Comprimer des fichiers

La question de l'économie d'espace pour représenter l'information s'est posée très tôt. Alors que la technologie a fait des progrès fantastiques, les besoins en communication et en stockage ont également crû dans les mêmes proportions. De ce fait, on cherche constamment à réduire la taille des fichiers et la bande passante consommée.

Nous présentons deux algorithmes à la fois très connus et toujours très employés, en particulier dans les logiciels d'archivage du marché ou encore dans les modules de compression mis en œuvre pour Internet et pour le cloud.

1. Approche par statistique : l'algorithme d'Huffman

On parle aussi de codage d'Huffman, du nom de son inventeur. Le principe de cet algorithme consiste à réduire le nombre de bits pour coder des caractères fréquents (octets) dans le fichier à comprimer, et rallonger les autres.

En langue française, la lettre E est la plus fréquente, tandis que les lettres K ou Z le sont beaucoup moins. D'une façon empirique, un codage du E sur 7 bits au lieu de 8 fera gagner 1 bit à chaque occurrence.

Comme il faut toujours 256 codes différents pour représenter chaque caractère ASCII (1 octet), on admettra que les lettres les moins fréquentes aient des codages plus longs, par exemple sur 9 bits.

L'algorithme d'Huffman construit un codage binaire (un arbre à deux branches ou B-tree) à partir des statistiques des caractères. Selon les implémentations, les statistiques sont prédéfinies en fonction du type de fichier ou dynamiquement calculées à partir du fichier à compresser. Des implémentations encore plus élaborées modifient le codage en actualisant les statistiques au fur et à mesure du traitement du fichier.

a. Implémentation du codage

Le codage démarre par un comptage des occurrences de chaque caractère (représenté sur un octet) dans le fichier source. On utilise pour cela un tableau de classe

`Noeud` qui servira à la construction de l'arbre pour le codage :

```
class Noeud
{
public:
  int code, lcode;
  int freq, sauve;

  int fils1, fils2;

  Noeud()
  {
     code = lcode = freq = sauve = 0;
     fils1 = fils2 = -1;
  }
};
```

La classe `Huffman` porte toute la logique de l'algorithme pour la compression et la décompression :

```
#define MAX 512

class Huffman : public ICompresseur
{
private:
  Noeud**stats;
  int lus, ecrits;
  int sommet;

  void compter(FILE*src);
  void afficher_stats();
  int rechercher_min();
  void genere_code(Noeud*n, int code, int lcode, int m);
  int ecrire_statistiques(FILE*dest);
  void lire_statistiques(FILE*src);
  void encoder_fichier(FILE*src, BitFile*dest);
  void decoder_fichier(BitFile*src, FILE*dest,int taille_originale);

public:
```

```
  Huffman(void);
  ~Huffman(void);
  virtual int comprimer(char*nom, FILE*source, FILE*dest);
  virtual void expanser(char*nom, int taille_originale, int
taille_archive, FILE*source);
};
```

La méthode `compter()` lit une première fois le fichier source (à compresser) pour déterminer les statistiques de chaque caractère :

```
#pragma region compter
void Huffman::compter(FILE*src)
{
  int c;
  lus = 0;
  while ((c = fgetc(src)) != -1)
  {
    stats[c]->freq++;
    lus++;
  }
}
#pragma endregion
```

La construction de l'arbre est un processus itératif qui recherche les deux indices de plus faibles fréquences et les regroupe pour former un nouveau nœud. Le même arbre sera d'ailleurs construit à la décompression.

