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 Sujet

Dans la théorie des codes correcteurs d'erreurs, les codes dits « Low Density Parity Check » (LDPC) occupent une place très importante. Ces codes sont définis par des équations de petit poids, que l'on peut donc regrouper dans une matrice de parité creuse, d'où leur nom. Ils ont été proposés dans les années 1960, et ont donné lieu à des développements extrêmement variés à partir des années 1990. Des variantes des codes LDPC trouvent également des applications en cryptographie. Il s'agira de comprendre comment fabriquer des instances de tels codes, ainsi que leur décodage, qui dans sa variante la plus simple consiste en une décision majoritaire sur le nombre d'équations non satisfaites par un mot de code erroné. On pratiquera un certain nombre d'expérimentations sur des exemples.

2 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 4 présente les étapes de fabrication d'un code LDPC ainsi qu'une matrice de contrôle qui répond à des conditions bien spécifiques. La section 6 présente l'algorithme détaillé de décodage LDPC. La section 7 est dédiée aux expériences pratiques qu'on a effectué durant l'implémentation de l'algorithme.

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.

3 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.

3.1 Code linéaire

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

3.2 Matrice génératrice

Définition d'une matrice génératrice : Soit C $\subset {\mathbb{F}_{||}}^n$ 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 :

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

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

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

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

3.3 Matrice de contrôle

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

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

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

$$\text{H}= (-\ {}^t A.\text{Id}_{n-k})$$

⇒ On applique donc cette règle et on obtient :

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

3.4 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 :

$$\text{H}\ {}^t y = \text{H}\ {}^t(x + r) = \text{H}\ {}^t x + \text{H}\ {}^t r = \text{H}\ {}^t r$$

car H ${}^t x$ =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.

4 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.

4.1 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 colonne égal à 1 et un poids de ligne γ. Ensuite on doit trouver des permutations des 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**.

4.2 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 consécutif ordonné 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

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

4.3 Robert Gray Gallager

Robert Gray Gallager, né en 1931 est un ingénieur américain en éléctricité qui consacra sa vie à travailler sur 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 servit pour l'une des branches de l'armée des Etats-Unis, qui est responsable de la création des communications et des systèmes d'information du **Corps des transmissions de l'armée des Etats-Unis (USASC)**. En 1957, Gallager décrocha un Master en Sciences (S.M) et en 1960, il devint docteur en sciences spécialisé en ingénierie éléctrique.
Sa thèse de doctorat portait sur les codes de contrôle de parité de basse densité (LDPC) qui fût 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 s'est dédié aux 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. Ce dernier a contribué à fournir une conception au sein de 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 écrivit 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** durant la période de 1963 à 1964 et en tant que rédacteur associé pour les communications informatiques de 1977 à 1980.
En tant que professeur, il reçut le **MIT Graduate Student Council Teaching Award** en 1993 et le **Prix du Japon** en 2020 comme une récompense pour sa 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).

5 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 .

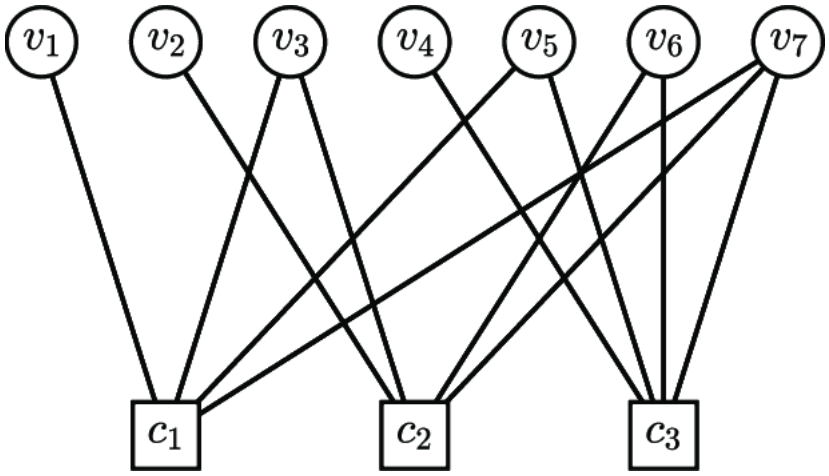

FIGURE 1 – Graphe de Tanner associé

Autre exemple : Soit la matrice de Parité :

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

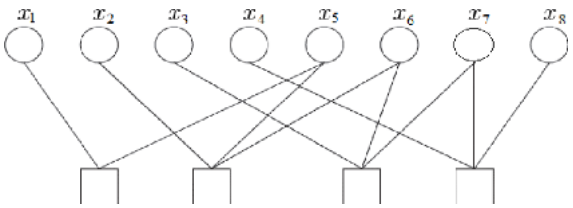

FIGURE 2 – Graphe biparti de Tanner associé

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.

