Rapport de Projet : Compression et Décompression avec le Codage de Huffman

Rapport

Mohamed Abe Mohamed
Brahim Jedou Cheikh

Encadré par : Zouheir Hamrouni

20 janvier 2025
ENSEEIHT – 1SN

Table des matières

1 Introduction

Dans le cadre de notre première année du cycle ingénieur en Sciences du Numérique à l'ENSEEIHT, ce projet vise à développer deux programmes en langage Ada pour la compression et la décompression de fichiers en utilisant le codage de Huffman. Cet algorithme sans perte attribue des codes binaires de longueur variable aux caractères en fonction de leur fréquence.

Le projet se concentre sur :
- La compression de fichiers pour réduire leur taille.
- La décompression pour reconstruire les fichiers originaux.

Les programmes doivent être exécutables via la ligne de commande avec des options de mode bavard ou silencieux.

1.1 Objectifs du rapport

Ce rapport détaille les méthodes, algorithmes, raffinages et architectures adoptées, ainsi que les difficultés rencontrées et les solutions apportées.

2 Raffinage Compresser

R0 : Compresser un fichier

R1 : Comment Compresser un fichier ?

- Lire le fichier et le stocker dans un tableau
- Créer le tableau d'occurrence
- Construire l'arbre de Huffman du fichier
- Créer la table de Huffman
- Encoder le texte
- Créer le fichier compressé avec l'extension .hff

R2 : Comment Lire le fichier ?

- Ouvrir Le ficher
- Stocker Les caracters de Fichier dans un Tableau (Fichie : out Tableau des octets)

R2 : Comment Créer le tableau d'occurrence ?

- Créer Module Dic – Pour stocker les
- caractères et leur nombre d'occurrence
- Initialiser Dic (Tab occurence) , (Tab occurence : out T Dic)
- Enregistre les caractères et leur occurrence (Tab occurence : in out T Dic)

R2 : Comment Construire l'arbre de Huffman de Fichier ?

— Créer Le module Arbre – Pour
— modeliser l'arbre de Huffman
— Créer Le type Tab Arb – Pour Stocker les arbres premieres
— Créer Un tableau des Racine d'Arbre (Racine : out Tab Arb)
— Créer le Procedure min des Racines (Racine : in) (min : out entier)
— Fusionner itérativement les deux minimale de Racine (Racine : in out)
— Retourner l'Arbre de Huffman (Arb Huff : out T Arbre)

R2 : Comment Créer la table de Huffman ?

— Créer le module Tableau Code pour stocker les codes de Huffman et les caractères
— Remplir les codes dans Tab Huff (Tab Huff : out T Code, Arb_Huff : in T_Arbre)

R2 : Comment Activer l'affichage selon l'option choisie (b ou s) ?

— Si option = b, afficher(Arb_Huff : in T_Arbre)
— Sinon, ne rien afficher

R2 : Comment Encoder le texte ?

— Déclarer le fichier_Comp de type tableau des bits (Fichier_Comp : Tableau)
— Remplir fichier_Comp (Fichier_Comp : in out Tableau)

3 Raffinage Decompresser

R0 : Décompresser les fichiers dont l'extension .hff

R1 : Comment « Décompresser le fichier » ?

- Lire le fichier compressé et le stocker dans un tableau (Fichier : out T-Fichier)
- Déterminer les octets au début du fichier qui représentent les caractères
- Déterminer la signature de l'arbre
- Créer l'arbre de Huffman à partir d'une liste de caractères et d'une signature d'arbre
- Commencer à décoder le fichier à partir du bit qui suit la dernière bit de la signature

R2 : Comment déterminer la liste des octets (caractères) ?

- Liste : list_caractere
- Charac_preced : T_Octet := Fichier(2)
- Charac_Actu : T_Octet := Fichier(3)
- Count : Integer := 1
- Tant que Charac_preced $\neq$ Charac_Actu Faire
  - Liste.taille = Liste.taille + 1
  - Count = Count + 1
  - Liste.car(Liste.taille) = Charac_preced
  - Charac_Actu $\leftarrow$ Fichier(Count)
  - Charac_preced $\leftarrow$ Charac_Actu
- Fin Tant Que

R2 : Comment déterminer la signature de l'arbre ?

