MA203 — TP Examen
Encodage et décodage du code BCH 2-correcteur

15 janvier 2016

1 Introdution

L'objectif de ce TP est de mettre en pratique deux méthodes d'encodage et un algorithme de décodage du code BCH 2-correcteur vu en TD4.

Consignes : Avant le vendredi 22 janvier minuit, vous devrez avoir envoyé par e-mail à l'adresse `julien.lavauzelle@inria.fr` :
- un fichier `main.c` qui contiendra vos fonctions implantées en `C` ;
- de manière facultative, un fichier `reponses.pdf` qui contiendra vos réponses aux questions bonus.

Remarquez que :
- seules les implantations correctes et **commentées** rapporteront le maximum de points ;
- une fonction qui ne compile pas se verra attribuer la note 0 (la commande utilisée pour la compilation étant `gcc -o main main.c util.c -W -Wall`, essayez aussi de minimiser le nombre de *warnings* et d'éviter les fuites-mémoire) ;
- des idées particulièrement efficaces se verront récompensées d'un bonus.

1.1 Notations et rappels

On pose $n = 15$ et $q = 16 = 2^4$ et on note $\mathbb{F}_q$ le corps fini à q éléments défini par le quotient $\mathbb{F}_2[x]/(x^4+x+1)$. On considère $\alpha \in \mathbb{F}_q$ une racine du polynôme primitif x^4+x+1, et on rappelle que α est donc un générateur du groupe cyclique $\mathbb{F}_q^*$. Les conjugués d'un élément $\beta \in \mathbb{F}_q^*$ sont $C(\beta) = \{\beta^{2^j},\, j \in \mathbb{Z}\}$. On note $\mu_\beta(X) \in \mathbb{F}_2[X]$ le polynôme minimal de β sur $\mathbb{F}_2$, et on rappelle que μ_β s'annule sur tous les conjugués de β.

Par ailleurs, on note $\mathrm{d_H} : (\mathbb{F}_2^n)^2 \to [0,\, n]$ la distance de Hamming définie par :

$$\mathrm{d_H}(x,\, y) = |\{i \in [0,\, n-1],\, x_i \neq y_i\}|\,,$$

et on appelle *poids (de Hamming)* de $x \in \mathbb{F}_2^n$ la quantité $\mathrm{w_H}(x) = \mathrm{d_H}(x,\, 0)$.

Enfin, pour $y \in \mathbb{F}_2^n$, on note $P_y = \sum_{i=0}^{n-1} y_i X^i \in \mathbb{F}_2[X]$ la représentation polynomiale de y.

1.2 Le code BCH 2-correcteur

Soit $\mathcal{M} = \mathbb{F}_2^k$ l'ensemble des messages binaires de taille k. Le code BCH 2-correcteur de longueur $n = 15$, noté $\mathcal{C}$, est l'ensemble des mots de code $c = (c_0, \ldots,\, c_{n-1}) \in \mathbb{F}_2^n$ dont la représentation

polynomiale s'annule sur les quatre premières puissances positives de α. Autrement dit,

$$\mathcal{C} = \{c \in \mathbb{F}_2^n,\ \forall i \in [1, 4],\ P_c(\alpha^i) = 0\}\,. \tag{1}$$

On peut alors montrer que l'ensemble $\mathcal{C}$ forme un sous-espace vectoriel de $\mathbb{F}_2^n$ de dimension $k = 7$.

Rappelons que l'utilisation courante d'un code correcteur est la suivante. Si l'on désire envoyer un message $m \in \mathcal{M}$ sur un canal bruité, on transforme de manière bijective m en un mot de code $c \in \mathcal{C}$. C'est ce mot c qui est envoyé sur le canal et peut y subir quelques altérations. À la reception de $y = c + e \in \mathbb{F}_2^n$, on espère ensuite retrouver c puis m au moyen d'un algorithme de décodage.

On appelle fonction d'encodage toute application $\phi : \mathcal{M} \to \mathcal{C}$ bijective. Bien évidemment, il existe beaucoup de telles fonctions ϕ ; le but de la section 2 est d'en implanter deux.

Par ailleurs, si le poids de l'erreur $\mathrm{w_H}(e) = \mathrm{d_H}(e, 0) \leq 2$, il est possible de retrouver n'importe quel mot de code $c \in \mathcal{C}$ à partir de $c + e$. La section 3 est ainsi consacrée à l'implantation d'un algorithme de décodage spécifique aux codes BCH.

Vous trouverez en annexe A une documentation succincte concernant les fonctions déjà fournies dans les fichiers `util.c` et `util.h`.

