Polytech'Marseille
Case 925 – 163, avenue de Luminy
13288 Marseille cedex 9

Informatique – 3ème année

Rapport du projet d'algorithmique et de programmation

Compression de données avec LZW

(Lempel-Ziv-Welch)

Delphine BURNOTTE
Yohann DUFOUR

Tuteur : Nicolas DURAND

2012-2013

Table des matières

Introduction

La compression LZW (Lempel-Ziv-Welch) est actuellement la méthode de compression qui offre le meilleur pourcentage de réduction d'un fichier, sans aucune destruction de données. C'est à dire qu'il n'y a aucune perte lorsqu'on reconstitue les données initiales à partir des données compressées.

La compression LZW consiste, en fait, à éviter les répétitions dans les chaînes de caractères du fichier à compresser, pour économiser de la place.

L'ancêtre de cette méthode était la méthode LZ, découverte en 1978 par Lempel et Ziv. Elle fût complétée par Welch en 1984.

L'avantage de cette compression est qu'elle utilise un *dictionnaire* qui se construit de façon dynamique, au cours de la compression mais aussi de la décompression. En effet, le dictionnaire n'est pas stocké dans le fichier compressé et la compression s'effectue en une seule lecture. La méthode étant basée sur les répétitions de caractères, elle est donc plus efficace sur des gros fichiers.

Malheureusement, ce type de compression n'est pas l'un des plus utilisés à cause du brevet déposé par la société américaine Unisys, qui empêcha le développement de cette méthode (bien qu'il ait expiré il y a quelques années).

Notre travail concernant ce projet est de créer un programme qui réalise la compression LZW sur un fichier texte.
Nous devons alors principalement réaliser deux algorithmes, un de compression et un de décompression. Pour cette réalisation, il est nécessaire d'élaborer des fonctions, ainsi que des structures de données afin d'implémenter une gestion de dictionnaire. Enfin, nous devons créer des fonctions pour lire et écrire, dans le fichier de sortie, les codes en binaire.

1. Algorithme de compression LZW

1.1 Principe

Des suites de caractères lus sont ajoutées au fur et à mesure dans un dictionnaire, et sont associées à un code de compression.

A chaque nouveau caractère lu, on regarde dans le dictionnaire si la chaîne formée par la concaténation du tampon et du caractère lu existe déjà. Si c'est le cas, on remplace le tampon par cette nouvelle chaîne de caractères, sinon on rajoute cette nouvelle chaîne de caractères à la fin du dictionnaire, puis on écrit son code dans le fichier de sortie et on réinitialise le tampon de lecture par le dernier caractère lu.

A noter que les nouveaux codes des chaînes du dictionnaire commenceront à 260, car le dictionnaire est initialisé avec les caractères de la table ASCII.

1.2 Pseudo-code

```
Fonction compression (dictionnaire, fichier_lecture, fichier_ecriture) :
     tampon ← ""
     Tant que fichier_lecture contient des caractères à lire :
            caractere_lu ← lire le caractère suivant de fichier_lecture

            Si tampon+caractere_lu existe dans le dictionnaire :
                   tampon ← tampon+caractere_lu
            Sinon :
                   Ajouter le mot tampon+caractere_lu au dictionnaire
                   Ecrire le code de tampon dans fichier_ecriture
                   tampon ← caractere_lu
            Fin Si
     Fin Tant que
     Ecrire le code de tampon dans fichier_ecriture
Fin fonction
```

1.3 Exemple

Faisons un essai avec la chaîne ‘‘cocorico’’ :

Etape	Tampon	Caractère lu	Code émis	Ajout au dictionnaire
0		c		
1	c	o	99	co -> 260
2	o	c	111	oc -> 261
3	c	o		
4	co	r	260	cor -> 262
5	r	i	114	ri ->263
6	i	c	105	ic -> 264
7	c	o	260	

On lit le premier caractère : c, il est déjà dans la table donc on mémorise ce caractère dans le tampon.

On lit le deuxième caractère : o, la concaténation du tampon avec le caractère lu : co n’est pas dans le dictionnaire, donc on rajoute co au code 260. On écrit dans le fichier de sortie le code de c : 99. On réinitialise le tampon avec le dernier caractère lu : o.

On lit le troisième caractère: c, la concaténation du tampon avec le caractère lu : oc n’est pas dans le dictionnaire, donc on rajoute oc au code 261. On écrit dans le fichier de sortie le code de o : 111. On réinitialise le tampon avec le dernier caractère lu : c.

On lit le quatrième caractère : o, la concaténation du tampon avec le caractère lu : co est dans le dictionnaire, donc on mémorise co dans le tampon. Etc.

Les codes de compression obtenus sont 99 111 260 114 105 260.

2. Algorithme de décompression LZW

2.1 Principe

L'algorithme de décompression a l'avantage d'avoir seulement besoin du texte compressé en entrée. En fait, il reconstruit le dictionnaire à mesure qu'il décompresse le fichier. Le processus de décompression suit donc les mêmes règles que pour la compression mais en partant du code compressé.

On lit le premier code que l'on décompresse facilement (on sait que son code correspond à une entrée de la table ASCII). Ensuite, à chaque code lu, soit le code appartient au dictionnaire et on décode en allant chercher le mot dans le dictionnaire, soit le code n'appartient pas au dictionnaire et on construit le mot décodé en raisonnant : si le code n'appartient pas au dictionnaire, c'est que lors de la compression, le code venait juste d'être ajouté dans le dictionnaire, on sait donc que le mot ajouté correspondait au mot précédent (le tampon) concaténé avec le premier caractère de la chaîne que l'on cherche à décoder (la chaîne que l'on cherche à décoder commence par le mot précédent, c'est donc le premier caractère du mot précédent).

Pour finir, on écrit le mot correspondant dans le fichier de sortie, on l'ajoute au dictionnaire et on réinitialise le mot précédent par le mot qui vient d'être décodé.

2.2 Pseudo-code

```
Fonction decompression (dictionnaire, fichier_lecture, fichier_ecriture) :
      code_lu ← lire le premier code de fichier_lecture
      mot_precedent ← code_lu
      Ecrire mot_precedent dans fichier_ecriture

      Tant que fichier_lecture contient des codes à lire :
            code_lu ← lire le code suivant de fichier_lecture
            Si code_lu existe dans le dictionnaire :
                  mot_decode ← rechercher le mot correspondant à code_lu
            Sinon :
                  mot_decode ← mot_precedent + mot_precedent[0]
            Fin Si
            Ecrire mot_decode dans fichier_ecriture
            Ajouter le mot mot_precedent+mot_decode[0] au dictionnaire
            mot_precedent ← mot_decode
      Fin Tant que
Fin Fonction
```

2.3 Exemples

Reprenons le même exemple.

