

Etudiants : NICLAS Clara
BRUMELOT Noé
MAHRAZ Camélia
RAJAOSAFARA Laura
PISCOPELLO Enzo

Enseignant : M.Eutamene

Matière : SDD

Promotion : L2-BN



RAPPORT DU PROJET : Codage Huffman

Remarques de l'enseignant :

Note : /20

Table des matières

INTRODUCTION	3
Analyse	4
Conception.....	6
Organisation	9
➤ Problèmes techniques	9
➤ Organisation dans le groupe.....	11
CONCLUSION	13

INTRODUCTION

L'objectif de ce projet est de concevoir un code Huffman, c'est-à-dire d'implémenter un algorithme de compression. Cela signifie que le programme va chercher à réduire la place que prend une information sans pour autant perdre des données. En effet, notre but est de traduire un caractère en un code binaire en fonction de sa fréquence d'apparition. Dans ce projet le but sera de compresser des fichiers textes, en réduisant donc le nombre de bits codés par chaque caractère. Le projet est divisé en trois grandes parties afin de mieux nous accompagner dans notre conception du programme :

- La conversion de lettres en bits
- Les prémisses du code Huffman
- L'optimisation du code

Ce projet retrace et utilise presque toutes les notions vues au cours de ce semestre telles que les arbres, les listes doublements chaînées etc... ce qui nous permet de les appliquer dans un exercice concret et complet.

ANALYSE

Comme introduit précédemment, le problème qui nous a été posé est celui de la compression de fichiers textes. Notre objectif a été de réaliser un code Huffman afin d'associer à chaque caractère présent dans un fichier texte un code assez court pour gagner de la place en mémoire. Nous avons donc souhaité réduire au maximum le nombre de bits correspondant aux caractères correspondants.

Pour commencer, nous avons réalisé un code permettant de traduire un texte en bits correspondants aux caractères présents dans le fichier, dans le but de déterminer combien de bits prends notre texte en mémoire à l'origine, donc avant compression. Ensuite, nous avons traduit notre texte à l'aide du code Huffman afin de comparer et d'estimer la place que nous avons pu gagner en mémoire.

Pour parvenir à réaliser cela, nous avons dû structurer notre projet en différentes fonctions, remplissant un objectif chacune. Les fonctions les plus importantes à réaliser ont été les suivantes :

- **Element* occurrence_caractere()** : cette fonction a eu pour but d'obtenir une liste constituée des caractères et de leur occurrence à partir des caractères présents dans un fichier texte (Alice.txt). Le retour de la fonction est donc la liste créée, et elle ne prend pas de paramètres mais elle prend bien en compte le fichier texte utile à la fonction dans son développement. On a utilisé plusieurs fonctions codées préalablement dans cette fonction, comme **Element* condition_liste_caracteres_occurrences(Element* liste, char caractere)** et **void ajouter_valeur_liste_fin(Element* l, char caractere)**. La première fonction permettant de savoir si un caractère est présent dans le texte, afin d'ajouter 1 à son occurrence, et la deuxième permettant d'ajouter chaque élément créé de la liste à la fin de celle-ci.
- **Noeud* creation_liste_de_noeud(Element* l, int poids_precedent)** : Cette fonction permet de créer une liste de nœuds qui représente notre arbre Huffman. Cette fonction utilise plusieurs fonctions :
 - **void supprimer_maillon_liste(Element** l, int pos)** : Cette fonction permet de supprimer un élément d'une liste selon une position donnée en paramètre.

- **int position_min_liste(Element* l, int minimum)** : Cette fonction permet de déterminer la position de l'élément qui possède la plus petite occurrence dans une liste.
- **void afficher_Liste_de_noeuds(Noeud* arbre)** : Cette fonction permet d'afficher une liste de nœuds. Cette liste de nœuds représente notre arbre Huffman.
- **void fichier_texte_arbre_Huffman(Noeud* arbre)** : Cette fonction permet de stocker dans un fichier texte le dictionnaire issu de l'arbre Huffman.

Dans notre fichier main.c, on convertit dans un premier temps les caractères du texte « Alice » à partir du fichier « Alice.txt » en binaire et on les stocke dans le fichier « AliceBinaire.txt ». On compte ensuite le nombre de caractères présents dans chacun de ces fichiers à l'aide des fonctions **int compter_carac_txt()** et **int compter_carac_txt_bin()**.

