Codage de Huffman

Issam Alouane
Jean-Baptiste Prevost

Équipe GH11

Département Sciences du Numérique - Première année
2021-2022

Table des matières

Table des figures

1 Résumé

L'objectif de ce rapport est d'exposer la méthode employée pour répondre a certains problèmes liés a ce projet. Notamment la représentation de l'arbre de Huffman ainsi que sa table, mais aussi comment savoir si le codage de tous les caractères peut se faire en moins d'un octet, la représentation du code des caractères. Pour cela nous allons dans ce voir tout au long de ce rapport la confection et les choix des algorithmes utilisés ainsi que la démarche associée. Pour enfin aboutir à un bilan technique et personnel.

2 Introduction

Ce projet est basé sur le codage d'Huffman qui est utilisé pour compresser des fichiers. Le problème est de pouvoir compresser un fichier a l'aide de ce codage, mais également d'avoir la possibilité de le décompresser. Ce qui sans cette possibilité rendrait la compression inutile.

3 L'architecture de l'application en modules

3.1 Architecture programme compresser

La figure 1 présente l'architecture en module du programme principal de compression de fichier.

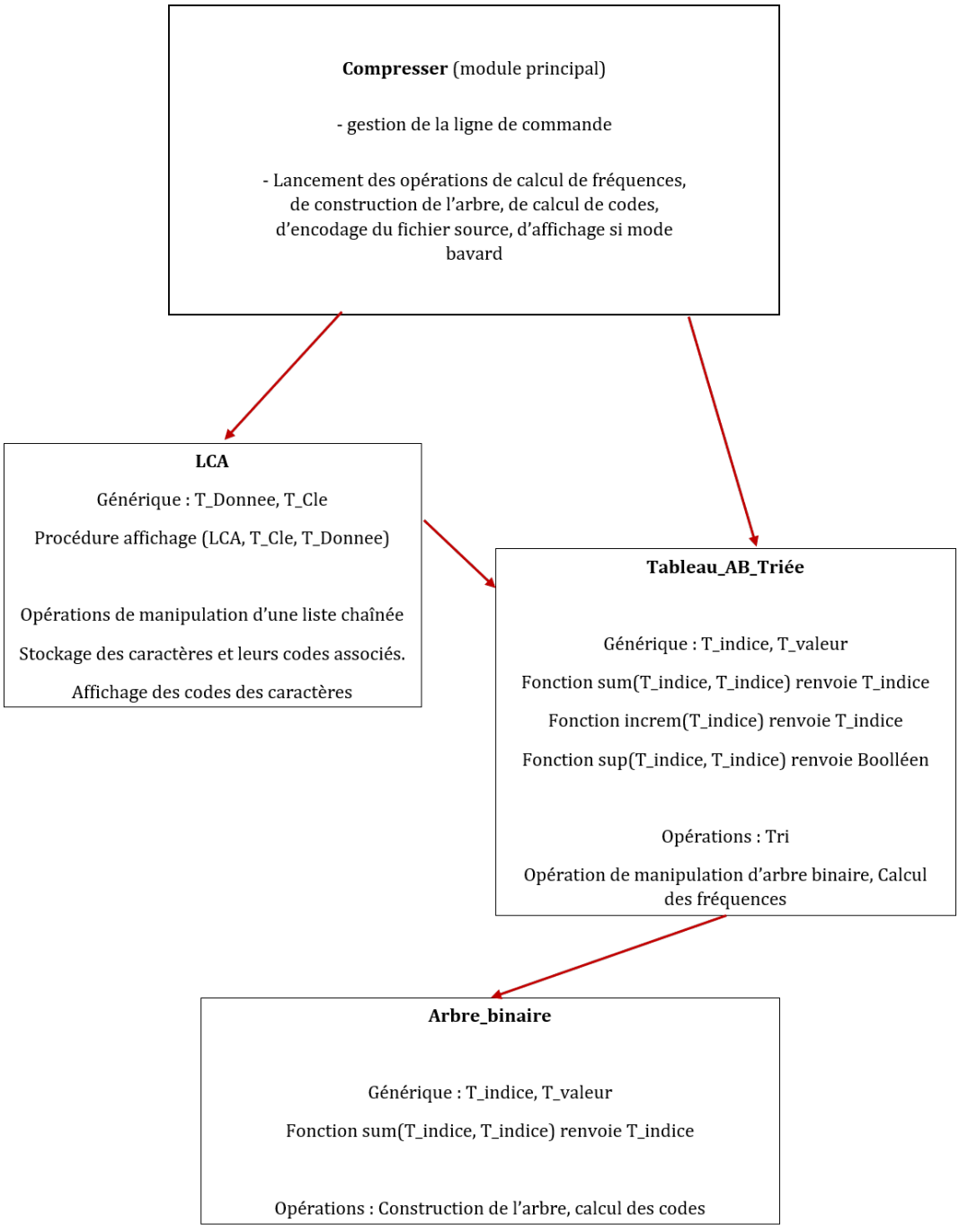


FIGURE 1 – Architecture en module de la compression

3.2 Architecture programme décompresser

La figure 2 présente l'architecture en module du programme principal de ddécompression de fichier.

