Polytech Marseille - 163, avenue de Luminy
13288 Marseille

Département Informatique
3ème année

Tuteur : M. DURAND Nicolas

Compression Lempel-Ziv-Welch

Rapport de projet d'algorithmique et de programmation

Léo JEAN & Loïc MAYOL

Février 2018

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

La compression Lempel-Ziv-Welch (LZW) est une méthode de compression de données *non-destructive*. C'est-à-dire, qu'il n'y aucune perte de données lors de la décompression des données codées compressées.

Cette méthode de compression est une amélioration par Terry Welch de l'algorithme inventé par Abraham Lempel et Jacob Ziv en 1978.

Le principal objectif de la compression LZW est d'éviter les répétitions dans les chaînes de caractères du fichier à compresser, afin d'économiser de la place.

L'algorithme utilise une *table de traduction*, que nous appellerons *dictionnaire* qui se contruit dynamiquement, au cours de la compression mais aussi de la décompression.

Dans notre premier chapitre et notre second chapitre, nous parlerons de l'algorithme de compression et de décompression, et nous les illustrerons avec des exemples.

Dans un troisième chapitre, nous aborderons la gestion et la création du dictionnaire. Nous expliquerons nos choix quant à la structure de celui-ci mais aussi les fonctions majeures que nous utilisons.

Dans le quatrième chapitre, nous commenterons nos choix quant à l'écriture et la lecture des fichiers en binaire.

Dans un dernier chapitre, nous parlerons de l'architecture modulaire de notre programme.

Chapitre 2

Algorithme de compression

2.1 Fonctionnement

L'algorithme de compression permet, à partir d'un texte parcouru de façon linéaire, d'ajouter à un dictionnaire fourni les caractères du texte. Le dictionnaire est dans un premier temps initialisé avec tous les caractères puis dans un second temps nous y ajoutons les chaînes qui composent le texte. Un code est attribué à chaque entrée (voir Chapitre 4). L'algorithme écrit ensuite dans le fichier de sortie la série de codes correspondant au texte initial.

Le fonctionnement de l'algorithme se décrit comme suit : nous parcourons le fichier d'entrée lettre par lettre, puis, nous stockons dans un " tampon " les caractères en attente de codage. Deux cas se présentent :

— si la concaténation du tampon et de la lettre actuellement lue se trouve dans le dictionnaire, nous rallongeons le tampon en y ajoutant la lettre.

— sinon, nous ajoutons ce nouveau " mot " au dictionnaire et nous écrivons en sortie le code correspondant au tampon. Le tampon est ensuite vidé et remplacé par la lettre lue.

L'opération est ainsi répétée tant qu'il reste du texte à lire et nous ajoutons finalement le code de l'ultime tampon.

2.2 Un exemple

Afin d'illustrer l'algorithme, prenons l'exemple de la chaîne de caractère suivante : *barbapapa*.

N. étape	Tampon	Caractère lu	Code émis	Ajout au dictionnaire
0	(vide)	b		Non
1	b	a	98	ba→ 260
2	a	r	97	ar→ 261
3	r	b	114	rb→ 262
4	b	a		Non
5	ba	p	260	bap→ 263
6	p	a	112	pa→ 264
7	a	p	97	ap→ 265
8	p	a		Non
9	pa	EOF	264	

TABLE 2.1 – Exemple de compression de la chaîne *barbapapa*

2.2.1 Explication des étapes

A l'étape 0 : nous lisons le premier caractère "b", celui-ci se trouve déjà dans le dictionnaire et nous le rangeons dans le tampon.

A l'étape 1 : nous lisons le deuxième caractère "a", nous vérifions ensuite que la concaténation avec le tampon n'est pas déjà dans le dictionnaire. Nous ajoutons donc "ba" avec le code 260 et nous imprimons en sortie le code qui correspond au tampon "b". Le tampon est remplacé par "a".

