dessus. Nous avons notamment utilisé des procédures Affiche_Table et Affiche_Arbre à plusieurs reprises afin de vérifier si les constructions et modifications apportées y sont correctes.

2) Présentation des principaux choix réalisés, des difficultés rencontrées et les solutions adoptées

La difficulté principale rencontrée lors du développement de notre programme est la manipulation des octets, notamment lors de la manipulation du fichier compressé lorsque les "octets compressés" sont de taille variable (exemple : 010). Nous avons alors choisi de manipuler des données de type **Unbounded_String** afin de pouvoir les concaténer et d'en construire de taille 8 afin de les transformer en octets (exemple : *010 111 01 → "010" "111" "01" → "01011101"*) ensuite afin de pouvoir les enregistrer, on applique un algorithme sur cette chaine de caractère afin de la transformer en **T_Octet** (exemple : *"01011101" → 01011101*), on peut alors ensuite enregistrer les octets dans le fichier compressé. De la même manière lors de la lecture du fichier compressé, on lit les octets codés bit par bit afin de trouver les correspondances dans la table de Huffman (exemple : *si dans la table de Huffman on a les octets compressés 0, 10 et 11 et que dans le fichier codé on a le code 01011010, on applique la méthode décrite : "" pas dans la table, on continue →"0" dans la table, on écrit alors dans le fichier décompressé → "" → "1" pas dans la table, on continue → "10" dans la table, on écrit alors dans le fichier décompressé → "" , etc…*).
De plus, nous avons eu du mal à gérer les fuites de mémoire, nous avons donc essayé de les diminuer au maximum cependant certaines restent non résolues.
Pour *test_compression* :

```
==136770==
==136770== HEAP SUMMARY:
==136770==     in use at exit: 768 bytes in 44 blocks
==136770==   total heap usage: 573 allocs, 529 frees, 111,728 bytes allocated
```

Pour *test_decompression* :

```
==137278==
==137278== HEAP SUMMARY:
==137278==     in use at exit: 3,544 bytes in 117 blocks
==137278==   total heap usage: 5,313 allocs, 5,196 frees, 634,156 bytes allocated
==137278==
```

3) L'organisation de l'équipe

Nous avions au départ pensé à se répartir les tâches dès le départ : l'un travaille sur la compression, l'autre sur la décompression. Cependant nous nous sommes rendu compte que cette méthode n'était pas envisageable au vu de la difficulté du problème et de la quantité de connaissances qu'il nous manquait dès le départ. Travailler sur la décompression nous était alors impossible sans la compréhension approfondie du travail de compression. Nous avons alors décidé de travailler simultanément avec le même objectif en

répartissant les tâches au fur et à mesure sur des besoins beaucoup plus rudimentaires. Ce travail nous a permi de suivre l'avancée du projet dans les détails et en suivant la progression de chacun, ainsi lorsque l'un travaillait sur une procédure, il pouvait sans problème s'appuyer sur le travail préalable de l'autre.

III) *Bilan*

1) Bilan technique

Notre approche du problème ainsi que les solutions que nous avons apportées nous ont permis d'obtenir des programmes capables de compresser et décompresser de simples fichiers (ex : *.txt*). Une amélioration éventuelle permettrait d'appliquer ces programmes à d'autres types de fichier (ex : *.jpeg* ou *.mp3*).

2) Bilan personnel et individuel

Emile : Je trouve que le sujet est bien choisi, il permet de bien comprendre le principe de compression. Le fait que le problème posé soit ouvert m'a dans un premier temps frustré car beaucoup de concepts m'étaient flous et donc je ne savais pas dans quelle direction aller, cela m'a donc poussé à me documenter, à essayer de comprendre les enjeux et les concepts liés au problème, et j'ai l'impression que ce travail de recherche m'a d'autant plus permi de comprendre ce que je faisais, et de savoir comment travailler avec une quantité moindre d'informations. Le fait que ce projet se fasse en binôme m'a aussi aidé à comprendre comment travailler à plusieurs et de manière efficace sur un projet.

Anas : Le départ ne fut pas facile, bien cerner le sujet m'à pris beaucoup de temps : savoir on manipule quel type de donnée… Ensuite, une fois tout ça bien appréhendé, le code ne m'a pas posé trop de problèmes. Peut-être revoir le cours sur la généricité serai une bonne idée. Aussi le fait que le projet se fasse à plusieurs m'à encore plus motivé à avancer et aussi à avoir un autre point de vue sur la conception d'un algorithme.