Algèbre Linéaire Numérique : Méthodes de réductions de modèles pour des problèmes PDE - Phase 1

Ava Dieng, Yvan Rameliarison, Victor Drouin Viallard

7 mai 2015

Table des matières

1 Introduction **5**
- 1.1 But du projet . . . 5
- 1.2 Cahier des charges . . . 5

2 Choix de conception **7**
- 2.1 Organisation et utilisation des packtages . . . 7
- 2.2 Les Arbres de Huffman . . . 7
  - 2.2.1 Le type A_Arbre_Huffman . . . 7
  - 2.2.2 Les fonctions parcourant un arbre de Huffman . . . 8
- 2.3 Les Mots Binaires . . . 8
  - 2.3.1 La Super Liste . . . 8
  - 2.3.2 Les Binaires . . . 8
- 2.4 Les chaines de caractères . . . 8

3 Quelques raffinages **9**
- 3.1 Ch_Decoder = Décodage de Huffman . . . 9
  - 3.1.1 Raffinage R_0 . . . 9
  - 3.1.2 Raffinage R_1 . . . 9
  - 3.1.3 Raffinage R_2 . . . 9
  - 3.1.4 Raffinage R_3 . . . 10
- 3.2 Ah_Rechercher_Feuille . . . 10
  - 3.2.1 Raffinage R_0 . . . 10
  - 3.2.2 Raffinage R_1 . . . 11
  - 3.2.3 Raffinage R_2 . . . 11
- 3.3 Maj_Nouvelle_Feuille . . . 11
  - 3.3.1 Raffinage R_0 . . . 11
  - 3.3.2 Raffinage R_1 . . . 11

4 Conclusion **13**
- 4.1 Quelques remarques et difficultés rencontrées . . . 13
  - 4.1.1 L'échange de noeuds . . . 13
  - 4.1.2 Creer_Nouvelle_Feuille et Maj_Nouvelle_Feuille . . . 13
  - 4.1.3 Limites du programme . . . 13
- 4.2 Bilan personnel du projet . . . 13

Chapitre 1

Introduction

1.1 But du projet

L'algorithme d'encodage de Huffman est très utilisé dans le domaine de la compression de donnée pour sa qualité d'algorithme de compression sans perte. Il est plus particulièrement utilisé dans la seconde phase de compression, lorsque les redondance propres au type de données encodées ont déjà été traitées (fréquences pour un fichier audio, couleur pour un fichier image, etc.), puisqu'il ne se base que sur la fréquence relative aux symboles qui lui sont fournis.

Le but de ce projet est ainsi de mettre en place l'algorithme d'encodage adaptatif de Huffman pour encoder et décoder des chaînes de caractères en ASCII ou en ASCII étendu.

1.2 Cahier des charges

Travail demandé

Afin de procéder à l'encodage et au décodage de textes selon l'algorithme de Huffman, il nous est demandé de :

- Définir le type arbre_huffman ainsi que les fonctions et procédures nécessaires à leur manipulation.
- Définir le type mot_binaire pour contenir la suite de bits d'une message encodé.
- Créer les fonctions d'encodage et de décodage de Huffman et les appeler depuis une procédure de test.

Contraintes

Dans un soucis de réusabilité, de simplicité (notamment pour le débuggage), et de lisibilité, je me suis personnellement contraint à utiliser au mieux des packages génériques et efficaces en temps. C'est par exemple le cas de la structure générique de super_liste que j'ai créé et utilisé pour mon projet et dont je détaille les spécificité plus bas.

De plus j'ai fais le choix de paramètrer lors de l'appel des fonctions de codage et de décodage la taille en nombre de bits des ASCII utilisés, pour permettre l'utilisation des ASCII et des ASCII étendus en ne modifiant qu'une unique constante.

Chapitre 2

Choix de conception

2.1 Organisation et utilisation des packtages

Dans le soucis de lisibilité et de réusabilité évoqué plus haut, j'ai pris soins de séparer aux mieux les différents types (et leurs méthodes associées) au sein de packtages distincts. J'utiliserai alors les types suivants :

- Arbre_Huffman
- Chaine pour le traitement des "String"
- Binaire ($\approx$ bit)
- Mot_Binaire : instance du packtage Super_Liste pour des éléments de type Binaire

chacun d'entre eux dépendant d'un packtage spécifique.

Pour le codage et de décodage j'ai aussi créé un autre packtage (la structure d'arbre de Huffman ne dépend pas intrinsèquement du codage et du décodage), bien que j'eu pu écrire ces fonctions dans le packtage Arbre_Huffman étant donné qu'elles semblent être l'unique cas d'application pour une telle structure.

