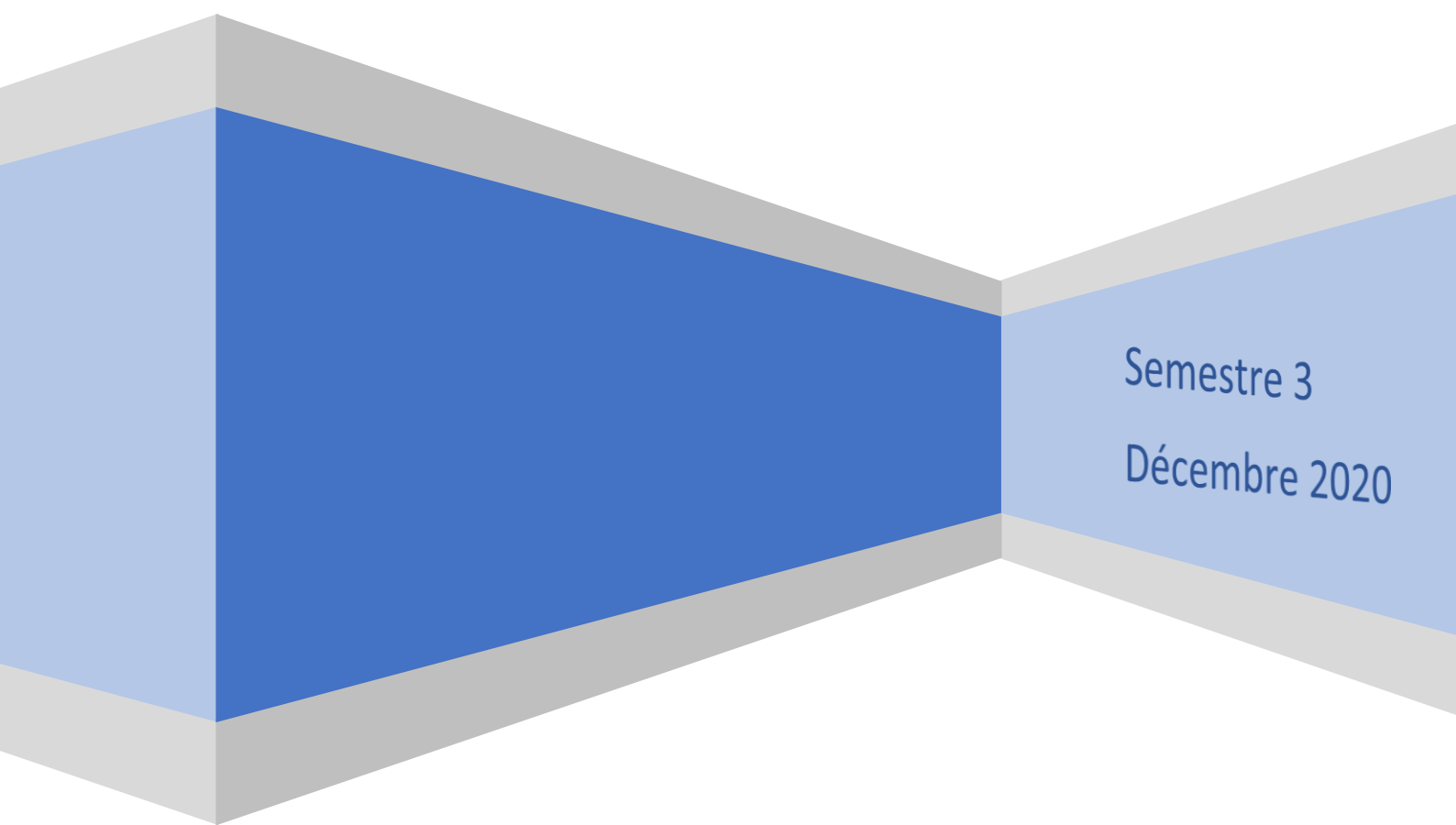


Algorithmique

Projet : Huffman

Hugo Ravaille
Corentin Péron
Arthur Loot
Jacques Klein
Nicolas Ferreira



Semestre 3
Décembre 2020

Table des matières

Introduction.....	3
Le projet en lui-même	3
Nos algorithmes	4
Optimisation.....	7
Difficultés rencontrées	7
Conclusion	8
Annexe.....	9
Structures.c	9
Compression.c	10
decompression.c	11
fonctions.c	11
Creation.c	12
Suppression.c	12
Print_structutres.c.....	12
AVL.c.....	13

