Collège Sciences et Technologies
UF Mathématiques et Interactions - Informatique

Codes LDPC

Corentin Banier
Maher Karboul

Licence Mathématiques Informarique

2020 – 2021

Projet tutoré

Table des matières

1 Introduction

La conception des codes LDPC binaires avec un faible poids d'erreurs demeurre un problème non entiérement résolu. Les codes LDPC (Low Density Parity Check) sont des codes linéaires correcteurs d'erreurs qui assurent la transmission d'informations. Ils forment une classe de codes en bloc qui se caractérisent par une matrice de contrôle creuse. Ils ont été décrits pour la première fois dans la thèse de Gallager au début des années 60. Dans ce travail, nous allons étudier comment fabriquer des instances de ce code notamment avec le modèle de Gallager. Puis, nous comprendrons comment décoder des codes LDPC, on essaiera d'optimiser ce dernier en limitant le nombre d'équation à satisfaire par un mot de code erroné.

Le rapport est organisé de la manière suivante ; la section 3 présente les étapes de fabrication d'un code LDPC ainsi qu'une matrice de contrôle qui répond à des conditions bien spécifiques. La section 5 présente l'algorithme détaillé de décodage LDPC. La section 6 est dédiée aux expériences pratiques qu'on a effectué durant l'implémentation de l'algorithme. Enfin, la section 7 sera notre conclusion sur le projet.

Avant tout, nous allons introduire la notion de code correcteur d'erreur(s).

PS : L'ensemble du code réalisé durant le projet est disponible sur le dépôt GitHub à l'adresse suivante :
`https://github.com/cbanier/codes-LDPC`
Nous utilisons le module NUMPY.

2 Codes correcteurs

Lors de la transmission d'une information, des erreurs peuvent se produirent. Cette problématique de correction des erreurs de transmission est très importante dans notre monde connecté, qu'il s'agisse des communications entre ordinateurs par internet, des conversations téléphoniques etc...
Un code correcteur, souvent désigné par le sigle anglais ECC (Error-correcting code), est une technique de codage basée sur la redondance.
Un code est une application injective $\Phi : \{0,1\}^k \to \{0,1\}^n$.
Le paramètre k est appelé la **dimension** du code Φ et le paramètre n est appelé la **longueur** du code : on dit que Φ est un code de paramètres (k,n).

Soit Φ un code d'image C.
On appelle **capacité de correction** de Φ le plus grand entier e_c tel qu'on soit toujours capable de corriger e_c erreurs ou moins.
On appelle **distance minimale** de Φ et on note d_c la plus petite distance non nulle entre deux mots de code.
Ainsi, on a $e_c = \frac{d_c-1}{2}$

Parmi les exemples de codes correcteurs, on peut citer les codes de répétition, les codes carré et on encore les codes LDPC.

2.1 Code de répétition

Le code de répétition se résume par transmettre le message deux fois pour s'assurer contre les erreurs. Par exemple, Alice veut transmettre un mot de quatre bits à Bob.

m=0111

Elle va donc envoyer le mot codé

c=01110111

le mot reçu par Bob sera noté y. Par exemple, si Bob reçoit

y=01110110

Il peut constater qu'une erreur, au moins, s'est produite. Il peut dire que l'erreur est soit sur le quatrième ou le huitième bit.

2.2 Code carré

On suppose que Alice veut transmettre le mot de quatre bits, **m**=$x_1x_2x_3x_4$. Pour cela, elle l'écrit dans un carré de la façon suivante

$$\begin{matrix} x_1 & x_2 \\ x_3 & x_4 \end{matrix}$$

Elle rajoute ensuite les bits, dits de parité, p_1,p_2,p_3,p_4 de sorte qu'il y ait un nombre pair de 1 sur chaque ligne et chaque colonne.
La somme sur chaque ligne et chaque colonne est 0 modulo 2.
Le code sera affiché de cette façon :

$$\begin{matrix} x_1 & x_2 & p_1 \\ x_3 & x_4 & p_2 \\ p_3 & p_4 & \end{matrix}$$

Le mot codé envoyé par Alice est :

c=$x_1x_2x_3x_4p_1p_2p_3p_4$

On prend l'exemple suivant : **m=0011**, la première étape est :

0 0
1 1

En rajoutant les bits de parité, on obtient

0 0 0
1 1 0
1 1

Le mot codé est donc **c=00110011**

2.3 Code linéaire

Définition : Soient $\mathbb{F}_q$ un corps fini à q éléments, n⩾1 un entier. On dit que C $\subset \mathbb{F}_q{}^n$ est un code linéaire si C est un sous-espace vectoriel de $\mathbb{F}_q{}^n$. Comme tout espace vectoriel, C a une dimension k.
La construction de ce type de code est : $\phi : \mathbb{F}_q{}^k \to \mathbb{F}_q{}^n$. D'où C=Im$\phi$ est un sous espace-vectoriel de $\mathbb{F}_q{}^n$, et par le théorème du rang dimC = k.
Exemple de code linéaire : Le code carré.

