



Codes LDPC

Anthony PERRONI ^t

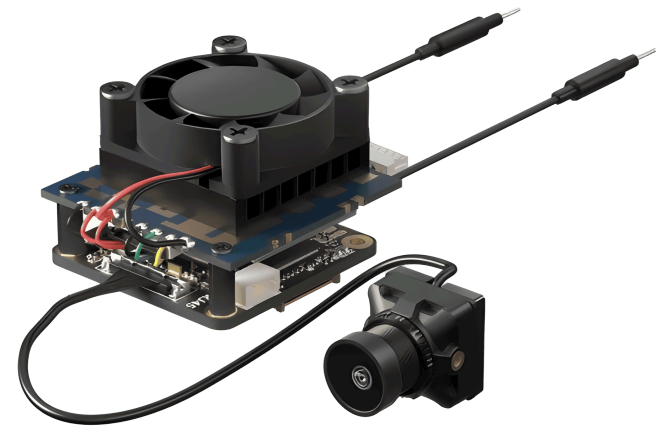
n°49871

2025 - 2026

Introduction : Utilisation

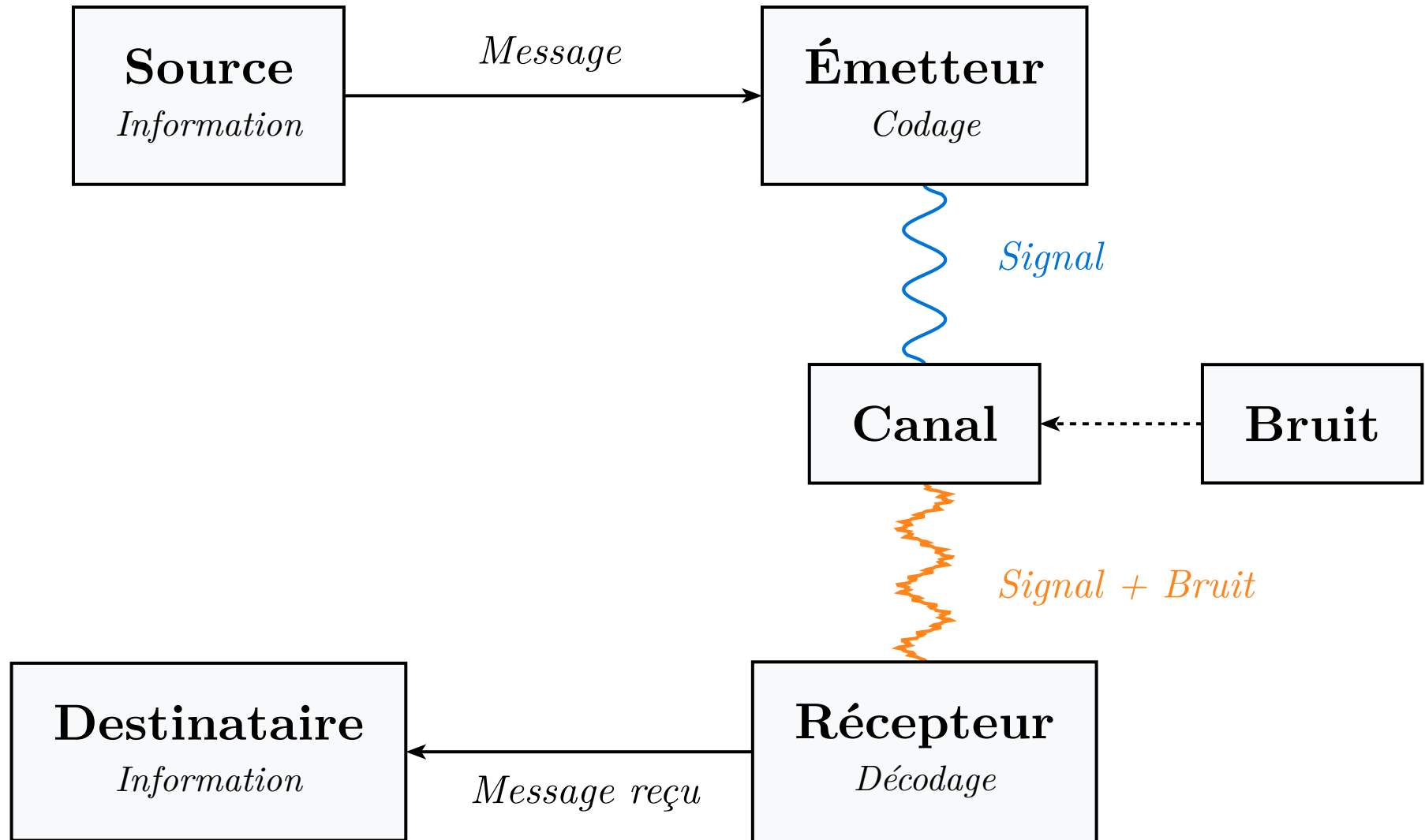


Athena-Fidus



Module OpenIPC

Introduction : Communication Numérique

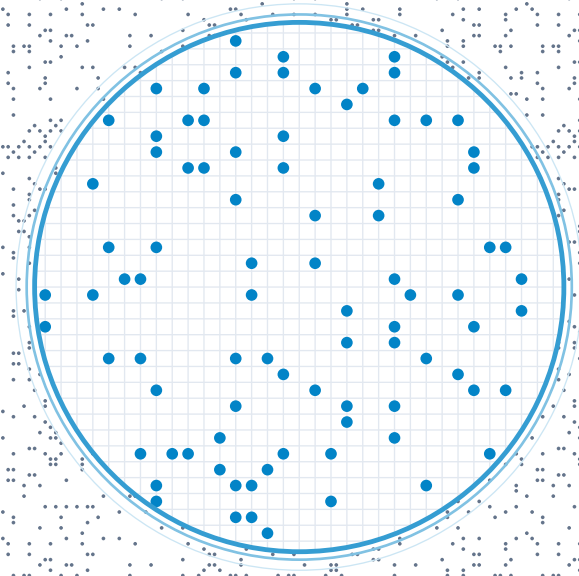


Problématique

Comment utiliser les codes LDPC pour garantir la fiabilité d'une transmission en présence de bruit ?

Plan

- Introduction
- Codes linéaires
- LDPC
- Codage
- Décodage
- Analyse



Définition : Codes Linéaires en Bloc

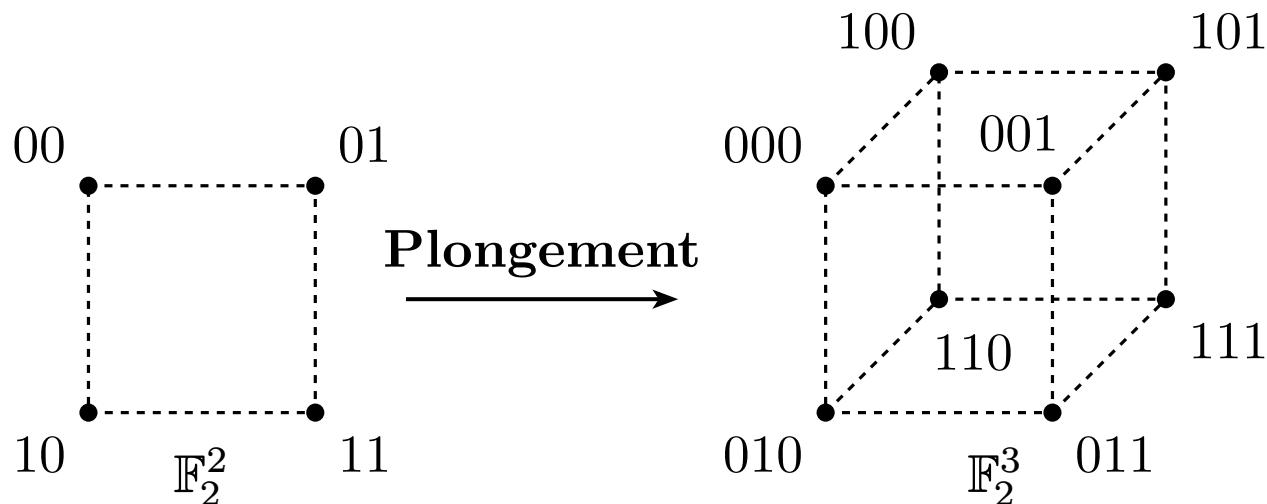
Code $(n, k) \in \mathbb{N}^2$

\mathcal{C} sous-espace vectoriel de dimension k de \mathbb{F}_2^n

- k : longueur du message original
- n : longueur du mot de code
- $m = n - k$: nombre de bits de parités

