LAOUAR Augustin, ZEDDAM Thinhinane

RAPPORT DE PROJET DE COMPRESSION DE DONNEES

LZW - HUFFMAN

Avant-propos :

Nous avons réaliser ce projet dans le but d'avoir une compression et une décompression optimale via les algorithmes LZW et Huffman. Nous avons conscience que ce n'est pas encore parfaitement optimisé, mais nous avons plusieurs idées d'optimisation, dont nous vous avons fait part dans ce rapport.

Nous tenons aussi à précisé que nous avons travailler avec la table ascii standard (et non étendu), car nos compilateurs n'ont pas de table ascii étendu. Cependant, notre code serait facilement adaptable à une version sur table ascii étendu, car il faudrait simplement remplacé à chaque fois la taille de nos tableaux par 256 au lieu de 128.

LZW

Structures utilisés :

dicot_t

```
typedef struct {
        char * mots[3840];
        int nb_mots;
} dico_t;
```

Cette structure représentera notre dictionnaire.

Compression :

Fonctions Clés :

int ajouterDico(char* c,int reset)

Permet d'ajouter un mot (char *) dans le dictionnaire. Le parametre reset permet de spécifié à partir de quel valeur il faudra supprimer les données lorsque le dictionnaire est plein, nous vous donnerons plus de détails plus bas.

int estDansDico(char * c)

Vérifie si un mot (char *) est présent dans le dictionnaire ou non. Si le mot est un caractère simple (donc une chaine de longueur 1), on va renvoyer le code ascii de ce caractère, sinon cette fonction va simplement parcourir le dictionnaire entièrement, et renvoyer l'indice au quel est stocké le mot, ou -1 si il n'est pas présent sans le dictionnaire.

unsigned long int compressionLZW(char * nomEntre, int reset, int sock)

Pour compresser un fichier, notre algorithme LZW va parcourir le fichier caractère par caractère, les caractères seront stockés temporairement dans une chaine de caractère, et à chaque fois nous allons vérifier si la chaine existe dans le dictionnaire. Si c'est le cas, alors nous continuons à parcourir le fichier. Sinon, nous allons émettre la dernière chaine connu du dictionnaire (donc celle obtenu avant l'ajout du dernier caractère), puis ajouter celle nouvelle chaine dans le dictionnaire, en prenant le premier code disponible. Puis nous allons recommencer l'opération cette fois ci en repartant du dernier caractère parcouru.

Nous utilisons une variable 'used' ainsi qu'un tableau de 2 entiers afin de pouvoir envoyé les codes deux par deux, lorsque nous devons émettre un code, si used = 0, alors on le stock dans la première case du tableau, sinon, on le stock dans la deuxième case puis on émet le tableau, qui sera par la suite transformer en 3 octets, afin d'économiser 8 bits, car nous passons de 2 entiers (16 bits) à 3 char (8 bits). Sur certaine machine le gain est encore plus important car les entiers sont codés sur 4 octets. Ceci est possible seulement car la valeur des codes ne dépassera pas une valeur supérieur à 4096, et sera donc au maximum codé sur 12 bits.

Décompression :

Fonctions Clés :

int ajouterDico(char* c,int reset)

Même fonction que pour la compression.

bool codeDansDico(int code)

Cette fonction utilise le même principe que estDansDico, à la différence que nous ne cherchons pas un mot mais un code. Si ce code est un code de la table ascii, nous renverrons vrai, sinon nous allons vérifié si la case du tableau des mots du dictionnaire à l'indice de la valeur de code n'est pas null.

void decompressionLZW(char * nomEntre, char* nomSortie, int reset)

La décompression utilise le même principe que la compression. L'algorithme va parcourir le fichier de code, et reconstruire le dictionnaire, exactement dans le même sens que la compression, ce qui permet de reconstruire le dictionnaire sans avoir à le transmettre avec le fichier compresser. Nous allons donc parcourir le fichier compressé code par code, et reconstruire le dictionnaire et le fichier en même temps.

Optimisations :

Lorsque le dictionnaire est plein, il faut un moyen de pouvoir continuer la compression. Le meilleur moyen que nous avons trouvé est de réinitialisé le dictionnaire.

