TP ADA
"*Compression de Huffmann*"

Koenig Romain, Leymarie Valmon

Mercredi 21 Novembre 2012

1 Présentation du principe de compression

Dans un texte, certains caractères apparaissent plus que d'autres. Par exemple, dans la langue française, le caractère le plus courrant est la lettre 'e'. Le principe de compression d'un fichier texte repose sur ce constat et l'exploite. Comme nous le savons tous (depuis peu !) un caractère est codé sur 8 bits, soit 1 octet. L'idée ingénieuse pour compresser un fichier texte est d'affecter aux caractères les plus fréquents un code qui tient sur le minimum de bits possibles. Ainsi, on crée un fichier compressé écrit en binaire qui contient exactement les mêmes caractères que le texte initial, si ce n'est qu'ils sont écrits avec leurs codes binaires !
Pour coder nos caractères, nous construirons un arbre de huffmann, qui aura pour fonction de stocker le code ascii et le nombre d'occurences des caractères dans ses feuilles. Les caractères les plus fréquents seront haut dans l'arbre et le chemin pour aller à une feuille constituera le noveau code du caractère : à chaque fois qu'on se déplace dans le fils gauche de l'arbre, on ajoute le bit 1 au code du caractère, à chaque fois qu'on se déplace dans le fils droit de l'arbre, on ajoute le bit 0 au code du caractère et dès qu'on atteint une feuille, on retourne le code du caractère !
Une fois cet arbre généré, ils nous sera facile de coder et de décoder chaque caractère du texte. Cependant, encore faut-il en disposer. En effet, une fois le fichier compressé, comment avoir accés à cet arbre ? Ainsi, avant de compresser notre fichier texte, nous sauvegarderons en entête du fichier compressé la file qui aura servit à l'élaboration de notre arbre afin d'être capable de le reconstruire lors de la décompression de notre fichier.

2 Structures de données

Nous avons pris la liberté de coder nous même toutes les structures de données et n'avons que trés peu utilisé celles fournis. Dans cette partie nous allons expliciter chacune de nos structures de données. Au niveau du fonctionnement de notre programme principal, tp huffmann.adb, nous appelons deux sous fichiers : compresseur.adb et decompresseur.adb. Ces deux fichiers sont des packages, ils contiennent donc les procedures et fonctions permettant à notre programme de fonctionner, mais ils ne s'executent pas à proprement parler. On aura pris soin d'inclure dans chacun d'entre eux des packages "secondaires" dans lesquels on pourra trouver les structures et procedures de bases (tableaux, files, arbres, code binaire).

2.1 Tableau

Données Le type "tableau" que nous definissons ici est un tableau d'entier de 256 cases (0...255). Cette implémentation nous permettra de mettre à la case du code ascii correspondant au caractère

considéré son nombre d'apparition dans le fichier texte. Ce tableau sera necessaire pour créer la file de priorité permettant l'élaboration de l'arbre de Huffmann.

Procédures et fonctions -*est vide* est une fonction qui retourne "true" si toutes les cases du tableaux sont à 0.
-*indice max* est une fonction qui retourne l'indice de la case du tableau qui a la plus grande valeur.
-*afficher*est la procedure d'affichage de notre tableau, surtout utilisée pour nos tests.

Les fonctions et procédures de creation, liberation, test de nullité et d'affichage de nos structures étant sensiblement les mêmes, nous n'en parlerons plus.

2.2 Arbre

Données Le type "Arbre" sera un pointeur sur noeud, une structure présentant les champs "prio" (nombre d'occurences de notre caractère), "ascii" (code ascii du caractère) et fd et fg qui pointent respectivements vers le fils droit et le fils gauche de notre arbre.

Procédures et fonctions -*libere arbre* est la procédure, comme indiqué qui libere notre arbre, de ses feuilles à sa racine.

2.3 File chainée

Données Le type "file chainee" sera un pointeur sur cellule, une structure ayant un champ "abr" et un champ "suiv" pour nous déplacer au maillon suivant. Nous avons choisit de faire une file d'arbres dès le début, même pour notre file de priorité, afin d'éviter d'avoir un trop grand nombre de structures se ressemblant.

Procédures et fonctions -*enfiler* est une procédure qui parcourt la file et met l'élément voulu (ici un arbre) en queue.
-*defiler* est une procédure qui met dans l'arbre placé en paramétre, le premier maillon de la file tout en l'enlevant.
-*met a jour*est une procedure qui prend en paramétre un noeud et le place au bon endroit de la file en fonction de son champ "prio" (nombre d'occurences).