Etape	Code lu	Mot précédent	Mot décodé	Ajout certain au dictionnaire
0	99		c	
1	111	c	o	co -> 260
2	260	o	co	oc -> 261
3	114	co	r	cor -> 262
4	105	r	i	ri ->263
5	260	i	co	ic -> 264

On lit le premier code : 99, on mémorise son décodage et on l'écrit dans le fichier de sortie : c.

On lit le deuxième code : 111, il est dans le dictionnaire, donc on mémorise son décodage et on l'écrit dans le fichier de sortie : o. On rajoute au dictionnaire le mot précédent concaténé avec le premier caractère du mot décodé : co au code 260.

On lit le troisième code : 260, il est dans le dictionnaire, donc on mémorise son décodage et on l'écrit dans le fichier de sortie : co. On rajoute au dictionnaire le mot précédent concaténé avec le premier caractère du mot décodé : oc au code 261. Etc.

La chaîne de caractères initiale est c o co r i co.

Prenons maintenant le cas où la chaîne est encodée avec une valeur associée à aucune chaîne au moment de la décompression : abababab dont les codes de compression sont 97 98 260 262 98.

Etape	Code lu	Mot précédent	Mot décodé	Ajout certain au dictionnaire
0	97		a	
1	98	a	b	ab -> 260
2	260	a	ab	ba -> 261
3	262	ab	aba	aba -> 262
4	98	b	b	

On lit le premier code : 97, on mémorise son décodage et on l'écrit dans le fichier de sortie : a.

On lit le deuxième code : 98, il est dans le dictionnaire, donc on mémorise son décodage et on l'écrit dans le fichier de sortie : b. On rajoute au dictionnaire le mot précédent concaténé avec le premier caractère du mot décodé : ab au code 260.

On lit le troisième code : 260, il est dans le dictionnaire, donc on mémorise son décodage et on l'écrit dans le fichier de sortie : ab. On rajoute au dictionnaire le mot précédent concaténé avec le premier caractère du mot décodé : ba au code 261.

On lit le quatrième code : 262, il n'est pas dans le dictionnaire, on décode le mot en prenant la dernière chaîne décodée à laquelle on ajoute la première lettre de cette même chaîne. Enfin on l'écrit dans le fichier de sortie : aba.

On lit le dernier code : 98, il est dans le dictionnaire, alors on l'écrit dans le fichier de sortie : b.

La chaîne de caractères initiale est a b ab aba b.

3. Gestion du dictionnaire

Comme nous l'avons déjà dit, les algorithmes de compression et de décompression LZW utilisent un dictionnaire. Ce dictionnaire sera représenté par un arbre lexicographique car les opérations faites sur les arbres (comme l'insertion ou la recherche) sont très efficaces.

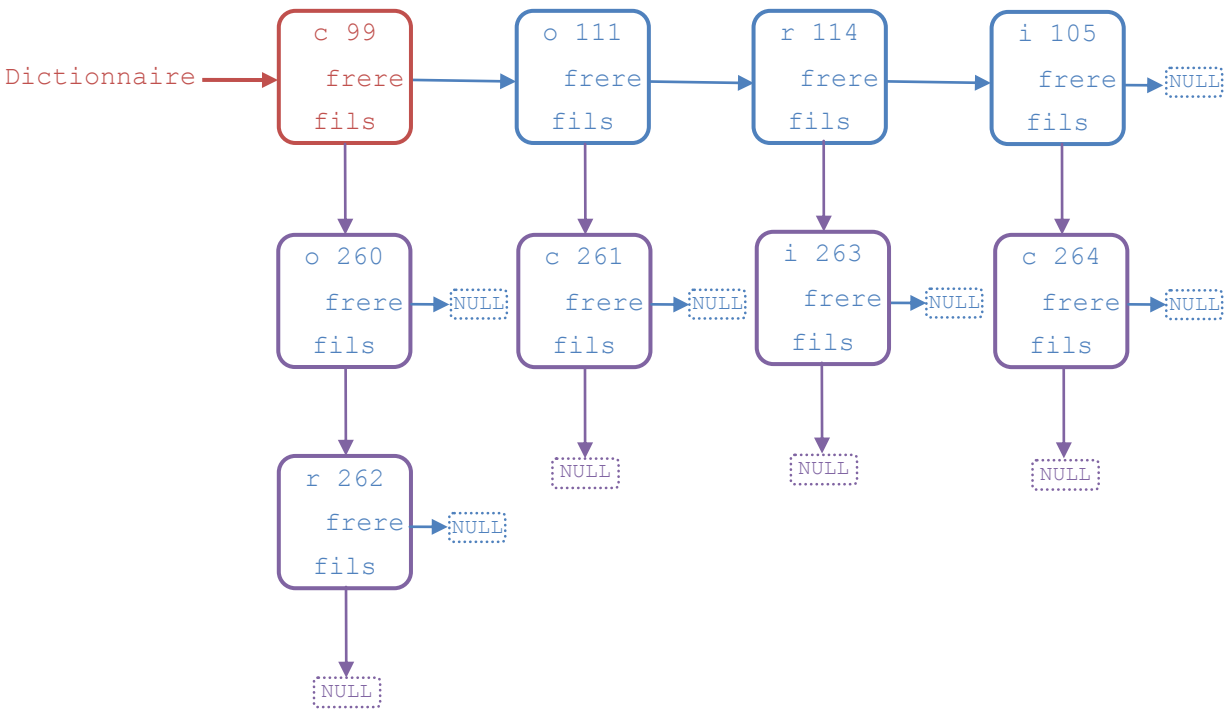
3.1 Représentation de la structure de données

L'arbre lexicographique utilisé pour notre algorithme est un dérivé de la représentation classique des arbres lexicographiques. En effet, dans le cadre de la compression LZW, on peut remarquer que si un mot est ajouté au dictionnaire, cela signifie nécessairement que l'ensemble des préfixes de ce mot a déjà été ajouté au dictionnaire. Ainsi, le caractère de fin de chaîne `'\0'` devient inutile car chaque sous-chaîne d'un mot est forcément un mot du dictionnaire.
En partant de cette constatation, la représentation classique de l'arbre lexicographique peut être optimisée afin de réduire la mémoire utilisée par l'algorithme. Cette optimisation consiste à supprimer le dernier maillon de chaque chaîne de caractère (le maillon contenant le caractère `'\0'` et le code de compression associé au mot) et à stocker le code de compression de chaque mot dans le maillon de la dernière lettre du mot.

