Projet Huffman

Projet commun aux modules TI301-Fondamentaux de l'algorithmique 3 et TI 304-Introduction au génie logiciel

Rapport TI301-Fondamentaux de l'algorithmique 3

Membres de l'équipe :

- Gianlucca ANNICHIARICO
- Valère GOMEZ
- Lisa SANGLAR
- Quentin VINCENT

Lien du dépôt : https://github.com/NozyZy

Table des matières

Analyse

Contexte et module

Au sein du module Fondamentaux de l'algorithmique 3, aussi surnommé Structure De Données, nous avons dû programmer un logiciel afin de mettre en pratique les notions de programmations vues au cours de ce semestre. Ce module tient bien son surnom, car nous avons dû appliquer toutes nos connaissances des structures de données en langage C pour construire un logiciel de compression de fichier texte, se basant sur l'arbre de Huffman.

Ainsi, nous nous sommes répartis en équipe de 4 ou 5 étudiants. Afin de débuter ce projet sur une base solide et à un niveau égal, ces équipes de projet seront inchangées lors des TP d'Introduction au génie logiciel et de Fondamentaux de l'algorithmique 3, le sachant commun aux deux modules.

Le logiciel

Le projet correspond à un logiciel de compression de texte sans perte de données, se basant sur le principe de l'arbre de Huffman. Il s'agit de compresser le contenu d'un fichier texte converti en bit, afin de réécrire un code binaire encodé, qui prend moins de place que l'original. L'application devra, sur une interface console, demander à l'utilisateur de lui donner le chemin d'accès à un fichier d'origine, pour lui en créer un nouveau compressé, ainsi qu'un fichier dictionnaire permettant la décompression. Il faudra aussi présenter à l'utilisateur le taux de réussite de la compression (si des caractères n'ont pas réussi à être compressés), ainsi que le taux de compréhension (rapport du nombre de bits d'origine sur nombre de bits d'arrivé). Ce logiciel a donc besoin de pouvoir interagir avec des fichiers extérieurs, tout en utilisant des structures de données adaptées à son bon fonctionnement. La première étape était donc de comprendre le sujet dans sa globalité, mais surtout ces quelques étapes cruciales nécessitant les structures de données, et par là nous entendons surtout la création de l'arbre de Huffman.

Conception

Structures de données

Au sein du dossier du logiciel se trouve plusieurs sous-dossiers, servant à organiser les différentes fonctions et structure de données du logiciel. Penchons-nous sur le dossier DataStructures et son fichier .h associé :

/DataStructures

DataStructures.h

```
Structure type: Element
     ch: caractère
     occ: entier
     suivant: rèf à Element
FinType
Structure type: Liste: rèf à Element

Structure type: Noeud
     ch: caractères
     occ: entier
     bin: chaine de caractères
     sad: rèf à Element
     sag: rèf à Element
FinType
Structure type: Arbre: rèf à Nœud

Structure type: ElementNode
     data: Arbre
      suivant: rèf à ElementNode
FinType

Structure type: Queue
      last: rèf à ElementNode
```

```
FinType
```

Solutions algorithmiques

Nous avons donc dû faire une première version du logiciel en utilisant des listes, tableaux et un arbre, puis passer sur une partie optimisée en utilisant les dits AVL précédents, remplaçant listes et tableaux.

Voici quelques algorithmes pour comparer :

```
Algo: creerListeOccurrences(content: chaine de caractères, nbCaracte
res: long entier positif): Element
En-tête :
    Variables locales: l, lPremiere: rèf à Element
                       i: entier
                       ch: caractère
    Données: content: chaine de caractères
             nbCaracteres: long entier positif
    Sortie: Element
Corps :
Debut
    Si (nbCaractere = 0 OU taille(content) = 0) retourner ø
    Sinon
        lPremiere <- creerElement(content[0])
        l <- lPremiere
        i <- 1
        Tant que (i != nbCaractere):
            ch <- content[i]

            Si (ch != ø)
                Si (verifElement(lPremiere, ch) == 1)
                    ajoutListe(rèf à lPremiere, ch)
                Sinon
                    l->suivant <- creerElement(ch)
                    l <- l->suivant
            i <- i+1
            retourner lPremiere
        Fin Tant que
    }
```

```
Fin
```