Introduction

Ce projet a pour but le codage d'un « algorithme de compression de fichier » en C. Ce code a donc pour but de réduire la place que prends une information dans la mémoire sans avoir de pertes de données. Dans ce développement, nous allons étudier les fonctionnements d'un système de compression afin de pouvoir créer un programme en C, permettant de gérer « une compression » via un fichier externe et bien entendu de perfectionner notre savoir et notre méthode en programmation en C. Que ce soit avec les arbres et/ou les listes chaînées.

Le projet en lui-même

Comme dit précédemment, le projet a pour but de coder un algorithme de compression en C et donc de développer notre savoir dans ce langage. Le but de ce code est simple :

La première étape à suivre sera de transformer les caractères en un nombre binaire sur un octet de 8 bits. Chaque caractère aura ainsi un nombre associé selon les normes du tableau ASCII. La lettre "a" sera donc traduit par "01100001".

Ensuite pour simplifier la compression nous utiliserons le codage de Huffman, celui-ci a pour but de traduire une chaîne de caractère en code en fonction de la répétition d'un caractère par exemple le mot « vous » aura un code plus long que le mot « boss » car le mot « boss » a deux « s » donc plus une chaîne de caractère a de caractère identique plus sa traduction sera courte.

Nous allons utiliser ce que nous avons présenté précédemment en allant chercher nos caractères sur un fichier texte externe pour ensuite le traduire en binaire et le compresser.

Nos algorithmes

Dans cette partie, nous allons présenter quelques programmes qui nous ont servi de base et de croquis pour arriver au programme final. Tout ce que nous présenterons ici sera sous la forme d'algorithme afin de permettre une lisibilité maximale.

En premier lieu, nous allons montrer un croquis de programme principal qui nous a servi pour la structure et l'organisation du travail ; puis dans un deuxième temps, nous présenterons les fonctions principales

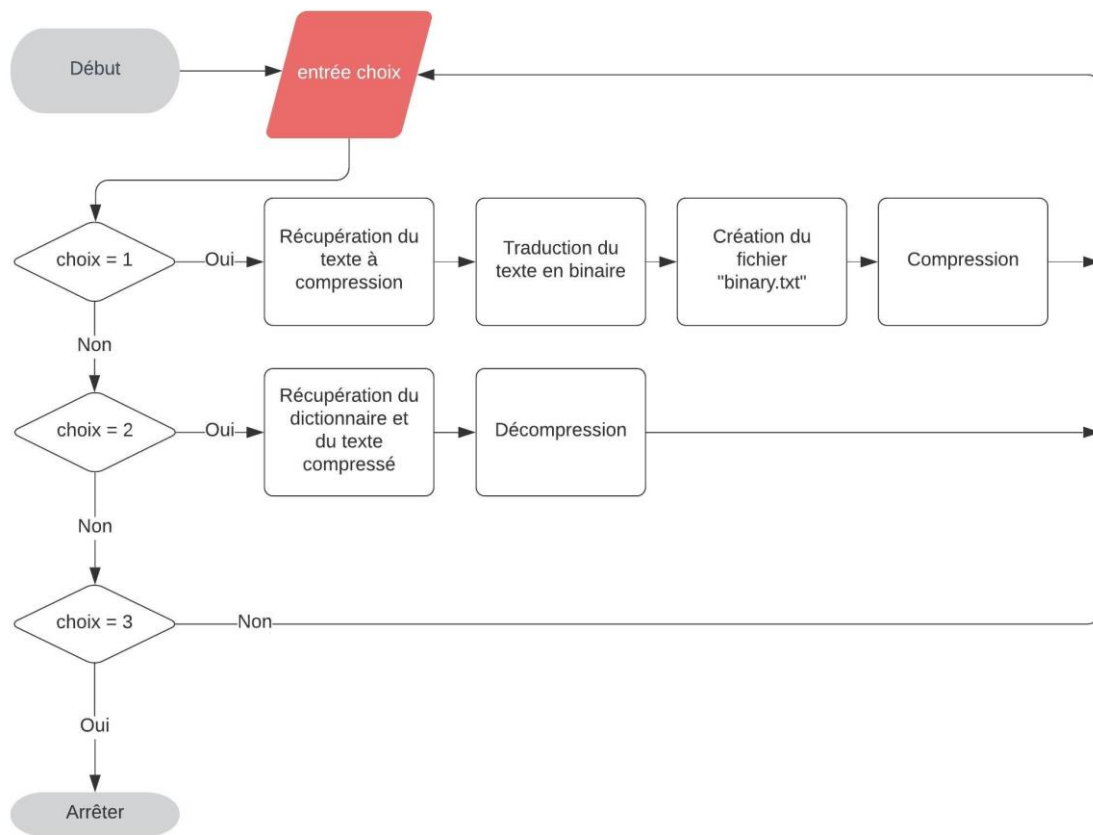
Donc pour commencer, voici le squelette de notre projet :

```
Fonctions/  
    AVL.c  
    AVL.h  
    creation.c  
    creation.h  
    fonctions.c  
    fonctions.h  
    print_structures.c  
    print_structures.h  
    suppression.c  
    suppression.h|  
.gitignore  
README.md  
compression.c  
compression.h  
decompression.c  
decompression.h  
main.c  
structures.h
```

