LZW

1 Structures de données

Conceptuellement, on a besoin de deux structures de données pour implémenter l'algorithme LZW :
— une structure (qu'on appellera dictionnaire) donnant les associations code vers motif ;
— une structure (qu'on appellera table inverse) donnant les associations motif vers code (et permettant de tester si un motif est associé à un code).
On peut imaginer la solution suivante :
— la première structure (le dictionnaire) est réalisée par un tableau, avec le motif associé au code c dans la case d'indice c ;
— la seconde structure (la table inverse) est réalisée par une table de hachage, avec les motifs comme clés et les codes comme valeurs.

1. En supposant qu'on choisisse cette solution :
   — les deux structures sont-elles utiles dans la phase de compression ?
   — et dans la phase de décompression ?

La solution que nous allons utiliser est légèrement différente, et tire parti du fait que la structure des motifs « connus » (*i.e* associés à un code) est très contrainte. Un motif présent dans la table est :
— soit réduit à un octet ;
— soit de la forme mx, avec m un autre motif présent dans le dictionnaire et x un octet.
On peut donc éviter de manipuler des motifs en tant que tels, et n'utiliser que des couples c, x où :
— c est un code déjà existant ;
— x est un octet.
On définit les alias de type suivants :

```
// codeword type
typedef uint32_t cw_t;
// byte type
typedef uint8_t byte_t;
```

Une entrée du dictionnaire est un couple (code, octet) :

```
struct dict_entry_t {
    cw_t pointer;
    byte_t byte;
};

typedef struct dict_entry_t dict_entry_t;
```

Le dictionnaire en lui même est une structure contenant :
— le prochain code disponible ;
— la largeur d'un code (en nombre de bits) – pour l'instant, cette taille sera constante ;
— un tableau statique de `dict_entry_t`. La taille de ce tableau est une constante globale, et vaut 2^d, où d est la largeur maximale d'un code.

```c
#define CW_MAX_WIDTH 16
#define DICTFULL (1u << CW_MAX_WIDTH)

const cw_t NO_ENTRY = DICTFULL;
const cw_t NULL_CW = DICTFULL;
const cw_t FIRST_CW = 0x100;
const int CW_MIN_WIDTH = 16;

struct dict_entry_t {
    cw_t pointer;
    byte_t byte;
};

typedef struct dict_entry_t dict_entry_t;

struct dict_t {
    cw_t next_available_cw;
    int cw_width;
    dict_entry_t data[DICTFULL];
};

struct dict_t dict;
```

Remarques

⇨ On utilise la directive du pré-processeur `#define` pour les constantes `CW_MAX_WIDTH` et `DICTFULL` pour pouvoir définir un tableau statique de taille `DICTFULL` (ce ne serait pas possible si `DICTFULL` était défini comme un `const int`, par exemple). Ce point peut être ignoré.

⇨ `dict` est une variable globale.

⇨ `NO_ENTRY` et `NULL_CW` sont des constantes dont on sait qu'elles ne peuvent correspondre à un code valide. On a trois constantes différentes pour la même valeur 2^d, mais elles seront utilisées dans des contextes différents.

⇨ `FIRST_CW` indique le premier code créé dynamiquement (après les codes pour les motifs de un octet). On a donné sa valeur en hexadécimal, qui correspond à 256 en décimal.

1. Avec les valeurs données ci-dessus pour les différentes constantes (et pour le type `cw_t`), quelle quantité de mémoire le dictionnaire occupe-t-il ?
2. Écrire une fonction `initialize_dictionary` qui initialise les champs `next_available_cw` et `cw_width`, ainsi que la partie du tableau correspondant aux motifs de un octet.
   — Pour `cw_width`, on initialisera à `CW_MIN_WIDTH`.
   — On mettra le champ `pointer` à `NULL_CW` pour les motifs de un octet.

```c
void initialize_dictionary(void);
```

Pour la table inverse (association motif vers code), on utilise un tableau bidimensionnel de codes :

```c
cw_t inverse_table[DICTFULL][256];
```

La case `inverse_table[c][x]` contiendra le code correspondant au couple (c, x) s'il existe, une valeur quelconque sinon.