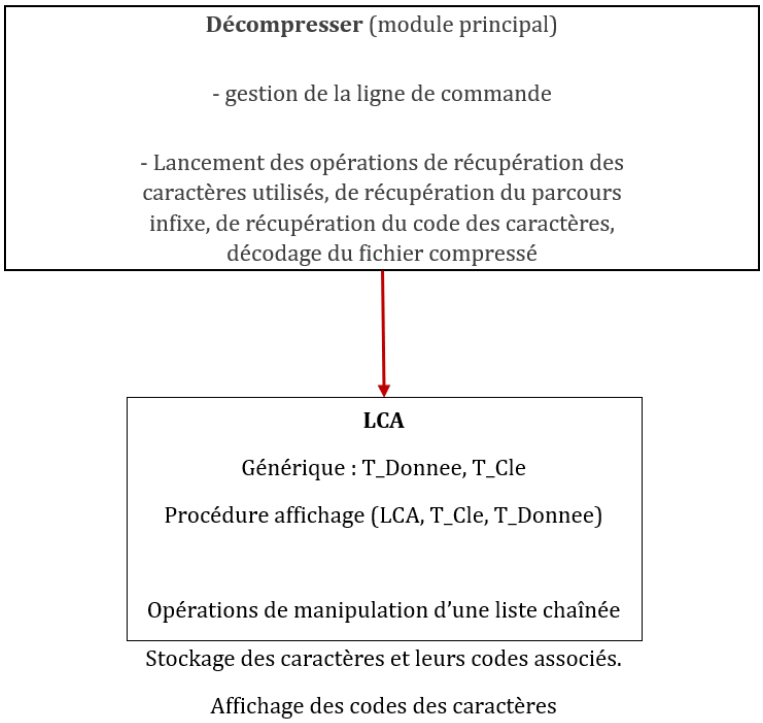


FIGURE 2 – Architecture en module de la décompression

4 Présentation des principaux choix réalisés

- Un module pour gérer les arbres binaires pour pouvoir recréer l'arbre d'Huffman et ainsi pouvoir obtenir les codes des caractères.
- Un module de tableau d'arbre binaire qui va permettre de trier ces arbres et ainsi nous permettre de reconstruire l'arbre d'Huffman. Ce module se servira du module d'arbre binaire. De plus, il pourra nous servir pour stocker la fréquence d'apparition des caractères.
- Un module générique de liste chaînée associative pour nous permettre de stocker les caractères et leurs codes associés. Comme le nombre de caractère possible est assez limité (256 caractères possibles). Le problème de rapidité de lecture d'une LCA n'est pas trop contraignant.
- Pour la décompression, il faut stocker les caractères utilisés dans le codage, nous avons donc utilisé un tableau. Ce qui nous permet de garantir l'obtention de l'ordre d'apparition des caractères lors du parcours infixe de l'arbre.
- Lors de la décompression, nous utiliserons un Unbounded_String pour stocker le parcours infixe qui est stocké dans le fichier compresser, ce qui nous servira pour décoder les symboles du texte d'origine. Un Unbounded_String a l'avantage d'être facilement manipulable, étant donné que la lecture du document se fait octet par octet, il est nécessaire de pouvoir stocker ces octets en allant.