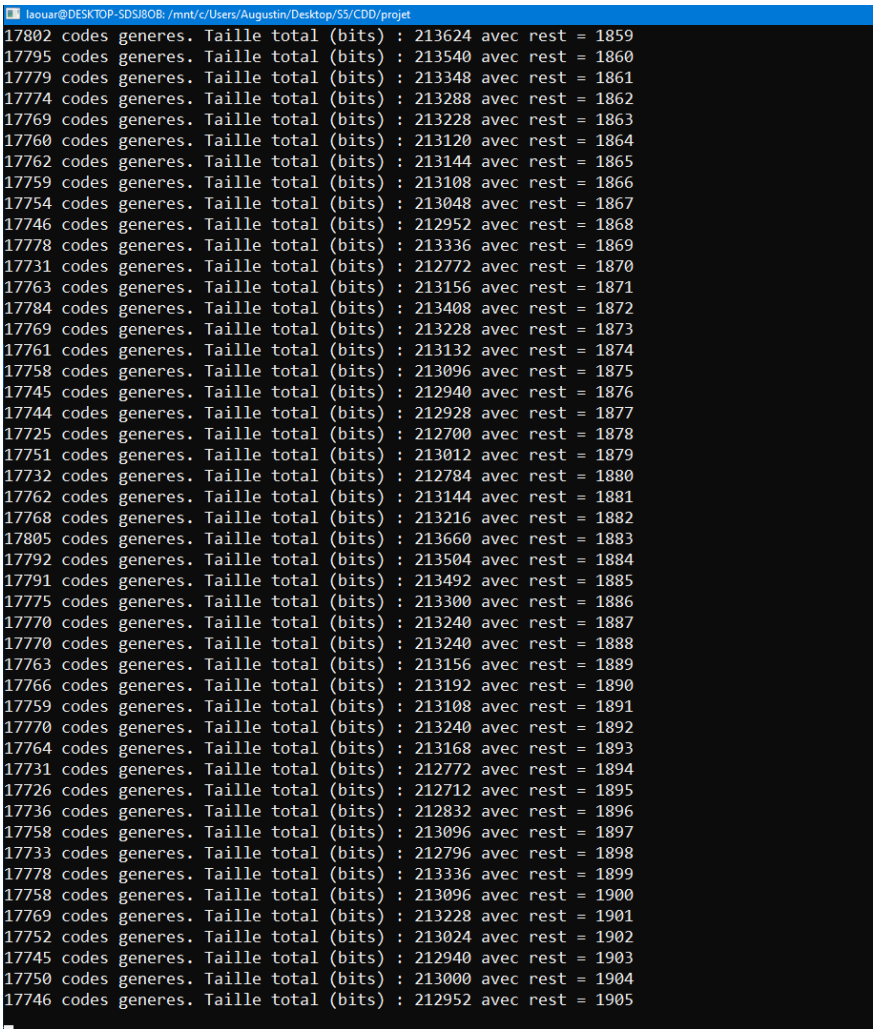
Problème : à quel valeur réinitialisé le dictionnaire pour un codage optimal ?

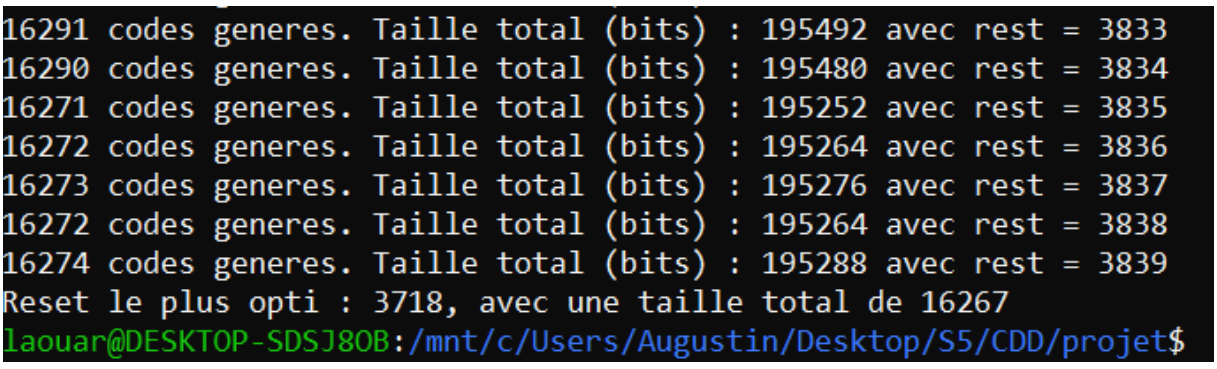
La première valeur nous venant en tête est 0. Pourtant, après réflexion, perdre tous les mots déjà codés dans le dictionnaire pour recommencer à 0 n'est pas très judicieux.

Nous avons alors tester de retirer seulement la dernière valeur du dictionnaire pour la remplacer pour une nouvelle. Résultat : pour un fichier de 69 000 octets, on obtient une compression à 24 000 octets en remettant à 0 et 19 000 en enlevant seulement la dernière valeur.

Cependant, si le début et la fin du fichier sont très différents, cette dernière solution n'est vraiment pas optimale.

Nous avons alors décidés de tester toutes les valeur possibles de réinitialisation du dictionnaire entre 0 et 3840.