2 Encodage

2.1 Échauffement : fonctions annexes

Question 2.1. Écrivez une fonction `random_message` qui renvoit un message m tiré uniformément dans $\mathbb{F}_2^k$.

Question 2.2. Écrivez une fonction `poly_eval` qui renvoit l'évaluation d'un polynôme $Q \in \mathbb{F}_2[X]$ en un point $\beta \in \mathbb{F}_q$.

2.2 Première méthode : encodage polynomial

Soit $g(X) = \mu_\alpha(X)\mu_{\alpha^3}(X) = X^8 + X^7 + X^6 + X^4 + 1 \in \mathbb{F}_2[X]$. On peut alors remarquer que $\mathcal{C}$ est l'ensemble des mots c dont la représentation polynomiale $P_c(X)$ s'annule sur toutes les racines de $g(X)$. Ainsi, ces polynômes $P_c(X)$ doivent essentiellement être des multiples de $g(X)$. Plus précisément, en notant $\mathbb{F}_2[X]_{<k} = \{f \in \mathbb{F}_2[X],\ \deg f < k\}$, on peut montrer que :

$$\mathcal{C} = \{c \in \mathbb{F}_2^n,\ \exists a(X) \in \mathbb{F}_2[X]_{<k},\ P_c(X) = a(X)g(X)\}\,.$$

En particulier, la fonction

$$\begin{array}{rccll} \phi : & \mathcal{M} & \to & \mathcal{C} & \\ & m & \mapsto & c & \text{tel que } P_c(X) = P_m(X)g(X) \end{array}$$

est une fonction d'encodage du code $\mathcal{C}$.

Question 2.3. Écrivez une fonction `polynomial_encode` qui prend en paramètre un message m et calcule son encodage *via* cette première méthode.

2.3 Seconde méthode : encodage systématique

Lors du TD4, la fonction d'encodage suivante est proposée : pour $m \in \mathcal{M}$, le mot de code $c = \phi(m)$ est l'unique élement de $\mathbb{F}_2^n$ qui a pour représentation polynomiale :

$$P_c(X) = P_m(X)X^{n-k} + (P_m(X)X^{n-k} \mod g(X))\,.$$

Notez que $P_m(X)X^{n-k}$ est la représentation polynomiale du message m décalée de $n-k$ positions.

Question 2.4. Écrivez une fonction `remainder_encode` qui prend en paramètre un message m et calcule son encodage par la seconde méthode présentée ci-dessus.

3 Décodage

Question 3.1. Écrivez une fonction `add_noise` qui prend en paramètre $\tau \in \{0, 1, 2\}$ et un mot de code $c \in \mathcal{C} \subset \mathbb{F}_2^n$, et retourne le mot c bruité en τ position(s) tirée(s) uniformément dans $[0, n-1]$.

L'algorithme de correction vu en TD4 est un cas simplifié de l'algorithme de Peterson-Gorenstein-Zierler. On note $e_i \in \mathbb{F}_2^n$ le vecteur nul partout sauf en la coordonnée i où il vaut 1.

Algorithme 1 : Algorithme de Peterson-Gorenstein-Zierler simplifié pour $\tau \leq 2$

Entrée : un mot $y \in \mathbb{F}_2^n$ tel que $y = c + e$ où $c \in \mathcal{C}$ et $\mathrm{w_H}(e) = \tau \leq 2$.
Sortie : le mot de code $c \in \mathcal{C}$

Calculer $S_1 = P_y(\alpha)$ et $S_3 = P_y(\alpha^3)$.
Si $S_1 = S_3 = 0$ **alors**
    Retourner y.
Sinon
    Si $S_3 = S_1^3$ **alors**
        Trouver $i \in [0, n-1]$ tel que $S_1 = \alpha^i$.
        Retourner $y + e_i$.
    Sinon
        Trouver les deux racines distinctes x_1, x_2 de $Q(X) = S_1X^2 + S_1^2X + S_1^3 + S_3$.
        Trouver i, $j \in [0, n-1]$ tels que $x_1 = \alpha^i$ et $x_2 = \alpha^j$.
        Retourner $y + e_i + e_j$.

Question 3.2. Implantez une fonction `solve_quadratic_equation` qui prend en argument un polynôme $Q \in \mathbb{F}_2[X]$ de degré 2 ayant exactement deux racines distinctes dans $\mathbb{F}_q$, et qui renvoit ses deux racines. *Indice : une recherche exhaustive sur $\mathbb{F}_q$ fonctionne très bien lorsque q n'est pas très grand.*

Question 3.3. Implantez l'algorithme simplifié de Peterson-Gorenstein-Zierler dans une fonction `pgz_decode`.

