Compte-Rendu - Projet PIM

Codage de Huffman

François Lauriol, Yael Gras

15 janvier 2022

Table des matières

1 Résumé

Ce rapport porte sur un projet étudiant de compression et décompression utilisant le principe de Huffman. Vous trouverez après une courte introduction l'architecture des programmes et les schémas fonctionnels des algorithmes. Ensuite nous vous présenterons nos choix principaux. Pour continuer, nous avons choisi 5 sous-programmes et 3 types que nous considérons essentiels à notre solution qui seront explicités. De surcroît, vous trouverez les démarches adopté pour tester nos programmes et les difficulté rencontrée. Par la suite, vous aurez un aperçu de l'organisation au sein de notre équipe. Pour conclure, ce rapport vous trouverez des bilans technique et personnels du travail réalisé.

2 Introduction

Dans le cadre du module d'enseignement de PIM (Programmation impérative), nous avons dû réaliser un projet complet sur le principe du codage de Huffman. En utilisant ce principe, nous devions réaliser un premier programme capable de compresser un fichier texte puis un second programme qui décompresse le fichier obtenu. Pour ce faire nous devions comprendre les différents aspect du code de Huffman et fournir un architecture des programme en raffinant les problèmes posé par le sujet.

3 Architecture du programme

Le programme est composé d'un module comprenant tous les sous-programmes nécessaires à la réalisation de la compression et la décompression comme montré sur les figures 1 et 2. Cela permet de pouvoir rendre le code lisible et que chaque personne utilisant le module puisse voir l'évolution de la compression ou de la décompression. Ainsi les deux programmes principaux communiquent avec ce module pour arriver à créer le fichier final. Pour lancer les programmes principaux, une interface a été conçu de manière à afficher ou non l'arbre et la table de Huffman et de faire le processus pour chaque fichier demandé par l'utilisateur via une ligne de commande. Les figure 3 et 4 permettent une vue simplifié des différents programme et omettent volontairement les sous-programmes contenu dans le module qui ne sont pas visible dans l'interface de ce dernier.