- Signature : out T_Signature
- Count : Integer := Liste.taille + 1
- Nombre_de_un : Integer := 0
- Tant que Nombre_de_un < Liste.taille Faire
  - Transformer l'octet en un tableau de 8 bits (Bits : out T_Array)
  - Bits $\leftarrow$ transformer(Fichier(Count))
  - Pour i de 1 à 8 Faire
    - Si Nombre_de_un $\neq$ Liste.taille Alors
      - Signature.taille $\leftarrow$ Signature.taille + 1
      - Signature.tab(Signature.taille) = Octet(Bits(i))
      - Nombre_de_un $\leftarrow$ Nombre_de_un + Bits(i)
  - Fin Pour
  - Count $\leftarrow$ Count + 1
- Fin Tant Que

R2 : Comment créer l'arbre de Huffman à partir d'une liste de caractères et d'une signature d'arbre ?

- $i\_car \leftarrow 1$
- $i\_bit \leftarrow 0$
- **Procédure** cree_sgn_arb(i_car, Carcs, sign, i_bit, Arb)

- Si $i\_bit = 0$ Alors
  - Arb ← Nouveau Noeud
  - Arb.Occurence ← 0
  - $i\_bit \leftarrow i\_bit + 1$
  - Appeler cree_sgn_arb(i_car, Carcs, sign, i_bit, Arb.Gauche)
  - $i\_bit \leftarrow i\_bit + 1$
  - Appeler cree_sgn_arb(i_car, Carcs, sign, i_bit, Arb.Droite)
- Sinon Si ($i\_bit = sign.Taille$) ou ($i\_car = Carcs.Taille$) Alors
  - Arb ← Nouveau Noeud
  - Arb.Occurence ← 0
  - Arb.Caractere ← Carcs[i_car]
- Sinon
  - Répéter selon la structure décrite

R2 : Comment commencer à décoder le fichier à partir du bit qui suit la dernière bit de la signature ?

- j ← Liste_caractere.taille + Signature.Taille
- Répéter
  - Lire le bit du fichier ($bit \leftarrow Fichier(j)$)
  - Pour k allant de 1 à 8 Faire
    - Si Est_Feuille(curseur) Alors
      - Récupérer le caractère à la feuille
      - Si le caractère $\neq$ fin_fichier Alors
      - Écrire le caractère dans la sortie
      - Réinitialiser le curseur à la racine

4 Présentation des principaux choix réalisés et de l'architecture

1) Algorithmes et types utilisés

Dans le module Arbre :

a) Les Procédures :

Nom : `Initialiser`
Sémantique : Initialiser l'arbre avec `Null`.

Nom : `Detruire`
Sémantique : Détruire un arbre.

Nom : `Ajouter`
Sémantique : Ajouter une feuille avec un caractère et son nombre d'occurrence.

Nom : `Afficher_A`
Sémantique : Afficher l'arbre de Huffman.

Nom : `Signature`
Sémantique : Obtenir la signature de l'arbre de Huffman.

b) Les Fonctions :

Nom : `Fusionner`
Sémantique : Fusionner deux arbres.

Nom : `Caracter`
Sémantique : Fonction permettant de retourner le caractère d'une feuille.

Nom : `Est_Feuille`
Sémantique : Vérifier si un nœud correspond à une feuille.

Nom : `créer_Arb`
Sémantique : Obtenir l'arbre de Huffman à partir d'une liste d'octets avec la signature.

Dans le module Huffman :

c) Les Procédures :

Nom : `Creation_Arbre_Huffman`
Sémantique : Créer l'arbre de Huffman à partir d'un tableau d'octets avec sa taille.

Nom : `Creation_Tableau_Huffman`
Sémantique : Créer le tableau de Huffman à partir de l'arbre de Huffman.

Nom : `Tab_occurence`
Sémantique : Déterminer le tableau d'occurrence des caractères d'un fichier.

Nom : `Aff_Tab_Huffman`
Sémantique : Afficher le tableau de Huffman.

d) Les Fonctions :

Nom : `min`
Sémantique : Fonction pour obtenir l'indice de l'arbre avec le poids minimal dans un tableau d'arbres.

Nom : `place`
Sémantique : Fonction permettant de retourner l'indice d'un octet dans le tableau de Huffman.

Nom : `Transformer_Octet`
Sémantique : Fonction permettant de transformer un octet en un tableau de taille 8 constitué de 0 et 1.

Nom : `Lenght_Code`
Sémantique : Fonction permettant de retourner la longueur d'un code.