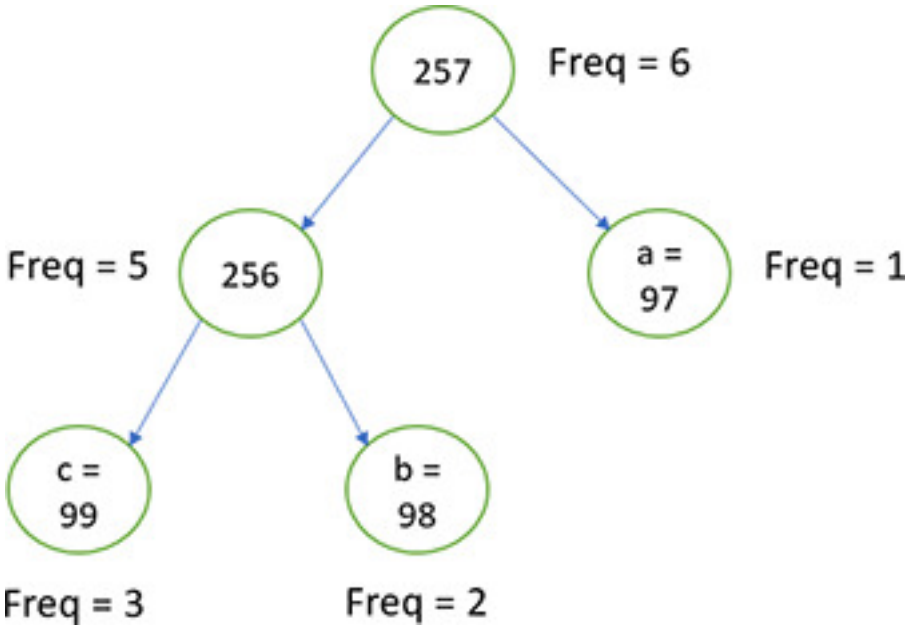
```
// construction de l'arbre
sommet = 256;
int min1, min2;
min1 = min2 = -1;
int poids = 0;

while (poids <= lus && sommet < MAX)
{
  // recherche le plus petit non nul
  min1 = rechercher_min();
```

```
        if (min1 < 0)
            break;

        stats[min1]->sauve = stats[min1]->freq;
        stats[min1]->freq = 0;

        // recherche le second plus petit non nul
        min2 = rechercher_min();
        if (min2 < 0)
            break;

        stats[sommet]->fils1 = min1;
        stats[sommet]->freq = stats[min1]->sauve;

        stats[min2]->sauve = stats[min2]->freq;
        stats[min2]->freq = 0;

        stats[sommet]->fils2 = min2;
        stats[sommet]->freq += stats[min2]->sauve;

        poids = stats[sommet]->freq;
        sommet++;
    }
```

Considérons pour l'exemple un fichier comprenant trois caractères a, b, et c avec les statistiques suivantes :

a	3 occurrences
b	2 occurrences
c	1 occurrence

Le fichier à compresser contient donc six caractères. Nous en déduisons un arbre en rassemblant deux à deux les nœuds de fréquences les plus basses portant au sommet la somme de toutes les fréquences, autrement dit la taille du fichier à compresser :



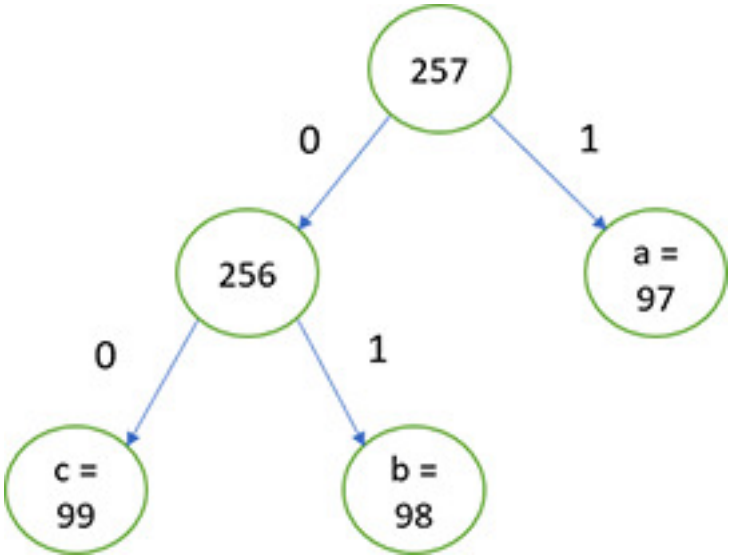
La génération du codage est un processus récursif : en partant de la racine de l'arbre, on code 0 vers le premier fils (conventionnellement à gauche), 1 vers le second fils (à droite) :

```
// génération du codage
sommet--;
genere_code(stats[sommet], 0, 0, 0);
```

