TD sur Huffman : micro rapport

Introduction

L'objectif de ce TD est de réaliser en C un programme qui soit capable de compresser un fichier en appliquant le codage de Huffman (programme appelé huf.c) puis un autre programme qui soit capable de le décompresser (programme appelé dehuf.C). Enfin, un programme Python (nommé par) devra appeler ces 2 programmes pour compresser / décompresser un répertoire et les fichiers qu'il contient.

Avant de présenter dans le détail notre travail, il est important de rappeler le principe du codage de Huffman : on va utiliser un code de longueur variable pour représenter un caractère du fichier que l'on veut compresser. Pour trouver ce code, on doit connaître la probabilité d'apparition de ce caractère dans tout le fichier. Plus ce caractère sera présent, plus court sera son code. C'est grâce à ce principe que le fichier contenant les codes sera plus petit que le fichier d'origine.
Dans notre TD, nous avons utilisé non pas la probabilité d'apparition mais le nombre d'occurrences, ce qui revient au même. Le codage de Huffman organise les caractères dans un arbre, chaque nœud de l'arbre étant soit un des caractères soit un nœud regroupant 2 caractères ou 2 autres nœuds. Il a fallu gérer plusieurs informations pour chaque caractère : son nombre d'occurrences, quel était son nœud père mais aussi, est-ce qu'il avait un fils (un autre nœud) à gauche ou à droite. Pour gérer toutes ces informations pour 1 caractère, nous avons utilisé un tableau dynamique dans une structure nommée arbre (ne sachant pas le nombre de caractères distincts que nous allions avoir dans le fichier en entrée) dans lequel chaque élément est une structure (appelée nœud) qui va contenir toutes les informations.

Ce tableau sera indicé par le code ASCII du caractère, ainsi la cellule 97 de notre tableau arbre contiendra les informations pour le caractère a.
En annexe, j'ai mis le fichier Excel qui m'a permis de m'y retrouver car j'ai été souvent un peu « perdue »…

Programme huf.c

Dans le main :

On appelle dans l'ordre les fonctions suivantes :

- arbreCreer qui va compter le nombre de caractères de notre fichier et le nombre de leur occurrences pour pouvoir remplir le tableau de la structure arbre, suivant le principe : 1 caractère est égal à une cellule de mon arbre, cette cellule est indicé par le code ASCII du caractère et cette cellule est constitué par un structure nommé « nœud ».

  Une fois que c'est fait, on va pouvoir remplir le tableau en commençant par le caractère du fichier ayant le plus petit code ASCII. On rentre le caractère, son nombre d'occurrence et on initialise 'gauche', 'droit' et 'parent' à '-1'. Ensuite on reboucle sur ce tableau pour rentrer les nœuds intermédiaires à la suite des caractères. Un nœud intermédiaire est constitué des nœuds ayant les plus petites occurrences, l'occurrence d'un nœud étant la somme des 2. On peut donc compléter les variables 'gauche', 'droit' et 'parent'. Il faut donc bien comprendre que lorsque l'on créé un nœud intermédiaire, on créé un des éléments du code de Hufman, donc plus les nœuds apparaissent peu de fois, plus leur code sera grand et qu'à l'inverse, les caractères qui apparaissent souvent seront traités en dernier et auront donc un « petit » code de Huffman, ce qui permet d'obtenir un fichier plus petit. (voir fichier exel)

- arbreAfficherCodes qui va afficher l'arbre.

- CreerCode qui est une fonction récursive. Elle va lire l'arbre créé précédemment en partant de la « racine » le nœud qui a le plus grand indice, c'est la taille de notre arbre.

  Tant que l'on tombe sur un nœud qui a un fils gauche, on descend dans l'arbre est on ajoute un '0' au code. Une fois que l'on ne peut plus aller à gauche, on fait la même chose mais à droite cette fois ci et on ajoute un '1' à chaque fois que l'on descend.

  Une fois que l'on tombe sur une feuille, on stocke ce code de '0' et de '1' dans une cellule du tableau code. On ne connait pas la taille du code de '0' et de '1', c'est pour ça que la cellule est une variable dynamique. La cellule a pour indice le code ASCII du caractère, cela nous permet d'associer le code binaire que l'on vient de trouver au

caractère de la feuille.