Chaque nœud du dictionnaire dispose d'un lien pour passer à son fils et d'un lien pour passer à son frère. Le dictionnaire sera représenté par la racine de l'arbre, et pointera donc sur le premier caractère de l'arbre.

De plus, chaque nœud contiendra deux informations : un caractère et le code de compression de la chaîne de caractères définie par la concaténation (ordonnée de la racine vers le nœud en question) des caractères des nœuds pour arriver à ce nœud.
Nous utiliserons typiquement le pointeur `NULL` pour indiquer la fin horizontale (le frère pointe sur `NULL`) ou verticale (le fils pointe sur `NULL`) d'une branche de l'arbre.

Schéma : Par exemple, pour un dictionnaire qui contient les mots : c, o, r, i, co, oc, cor, ri, ic



3.2 Fonctions primaires

Dans la suite de cette partie, nous allons considérer les fonctions primaires de gestion du dictionnaire suivantes :

- `est_vide (dictionnaire)` renvoie si `dictionnaire` est vide ou non.
- `creer_nœud (caractere, fils, frere, init)` renvoie le nœud ayant `fils` comme fils, `frere` comme frère, `caractere` comme caractère (le code de compression est attribué indépendamment). Le paramètre `init` est un booléen qui, lorsqu'il vaut 1, réinitialise la valeur des codes attribués.
- `fils (nœud)` renvoie le lien du fils de `nœud`.
- `frere (nœud)` renvoie le lien du frère de `nœud`.
- `caractere (nœud)` renvoie le caractère contenu dans le `nœud`.
- `code (nœud)` renvoie le code de compression associé à la chaîne de caractères définie par la concaténation (ordonnée de la racine vers `nœud`) des caractères des nœuds pour arriver à `nœud`.
- `initialiser_dictionnaire ()` renvoie l'arbre représentant le dictionnaire initialisé par l'ensemble des caractères de la table ASCII.
- `modifier_caractere (nœud, caractere)` modifie le caractère du nœud.
- `modifier_fils (nœud, fils)` modifie le fils du nœud.
- `modifier_frere (nœud, frere)` modifie le frère du nœud.
- `détruire_dictionnaire (dictionnaire)` détruit le dictionnaire.

Il faut noter que la variable `arbre` ou la variable `nœud` sont, en mémoire, de même type, seul leur interprétation au niveau sémantique est différente. Cependant, notre représentation de la structure de données fait qu'un nœud a la même représentation que le sous arbre considéré à partir de ce nœud. Par souci de simplicité, on pourra donc les confondre dans l'écriture des algorithmes. Il en sera de même pour une feuille, qui est le cas particulier d'un nœud sans fils et sans frère.

3.3 Affichage du dictionnaire

Un parcours en profondeur de l'arbre permet de parcourir tous les nœuds de l'arbre et de lire tous les caractères des mots du dictionnaire. Cependant, chaque nœud contient un seul caractère, il faut donc retenir la chaîne déjà parcourue récursivement afin de l'afficher lorsque le parcours se termine et que le nœud est une feuille de l'arbre. La fonction récursive stocke donc au fur et à mesure la chaîne parcourue dans la variable `chaine`.

```
Fonction afficher_dictionnaire (dictionnaire, afficher_table_ascii) :
    chaine ← ""
    parcours (dictionnaire, chaine, afficher_table_ascii)
Fin Fonction

Fonction parcours (nœud, chaine, afficher_table_ascii) :
    Copier chaine dans copie
    Ajouter caractere (nœud) dans chaine

    Si (afficher_table_ascii ou (non afficher_table_ascii et code (nœud) > 255)) :
          Afficher chaine et code (nœud)
    Fin Si

    Si nœud a un fils :
          parcours (fils (nœud), chaine, afficher_table_ascii)
    Fin Si

    Si nœud a un frere :
          parcours (frere (nœud), chaine, afficher_table_ascii)
    Fin Si
Fin Fonction
```

Exemple :

Si l'on affiche le dictionnaire qui correspond à la compression (et décompression) de `"cocorico"` :

```
AFFICHAGE DU DICTIONNAIRE
co 260
co\n 265
cor 262
ic 264
oc 261
ri 263
```

3.4 Recherche d'un code dans le dictionnaire à partir du mot

La recherche du code correspondant à une chaîne de caractères donnée peut se faire récursivement.

En effet, pour un nœud donné, on regarde si le premier caractère de `chaine` appartient au nœud ou à ses frères. Si ce n'est pas le cas, cela signifie que le mot n'existe pas dans le dictionnaire. Si ce n'est pas le dernier caractère à rechercher, on rappelle récursivement la fonction sur le fils du nœud contenant le caractère trouvé et sur la sous-chaîne de caractères privée de son premier caractère.

La fonction récursive possède deux conditions d'arrêt : soit le dictionnaire est vide et on renvoie le code signifiant qu'aucune chaîne n'a été trouvé, soit la chaîne de caractères ne contient qu'un seul caractère qui est celui du nœud et on renvoie le code du nœud actuel.

```
Fonction rechercher_code (dictionnaire, chaine) :
    nœud ← dictionnaire

    Si dictionnaire est vide :                      /* Cas 1 */
          Retourner 0
    Sinon Si caractere (nœud) = chaine[0] :         /* Cas 2 */
          Si longueur (chaine) = 1 :
                Retourner code (nœud)
          Sinon :
                Supprimer le caractère 0 de chaine
                Retourner rechercher_code (fils (nœud), chaine))
          Fin Si
    Sinon :                                         /* Cas 3 */
          Si caractere (nœud) > chaine[0] :
                Retourner 0
          Sinon :
                Retourner rechercher_code (frere (nœud), chaine)
          Fin Si
    Fin Si
Fin Fonction
```

Exemple :