5 Principaux algorithmes et types de données

5.1 Types de données

Arbre :

```
type T_noeud;
```

```
type T_AB is access T_noeud;
type T_noeud is
        record
                Indice : T_indice;
                Valeur : T_valeur;
                Fils_D : T_AB;
                Fils_G : T_AB;
        end record;
```

Tableau :

```
type Tab_AB is array (1..Taille_table) of T_AB;
type T_Tableau is
        record
                Tableau : Tab_AB;
                Taille : integer;
        end record;
```

Liste chainée associative :

```
type T_Cellule;
type T_LCA is access T_Cellule;
type T_Cellule is
        record
                Cle : T_Cle;
                Donnee : T_Donnee;
                Suivant : T_LCA;
        end record;
```

5.2 Algorithmes

- Construction de l'arbre
- Affichage de l'arbre
- Construction de la signature de l'arbre
- Construction des codes des symboles
- Encodage du fichier à compressé
- Récupération de la signature de l'arbre
- Reconstruction des codes
- Décodage du fichier compressé

Construction de l'arbre :

ptr_frequence :in out, arbre_Huffman : out

Tant que taille(ptr_frequence)/=1 faire
        Associer les deux éléments ayant les plus petites fréquences –à savoir le premier élément et le deuxième élément.
        Trier la nouvelle arbre entre les arbres –à savoir le premier élement
Fin Tant Que
arbre_Huffman=ptr_frequence(1)

Affichage de l'arbre :

arbre : in, nombre : in –nombre sera toujours égal à 0

```
Procedure affichage_arbre(arbre, nombre) est
    Si arbre.gauche=null et arbre.droite=null faire
        Ecrire ("(")
        Écrire(arbre.indice)
        Ecrire(")")
        Écrire(" ")
        Écrire(arbre.valeur)
    Sinon
        Ecrire ("(")
        Écrire(arbre.indice)
        Ecrire(")")
        Nouvelle ligne
        Écrire(" - - 0 - - ")
        affichage_arbre(arbre.gauche, nombre + 1)
        Pour i de 1..nombre Faire
            Écrire(" ")
            Nouvelle ligne
        Fin Pour
        Écrire(" - - 1 - - ")
        affichage_arbre(arbre.droite, nombre + 1)
    Fin Si
Fin affichage_arbre
```

Construction de la signature de l'arbre :

```
Arbre : in T_AB      – Arbre d'Huffman
LCA : in out LCA_codage.T_LCA      – Stockage caractère/code
Parcours_Huffman : in out Unbounded_String      – signature de l'arbre
Parcours_caractere : in out Unbounded_String      – code des caractères

Procedure Parcours_infixe(arbre, LCA, Parcours_Huffman, Parcours_caractere) est
Fils_droit(Arbre, fils_D)
Fils_gauche(Arbre, fils_G) ; Si arbre.Gauche /= null alors
    Parcours_Huffman <- Parcours_Huffman & "0"
    Parcours_caractere <- Parcours_caractere & "0"
    Parcours_infixe(fils_G,LCA, Parcours_Huffman, Parcours_caractere)
Sinon
    Enregistrer(lca_code,arbre.Valeur,Parcours_caractere)
Fin Si
Si arbre.Droite /= null alors
```

```
    Parcours_huffman <- parcours_hufman & "1"
    Parcours_caractere <- Parcours_caractere & "1"
    Parcours_infixe(fils_D,LCA, Parcours_Huffman, Parcours_caractere)
Sinon
    Enregistrer(lca_code,arbre.Valeur,Parcours_caractere)

    Fin Si
Fin Procedure Parcours_infixe Parcours_caractere := To_Unbounded_String( To_String ( Par-
cours_caractere)(1 .. Length(Parcours_caractere)-1))
```

Construction des codes des symboles :

La construction de ces codes ce fait dans la construction de la signature

---

Encodage du fichier à compresser :

```
texte_code : Chaine de caractère
j , k : entier

k <- 1
Pour i dans 1..taille(texte) faire
    j <- 1
    Tant que j < Taille(Donnee(lca_code, texte(i))) faire
        texte_code <- texte_code & Donnee(lca_code, texte(i))(j)
        k <- k+1
        Si k % 8 = 0 faire
            texte_code <- texte_code & "."
        Sinon
            Rien
        FinSi
        j <- j + 1
    Fin Tant Que
    Pour i dans 1..(8-k % 8) faire
        texte_code <- texte_code & "0"
    Fin Pour
Fin Pour
```