On peut ensuite créer et afficher notre liste dont les éléments sont constitués d'un caractère et de son occurrence dans le fichier « Alice.txt ».

Pour pouvoir créer l'arbre de Huffman, on crée une liste de nœuds à partir des éléments présentant le moins d'occurrences, puis les parents de ces éléments là en veillant à bien faire la somme des poids des enfants. On inverse ensuite notre liste de sorte à pouvoir afficher notre arbre du haut vers le bas, et non du bas vers le haut comme il a été créé.

On crée ensuite notre dictionnaire Huffman, et on remplit notre fichier « Dico.txt ». Enfin, on remplit notre fichier « huffman.txt » dans lequel la phrase du texte « Alice » est traduite à partir du dictionnaire que l'on a créé.

CONCEPTION

Afin de remplir nos objectifs, nous avons créé 2 fichiers.c, nommés « PARTIE1.c » et « PARTIE2.c » dans lesquels nous avons implémenté le code de plusieurs fonctions. Dans les fichiers.h correspondant, nous avons défini les prototypes des fonctions en question. Nous avons également créé trois fichiers textes « Alice.txt », « Dico.txt » ainsi que « Huffman.txt ».

Dans le premier, nous avons inséré la phrase présentée dans le sujet, à compresser en mémoire. Dans le second, notre dictionnaire en fonction de l'arbre Huffman que nous avons codé apparaît. Enfin dans le troisième, après compilation du code, le texte (à propos d'Alice) apparaît après y avoir appliqué notre dictionnaire Huffman.

Nous avons créé plusieurs fonctions pour compter les occurrences des caractères de notre fichier texte.

Element* occurrence_caractere() : Fonction qui parcourt le fichier texte et ajoute le caractère qu'elle lit dans une LSC s'il n'existe pas encore dans la liste et augmente son occurrence si le caractère est présent dans la liste. Cette fonction renvoie le pointeur du premier élément de la liste (liste composée de chaque caractère du fichier texte ainsi que leur occurrence). Dans cette fonction, nous appelons plusieurs fonctions :

- **Element* condition_liste_caracteres_occurrences(Element* liste, char caractere)** : Fonction qui vérifie si le caractère existe ou non dans la liste et s'il existe, on l'ajoute à une LSC et on augmente son occurrence. Cette fonction prend en entrée la liste des caractères et le caractère dont on va compter l'occurrence. Cette fonction renvoie le pointeur du premier élément de la liste.
- **Element* creer_element(char caractere)** : Fonction qui crée un élément d'une LSC avec le caractère et son occurrence. Elle retourne le nouvel élément créé.

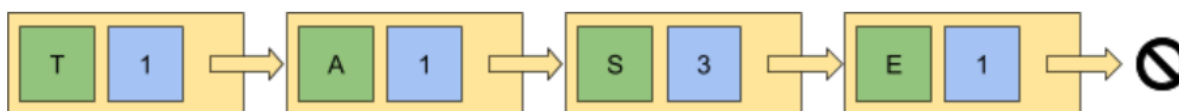


Figure 1 : exemple de la LSC qui contient les caractères du mot TASSES ainsi que leur occurrence

Pour élaborer l'arbre Huffman, nous avons réalisé plusieurs fonctions.

Noeud* creation_liste_de_noeud(Element* l,int poids_precedent) : Fonction qui permet de créer une liste de nœuds. Cette fonction prend en entrée une liste composée des caractères et de leurs occurrences, issus du fichier texte nommé "Alice.txt" et un entier nommé poids_precedent qui va nous être utile pour calculer les poids des différents nœuds de notre arbre. Cette fonction va nous renvoyer la liste de nœuds créée.

Dans cette fonction, nous faisons appel à d'autres fonctions :

- **void supprimer_maillon_liste(Element** l, int pos)** : Fonction qui permet de supprimer un élément d'une liste selon une position donnée. Cette fonction prend en entrée la liste où on veut supprimer un élément et la position de l'élément que l'on veut supprimer.
- **int position_min_liste(Element* l, int minimum)** : Fonction qui permet de déterminer la position de l'élément qui possède la plus petite occurrence dans une liste. Cette fonction prend en entrée la liste que nous allons parcourir et l'entier minimum. A l'issue de cette fonction, ce dernier prendra la valeur minimum de la liste parcourue. Ce minimum sera ainsi renvoyé.

void afficher_Liste_de_noeuds(Noeud* arbre) : Fonction qui permet d'afficher une liste de nœuds. Elle prend en entrée la liste de nœuds que nous voulons afficher et plus précisément notre arbre Huffman.