6 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, ∀i ∈ ⟦0 ; n⟧ les colonnes de la matrice H.
2  Et on note ω(e) le poids de e.
3  S = σ(E) = H. ᵗE
4  for i ← 0 to n do
5  |  if ω(S + h_i) ⩽ ω(S) then
6  |  |  E_1 ← E_1 + h_i
7  |  end
8  end
9  if S = ω(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 = σ(E) + σ(E_1) = σ(E + E_1);
15 |  return E_1+ repeat
16 |  until ω(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 7.

```
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 9.

7 Expérimentations et résultats

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

Nous avons implémenté des fonctions9 de façon à pouvoir créer des matrices creuses. La fonction *matrixFromWeight(poids,n)* crée une matrice de dimension $\frac{n}{2}$ par n de telle sorte que chaque colonne ait 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 optimal sur chaque colonne pour décoder un code LDPC de longueur n.

La méthode adoptée 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 positions parasites et donc essayer de décoder plus de vecteurs erreurs (ou non).
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 érronée. En outre, on va durcir la condition (1) de façon à 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)

7.1 $n = 1000$

Nous réalisons une série d'expériences sur des matrices de dimension 500 par 1000. Pour ce faire, on a regardé combien de vecteurs erreurs sommes-nous capable de décoder. On se définit un échantillon de 200 vecteurs erreurs puis on calcule la probabilité de décodage en fonction d'un type de matrice donné. En effet, on définit une matrice avec un certain poids sur chaque colonne puis on appelle l'algorithme de décodage.

La Table 1 présente un résumé des résultats de nos expérimentations pour $n = 1000$ à l'aide de la fonction *decode_LDPC* (étape 2 de la démarche expérimentale).

La Table 2 présente un résumé des résultats de nos expérimentations pour $n = 1000$ à l'aide de la fonction *decode_LDPC_strict* (étape 3 de la démarche expérimentale). Pour la fonction *decode_LDPC_strict*, nous avons choisi d'essayer de faire baisser le poids du syndrome de 2 à la première itération puis de 1 sur les deux suivantes. Ensuite, l'algorithme se déroule sans contraintes sur les 22 autres itérations.

On peut observer plusieurs choses. Premièrement, sur la table 1 on arrive à décoder les 200 vecteurs erreurs de poids 8 sur une matrice dont les colonnes sont de poids 11. Sur cette même colonne, les vecteurs erreurs de poids 7 ne sont pas tous décodés (99% de décodage). On peut expliquer cela à cause de l'aléatoire, en effet, nous générons des vecteurs erreurs aléatoirement et des matrices aléatoirement aussi. Donc il est possible d'obserser ce type de phénomène puisque les positions des 1 ne vont pas s'intersecter. De ce fait, le poids du syndrome ne va pas diminuer mais augmenter.
Étant donné que nous décodons le plus de vecteurs dont les colonnes sont de poids 11, ce dernier semble être le poids optimal pour $n = 1000$. Lorqu'on observe la table 2 les résultats sont convaincants jusqu'au poids 6 sur les vecteurs erreurs. Mais l'optimal semble se situer entre des poids de colonnes qui se situent entre 10 et 12 inclus. On ne décode pas autant de vecteurs erreurs car en 25 itérations, nous ne parvenons pas à trouver toutes les solutions.

Deux explications possibles :

1. On diminue trop le poids du syndrome et on oublie des positions
2. Ou, on exécute pas assez d'itérations

Il est difficile de répondre directement à la première explication, cependant on peut lancer nos tests sur plus d'itérations par exemple. Nous faisons 50. On arrive à observer des résultats similaires que pour 25 itérations. On ne manipule jamais les mêmes matrices ni les même échantillons de vecteurs erreurs puisque nous générons tout aléatoirement. Parfois, on arrive à décoder les 200 vecteurs erreurs de poids 7 mais le plus souvent non. On peut donc en déduire expérimentalement que l'on diminue trop le poids du syndrome.