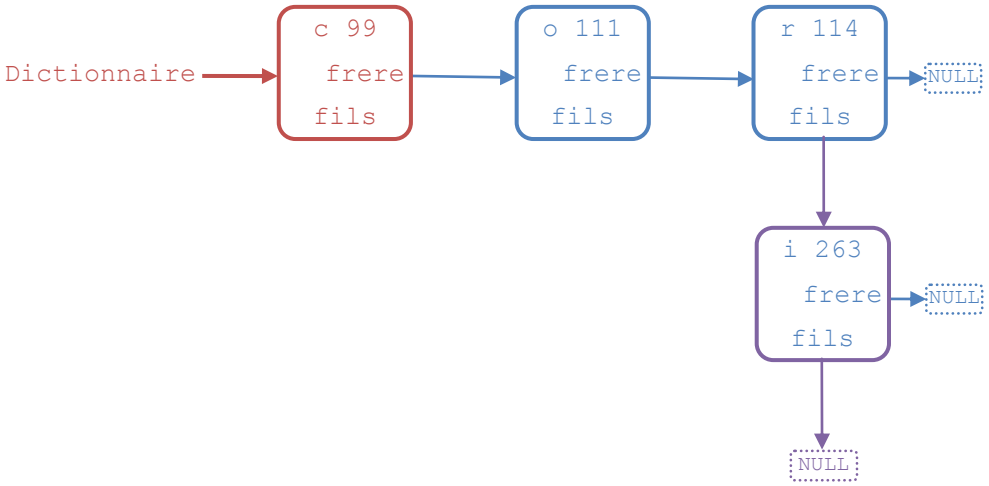
A partir du dictionnaire de la compression de "cocorico", si l'on recherche le code correspondant à "ri".

Appel 1 : On est dans le cas 3 et 'r' > 'c' donc on passe à l'appel 2.

Appel 2 : On est encore dans le cas 3 et 'r' > 'o', on passe à l'appel 3.

Appel 3 : On est dans le cas 2 et 'r' = 'r', mais la longueur ("ri") = 2, donc on supprime 'r' de "ri" et on recherche "i" dans le fils de 'r'.

Appel 4 : On est dans le cas 2, 'i' = 'i' et longueur ("i") = 1, donc on retourne le code correspondant : 263.



3.5 Recherche d'un mot dans le dictionnaire à partir du code

La recherche d'un mot dans le dictionnaire se fait à l'aide d'un parcours récursif du dictionnaire. En effet, on appelle récursivement la fonction sur les fils du nœud et sur les frères du nœud afin de trouver le code recherché.

Dès que l'on trouve le code, on renvoie le caractère du nœud correspondant et on reconstitue la chaîne de caractère entière en remontant la récursion, en rajoutant en position 0 le caractère de chaque nœud.

La condition d'arrêt de la fonction récursive est si le code du nœud est égal au code recherché. On retourne alors la chaîne de caractères contenue dans fin_mot.

```
Fonction rechercher_mot (dictionnaire, code_mot) :
    nœud ← dictionnaire

    Si code (nœud) = code_mot :                                    /* Cas 1 */
          fin_mot ← "caractere (nœud)"
          Retourner fin_mot

    Sinon :                                                        /* Cas 2 */
          chaine ← NULL

          Si fils (nœud) n'est pas vide :                          /* Sous-cas 1 */
                chaine ← rechercher_mot (fils (nœud), code_mot)
          Fin Si

          Si chaine = NULL et frere (nœud) n'est pas vide :        /* Sous-cas 2 */
                Retourner rechercher_mot (frere (nœud), code_mot)
          Fin Si

          Si chaine != NULL :                                      /* Sous-cas 3 */
                Insérer à la position 0 de chaine : caractere (nœud)
          Fin Si

          Retourner chaine
    Fin Si
Fin Fonction
```

Exemple :

A partir du dictionnaire de la compression de "cocorico", on recherche la chaîne de caractère correspondante au code 263.

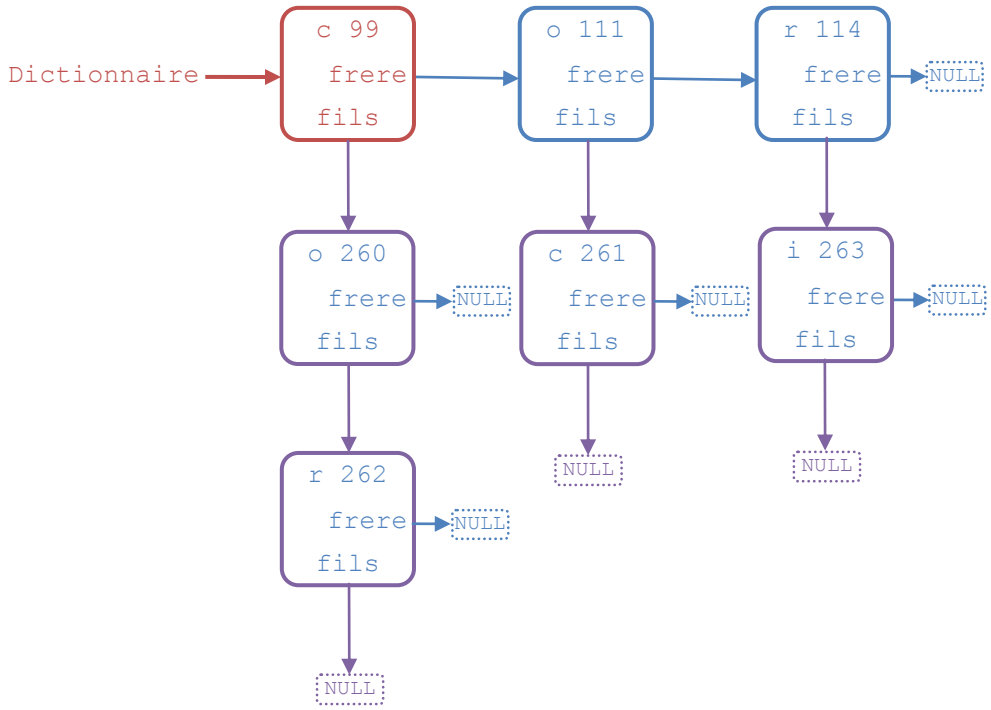
Appel 1 : Nœud qui contient le code 99 : on est dans le cas 2 et le fils du nœud n'est pas vide (sous-cas 1).

Appel 2 : Nœud qui contient le code 260 : on est dans le cas 2 et le fils du nœud n'est pas pas vide (sous-cas 1).

Appel 3 : Nœud qui contient le code 262 : on est dans le cas 2 et le fils du nœud est vide donc on passe au sous-cas 2, or le frère du nœud est vide et la chaîne est nulle. On arrive alors au sous-cas 3, mais la chaine est nulle. On remonte au nœud précédent (nœud de code 260), puis par le même cheminement, on remonte au nœud encore d'avant, et on est revenu au tout premier nœud (nœud de code 99). Ici le frère de ce nœud n'est pas vide.

On parcourt l'arbre de cette façon jusqu'au nœud qui contient le code 263 (atteint, ici, à l'appel 7).

