CODES CORRECTEURS D'ERREURS

RAPPORT DE PROJET : CODE LDPC

Auteur : *Shérif MAKALOU*

1- Première tâche : l'encodage

- Exercice 4 :

```java
public Matrix sysTransform(){

    for (int i = 0, j = cols-rows; i < rows && j < cols; i++, j++) { //première partie: échange des lignes
        for (int k = i; k<rows; k++){
            if (getElem(k, j) == (byte) 1){
                shiftRow(i, k);
                break;
            }
        }
        for (int p= 0; p<rows; p++){ //addition des lignes afin d'obtenir une matrice systématique
            if (getElem(p, j) == (byte) 1 && p!=i){
                addRow(i, p);
            }
        }
    }
    return this;
}
```

L'objectif de cet exercice était d'écrire une fonction **sysTransform** capable de mettre une matrice de contrôle H=[R|L] sous forme systématique, donc sous la forme H=[R|Id]. Pour ce faire, mon code se déroule en deux étapes principales :

1- Première étape : L'objectif de cette étape est de mettre des **1** sur toute la diagonale de la matrice L.
Pour ce faire, pour chaque ligne d'indice i, je regarde toutes les lignes d'indice k en dessous de la ligne i, et dès que je trouve un **1** au début d'une ligne k (grâce à la fonction **getElem**), je l'échange avec la ligne i grâce à la fonction **shiftRow(i,k)** en s'assurant bien de ne plus examiner les lignes d'indice >k si l'échange a déjà été fait(d'où l'utilité du **break** après l'appel à shiftRow). Dans la boucle **for** globale, **j'itère à la fois sur i et j** afin de descendre de manière diagonale sur la matrice L. Ma variable **j** est initialisée à **cols-rows** pour les colonnes afin de bien traiter la partie **L** de H et non la matrice R.

2- Deuxième étape : Cette étape sert à transformer la matrice L en une matrice identité. Et donc, de mettre H sous forme systématique. L'idée est de mettre des 0 partout dans L, sauf sur la diagonale où se trouvent déjà les 1.
Pour ce faire, j'additionne chaque ligne de L avec toutes les autres afin d'éliminer les 1. Ce sont les **1 sur les diagonales** qui, après addition, éliminent tous les autres 1 de la matrice. Pour éviter d'additionner une ligne avec elle-même, je précise que

p != i. Le résultat suivant est obtenu en utilisant la matrice H de taille (15,20) fournie.

```
Forme systématique:
[[1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
 [1 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
 [1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]]
```

2-Deuxième tâche : le décodage

Exercice 8 : L'objectif était de décoder des mots de code afin de les corriger s'ils contiennent des erreurs.

Pour ce faire, j'ai utilisé l'algorithme vu en cours.

1- Initialisation :

```
//initialisation
for(int i=0; i<n_c;i++) {

    right[i][0] = code.getElem(i:0, i);

}


int verif = 0;
byte [][] tab = new byte[1][code.getCols()];
Matrix x;
int [] count = new int[code.getCols()];
```

Ici j'initialise la première colonne tableau **right** qui correspond au tableau des **nœuds variables** par les valeurs du mot de code **y** que l'on souhaite corriger. J'en profite pour déclarer quelques variables dont j'aurais besoin dans la suite.

2- Calcul des parités :

```
//Boucle principale
for(int i=0; i<rounds; i++) {

    //Calcul des parités
    for(int j=0; j<n_r; j++) {

        left[j][0] = 0;

        for(int k=1; k<w_r+1; k++) {
            left[j][0]= (left[j][0] + right[left[j][k]][0])%2;

        }
    }
```

Ici on initialise d'abord tous les nœuds fonctionnels à 0 dans le tableau left. Ensuite on additionne à chaque nœud fonctionnel, ses nœuds voisins, donc les nœuds auxquels il est relié dans le graphe de Tanner. Pour ce faire, il faut remarquer que les **voisins** d'un nœud fonctionnel **left[j][0]** sont les **right[left[j][k]][0]** avec 0<k<w_r + 1. On n'oublie pas l'addition modulo 2 car on ne veut que des 1 et des 0. Ainsi on arrive à calculer les parités sur les nœuds fonctionnels du graphe.