3. Quelle quantité de mémoire `inverse_table` consomme-t-elle ?
4. Écrire une fonction `lookup` qui prend en entrée un couple c, x et renvoie :
   — le code correspondant à c, x s'il y en a un ;
   — `NO_ENTRY` sinon.

```c
cw_t lookup(cw_t cw, byte_t byte);
```

5. Écrire une fonction `build_entry` qui prend en entrée un couple c, x et ajoute une entrée dans le dictionnaire pour ce motif (en mettant également à jour `inverse_table`). On pourra supposer sans le vérifier que le motif n'est pas déjà présent dans le dictionnaire.

   Dans le cas où le dictionnaire est déjà plein, cette fonction ne fera rien.

```c
void build_entry(cw_t cw, byte_t byte);
```

2 Compression

Pour compresser, nous allons lire le fichier d'entrée octet par octet et émettre des codes sur un fichier de sortie. Pour l'instant, ces codes seront émis en ASCII : si l'on émet 517, on écrira `"517"` (trois caractères ASCII) sur le fichier de sortie. Bien sûr, cela ne permet pas de compresser réellement, mais cela nous permettra de vérifier la correction de notre algorithme de compression.

1. Écrire une fonction `mock_compress` qui prend en entrée un fichier d'entrée et un fichier de sortie et écrit dans le fichier de sortie les codes générés, au format ASCII, séparées par une espace. Par exemple, avec en entrée un fichier réduit à `"ABABCABBAB\n"`, et sachant que le code ASCII de `A` est 65, on doit obtenir le résultat suivant : `"65 66 256 67 256 257 66 10"`.

   ```c
   void mock_compress(FILE *input_file, FILE *output_file);
   ```

   Pour lire un octet du fichier d'entrée, on utilisera la fonction `getc`. Son prototype est :

   ```c
   int getc(FILE *stream);
   ```

   L'entier qu'elle renvoie est soit `EOF` (une constante prédéfinie, strictement négative, qui signifie qu'on est arrivé à la fin du fichier), soit une valeur entre 0 et 255 (qui peut donc être transtypée sans problème en `byte_t`).
2. Écrire un programme ayant le comportement suivant :
   — il accepte entre un et trois arguments en ligne de commande ;
   — le premier argument est réduit à un caractère :
     — `c` pour compresser en mode binaire à l'aide de la fonction `compress` (qui reste à écrire) ;
     — `C` pour compresser en mode texte avec `mock_compress` ;
     — `d` pour décompresser en mode binaire ;
     — `D` pour décompresser en mode texte ;
   — le deuxième argument, s'il est présent, indique le fichier d'entrée (sinon, on prend l'entrée standard) ;
   — le troisième argument, s'il est présent, indique le fichier de sortie (sinon, on prend la sortie standard).
   On en profitera bien entendu pour tester la fonction `mock_compress` sur l'exemple ci-dessus, et sur d'autres exemples de préférence !
3. Quelle est la complexité temporelle totale de votre programme, en fonction de la largeur d des codes et du nombre n d'octets à compresser ?

3 Décompression

La décompression est un peu plus délicate que la compression à cause de la manière dont nous avons choisi de représenter le dictionnaire. Implicitement, le dictionnaire contient les motifs sous forme de listes chaînées de caractères, avec le dernier caractère du motif en tête de liste. Nous voulons bien sûr émettre les caractères dans l'ordre, ce qui peut se faire de deux manières :

— soit à l'aide d'une pile ;
— soit à l'aide d'une fonction récursive.

