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

Informatique – 3ème année

Document d'analyse 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 3

1. Algorithme de compression LZW 4
   1.1 Principe 4
   1.2 Pseudo-code 4
   1.3 Exemple 4
2. Algorithme de décompression LZW 5
   2.1 Principe 5
   2.2 Pseudo-code 5
   2.3 Exemple 5
3. Gestion du dictionnaire 6
   3.1 Représentation de la structure de données 6
   3.2 Fonctions primaires 6
   3.3 Affichage du dictionnaire 7
   3.4 Ajout d'un mot au dictionnaire 8
   3.5 Recherche d'un code dans le dictionnaire à partir du mot 7
   3.6 Recherche d'un mot dans le dictionnaire à partir du code 8
   3.7 Gestion de la taille du dictionnaire 9
4. Lecture et écriture dans les fichiers 10
   4.1 Lecture du fichier d'entrée 10
   4.2 Ecriture du fichier de sortie 10
5. Organisation modulaire 11
6. Planning Prévisionnel 12

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 codées 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 compléter 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).

1. Algorithme de compression LZW

1.1 Principe

Avant l'algorithme de compression, le dictionnaire est initialisé avec tous les caractères de la table ASCII.

Ensuite, toutes les suites de caractères lus sont ajoutées au fur et à mesure dans le dictionnaire, et sont numérotés par un code de compression (sur `TAILLE_DICTIONNAIRE` = 10 bits). En effet, à 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 255, car le dictionnaire est initialisé avec la table des caractères 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).

Une fois le code décrypté, 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 Exemple

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.

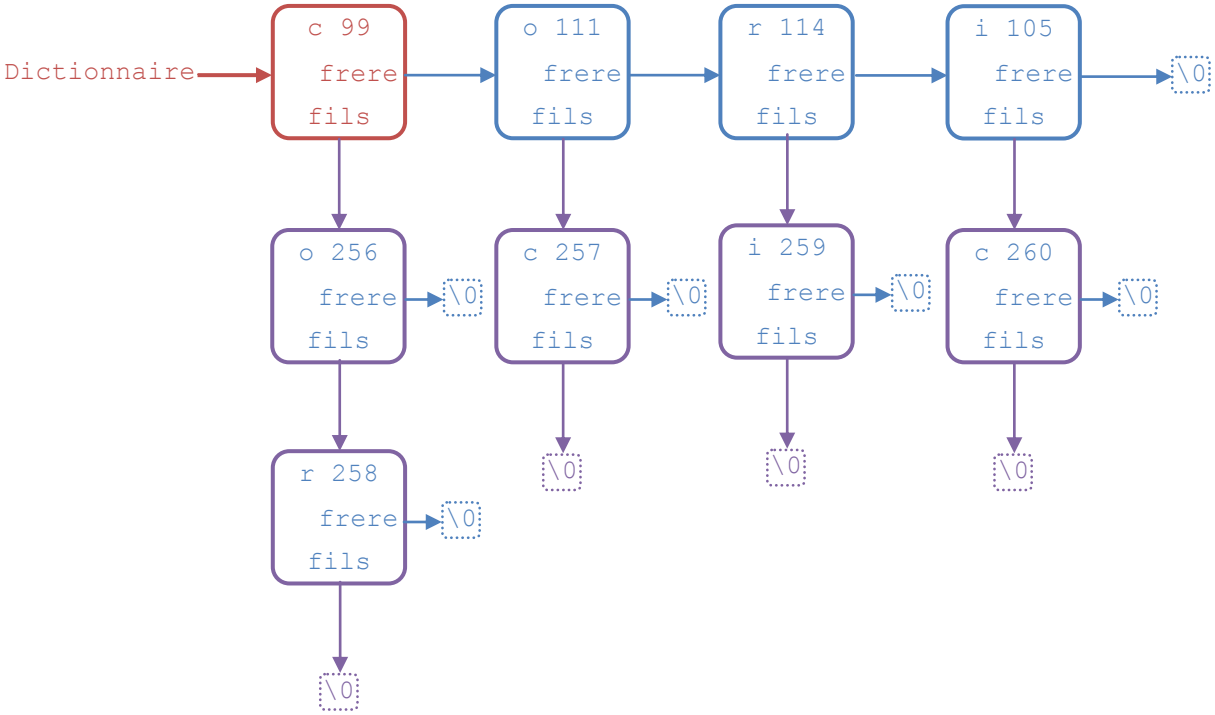
3. Gestion du dictionnaire

Les algorithmes de compression et de décompression LZW utilisent un dictionnaire. Ce dictionnaire sera représenté par un arbre lexicographique.
Le choix d'utiliser un arbre plutôt qu'une autre structure de donnée est judicieux puisque 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 sera représenté de la manière classique « fils-frère », c'est-à-dire que chaque nœud 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, le dictionnaire pointera donc sur le premier caractère de l'arbre.
De plus, chaque nœud contiendra deux informations : le 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 caractère de fin de chaîne `\0` pour indiquer la fin horizontale (le frère pointe sur le caractère nul) ou verticale (le fils pointe sur le caractère nul) 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 suivante :