À chaque récursion, la longueur du code est augmentée d'une unité :

```
#pragma region genere_code
void Huffman::genere_code(Noeud*n, int code, int lcode, int m)
{
  n->code = (code << 1) | m;
  n->lcode = lcode;

  if (n->fils1 >= 0)
     genere_code(stats[n->fils1], n->code, lcode + 1, 0);

  if (n->fils2 >= 0)
     genere_code(stats[n->fils2], n->code, lcode + 1, 1);
}
#pragma endregion
```

Ceci produit pour l'exemple précédent un codage plus court pour le caractère a (code ascii 97) et un codage plus long pour les caractères b (code ascii 98) et c (code ascii 99) :



b. Compression du fichier

La compression du fichier proprement dite consiste à relire le fichier et à substituer à chaque caractère lu son codage Huffman :

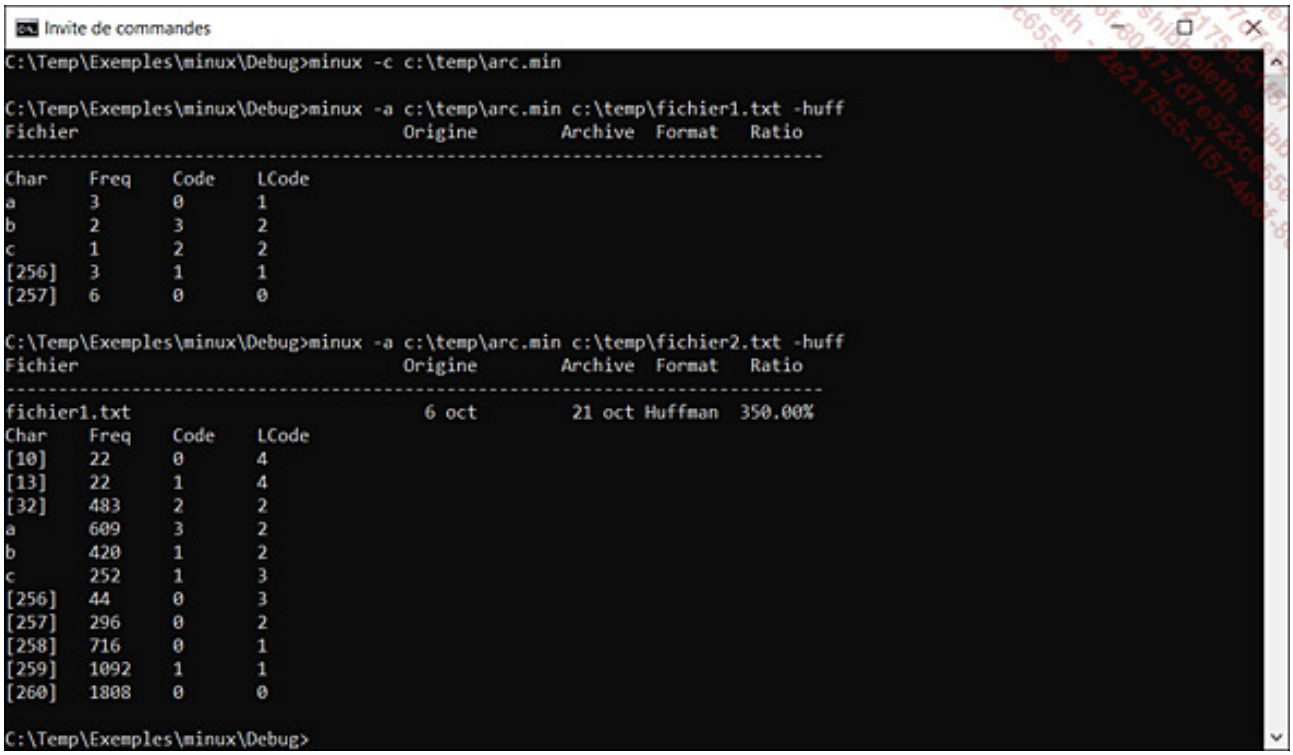
```
#pragma region encoder_fichier
void Huffman::encoder_fichier(FILE*src, BitFile*dest)
{
  int c;
  lus = 0;
  while ((c = fgetc(src)) != -1)
  {
     dest->outputBits(stats[c]->code, stats[c]->lcode);
     lus++;
  }
}
#pragma endregion
```