Apres ce test, nous avons remarqué qu'il est préférable de supprimer seulement les dernières valeurs du dictionnaire, nous avons donc rajouté un paramètre nommé 'reset' à notre algorithme, qui sert à indiqué à partir de quel valeur il faut réinitialisé le dictionnaire pour une compression optimal.

Une idée d'optimisation serait de pouvoir donner le choix à l'utilisateur de choisir entre optimisé ou non sa compression. Si il décide que oui, alors nous allons faire des tests de compression, et voir la quel est la plus optimal (qui à le meilleur taux de compression). Lorsque celle-ci est trouvé, nous compresserons le fichier avec cette dernière. Evidement, nous n'allons pas tester toutes les valeurs, mais seulement des valeurs clés : par exemple le niveau 1 d'optimisation serait de faire le test avec 0, 1900 et 3800. Un niveau 2 pourrait être de tester avec une dizaine de valeur, et le niveau 3 par chaque tranche de 100 (0, 100, 200 …). Plus le niveau d'optimisation est élever, plus la compression prendrais du temps.

HUFFMAN :

Les structures utilisés :

Les tableaux pour stocké la table de codes ainsi que le tableau d'occurrences.

Les listes doublement chainés pour stockés la liste des éléments à ajoutés dans l'arbre.

Les arbres binaires, avec un pointeur sur le nœud père afin de faciliter le parcours depuis les feuilles jusqu'à la racine.

Compression :

void nbOccurences(ulint * tab, FILE * entre)

Nous parcourons le fichier passer en entrer, et pour chaque caractère rencontré, nous allons incrémenter la case du tableau à l'indice associé à son code ascii, les cases du tableau son initialement initialisé à 0. Nous utilisons des unsigned long int car le nombre d'occurrences de chaque caractère peut vite dépassé la capacité d'un int avec des gros fichiers.

void supprOccurrencesNull(listeNoeud * l)

Cette fonction n'aura d'utilité que pour « nettoyer » la liste de Nœud, en supprimant les Nœuds associés à des caractères ascii qui ne sont pas présents dans le fichier à compressé. Pour ce faire, nous regardons seulement si le nombre d'occurrence de ce caractère est supérieur à 0 ou non.

elemListeNoeud ** faibleFrequence(listeNoeud * l)

Retourne les deux nœuds de plus faible fréquence, qui auront pour but d'être relié pour construire l'arbre d'Huffman. Pour ce faire, nous parcourons seulement la liste de nœuds et effectuons une recherche des deux minimums.

void ecrireTable(char **t,int sock)

Cette fonction va envoyer au client la table représentant les codes de chaque caractère. Seuls les caractère présent dans l'arbre de Huffman (et donc présent dans le texte) seront envoyés.

unsigned char modifOctet(unsigned char c ,int pos, char* code)

Cette fonction va ajouter le code passé en paramètre dans l'octet à partir de la position pos, le code sera une chaine de caractère ne contenant que des '0' et des '1'. Elle servira lors de la compression, dans le but de concaténer les codes (char * code) et les envoyés sous forme d'octet), car ces codes ont une longueur variable.

arbre * initArbreHuffman(listeNoeud * elementsNonConnectes)

Permet de créer l'arbre d'Huffman à partir d'une liste d'éléments à connectés.

char * coderNoeud(noeud *n)

Permet de trouver le code associés au caractère ascii que représente un nœud de la liste. Pour ce faire nous allons simplement parcourir l'arbre depuis le nœud jusqu'à la racine, et vérifier si à chaque fois nous sommes sur un fils gauche ou un fils droite, et ajouter 0 ou 1 au code en conséquence. Le code sera renvoyé sous forme d'un char * .

void compressionHuffman(char * entre,int sock)

Avant de compresser, nous nous assurons que le caractère \0 est bien en fin de fichier, car c'est celui-ci qui marquera la fin du fichier. Si ce n'est pas le cas, il sera ajouté par le programme.

Pour compresser, nous allons tout d'abord avoir besoin de connaitre le nombre d'occurrences de chaque caractère dans le fichier. Pour ce faire nous allons créer un tableau de 128 unsigned long int (256 avec la table ascii étendu) tous initialisés à 0, et pour chaque caractère rencontré, nous allons incrémenter la case à l'indice correspondant à son code ascii de 1 .

Nous allons ensuite devoir créer l'arbre d'Huffman, mais contrairement à d'habitude, nous allons commencer par les feuilles puis connecté les nœuds 2 à 2 jusqu'à avoir un arbre complet. Pour ce faire nous allons d'abord utilisé une liste doublement chainés pour stockés les nœuds de l'arbre qui sont encore à connectés, dans la quel nous allons ajouté tous le tableau créer précédemment, puis, via une fonction de 'nettoyage', retirer tous les nœuds inutiles, c'est-à-dire les nœuds représentants des caractères qui ont une occurrence à 0, donc non présent dans le fichier.