Enfin j'ai créé conformément au cahier des charges un programme de test qui permet à l'utilisateur de tester le codage et le décodage de Huffman.

Remarque Pour plus de lisibilité mes types énumérés et enregistrements sont préfixés de "T_" tandis que mes types pointeurs sont préfixés de "A_". Les noms des packtages sont quant-à-eux préfixés de "P_".

2.2 Les Arbres de Huffman

Le packtage Arbre_Huffman contient la déclaration du type associé ainsi que les fonctions et procédures nécessaire à l'utilisation d'une telle structure.

2.2.1 Le type A_Arbre_Huffman

Un arbre de Huffman est un arbre binaire. J'ai donc choisis de définir le type T_Arbre_Huffman comme étant un enregistrement contenant :

- Un pointeur vers un fils gauche (un autre T_Arbre_Huffman)
- Un pointeur vers un fils droit (un autre T_Arbre_Huffman)

Il est constitué de noeuds, de feuilles, et d'un feuille vide qui doivent vérifier les conditions suivantes :

- Un noeud ne contient pas de valeur et porte un poids égal à la somme de celui de ses fils
- Une feuille contient une valeur unique et une poids (qui sera égal à son nombre d'occurences dans un texte lu)
- Une feuille vide ne contient pas de valeur et a un poids nul.

J'ai donc choisis d'ajouter à l'enregistrement constituant un T_Arbre_Huffman :

- Un entier représentant le poids de l'arbre (= Frequence)
- Un caractère représentant la valeur stockée dans le noeud (ou la feuille) courant (ASCII.NUL pour les noeuds et pour la feuille vide) (= Valeur).

Enfin il est utile pour un arbre de Huffman de connaître son père et de connaître son précédent et son suivant selon l'odre de Gallager, paramètres que j'ai donc rajouté à l'enregistrement.

Remarque L'utilisateur d'un arbre de Huffman aura accès à un pointeur vers un élément de type T_Arbre_Huffman qui sera alorechercher dans le fils gauche si on trouve la feuille etrs de type A_Arbre_Huffman.

2.2.2 Les fonctions parcourant un arbre de Huffman

Plusieurs fonctions utiles pour le traitement des arbres de Huffman et nécessitant un parcours (en profondeur ou suivant l'odre de Gallager) ont du être implémentée. Plutôt que de me perdre dans l'utilisation de pointeurs temporaires, j'ai opté pour l'utilisation de la récursion ; et bien qu'Ada traite indifférement les différentes récursivités , j'ai pris soins d'employer celle terminale.

2.3 Les Mots Binaires

Les mots binaires sont, dans mon cas, des listes de binaires. J'utilise pour cela une instance de mon packtage P_Super_Liste.

2.3.1 La Super Liste

L'utilisation d'un type dérivé de la liste parraît tout approprié pour le codage de Huffman puisque la taille des codes en sortie de l'algorithme de codage pourra amplement varier d'un texte à l'autre et qu'en revanche il ne sera pas nécessaire d'effectuer des accès rapide au milieux d'un mot binaire (ça aurait été le cas si on avait utilisé un tableau car alors on n'aurait pas retiré à chaque élément lu la tête du mot, mais dans notre cas conserver un mot binaire lorsqu'on le lit n'est pas nécessaire donc on peut se permettre de retirer la tête à chaque lecture d'un nouvel élément).

Cela permet en outre d'effectuer des concaténations en deux mots binaires d'une façon redoutablement efficace à condition de retinir continuellement un pointeur vers la queue (dernier élément !) de la liste. Mon type T_Super_Liste est donc un enregistrement contenant un pointeur vers une case de liste simplement chaîné et un autre pointeur vers la queue de cette même liste.

Le parcours des listes Là encore j'ai préféré utiliser des fonctions récursives plutôt que des boucles pour améliorer la lisibilité du code.

Remarque Il "manque" à mon packtage P_Super_Liste une fonction permettant d'accéder au k^{ieme} élément de la structure de stockage, fonction qui me paraissant sortir de "l'esprit" d'une telle structure n'a pas été implémentée.

2.3.2 Les Binaires

Le type binaire est simple (j'aurai aussi bien pu utiliser des booleens) et ses valeurs possibles (Zero et Un) sont représentées par un type énuméré. Il n'a besoin que de trois fonctions associées qui sont :

- Bi_Creer_Zero pour créer un binaire égal à 0
- Bi_Creer_Un pour créer un binaire égal à 1
- Bi_Est_Un pour vérifier si un binaire est égal à 1 ou non.