Nous continuons ce processus jusqu'à l'étape 4 : la concaténation du tampon et du caractère ("ba") se trouve déjà dans le dictionnaire, nous remplaçons donc le tampon par cette concaténation à l'étape 5 et le code émis est celui que nous avions déjà enregistré dans notre dictionnaire, le code 260. Nous ajoutons ensuite le tampon concaténé avec la lettre "p" au dictionnaire et nous réinitialisons le tampon.

En continuant jusqu'à l'étape 9 où le texte se termine (`End Of File`) , nous obtenons le code suivant : [98 97 114 260 112 97 264]

2.3 Pseudo-code

Le pseudo-code qui correspond à l'algorithme de compression est donc le suivant :

Algorithm 1 compression(fichier_entree, fichier_sortie, dictionnaire)

```
1: tampon, lettre, concat sont des chaînes de caractères ;
2: tampon = " " ;
3: while caractère à lire do
4:    lettre = lire le prochaine caractère de fichier_entree ;
5:    concat = concaténation de tampon et lettre ;
6:    if concat ∈ dictionnaire then
7:       tampon = concat ;
8:    else
9:       Ajouter concat à dictionnaire ;
10:      Écrire le code de tampon dans fichier_sortie ;
11:      tampon = c
12:   end if
13: end while
14: Écrire le code de tampon dans fichier_sortie ;
```

Chapitre 3

Algorithme de décompression

3.1 Fonctionnement

L'algorithme de décompression, quant à lui, se sert du texte compressé pour reconstituer le texte de départ. Il ne nécessite pas de dictionnaire car il le reconstruit au fur et à mesure que le texte est généré.

L'algorithme lit le fichier code après code. Le premier code lu correspond forcément au code ASCII du premier caractère du texte.

Puis, pour chaque code, nous avons deux cas possibles :

— soit le code se trouve déjà dans le dictionnaire, il suffit donc de décoder et de l'écrire dans le fichier où nous mettons le résultat de la décompression,

— soit le code ne se trouve pas dans le dictionnaire, cela veut donc dire que lors du passage dans l'algorithme de compression, un mot a été ajouté avec ce code. Il faut donc faire la concaténation du mot précédent que nous avons sauvegardé au préalable, avec le premier caractère de la chaîne en train d'être décodée, c'est-à-dire le premier caractère du mot précédent.

Finalement, nous ajoutons le mot au dictionnaire et nous l'imprimons dans le fichier, nous remplaçons ensuite le mot précédent par la chaîne qui était en train d'être décodée.

Nous répétons tant qu'il reste des codes à lire.

3.2 Exemples

3.2.1 Premier exemple

A la sortie de l'exemple précédent, nous avons le code suivant : [98 97 114 260 112 97 264] ; la décompression se passe de la façon suivante :

N. étape	Code lu	Previous	Mot décodé	Ajout au dictionnaire
0	98	(vide)	b	Non
1	97	b	a	ba→ 260
2	114	a	r	ar→ 261
3	260	r	ba	rb→ 262
4	112	ba	p	bap→ 263
5	97	p	a	pa→ 264
6	264	a	pa	ap→ 265

TABLE 3.1 – Premier exemple de décompression

3.2.2 Explication des étapes

Étape 0 : le code 98 est lu, nous savons qu'il correspond à "b" dans la table ASCII, nous le rangeons ensuite dans Previous (le mot précédent).

Etape 1 : le code 97 est lu, nous décodons donc "a" et nous ajoutons "ba", la concaténation de Previous et du mot lu dans le dictionnaire au code 260.

Nous continuons ainsi jusqu'à l'étape 3 : le code 260 est lu, celui-ci étant déjà dans le dictionnaire, nous pouvons directement écrire "ba" en sortie, nous ajoutons "rb" au dictionnaire et Previous devient "ba".

En continuant de telle sorte, nous arrivons à la fin où il faut écrire le code correspondant au tampon. Le mot reconstitué est donc bien "*barbapapa*" qui est le bon mot en entrée.

3.2.3 Deuxième exemple

Prenons un autre exemple où nous trouvons un code qui ne se trouve pas encore dans le dictionnaire : le code [116 97 260 262 97] correspond à la chaîne "*tatatata*" et vérifie ce cas.