Observations supplémentaires : L'algorithme strict nous permet de décoder beaucoup plus de vecteurs erreurs que le premier algorithme. On a fait le choix d'arrêter nos algorithmes lorque la dernière probabilité de décodage calculé est inférieure ou égale à 0.8. En effet, nous avons pu observer que dans tous les cas, si la probabilité chute drastiquement après ce palier donc nous avons jugé ces résulats non pertinents. Donc on peut observer que sur la table 1 les résultats chutent d'un coup après que la probabilité de décodage soit raisonnable. On peut observer ce phénomène sur la colonne des poids 14 par exemple. Si on observe cette même colonne dans la 2 on peut voir que les résultats sont plus lisses.

7.2 $n = 2000$

La Table 3 présente un résumé des résultats de nos expérimentations pour $n = 2000$ à l'aide de la fonction *decode_LDPC_strict*.

Les expériences sur les matrices de dimension 1000 par 2000 sont gourmandes en temps et en ressource. La création de la matrice n'est pas instantanée. Par exemple, créer une matrice ayant un poids de 9 sur les colonnes, environ 15 secondes sont nécessaires. Ainsi, nous allons présenter les résultats de *decode_LDPC_strict* sur un échantillon de 25 vecteurs erreurs. Et sur 25 itérations maximales de décodage.

On observe que l'algorithme de décodage fonctionne bien, encore mieux que lorsque $n = 1000$. On arrive à décoder jusqu'à 25 vecteurs erreurs de poids 21. Ici, encore, on observe le même phénomène qui est dû à l'aléatoire. Pour ce poids de colonne, on n'observe pas que des probabilités certaines jusqu'au poids 21, cependant il porte à croire que 9 est le poids optimal pour $n = 2000$. En effet, c'est le poids de colonne qui permet de décoder le plus de vecteurs erreurs d'un poids élévé.

Dans l'ensemble, on a observé que supprimer les positions parasites était beaucoup plus efficace pour $n = 2000$. En effet, on arrive à gagner en terme de capacité de décodage.

Dans l'ensemble, on a observé que supprimer les positions parasites était beaucoup plus efficace pour $n = 2000$. En effet, on arrive à gagner en terme de capacité de décodage. En laissant tourner la fonction sur 50 itérations, on ne parvient pas à obtenir une variation des résultats. Cependant, on peut observer une amélioration dans la qualité de ces derniers puiqu'on obtient une série de probabilité qui vaut 1 jusqu'au poids de vecteurs erreur 18.

8 Annexe

TABLE 1 – Probabilité de décodage sur 200 vecteurs erreurs d'un certain poids en fonction du poids des colonnes d'une matrice de code LDPC

	Poids des colonnes								
Poids des vecteurs erreurs	3	4	5	6	7	8	9	10	11
1	0.915	0.745	1.0	0.96	0.99	1.0	1.0	1.0	1.0
2	-	-	1.0	-	-	1.0	1.0	1.0	1.0
3	-	-	1.0	-	-	1.0	1.0	1.0	1.0
4	-	-	1.0	-	-	1.0	1.0	1.0	1.0
5	-	-	1.0	-	-	1.0	1.0	1.0	1.0
6	-	-	0.995	-	-	1.0	1.0	1.0	1.0
7	-	-	1.0	-	-	0.945	1.0	0.995	0.99
8	-	-	0.98	-	-	0.74	0.995	0.825	1.0
9	-	-	0.975	-	-	-	0.985	0.55	0.98
10	-	-	0.95	-	-	-	0.955	-	0.915
11	-	-	0.925	-	-	-	0.885	-	0.725
12	-	-	0.87	-	-	-	0.75	-	-
13	-	-	0.775	-	-	-	-	-	-
Temps CPU (s)	263	560	3458	203	131	3050	3318	3176	2748
-	-	-	-	-	-	-	-	-	-
Poids des vecteurs erreurs		12	13	14	15	16	17	18	19
1		1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2		1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
3		1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
4		1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
5		1.0	1.0	1.0	0.995	1.0	0.995	0.995	0.985
6		1.0	1.0	1.0	1.0	1.0	0.985	0.985	1.0
7		0.985	1.0	0.99	0.995	0.955	0.97	0.92	0.995
8		0.905	0.985	0.8	0.99	0.685	0.955	0.605	0.93
9		0.51	0.97	0.395	0.94	-	0.895	-	0.645
10		-	0.855	-	0.76	-	0.8	-	-
11		-	0.57	-	-	-	0.58	-	-
12		-	-	-	-	-	-	-	-