Question finale. Écrivez une procédure principale `main` qui tire un message aléatoirement, l'encode, bruite le mot de code associé puis le corrige à l'aide de l'ensemble des fonctions écrites précédemment. Affichez les étapes en sortie standard.

4 Questions subsidiaires

Les questions qui suivent sont facultatives. Elles sont un complément théorique aux questions d'implantation précédentes qui, elles, sont obligatoires. **Ne répondez à ces questions que si vous avez terminé TOUTES les questions précédentes !**

Question bonus (i) - Quel avantage présente la seconde méthode d'encodage par rapport à la première ?

Question bonus (ii) - Quelle est la complexité (fonction de n et k) en temps et mémoire de l'algorithme de décodage proposé ?

A Documentation

Dans les fichiers `util.c` et `util.h` sont fournies les structures et fonctions de base qui vous permettront de répondre aux questions du TP. En voici une courte documentation :

- `GF2` et `GF16` sont des redéfinitions du type `unsigned int` qui permettent de représenter un élément des corps finis $\mathbb{F}_2$ et $\mathbb{F}_{16}$. Naturellement, 0 et 1 (éléments de $\mathbb{F}_2$ ou $\mathbb{F}_{16}$) sont représentés par les entiers non signés 0 et 1. Plus généralement, l'élement $\beta = \sum_{i=0}^{3} b_i \alpha^i \in \mathbb{F}_{16}$ est représenté par $\sum_{i=0}^{3} b_i 2^i$.
- `poly` est une structure représentant un polynôme sur $\mathbb{F}_2$. Il a pour champs son degré `deg` et ses coefficients `coeff`.
- `unsigned int Log16[16]` est la table des logarithmes en base α des éléments de $\mathbb{F}_{16}$. Par exemple, $\alpha + \alpha^2 = \alpha^5$, donc `Log16[6] == 5`. Par convention, `Log16[0] == 15`.
- `GF16 Exp16[16]` est la table des exponentiations en base α. Par exemple, $\alpha + \alpha^2 = \alpha^5$, donc `Exp16[5] == 6`. On a donc `Exp16[Log16[i]] == i` pour tout entier non signé `i` $\in [0, 15]$ (attention, par convention, `Exp16[15] == 0`).
- Les fonctions `GF2 add2(GF2 a, GF2 b)` et `GF2 mul2(GF2 a, GF2 b)` renvoient respectivement la somme et le produit d'éléments de $\mathbb{F}_2$.
- `GF16 add16(GF16 a, GF16 b)`, `GF16 mul16(GF16 a, GF16 b)` et `GF16 div16(GF16 a, GF16 b)` retournent la somme, le produit et le quotient de deux élement de $\mathbb{F}_{16}$. `GF16 inv16(GF16 a)` renvoit l'inverse de `a` $\in \mathbb{F}_{16}$ et `GF16 pow16(GF16 a, unsigned int n)` renvoit $\texttt{a}^{\texttt{n}}$.
- `poly* poly_init()` initialise un polynôme et renvoit un pointeur vers la structure associée.
- `unsigned int poly_coeff(poly* a, int deg)` retourne le coefficient de degré `deg` du polynome `a`.
- `void poly_add(poly* res, poly* a, poly* b)` assigne à `res` la somme `a+b`.
- `void poly_mul(poly* res, poly* a, poly* b)` assigne à `res` le produit `a*b`.
- `void poly_gcd(poly* res, poly* a, poly* b)` assigne à `res` le pgcd des polynômes `a` et `b`.
- `void poly_div(poly* res, poly* a, poly* b)` assigne à `res` le quotient de la division euclidienne entre `a` et `b`.
- `void poly_mod(poly* res, poly* a, poly* b)` assigne à `res` le reste de la division euclidienne entre `a` et `b`.
- `void poly_free(poly* a)` libère la mémoire occupée par `a`.
- `void poly_null(poly* a)` assigne à `a` le polynôme nul.
- `void poly_add_monomial(poly* res, GF16 coeff, int deg)` ajoute au polynôme `res` le monôme $\texttt{coeff} \times X^{\texttt{deg}}$.
- `void vector_copy(GF2* res, GF2* vect, int len)` copie le vecteur `vect` de longueur `len` dans `res`.
- `void vector_to_poly(GF2* vect, int len, poly* p)` assigne à `p` la représentation polynomiale de `vect` de longueur `len`.
- `void poly_to_vector(GF2** vect, int len, poly* p)` renvoit dans le pointeur vers `*vect` le mot associé au polynôme `p`.
- `void poly_print(poly* a)` affiche `a` de façon lisible.
- `void vector_print(GF2* vect, int len)` affiche le vecteur `vect` de longueur `len`.