Encodage

$\Phi : \mathbb{F}_2^k \rightarrow \mathbb{F}_2^n \in \mathcal{L}(\mathbb{F}_2^k, \mathbb{F}_2^n)$



Définition : Matrice Génératrice

Matrice Génératrice

$G \in \mathcal{M}_{k,n}(\mathbb{F}_2)$ dont les lignes sont une base de \mathcal{C}

Encodage

Pour un message $u \in \mathbb{F}_2^k$ le mot de code $c \in \mathcal{C}$ est :

$$c = \Phi(u) = uG$$

Forme systématique

$$G = \begin{bmatrix} I_k & P \end{bmatrix}$$

- $P \in \mathcal{M}_{k,n-k}(\mathbb{F}_2)$ matrice de parité

Définition : Matrice de Contrôle

Matrice de Contrôle

$$H = \begin{bmatrix} P^\top & I_{n-k} \end{bmatrix}$$

- $\mathcal{C} = \ker(H) - Hc^\top = 0 \Rightarrow c \in \mathcal{C}$

Syndrome

Pour un vecteur reçu $r = c + e$, $s \in \mathbb{F}_2^{n-k}$

$$s = Hr^\top = Hc^\top + He^\top = 0 + He^\top$$

- $s = 0 \Rightarrow r \in \mathcal{C}$
- $s \neq 0$ donne la signature de l'erreur e

Exemple d'un code linéaire

Exemple d'un code (5, 2)

- On choisit la matrice de parité P :

$$P = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

- Alors la matrice génératrice G est :

$$G = \left[\begin{array}{cc|ccc} 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{array} \right]$$

- Message $u = [1 \ 1]$
- Mot de code $c = uG$:

$$c = [1 \ 1] \left[\begin{array}{cc|ccc} 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{array} \right] = [1 \ 1 \ 1 \ 0 \ 1]$$

Exemple d'un code linéaire

Structure systématique de H :

$$H = \begin{bmatrix} P^\top & I_3 \end{bmatrix}$$

Ainsi :

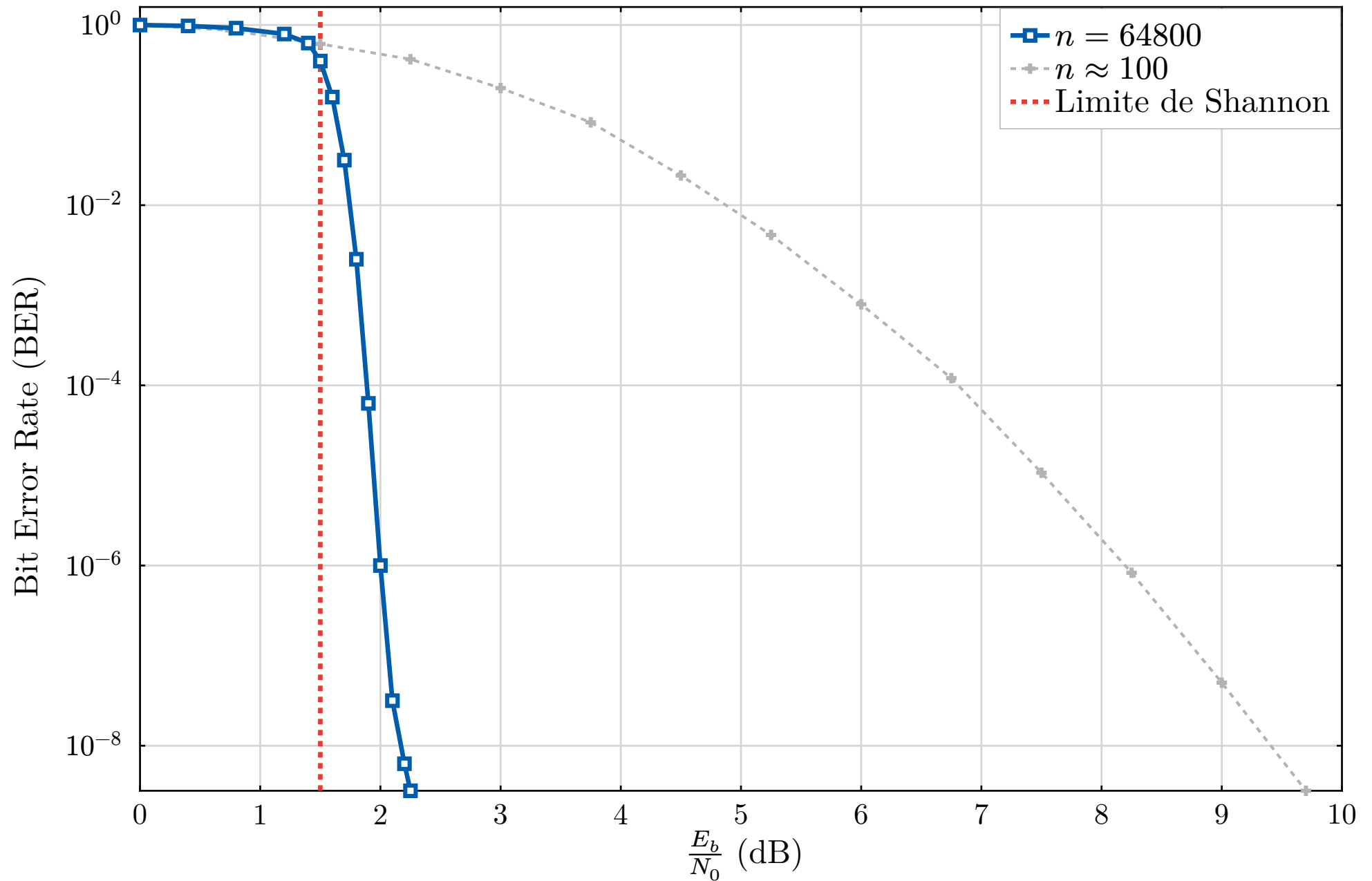
$$H = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Mot de code valide $c = (1, 1, 1, 0, 1)$: $Hc^\top = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$ ✓

Mot reçu avec une erreur : $r = (1, 1, 0, 0, 1)$

$$Hr^\top = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 & 1 \end{bmatrix}^\top = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \neq \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Approcher la Limite de Shannon



Le Mur de la Complexité

Décodage par Maximum de Vraisemblance

Chercher le mot de code $\mathbf{c} \in \mathcal{C}$ le plus probable sachant \mathbf{r} reçu :

$$\hat{\mathbf{c}} = \arg \min_{\mathbf{c} \in \mathcal{C}} d_H(\mathbf{r}, \mathbf{c})$$

- Équivalent à chercher l'erreur \mathbf{e} de poids minimal tel que $\mathbf{H}\mathbf{e}^\top = \mathbf{s}$.

Le Problème du décodage par Syndrome

NP-Difficile et pour H quelconque : $\mathcal{O}(2^k)$

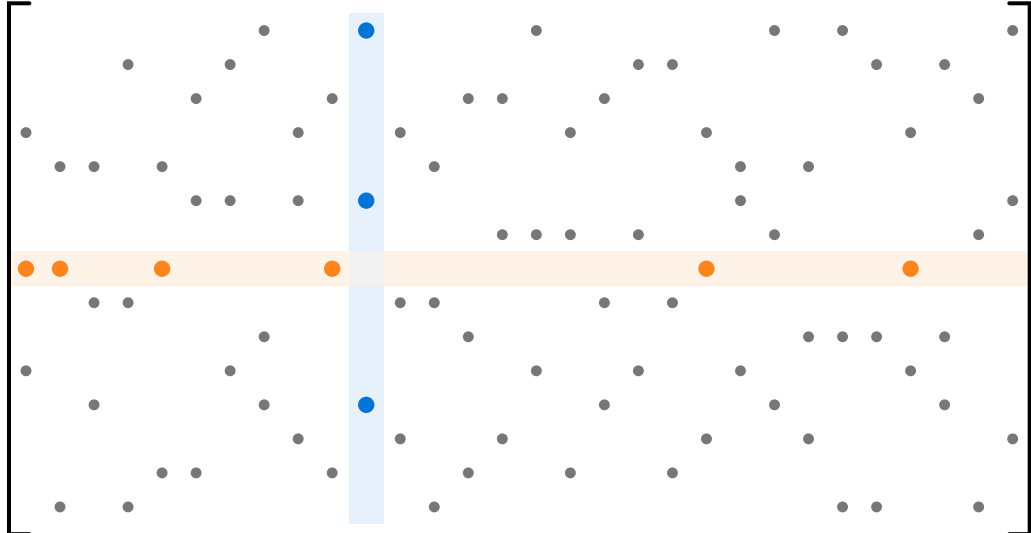
Définition des Codes LDPC

Codes LDPC Réguliers

Code linéaire en bloc avec une matrice de contrôle H clairsemée.