N. étape	Code lu	Previous	Mot décodé	Ajout au dictionnaire
0	116	(vide)	t	Non
1	97	t	a	ta→ 260
2	260	a	ta	at→ 261
3	262	ta	tat	tat→ 262
4	97	tat	a	tata→ 263

TABLE 3.2 – Deuxième exemple de décompression

3.2.4 Explication des étapes

La même chose se passe comme dans l'exemple précédent jusqu'à l'étape 3, où le code 262 n'est pas dans la table, cela signifie donc que notre mot est Previous, concaténé avec son premier caractère, on ajoute donc "*tat*" au dictionnaire.

3.3 Pseudo-code

Le pseudo-code qui correspond à l'algorithme de décompression est donc le suivant :

Algorithm 2 decompression(fichier_entree, fichier_sortie, dictionnaire)

```
1: code ∈ N
2: precedent, mot_lu, concat sont des chaînes de caractères ;
3: Écrire precedent dans fichier_sortie ;
4: while code à lire dans fichier_entree do
5:    code = lire le code suivant de fichier_entree ;
6:    if code ∈ dictionnaire then
7:       mot_lu = rechercher code dans dictionnaire ;
8:    else
9:       mot_lu = concaténation de precedent et premier caractère de precedent ;
10:   end if
11:   Écrire mot_lu dans fichier_sortie ;
12:   concat = concaténation de precedent et premier caractère de mot_lu ;
13:   Ajouter concat à dictionnaire ;
14:   precedent = mot_lu ;
15: end while
16: Écrire le code de tampon dans fichier_sortie ;
```

Chapitre 4

Le gestion du dictionnaire

Comme dit précedemment, les algorithmes de compression et décompression utilisent tout les deux un dictionnaire. Nous avons choisi de représenter ce dictionnaire sous la forme d'un arbre.

4.1 Représentation de la structure de données

Afin de créer notre structure de données, nous nous sommes basés sur l'arbre lexicographique utilisé dans le cadre du projet : *complétion automatique des mots*. Nous l'avons cependant modifié afin d'enlever les éléments superflus. En effet, dans notre cas, l'ajout du caractère de fin de chaîne `\0` est inutile car chaque sous-chaîne d'un mot est forcément un mot du dictionnaire.

Ainsi, chaque nœud du dictionnaire dispose d'un pointeur pour passer à son fils, mais aussi à son frère et à son père. Dans le cas de table ASCII, le père des 255 premiers élements sera forcément `NULL` et le frère sera son voisin de droite.

Pour représenter une chaîne de caractère, nous utiliserons deux informations, la lettre et le code de compression de la chaîne de caractères définie par la concaténation ordonnée (racine, puis fils) des lettres des différents noeuds. Nous nous servirons donc du pointeur `NULL` pour indiquer une fin horizontale (lorsque le frère pointe sur `NULL`) mais aussi une fin verticale (lorsque le fils pointe sur `NULL`).

De plus, afin de simplifier la recherche de mot, nous utilisons aussi un tableau dynamique qui pointe sur tous les nœuds créés et ayant comme indice le code du nœud.

Pour récapituler, chaque nœud du dictionnaire dispose de six informations :
— la lettre,
— le code de compression de la chaîne de caractères,
— son fils,
— son frère,
— son père,

— table[code].