La version utilisant une fonction récursive est légèrement plus facile à écrire, mais peut être problématique dans les cas pathologiques puisque elle utilise un espace proportionnel à la longueur du motif sur la pile d'appel. On fournit donc une réalisation très simple de la structure de pile *via* le *header* `stack.h`, qui déclare les fonctions suivantes :

```c
typedef struct stack stack;

stack *stack_new(int capacity);
void stack_free(stack *s);
int stack_size(stack *s);
byte_t stack_pop(stack *s);
void stack_push(stack *s, byte_t byte);
```

1. Écrire une fonction `decode_cw` ayant le prototype suivant :

   ```c
   byte_t decode_cw(FILE *fp, cw_t cw, stack *s);
   ```

Cette fonction émettra, dans l'ordre, tous les octets du motif dont le code est `cw` sur le fichier `*fp`. La pile est fournie pour éviter de la réallouer à chaque appel : on pourra supposer qu'elle est de taille suffisante pour contenir tous les octets du motif, et son état tant avant qu'après l'appel n'a pas d'importance. De plus, elle renverra le dernier octet du motif (qui nous sera utile par la suite). Pour émettre un octet sur le flux de sortie, on utilisera :

```
int putc(int ch, FILE *stream);
```

L'argument `ch` est automatiquement converti en `unsigned char` avant d'être écrit, on donnera donc directement un `byte_t` comme argument. La valeur de retour est égale à `ch` si tout s'est bien passé, à `EOF` sinon : on pourra l'ignorer.

2. Écrire une fonction `get_first_byte` qui renvoie le premier octet du motif associé à un code.

```
byte_t get_first_byte(cw_t cw);
```

3. Écrire une fonction `mock_decompress` qui lit un fichier compressé au format produit par `mock_compress` et écrit le flux décompressé correspondant sur `output_file`.
   — Il est conseillé de commencer par retrouver l'algorithme de décompression par soi-même : il n'est pas inenvisageable que l'on vous demande cela le jour d'un concours…
   — Après avoir fourni cet effort, consulter le cours est quand même une bonne idée.
   — La fonction `decode_cw` suffit pour traiter le cas « usuel » ; la fonction `get_first_byte` est utile pour traiter le cas dit KwK (celui où l'on lit un code que l'on n'a pas encore ajouté au dictionnaire).
   — La table inverse ne sert pas pour la décompression.

```
void mock_decompress(FILE *input_file, FILE *output_file);
```

On pensera à tester la fonction sur une entrée contenant un cas KwK (par exemple `"ABABABA\n"`) !

4 Lecture et écriture binaires

Pour réaliser une véritable compression, il est nécessaire qu'un code de largeur d utilise d bits sur le fichier de sortie. Comme d n'a aucune raison d'être un multiple de 8, on est ramené à un problème similaire à celui que l'on a résolu en OCaml pour le code de Huffman. Nous allons procéder de manière très légèrement différente :

— on maintiendra toujours un accumulateur (que nous appellerons `buffer`) et sa taille (en nombre de bits significatifs), et l'on écrira toujours un octet sur le fichier de sortie quand le nombre de bits de l'accumulateur atteindra ou dépassera 8 ;
— cependant, au lieu de recevoir les bits à écrire un par un, nous les recevrons par paquet (un code complet à chaque appel) ;
— d'autre part, la clôture du fichier sera plus simple : on pourra se contenter de compléter le dernier octet par des zéros. En effet, lors de la décompression, on saura toujours combien de bits on souhaite lire, et ce nombre sera toujours supérieur ou égal à 9 (longueur minimale possible d'un code).

On utilise la structure suivante :

```c
const int BUFFER_WIDTH = 64;
const int BYTE_WIDTH = 8;
const uint64_t BYTE_MASK = (1 << BYTE_WIDTH) - 1;

struct bit_file {
    FILE *fp;
    uint64_t buffer;
    int buffer_length;
};

typedef struct bit_file bit_file;

bit_file *bin_initialize(FILE *fp){
    bit_file *bf = malloc(sizeof(bit_file));
    bf->fp = fp;
    bf->buffer = 0;
    bf->buffer_length = 0;
    return bf;
}
```

1. En utilisant un *buffer* de 64 bits, quelle taille de code peut-on traiter au maximum sans problème ? La limitation est-elle gênante ?
2. Écrire une fonction `output_bits` de prototype :

   ```c
   void output_bits(bit_file *bf, uint64_t data, int width, bool flush);
   ```