- Poids de Colonne w_c
- Poids de Ligne w_r

$H \in \mathcal{M}_{15,30}(\mathbb{F}_2), R = \frac{1}{2}$

$H =$ 

Faible Densité

$$w_c \ll m \quad w_r \ll n$$

Rendement

$$R = 1 - \frac{m}{n}$$

De la Matrice aux Équations de Parité

$$H = \begin{bmatrix} \text{dots} \\ \text{dots} \\ \text{dots} \\ \text{dots} \\ \text{dots} \\ \text{dots} \end{bmatrix} \begin{bmatrix} r_0 \\ r_1 \\ \vdots \\ r_{29} \end{bmatrix}$$

Mot reçu $r \in \mathbb{F}_2^{30}$

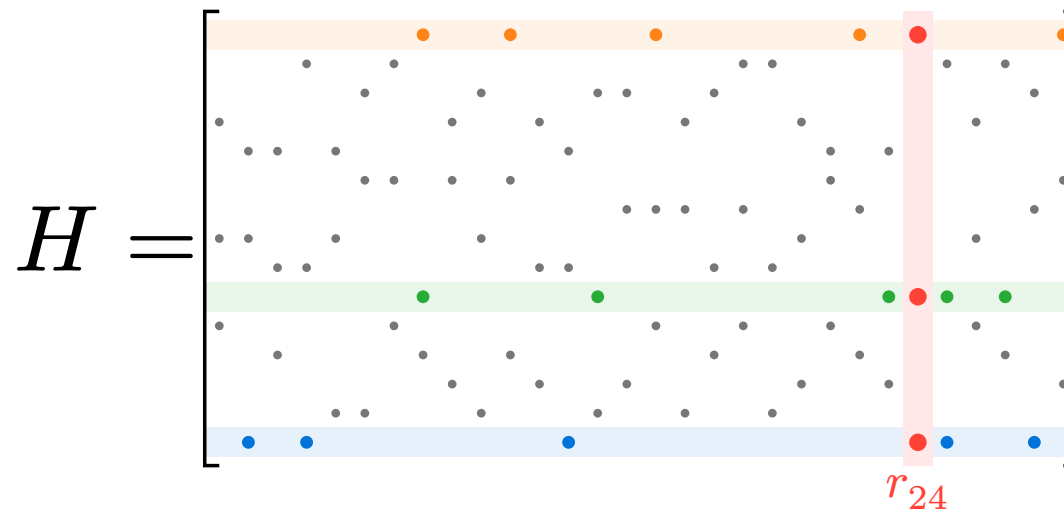
- L_j définit une équation de parité f_j
- Pour r , on vérifie le syndrome : $Hr^\top = 0$

Équations de Parité

$$f_0 : r_7 \oplus r_{10} \oplus r_{15} \oplus r_{22} \oplus r_{24} \oplus r_{29} = 0$$

- Si $f_j = 1$, un nombre impair de bits a été inversé par le canal.

L'Entrelacement des Contraintes



- Chaque bit r_i participe à $w_c = 3$ équations distinctes :

$$\begin{cases} r_7 \oplus r_{10} \oplus r_{15} \oplus r_{22} \oplus r_{24} \oplus r_{29} = 0 & (f_0) \\ r_7 \oplus r_{13} \oplus r_{23} \oplus r_{24} \oplus r_{25} \oplus r_{27} = 0 & (f_9) \\ r_1 \oplus r_3 \oplus r_{12} \oplus r_{24} \oplus r_{25} \oplus r_{28} = 0 & (f_{14}) \end{cases}$$

- r_{24} : Surveillé simultanément par f_0 , f_9 et f_{14} .
- Si $\forall j \in \{0, 9, 14\}$, $f_j = 1$, alors le bit est considéré suspect.

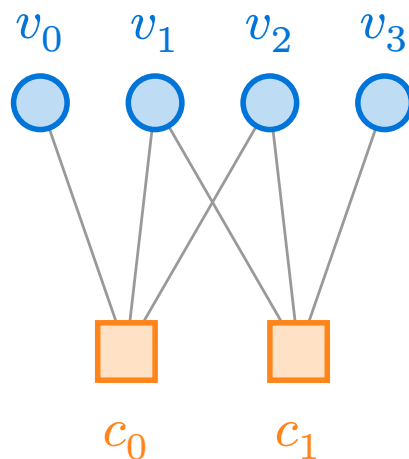
Graphe de Tanner : Définition

Graphe de Tanner $\mathcal{G}(H)$

Graphe bipartite $\mathcal{G} = (V \sqcup C, A)$:

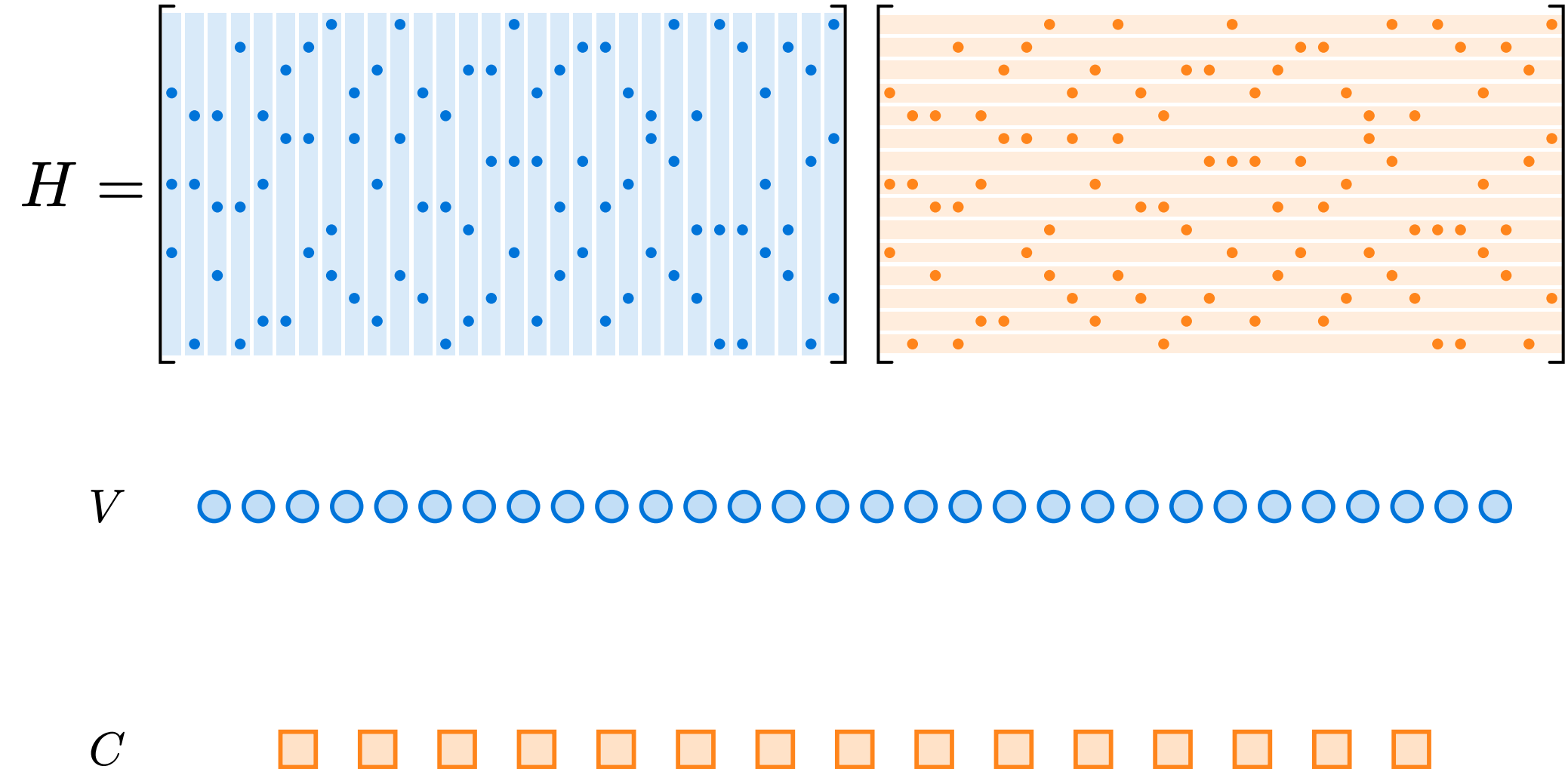
$$(v_j, c_i) \in A \iff H_{i,j} = 1$$

- $V = \{v_0, \dots, v_{n-1}\}$ nœuds de **variable**
- $C = \{c_0, \dots, c_{m-1}\}$ nœuds de **contrôle**
- $\deg(v_j) = w_c$
- $|A| = n \cdot w_c = m \cdot w_r$
- $H \cong \mathcal{G}$
- $\deg(c_i) = w_r$