Le projet est découpé en plusieurs fichiers, voici l'arborescence de ce dernier. Nous avons fait cela afin de permettre une meilleure lisibilité et modulation du projet. Ainsi chaque fichier permet d'effectuer des fonctions précises appelées dans les fichiers principaux.

Nous allons donc ensuite étudier notre programme principal et nous verrons avec celui-ci où sont localisées nos difficultés détaillées dans la partie suivante. Il y a dans cet algorithme plusieurs fonctions qui servent au bon fonctionnement du programme que nous détaillerons aussi par la suite :

Voici l'algorithme du main :



Lors du lancement du programme, l'utilisateur arrive sur un menu, où il doit choisir différentes options.

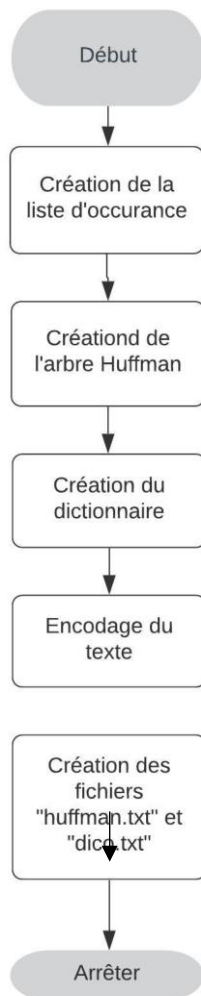
La première, lorsque l'utilisateur entre « 1 », il doit entrer le chemin du fichier à compresser. Le programme récupère le texte du fichier, pour ensuite produire une traduction en binaire du fichier, qui sera conservé dans le fichier « binary.txt ». Après avoir fait cela, le programme va appeler la fonction Compression. Pour finir il retourne au début du programme.

La deuxième, lorsque l'utilisateur entre « 2 », il doit entrer le chemin du fichier à décompresser. Le programme récupère le texte du fichier compressé et le dictionnaire correspondant. Ensuite il appelle la fonction Decompression. Pour finir il retourne au début du programme. Pour finir il retourne au début du programme.

La troisième, lorsque l'utilisateur entre « 3 », le programme s'arrête.

La dernière, si l'utilisateur insère une mauvaise réponse le programme revient au début.