2.4 Matrice génératrice

Définition d'une matrice génératrice : Soit C $\subset \mathbb{F}_q{}^n$ un code linéaire de dimension k. Une matrice G dont les lignes forment une forme de C s'appelle **matrice génératrice** de C. Elle aura donc k lignes et n colonnes.
G est de la forme :

$$\mathrm{G} = (\mathrm{Id}_k.\mathrm{A})$$

Exemple : Le code carré : On rappelle que c'est l'image de l'application linéaire injective

$$\phi : \mathcal{F}_2^4 \to \mathcal{F}_2^8$$
$$(x_1, x_2, x_3, x_4) \mapsto (x_1, x_2, x_3, x_4, x_1 + x_2, x_3 + x_4, x_1 + x_3, x_2 + x_4)$$

Une base de l'image est l'image d'une base. Par exemple l'image de la base canonique.
Donc ϕ(1000) = **10001010**
ϕ(0100) = **01001001** .
De même pour ϕ(0010)=**00100110**
Et aussi ϕ(0001) = **00010101**
On obtient donc une matrice génératrice

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

2.5 Matrice de contrôle

Une matrice de contrôle d'un code C est une matrice H de dimension n*(n-k) tel que :

$$\mathrm{x} \in \mathrm{C} \iff \mathrm{H}.\,{}^t x = 0$$

C'est une matrice d'une application linéaire surjective ϕ de **A** dans un espace vectoriel ayant pour noyau le code C.
⇒ L'ensemble d'arrivée de l'application linéaire ϕ associée à la matrice de contrôle est de dimension n-k (d'après le théorème du rang)

Exemple de calcul de matrice de contrôle à partir de la matrice génératrice :
Soit C le code linéaire qui a pour matrice génératrice

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

⇒ On échelonne et on réduit la matrice génératrice jusqu'à ce qu'on obtient une matrice de la forme suivante :

$$G' = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

La règle qui s'applique à ce stade : une matrice de contrôle de C est de la forme suivante

H= (- ${}^tA$.Id$_{n-k}$)

⇒ On applique donc cette règle et on obtient :

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

2.6 Syndrome

Soient C un code linéaire et H une matrice de contrôle de C. Un mot x de C est transmis. y est le mot reçu.
On pose r = y - x l'erreur de transmission. Ce qui implique que y = r + x
On a donc :

$$\text{H } {}^ty = \text{H } {}^t(x+r) = \text{H } {}^tx + \text{H } {}^tr = \text{H } {}^tr$$

car H ${}^tx$ =0 vu que x est un mot de C. Donc la méthode de décodage par syndrome permet de repèrer les indices érronés dans un code reçu à l'aide de la matrice de contrôle.

3 Comment fabrique-t-on un code LDPC ?

Les codes LDPC ont été découverts par Gallager dans les années 60, cependant, ce dernier a seulement proposé une méthode générale pour construire des codes LDPC pseudo-aléatoire.
Les longs codes LDPC sont générés par des ordinateurs et leurs décodage est complexe. Ceci est notamment dû au manque de structure. Tanner, en 1981, a donné une nouvelle interprétation d'un point de vue graphique qui a contribué au décodage itératif des codes LDPC.

Mais ce n'est que dans les années 2000 que les chercheurs Lin, Kou et Fossorier élaborent une construction algébrique et systématique des codes LDPC sous les géométrie finies. La construction et le décodage des codes LDPC peuvent être fait de plusieurs manières. Un code LDPC est caractérisé par sa matrice de parité.

DÉFINITION : Un code LDPC régulier est défini comme l'espace nul d'une matrice de contrôle de parité H, qui a les propriétés suivantes :

1. Chaque ligne et colonne contient un nombre bien défini de 1.
2. Ce nombre là a une valeur petite en comparaison avec la longueur du code et avec le nombre de lignes de H.

DÉFINITION : Une matrice **H** est dite creuse si elle possède une faible densité de 1.