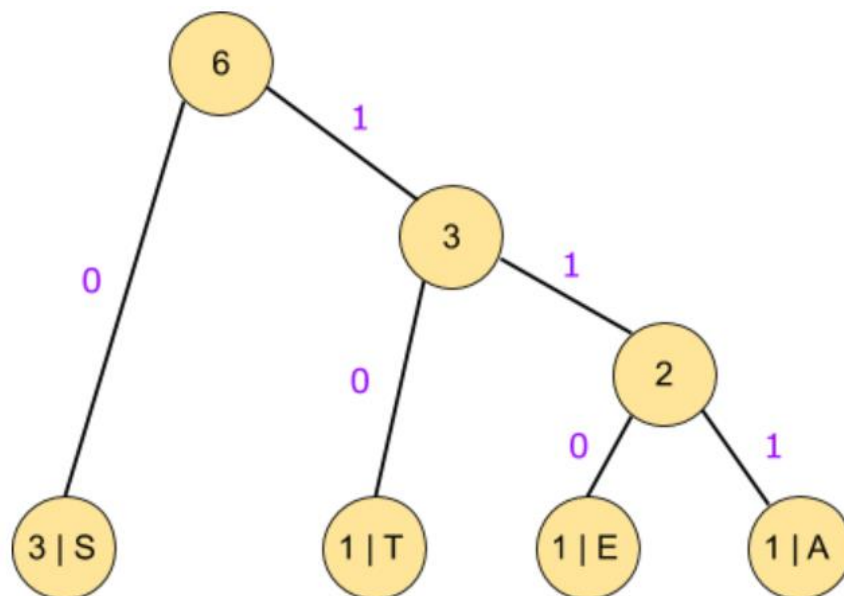
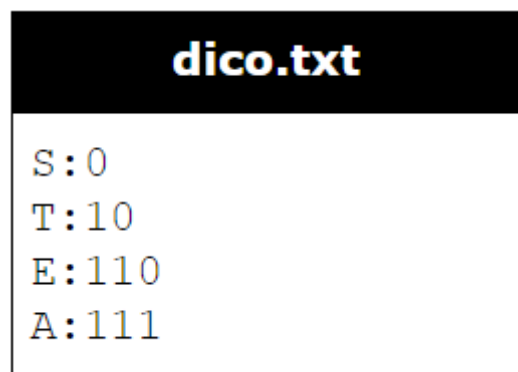


Figure 2 : exemple de l'arbre d'Huffman issu du mot TASSES

A partir de cet arbre de Huffman, nous avons créé son dictionnaire grâce à la fonction suivante :

void fichier_texte_arbre_Huffman(Noeud* arbre) : Fonction qui permet de stocker dans un fichier texte le dictionnaire issu de l'arbre Huffman. Elle prend en entrée la liste de nœud qui représente notre arbre Huffman.

Après avoir appelé cette fonction, nous avons constaté que notre arbre possédait les bonnes valeurs mais que ces dernières n'étaient pas triées correctement. Ainsi nous avons créé la fonction **void inverser_liste(Noeud** debut_arbre)**. Elle permet d'inverser les valeurs de notre liste de nœuds : nous avons ainsi obtenu notre arbre Huffman.



```
dico.txt
S:0
T:10
E:110
A:111
```

Figure 3 : exemple du dictionnaire issu de l'arbre d'Huffman à partir du mot TASSES

Pour maintenant utiliser l'arbre de Huffman, nous avons créé ces 3 fonctions qui permettent de compresser un fichier texte grâce à une recherche du caractère dans le dictionnaire issu de l'arbre Huffman :