Récupération de la signature de l'arbre :

```
Octet_Val : Entier In out
Nbr_un, Nbr_zero : Entier In out


Octet_val <- Character'Pos(un_char)
Pour i allant de 1..8 pas de -1 Faire – Conversion de la valeur du caractere en bits
    Si Octet_val >= (2^(i-1)) Et Nbr_un /= (Nbr_zero + 1) Alors
        Octet_val <- Octet_val - *(i-1)
        Parcours_Huffman <- Parcours_Huffman & "1"
        Nbr_un <- Nbr_un + 1
    Sinon Si Octet_val < (2^(i-1)) Et Nbr_un /= (Nbr_zero + 1) Alors
        Parcours_Huffman <- Parcours_Huffman & "0"
        Nbr_zero <- Nbr_zero + 1
    Sinon
        Rien
    Fin Si
Fin Pour
```

Reconstruction des codes :

```
parcours_huffman : in, indice_fin : in –la position de symbole de fin
Tab_caractere : in –le tableau des caractères
Code_caractere : Chaine de caractère
Initialiser(lca_code) – Procedure du module lca

j : Entier <- 1 –paramètre de parcours du tableau qui contient les caractères
Pour i de 1..length(parcours_huffman) faire
    Code_caractere <- Code_caractere & Parcours_Huffman(i)
    Si Parcours_Huffman(i+1) = '1' faire
        Si j=indice_fin faire
            Enregistrer(lca_code, Code_caractere, "$")
        Sinon
            Enregistrer(lca_code, Code_caractere, Tab_caractere(j))
        Fin Si
        Code_caractere <- Code_caractere(1,Length(Code_caractere) - j)
        j <- j + 1
        Tant que Parcours_caractere (length(Parcours_caractere)) /= '0' faire
            Parcours_caractere <- Parcours_caractere(1..length(Parcours_caractere)-1) ;
        Fin Tant Que;
        Parcours_caractere <- Parcours_caractere (1..length(Parcours_caractere)-1) ;
    Sinon
        Rien
    Fin Si
Fin Pour
```

Décodage du fichier compressé :

Texte_original : in out Chaine de caractere
Code_bin : in out Chaine de caractère

```
indice <- 1
Tant Que Texte_original(-1) /= "$" faire
    Code_bin <- Code_bin & Texte_encode(indice)
    Si Cle_presente(lca_code, Code_bin) faire
            Texte_original <- Texte_original & La_Donnee(lca_code,Code_bin)
            Code_bin <- ""
        Sinon
            Rien
        Fin Si
        indice <- indice + 1
Fin Tant Que
Générer le fichier décompressé
```

6 Démarche adoptée pour tester le programme

Dans un premier temps, nous avons testé chaque module avec des programmes dédié. Cela nous a permis de détecter certaines erreurs avant que celles-ci ne gênent pas les programmes principaux. Après que les modules aient été testés séparément nous avons fait des tests pour chaque fonction principale qui venait d'être achevée en utilisant le même fichier que celui du sujet. Lorsque les programmes principaux ont été terminés, nous avons testé sur des cas limite. Par exemple avec des arbres profond.

7 Difficultés rencontrées et solutions adoptées

Dans un premier temps, les difficultés étaient dues à la généricité des modules que nous souhaitions implanter. Pour pallier ce problème, nous avons décidé de n'avoir qu'un seul module générique sur les trois existants pour faire les programmes principaux. Une fois ces programmes principaux fini nous avons rendu générique les deux modules qui ne l'étaient pas.
La récupération du parcours infixe nous a posés quelques problèmes, car nous n'obtenions pas le bon parcours infixe. Ce problème a été résolu en créant une fonction de complétion d'octet dans le programme de compression. En effet, le problème survenait lorsque le dernier octet du parcours n'était pas complet, la procédure complète donc les bits manquants.
De plus, la récupération des caractères était une des difficultés rencontrés, pour pallier ce problème nous avons mis en place un slicing de la trame de bits de la signature de l'arbre.
Un autre problème était celui des fuites de mémoire, qui était du a notre procédure de suppression du tableau d'arbre binaire. Ce problème de fuite de mémoire venait du module Tableau dans la procédure d'enregistrement et d'incrémentation en modifiant ces structures et en ne vidant pas l'arbre binaire seul, le problème a disparu.

