Compte rendu code détecteurs et correcteurs d'erreurs

Bazan Clément, Benjelloun Youssef, Olivier Tom

20 mars 2022

Table des matières

1 Générer tous les mots possibles de longueur k.

Pour répondre à cette question on crée une fonction d'aide qui transforme un entier en une liste de bits de longueur *length* (voir algorithme 1).

Algorithme 1 – Code générateur

```
def int_to_bin_lst(num : int, length : int):
    """
     Transform num in a bits list with length size
    """
    l = []
    while num:
        l.append(num&1)
        num >>= 1
    # Add zeroes to fill the list
    while (len(l) < length):
        l.append(0)
    l.reverse()
    return l
```

Nous utiliserons l'algorithme 1 pour rendre plus clair l'écriture de l'algorithme 2.

Algorithme 2 – Génerateur

```
    def all_binary_words(k: int):
        res = []
        for i in range(2**k):
            res.append(int_to_bin_lst(i))
        return res
```

À titre d'exemple pour $k = 4$ on obtient :

$$V_4 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

2 Générer le code de chaque mot possible.

Pour pouvoir encoder notre message on doit manipuler des matrices pour cela on implante des algorithmes élémentaires comme la multiplication de matrices et la transposition.

Algorithme 3 – Produit matriciel et tranposition

```
def mult_matrix(H, G):
    """
     matrix multiplication H * G
    """
    lines_h = len(H)
    cols_h = len(H[0])
    lines_g = len(G)
```

```
    cols_g = len(G[0])
    assert(cols_h == lines_g)

    C = []
    i = 0
    while (i < len(H)):
        j = 0
        l = []
        while (j < len(G[0])):
            k = 0
            res = 0
            while(k < lines_g):
                res ^= H[i][k] * G[k][j]
                k += 1
            l.append(res)
            j += 1
        C.append(l)
        i += 1
    return C
def transpose(matrix):
    """
     Transpose of matrix
    """
    res = []
    for i in range(len(matrix[0])):
        new_line = []
        for j in range(len(matrix)):
            new_line.append(matrix[j][i])
        res.append(new_line)
    return res
```

Pour la génération du code il suffit de réaliser une multiplication matrice vecteur. On définit deux matrices génératrices G_1 et G_2 (en forme canonique) comme suit :

$$G_1 = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad G_2 = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

La matrice de contrôle associée à G_1, H_1 est :

$$H_1 = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

La matrice de contrôle associée à G_2, H_2 se calcule comme suit $(-P^t \quad | \quad I_{n-k})$. Cela donne la matrice de contrôle H_2 définit telle que :

$$H_2 = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Désormais pour générer les codes il suffit de multiplier la matrice V_4 par la matrice G, on obtient :

$$C_1 = V_4 \cdot G_1 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \qquad C_2 = V_4 \cdot G_2 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

3 Comparer les codes générés avec G_1 et avec G_2 (en forme canonique)

On remarque que les 4 premiers colonnes de C_2 sont les messages envoyés en clair. Également les 4 derniers colonnes de C_1 sont les messages en clair. En effet, ces colonnes correspondent à la sous-matrice identité de taille 4 en G_1 et G_2.

4 Vérifier que $C \cdot H = 0$.

Pour vérifier cela il faut d'abord transformer le vecteur ligne en vecteur en forme de colonne. La matrice H étant un matrice (3x7) lorsque on transpose la matrice C on

obtient une matrice (7x16) ce qui nous permet de multiplier H par C^t.

$$H \cdot C^t = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Remarquons que tous les syndromes sont à zéro, donc aucune erreur n'a été détecté.

5 Corriger les erreurs

Pour corriger les erreurs, on choisit d'abord aléatoirement des lignes dans C où l'on change aléatoirement un bit ou non.

Algorithme 4 – Injection d'erreur

```
def injection(nb_lines, matrix):
    """
    Takes nb_lines randmoly of matrix and change a bit randomly also
        those lines
    """
    chosen_lines = [-1]*nb_lines
    i = 0
    while i < nb_lines:
        rand = rd.randint(0, len(matrix)-1)
        if rand not in chosen_lines:
            chosen_lines[i] = rand
            i+=1
    for i in range(len(chosen_lines)):
        matrix[chosen_lines[i]][rd.randint(0, len(matrix[0])-1)] = rd.
            randint(0,1)
    return matrix
```

Lorsqu'on injecte des erreurs et que l'on multiplie par la matrice de contrôle on obtient des vecteurs qui ne sont pas à zéro. Par exemple supposons le message encodé m :

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

Lors que on injecte une erreur, le message m devient

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

Lors que on calcule le syndrome, on obtient $\begin{pmatrix} 0 & 1 & 0 \end{pmatrix}$. L'algorihme 5 suivant calcule le mot correcteur dans notre cas $\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$ et réalise la somme dans $\mathbb{F}_2$.

Algorithme 5 – Correction d'erreur

```
    def correction(syndrome, msg, control_matrix):
        control_matrix = transpose(control_matrix)
        correction_word = []
        for vec in control_matrix:
            if syndrome == vec:
                correction_word.append(1)
            else :
                correction_word.append(0)
        return xor(msg, correction_word)
```

En réalisant la somme on retrouve le bon message et, lorsque l'on multiple par la matrice H le contrôle d'erreurs est revenu a 0.

6 Décoder les messages

Pour décoder les messages il suffit de détecter la sous-matrice identité dans G et, préserver les bits utiles dans le message encodé.

Algorithme 6 – Décoder un message

```
def decode(encoded_msg, matrix):
    decoded_msg = [0]*len(matrix)
    for i in range(len(matrix[0])):
        nb_one = 0
        found = (None, None)
        for j in range(len(matrix)):
            if (matrix[j][i]) :
                nb_one += 1
                found = (j, i)
            if nb_one > 1:
                break;
        if nb_one == 1:
            decoded_msg[found[0]] = encoded_msg[0][found[1]]
    return decoded_msg
```