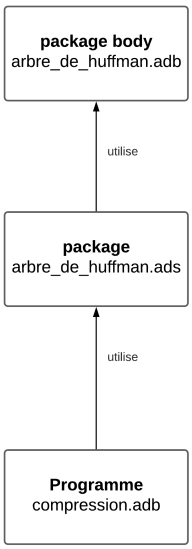

FIGURE 1 – Architecture du programme de compression

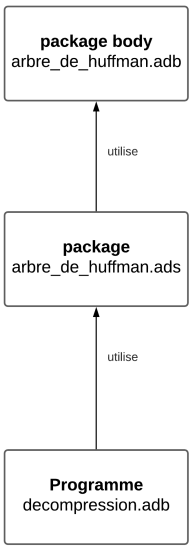

FIGURE 2 – Architecture du programme de décompression

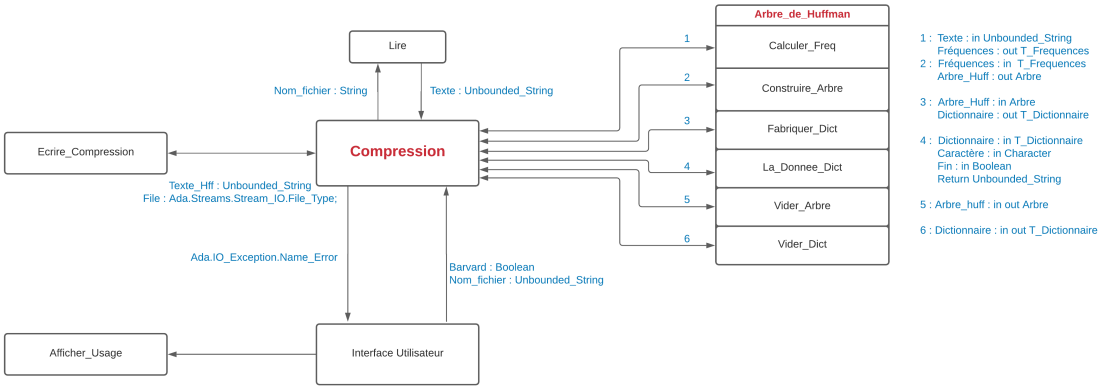

FIGURE 3 – Schéma fonctionnel du programme de compression

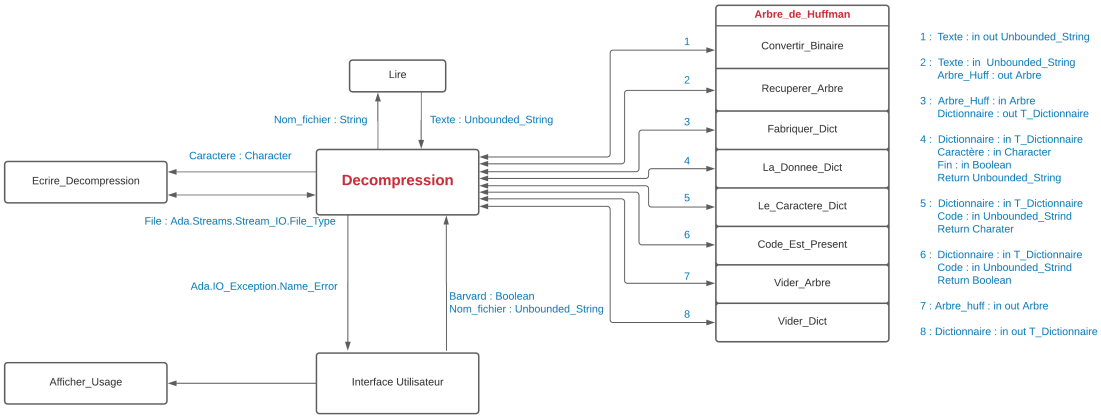

FIGURE 4 – Schéma fonctionnel du programme de décompression

4 Principaux choix réalisés

Un de nos principaux choix a été de décider de décrire quasi-totalement nos deux programmes principaux en sous-programmes issues d'un unique module et de ne mettre dans l'interface que les sous-programmes que nous utilisons pour les programmes principaux et non ceux utilisés par ces sous-programmes.

Nous avons aussi choisi de donner un statut particulier à '\$', il a été considéré comme étant le 257ème caractère du code ASCII et de fait a dans tous les programmes eu une manière différente de le traiter par rapport aux autres caractères.

5 Principaux algorithmes et types

Si on devait choisir 5 algorithmes parmi tous ceux que l'on a réaliser, nous choisirions ces 5 là : Construire_Arbre, Fabriquer_Dict, Recuperer_Arbre, Afficher_Arbre et Afficher_Dict.

Pour ces 5 algorithmes, 3 types ont été très important : Arbre, T_Dictionnaire et L_Arbres. Les deux premiers consistent à représenter respectivement l'arbre de Huffman et la table de Huffman. Le dernier à été très utile pour former l'arbre de Huffman.

Construire_Arbre consiste à construire grâce à une table de fréquences de caractères, l'arbre de Huffman. Pour cela, on a créé une "forêt" à l'aide du type L_Arbres et de la table de fréquences, puis on a assemblé deux à deux les arbres en commençant par ceux qui ont les plus petites fréquences. Après avoir assembler les deux arbres nous obtenons un nouvel arbre avec une fréquence égale à la somme des deux sous arbres. Nous réitérons l'opération jusqu'à n'avoir qu'un seul arbre final. Cet arbre est notre Arbre de Huffman. Nous n'utilisons cette procedure uniquement pour la compression.

Fabriquer_Dict consiste à fabriquer grâce à l'arbre de Huffman, sa table de Huffman. Pour cela, il fait un parcours infixe de l'arbre de Huffman et ajoute au fur et à mesure les feuilles à la table de Huffman. Le dictionnaire ainsi créé est une table de hachage. Nous avons choisi une table de hachage de longueur 27 car cela nous permet de répartir équitablement toutes les lettres de l'alphabet dans les différentes files (sous-dictionnaire). La case supplémentaire est quant à elle réservée au caractère '\$'. Cette méthode d'enregistrement de la table de Huffman est inspirée du module TH fait durant le TP10 et permets d'avoir une efficacité lors de la compression.

Recuperer_Arbre consiste à extraire l'arbre de Huffman d'un texte préalablement compressé. Pour cela, il effectue premièrement une liste des caractères présents dans le fichier initial dans l'ordre du parcours infixe de l'arbre de Huffman utilisé lors de la compression. Puis, il reconstruit l'arbre en plaçant sur chaque feuille trouvé son caractère correspondant et à l'aide du type L_Arbres assemble les "branches" de l'arbre au fur et à mesure que les feuilles les complètent.