Se pose toutefois la question de l'arbre à utiliser pour la décompression. Si les statistiques ne sont pas prédéterminées (en fonction du type de fichier par exemple), il faut aussi enregistrer les statistiques dans le flux de sortie, en se limitant aux caractères

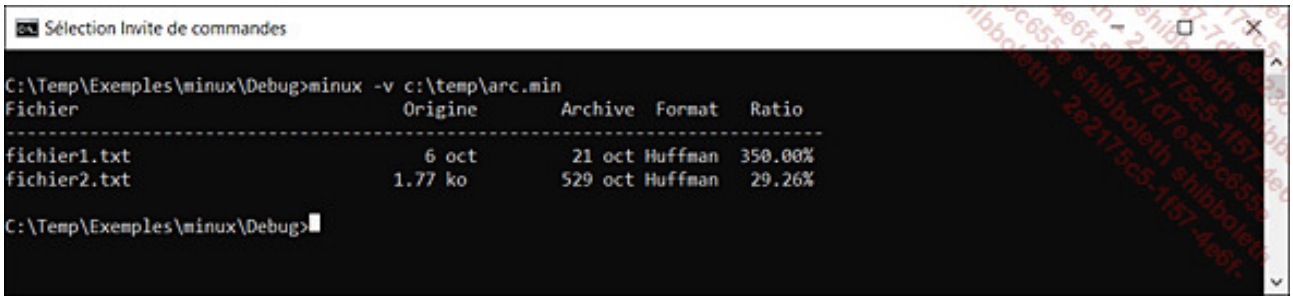
d'occurrence non nulle :

```
#pragma region ecrire_statistiques
int Huffman::ecrire_statistiques(FILE*dest)
{
  int cb = 0;

  // compte le nombre de caractères présents
  int nb_present = 0;
  for (int i = 0; i < 256; i++)
    if (stats[i]->sauve != 0)
      nb_present++;

  fwrite(&nb_present, 1, sizeof(int), dest);
  cb += sizeof(int);

  // sauve les stats pour chaque caractère présent
  for (int i = 0; i < 256; i++)
  {
    if (stats[i]->sauve != 0)
      cb += stats[i]->write(i, dest);
  }
  return cb;
}
#pragma endregion
```

Cette copie d'écran montre que la sauvegarde des statistiques s'avère coûteuse pour les très petits fichiers (fichier1.txt fait 6 caractères et 21 octets une fois compressé) :



Cependant, avec un fichier un peu plus conséquent, l'algorithme donne de bons résultats et compresse un fichier de 1,77 Ko en seulement 350 octets, statistiques comprises :



c. Décompression

Le code C++ de décompression reprend en partie la logique de la compression : lecture des statistiques, construction de l'arbre, génération du codage.

Ensuite, il faut lire bit par bit le fichier d'entrée et suivre l'arbre jusqu'à atteindre des feuilles qui correspondent aux caractères ASCII d'une valeur comprise entre 0 et 255 :

```
void Huffman::decoder_fichier(BitFile*src, FILE*dest,int taille_originale)
```

```
{
  ecrits = 0;
  int s;
  while (ecrits < taille_originale)
  {
    s = sommet;
    while (stats[s]->fils1 >= 0 && stats[s]->fils2 >= 0 && s>255)
    {
      int b = src->inputBit();
      if (b == 0)
        s = stats[s]->fils1;
      else
        s = stats[s]->fils2;
    }
    // en principe 0 <= s <= 255
    char c = (char)s;
    fwrite(&c, 1, sizeof(char), dest);
    ecrits++;
  }
}
```

2. Approche par dictionnaire : l'algorithme LZW

Cet algorithme a été mis au point en plusieurs étapes. À la fin des années 70, deux chercheurs, Abraham Lempel et Jacob Ziv, ont conçu plusieurs algorithmes de compression de données basés sur des dictionnaires, LZ77, LZ78, LZSS. Quelques années plus tard leur concept a été amélioré par Terry Welch, leur donnant l'acronyme LZW. Cet algorithme a été popularisé par de nombreux formats de fichiers tels que .zip ou encore .gif ; il est parfois utilisé en combinaison avec le codage Huffman (LZH, deflate…).

a. Fonctionnement de l'algorithme

L'algorithme remplace les séquences de caractères (on pourrait dire des mots) par un index dans une table. En prenant l'analogie du dictionnaire, on remplacerait chaque mot dans le flux de sortie par un indice formé par le numéro de page et le numéro d'entrée dans la page du dictionnaire.