REMARQUE : Dans le cas où toutes les colonnes ou toutes les lignes de H n'ont pas le même poids, le code LDPC est dit irrégulier.

CONSTRUCTION : La construction d'un code LDPC binaire revient à attribuer un petit nombre de 1 dans une matrice essentiellment composé de 0. Il existe plusieurs méthodes afin de construire de bons codes LDPC. Elles se distinguent en deux classes :

1. La construction aléatoire
2. La construction structurelles

La première classe de construction est basée sur des géométries finies, c'est à dire à l'aide d'un nombre fini de point. La seconde repose sur la notion des matrices de permutations circulantes.

3.1 Construction des codes LDPC de Gallager

Afin de construire la matrice de parité **H** d'un code LDPC de Gallager, il faut d'abord construire une sous matrice H_i ayant un poids de colonnes égal à 1 et un poids de lignes γ. Ensuite on doit trouver des permutations de colonnes de cette sous-matrice afin de former les autres sous-matrices avec lesquels on forme la **matrice de Gallager** de la manière suivante :

$$H = \begin{bmatrix} H_1 \\ H_2 \\ H_3 \\ \vdots \\ H_n \end{bmatrix}$$

Lorsqu'on choisit les permutations de colonnes des sous-matrices il faut faire attention à garder une bonne distance minimale de la matrice de parité **H**.

3.2 Exemple de construction de Gallager

Les lignes de contrôle de parité des matrices de Gallager sont divisées en ensemble w_c avec $\frac{M}{w_r}$ lignes dans chaque série. Le premier ensemble de lignes contient w_r nombre de '1' consécutifs ordonnés de gauche à droite à travers les colonnes, ce qui veut dire que pour i $\leqslant \frac{M}{w_r}$, la i$^{\text{ème}}$ ligne n'est pas nulle de la ((i-1)+1$^{\text{ème}}$) jusqu'à la i$w_r{}^{\text{ème}}$ colonne).
Par conséquent, toutes les colonnes de **H** comportent un seul '1' dans chacun des ensembles w_c.

Exemple : Une matrice de controle de parité régulière(Gallager) tels que : M=10 (colonnes), w_c=3, w_r= 5

$$H = \left[\begin{array}{cccccccccc} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ \hdashline 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ \hdashline 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{array}\right]$$

3.3 Robert Gray Gallager

Robert Gray Gallager né en 1931 est un ingénieur américain en éléctricité qui consacré sa vie à travailler dans la théorie de l'information et les réseaux de communication.
Gallager a reçu son Bachelor of Science Electrical Engineering (BSEE) degree de l'université de Pennsylvanie en 1953. Durant cette époque, il était membre du **laboratoire téléphonique Bell** puis il a servi pour l'une des branches de l'armée des Etats-Unis qui crée et gère les communications et les systèmes d'information qui est le **Corps des transmissions de l'armée des Etats-Unis (USASC)**.
En 1957, il a décroché sont Master en Sciences (S.M) et en 1960, il est devenu docteur en sciences spécialisé en ingénierie éléctrique.
Sa thèse de doctorat portait sur les codes de contrôle de parité de basse densité (LDPC) qui a été publié comme un travail d'écriture spécialisé (Monographie) par la presse universitaire de Massachusetts Institute of Technology (MIT) en 1963. Ces **codes LDPC** parfois appelés **codes Gallager** sont restés utiles pendant plus de 50 ans.
Pendant les années 70, Gallager a déplacé son viseur sur les réseaux de données, les algorithmes distribués et les techniques d'accès aléatoire. Dans ce cadre, il a écrit **Réseaux de données, Pentice Hall**, publié en 1988, avec la deuxième édition en 1992, co-écrite avec **Dimitri Bertsekas** qui est un mathématicien, ingénieur éléctricien et informaticien qui a contribué à fournir une conception pour ce domaine.
Quelques années après, la passion de Gallager par la théorie de l'information est revenue, il a misé beaucoup d'énergie sur le thème de la communication sans fil, les réseaux optiques et les processus stochastiques. Il a écrit le **manuel de 1996, Processus stochastiques discrets**.
Durant sa carrière, Gallager était **président de la Société de théorie de l'information IEEE** en 1971, membre de son conseil de gouverneurs 1965-1972 et de nouveau 1979-1988. Il a servi en tant que rédacteur en chef adjoint pour le codage au sein de **l'IEEE Transactions on Information Theory** pendant la période 1963-1964 et en tant que rédacteur associé pour les communications informatiques de 1977 à 1980.
En tant que professeur, il a reçu le **MIT Graduate Student Council Teaching Award** en 1993 et le **Prix du Japon** en 2020 comme une récompense à cette carrière embellie de beaucoup de succès et d'encadrement d'étudiants qui sont maintenant devenus eux-mêmes des chercheurs.
Dans ce travail, on va s'intéresser aux travaux de Gallager effectués sur le thème des codes LDPC (Low Density Parity Check).