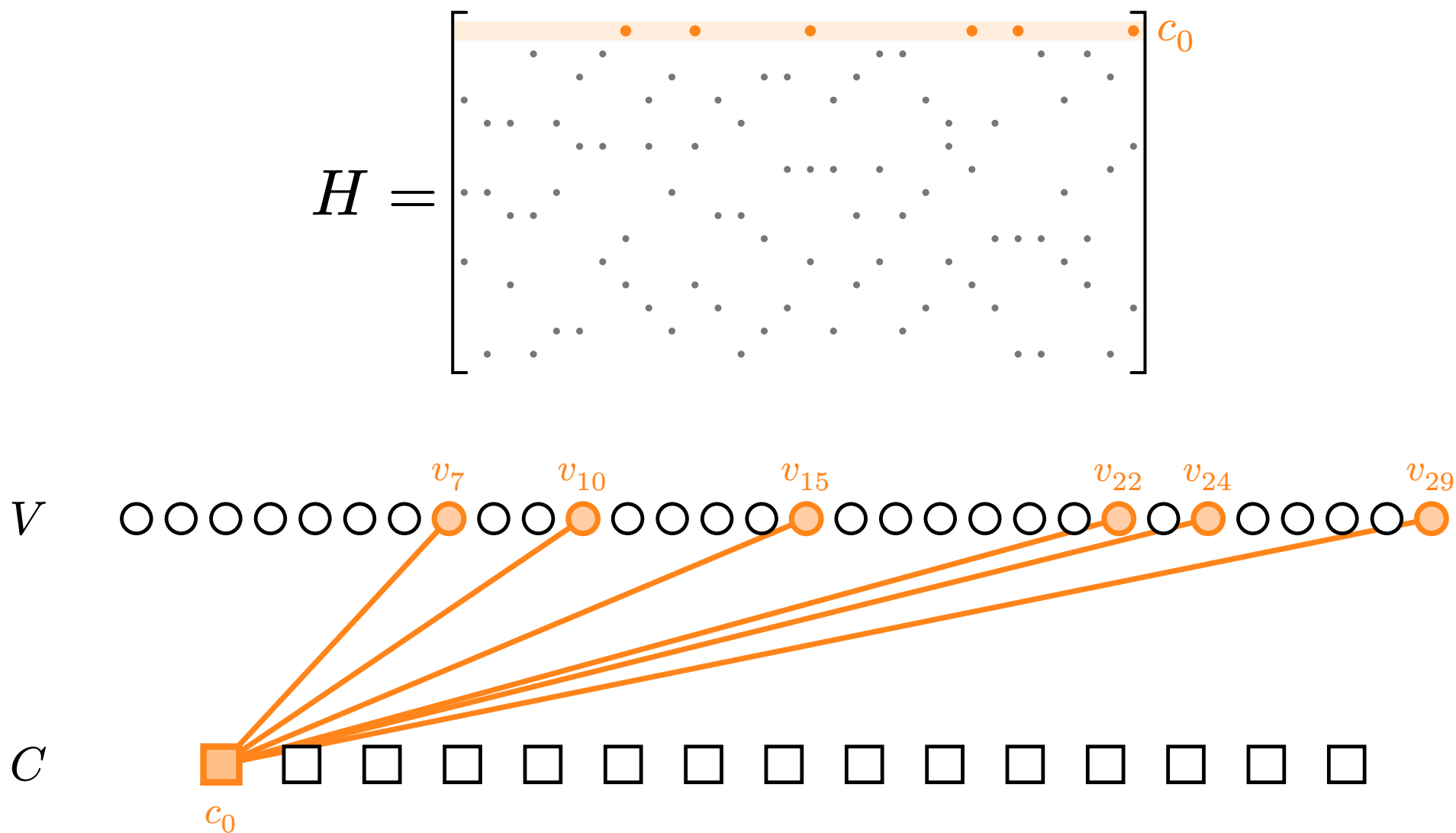


Exemple $n = 4, m = 2$

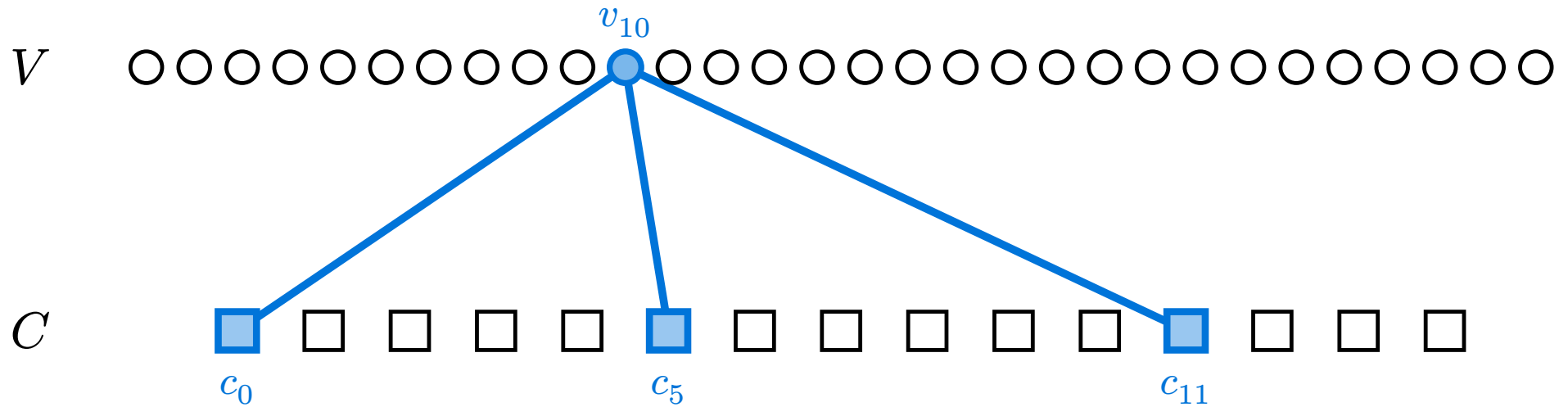
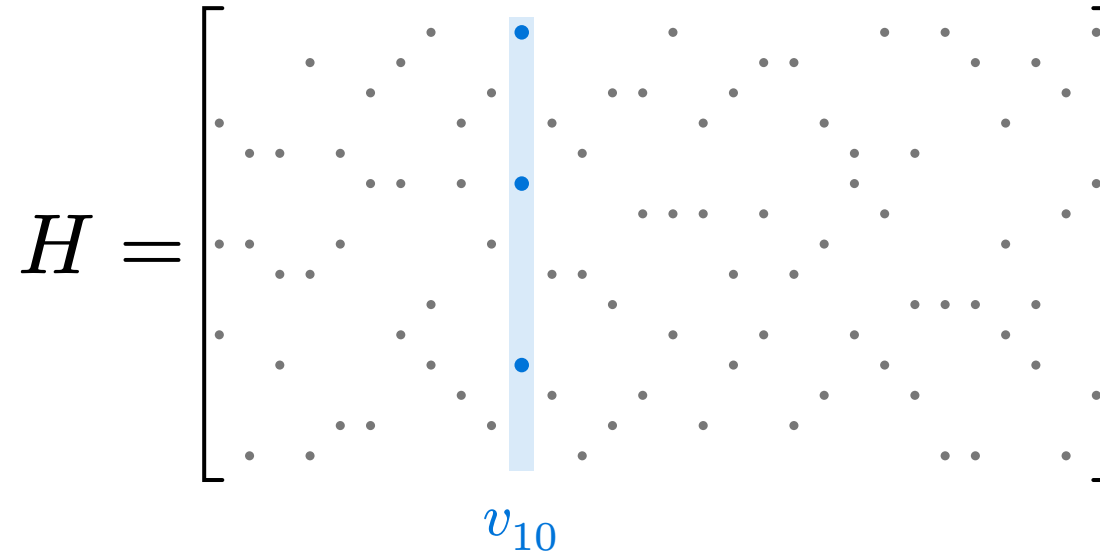
Construction du Graphe : Les Nœuds