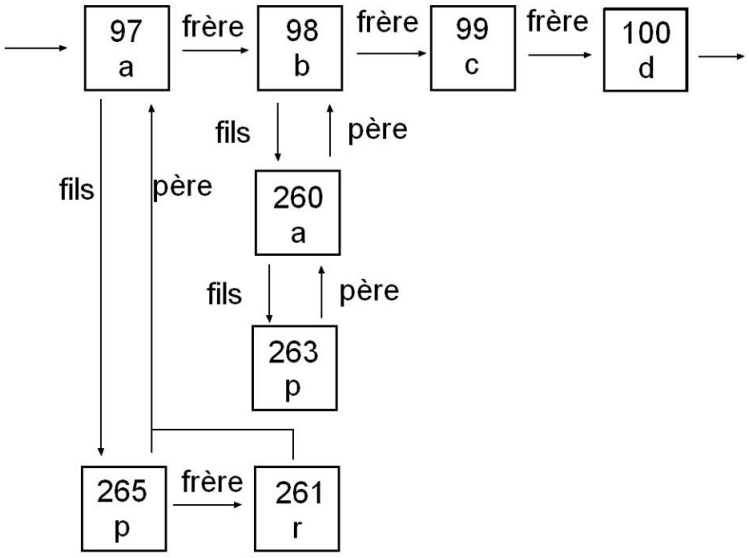

FIGURE 4.1 – Exemple de représentation de la structure

4.2 Fonctions basiques

Dans le but de simplifier la rédaction des algorithmes, nous avons utilisé les fonctions suivantes :

— `creer_noeud(lettre, frere, fils, pere)` qui créé un nœud ayant `lettre` comme lettre, `frere` comme frère, `fils` comme fils et `pere` comme père. Le code de compression sera attribué indépendamment et le nœud créé sera stocké dans `table[code]`.
— `code_noeud(noeud)` qui renvoie le code de `noeud`.
— `lettre_noeud(noeud)` qui renvoie la lettre de `noeud`.
— `frere_noeud(noeud)` qui renvoie le pointeur vers le frère de `noeud`.
— `fils_noeud(noeud)` qui renvoie le pointeur vers le fils de `noeud`.
— `pere_noeud(noeud)` qui renvoie le pointeur vers le père de `noeud`.
— `initialiser_dictionnaire()` qui renvoie un arbre représentant les 255 caractères de la table ASCII.
— `modifier_lettre(noeud, lettre)` qui modifie la lettre de `noeud` par `lettre`.
— `ajouter_fils(noeud, fils)` qui modifie le fils de `noeud` par `fils`.
— `ajouter_frere(noeud, frere)` qui modifie le frère de `noeud` par `frere`.

— `ajouter_pere(noeud, pere)` qui modifie le père de `noeud` par `pere`.

4.3 Recherche d'un code dans le dictionnaire

La recherche du code correspondant à une chaîne de caractère sera effectuée par la fonction `rechercher_caractere(dico, chaine)`. Cette recherche peut s'effectuer récursivement.

Nous allons traiter individuellement chaque caractère de la chaîne. Nous regarderons donc si le premier caractère de la chaîne appartient à un nœud ou à ses frères. Si le premier caractère n'est dans aucun nœud, alors le mot n'existe pas dans notre dictionnaire. Si ce n'est pas le cas, nous appelererons récursivement la fonction sur chacun des fils du nœud avec une chaîne de caractère cette fois-ci privée de son premier élement.

Cette fonction a donc deux cas d'arrêt :

— le dictionnaire est vide. Dans ce cas-là, nous renverrons `0` pour dire que le mot n'a pas été trouvé.

— le dernier caractère de la chaîne de caractère a été trouvé et nous renvoyons donc le code du nœud de ce caractère.

Algorithm 3 rechercher_caractere(dico, noeud)

```
1:  noeud = dictionnaire ;
2:  if (dico == NULL) then
3:     Return 0 ;
4:  end if
5:  if (lettre_noeud(noeud) == chaine[0]) then
6:     if (longueur(chaine) == 1) then
7:        Return code_noeud(noeud) ;
8:     else
9:        chaine = chaine privée du premier caractère ;
10:       Return rechercher_caractere(fils_noeud(noeud), chaine) ;
11:    end if
12: else
13:    if (lettre_noeud(noeud) > chaine[0]) then
14:       Return 0 ;
15:    else
16:       Return rechercher_caractere(frere_noeud(noeud), chaine) ;
17:    end if
18: end if
```

4.3.1 Exemple

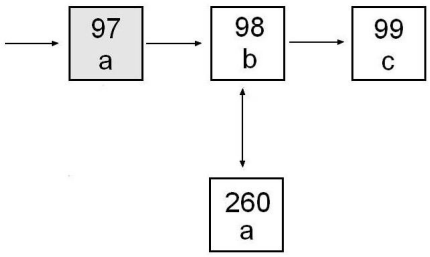

FIGURE 4.2 – Exemple rechercher code 1

Dans le cas où nous voulons recherche le code correspondant à la chaîne "ba" du mot barbapapa, nous parcourons d'abord tous les frères du dictionnaire (la table ASCII).