4 Graphe de Tanner

Dans la théorie des codes correcteurs d'erreurs, un **graphe de Tanner**, nommé après Michael Tanner , est un graphe biparti utilisé pour indiquer des contraintes ou des équations spécifiques aux codes correcteurs d'erreurs. Dans cette théorie, les graphes de Tanner sont utilisés pour créer des codes longs à partir de codes plus court. Ce graphe est utilisé de manière intensive dans le codage et le décodage.

Le **graphe de Tanner** est compsé de :

1. **Noeuds de variables :** associés aux bits du mot de code
2. **Noeuds de parité :** associés aux équations de parité
3. **Branches :** lien entre noeuds de variables et noeuds de parité. Un noeud de variable n est connecté au noeud de parité m si $h_m n = 1$ dans la matrice de parité H.

Exemple : Graphe de Tanner du code de Hamming

Soit la matrice de Parité :

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

Les coefficients H_{11}, H_{13}, H_{15} et H_{17} sont égaux à 1 donc le noeud c_1 est relié aux noeuds v_1, v_3, v_5 et v_7 .

Les coefficients H_{22}, H_{23}, H_{26} et H_{27} sont égaux à 1 donc le noeud c_2 est relié aux noeuds v_2, v_3, v_6 et v_7 .

De même, Les coefficients H_{34}, H_{35}, H_{36} et H_{37} sont égaux à 1 donc le noeud c_3 est relié aux noeuds v_4, v_5, v_6 et v_7 .

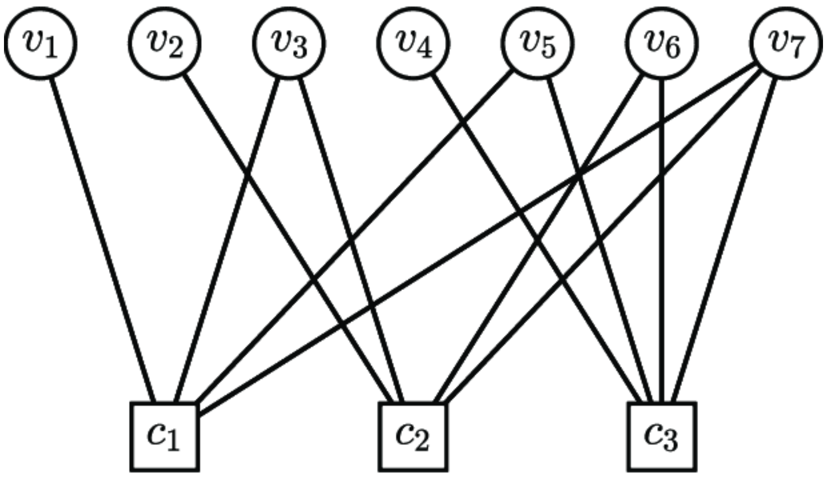

FIGURE 1 – Graphe de Tanner associé

Autre exemple : Soit la matrice de Parité :

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

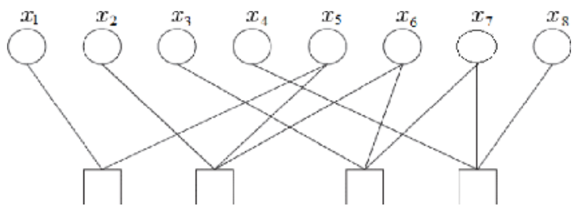

FIGURE 2 – Graphe biparti de Tanner associé

Le graphe de Tanner est utilisé comme support pour la plupart des algorithmes de décodage. Le principe principal d'un algorithme de décodage utilisant cette représentation est de considérer chaque branche du graphe comme un message d'un noeud c_j vers un noeud v_i et réciproquement.