13		-	-	-	-	-	-	-	-
Temps CPU (s)		4246	5099	4169	2204	1968	4503	2317	1997

TABLE 2 – Probabilité de décodage sur 200 vecteurs erreurs d'un certain poids en fonction du poids des colonnes d'une matrice de code LDPC

	Poids des colonnes								
Poids des vecteurs erreurs	3	4	5	6	7	8	9	10	11
1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
3	0.97	1.0	0.99	1.0	1.0	1.0	0.995	1.0	1.0
4	0.895	0.99	0.99	0.99	1.0	1.0	1.0	1.0	0.99
5	0.835	0.98	0.965	0.995	0.99	0.995	0.99	1.0	0.99
6	0.725	0.97	0.91	0.975	0.95	0.985	0.985	1.0	0.99
7	-	0.945	0.82	0.96	0.915	0.98	0.96	0.975	0.965
8	-	0.905	0.81	0.965	0.865	0.965	0.925	0.97	0.915
9	-	0.86	0.67	0.905	0.835	0.925	0.91	0.955	0.88
10	-	0.82	-	0.885	0.825	0.9	0.81	0.93	0.81
11	-	0.775	-	0.845	0.635	0.845	0.705	0.855	0.725
12	-	-	-	0.785	-	0.825	-	0.82	-
13	-	-	-	-	-	0.69	-	0.745	-
Temps CPU (s)	890	2082	1405	2035	1475	3694	1457	1953	1661
-	-	-	-	-	-	-	-	-	-
Poids des vecteurs erreurs		12	13	14	15	16	17	18	19
1		1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2		1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
3		1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
4		1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
5		0.995	0.995	1.0	1.0	1.0	0.995	1.0	0.995
6		1.0	0.985	1.0	0.99	0.995	0.985	0.995	0.985
7		0.995	0.96	0.99	0.98	0.99	0.97	0.98	0.975
8		0.985	0.93	0.98	0.94	0.945	0.955	0.95	0.935
9		0.95	0.9	0.965	0.865	0.9	0.895	0.935	0.81
10		0.905	0.83	0.935	0.78	0.83	0.8	0.785	0.7
11		0.85	0.745	0.855	-	0.655	0.58	-	-
12		0.82	-	0.7	-	-	-	-	-
13		0.625	-	-	-	-	-	-	-
Temps CPU (s)		2652	1629	3899	1297	1951	1456	1609	1606

TABLE 3 – Probabilité de décodage sur 25 vecteurs erreurs d'un certain poids en fonction du poids des colonnes d'une matrice de code LDPC (n=2000)

	Poids des colonnes									
Poids des vecteurs erreurs	3	4	5	6	7	8	9	10	11	12
1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
4	0.92	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
5	0.88	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
6	0.76	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
7	-	0.96	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
8	-	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
9	-	0.92	1.0	1.0	1.0	1.0	1.0	1.0	0.96	1.0
10	-	1.0	0.92	0.96	0.92	0.96	1.0	1.0	0.96	0.96
11	-	0.96	0.88	1.0	0.96	1.0	1.0	1.0	1.0	0.96
12	-	0.8	0.88	0.96	0.92	0.96	0.96	1.0	1.0	1.0
13	-	0.88	0.76	0.92	0.92	1.0	1.0	1.0	0.92	1.0
14	-	0.72	-	0.88	0.8	0.92	1.0	0.96	0.88	1.0
15	-	-	-	1.0	0.76	1.0	0.92	0.96	0.8	0.96
16	-	-	-	0.84	-	1.0	1.0	0.84	0.92	1.0
17	-	-	-	0.76	-	0.96	0.92	0.88	0.96	0.88
18	-	-	-	-	-	0.92	1.0	0.88	0.84	0.84
19	-	-	-	-	-	0.84	0.96	0.84	0.8	0.8
20	-	-	-	-	-	0.92	0.84	0.76	0.76	0.88
21	-	-	-	-	-	0.72	1.0	-	-	0.8
22	-	-	-	-	-	-	0.88	-	-	0.72
23	-	-	-	-	-	-	0.8	-	-	-
24	-	-	-	-	-	-	0.72	-	-	-
-	-	-	-	-	-	-	-	-	-	
Poids des vecteurs erreurs	13	14	15	16	17	18	19	20	21	22
1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
4	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
5	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
6	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
8	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
11	1.0	1.0	1.0	0.96	1.0	0.96	1.0	1.0	1.0	1.0
12	1.0	1.0	0.96	1.0	1.0	0.96	1.0	0.96	1.0	1.0
13	1.0	1.0	1.0	1.0	1.0	0.92	0.96	1.0	1.0	0.88
14	1.0	0.96	0.96	1.0	1.0	1.0	1.0	0.92	0.96	1.0
15	0.96	0.92	1.0	0.96	1.0	0.92	1.0	0.92	0.96	0.84
16	1.0	0.88	1.0	0.92	1.0	0.96	0.96	0.7	0.84	0.92
17	1.0	0.84	1.0	0.88	0.84	0.96	0.96	-	0.56	0.92
18	0.96	0.8	0.84	0.84	0.88	0.76	0.48	-	-	0.68
19	0.88	0.92	0.88	0.8	0.64	-	-	-	-	-
20	0.7	0.88	0.72	0.84	-	-	-	-	-	-
21	-	0.64	-	0.48	-	-	-	-	-	-
22	-	-	-	-	-	-	-	-	-	-
23	-	-	-	-	-	-	-	-	-	-
24	-	-	-	-	-	-	-	-	-	-

9 Implémentation

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)
```

```
    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)
```