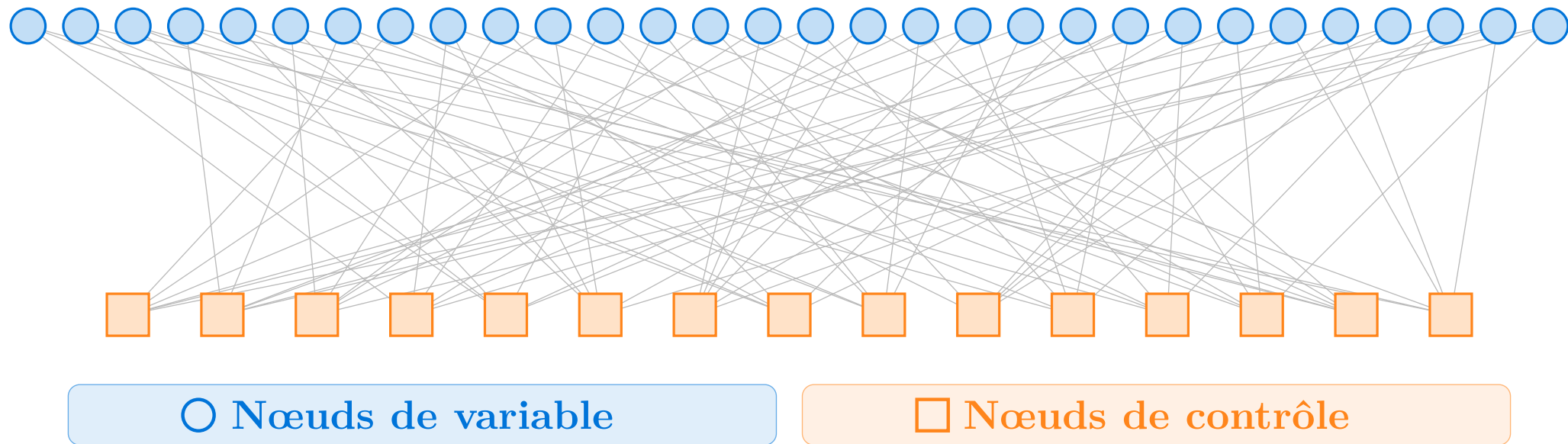
Construction du Graphe : Nœud de Contrôle



Construction du Graphe : Nœud de Variable



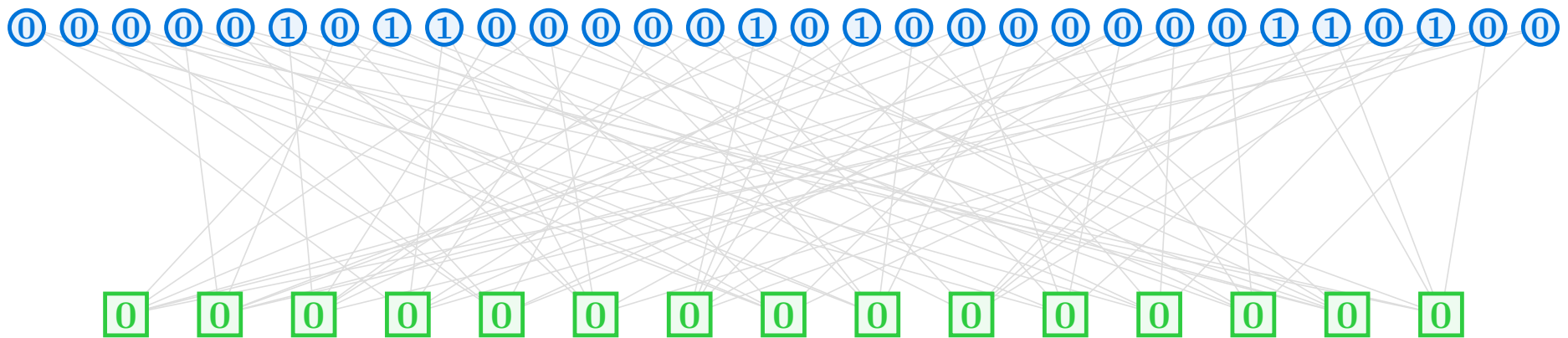
Graphe de Tanner Final





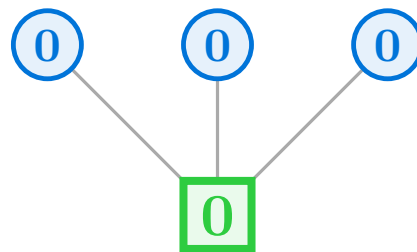
La Contrainte de Somme Nulle

Vision Graphe

Si $s = 0$ alors que chaque nœud de contrôle est localement satisfait



Chaque  calcule le xor de ses voisins  : $f_i = \bigoplus_{j \in \mathcal{N}(c_i)} v_j$

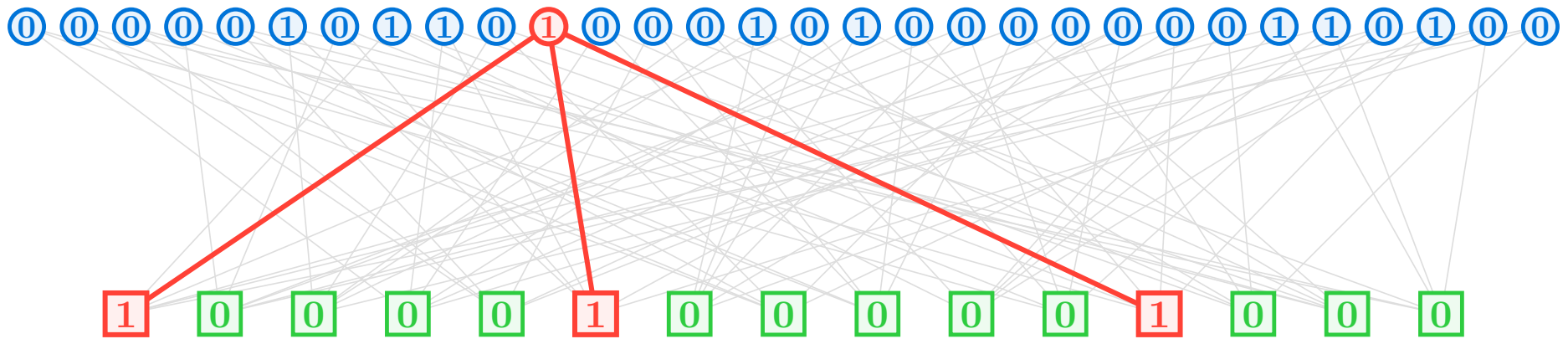


$$0 \oplus 0 \oplus 0 = 0$$

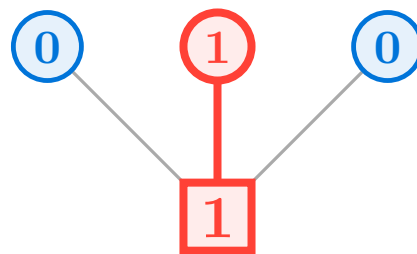
La Contrainte de Somme Nulle

Détection d'Erreur

Si un bit est inversé, toutes les contraintes associées sont à 1



$$0 \oplus 1 \oplus 0 = 1 \rightarrow \text{Erreur détectée}$$

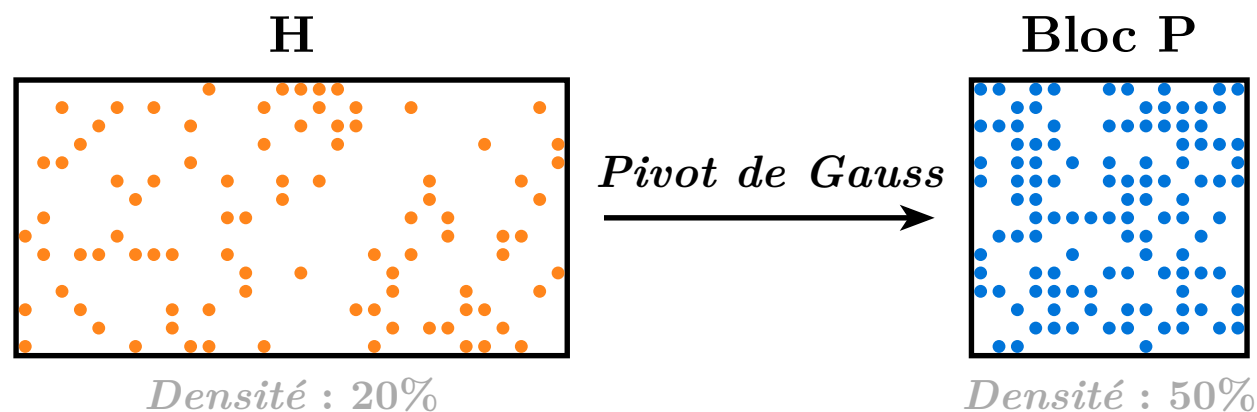


$$0 \oplus 1 \oplus 0 = 1$$

Encodage LDPC : Calcul de G

Encodage

Mot de code \mathbf{c} généré à partir d'un message \mathbf{u} : $\mathbf{c} = \mathbf{u}\mathbf{G}$



- Forme Systématique

$$\mathbf{H} = \begin{bmatrix} \mathbf{P}^\top & \mathbf{I}_{n-k} \end{bmatrix} \longrightarrow \mathbf{G} = \begin{bmatrix} \mathbf{I}_k & \mathbf{P} \end{bmatrix}$$




- La matrice \mathbf{G} devient dense \Rightarrow encodage en $\mathcal{O}(n^2)$

Décodage : Bit-Flipping

Décision Stricte (Hard Decision)

Algorithme **itératif** : les nœuds **échangent des bits** pour localiser les erreurs.