Appel 7 : Nœud qui contient le code 263 : on est dans le cas 1, le code du nœud correspond au code cherché. Donc on met 'i' dans fin_mot. On remonte au nœud précédent qui contient 'r' et on l'ajoute devant 'i' dans fin_mot.



3.6 Ajout d'un mot au dictionnaire

L'ajout d'un mot au dictionnaire peut être fait de manière récursif. A chaque appel de la fonction, on cherche à réduire la taille de la chaîne de caractères à ajouter d'un caractère. Ainsi, dans le cas général, on regarde si le premier caractère de la chaîne est présent dans les frères de la racine de l'arbre. Si c'est le cas, alors on rappelle récursivement sur le fils de ce frère, sinon on le crée (création du frère avec comme caractère le premier caractère de la chaîne) avant de lancer l'appel récursif sur ce nouveau frère créé en retirant le premier caractère (maintenant traité) de la chaîne.

On distingue ensuite le cas où le dictionnaire, donné en paramètre, est vide. Dans ce cas, nullement besoin de lancer la moindre recherche, il suffit de créer en cascade (de fils en fils) des fils auxquels on attribue un caractère de `chaine` et un code de compression par fils. Ce cas particulier constitue un des deux cas d'arrêt de notre fonction récursive.

Le deuxième cas d'arrêt est plus typique, il s'agit d'arrêter la récursion lorsque la fonction n'a plus de caractères à ajouter au dictionnaire, c'est-à-dire lorsque la chaîne de caractères placée en paramètre est vide.

```
Fonction ajouter_mot (dictionnaire, chaine) :
    Si chaine est vide :
          Retourner dictionnaire

    Sinon Si le dictionnaire est vide:
          Retourner creer_fils_cascade(chaine)
    Sinon :
          nœud ← dictionnaire
```

```
            Si caractere (nœud) > chaine[0] :
                  sous_chaine ← chaine – chaine[0]
                  nœud ← creer_nœud (chaine[0], creer_fils_cascade (sous_chaine), nœud, 0)
chaine[0])
            Sinon Si caractere (nœud) < chaine[0]:
                  frere (nœud) ← ajouter_mot (frere (nœud), chaine)
            Sinon :
                  sous_chaine ← chaine – chaine[0]
                  fils (nœud) ← ajouter_mot (fils (nœud), sous_chaine)
            Fin Si

            Retourner nœud
      Fin Si
Fin Fonction
```

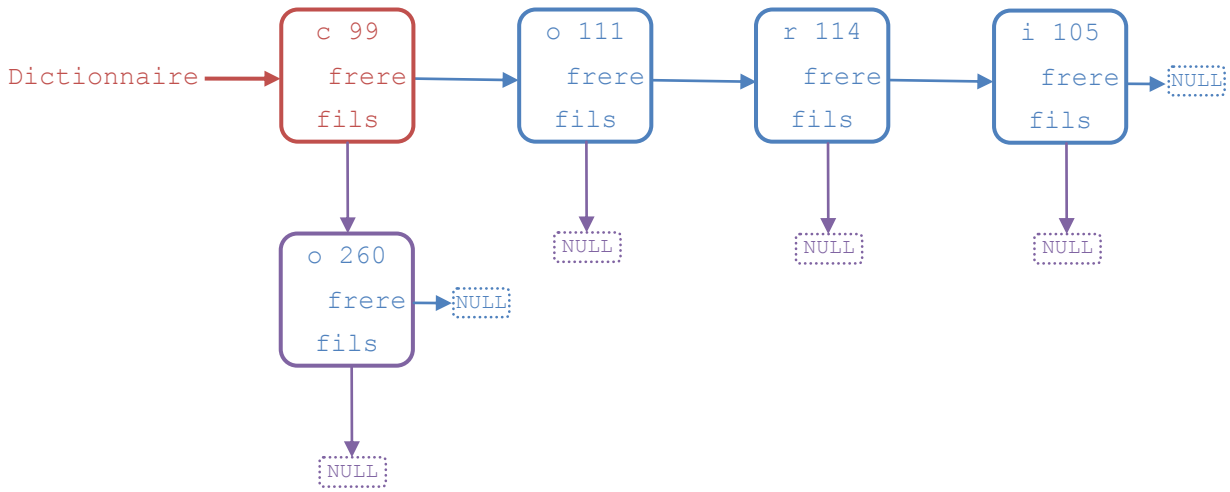
Afin de créer des nœuds fils en cascade, la solution qui semble être la plus efficace est celle qui consiste à ajouter les fils un à un (un pour chaque caractère de `chaine`) en partant de la fin du mot `chaine`. En effet, cela permet de connaitre à l'avance le fils du prochain nœud (on vient de créer son fils à l'itération précédente) lorsqu'on créé ce nœud.

```
Fonction creer_fils_cascade (chaine) :
    nœud ← NULL
    n ← longueur (chaine)-1

    Tant que n >= 0 :
          nœud ← creer_nœud (chaine[n], nœud, NULL,0)
          Décrémenter n
    Fin Tant que

    Retourner nœud
Fin Fonction
```

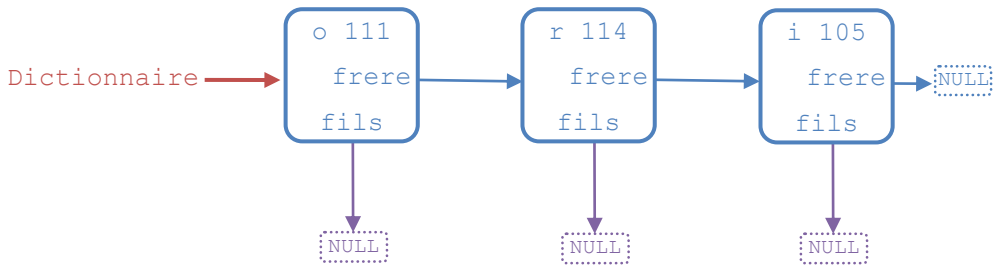
Exemple :

A partir du dictionnaire en cours de la compression de "cocorico", on ajoute la chaîne "oc".