5 Algorithme de décodage

Il existe plusieurs types d'algorithmes de décodage pour les codes LDPC. Cependant, nous en avons étudier un seul. Ce dernier consiste à faire baisser le poids du syndrome jusqu'à que ce dernier soit nul. L'objectif de cet algorithme est de relever les colonnes d'une matrice de parité H qui sont erronés. Pour ce faire, les positions qui sont en erreurs vont faire baisser le poids de notre syndrome. Ainsi, on va parcourir chaque colonnes de la matrice H afin de trouver ces dernières. On définit le code LDPC C à l'aide d'une matrice de parité $\mathbf{H}$, qui est creuse. Soit $y = x + e$, où y est le message reçu, x le message chiffré et e le vecteur erreur.

Voici l'algorithme :

Algorithme 1 : Algorithme de décodage LDPC

Data : Soient E un motif d'erreur et H une matrice de parité de taille n.
Result : Syndrome de l'erreur courant
1 On définit les h_i, $\forall i \in [\![0\ ;\ n]\!]$ les colonnes de la matrice H.
2 Et on note $\omega(e)$ le poids de e.
3 $S = \sigma(E) = H.\,{}^tE$
4 **for** $i \leftarrow 0$ **to** n **do**
5   **if** $\omega(S + h_i) \leqslant \omega(S)$ **then**
6         $E_1 \leftarrow E_1 + h_i$
7   **end**
8 **end**
9 **if** $S = \omega(E_1)$ **then**
10   le syndrome de l'erreur trouvé est le syndrome de l'erreur courante;
11   **return** E_1
12 **else**
13   on répète l'algorithme avec le nouveau motif d'erreur trouvé;
14     $S = \sigma(E) + \sigma(E_1) = \sigma(E + E_1)$;
15   **return** E_1+ **repeat**
16   **until** $\omega(S) = 0$;
17 **end**

On peut remarquer que l'algorithme peut vite faire une boucle infini si la condition à la ligne **17** n'est pas satisfaite. Nous allons définir un nombre maximal d'itération à exécuter. En effet, il se peut que l'algorithme de décodage LDPC ne trouve pas de solution, de ce faite, on évite une boucle infini.

Dans notre implémentation Python, nous avons fait le choix d'éxécuter au plus 25 fois la recherche puisque si le poids ne diminue plus, on ne peut pas décoder le code LDPC. Nous verrons cela dans la partie 6.

```python
def decode_LDPC_aux(H,S,weight_S,n,break_cpt):
    if break_cpt >= 25:
        #print(f'Error: LDPC code not found after {break_cpt} iterations')
        return []
    #On stocke dans L, les positions erronées
    L = []
    for i in range(n):
        if weightOfCol(mod2(S + getCol(H,i), n//2),n//2) <= weight_S:
            L.append(i)
    #S2 est la somme de toutes les colonnes de H qui sont erronées
    S2 = np.zeros(n//2)
    for i in L:
        S2 = S2 + getCol(H,i)
    S2 = mod2(S2, n//2)
    #S3 est notre nouveau syndrome
    S3 = np.zeros(n//2)
    S3 = mod2(S+S2, n//2)
    #si S3 est nul alors on a trouvé les positions erronées
    #sinon, on continue
    if weightOfCol(S3,n//2) == 0:
        #print(f'Error found in {break_cpt} times')
        return L
    else:
        return L + decode_LDPC_aux(H,S3,weightOfCol(S3,n//2),n,break_cpt+1)

def decode_LDPC(H,e,n):
    S = mod2(H@(e.T),n//2)
    weight_S = weightOfCol(S,n//2)
    ldpc_aux = decode_LDPC_aux_strict(H,S,weight_S,n,1)
    ldpc = []
    for i in ldpc_aux:
        if listCounter(i,ldpc_aux) % 2 == 1 and i not in ldpc:
            ldpc.append(i)
    return ldpc
```

Les fonctions auxiliaires apparaissent sur l'annexe 8.

6 Expérimentations et résultats

Nous avons réaliser plusieurs types d'expériences. Nous avons découvert que plus la taille des matrices est grandes, plus l'algorithme de décodage fonctionne. En effet, Pour livrer nos résultats, nous avons choisit des matrices de longueur 1000 et 2000.

Nous avons implémenter des fonctions8 de façon à pouvoir créer des matrices creuses. La fonction *matrixFromWeight(poids,n)* créer une matrice de dimension $\frac{n}{2}$ par n de tel sorte que chaque colonne est un poids de valeur *poids* et est unique. Sinon, notre matrice de parité ne représente pas un code LDPC régulier.

Objectif de l'expérimentation : Nous allons essayer de déterminer le poids optimale sur chaque colonne pour décoder un code LDPC de longueur n.