2.4 Code binaire

Données -*Bit* est le type qui définira un bit. Il pourra prendre 2 valeurs, "UN" (1) ou "ZERO" (0).
-*code binaire* est le type qui va nous servir à écire un code binaire. Nous avons choisit pour cela de l'implenter comme une file ayant une valeur de type "bit" et un champ "suiv".
-*Dico* (table de hachage) est en fait un tableau de type "tab" qui aura dans la case correspondant au code ascii du caractère le "code binaire" qui lui est associé. Cette structure de données est construite à partir de l'arbre de Huffmann.

Procédures et fonctions -*copie*Permet de copier cellule par cellule une file avec une autre.

3 Fonctionnement du programme principal

Notre programme principal a pour nom "tp huffmann.adb" et fait appel à deux procédures : compresseur.adb et decompresseur.adb.

3.1 Compresseur

Dans un premier temps, pour cette procédure, on va lire une première fois le texte afin de construire "T Ascii", un tableau d'analyse fréquentiel de chaque caractère. Grâce à celui-ci, on crée la file de priorité "F Prio" qu'on écrit ensuite en entête du nouveau fichier compressé. On prendra le soin d'utiliser un caractère d'arrêt inutilisé (ici l'integer 0). A l'aide de notre file de priorité, on crée l'arbre de huffmann, "huffmann" comme indiqué dans le sujet, et on construit en le parcourant notre dictionnaire de traduction "T Trad". On utilise enfin la procédure "Text Bin" qui va lire le texte, caractère par caractère, chercher le code binaire du caractère considéré dans le dictionnaire et l'écrire dans le fichier compressé.

3.2 Decompresseur

Pour decompresser le fichier considéré, on doit d'abord reconstruire l'arbre de Huffmmann. La prémiére étape de notre décompression consiste donc à lire l'entête du fichier compressé afin de reconstruire notre file de priorité. A partir de celle-ci, l'arbre de huffmann est re-généré à l'aide de la même fonction que celle utilisé pour la compression. Il ne nous reste alors plus qu'à parcourir le fichier compressé "bit" à "bit" et de se déplacer dans l'arbre jusqu'à tomber sur une feuille afin d'écrire le caractère lui correspondant dans le nouveau fichier décompressé.

4 Difficultés et optimisations

Difficultés -Difficulté de gérer les flux bit à bit alors qu'on manipule des octets : Nous avons dû effectuer des opérations "astucieuses" afin de lire le fichier compressé bit à bit.
Nous avons eu un soucis lors de notre première décompression : certains caractères étaient échangés (un 'l' était un 'v'..etc). Aprés quelques tests, nous avons constatés que lors de la re-creation de la file de priorité à partir du fichier binaire, les "prio" n'étaient pas rangées de la même façon que lors de la compression, ce qui nous donnait un arbre de huffmann différent.(voir presquebon.txt qui est la décompression de sand.comp).
Nous avons enfin rencontrés quelques fuites mémoires. Toujours lors de notre première execution sur le "fichier sand.txt", nous avions pour HEAP SUMMARY :
"in use at exit 13,368 bytes in 736 blocks total heap usage : 848 allocs, 112 free, 17952 bytes allocated."
Aprés quelques liberations et beaucoup de patience, nous sommes parvenus à libérer la totalité des allocations effectuées, pour la compression et la décompression !

Optimisations -La plus grosse optimisation que nous voyons pour ce code pourrait se faire au niveau de la construction de l'arbre de Huffmann. En effet, l'arbre n'est pas totalement équilibré (c'est le principe de son fonctionnement), de ce fait, certains caractères sont codés sur plus de 8 bits (le 'L' par exemple) et prennent donc plus d'espace mémoire qu'un caractère normal.

5 Conclusion

Sans aucune intention d'amadouer le correcteur, nous voulons souligner le plaisir que nous avons pris lors de ce projet et remercier l'ENSIMAG de nous l'avoir proposé. En effet, en plus de nous avoir permis de consolider nos connaissances en ada (écriture et lecture dans les fichiers), le sujet était ludique et interessant. Le niveau était trés bien adapté et nous avons pu nous familiariser avec les outils de debuggages (ddd) et de fuite mémoire (valgrind).