La contrainte qu'adresse l'algorithme est de constituer le dictionnaire au fur et à mesure de la lecture du fichier d'entrée et de ne pas avoir à fournir tel quel le dictionnaire pour la décompression.

De ce fait, les programmes de compression et de décompression ont un fonctionnement assez symétrique.

Initialement, le dictionnaire contient chaque caractère individuel (disons les lettres de l'alphabet A, B, C… Z). Chaque fois qu'un caractère est lu, on cherche si une chaîne existe dans le dictionnaire en ajoutant ce caractère. Si l'entrée existe, l'index de la chaîne est fourni dans le flux de sortie, sinon on crée une nouvelle entrée dans le dictionnaire et le nouveau caractère est envoyé dans le flux de sortie. Le programme de décompression pourra alors maintenir son propre dictionnaire au même rythme.

À la décompression, on lit un code et on cherche dans le dictionnaire s'il existe une entrée qui correspond. Si c'est le cas, la chaîne du dictionnaire est replacée dans le flux de sortie, autrement le code participe à la création d'une nouvelle entrée qui est enfin recopiée dans le flux de sortie.

Il y a cependant un cas particulier : dans la séquence répétitive chaîne / caractère / chaîne / caractère / chaîne, le décompresseur ne trouve pas la nouvelle entrée et ne reçoit aucun code lui permettant de la créer. Heureusement, c'est le seul cas de figure où le décompresseur ne trouvera pas le code. On applique donc de facto la création d'une nouvelle entrée comme le ferait le compresseur.

b. Implémentation du dictionnaire

Une bonne part du processus réside dans la recherche d'entrées dans le dictionnaire. L'approche itérative naïve s'avère désastreuse du point de vue des performances.

Considérons la capacité du dictionnaire (en nombre d'entrées, donc de chaînes) en fonction de la largeur du codage et le nombre d'opérations nécessaires pour être certain de balayer tout le dictionnaire afin de trouver une chaîne. La longueur moyenne des chaînes est déterminée en partant du principe qu'un dictionnaire de 512 entrées, dont les 256 premières sont réservées aux codes ASCII, ne contiendra pas en moyenne des chaînes de plus de 3 caractères. Cette règle de calcul arbitraire peut bien entendu être remise en question selon le fichier à compresser mais elle convient à notre raisonnement.

Largeur du codage en bits	Nombre d'entrées dans le dictionnaire	Longueur moyenne de chaîne *	Nombre de comparaisons de caractères pour trouver une chaîne
9	512	3	1 536
10	1 024	4	4 096
11	2 048	5	10 240
12	4 096	6	24 576
13	8 192	7	57 344
14	16 384	8	131 072
15	32 768	9	294 912

Plus large est le dictionnaire, plus il pourra s'enrichir de nouvelles combinaisons de caractères. On peut donc espérer de ne pas le saturer prématurément avec des séquences présentes seulement au début du fichier à compresser.

Ceci dit, à chaque caractère du fichier à compresser, selon la largeur du codage utilisé, une technique de recherche trop simpliste nécessitera entre 1 536 et presque 300 000 comparaisons, ce qui est bien trop coûteux.

Une solution classique pour contourner ce problème consiste à hacher les entrées du dictionnaire ; nous avons rencontré cette technique qui consiste à associer un code numérique à une chaîne dans des structures de la STL, comme le `set` ou le `map`.

Autrement dit, si le dictionnaire est un tableau (au sens C++), au lieu de stocker consécutivement les entrées par des index 1, 2, 3, 4…, N, on calcule une clé à partir

d'un algorithme qui minimise le nombre de comparaisons pour atteindre une entrée et trouver une correspondance. Ce type de calcul s'appelle hachage.