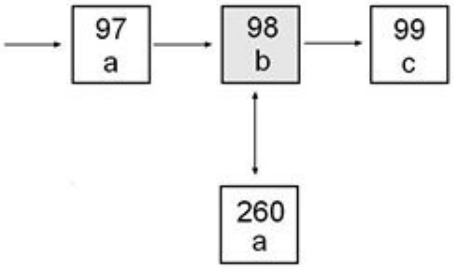

FIGURE 4.3 – Exemple rechercher code 2

Puis, après avoir trouver "b", nous déscendons d'un rang et appelons la fonction sur la chaîne sans son premier caractère : "a". Nous obtenons donc le code ba : 260.

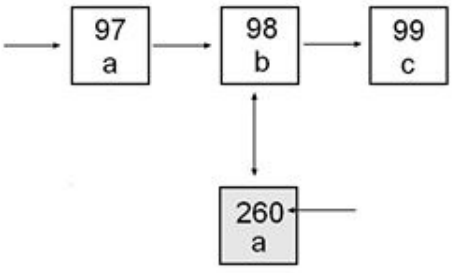

FIGURE 4.4 – Exemple rechercher code 3

4.4 Rechercher un mot dans le dictionnaire

Pour rechercher un mot (composé d'une seule ou de plusieurs lettres) dans le dictionnaire à partir d'un code, nous utilisons la fonction rechercher_mot.

Tout d'abord, nous associons au code un nœud dans notre tableau " table " qui peut être initialisé ou non lors de l'ajout du mot au dictionnaire ; deux cas se présentent :

— Si le code ne correspond à rien dans " table ", cela veut donc dire que nous n'avons pas créé ce nœud et donc pas ajouté le mot au dictionnaire (lors de la compression ou de la décompression) . Nous retournons donc le résultat : " Le mot n'existe pas dans le dictionnaire ".

— Si notre code est bien présent dans " table " alors il correspond à un nœud associé à un unique caractère ainsi que de divers liens de parenté (père, fils, frère) .

Dans le deuxième cas, nous procédons ainsi :

— Notre caractère issu du nœud est sauvegardé, nous remontons ensuite d'un " rang " vers le père du nœud et faisons la concaténation du caractère de ce nœud avec le caractère précédent. Cette opération continue tant que le père du nœud existe.

Finalement, nous obtenons la concaténation du dernier caractère de la chaîne jusqu'au premier et il suffit alors d'inverser les lettres de la chaîne obtenue pour avoir notre mot initial dans le bon ordre.

Algorithm 4 rechercher_mot(dictionnaire, chaine)

```
1: code ∈ N
2: noeud = table[code]
3: if noeud n'existe pas then
4:    Return mot = "Le mot n'existe pas" ;
5: else
6:    mot[0] = lettre_noeud(noeud) ;
7:    while pere_noeud existe do
8:       noeud = pere_noeud(noeud) ;
9:       mot = concaténation de mot et lettre_noeud(noeud) ;
10:   end while
11: end if
12: mot = inverser mot ;
13: Return mot ;
```

4.4.1 Exemple

Nous avons ajouté lors de la compression le mot "*abc*" avec les codes suivants : a=97 (code ASCII) , ab=300, abc=301. La fonction ajout_dico fournit donc entre les trois nœuds les relations suivantes :

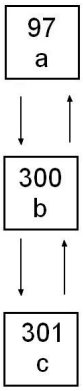

FIGURE 4.5 – Relation entre les nœuds

Il faut rechercher le mot correspondant au code 301, notre tableau de nœuds "table" ressemble à :

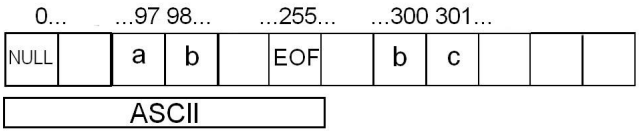

FIGURE 4.6 – Exemple de table

A partir du nœud de code 301, nous remontons au père et concaténons les deux caractères ("*c*" et "*b*"), puis nous remontons à "*a*" (code=97) et concaténons la chaîne. Le nœud "*a*" n'a pas de père nous pouvons donc nous arrêter ici.