Nous allons ensuite créer l'arbre, en créant un nœud reliant à chaque fois les deux nœuds de plus petites fréquences, ces deux nœuds seront donc retirer de la liste d'éléments à connecter, et on y rajoutera le nouveau nœuds. On répète ceci jusqu'à ce que la liste ne contienne qu'un seul nœud (la racine).

Il faut maintenant construire la table contenant les codes associés à chaque caractère. Pour ce faire, nous allons simplement parcourir l'arbre depuis chaque feuille, jusqu'à la racine, et construire le code en même temps que nous parcourons. Si le nœud courant est un fils droit, alors nous ajouterons 1 au code, sinon 0. Le code sera stockés sous forme d'une chaine de caractère, tout ceci sera fait par la fonction 'coderNoeud'.

Lorsque la table de code est créer, nous allons l'envoyer via la socket, afin qu'elle soit récupérer par le programme client. Le code -1 marquera la fin de ces envois.

Pour compresser le fichier, nous allons maintenant le reparcourir (en prenant bien soin de replacé la tête de lecture au début du fichier avec la fonction lseek), puis coder chaque caractère avec son code associé, que nous allons ajouter dans un octet avec la fonction 'modifOctet', cela est particulièrement délicat car les codes ont des longueurs variables. Il faudra donc remplir les octets progressivement, comme l'on remplirait des wagons progressivement avec des groupes de personnes de tailles différentes. Dès qu'un octet est pleins, il est envoyé au client via la socket.

Décompression :

char ** LireTable(char *f,char ** res)

Permet de lire la table contenant la liste des codes de Huffman, stockés dans le fichier de nom f passé en paramètres et les stockent dans un tableau de chaines de caractères qu'elle retourne , les indices seront les codes ascii des caractères, et les chaines de caractères seront les codes associés ('0101' par exemple).

char * supprimerDebut(char* s, int i)

Permet de supprimer les i bit de poids fort d'une chaine de caractère passé en paramètre, puis retourne la nouvelle chaine.

char * trouverCode(char * s, char ** table,char * c)

Permet de trouvé un code dans la table de codes (représenter sous forme d'une liste de char *). Parcours simplement le tableau pour trouver le code passé en paramètre (char * s). Si il n'est pas dans le tableau, retourne NULL.

void decompressionHuffman(char * entre, char * sortie,char* fichierTable)

Avant toute chose, il faut récupérer la table que nous a envoyé le serveur. Le programme principale l'aura récupérer et stocké dans un fichier .txt, il faut maintenant seulement aller lire ce fichier et en déduire le tableau de codes associés. Cela est possible grâce à la fonction 'LireTable'.

Pour décompresser, le principe est simple, il faut prendre les codes et retrouvé leurs caractères associés dans la table que nous avons récupérer. Ceci est possible car lors de la création de l'arbre d'Huffman, chaque caractère aura reçu un code de sorte à ce qu'il n'y aura aucune ambiguïté entre eux. Il faut donc maintenant lire le fichier, octet par octet, et pour chaque octet lire bit par bit, et comparé avec chaque code présent dans la table. Nous utiliserons une variable temporaire qui stockera les bits un à un (sous forme d'une chaine de caractère), puis comparera avec les codes de la tables. Lorsqu'un code est trouvé, on écrit dans le fichier de sortie le caractère associé, puis on vide la variable temporaire, et ainsi de suite jusqu'à ce que le fichier soit entièrement recréer (c'est-à-dire que nous avons atteint ce fameux '\0').

Client / Serveur

Nous avons utilisé la bibliothèque socket en C afin de réaliser une interface client / serveur. Le serveur va compresser le fichier demandé puis l'envoyé octet par octet pour Huffman, ou bien par triplet d'octets pour LZW. De son coté, le client va lui lire ces octets puis recréer le fichier compresser de son côté. Il va ensuite appliqué l'algorithme de décompression à ce nouveau fichier afin de recréer le fichier demandé.

Attention : nous avons remarqué qu'il est possible que la connexion entre client et serveur ne fonctionne par pour une raison qui nous est inconnu (nous suspectons un problème de réseaux et non de programmation) . Si cela arrive, il suffit de relancer le terminal du serveur.

Optimisation :

Une première optimisation serait de pouvoir effectué la compression à la volé en pendant que nous recevons les octets du serveur. Cela permettrais une petite économie de temps.

Deuxièmement, nous pourrions non pas envoyé les octets 1 par 1 ou 3 par 3, mais par gros paquets, de plusieurs milliers, afin de limiter le nombre de requêtes sur le réseaux, car avec des gros fichiers, nous atteignons facilement des millions d'octets , et donc des millions de requêtes.