Message Passing




-  envoie son bit courant à ses voisins 
 -  renvoie son **verdict de parité** (0 ou 1)
-
- Si v_j participe à **trop d'équations non satisfaites** \Rightarrow on l'inverse.

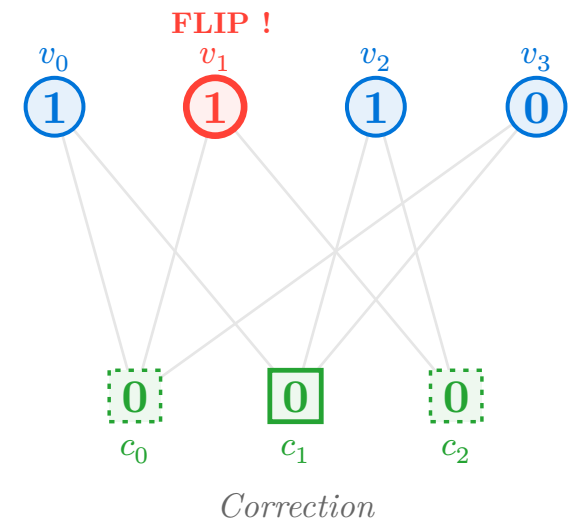
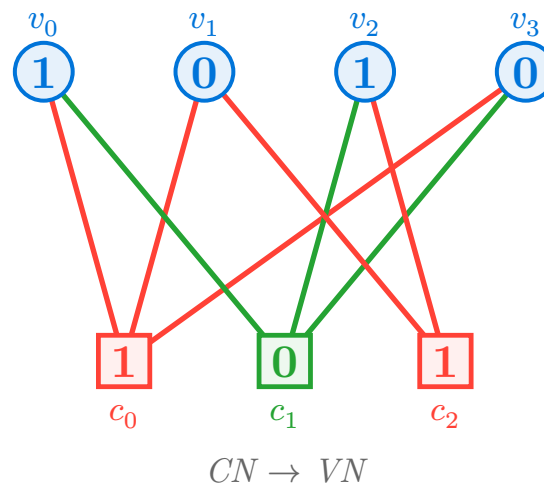
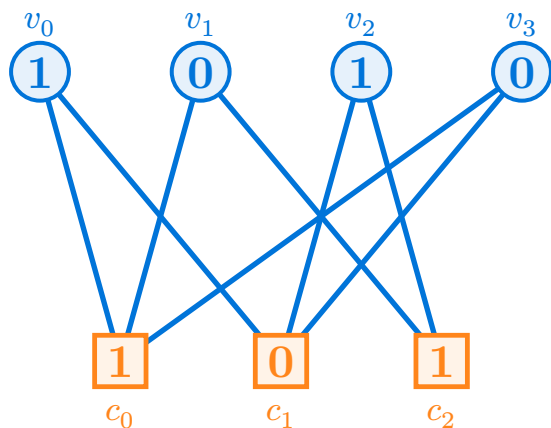
Décodage : Bit-Flipping

Décision Stricte (Hard Decision)

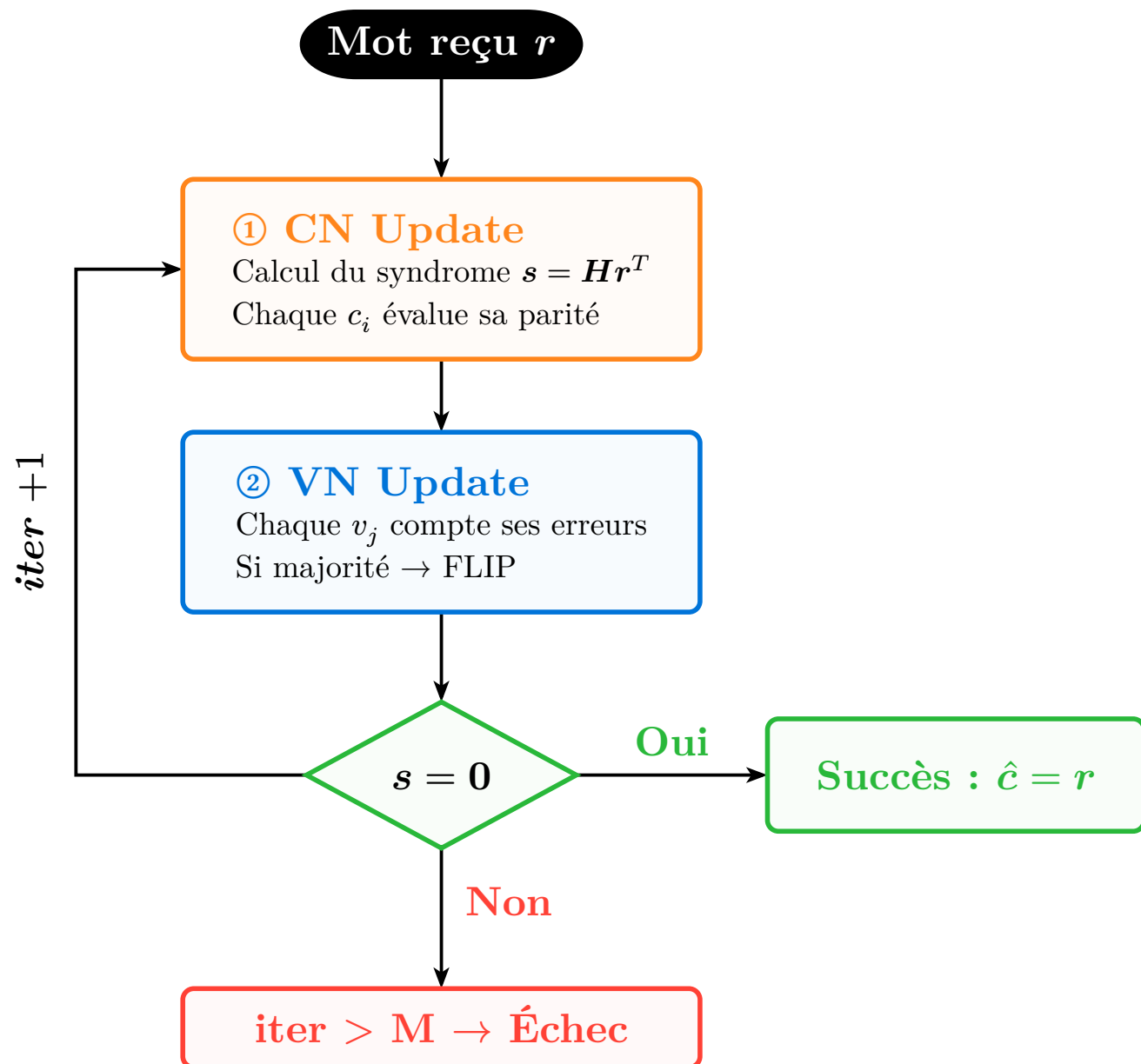
Algorithme **itératif** : les nœuds **échangent des bits** pour localiser les erreurs.