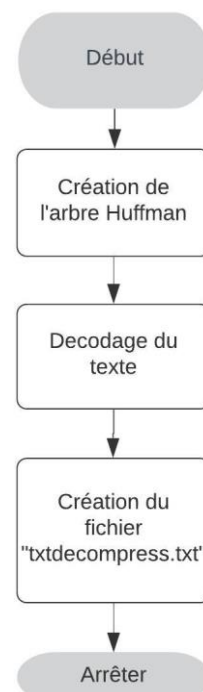
Ci-dessous l'algorithme de la fonction Compression :



Tout d'abord une liste d'occurrence sera créée à partir du texte à compresser. L'arbre Huffman sera ensuite créé à partir de la liste. Un dictionnaire contenant les caractères et leur nouvel encodage en binaire sera créé grâce à l'arbre Huffman. Ensuite le texte sera compressé en utilisant l'encodage des caractères dans le dictionnaire. Le texte compressé et le dictionnaire seront stockés dans leur fichier respectif, « huffman.txt » et « dico.txt ».

A droite l'algorithme de la Decompression :

Un arbre Huffman sera créé à partir du dictionnaire. Ensuite le texte pourra être décodé grâce à l'arbre. Le texte décodé sera ensuite stocké dans le fichier « txtdecompress.Txt ».



Optimisation

Nous avons procédé à certaines optimisations du programme. La première fut d'avoir utilisé un AVL pour créer la liste d'occurrence. Au lieu de parcourir la liste à la recherche d'un caractère afin d'ajouter +1 à son occurrence, l'AVL permet de réduire le parcours en classant les nœuds en fonction du code ascii de leur caractère. Ainsi lorsque nous recherchons un caractère dans l'arbre nous avons juste à comparer le caractère recherché et celui du nœud actuel.

La deuxième fut de aussi construire le dictionnaire sous forme d'AVL, ce qui permet de rechercher efficacement le caractère et son nouveau code binaire.

De plus pour la création de l'arbre Huffman, nous avons créé une fonction de tri. Qui nous permettra de trier la liste des arbres en fonction du nombre d'occurrence total des arbres. Pour ensuite récupérer les deux premiers arbres de la liste et les assembler. Et ensuite réinsérer le nouvel arbre dans la liste.

Difficultés rencontrées

Au cours de la création du projet nous avons rencontré plusieurs difficultés.

Dans un premier temps nous avons eu du mal avec les tableaux. Je m'explique, nous voulions utiliser les tableaux pour stocker le dictionnaire contenant le code binaire des caractères et pouvoir mettre ce même dictionnaire dans un fichier texte. Le problème c'est qu'en récursivité les tableaux n'étaient pas très maniables et nous avons dû trouver une solution qui répondait aux attentes tout en utilisant la récursivité pour stocker tout le dictionnaire et les fichiers texte. Nous avons donc remplacé les tableaux par les listes chaînées beaucoup plus maniables et pratiques en ce qui concerne la récursivité et l'utilité dont nous avons besoin.

Ensuite nous avons eu de grosses difficultés sur l'optimisation du projet. En effet, nous passions trop d'une structure de programmation (Tableaux, arbre, liste chaînée) à une autre. Par exemple quand nous devions passer d'un arbre à un tableau nous passions entre les deux par une liste pour nous simplifier la tâche et en sacrifiant l'optimisation du programme et aux répercussions que cela pourrait avoir.

Conclusion

Pour conclure, ce projet nous a permis de mettre en pratique ce que nous avons appris en cours comme les arbres binaire et les listes chaînées. Malgré les difficultés que nous avons rencontrées, avec les tableaux par exemple, nous avons su faire face et trouver des solutions adéquates afin de proposer un code fonctionnel et qui répondaient au maximum aux attentes du cahier des charges.

Ce projet nous a donc permis de perfectionner notre programmation dans le langage de programmation C.

La compression et la décompression de Huffman a donc été un projet assez difficile avec moins d'un mois pour le faire, mais malgré ça nous avons su surmonter ces difficultés et mettre en œuvre nos connaissances.