2.4 Les chaines de caractères

Le traitement des chaines de caractères à fait l'objet d'un packtage annexe pour simplifier l'utilisation de leur implémentation native (string), notamment en gérant automatiquement la taille de celles-ci.

Chapitre 3

Quelques raffinages

3.1 Ch_Decoder = Décodage de Huffman

3.1.1 Raffinage R_0

Spécification

```
.Decoder le mot binaire fourni en une chaine de caracteres ASCII
```

Paramètres

```
.Mb le mot binaire a decoder
.Taille_ASCII le nombre de bits sur lesquels rechercher dans le fils
        gauche si on trouve la feuille etuels les ASCII ont ete encodes
```

3.1.2 Raffinage R_1

```
-- Decoder le mot binaire fourni en une chaine de caracteres ASCII
.Initialiser les parametres
.Si le code n'est pas vide traiter le premier caractere qui est un ASCII
.Tant qu'il reste des caracteres dans le code e decoder, completer le message
        decode en les lisant
```

3.1.3 Raffinage R_2

```
-- Initialiser les parametres
.Faire une copie du code a decoder
.Initialisation de l'arbre
.Initialisation du noeud courant
-- Si le code n'est pas vide traiter le premier caractere qui est un ASCII
Si le code n'est pas vide alors
        .Determiner le caractere a partir de l'ascii lu
        .Retirer du message restant a decoder les elements lus
        .Ajouter le caractere dans le message decode
        .Ajouter une feuille dans l'arbre pour le nouveau caractere lu
        .Mettre a jour l'arbre
Fin Si;
-- Tant qu'il reste des caracteres dans le code e decoder...
Tant que le code n'est pas vide faire
        .Se deplacer dans l'arbre selon binaire en tete du code restant a
                decoder
        .Retirer le binaire lu du debut du code restant a decoder
        .Si on est arrive sur une feuille vide lire l'ascci et mettre a jour
                arbre et message decode
```

```
        .Sinon si on est sur une feuille recuperer le caractere lu et mettre a
                jour arbre et message decode
Fin Tant que;
-- Tous les binaires du mot a decoder ont ete lus.
```

3.1.4 Raffinage R_3

```
-- Initialiser les parametres
.Faire une copie du code a decoder
.Initialisation de l'arbre
.Initialisation du noeud courant
-- Si le code n'est pas vide traiter le premier caractere qui est un ASCII
Si le code n'est pas vide alors
        .Determiner le caractere a partir de l'ascii lu
        .Retirer du message restant a decoder les elements lus
        .Ajouter le caractere dans le message decode
        .Ajouter une feuille dans l'arbre pour le nouveau caractere lu
        .Mettre a jour l'arbre
Fin Si;
-- Tant qu'il reste des caracteres dans le code e decoder...
Tant que le code n'est pas vide faire
        .Se deplacer dans l'arbre selon binaire en tete du code restant a
                decoder
        .Retirer le binaire lu du debut du code restant a decoder
        -- Si on est arrive sur une feuille vide lire l'ascci...
        Si le noeud courant est la feuille vide alors
                .Determiner le caractere a partir de l'ascii lu
                .Retirer du message restant a decoder les elements lus
                .Ajouter le caractere dans le message decode
                .Ajouter une feuille dans l'arbre pour le nouveau caractere
                .Mettre a jour l'arbre
                .Remettre le noeud courant en haut de l'arbre
        -- Sinon si on est sur une feuille recuperer le caractere lu...
        Sinon si le noeud courant est une feuille alors
                .Recuperer le caractere a la racine de l'arbre
                .Ajouter le caractere dans le message decode
                .Mettre a jour l'arbre
                .Remettre le noeud courant en haut de l'arbre
        Fin Si;
Fin Tant que;
-- Tous les binaires du mot a decoder ont ete lus.
```

3.2 Ah_Rechercher_Feuille

3.2.1 Raffinage R_0

Spécification

```
.Rechercher la feuille de caractere celui fourni en parametre
```

Paramètres

```
.A_Ah un arbre de Huffman = un pointeur vers un noeud
.Valeur un caractere, celui dont il faut chercher la feuille
```

3.2.2 Raffinage R_1

```
-- Rechercher la feuille de caractere celui fourni en parametre
.Si l'arbre fourni est une feuille, la renvoyer si c'est la bonne et
        sinon retourner null
.Sinon renvoyer la feuille trouvee dans l'un des fils du noeud
```