Message Passing

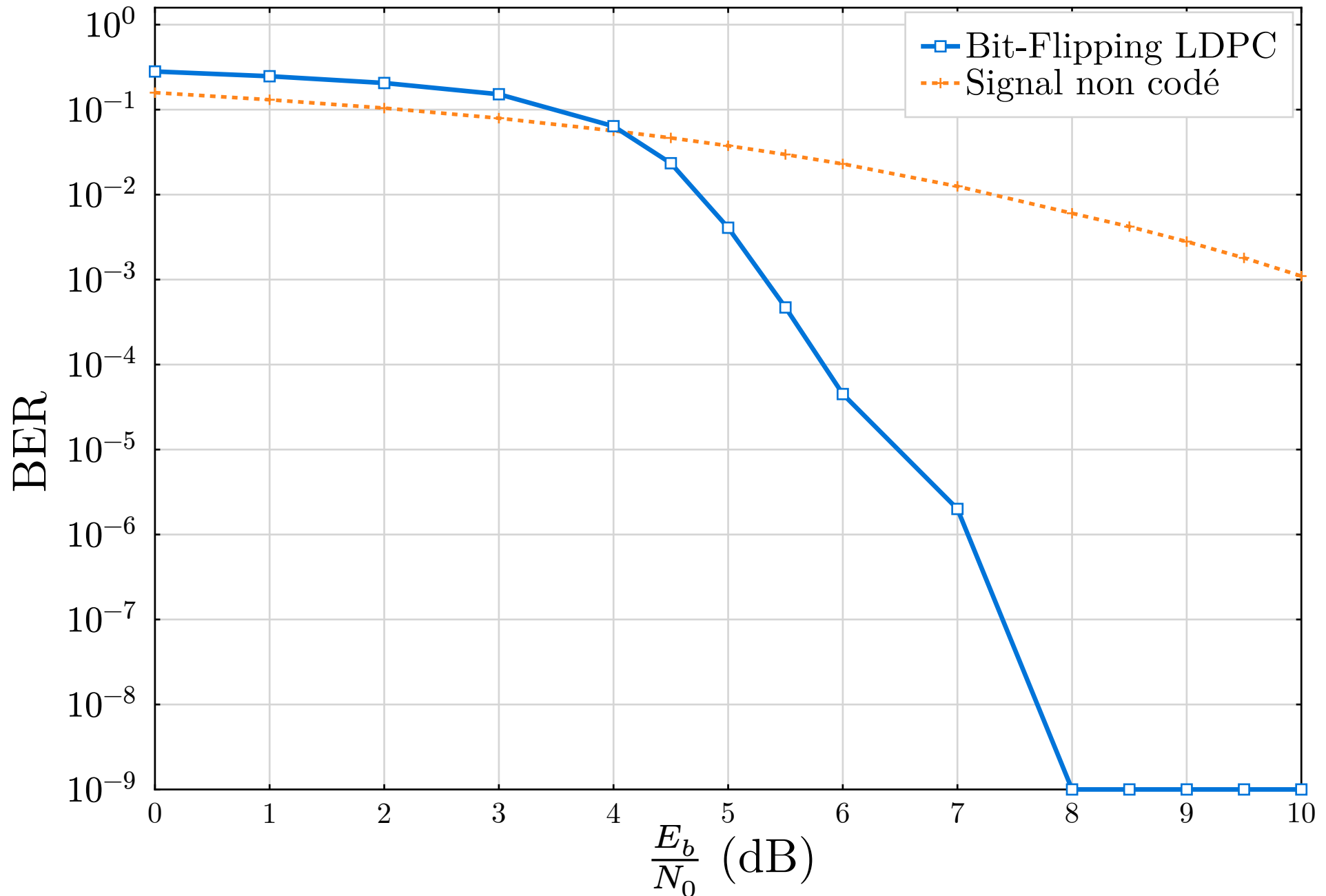
-  envoie son bit courant à ses voisins 
 -  renvoie son **verdict de parité** (0 ou 1)
- Si v_j participe à **trop d'équations non satisfaites** \Rightarrow on l'inverse.



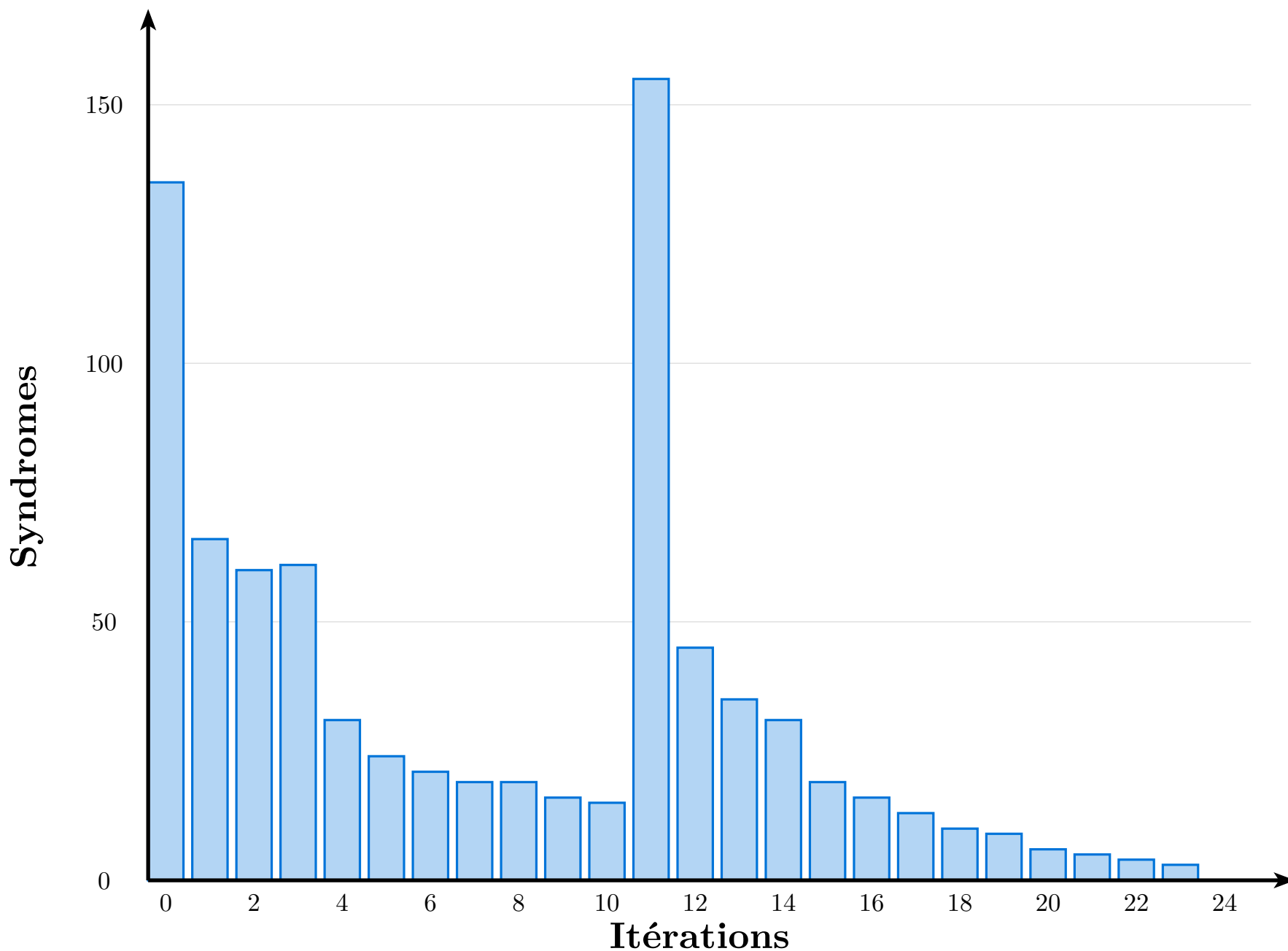
Bit-Flipping : Graphe de flot de contrôle