Appel 1 : chaine = "oc"

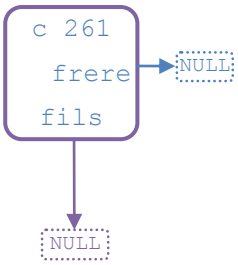


Appel 2 : chaine = "oc"

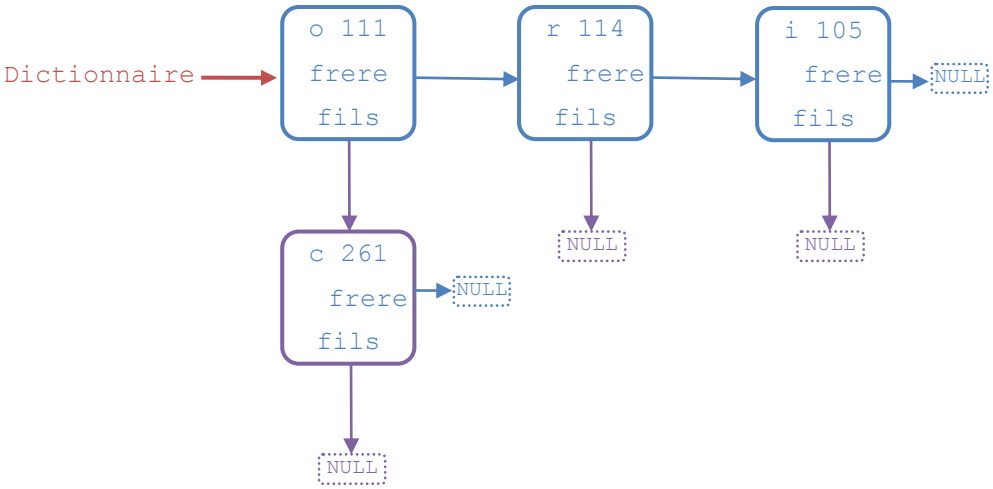


Appel 3 : chaine = "c"

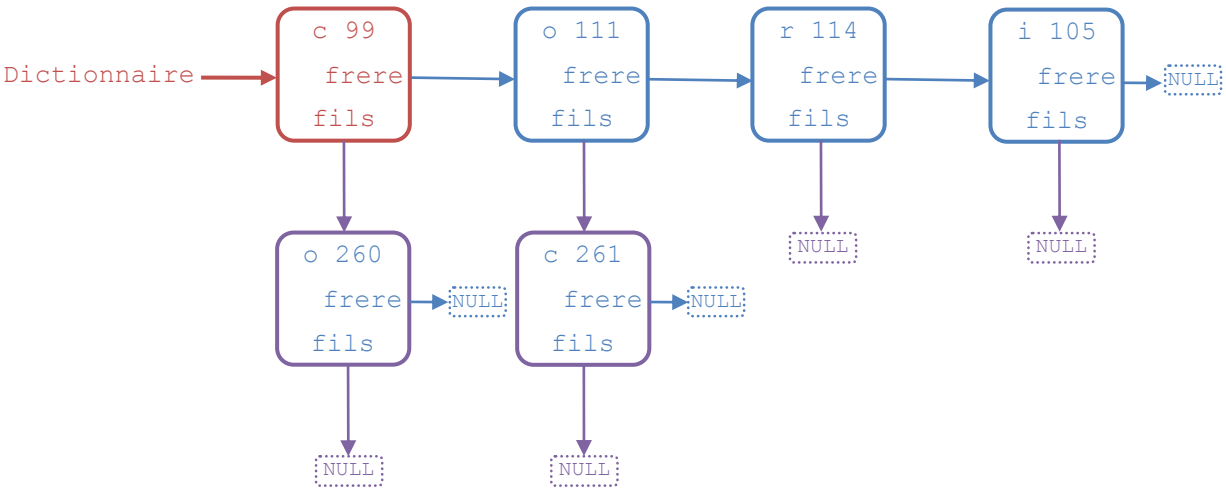
Appel de la fonction creer_fils_cascade ("c") qui renvoie le maillon contenant 'c' avec le code 261.



Retour à l'appel 2 : chaine = "oc"



Retour à l'appel 1 : chaine = "oc"



3.7 Gestion de la taille du dictionnaire

La taille du dictionnaire est l'un des facteurs qui a le plus d'impact sur l'efficacité de l'algorithme de compression.

Chaque code de compression est stocké dans le fichier de sortie sur un certain nombre de bits. En effet, ce nombre doit être connu car lors de la lecture et de l'écriture des codes de compression, nous avons besoin d'avoir un moyen de délimiter les différents codes de compression entre eux.

La méthode la plus simple consiste à fixer la taille du dictionnaire. On utilise alors une constante NB_BITS_CODE qui définit le nombre de bits à allouer pour chaque code. Le dictionnaire peut alors contenir $2^{\texttt{NB\_BITS\_CODE}} = 2^{11} = 2048$ codes. La constante `NB_BITS_CODE` sera définie par une constante de préprocesseur ce qui permettra de changer facilement la taille du dictionnaire (attention toutefois, la taille du dictionnaire doit rester strictement supérieure à 8 bits, pour stocker au moins la table des caractères ASCII et avoir au moins 1 bit de libre pour émettre d'autres codes).

Avec cette méthode, une fois que tous les codes possibles sont émis, le dictionnaire sera considéré comme plein et la chaîne de caractères restante à compresser, sera compressée uniquement à l'aide des codes déjà présents dans le dictionnaire. Cela est effectivement possible car l'ensemble de la table ASCII est initialement chargée dans le dictionnaire, cependant cela réduira considérablement l'efficacité de la compression.

Une autre méthode aurait été d'utiliser une taille de dictionnaire dynamique, c'est-à-dire qu'on augmente la taille des codes émis de 1 bit dès que tous les codes du dictionnaire ont été émis. Cette méthode permet de s'adapter automatiquement à la taille du texte, et donc d'être le plus efficace possible. Cependant, par manque de temps, seule la première méthode a été implémentée.

4. Lecture et écriture dans les fichiers

Ce projet nous fait manipuler deux types de fichiers : texte et binaire. Pour chacun de ces types de fichiers, nous devons penser aux fonctions de lecture et d'écriture.

4.1 Lecture binaire du fichier d'entrée

La librairie standard du C contient des fonctions de lecture binaire qui permettent de lire octet par octet (paquet de 8 bits). Or, les codes de compression de notre algorithme sont stockés sur NB_BITS_CODE bits, et NB_BITS_CODE n'est pas forcément un multiple de 8.