3- Vérification :

```
    //Vérification
    for(int k=0; k<n_r; k++) {
        if(left[k][0] != 0) {
            verif = 0;
            break;
        }
        verif = 1;
    }

        if(verif==1) {


            for(int k=0; k<n_c; k++) {
                tab[0][k] = (byte)right[k][0];

            }

            x = new Matrix(tab);
            return x;

        }
```

Cette étape consiste à vérifier si on a un mot de code. En effet, il est inutile de dérouler la suite de l'algorithme si le mot a été corrigé. Nous savons que nous avons un mot de code quand

tous les nœuds fonctionnels du graphe de Tanner sont à 0. Dans ce cas, il suffit de récupérer les valeurs des nœuds variables pour obtenir notre mot de code.

Pour ce faire, j'utilise ici une variable **verif** qui me permettra de savoir si j'ai un mot de code ou non (0 pour non, et 1 pour oui). Ainsi, dès qu'un des nœuds fonctionnels a une valeur différente de 0, on en déduit que le mot contient encore des erreurs, et on quitte complètement la boucle de vérification grâce à l'instruction **break**. Dans le cas contraire, on a un mot de code, donc **verif=1.** Et là, on donne à x les valeurs des nœuds variables (contenues dans right) qui forment le mot corrigé.

4- Calcul du max :

```
    //Calcul du max
    int max = 0;
    for(int k=0; k<n_c; k++) {
        count[k] = 0;

        for(int l=1; l<w_c+1; l++) {
            count[k] = count[k] + left[right[k][l]][0];
        }

        if(count[k]> max) {
            max = count[k];
        }

    }
```

Cette partie est traitée si on n'a pas obtenu de mot de code après la première étape. L'objectif est de trouver les bits du mot qui faussent le plus d'équations de parité. On utilise alors une liste **count** qui se chargera de stocker **le nombre d'équations de parité non vérifiées** pour chaque nœud variable. Les équations non vérifiées sont les nœuds voisins dont les valeurs sont à **1** dans les nœuds fonctionnels. Il suffit alors d'additionner les valeurs des nœuds afin d'obtenir le nombre d'équations non vérifiées. Cela est nécessaire pour pouvoir faire le **renversement de bits**.

5- Renversement de bits :

```
//Renversement de bits
for(int k=0; k<n_c;k++) {
    if(count[k] == max) {

        right[k][0] = 1 - right[k][0];

    }
}
```

Une fois qu'on a identifié le bit qui fausse le maximum d'équations de parité, on peut procéder au renversement de bits en changeant la valeur du bit faussé identifié. Une

fois l'opération effectuée, on retourne au début de la boucle principale pour effectuer les mêmes opérations que tout à l'heure (Calcul de parités, vérification …).
A la fin de la boucle principale, si on n'arrive toujours pas à trouver un mot de code, on renvoie un vecteur d'erreur pour traduire l'erreur de décodage.

```
        //Si aucune valeur n'est retournée, échec
        byte[][] err = new byte[1][n_c];
        for (int i=0; i<n_c; i++){
            err[0][i] = -1;
        }
        Matrix error =new Matrix(err);
        return error;

    }
```

3- Troisième tâche : l'évaluation

Exercice 9 : Les mots y1, y2 et y3 ont été correctement corrigés mais la correction a échoué pour y4.

Ceci est peut-être dû au fait que e4 est un **vecteur d'erreur de poids 3**. L'algorithme serait alors **2-correcteur**.

Exercice 10 :

H est une matrice de contrôle de 2048 lignes et 6144 colonnes.

Donc **n=6144**. On sait que rang(H) = n-k.

H étant une **matrice de rang plein**, les colonnes de la matrice ne sont pas linéairement dépendantes d'où le rang de H sera le nombre de lignes de la matrice => **rang(H) = 2048**. Ainsi **k** = n - rang(H) = 6144 - 2048 = **4096.**

Redondance = n-k = 6144 – 4096 = **2048**

Rendement = k/n = 4096/6144= **2/3**

Nombre moyen d'erreurs reçues :

Pour **$p=2.10^{-2}$:**

Nb = p * n = 123 erreurs

Pour **$p=2.5.10^{-2}$:**

Nb = p * n = 153 erreurs

Le nombre moyen d'erreurs augmente beaucoup pour une faible augmentation de la probabilité d'erreurs.