On rappelle la fonction jusqu'à ce que tout l'arbre soit parcourus et que l'on est un code pour chaque feuille.

On obtient ainsi un tableau indicé par le code ASCII du caractère qui contient le code de Huffman. Cela nous a semblé plus compréhensible, que de rajouter une nouvelle variable à notre structure « nœud ».

- ajoutCode, cette fonction écrit dans le fichier de sortie, à la fois le nombre de bits total et le nombre de codes. Pour cela, on fait appel à la fonction ReturnNbBit, puis on écrit :
    - le nombre de bits total dans le fichier (si le nombre de bits est supérieur à 255, on écrit autant d'octets de 8 bits de 1 (1111 1111) tous les 255 bits puis le reste.
    - le nombre de code distinct dans le fichier (qui lui est au maximum de 255)

  Pour chaque code (non nul) :
    - la valeur de son indice
    - la longueur de son codage
    - la valeur décimale de code binaire. Pour cela, on fait appel à la fonction convertirN qui va convertir la valeur des bits en décimal. Là aussi, si la valeur est supérieure à 255, on rentre cette valeur par paquet de 256 dans le fichier.

  Toutes ces valeurs constituent l'entête de notre fichier compressé et va servir lors de la décompression.

- ReturnNbBit, c'est une fonction qui va nous servir à connaitre : le nombre de codes distincts et le nombre de bits dans le tableau code. Si on tombe sur une feuille, on rajoute 1 au nombre de code et si non, on rappelle la fonction avec son fils gauche et son fils droit.Elle retourne un entier qui est la longueur de tous les codes multiplié par leur nombre d'occurrences (donc le nombre de bits total)

- convertirN, elle prend en entrée le code et sa longueur. Elle va convertir le code binaire qu'elle reçoit en entrée en décimal. Pour cela elle utilise les formules :

  r+=mul*(code[i]-48)

  mul*=2

  r est le résultat. Mul représente la puissance de 2.

  On enlève 48 au chiffre binaire du code car en ascii, la valeur '0' est codée '48' donc lors de la conversion du binaire 0 en ascii, le résultat donne 48, pour retrouver la même valeur mais en décimal, on est donc obligé d'enlever 48 au résultat final.

- ajoutHuff, dans cette fonction, on relit les caractères de notre fichier un par un. Pour chaque caractère lu, on met son code dans un buffer, ce buffer va être découpé en paquet de 8 bits pour former des « caractères », une fois les 8 bits remplit on peut les écrire dans notre fichier (c'est la fonction convertirAscii) on réitère l'opération avec la suite de nos codes et si jamais les derniers bits ne remplissent pas les 8 bits, on rajoute des 0 pour compléter le dernier caractère et pouvoir écrire ce dernier dans le fichier.

- convertirAscii, converti un octet en ASCII.

Programme decomp.c

Dans le main :

On ouvre le fichier compressé en lecture et le fichier final en écriture.

On récupère une partie l'entête du fichier compressé, le nombre de bit du fichier compressé et le nombre de code distinct.

On appel dans l'ordre les fonctions suivantes :

- rentreCode qui permet de reconstituer le tableau code qui permet d'associer un caractère en ascii a son code binaire.

  On récupère d'abord le code ASCII du caractère puis sa longueur binaire. On créer une zone allouée dynamiquement lors de la compression qui va contenir le code binaire. On récupère la valeur décimale du code ASCII puis on boucle pour reconstituer ce code dans le tableau mais en valeur binaire. (On éclate le code ASCII en '0' et '1')

  Exemple :

  En entrée on reçoit un caractère ASCII d'une valeur de 110, on boucle pour constituer le code binaire composé des 3 caractère '1' 1' '0'.

- decomp qui sert à réécrire le fichier final. Grace à la fonction trouveCode on a associé les caractères avec leur code, il ne reste plus qu'à changer chaque code en caractères.

- intToBin permet de changer 8bits (le caractère que je viens de lire) en 8 caractères de '1' et '0' que l'on stocke dans le tableau de char 'buffer'.