La méthode adopter est la suivante :

1. Construire une matrice dont les colonnes ont un certain poids.
2. Observer combien de vecteurs erreurs d'un certain poids, nous arrivons à retrouver grâce à notre algorithme de décodage classique.
3. Essayer d'optimiser l'algorithme de décodage en supprimant les position parasites et donc essayer de décoder plus de vecteurs erreurs.
4. Comparer les résultats des deux algorithmes.

Remarque : Une position parasite est une colonne de H qui satisfait la condition $\omega(S + h_i) \leqslant \omega(S)(1)$ mais qui n'est pas une position érroné. En outre, on va durcir la condition (1) de façon a ignorer ces positions. Pour cela, on essaiera de récupérer les colonnes de H qui font baisser le syndrome. ($\omega(S + h_i) < \omega(S)$ ou encore $\omega(S + h_i) < \omega(S) - 1$, par exemple)

6.1 $n = 1000$

Nous réalisons

Nota bene :

La Table 1 présente un résumé des résultats de nos expérimentations.

Table 1 – Comparaison des différentes méthodes

Algorithme	Temps CPU (ms)		Gap MS		#Données MS
	Moyenne	Ecart type	Moyenne	Ecart type	
Heuristique PPV	29	96	8.4%	**36%**	6/38
Heuristique MI	16984	78093	**1.3%**	42%	**30/38**

7 Conclusion

C'est la fin.

8 Annexe

Fichier tools.py

```
import random
import numpy as np

def mod2(a,n):
    for i in range(n):
        a[i]%=2
    return a

# retourne la ind-ème colonne de la matrice a
def getCol(a,ind):
    a = a.T
    return a[ind]

# Retourne une liste d'indices qui contiennent des 1
def getInd1inCol(a,n):
    res = []
    for i in range(n):
        if a[i] == 1:
            res.append(i)
    return res

# Calcul le poids d'un vecteur colonne
def weightOfCol(a,n):
    cpt = 0
    for i in range(n):
        if a[i] == 1:
            cpt+=1
    return cpt

# Teste si deux colonnes sont égales
def colEquals(a,b):
    return np.array_equal(a,b)

# Teste si deux matrices ont une colonne en commun
def colsEquals(a,b,n):
    a = a.T
    b = b.T
    for i in range(n):
        for j in range(n):
            if colEquals(a[i],b[j]):
                return True
    return False

# Permutation au hasard des colonnes d'une matrice a
def createRandFlipMatrix(a):
    return np.random.permutation(a.T).T

# Calcule le nombre de 1 dans une liste L
def nbOfOneFromList(L,n):
    cpt = 0
    for i in range(n):
```

```
        if L[i] == 1:
            cpt+=1
    return cpt

# Retourne une liste contenant "poids" 1 aléatoirement.
def randomIndOne(weight,n):
    L = [0 for i in range(n)]
    while nbOfOneFromList(L,n) != weight:
        L[random.randint(0,n-1)] = 1
    return L

# Retourne une liste contenant toutes les listes de randomIndOne.
# On fait attention de ne pas produire de doublon.
def listOfRandomIndOne(weight,n):
    L = []
    while len(L) != n:
        tmp = randomIndOne(weight,n//2)
        if tmp not in L:
            L.append(tmp)
    return L

# Détermine la taille de la matrice pour un poids de
# ligne donnée
def determineSizeForGallager(weight_row):
    # init
    i , j = 5 , 6
    while i < weight_row:
        i+=1
        j+=3
    if i == weight_row:
        return i*j
    else:
        raise ValueError

def listCounter(elem,L):
    cpt = 0
    for i in L:
        if i == elem:
            cpt+=1
    return cpt
```