- **void dico_recherche(char caractere, FILE** fichier3, FILE* fichier)** : Fonction qui permet de parcourir le dictionnaire issu de l'arbre Huffman avec le caractère qui a été récupéré du fichier texte à compresser. Elle prend en entrée le caractère, ainsi que le fichier dico et le fichier dans laquelle la nouvelle liste binaire va être affichée.
- **void trad_text_bin_huffman()** : Fonction qui permet de lire le fichier « Alice.txt » caractère après caractère et de le traduire selon le dictionnaire de Huffman à partir du fichier « Dico.txt ». Cette traduction s'affiche dans le fichier « huffman.txt ». On fait appel dans cette fonction à la fonction void dico_recherche(char caractere, FILE** fichier3, FILE* fichier). Ainsi, on parcourt le fichier « Dico.txt » et on repère les caractères similaires avec ceux du fichier « Alice.txt ». Lorsque ceux-ci correspondent, on les insère dans le fichier « Huffman.txt » en respectant le code du dictionnaire.

ORGANISATION

➤ Problèmes techniques :

Lors de la réalisation de notre projet, nous avons rencontré plusieurs problèmes techniques. Voici quelques exemples de problèmes rencontrés :

- Fonctions pour compter les occurrences

Nous avons rencontré un important problème lors de la conception des fonctions pour compter les occurrences notamment avec `occurrence_caractere()`. En affichant le résultat de cette fonction, on a constaté qu'elle ne comptait les occurrences de notre fichier texte que jusqu'à la 4ème lettre avant que le programme ne s'arrête automatiquement.

On a ainsi créé un projet Test permettant d'effectuer des tests de nos fonctions.

```
Element* occurrence_caractere()
{
    char caractere;
    Element* liste = NULL;
    int i = 0;

    FILE* fichier = NULL;
    fichier = fopen("Texte.txt", "r");

    if(fichier != NULL){
        while((caractere = fgetc(fichier)) != EOF){
            //printf("%c\n", caractere);
            /*if(i == 0){
                liste = creer_Element(caractere);
                i = i+1;
            }
            else{
                ajouter_valeur_liste_fin(liste, caractere);
            }*/

            if(i == 0){
                liste = creer_Element(caractere);
                i = i + 1;
            }
            else{
                liste = condition_liste_caracteres_occurrences(liste, caractere);
            }
        }

        fclose(fichier);
        return liste;
    }
}
```

Nous pouvons observer les différents tests que nous avons effectué à travers le code mis en commentaire ci-dessus. Nous avons décidé d'afficher à chaque fois le caractère étudié pour vérifier que chaque caractère soit pris en compte.

En testant la fonction `creer_element`, on remarque cependant qu'il y a un problème lors de l'allocation de la mémoire de l'élément « `new_element` ». Cela explique donc le problème de notre fonction qui compte les occurrences des caractères.

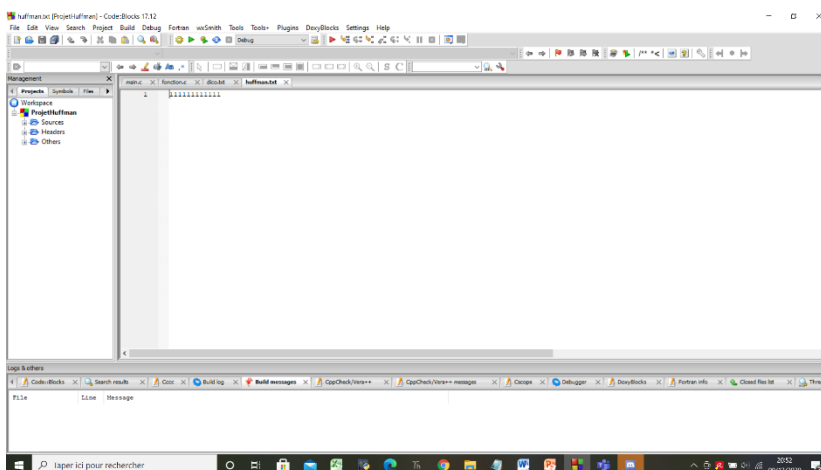
Après avoir réglé ce problème, nous avons pu obtenir le résultat attendu : une liste composée de chaque caractère du fichier texte avec son occurrence.

```
194 Element* occurrence_caractere()
195 {
196     char caractere;
197     Element* liste = NULL;
198     int i = 0;
199
200     FILE* fichier = NULL;
201     fichier = fopen("Allice.txt", "r");
202
203     if(fichier != NULL){
204         while((caractere = fgetc(fichier)) != EOF){
205             //printf("%c\n", caractere);
206             /*if(i == 0){
207                 liste = creer_Element(caractere);
208                 i = i+1;
209             }
210             else{
211                 ajouter_valeur_liste_fin(liste, caractere);
212             }*/
213
214             if(i == 0){
215                 liste = creer_Element(caractere);
216                 i = i + 1;
217             }
218             else{
219                 liste = condition_liste_caracteres_occurrences(liste, caractere);
220             }
221         }
222
223         fclose(fichier);
224         return liste;
225     }
226 }
```