Cet algorithme répertorie tous les caractères et leurs occurrences d'un fichier texte dans une liste, il est assez rapide et efficace, comparé à son amélioration en AVL :

```
Algo: createAVLcaractere(AVL: rèf à Arbre, content: chaine de caract
eres, taille: entier positif)
En-tête :
    Variables locales: tmp: rèf à Noeud
                       i: entier positif
    Données: AVL: rèf à Arbre
             content: chaine de caracteres
             taille: entier positif
Corps :
Debut
    i <- 0
    Pour i allant de 0 à taille-1:
        tmp <- creerNoeud(content[i], 1, ø)
        Si (tmp != ø)
            addNodeAVL(AVL, tmp)
Fin
```

Le code est beaucoup plus court, récursif, mais aussi beaucoup plus long, dû à la création d'un AVL. Cependant, l'avantage de cet algorithme est qu'il permet aux fonctions suivantes une recherche très rapide des différents caractères, puisque l'AVL trie les nœuds en fonctions de leur code ASCII.

Intéressons-nous désormais à la fonction qui crée un arbre de Huffman. C'est- une fonction qui prend en paramètre un AVL contenant tous les caractères du fichier texte, trié par occurrence croissante. Cette fonction nécessite l'utilisation de 2 files, une étant tous les caractères rangés dans l'ordre croissant d'occurrence (chose très facile à faire grâce à l'AVL en paramètre), la seconde file elle contient tous les nœuds de l'arbre, construit et ajoutés dans la file au fur et à mesure, permet de le construire des feuilles à la racine. Les fonctions propres aux files sont :

createQueue() : crée une file vide

sizeQueue(Arbre data) : retourne la taille de la file de nœuds

getMinQueues(Queue q1, Queue q2) : retourne et défile le nœud ayant la plus petite occurrence parmi les 2 files.

pushQueue(Queue q, Arbre a) : enfile le nœud donné dans la file

popQueue(Queue q) : retourne et défile la file

Finalement, voilà à quoi ressemble l'algorithme :

```
Algo: creerArbreHuffman(Arbre AVLocc): Arbre
En-tête :
    Variables locales: occQueue, nodeQueue: rèf à Queue
                       minD, minG, huffmanTree: rèf à Noeud
    Données: AVLocc: Arbre
    Sortie: Arbre
Corps :
Debut
    occQueue <- createQueue()
    creerOccQueue(AVLocc, occQueue)

    Si (sizeQueue(occQueue->last) <= 1)
        retourner popQueue(occQueue)

    nodeQueue = createQueue()
    Tant que ((sizeQueue(occQueue->last) + sizeQueue(nodeQueue-
>last)) > 1):
        minD <- getMinQueues(occQueue, nodeQueue)
        minG <- getMinQueues(occQueue, nodeQueue)
        if (minD->ch != ø) minD->sad <- minD->sag <- ø
        if (minG->ch != ø) minG->sad <- minG->sag <- ø

        huffmanTree <- creerNoeud(ø, minD->occ + minG->occ, ø)
        huffmanTree->sad <- minD
        huffmanTree->sag <- minG

        pushQueue(nodeQueue, huffmanTree)
    Fin Tant que

    retourner popQueue(nodeQueue)
Fin
```

Cette fonction est la version la plus optimisée, nous n'avons pas réussi à faire celle utilisant des listes et des tableaux, et avons préféré directement passer à une version optimisée. Maintenant, pour l'expliquer, rapidement, on commence par créer la file d'occurrence, si cette dernière n'a pas plus de 1 nœud, on la retourne : pas besoin de créer d'arbre. Sinon, on crée une queue de nœuds, et on commence à effectuer le schéma suivant : récupérer les 2 nœuds minimum (pouvant être inexistant) parmi les 2 files, si ces nœuds ont des caractères, ils sont des feuilles et n'ont pas d'enfants ; sinon, on crée un nœud parents des 2 nœuds minimum, que l'on enfile dans la file de nœuds. On effectue ce schéma tant que la somme de la taille des files

est supérieure à 1, puis on récupère le dernier nœud contenu dans la file de nœud, étant la racine de l'arbre.

C'est l'algorithme que nous avons eu le plus de mal à comprendre la logique avant de réussir à le coder, mais il est finalement fonctionnel, optimisé et prenant en compte toutes les possibilités.