En pratique, on utilise une formule basée sur une opération de type XOR et sur un décalage de bits. Avec ce type de formule, la taille de la table est habituellement calibrée avec une marge de 25% par rapport à la taille utile. Pour une table de 512 entrées il faut prévoir un conteneur d'environ 750 places. Un deuxième facteur conditionne le nombre d'opérations de comparaison. Pour éviter que plusieurs chaînes partagent le même code de hachage (on parle de collision ou d'accumulation), la taille de la table doit être un nombre premier assez éloigné d'une puissance de 2. Toujours dans notre exemple de dictionnaire sur 9 bits, 750 entrées ne convient pas mais 751 est un bon choix, puisque à la fois premier et relativement au milieu de deux puissances de 2, soit [2^9 = 512 ; 2^10 = 1 024].

On définit le dictionnaire comme une structure arborescente. Une chaîne de caractères est une liste de caractères. Chaque élément de la liste peut se retrouver à n'importe quel endroit de la table.

Par exemple, pour la chaîne "abc" d'index 500, le nœud 'a' se retrouve à l'index 200, le nœud "ab" qui ajoute 'b' à la chaîne "a" se trouve à l'index 708…

La classe `Dictionnaire` débute la définition :

```
/*
 * Le dictionnaire de chaînes est un arbre
 *
 */
struct Dictionnaire
{
  int valeur_code;   // code LZW
  int code_parent;   // chaînage
  char car;          // caractère
};
```

Nous optons pour un codage qui pourra évoluer de 9 bits à 15 bits ; la table à adressage ouvert doit contenir plus de 32768 entrées, en choisissant un nombre premier, pas trop proche de 32768, sans être trop grand pour ne pas occuper trop de mémoire à l'exécution.

```
/*
 * Constantes relatives à l'implémentation LZW
 * Le codage varie de 9 à 15 bits
 */
#define BITS       15
#define CODE_MAX       ((1 << BITS ) -1)

/* nombre premier compris entre 2^15 et 2^16 */
#define TAILLE_TABLE       35311

/* 137 pages de 256 entrées */
#define TAILLE_PAGE_TABLE (( TAILLE_TABLE >> 8) +1)

#define FIN_DE_FLUX_LZ 256
#define CHANGEMENT_CODAGE 257
#define CODE_PURGE 258
#define PREMIER_CODE 259
#define NON_UTILISE -1
```

Dans la classe `LZW`, le dictionnaire est implémenté comme un ensemble de pages de 256 entrées :

```
class LZW : public ICompresseur
{
private:
  Dictionnaire* dict[TAILLE_PAGE_TABLE];
  Dictionnaire& get_dict(int i)
  {
     return dict[i >> 8][i & 0xff];
  }
```

Deux méthodes sont chargées d'allouer la mémoire pour le dictionnaire et de le réinitialiser :

```
#pragma region initialise_dictionnaire
void LZW::initialise_dictionnaire()
{
```

```
  int i;
  for (i = 0;i < TAILLE_TABLE;i++)
    get_dict(i).valeur_code = NON_UTILISE;

  code_suivant = PREMIER_CODE;
  largeur_codage = 9;
  prochain_code_saut = 511;
}
#pragma endregion

#pragma region initailise_stockage
void LZW::initialise_stockage()
{
  int i;
  for (i = 0;i < TAILLE_PAGE_TABLE;i++) {
    dict[i] = (struct Dictionnaire*) calloc(256,
sizeof(struct Dictionnaire));
  }
}
#pragma endregion
```

Vient enfin la méthode de hachage `trouve_chaine()`, basée sur un calcul assez classique XOR (opération logique ou exclusif). En cas de collision, l'algorithme détermine si la prochaine recherche se fait juste à côté ou au contraire diamétralement à l'opposé de la table. On peut proposer d'autres méthodes de hachage et même, à titre de test, bâtir son dictionnaire à l'aide d'une structure de la STL.