Afficher_Arbre affiche l'arbre de Huffman comme la figure 20 du polycopié du projet. Pour cela, on a décidé de l'effectuer de manière récursive. On affiche les espaces se trouvant à gauche

des racines ou des feuilles, puis les caractéristiques des feuilles ou racines et on affiche les deux branches à la suite si c'est une racine à laquelle on a affaire.

Afficher_Dict affiche la table de Huffman comme la figure 21 du polycopié du projet. Pour cela, on a décidé de le faire de manière itérative, défensive et finie. On utilise alors une boucle *Pour* que l'on combine avec une gestion des exception, afin d'afficher un à un les caractères avec leur code de Huffman correspondant et d'ignorer ceux ne faisant pas partie du fichier texte .txt initial.

6 Démarche adopté pour tester le programme

On a voulu tester 3 cas de fichiers : le cas du fichier vide, fichier avec 1 caractère écrit, fichier avec plusieurs caractères écrits ('\n' non compris). 3 cas dont nous connaissons d'avance ce qu'il se doivent avoir comme arbre et table de Huffman.

Ces cas ont été étudiés avec et sans l'option bavard et nous avons comparé leurs arbre et table de Huffman après la compression et la décompression. Et nous avons aussi comparé les fichiers initiaux et ceux décompressés après avoir été compressés.

Ils se trouvent que tous ces tests se sont trouvés concluants avec des rapports de compression qui sont de l'ordre de ceux attendu.

7 Difficultés rencontrées et solutions

Le fait de traiter le cas du '\$' a été une d'une grande difficulté, en grande partie à cause du fait d'être le 257ème caractères. Ce qui nous a forcé à lui faire un traitement de faveur particulier (une file entière du dictionnaire lui a été consacrée).

Un autre grand problème a été de rassembler les arbres entre eux. Il nous a fallu définir des types pour faciliter les rassemblements (L_Complets, L_Arbres, L_Caracteres).

8 Organisation de l'équipe

L'organisation de l'équipe s'est faite selon les disponibilité de chacun. Au début du projet, François, ayant plus de temps et comme il fallait être rapide pour les deadline, a pris en charge la majeure partie des raffinages. Ensuite nous avons partagé équitablement la suite du projet et du rapport.

9 Bilan technique

Le projet est en soit terminé, il respecte le cahier des charges et les programmes se conforment bien à nos attentes. Nous pouvons compresser et décompresser à volonté des fichiers texte (format .txt) et compressés (format .hff).

Une piste d'évolution de notre programme aurait été d'essayer de le rendre plus générique de manière à compresser d'autre fichier que le fichier texte. Une autre, que nous aurions pu faire, aurait été de compléter le module de manière à avoir plus d'option de manipulation des arbres de recherche dans le cas particulier d'un arbre de Huffman.

10 Bilan personnel

10.1 François

Le sujet m'a beaucoup intéressé, notamment pour l'importance que le projet porte sur l'organisation du programme à laquelle nous avons particulièrement pris soin. Au total, j'ai du passé 64 heures sur ce projet, 35 h pour la conception globale, 20 h pour l'implantation, 1h sur la mise en point, 9h sur le rapport.

Au cours de ce projet, j'ai pu apprendre, une quantité de "petites choses" concernant les types (comme par exemple les String et Unbounded_String) et les opérations qu'ils me permettent. Ces "petites choses" m'ont permis de mieux comprendre comment le langage ada fonctionnait (l'opération & par exemple). Cette expérience m'a aussi permis de voir l'intérêt d'avoir un ads et un adb dans nos modules, me montrant à quelle point il n'était pas nécessaire de montrer toutes les petites opérations que réalisent mes programmes dans l'adb mais seulement les grandes opérations vraiment utiles pour l'utilisateur qui sont affichées dans l'ads.

10.2 Yael

Le sujet proposé ne m'a pas particulièrement intéressé. Au total, j'ai du passé environ 41 heures sur ce projet, 10h pour la conception global et les raffinages, 20h pour l'implantation, 1h sur la mise point et 10h sur le rapport.

Grâce à ce projet j'ai pu me rendre compte de l'importance de bien organiser l'architecture du programme en amont de l'implantation. Cela nous a permis de prendre de l'avance sur le projet dès le début. De plus, nous avons réussi à mettre en avant les atouts de chaque membre du binôme et ainsi avancer rapidement.