Une fois l'arbre de Huffman créé, nous allons remplir les champs 'bin' de chaque nœud, c'est à dire le code binaire compressé de chaque nœud, grâce à la conception de Huffman. Pour cela, nous utilisons l'algorithme suivant :

```
Algo: createBinCode(huffmanTree: Arbre, binCode: chaine de caractere
s, index: entier)
En-tête :
    Variables locales:
    Données: huffmanTree: rèf à Noeud
             binCode: chaine de caracteres
             index: entier
    Sortie:
Corps : ’
Debut
    Si (huffmanTree != ø)
        Si (huffmanTree->ch != ø)
            huffmanTree->bin <- réserver(taille: index + 1)
            huffmanTree->bin <- binCode
        binCode[index] <- '0'
        createBinCode(huffmanTree->sag, binCode, index+1)
        binCode[index] <- '1'
        createBinCode(huffmanTree->sad, binCode, index+1)
Fin
```

Son principe est simple, le premier appel se fait avec la racine de l'arbre de Huffman, un binCode vide, et un index 0. Ensuite, tant que l'on est sur un nœud qui n'est pas une feuille (donc qui ne contient pas de caractère), on appelle récursivement cette fonction à gauche et à droite du nœud actuel, en ajoutant 0 ou 1 à la chaine de caractère binCode, en augmentant l'index de cette chaine de 1. Ainsi, quand on arrive sur une feuille, on réserve et copie le code binaire dans le nœud de l'arbre. La propagation récursive permet de parcourir tout l'arbre facilement.

Une fois l'arbre de Huffman créé et rempli avec tous ses codes binaires, il suffit de créer l'AVL du dico, que l'on trie selon le code ASCII, chose assez simple car nous l'avons déjà fait auparavant. Il faut juste rajouter la condition d'ajouter un nœud, si et seulement si ce nœud contient un caractère non null (est une feuille de Huffman)

La dernière étape pour compresser le fichier, une fois l'AVL du dico créé, il suffit de parcourir itérativement tous les caractères contenus dans le fichier texte à compresser, et de rechercher

son code binaire dans le dico, puis de l'écrire dans un nouveau fichier texte. Voici à quoi ressemble l'algorithme de recherche de code depuis le caractère :

```
Algo: codeFromChar(ch: caractere, dico: Arbre): chaine de caracteres
En-tête :
    Données: ch: caractere
             dico: Arbre
    Sortie: chaine de caractere
Corps :
Debut
    Si (dico != ∅)
        Si (ch < dico->ch) retourner codeFromChar(ch, dico->sag)
        Si (ch > dico->ch) retourner codeFromChar(ch, dico->sad)
        retourner dico->bin
    retourner ∅
Fin
```

Puisque le dico est un AVL trié par caractère, il est très simple d'en rechercher un dedans, et donc d'extraire le code binaire associé.

Désormais, nous avons compressé un fichier. Pour le chapitre fourni en annexe, voici les données résultantes :

```
all trees : 0.383000 sec
zip : 0.018000 sec

The file has been succesfully compressed !

all : 0.404000 sec
The file has been succesfully converted to binary !

Compression Ratio : 56.551277 %
```

La construction de tous les arbres prend environ 0.38s, la compressions 0.018s, en total 0.4s, pour une compression de 56.55% par rapport au fichier binaire d'origine. Le programme est donc pas mal optimisé, même si la création des AVL est très longue comparé au reste, et pourrait être optimisé.

Conclusion

Ainsi, ce projet fut extrêmement riche en apprentissage et plein de surprises. L'intérêt fut de mettre à profit ce que nous avions appris avec le module de programmation de S3, c'est-à-dire les structures de données.