8 Organisation de l'équipe

Issam Alouane :

- Tri du tableau d'arbre binaire
- Création de l'arbre de Huffman
- Affichage de l'arbre de Huffman
- Encodage du texte et création du fichier compressé
- Reconstruction des codes associés aux caractères a partir de la signature de l'arbre
- Décodage du texte et création du fichier décompressé

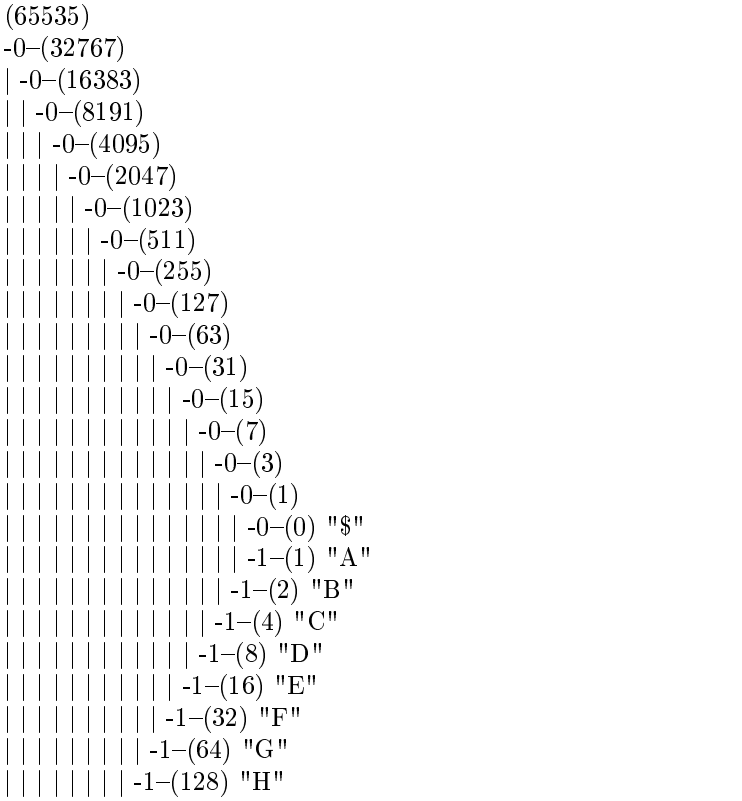
Jean-Baptiste Prevost :

- Calcul des fréquences d'apparition des caractères
- Création du parcours infixe
- Création des codes associés aux caractères
- Affichage de la table de Huffman
- Traitement des exceptions
- Récupération du parcours infixe
- Récupération de la signature de l'arbre
- Récupération de la trame a décoder

9 Bilan technique donnant un état d'avancement du projet

Compression complète :

Compression du fichier fic_huffman_profond16.txt



| | | | | | | -1–(256) "I"
| | | | | | -1–(512) "J"
| | | | | -1–(1024) "K"
| | | | -1–(2048) "L"
| | | -1–(4096) "M"
| | -1–(8192) "N"
| -1–(16384) "O"
-1–(32768) "P"

'$' –> 0000000000000000
'A' –> 0000000000000001
'B' –> 000000000000001
'C' –> 00000000000001
'D' –> 0000000000001
'E' –> 000000000001
'F' –> 00000000001
'G' –> 0000000001
'H' –> 000000001
'I' –> 00000001
'J' –> 0000001
'K' –> 000001
'L' –> 00001
'M' –> 0001
'N' –> 001
'O' –> 01
'P' –> 1

Comme on peut le remarquer les caractères ne sont pas forcément écrits sur un seul octet, mais cela ne doit pas poser de problème lors de la décompression. De plus, il sera toujours possible de donner la position du caractère de fin car il ne peut pas se trouver à la fin du parcours infixe, et ce même s'il existe 256 caractères différents.