Attention cependant à bien choisir ses paramètres, comme le facteur de charge, pour garantir les performances et une occupation raisonnable de la mémoire.

```
#pragma region trouve_chaine
unsigned int LZW::trouve_chaine(int code_parent, int car_suivant)
{
  unsigned int index;
  int decalage;

  // hachage de la chaîne à partir du code parent et du caractère
suivant
```

```
  index = (car_suivant << (BITS - 8)) ^ code_parent;

  // en cas d'autre hachage, détermine un décalage
  // on cherchera la prochaine entrée à côté ou de l'autre côté
de la table
  if (index == 0)
    decalage = 1;
  else
    decalage = TAILLE_TABLE - index;

  while(true)
  {
    // chaîne introuvable
    if (get_dict(index).valeur_code == NON_UTILISE)
      return ((unsigned int)index);

    // chaîne parent trouvée et caractère trouvé
    if (get_dict(index).code_parent == code_parent &&
      get_dict(index).car == (char)car_suivant)
      return index;

    // chaîne parent trouvée mais caractère non trouvé : on fait
un autre hachage (2ème, 3ème...)
    // jusqu'à trouver une entrée inutilisée ou une correspondance
    if ((int)index >= decalage)
      index -= decalage;
    else
      index += TAILLE_TABLE - decalage;
  }
}
#pragma endregion
```

c. Dimensionnement et gestion du dictionnaire

Maintenant que nous savons comment optimiser la recherche de chaînes dans la table, il faut penser au dimensionnement idoine du dictionnaire et à la vitesse à laquelle il va se remplir.

Si on prend d'emblée un codage sur 12 bits, chaque chaîne de 1, 2, 3, 4 caractères… est

codée sur un octet et demi. Il faut aussi compter un octet pour chaque nouveau caractère ajouté à la chaîne. Cette largeur de codage comprend jusqu'à 4 096 chaînes, ce qui est peut-être plus que nécessaire sur un très petit fichier. Sur un gros fichier, le dictionnaire risque d'être rapidement saturé de chaînes présentes uniquement au début, et le taux de compression risque de s'effondrer.

Les implémentations de LZW prévoient deux mécanismes pour optimiser le taux de compression. Primo la largeur de codage évolue, en partant de 9 bits jusqu'à 12,15 bits voire au-delà. Secundo le dictionnaire peut être vidé, soit lorsqu'il est plein, soit à la demande, lorsque le taux de compression chute.

Le compresseur a la main pour modifier les caractéristiques du dictionnaire. Il doit cependant informer le décompresseur de ces modifications :

```
if (code_suivant > CODE_MAX)
{
  output->outputBits(CODE_PURGE, largeur_codage);
  initialise_dictionnaire();
}
else if (code_suivant > prochain_code_saut)
{
  output->outputBits(CHANGEMENT_CODAGE, largeur_codage);
  largeur_codage++;
  prochain_code_saut <<= 1;
  prochain_code_saut |= 1;
}
```