Fichier matrix.py

```
import random
import numpy as np

from tools import *

# On créer une matrice de la forme suivante:
# 1 1 1 1 1 1 0 . . . . . . . 0 0 0 0
# 0 0 0 0 0 0 1 1 1 1 1 1 0 0 . . . .
# . . . . . . . . . . . . . . . . . .
def lowDensityMatrix(weight_row,n):
    a = np.zeros((n//weight_row,n))
```

```
    i , j = 0 , 0
    while j < n//weight_row and i < n:
        if (i+weight_row)%weight_row == 0 and i >= weight_row:
            j = j+1
        a[j,i] = 1
        i = i+1
    return a

# La fonction construit une matrice de Gallager ici, n doit
# être un multiple de 6 pour que la fonction marche
# Par exemple, on peut appeler la fonctions avec les couples suivants:
# (5,30) ; (6,54) ; (7;84) ; (8;120) ; ....
def createGallagerMatrix(weight_row,n):
    a = lowDensityMatrix(weight_row,n)
    b = createRandFlipMatrix(a)
    while not colsEquals(a,b,n//6):
        b = createRandFlipMatrix(a)
    c = createRandFlipMatrix(a)
    while not colsEquals(a,c,n//6) and not colsEquals(b,c,n//6):
        c = createRandFlipMatrix(a)
    return np.concatenate(((np.concatenate((a,b), axis=0)),c),axis=0)

# Retourne une liste de vecteur erreur d'un certain poids
def randomErrorVectorGenerator(poids,max_loop,n):
    errors = []
    for _ in range(max_loop):
        tmp = np.zeros(n)
        while weightOfCol(tmp,n) != poids:
            tmp[random.randint(0,n-1)] = 1
        errors.append(tmp)
    return errors

# Retourne une matrice de taille (n,n//2) tel que les colonnes est un
# poids prescrit. On ne produit pas de colonnes identiques.
def matrixFromWeight(poids,n):
    L = listOfRandomIndOne(poids,n)
    res = np.array(L[0])
    for col in L:
        if np.array_equal(col,L[0]) == False:
            res = np.concatenate((res,np.array(col)), axis=0)
    return np.reshape(res, (n,n//2)).T
```