jprevost@n7-ens-lnx017 : /Bureau/1A/pim/projet/GH11/src$ valgrind ./compression fic_huffman_profond16.txt
==30215== Memcheck, a memory error detector
==30215== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==30215== Using Valgrind-3.15.0 and LibVEX ; rerun with -h for copyright info
==30215== Command : ./compression fic_huffman_profond16.txt
==30215==
==30215==
==30215== HEAP SUMMARY :
==30215== in use at exit : 0 bytes in 0 blocks
==30215== total heap usage : 262,399 allocs, 262,399 frees, 9,474,680 bytes allocated
==30215==
==30215== All heap blocks were freed – no leaks are possible
==30215==
==30215== For lists of detected and suppressed errors, rerun with : -s
==30215== ERROR SUMMARY : 0 errors from 0 contexts (suppressed : 0 from 0)

Décompression complète :

jprevost@n7-ens-lnx017 : /Bureau/1A/pim/projet/GH11/src$ valgrind –leak-check=full ./decompression fic_huffman_profond16.txt.hff

```
==32916== Memcheck, a memory error detector
==32916== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==32916== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==32916== Command : ./decompression fic_huffman_profond16.txt.hff
==32916==
==32916==
==32916== HEAP SUMMARY :
==32916== in use at exit : 0 bytes in 0 blocks
==32916== total heap usage : 147,618 allocs, 147,618 frees, 8,597,290,640 bytes allocated
==32916==
==32916== All heap blocks were freed – no leaks are possible
==32916==
==32916== For lists of detected and suppressed errors, rerun with : -s
==32916== ERROR SUMMARY : 0 errors from 0 contexts (suppressed : 0 from 0)
```

10 Perspectives d'amélioration

— L'affichage n'est pas exactement ce qui est demandé en énoncé.
— Améliorer le programme de décompression afin d'optimiser le temps d'exécution.

11 Bilan personnel et individuel

Issam Alouane :

Personnellement, j'ai trouvé ce projet intéressant puisqu'il m'a poussé à faire de mon mieux devant quelques situations que je n'imaginai pas. Ce projet, paraît difficile, mais lorsqu'on commence tout s'enchaîne bien. L'une des situations difficiles que je n'ai pas oubliées est celle de la récupération des codes caractères, je me suis dit : "comment pourrai-je faire ça?". Après analyse de ma solution informelle et de l'implantation, j'ai conclu que ma solution n'était pas la bonne. J'ai donc continué à modifier jusqu'à ce que j'aboutisse au résultat voulu. J'ai aussi appris pendant ce projet l'importance d'être flexible, une idée qui ne marche pas doit être remplacé le plus rapidement possible pour ne pas perdre trop de temps.

Jean-Baptiste Prevost :

Pour ma part, ce fut un projet intéressant, car traitant d'un sujet qui, je pense est important dans un monde où les données sont de plus en plus volumineuses. Pour pouvoir aboutir à un résultat satisfaisant, j'ai passé entre 65 et 90h dessus. La partie la plus chronophage doit être l'implantation, car c'est à ce moment-la que je me suis rendu compte que certaines choses seront moins évidentes que d'autre à mettre en place avec les modules créée. La mise au point est une parti qui m'a pris un peu de temps, mais moins que l'implantation. Quant au rapport, celui m'a pris environ 6h à mettre au point.
De ce projet, je retiens que la nécessité de bien géré les modules dès le départ surtout s'ils utilisent des pointeurs, car si l'on continue avec des modules non-fiables, on peut se retrouver dans des situations inextricables.

12 Modifications faites par rapport aux premiers livrables

Une grande partie du raffinage a été modifiée, dans un but de simplification et d'optimisation, mais aussi parce que l'algorithme ne faisait pas son rôle correctement. Parmi ces algorithmes : la

construction de l'arbre a été simplifiée au maximum, le parcours infixe, l'encodage (au début on travaillait en modulo 9 alors qu'il fallait travailler en modulo 8). De plus, la partie de reconstruction des codes à partir du parcours infixe a été modifié car elle ne faisait pas son rôle.