d. Programme de compression

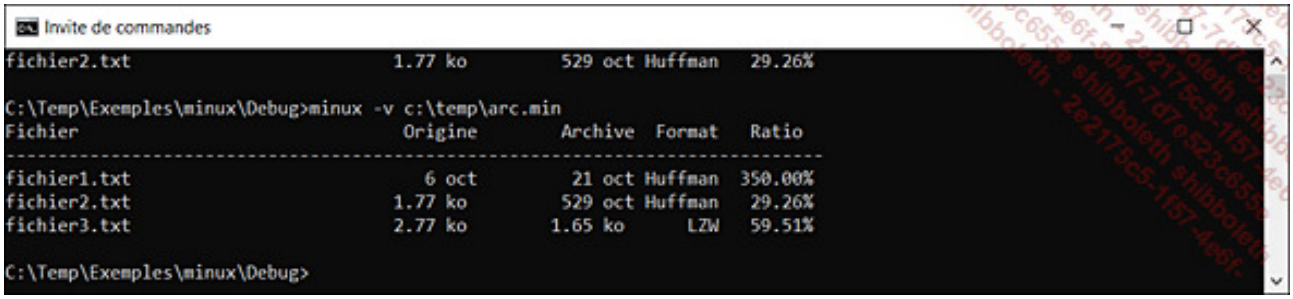
Le programme de compression est très simple et comparativement moins élaboré que le programme Huffman.

```
#pragma region compress_file
void LZW::compress_file(FILE*input, BitFile*output)
{
  int car;
  int code_chaine;
```

```
  unsigned int index;

  initialise_stockage();
  initialise_dictionnaire();

  cb++;
  if ((code_chaine = getc(input)) == EOF)
     code_chaine = FIN_DE_FLUX_LZ;

  while ((car = getc(input)) != EOF)
  {
     cb++;
     index = trouve_chaine(code_chaine, car);

     if (get_dict(index).valeur_code != NON_UTILISE)
        code_chaine = get_dict(index).valeur_code; // la chaîne
existe
     else
     {
        // nouvelle chaîne
        get_dict(index).valeur_code = code_suivant++;
        get_dict(index).code_parent = code_chaine;
        get_dict(index).car = (char)car;

        output->outputBits(code_chaine, largeur_codage);

        code_chaine = car;

        if (code_suivant > CODE_MAX)
        {
           output->outputBits(CODE_PURGE, largeur_codage);
           initialise_dictionnaire();
        }
        else if (code_suivant > prochain_code_saut)
        {
           output->outputBits(CHANGEMENT_CODAGE,
largeur_codage);
           largeur_codage++;
           prochain_code_saut <<= 1;
           prochain_code_saut |= 1;
```

```
        }
      }
    }
    output->outputBits(code_chaine, largeur_codage);
    output->outputBits(FIN_DE_FLUX_LZ, largeur_codage);
  }
  #pragma endregion
```

Ce programme LZW comprime une célèbre poésie de Lamartine avec un gain d'environ 40 % :



e. Programme de décompression

La décompression nécessite un tout petit peu plus de lignes. Il faut d'abord décoder une chaîne, c'est-à-dire remonter les index pour réécrire une séquence de caractères lisible et surtout conforme à l'originale. C'est le rôle de la méthode `decodage_chaine()` :

```
#pragma region decodage_chaine
unsigned int LZW::decodage_chaine(int compte, int code)
{
  // c'est une chaîne composée d'une chaîne et d'un car
  while (code > 255)
  {
    decode_pile[compte++] = get_dict(code).car;
    code = get_dict(code).code_parent;
  }
  decode_pile[compte++] = (char)code;
  return compte;
}
```

```
#pragma endregion
```

Ensuite, il faut mettre en place le gestionnaire d'exception pour la séquence chaîne / caractère / chaîne / caractère / chaîne. Inutile de rechercher cette combinaison, c'est l'unique cas où le décompresseur ne trouve pas la chaîne dans le dictionnaire :

```
#pragma region expand_file
void LZW::expand_file(BitFile*input, FILE*output)
{
  unsigned int nouveau_code;
  unsigned int ancien_code;
  int car;
  unsigned int compte;

  initialise_stockage();

  while(true)
  {
    initialise_dictionnaire();
    ancien_code = (unsigned int)input->
inputBits(largeur_codage);

    if (ancien_code == FIN_DE_FLUX_LZ)
      return;

    car = ancien_code;
    putc(ancien_code, output);

    while(true)
    {
      nouveau_code = (unsigned int)input->
inputBits(largeur_codage);

      if (nouveau_code == FIN_DE_FLUX_LZ)
        return;

      if (nouveau_code == CODE_PURGE)
      {
        initialise_dictionnaire();
```

```
            break;
        }

        if (nouveau_code == CHANGEMENT_CODAGE)
        {
            largeur_codage++;
            continue;
        }

        if (nouveau_code >= code_suivant)
        {
            // cas particulier chaîne + car + chaîne + car +
chaîne
            decode_pile[0] = (char)car;
            compte = decodage_chaine(1, ancien_code);
        }
        else
            compte = decodage_chaine(0, nouveau_code);

        // on repart avec le premier car de la chaîne
        car = decode_pile[compte - 1];

        // écrit la chaîne décodée
        while (compte > 0)
            putc(decode_pile[--compte], output);

        // ajouter chaîne + ancien_code au dictionnaire
        get_dict(code_suivant).code_parent = ancien_code;
        get_dict(code_suivant).car = (char)car;

        code_suivant++;

        ancien_code = nouveau_code;
    }
  }
}
#pragma endregion
```