L'algorithme suivant permet de lire en binaire un code de compression du fichier d'entrée. Pour ce faire, on lit le nombre nécessaire d'octets qui contiennent le code de compression. Tous les bits de chaque octet lu sont placés les uns à la suite des autres dans la variable code, à l'exception du dernier octet lu. Pour le dernier octet lu, on ne récupère que les bits appartenant à ce code de compression et les autres bits sont stockés dans un buffer (la taille du buffer est de 8 bits, soit 1 octet). Les bits du buffer appartiennent au code de compression suivant et seront utilisés au prochain appel de la fonction.

```
Buffer ← 0 /* Debut d'ecriture dans le buffer à droite */
indice_buffer ← 0

Fonction lire_code_binaire () :
    /* Récupération des bits restants dans le buffer */
    nb_bits_a_lire ← NB_BITS_CODE - indice_buffer
    code ← buffer << nb_bits_a_lire

    i ← 1
    Tant que i <= (nb_bits_a_lire / 8) :
          buffer ← Lire 8 bits du fichier d'entrée
          temp ← buffer << (nb_bits_a_lire - 8*i)
          code ← code | temp
          i ← i + 1
    Fin Tant que


    /* Lecture du dernier paquet de 8 bits, les bits restants iront dans le buffer */
    Si nb_bits_a_lire n'est pas un multiple de 8 :
          buffer ← Lire 8 bits du fichier d'entrée
          temp ← buffer >> (8 - nb_bits_a_lire % 8)
          code ← code | temp

          /* Stockage dans le buffer des bits restants */
          indice_buffer ←  8 - nb_bits_a_lire % 8
          buffer ← buffer << (8 - indice_buffer)
          buffer ← buffer >> (8 - indice-buffer)
    Sinon :
          buffer ← 0
          indice_buffer ← 0
    Fin Si

    Retourner code
Fin Fonction
```

4.2 Écriture binaire du fichier de sortie

La librairie standard du C contient des fonctions d'écriture binaire qui permettent également d'écrire octet par octet (paquet de 8 bits). Or, les codes de compression de notre algorithme sont stockés sur NB_BITS_CODE bits, et NB_BITS_CODE n'est pas forcément un multiple de 8.

L'algorithme suivant permet d'écrire en binaire un code de compression dans le fichier de sortie. Pour ce faire, on écrit par paquet de 8 bits autant de fois que la taille du code de compression le permet. Pour les bits restants, on les conserve dans un buffer (la taille du buffer est de 8 bits, soit 1 octet) afin de les coupler au prochain code de compression à écrire. Le booléen vider_buffer permet d'écrire les bits restants dans le buffer à la fin de l'algorithme de compression.

```
buffer ← 0 /* Debut d'ecriture dans le buffer a gauche */
indice_buffer ← 7

Fonction ecrire_code_binaire (code, vider_buffer) :
    i ← NB_BITS_CODE - 1
    Tant que i >= 0 :
          bit_i ← ((code >> i) & 1)

          Si bit_i = 1 :
                Mettre à un le bit n°indice_buffer de buffer
          Fin Si

          Si indice_buffer = 0 :
                Ecrire les 8 bits du buffer dans le fichier de sortie
                buffer ← 0
                indice_buffer ← 7
          Sinon :
                indice_buffer = indice_buffer - 1
          Fin Si

          i ← i - 1
    Fin Tant que

    Si vider_buffer :
          Ecrire les 8 bits du buffer dans le fichier de sortie
          buffer ← 0
          indice_buffer ← 7
    Fin Si
Fin Fonction
```

5. Organisation modulaire

Notre projet peut être découpé en différentes parties logiques, appelées « modules ». Chaque module sera découpé en deux fichiers : un fichier contenant la définition formelle des fonctions en relation avec le module (en .c) et un fichier contenant les déclarations des structures de données, les prototypes de fonctions ainsi que les directives préprocesseurs (en .h).

Voici l'ensemble des modules du projet :

- dictionnaire : gestion du dictionnaire et déclaration de sa structure.
- compression : algorithme de compression et les fonctions associées.
- decompression : algorithme de décompression et les fonctions associées.
- lecture_ecriture : gestion de la lecture et de l'écriture binaire dans un fichier.
- chaine_caracteres : gestion des chaînes de caractères.

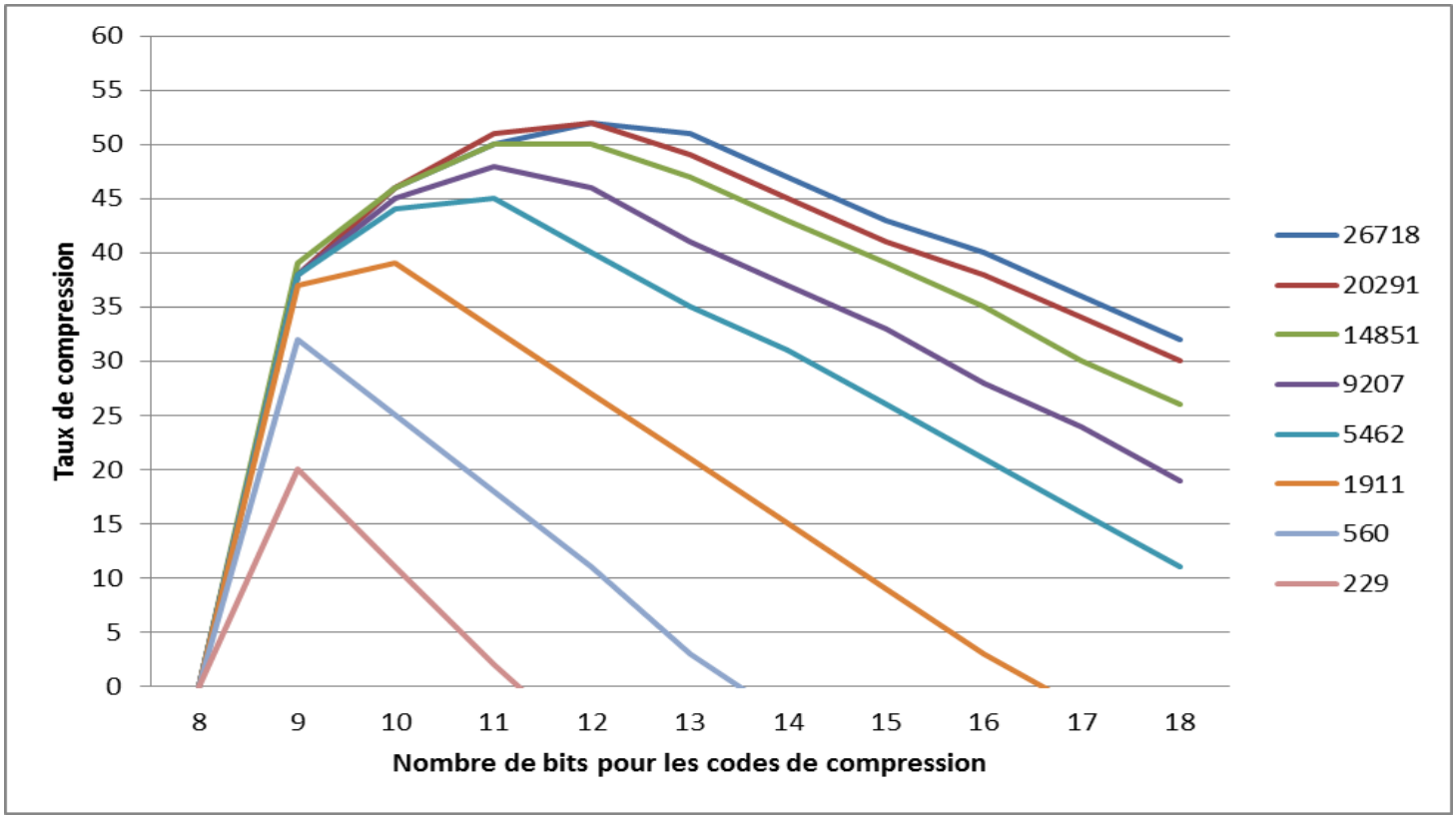
L'exécution du programme et l'appel aux différents modules seront gérés par le module main.c. Ce module aura pour rôle principal d'analyser les paramètres donnés par l'utilisateur lors de l'appel du programme et d'organiser ensuite le traitement de la tâche demandée.