La programmation en langage C du projet Huffman, n'a pas été de tout repos. Les premières fonctions à réaliser sur la conversion binaire et l'écriture dans des fichiers textes furent relativement simples, mais cela ne correspondait pas réellement aux points sur lesquels nous étions attendus. En effet, le cœur du projet introduit l'utilisation d'arbres AVL. Ces derniers sont utilisés pour le codage d'une lettre, il s'agit ici de la clé de notre compression. Les files et tableaux seront plus perçus tel des aide-mémoires bénéfiques à l'arbre. La création de l'arbre d'Huffman fut selon nous la partie la plus complexe du projet, notamment celle qui a demandé plus de temps. En effet, avant tout code, il fallait assimiler et intégrer le concept pas toujours évident de cet arbre, comprendre et visualiser la façon dont il était construit et comment le traverser pour obtenir le code voulu pour chaque caractère. Mais à force de détermination et de documentation sur ce dernier, nous en sommes venus à bout. Mis à part cet arbre, tout le reste a été un peu plus simple, quoique long vu le nombre de fonctions. Or l'utilisation des fonctions d'AVL et de files déjà travaillées en cours nous ont permis un gain de temps conséquent, afin de nous pencher sur l'utilisation de ces fonctions, de manière plus optimisée possible. Utiliser ces notions de cours dans un véritable logiciel nous a permis de nous rendre compte de leur intérêt et efficacité, mais aussi de leurs défauts.

Il est important de souligner que le logiciel Git mis en relation avec le service Github, sont des outils indispensables pour tout programmeur et équipe de développement. En effet, un projet de groupe peut parfois s'avérer fastidieux lorsqu'il s'agit de mettre en commun les travaux. Or grâce à son utilisation, nous avons pu partager facilement nos programmes, protéger nos codes et modifications, contrôler les différentes versions du projet, et avancer étape par étape en faisant bien attention de tout tester à chaque fois. Le travail collaboratif fut fluide sans réel problème.

Commentaires personnels :

Quentin Vincent :

Pour ma part c'est un projet qui a été très intéressant dans son ensemble, quoique loin d'être reposant.

J'ai programmé bon nombre de fonctions et ai aussi aidé mes collègues dans leur programmation, mais ce qui a demandé le plus d'effort a été de bien organiser les tâches de chacun, qu'elles soient faîtes à temps et efficacement. Nous avons pris du retard, certainement à cause d'une sous-estimation de la charge de travail que nécessitait ce projet. Je tiens à rajouter aussi que travailler entièrement à distance, avec des personnes que nous n'avons pas forcément eu l'occasion d'apprendre à connaître à cause des conditions de ce semestre, n'a vraiment pas facilité la fluidité de travail, et donc la progression du projet.

Je reste tout de même satisfait du rendu que nous faisons, et des choses que nous avons pu apprendre et approfondir tout au long de ces semaines.

Valère Gomez :

Pour ce projet, j'ai travaillé sur une grande partie de la programmation des fonctions qui touchent aux listes, tableaux et arbres ainsi qu'à leur amélioration.

Les enseignements du cours de Génie Logiciel sont nécessaires au bon déroulement des projets, de manière à être efficace. Les applications vues en cours de TD nous permettent d'exploiter chaque méthode l'une après l'autre. Ces exercices très bien construits viennent peut-être trop tardivement dans le semestre. Les divers projets de programmation devraient faire suite aux exercices et non se confondre. Cela permettrait d'avoir une vue d'ensemble des méthodes à exploiter et ainsi savoir les combiner pour une meilleure productivité.

Le travail d'équipe est toujours difficile à distance, le re-confinement est venu perturber la dynamique mis en place lors de la rentre en présentiel.

Gianlucca Annichiario :

Ce projet fut très enrichissant d'un point de vue gestion de mémoire, en effet on sait aujourd'hui à quel point la gestion de la mémoire notamment en C est importante. L'introduction des AVL à était une notion complexe mais importante. L'introduction du logiciel doxygène a été enrichissant car cela m'évite de commenter, grâce aux cours de génie logiciel j'ai améliorer ma façon de programmer et de gérer mes commentaires.

Lisa Sanglar :

J'ai trouvé le projet autant intéressant que complexe faisant appel à un bon nombre de connaissances.

Le travail d'équipe à distance ne fut pas des plus approprié, rendant la réalisation de ce projet moins agréable. Mais cela ne nous a pas empêché de communiquer et d'échanger nos idées sur la résolution de problèmes via certaines plateformes. De plus, j'ai rencontré de nombreuse difficulté de programmation ce qui n'a cependant pas atténué mon envie d'être utile et de m'impliquer pour ce projet. L'équipe reste bienveillante et toujours disposée à l'entraide.

Dans l'ensemble, le projet m'a permis de visualiser et de comprendre l'utilisation des concepts de programmation vu en S3.