Le résultat est "*cba*" que nous inversons pour obtenir "*abc*".

4.5 Ajout d'un mot dans le dictionnaire

Tout comme `rechercher_caractere`, l'ajout d'un mot dans le dictionnaire peut-être fait récursivement. Le but étant de réduire la taille de la chaîne à chaque appel de la fonction. Nous regardons donc si le premier caractère est dans le dictionnaire. Si c'est le cas, nous parcourons ses fils afin de rajouter la chaîne privée du premier caractère ou plus si plusieurs caractères sont déjà présents dans le dictionnaire.

Nous distinguons donc plusieurs cas d'arrêt :

— le fils du nœud est `NULL`. Si c'est le cas, nous appelons une autre fonction `ajout_plusieurs_fils` qui rajoutera les caractères restants de la chaîne sous forme de fils au nœud.

— le frère du nœud est `NULL`. Dans ce cas, nous créons un nouveau nœud auquel nous ajouterons des fils grâce à la fonction `ajout_plusieurs_fils` et qui deviendra le frère du nœud précédent.

— la lettre de la chaîne est inférieure à celle du frère de nœud. Si c'est le cas, nous faisons comme précedemment, en ajoutant un nœud ayant comme frère le nœud.

— la chaîne de caractère est vide (il ne reste plus que `\0`). Nous retournons donc le dictionnaire afin d'arrêter la récursion.

Algorithm 5 ajouter_mot(dictionnaire, chaine)

```
1: if (chaine[0] == findechaine) then
2:    Return dico ;
3: end if
4: noeud = dico ;
5: if (lettre_noeud(noeud) == chaine[0]) then
6:    if fils_noeud(noeud) == NULL) then
7:       chaine = chaine privée du premier caractère ;
8:       Return ajout_plusieurs_fils(noeud, chaine) ;
9:    else
10:      chaine = chaine privée du premier caractère ;
11:      ajouter_fils(noeud, ajout_dico(fils_noeud(noeud), chaine) ;
12:   end if
13: else if lettre_noeud(noeud) > chaine[0] then
14:   noeud = cree_noeud(chaine[0], noeud, NULL, pere_noeud(noeud)) ;
15:   chaine = chaine privée du premier caractère ;
16:   noeud = ajout_plusieurs_fils(noeud, chaine) ;
17: else
18:   if (frere_noeud(noeud) == NULL then
19:      noeud = cree_noeud(chaine[0], noeud, NULL, pere_noeud(noeud)) ;
20:      chaine = chaine privée du premier caractère ;
21:      Return ajout_plusieurs_fils(noeud, chaine) ;
22:   else
23:      ajouter_frere(noeud, ajout_dico(frere_noeud(noeud), chaine) ;
24:   end if
25: end if
26: Return noeud ;
```

Afin de créer plusieurs fils à la fois, nous utilisons une fonction qui va ajouter des fils un à un (en partant du premier caractère) au père. Nous prenons bien soin qu'à chaque fois que nous ajoutons un fils, nous désignons aussi son père pour pouvoir ensuite utiliser au mieux la fonction `rechercher_mot`.

Algorithm 6 ajouter_plusieurs_fils(pere, chaine)

1: $noeud = cree\_noeud(chaine[0], NULL, NULL, pere)$;
2: $tmp = NULL$;
3: $i = 1$
4: $ajouter\_fils(pere, noeud)$;
5: **for all** i tel que $i < longueur(chaine)$ **do**
6: $tmp = noeud$;
7: $noeud = cree\_noeud(chaine[i], NULL, NULL, tmp)$;
8: $ajouter\_fils(tmp, noeud)$;
9: **end for**
10: Return $pere$;

4.5.1 Exemple

Nous souhaitons ajouter au dictionnaire le mot "abc" de code 481. Le dictionnaire est déjà composé, en plus des codes ASCII, de la chaîne "aa" au code 444.