La compilation du projet sera automatisée par l'ajout d'un « Makefile ».

6. Tests et résultats

Afin d'analyser l'efficacité de l'algorithme de compression, nous avons effectué une série de tests. Pour ce faire, nous avons choisi un extrait des Misérables de Victor Hugo comportant 26718 caractères. Ce texte a été successivement compressé en utilisant des codes de 9 bits, puis de 10 bits, …, jusqu'à 18 bits. Pour chaque valeur, nous avons relevé dans un tableau le taux de compression obtenu. Cette opération a été répétée avec le même texte tronqué à 20291, puis à 14851, puis à 9207, puis à 5462, puis à 1911, puis à 560, puis à 229 caractères.

Nombre de bits	26718 caractères	20291 caractères	14851 caractères	9207 caractères	5462 caractères	1911 caractères	560 caractères	229 caractères
8	0	0	0	0	0	0	0	0
9	38	38	39	38	38	37	32	20
10	46	46	46	45	44	39	25	11
11	50	51	50	48	45	33	18	2
12	52	52	50	46	40	27	11	-6
13	51	49	47	41	35	21	3	-15
14	47	45	43	37	31	15	-3	-23
15	43	41	39	33	26	9	-10	-32
16	40	38	35	28	21	3	-18	-41
17	36	34	30	24	16	-2	-25	-50
18	32	30	26	19	11	-8	-33	-59



On observe donc, que plus un texte est long, plus la possibilité d'obtenir un taux de compression élevé est forte. Par ailleurs, plus un texte est court, plus un code sur un nombre de bits faible est optimal pour la compression. Ces observations sont cohérentes avec le fonctionnement de l'algorithme de compression. En effet, le principe de la compression est fondé sur l'élimination des chaînes de caractères répétitives. Ainsi, plus un texte est long, plus l'algorithme aura la possibilité de remplacer de longues chaînes de caractères par un simple code de compression. Il faut donc prévoir un nombre suffisant de bits afin de pouvoir continuer à attribuer des codes de compression lorsqu'on arrive vers la fin du texte à compresser. Cependant, si on utilise un trop grand nombre de bits pour un texte trop petit, toute la plage des codes de compression ne sera pas attribuée et donc un ou plusieurs bits seront inutilisés pour le stockage des codes de compression.

7. Planning récapitulatif des tâches effectuées

Semaines (nombre de séances)	Tâches
26/11/2012 (2)	➢ Création du module principal : main.c et du Makefile ➢ Gestion du dictionnaire : - Création de la structure de données - Implémentation des fonctions primaires - Affichage du dictionnaire
03/12/2012 (1)	➢ Gestion du dictionnaire : - Ajout d'un mot - Recherche d'un mot - Recherche d'un code
10/12/2012 (1) 17/12/2012 (1)	➢ Lecture d'un fichier d'entrée texte ➢ Écriture d'un fichier de sortie texte ➢ Algorithme de compression ➢ Algorithme de décompression
07/01/2013 (2)	➢ Ecriture binaire ➢ Gestion de la taille fixe du dictionnaire
14/01/2013 (2)	➢ Ajout des statistiques de compression ➢ Tests finaux sur des fichiers textes de différentes tailles ➢ Rédaction du rapport
21/01/2013 (2)	➢ Rédaction du rapport ➢ Préparation de la soutenance
04/02/2013 (0)	➢ Rédaction du rapport ➢ Préparation de la soutenance

8. Bilan du projet

Cette partie est une synthèse de notre gestion de ce projet.

Commencer par le document d'analyse a permis d'établir l'organisation générale du projet. Le fait d'organiser le travail à effectuer fût très utile pour nous répartir les tâches. Les modules ont été divisés en tâches qui ont été réalisées de façon indépendante par soucis d'efficacité. Cependant, nous avons régulièrement fais le point pour regrouper le travail et tester le programme.

En ce qui concerne les objectifs que nous nous étions fixés, nous les avons globalement tous réalisés.

Dans un premier temps, nous avons implémenté les algorithmes de compression et de décompression en mode texte. Il fallait réaliser cette première version du programme pour pouvoir le tester.

Dans un deuxième temps, nous avons rajouté les fonctions de lecture et d'écriture binaire ainsi que la gestion de la taille du dictionnaire.

La phase finale du projet consistait à faire le menu, la gestion du programme en ligne de commandes et l'affichage des statistiques de compression.

Pour terminer, il est important de signaler que notre binôme a fonctionné, tout du long, de manière la plus égale possible. Nos niveaux en programmation n'étant pas le même, il fallait néanmoins que nous comprenions au même niveau ce que nous codions, et pour cela nous nous sommes expliqués de nombreux points. C'est donc un partage qui fût bénéfique pour nous deux.

9. Bibliographie

- Article de Wikipedia : Lempel-Ziv-Welch (http://fr.wikipedia.org/wiki/Lempel-Ziv-Welch)
- Cours de Fondements de l'informatique (Licence 2) de M. Michel VAN CANEGHEM (professeur à Aix-Marseille Université, site de Luminy)