- `est_feuille (arbre)` renvoie si `arbre` est réduit à une feuille ou non.
- `est_vide (arbre)` renvoie si `arbre` est vide ou non.
- `arbre_vide ()` renvoie l'arbre nul, c'est-à-dire l'arbre vide.
- `creer_nœud (arbre_fils, arbre_frere, caractere)` renvoie le nœud ayant `arbre_fils` comme fils, `arbre_frere` comme frère et `caractere` comme caractère (le code de compression est attribué indépendamment).
- `fils (nœud)` renvoie le lien sur le fils de `nœud`.
- `frere (nœud)` renvoie le lien sur le frère de `nœud`.
- `caractere (nœud)` renvoie le caractère correspondant à `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.

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 affichage_dictionnaire (dictionnaire) :
    parcours (dictionnaire, "")
Fin Fonction

Fonction parcours (nœud, chaine) :
    Si nœud est une feuille :
          Ajouter caractere (nœud) à chaine
          Afficher chaine

          Retour Fonction
    Sinon :
          Ajouter caractere (nœud) à chaine
          parcours (fils (nœud), chaine)

          Si nœud a un frère :
                parcours (frere (nœud), chaine)
          Fin Si

          Retour Fonction
    Fin Si
Fin Fonction
```

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 le mot n'existe pas dans le dictionnaire. Sinon, on vérifie que ce n'est pas le dernier caractère de la chaîne à rechercher, car si c'est le cas, alors on renvoie le code du nœud. Si ce n'est pas le dernier caractère à rechercher, on rappelle récursivement la fonction sur le fils du nœud contenant le dernier caractère trouvé et sur la sous-chaîne de caractères privée de son premier caractère.

La condition d'arrêt de la fonction récursive est lorsque le dictionnaire est vide, on ne peut pas trouver de mot dans un dictionnaire vide, donc on renvoie le code signifiant qu'aucune chaîne n'a été trouvé.

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

    Si dictionnaire est vide :
          Retourner 0
    Sinon Si caractere (nœud) = chaine[0] :
          Si longueur (chaine) = 1 :
                Retourner code (nœud)
          Sinon :
                Supprimer le caractère 0 de chaine
                Retourner rechercher_code (fils (nœud), chaine))
          Fin Si
```

```
    Sinon :
          Si caractere (nœud) > chaine[0] :
                Retourner 0
          Sinon :
                Retourner rechercher_code (frere (nœud), chaine)
          Fin Si
    Fin Si
Fin Fonction
```

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

La recherche d'un mot dans le dictionnaire se fait récursivement. En effet, on appelle récursivement la fonction sur le fils du nœud en question puis si on a rien trouvé, sur le frère du nœud.

Sinon, si on trouve le code, on renvoie la chaîne de caractères contenant le caractère du nœud dont le code correspond. On reconstitue la chaîne de caractère entière en remontant la récursion, en rajoutant en position 0 le caractère de chaque nœud.

La fonction récursive s'arrête donc soit en trouvant le code dans le dictionnaire, soit en arrivant sur une feuille dont le code ne correspond pas au code recherché.

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

    Si nœud est une feuille :
          Si code (nœud) = code :
                Retourner la chaine contenant seulement caractere (nœud)
          Sinon :
                Retourner ""
          Fin Si
    Sinon :
          chaine ← ""

          chaine ← rechercher_mot (fils (nœud), code)
          Si chaine = "" ET nœud a un frere :
                chaine = rechercher_mot (frere (nœud), code)
          Fin Si

          Si chaine != "" :
                Insérer à la position 0 de chaine : caractere (nœud)
          Fin Si

          Retourner chaine
    Fin Si
Fin Fonction
```

3.1 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 de 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 et `K` comme code de compression) 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) le mot `chaine` dans le dictionnaire en attribuant un caractère du mot `chaine` par fils créé et un code de compression `K'` distinct pour chaque 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.

Cependant, comment détermine-t-on les codes de compression `K` et `K'`, c'est-à-dire les codes de compression attribués lorsque nous ajoutons des nœuds ou des feuilles à notre arbre ?

L'approche la plus simple voudrait définir une variable statique accessible uniquement dans la fonction creer_nœud qui servirait de compteur en retenant le dernier code de compression émis. A chaque ajout d'un nœud, le compteur est incrémenté afin d'émettre un nouveau code et ce code est stocké dans le nœud.

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

          Si caractere (nœud) > chaine[0] :
                sous_chaine ← chaine – chaine[0]
                nœud ← creer_nœud (creer_fils_cascade (sous_chaine), nœud, 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 (en effet, on vient de créer son fils à l'itération précédente) lorsqu'on créé ce nœud.