Annexe

Structures.c

Ce fichier contient toutes les structures nécessaires au projet.

```
typedef struct list{  
    char data;  
    int occu;  
    struct list *next;  
}List;
```

```
typedef struct code {  
    char data;  
    struct code *next;  
}Code;
```

```
typedef struct node{  
    char data;  
    int occu;  
    struct node *left;  
    struct node *right;  
}Node;
```

```
typedef struct listnode{  
    Node *data;  
    struct listnode *next;  
}ListNode;
```

```
typedef struct nodecode{  
    char data;  
    Code* code;  
    struct nodecode *left;  
    struct nodecode *right;  
}NodeCode;
```

Compression.c

Ce fichier contient toutes les fonctions nécessaires à la compression.

```
ListNode* AVLToListOccu(Node* B);  
Node* AddAVLOccurence(Node* B, char c, int* change);  
ListNode* CreationListOccurence(char* tab);  
void BubbleSort(ListNode** L);  
void BubbleSort(ListNode** L);  
Code* LetterToBinaryCode(Node* B, char letter, char c);  
ListCode* TreeToList(Node* B, ListNode* Occu);  
NodeCode* AddAVLDico(NodeCode* B, char c, Code* code, int* changement);  
NodeCode* CreationAVLDico(Node* B, ListNode* N);  
Code* CodeCompression(char c, NodeCode* B);  
char* EncodeTextAVL(char* txt, NodeCode* B);  
Code* DicoAVLToBinary(NodeCode* B);  
char* EncodeDico(NodeCode* B);  
void Compression(char* txt, char *input, int last);
```

decompression.c

Ce fichier contient toutes les fonctions nécessaires à la décompression.

```
Node* CreationHuffmanTreeByDico(const char* dico, int *i, char c, Node *B);
```

```
Node* DecodeDico(const char* dico);
```

```
char HuffmanChar(Node *B, char* binary, int *i);
```

```
char* DecodeBinary(Node *B, char* binary);
```

```
void Decompression(char* bin, char* dico, char* path, int last);
```

fonctions.c

Ce fichier contient des fonctions générales et nécessaire pour plusieurs fichiers.

```
void WriteTxt(char* txt, char* path);
```

```
int get_size(char *filename);
```

```
void ReadTxt(char* txt, char* path);
```

```
char* LetterToBinary(char lettre, char *tab, int length);
```

```
char* WordToBinary(const char* text);
```

```
int Maximum(int a, int b);
```

```
ListNode* Recuperation(ListNode **L);
```

```
ListNode* CopyListNode(ListNode *L);
```

```
char *Concatenation(const char *first, int firstlength, const char *second, int secondlength);
```

Creation.c

Ce fichier contient des fonctions nécessaires pour la création des structures du projet.

```
List* CreationList(char c);  
  
Node* CreationNode(char c, int occu);  
  
ListNode* CreationListNode(char c, int occu);  
  
ListCode* CreationListCode(char c, Code* code);  
  
NodeCode* CreationNodeCode(char c, Code* code);  
  
Code* CreationCode(char c);
```

Suppression.c

Ce fichier contient des fonctions nécessaires pour la suppression des structures du projet.

```
void SuppressionCode(Code **C);  
  
void SuppressionListCode(ListCode **L);  
  
void SuppressionList(List **L);  
  
void SuppressionNode(Node **B);  
  
void SuppressionNodeCode(NodeCode **B);  
  
void SuppressionListNode(ListNode **L);
```

Print_structutres.c

Ce fichier contient des fonctions nécessaires pour l’affichage des structures du projet.

```
void PrintListCode(ListCode *L);  
  
void PrintTabChar(char* tab);  
  
void PrintCode(Code* C);  
  
void PrintList(List* L);  
  
void PrintListNode(ListNode* L);  
  
void PrintABR(Node* N);  
  
void PrintABRCode(NodeCode *B);
```

AVL.c

Ce fichier contient des fonctions nécessaires pour l'équilibrage des AVL.

```
int HeightTree(Node* B);

Node* RotationLeft(Node* B);

Node* RotationRight(Node* D);

Node* RotationRightLeft(Node* B);

Node* RotationLeftRight(Node* F);

Node* Balancing(Node* B);

int HeightTreeCode(NodeCode* B);

NodeCode* RotationLeftCode(NodeCode* B);

NodeCode* RotationRightCode(NodeCode* D);

NodeCode* RotationRightLeftCode(NodeCode* B);

NodeCode* RotationLeftRightCode(NodeCode* F);

NodeCode* BalancingCode(NodeCode* B);
```