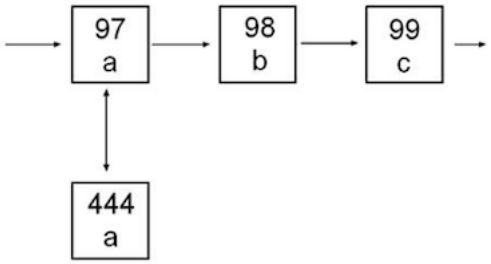

FIGURE 4.7 – Exemple ajouter mot 1

Tout d'abord, nous créons le fils "b" (code 480) à "a" (code 97). Le noeud "aa" devient ainsi le frère du noeud que nous venons d'ajouter.

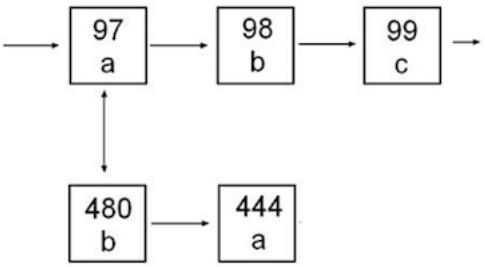

FIGURE 4.8 – Exemple ajouter mot 2

Finalement, nous ajoutons via la ajout_plusieurs_fils le noeud "c" (code 481) pour obtenir la chaîne "abc".

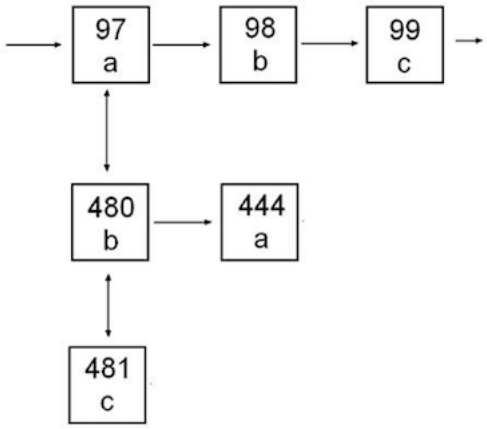

FIGURE 4.9 – Exemple ajouter mot 3

4.6 Gestion de la taille du dictionnaire

Il existe deux façons de gérer le dictionnaire. Nous pouvons fixer sa taille ou utiliser un dictionnaire à taille dynamique. La taille du dictionnaire aura forcément un impact sur l'efficacité de l'algorithme.

En effet, chaque code de compression est stocké dans le fichier de sortie sur un certain nombre de bits. Ce nombre doit être connu afin de pouvoir décompresser mais aussi pour délimiter chaque code lors de l'écriture du fichier compressé. Dans notre cas, la taille du dictionnaire étant fixe, il suffit de préciser lors de l'exécution de l'algorithme de compression et de décompression sur combien de bits nous travaillons. Pour cela, nous avons défini une constante `NB_BITS` qui sert à définir la taille maximale du dictionnaire s'exprimant sous la forme 2^{nb_bits}. Il est cependant très important que `NB_BITS` soit strictement supérieure à 8 bits car 2^8 ne peut contenir que la table ASCII.

En utilisant un dictionnaire à taille fixe, lorsque tous les codes possibles ont été utilisés, le dictionnaire est considéré plein et les chaînes de caractères restantes seront compressées uniquement à l'aide des codes déjà existants dans le dictionnaire.

Dans le cas d'un dictionnaire à taille dynamique, nous augmentons de un la taille du dictionnaire dès que tous les codes on était émis. Nous n'avons cependant pas eu le temps d'essayer cette méthode.

Chapitre 5

Lecture et écriture

La lecture et l'écriture de caractères et de codes (ASCII ou autres) se font automatiquement par des fonctions fournies par le langage C, or, lors de la lecture d'un texte décompressé composé uniquement de codes binaires, il nous faut un algorithme capable d'interpréter les bits selon la taille du code, celle-ci n'étant pas forcément définie par octet(s).