4.1 Architecture modulaire

Le programme est organisé en modules indépendants, chacun correspondant à un ou plusieurs fichiers spécifiques :
— **Module Arbre** : Gestion de la construction et de la manipulation de l'arbre de Huffman (`arbre.adb`, `arbre.ads`, `test_arbre.adb`).
— **Module Compression** : Encodage des données et génération des fichiers compressés (`compresser.adb`, `huffman.adb`, `huffman.ads`, `test_huffman.adb`).
— **Module Décompression** : Décodage des fichiers compressés (`decompresser.adb`, `lca.adb`, `lca.ads`, `test_lca.adb`).
— **Gestion des exceptions** : Définition des exceptions spécifiques au programme (`sda_exceptions.ads`).
— **Fichier d'exemple** : Fournit un fichier texte d'entrée pour tester la compression et la décompression (`exemple.txt`).

5 Principaux algorithmes et types de données

5.1 Algorithmes utilisés

— **Construction de l'arbre de Huffman** :
1. Calcul des occurrences des caractères.
2. Fusion de noeuds pour former un arbre binaire optimal.

— **Compression** : Transformation des caractères en codes binaires basés sur l'arbre de Huffman.
— **Décompression** : Parcours de l'arbre pour reconstruire les données originales.

5.2 Tests du programme

Des cas de tests ont été définis pour valider :
— La précision des données compressées.
— La reconstruction correcte des fichiers originaux.

6 Difficultés rencontrées et solutions adoptées

— **Problème de compréhension initiale** : Résolu par des séances supplémentaires de lecture et discussion avec l'encadrant.
— **Modélisation des types** : Plusieurs itérations ont permis de trouver des structures adaptées.
— **Analyse des lignes de commande** : Résolu après étude approfondie des modules Ada pertinents.
— **Affichage de l'arbre de Huffman** : Solutionnée avec des procédures auxiliaires pour visualiser la structure binaire.

7 Bilan individuel

7.1 Bilan de Brahim Jedou Cheikh

Au début du projet, j'ai rencontré des difficultés, notamment lors de la création de l'arbre et de l'encodage. Cependant, après avoir consacré du temps à bien assimiler l'idée en travaillant avec de petits exemples pour mieux comprendre la notion, j'ai pu surmonter ces obstacles. Ensuite, la notion de l'octet indiquant la fin de fichier m'a demandé plus de temps pour être comprise, mais grâce à plusieurs éclaircissements du professeur, j'ai pu la maîtriser. Concernant le programme de décompression, j'ai pris davantage de temps pour reconstruire l'arbre de Huffman à partir de la signature et d'une liste de caractères, en visualisant tous les chemins afin de décoder correctement le texte.

7.2 Bilan de Mohamed Abe Mohamed

J'ai commencé le travail sur la construction du module arbre, qui sert à la création de l'arbre de Huffman. L'une des procédures les plus complexes a été l'affichage de l'arbre en raison de la récursivité. Pour résoudre ce problème, j'ai développé une pile afin de faciliter la gestion de l'affichage. Une fois la procédure d'affichage de l'arbre créée, j'ai obtenu un outil très puissant pour tester les autres sous-programmes. Par la suite, lors du développement de la fonction de décompression, j'ai créé un sous-programme capable de reconstruire l'arbre à partir de la signature et des caractères. Ce travail m'a paru plus simple grâce à l'expérience acquise avec l'affichage de l'arbre. Enfin, j'ai rencontré des difficultés avec l'encodage, notamment lors de la lecture des octets. J'ai toutefois résolu ce problème en les transformant en un tableau de 8 colonnes, ce qui a considérablement simplifié la tâche.

8 Conclusion

Le projet de compression et de décompression de fichiers à l'aide de l'algorithme de Huffman a permis de mettre en œuvre des concepts fondamentaux de programmation impérative. En suivant une architecture modulaire bien définie, nous avons conçu des modules indépendants pour gérer l'arbre de Huffman, la compression, la décompression, ainsi que la gestion des exceptions et des tests.

Ce projet a offert une opportunité d'appliquer des techniques avancées telles que la manipulation d'arbres binaires, le traitement des fichiers binaires, et l'optimisation des algorithmes de compression. En outre, la gestion des tests et l'utilisation de raffinages progressifs ont assuré une conception robuste et évolutive.

Les principales difficultés rencontrées, comme la modélisation efficace des structures de données ou la gestion des fichiers compressés, ont été surmontées grâce à une approche méthodique et un travail d'équipe rigoureux. Ces défis nous ont permis de renforcer notre compréhension des structures algorithmiques et de leur implémentation pratique.

En conclusion, ce projet a été un excellent exercice pour consolider les compétences en programmation impérative et la conception modulaire, tout en illustrant les applications concrètes des algorithmes de compression dans le traitement des données. Les perspectives d'amélioration incluent l'optimisation des performances du programme et l'extension à d'autres formats de fichiers.