```
Fonction creer_fils_cascade (chaine) :
    nœud ← arbre_vide ()
    n ← longueur (chaine)

    Tant que chaine n'est pas vide :
          nœud ← creer_nœud (nœud, arbre_vide (), chaine[n-1])
          Supprimer le caractère n-1 de chaine
          Décrémenter n
    Fin Tant que

    Retourner nœud
Fin Fonction
```

3.2 Gestion de la taille du dictionnaire

Dans un premier temps, par souci de simplicité, nous allons considérer que la taille du dictionnaire est fixe et que les codes émis seront sur `TAILLE_DICTIONNAIRE` = 10 bits, ce qui limite le dictionnaire à $2^{\texttt{TAILLE\_DICTIONNAIRE}} = 2^{10} = 1024$ codes. La constante `TAILLE_DICTIONNAIRE` 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).

Une fois tous les codes possibles 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.

Cependant, nous avons à réfléchir sur l'incidence qu'un dictionnaire plein aura sur nos algorithmes. Est-ce l'algorithme de compression qui doit vérifier que le dictionnaire n'est pas plein ? Est-ce les fonctions de gestion du dictionnaire ? Dans un premier temps, la gestion la plus simple d'un dictionnaire plein est de définir une variable booléenne globale qui indique si le dictionnaire est plein ou pas. Cette variable serait utilisée par les algorithmes de compression et de décompression qui, dès que le dictionnaire est plein, alors on ne fait que utiliser (pour la compression) ou lire les codes (pour la décompression) des chaînes de caractères déjà incluses dans le dictionnaire.

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 du fichier d'entrée

Dans le cas où le fichier d'entrée est un fichier texte (compression), à priori, nous utiliserons directement les fonctions de lecture existantes dans la librairie standard du C.

Dans le cas où le fichier d'entrée est un fichier binaire (décompression), nous devrons établir une fonction intermédiaire (qui utilise tout de même les fonctions standards du C) permettant de lire un certain nombre de bits à la fois (10 bits pour notre choix). En effet, les fonctions standard de lecture binaire du C ne permettent de lire qu'octet (8 bits) par octet. Nous devrons donc lire deux octets à la fois, isoler les bits qui nous intéressent et stocker les bits restants dans un buffer qui sera alors lu lors du prochain appel à cette fonction de lecture.

4.2 Ecriture du fichier de sortie

Dans le cas où le fichier de sortie est un fichier texte (décompression), à priori, nous utiliserons directement les fonctions d'écritures existantes dans la librairie standard du C.

Dans le cas où le fichier de sortie est un fichier binaire (compression), nous devrons établir une fonction intermédiaire (qui utilise, tout de même, les fonctions standards du C) permettant d'écrire un certain nombre de bits à la fois (10 bits pour notre choix). En effet, les fonctions standards d'écriture binaire du C ne permettent d'écrire qu'octet (8 bits) par octet. Nous devrons donc écrire un octet à la fois, sauvegarder les bits en attente d'écriture dans un buffer, et à chaque appel de la fonction d'écriture, insérer les bits restants dans le buffer au début de la suite de bits à écrire.

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.

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. Planning Prévisionnel

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 - Gestion de la taille fixe du dictionnaire
10/12/2012 (1)	→ Lecture d'un fichier d'entrée texte → Algorithme de compression → Ecriture d'un fichier de sortie binaire (*)
17/12/2012 (1)	→ Lecture d'un fichier d'entrée binaire (*) → Algorithme de décompression → Ecriture d'un fichier de sortie texte
07/01/2013 (2)	→ Ajout des statistiques de compression (*) → Tests finaux sur des fichiers textes de différentes tailles (*) → Comparaison avec des outils classiques de compression (*) → Rédaction du rapport
14/01/2013 (2)	→ Rédaction du rapport
21/01/2013 (2)	→ Rédaction du rapport (à rendre le 05/02/2013) → Préparation de la soutenance
04/02/2013 (0)	→ Préparation de la soutenance (qui aura lieu le 08/02/2013)

(*) : Ces tâches seront effectuées selon le temps restants et l'avancement du projet.