5.1 Lecture en binaire

Pour lire un code en binaire, nous comptons le nombre d'octets présents dans le code (division euclidienne du nombre de bits par 8). Au dernier octet, nous récupérons les bits restants qui correspondent au code (le reste de la division), ceux restants sont rangés dans un tampon et sont restaurés lors du prochain appel de la fonction.

5.2 Écriture en binaire

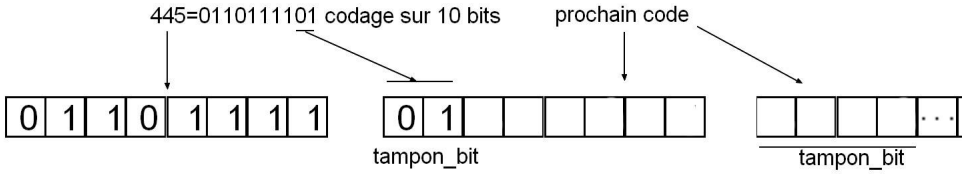

FIGURE 5.1 – Exemple écriture binaire

Pour écrire le résultat de notre compression en binaire dans le fichier de sortie, nous écrivons notre nombre octet par octet autant de fois que la taille du nombre en bits le permet, puis les bits restants (bits de poids faibles) sont rangés dans un tampon que nous réutilisons lors de l'appel de la fonction pour le prochain code à

écrire.

Prenons l'exemple du codage suivant : 445 sur 10 bits, les premiers bits sont codés sur l'octet et les deux bits de poids faibles sont rangés dans tampon_bit de taille 2.

Le prochain code est ensuite inscrit dans la suite de l'octet après tampon_bit et ses bits restants sont stockés sur tampon_bit, cette fois de taille 4.

Chapitre 6

Programmation modulaire

Afin de se séparer aux mieux nos tâches, nous avons découpé notre projet en différentes parties, appelées "*modules*". Chaque module est découpé en deux fichiers :

— un fichier contenant les prototypes des différentes fonctions ainsi que les déclarations des structures de données (en .h),

— un fichier contenant l'algorithme des fonctions (en .c).

Nous avons donc trois modules :

— `dictionnaire` : ce module contient les fonctions relatives à la gestion du dictionnaire mais aussi la déclaration de sa structure.

— `compression` : ce module contient les algorithmes de compression et de décompression.

— `lecture_ecriture` : ce module contient les fonctions relatives à la gestion de la lecture et de l'écriture mais aussi la gestion des chaînes de caractères (avec une fonction pour la concaténation notamment).

L'exécution du programme et l'appel des différents modules seront fait à l'aide du fichier `main.c`. Ce module nous permettra ainsi de lancer la compression et la décompression de nos fichiers.

La compilation se fera à l'aide d'un *Makefile*, nous compilerons en utilisant la commande `make` et nous nettoierons en utilisant la commande `make clean`.

Chapitre 7

Conclusion

À travers ce projet, nous avons pu constater les efforts à fournir ainsi que les nombreuses difficultés qu'il est possible de rencontrer lors de la programmation d'algorithmes. Néanmoins, la répartition des tâches en fonction des différents aspects du programme nous a permis de progresser comme nous le souhaitions.

Nous avons pu nous rendre compte de l'importance de bien comprendre les algorithmes (en particulier la compression et la décompression) afin de fournir un travail conforme aux consignes de départ.

Une conséquente partie du travail consistait à la gestion du dictionnaire, sa façon de le concevoir en terme de structure, de taille, puis en terme de relations avec les deux algorithmes (l'ajout de mot, la recherche,...).

Nous avons donc pu, au terme du projet, confirmer l'efficacité de l'algorithme LZW et donc son intérêt pour la compression d'un texte.

Finalement, nous pensons que cette expérience de travail en binôme a été très enrichissante, du point de vue de l'organisation à avoir, du travail à répartir, de la conception du code au coeur du programme, de nos différentes visions sur la manière de procéder.