Université de Strasbourg

Projet de réseau Locaux : Hadamard & Simplex

Rapport

Nathanaël Derousseaux--Lebert

Question n°1

La matrice est génératrice car elle commence par la matrice identité. Sur 16 bits, 4 bits encodent le message et les 12 bits sont des bits de contrôle. On a donc un code avec un rendement de $4/16 = 0{,}25$.

Question n°4.2

La distance de hamming de ce code de Hadamard est de $n = 8$ Cela signifique qu'il peut détecter $n - 1$ erreurs soit 7, et en corriger $\left\lfloor \frac{n-1}{2} \right\rfloor$ soit 3.

Question n°4.3

On constate rapidement que, le mot vide mis à part, la distance de Hamming de chaque mot du code est égale à 8.

Question n°5

La colonne nulle ne sert à rien, car lors de la multiplication de matrice, le résultat sera toujours le mot vide.

Question n°6

En effet, les fonctions précédentes donnent le même résultat, soit une distance de 8.

Question n°8

Pour calculer la matrice de parité d'une matrice génératrice systématique telle que $G = [I \,|\, A]$ alors la matrice de parité est $H = [A^T \,|\, I]$.

Donc pour la matrice de parité de G_s on a :

$$H_s = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Le dual du code simplexe est lui-même, car en transposant H_s , on retombe sur G_s.

Question n°10

1. Comme attendu, si l'on calcule le syndrome d'une même erreur sur plusieurs mots différents on remarque que le syndrome est toujours identique. C'est cela qui va nous permettre de corriger l'erreur. Car pour une erreur, nous avons un unique syndrome correspondant.

2. Déduire l'erreur en fonction du syndrome n'est pas triviale. La méthode souvent utilisée est de pré-calculer tout les couple erreur/syndrome. Ensuite, lorsque l'on veut corriger une erreur, on regarde dans la liste des syndrome l'erreur associée.

J'ai généré une longue liste d'erreur possibles (entre 1 et 3 bits modifiés) et leur syndrome associé avec la fonction `compute_syndromes`. J'ai mis ces listes dans le fichier `utils.c`.

Ensuite, ma fonction décode va calculer le syndrome du mot à décoder. Si celui-ci est présent dans ma liste, on peut décoder le mot. Si il ne l'est pas, mais n'est pas nul, on sait qu'il y a eu une erreur, mais on ne peut pas la corriger.

Une autre manière de faire aurait pu être d'exécuter la fonction `compte_syndrome` au lancement du programme et de stocker la liste erreur/syndrome en mémoire.

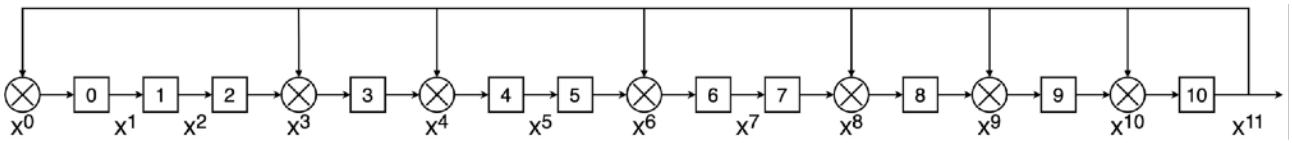
Afin de réduire la taille de cette liste, on aurait aussi pu calculer les erreurs portant uniquement sur les 4 premiers bits, car seuls eux codent de l'information. Cependant, afin d'avoir un code plus complet, permettant de corriger l'ensemble de mot codé, j'ai préféré calculer toutes les erreurs possibles.

3. Ici, on peut voir que la fonction décode fonctionne jusqu'à 3 erreurs. À la quatrième erreur, on se trouve exactement à égale distance entre deux mots valide du code.

Cependant, au dessus de 4 erreurs, il n'est pas impossible que l'on se rapproche d'un autre mot valide. Plus spécifiquement, avec 8 erreurs, il n'est pas impossible de retomber exactement sur un autre mot valide.

```c
int main(int argc, char *argv[]){
    unsigned short mot = 0b1010000000000000;
    mot = encode_simplexe(mot);

    mot = chg_nth_bit(0, mot); //0b0010101110011000
    decode(mot_erreur) //0b1010101110011000 //OK

    mot = chg_nth_bit(1, mot); //0b0110101110011000
    decode(mot_erreur) //0b1010101110011000 //OK

    mot = chg_nth_bit(2, mot); //0b0100101110011000
    decode(mot_erreur) //0b1010101110011000 //OK

    mot = chg_nth_bit(3, mot); //0b0101101110011000
    decode(mot_erreur) //0b0101101110011000 //KO

    //Si on cause 8 erreurs bien précises
    mot = chg_nth_bit(7, mot);
    mot = chg_nth_bit(10, mot);
    mot = chg_nth_bit(12, mot);
    mot = chg_nth_bit(13, mot);
    get_syndrome(mot) //0b00000000000

    return 0;
}
```

Question n°11



Question n°13

À l'instar de la question 10, on peut voir dans cet exemple que la correction d'erreur fonctionne bien jusqu'à trois erreurs. Une quatrième et la correction ne fonctionne plus.

```c
int main(int argc, char *argv[]){
    unsigned short mot = 0b1001000000000000;
    mot = code_poly(mot);

    mot = chg_nth_bit(0, mot); //0b0001000011110101
    decode_poly(mot); // 0b1001000011110101 //OK

    mot = chg_nth_bit(1, mot); //0b0101000011110101
    decode_poly(mot); // 0b1001000011110101 //OK

    mot = chg_nth_bit(2, mot); //0b0111000011110101
    decode_poly(mot); // 0b1001000011110101 //OK

    mot = chg_nth_bit(3, mot); //0b0110000011110101
    decode_poly(mot); // 0b0110000011110101 //KO

   return 0;
}
```

Question n°14