Fichier main.py

```
import numpy as np
from time import time

from matrix import *
from tools import *

def decode_LDPC_aux(H,S,weight_S,n,break_cpt):
    if break_cpt >= 25:
        #print(f'Error: LDPC code not found after {break_cpt} iterations')
        return []
```

```
    #On stocke dans L, les positions erronées
    L = []
    for i in range(n):
        if weightOfCol(mod2(S + getCol(H,i), n//2),n//2) <= weight_S:
            L.append(i)
    #S2 est la somme de toutes les colonnes de H qui sont erronées
    S2 = np.zeros(n//2)
    for i in L:
        S2 = S2 + getCol(H,i)
    S2 = mod2(S2, n//2)
    #S3 est notre nouveau syndrome
    S3 = np.zeros(n//2)
    S3 = mod2(S+S2, n//2)
    #si S3 est nul alors on a trouvé les positions erronées
    #sinon, on continue
    if weightOfCol(S3,n//2) == 0:
        #print(f'Error found in {break_cpt} times')
        return L
    else:
        return L + decode_LDPC_aux(H,S3,weightOfCol(S3,n//2),n,break_cpt+1)

def decode_LDPC(H,e,n):
    S = mod2(H@(e.T),n//2)
    weight_S = weightOfCol(S,n//2)
    ldpc_aux = decode_LDPC_aux_strict(H,S,weight_S,n,1)
    ldpc = []
    for i in ldpc_aux:
        if listCounter(i,ldpc_aux) % 2 == 1 and i not in ldpc:
            ldpc.append(i)
    return ldpc

def test_decode_LDPC(H,weight_e,max_loop,n):
    cpt = 0
    errors = randomErrorVectorGenerator(weight_e,max_loop,n)
    for e in errors:
        if decode_LDPC(H,e,n) == getInd1inCol(e,n):
            cpt+=1
    return cpt/max_loop

###############################################################
#                        Strict version                       #
###############################################################

def decode_loop(H,S,weight_S,n,break_cpt):
    L = []
    # On essaie de supprimer les positions parasites
    if break_cpt == 1:
        for i in range(n):
            if weightOfCol(mod2(S + getCol(H,i), n//2),n//2) <= weight_S - 2:
                L.append(i)
    else:
        for i in range(n):
            if weightOfCol(mod2(S + getCol(H,i), n//2),n//2) <= weight_S - 1:
                L.append(i)
    return L
```

```
def decode_LDPC_aux_strict(H,S,weight_S,n,break_cpt):
    if break_cpt >= 25:
        return []
    L = decode_loop(H,S,weight_S,n,break_cpt)
    S2 = np.zeros(n//2)
    for i in L:
        S2 = S2 + getCol(H,i)
    S2 = mod2(S2, n//2)
    S3 = np.zeros(n//2)
    S3 = mod2(S+S2, n//2)
    if weightOfCol(S3,n//2) == 0:
        return L
    else:
        return L + decode_LDPC_aux_strict(H,S3,weightOfCol(S3,n//2),n,break_cpt+1)

def decode_LDPC_strict(H,e,n):
    S = mod2(H@(e.T),n//2)
    weight_S = weightOfCol(S,n//2)
    ldpc_aux = decode_LDPC_aux_strict(H,S,weight_S,n,1)
    #print(sorted(ldpc_aux))
    ldpc = []
    for i in ldpc_aux:
        if listCounter(i,ldpc_aux) % 2 == 1 and i not in ldpc:
            ldpc.append(i)
    return ldpc

def test_decode_LDPC_strict(H,weight_e,max_loop,n):
    cpt = 0
    errors = randomErrorVectorGenerator(weight_e,max_loop,n)
    for e in errors:
        if decode_LDPC_strict(H,e,n) == getInd1inCol(e,n):
            cpt+=1
    return cpt/max_loop

################################################################
#                     Display functions                        #
################################################################

def displayTestLoop(H,weight_e,max_loop,n):
    print(f"""Test sur {max_loop} vecteurs erreurs de poids {weight_e}:
                            {test_decode_LDPC(H,weight_e,max_loop,n)}""")
    return None

def displayTestLoop_strict(H,weight_e,max_loop,n):
    print(f"""Test sur {max_loop} vecteurs erreurs de poids {weight_e}:
                    {test_decode_LDPC_strict(H,weight_e,max_loop,n)}""")
    return None

def displayTestLoopWithTime(H,weight_e,max_loop,n):
    start = time()
    print(f"""Test sur {max_loop} vecteurs erreurs de poids {weight_e}:
                            {test_decode_LDPC(H,weight_e,max_loop,n)}""")
    end = time()
    print("Time:",end - start)
```

```python
    return None

def displayTest(res,weight_e,max_loop):
    print(f"Test sur {max_loop} vecteurs erreurs de poids {weight_e}: {res}")
    return None

###################################################################
#                      Experimental tests                         #
###################################################################

def opt_weight_search(max_loop,n):
    for j in range(3,30):
        start = time()
        H = matrixFromWeight(j,n)
        print(f"Soit H une matrice de taille {n//2}x{n}")
        print(f"Dont le poids des colonnes est {j}.\n")
        i = 1
        # Arrêt lorsque la proba est inférieur à 0.4
        # Ou si la proba est différente de 1 pour le poids 1
        # (on gagne du temps, puisque'on sait que c'est pas optimal)
        tmp = test_decode_LDPC(H,i,max_loop,n)
        if tmp == 1:
            while tmp >= 0.67:
                displayTest(tmp,i,max_loop)
                i += 1
                tmp = test_decode_LDPC(H,i,max_loop,n)
            displayTest(tmp,i,max_loop)
            print("Time:",time() - start, " secondes \n")
        else:
            displayTest(tmp,i,max_loop)
            print("Time:",time() - start, " secondes \n")
    return None

# Lorsqu'on a trouvé le poids optimal pour le décodage d'une matrice
# de taille (n//2,n) on regarde si on peut améliorer le décodage.
def opt_weight_search_strict(borne_inf,borne_sup,max_loop,n):
    print("Version optimal")
    for j in range(borne_inf,borne_sup + 1):
        start = time()
        H = matrixFromWeight(j,n)
        print(f"Soit H une matrice de taille {n//2}x{n}")
        print(f"Dont le poids des colonnes est {j}.\n")
        i = 1
        tmp = test_decode_LDPC_strict(H,i,max_loop,n)
        while tmp >= 0.8:
            displayTest(tmp,i,max_loop)
            i += 1
            tmp = test_decode_LDPC_strict(H,i,max_loop,n)
        displayTest(tmp,i,max_loop)
        print("Time:",time() - start, " secondes \n")
    return None

def test_Gallager(max_loop,weight_row):
    n = determineSizeForGallager(weight_row)
    H = createGallagerMatrix(weight_row,n)
```

```
    print(f"Soit H une matrice de taille {n//2}x{n}")
    start = time()
    i = 1
    tmp = test_decode_LDPC(H,i,max_loop,n)
    if tmp == 1:
        while tmp >= 0.67:
            displayTest(tmp,i,max_loop)
            i += 1
            tmp = test_decode_LDPC(H,i,max_loop,n)
        displayTest(tmp,i,max_loop)
        print("Time:",time() - start, " secondes \n")
    else:
        displayTest(tmp,i,max_loop)
        print("Time:",time() - start, " secondes \n")
    return None

if __name__ == "__main__":
    opt_weight_search(15,2000)
    opt_weight_search_strict(3,14,200,1000)
```