Pour vérifier le résultat, nous avons décidé d'afficher le résultat obtenu lors de l'affichage de la liste comptant les occurrences des caractères du fichier texte.

- Fonctions pour traduire un fichier texte en binaire, basée sur un dictionnaire Huffman

Après avoir obtenu notre dictionnaire Huffman dans le fichier texte “dico.txt”, nous devons traduire le texte issu du fichier “Alice.txt” en binaire selon le dictionnaire Huffman. Pour cela, nous avons créé plusieurs fonctions qui se nomment void trad_text_bin_huffman() et void dico_recherche(char caractere, FILE** fichier3, FILE* fichier). En parcourant les fichiers “dico.txt” et “Alice.txt” dans ces fonctions, nous avons constaté que qu’elles ne prenaient en compte que la première lettre du fichier texte “Alice.txt” avant que le programme ne s’arrête.



Lecture du premier caractère du fichier “Alice.txt” soit “A” dans le fichier “huffman.txt”

Après plusieurs tests, nous avons constaté qu’il était nécessaire de placer notre curseur au début du fichier ”dico.txt” avec `fseek(fichier,0, SEEKSET)` pour que notre fonction puisse fonctionner.



Après avoir trouvé une solution à notre problème, nous avons bel et bien obtenu la phrase issue du fichier “Alice.txt” en binaire basé sur l’arbre Huffman.

Finalement, nous avons rencontré un problème lors du début de la partie 3, que nous avons codée mais qui n’était malheureusement pas fonctionnelle. En effet, nous avons compris le principe de la recherche dichotomique. Cependant, nous avons eu des difficultés à créer un tableau de nœuds, car nous ne connaissions pas la taille de celui-ci. Nous avons tenté de lui attribuer une taille assez aléatoire pour lire le fichier texte et ajouter au tableau de nœuds une occurrence si le caractère avait été trouvé ou le nœud du caractère dans le cas contraire, mais nous ne sommes pas parvenus à créer le tableau de nœud.

➤ Organisation dans le groupe :

Pour mener à bien ce projet, nous nous sommes organisés de sorte à répartir le travail entre les différents membres de l’équipe ainsi que dans le délai de temps imposé pour parvenir à réaliser le maximum de ce projet. Nous avons travaillé en commun pendant les 4 séances de TP accordées au projet en codant ensembles. Pour certaines fonctions, nous nous sommes partagé le travail à plusieurs, mais les fonctions plus simples ont pu être réalisées par un seul voire deux membres de l’équipe. Nous avons également consacré du temps à ce projet en dehors des séances de TP, et dès que l’un d’entre nous parvenait à coder une fonction ou présentait un problème, le reste des membres de l’équipe parvenait à l’aider ou à corriger ses erreurs.

Mis à part les problèmes techniques que nous avons réussi à régler, nous n'avons pas rencontré de problèmes d'autre ordre, donc pas de problèmes relationnels au sein de notre équipe efficace et organisée.

CONCLUSION

Pour conclure, ce programme permettant de coder l'arbre de Huffman nous a permis de compresser un fichier texte et de réduire sa taille en utilisant un dictionnaire à partir des caractères présents dans ce même fichier. Nous n'avons cependant pas terminé le programme mais nous aurions pu démontrer que ce code nous a permis de gagner de la place en mémoire et de l'optimiser afin de la réaliser en une très petite période de temps. Ce projet a été particulièrement enrichissant pour nous car nous avons pu utiliser la majorité des structures de données que nous avons appris à maîtriser durant ce semestre. Malgré quelques problèmes techniques rencontrés, nous avons persévéré afin d'obtenir notre résultat final (malheureusement non fini à 100%)

Lien GitHub :

<https://github.com/Clarinka/Projet-Huffman>