- trouveCode elle fait une boucle sur le tableau de char 'code' et on veut trouver la correspondance entre un des codes de ce tableau et le début du buffer. Une fois cette correspondance trouvée, on peut lui associer son caractère grâce à l'indice du tableau 'code' qui représente le code ASCII en décimal du caractère. Ensuite, on décale les bits du buffer de x places (x représente la longueur du code que l'on vient d'associer) et on réitère l'opération. Si on ne trouve aucune correspondance c'est que le buffer ne contient pas le code complet, on rajoute donc les bits suivants (cette opération se fait dans decomp)

Conclusion :

Lors de la conception de ce programme nous avons rencontré des problèmes pour comprendre le mécanisme de la création de l'arbre de Huffman, nous avons donc fait un fichier Excel qui décrit les différentes étapes de la création. Il sera en annexe de ce fichier.

Voici les différentes questions auxquels je n'ai pas répondu au cours de ce rapport :

— Quel est le nombre maximum de caractères (char) différents ?

256 (8bits) dont les 128 premiers forment le code ASCII (codé sur 7bits)

— Comment représenter l'arbre de Huffman ? Si l'arbre est implémenté avec des tableaux (fg, fd, parent), quels sont les indices des feuilles ? Quelle est la taille maximale de l'arbre (nombre de noeuds) ?

On peut représenter l'arbre dans une structure qui va contenir :

Un tableau contenant un « nœud ». Un « nœud » étant une structure composée de :

Un champ char qui stockera le caractère

Un champ int qui stockera le nombre d'occurrences de ce caractère dans le fichier (permet de trouver la proba d'apparition)

Un champ int « fils gauche » qui stockera l'indice du nœud fils « situé » à gauche

Un champ int « fils doit » qui stockera l'indice du nœud fils « situé » à droite

Un champ int parent qui stockera l'indice du nœud « parent »

Le nombre de nœuds (2 * feuilles -1) : pour stocker les feuilles et les nœuds internes (dont le maximum est nombre de feuilles – 1. Ça se prouve par récurrence). Les caractères sont stockés de l'indice 0 à l'indice « taille du tableau -1 », à l'indice 0 se trouve le caractère dont le code ASCI I est le plus petit.

Le nombre de feuilles max est 256, la taille max du tableau est : 2 * 256 – 1 = 511

— Comment les caractères présents sont-ils codés dans l'arbre ?

Les caractères sont représentés avec des '0' et des '1' en fonction de leur position dans l'arbre.

— Le préfixe du fichier compressé doit-il nécessairement contenir l'arbre ou les codes des caractères ou bien les deux (critère d'efficacité) ?

Le préfixe (que j'ai appelé en-tête dans mon rapport) est composé de

- le nombre de bits total dans le fichier (si le nombre de bits est supérieur à 255, on écrit autant d'octets de 8 bits de 1 (1111 1111) tous les 255 bits puis le reste.
- le nombre de code distinct dans le fichier (qui lui est au maximum de 255)

Pour chaque caractère distinct :

- la valeur de son indice (son code ASCII)
- la longueur de son codage
- la valeur décimale de code binaire (si le nombre de bits est supérieur à 255, on écrit autant d'octets de 8 bits de 1 (1111 1111) tous les 255 bits puis le reste.

— Quelle est la taille minimale de ce préfixe (expliquer chaque champ et sa longueur) ?

Si fichier en entrée a 1 seul caractère :

- 1 bit pour le nombre de bits total dans le fichier.
- 1 bit pour le nombre de code distinct dans le fichier
- 8 bits pour la valeur de son indice (son code ASCII)
- 1 bit pour la longueur de son codage
- 1 bit pour la valeur décimale de code binaire.

Je trouve 12 bits au minimum.

— Si le dernier caractère écrit ne finit pas sur une frontière d'octet, comment le compléter ? Comment ne pas prendre les bits de complétion pour des bits de données ?

On complète avec des bits à 0 (voir fonction AjoutHuff)

— Le décompresseur doit-il reconstituer l'arbre ? Comment ?

Non, il ne doit pas le reconstituer, il doit reconstituer le tableau permettant de trouver la relation

entre une chaine de bits et un des codes.

J'aimerais aussi conclure sur une remarque. Lors de nos nombreux tests, nous avons remarqué que sur les fichiers de petites taille, le fichier compressé prend au final plus de place que l'original à cause de l'entête.

Annexe