3.2.3 Raffinage R_2

```
-- Si l'arbre fourni est une feuille...
Si l'arbre fourni est une feuille alors
        Si la valeur est celle cherchee alors
                .retourner l'arbre
        Sinon
                .retourner null
        Fin Si;
-- Sinon renvoyer la feuille trouvee dans l'un des fils noeud
Sinon
        .Chercher la feuille dans le fils gauche de l'arbre
        Si la feuille est trouvee alors
                .la retourner
        Sinon
                .Retourner la recherche de la feuille dans le fils droit
        Fin Si;
Fin Si;
```

3.3 Maj_Nouvelle_Feuille

3.3.1 Raffinage R_0

Spécification

```
.Remplacer l'actuelle feuille vide de l'arbre a par un noeud dont le fils
        gauche est une feuille comportant une valeur, et le fils droit est la
        feuille vide.
```

Paramètres

```
.A_Ah un arbre de Huffman = un pointeur vers un noeud
.Valeur un caractere, celui pour lequel il faut creer une feuille
```

3.3.2 Raffinage R_1

```
-- Remplacer l'actuelle feuille vide de l'arbre a par un noeud...
.Creer la nouvelle feuille
.Creer la nouvelle feuille vide
.Chercher l'actuelle feuille vide
.La nouvelle feuille devient son fils gauche
.La nouvelle feuille vide devient son fils droit
.Changer la valeur du Pere des deux nouvelles feuilles
.Mettre a jour les valeurs suivant et precedent des feuilles et du noeud
```

Chapitre 4

Conclusion

4.1 Quelques remarques et difficultés rencontrées

4.1.1 L'échange de noeuds

La gestion de l'échange des deux noeuds - dans la procédure de mise à jour de l'arbre de Huffman - n'a pas été évidente. Penser tout d'abord à ne pas copier les pointeurs mais les valeurs pointées, songer ensuite à bien échanger chacun des suivants et précédents pour préserver l'ordre de Gallager : j'ai donc fait le choix d'échanger les valeurs contenues dans les noeuds et d'échanger les fils.

La chose amusante est qu'il n'a pas été besoin de modifier les précédents et suivants des fils pour que le code fonctionne. L'ordre de Gallager semble alors préservé (le suivant d'un noeud reste toujours de poids inférieur) mais ce n'est pas toujours ou bien le fils droit de son père, ou bien le noeud le plus à gauche au niveau tout juste inférieur de l'arbre.

En essayant d'échanger les suivants et précédents des noeuds fils, mon programme ne fonctionnait plus que pour des textes assez courts (moins d'une centaine de caractère) sinon une exception de type "STORAGE_ERROR (Stack Overflow)" était levé. J'ai alors essayé les deux versions de l'algorithme pour me rendre compte que, pour ces textes assez courts au moins, les messsages codés étaient bien identiques.

4.1.2 Creer_Nouvelle_Feuille et Maj_Nouvelle_Feuille

La spécification donnée pour Creer_Nouvelle_Feuille m'a semblé ambigü et, sans penser à la suite, j'ai commencé par créer une nouvelle feuille de poids 1. Puis lorsque j'ai "bloqué" sur Maj_Nouvelle_Feuille (pensant que cette fonction devait elle aussi s'occuper de mettre à jour l'arbre et son ordre comme le fait Maj_Arbre) je me suis rendu compte que la nouvelle feuille devait en fait être créée de poids nul ce qui n'était vraiment pas évident.

4.1.3 Limites du programme

Comme je l'explique deux paragraphes plus haut, mon codage et mon décodage de Huffman semble fonctionner mais je n'ai pu vérifier si c'était exactement le code attendu qui était retourné.

En terme de temps mon programme de codage et de décodage semble particulièrement efficace et je pense qu'il m'a été judicieux en ce sens d'utiliser une structure de stockage spécifique, plus évoluée que le tableau ou la liste simple.

4.2 Bilan personnel du projet

Au cours de la petite trentaine d'heure que j'ai passé sur ce projet (les deux premiers tiers du temps à concevoir les types et algorithmes, le dernier tiers à les coder), j'ai su prendre conscience de diverses choses qui me semblent pouvoir m'être utiles à l'avenir.

Tout d'abord de l'utilité d'utiliser au mieux la répartition des fonctions et types sous la forme de packages, si possible génériques, pour facilement pouvoir tester chaque parties à part entière d'un programme (Tests unitaires) mais aussi pour pouvoir les réutiliser.

Ensuite de l'utilité d'user encore et encore des commentaires car, les vacances de noël ayant scindé mon temps de travail, j'ai du de nombreuses fois me replonger dans un code que j'avais écrit plusieurs jours auparavant.