   Les données à écrire sont des les `width` bits de poids faible de `data`. Le paramètre `flush` détermine le comportement sur les bits restants après avoir écrit autant d'octets que possible dans `bf` :
   — s'il vaut `false`, les bits restants sont laissés dans `bf->buffer` ;
   — s'il vaut `true`, ils sont écrits dans `bf`, complétés par des zéros pour obtenir un octet.
   On écrira les données bit le moins significatif en premier : en particulier, quand on écrit un octet constitué du reste du code précédent et du début du code actuel, les bits de poids faible de l'octet correspondront à l'ancien code et ceux de poids fort au nouveau.
3. Écrire une fonction `input_bits` ayant le prototype suivant :

   ```c
   uint64_t input_bits(bit_file *bf, int width, bool *eof);
   ```

   Cette fonction lit `width` bits depuis le flux `bit_file` (de manière à lire correctement un flux écrit par `output_bits`, évidemment). La valeur pointée par `eof` sera mise à `true` si la lecture a échoué parce que l'on est arrivé à la fin du fichier sans parvenir à lire `width` bits, à `false` sinon.

   Les `width` bits lus seront placés dans les bits de poids faibles de la valeur de retour.
4. Écrire les fonctions `compress` et `decompress`, ayant les mêmes prototypes que `mock_compress` et `mock_decompress` mais écrivant les codes en version « compacte ».

   ```c
   void compress(FILE *input_file, FILE *output_file);
   void decompress(FILE *input_file, FILE *output_file);
   ```

5. Tester le taux de compression obtenu pour :
   — le fichier source de votre code C d'aujourd'hui ;
   — l'énoncé du TP en format `pdf` ;
   — l'exécutable obtenu en compilant votre code source ;
   — le texte intégral de *Moby Dick* fourni avec le sujet.
   On pourra comparer :
   — les taux de compression obtenus pour différentes largeurs de code ;
   — le taux de compression obtenus en utilisant l'utilitaire `zip`.

5 Codes de largeur variable

Pour améliorer le taux de compression, une solution simple est d'utiliser des codes à largeur variable. En effet, en supposant que l'on fixe la largeur des codes à 14 bits par exemple, on va mettre assez longtemps à émettre le premier code ne rentrant pas sur 13 bits (*i.e.* 8192) : jusque là, le ou les bits les plus significatifs des codes émis valent tous zéro, et l'on gaspille donc de la place.

Pour éviter cela, il suffit de se mettre d'accord (entre la fonction de compression et celle de décompression) sur une règle pour l'évolution de la largeur du code. La règle la plus simple est la suivante :

— au départ, la largeur d'un code est 9 bits ;
— on se fixe une largeur maximale (disons 16 bits), et l'on crée les structures `dict` et `inverse_table` avec une taille correspondant à cette largeur ;
— dès que l'on souhaite créer une entrée pour un code et que ce code ne tient pas sur le nombre actuel de bits, on augmente la largeur de 1 si c'est possible (sinon, le dictionnaire est plein et l'entrée n'est pas créée) ;
— le nouveau code est créé au moment où l'on émet un code déjà existant (systématiquement) : on convient que le code existant est émis avec l'ancienne largeur, ce qui revient à dire que l'on crée l'entrée pour le nouveau code après émission de l'ancien ;
— pour la décompression, il faut juste penser que l'on a toujours « un temps de retard » sur la compression (pour l'état du dictionnaire), et qu'il faut donc changer de largeur une étape plus tôt.

1. Modifier la fonction `build_entry` pour qu'elle mette à jour la largeur des codes. Le comportement étant légèrement différent suivant que l'on est en train de compresser ou de décompresser, on ajoute un paramètre booléen `compress_mode` pour indiquer le mode de fonctionnement.

   ```
   void build_entry(cw_t cw, byte_t byte, bool compress_mode);
   ```

2. Après avoir apporté les autres modifications nécessaires à votre code (s'il y a lieu), reprendre les mesures de taux de compression et les comparaisons avec `zip`. On pourra aussi comparer avec la compression de Huffman que nous avons déjà programmé, et tester si la composée de Huffman et de LZW, dans un sens ou dans l'autre, présente un intérêt.