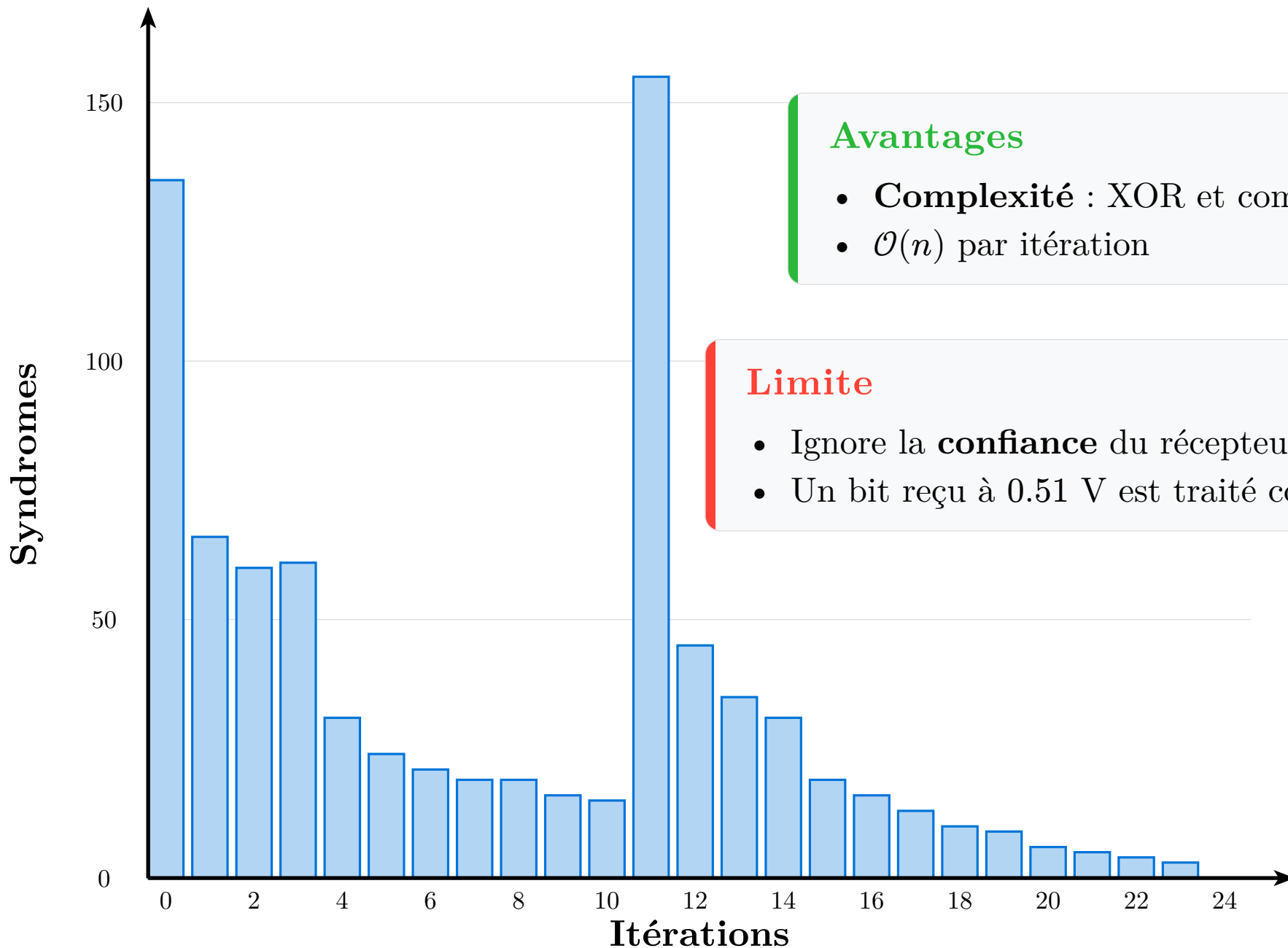
Waterfall : LDPC (3, 6) $n = 1296$, $k = 648$, $R = \frac{1}{2}$



Bit-Flipping : Syndrome et Analyse



Bit-Flipping : Syndrome et Analyse



Avantages

- **Complexité** : XOR et compteurs
- $\mathcal{O}(n)$ par itération

Limite

- Ignore la **confiance** du récepteur physique
- Un bit reçu à 0.51 V est traité comme 0

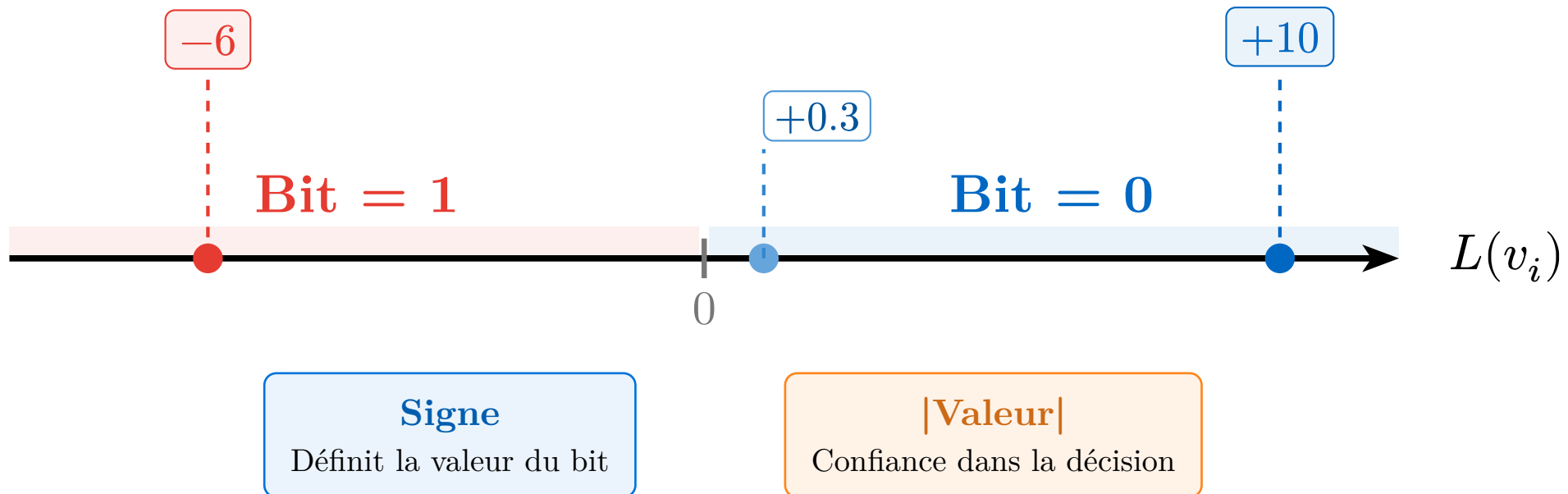
Décodage Soft : Le LLR

Signal

On reçoit une valeur y_i (ex: +4.5V ou -0.2V).

Log-Likelihood Ratio (LLR)

$$L(v_i) = \ln \left(\frac{P(v_i = 0 \mid y_i)}{P(v_i = 1 \mid y_i)} \right)$$



Sum-Product : Belief Propagation

Décodage Optimal

Échange itératif de croyances (LLR) entre les nœuds du graphe

Information Extrinsèque

Exclure l'avis du destinataire pour éviter l'auto-influence

Mise à jour 

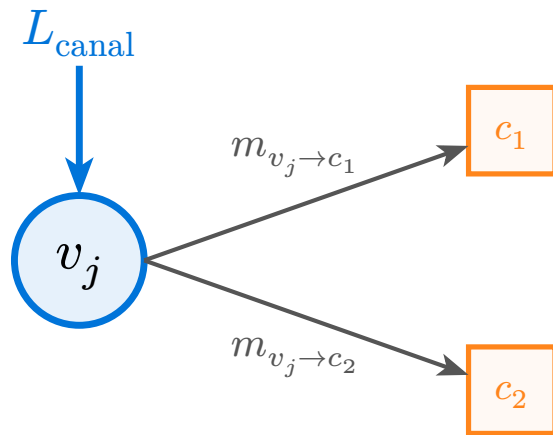
$$m_{c \rightarrow v} = 2 \tanh^{-1} \left(\prod_{u \in \mathcal{N}(c) \setminus \{v\}} \tanh \left(\frac{m_{u \rightarrow c}}{2} \right) \right)$$

Mise à jour 

$$m_{v \rightarrow c} = L_{v\text{canal}} + \sum_{c' \in \mathcal{N}(v) \setminus \{c\}} m_{c' \rightarrow v}$$

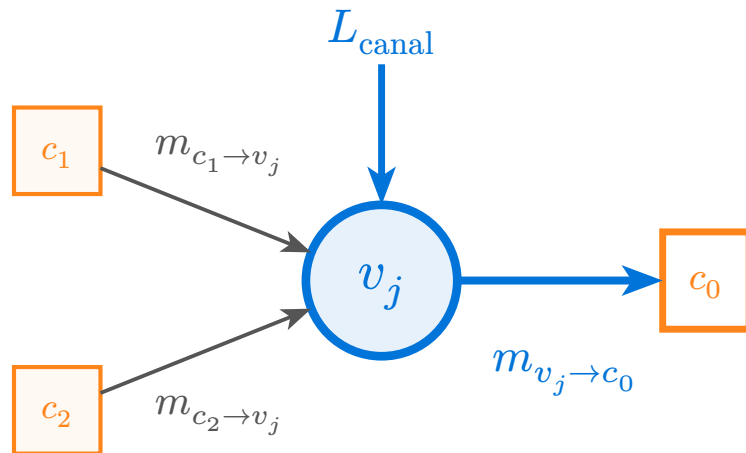
Sum-Product

Initialisation

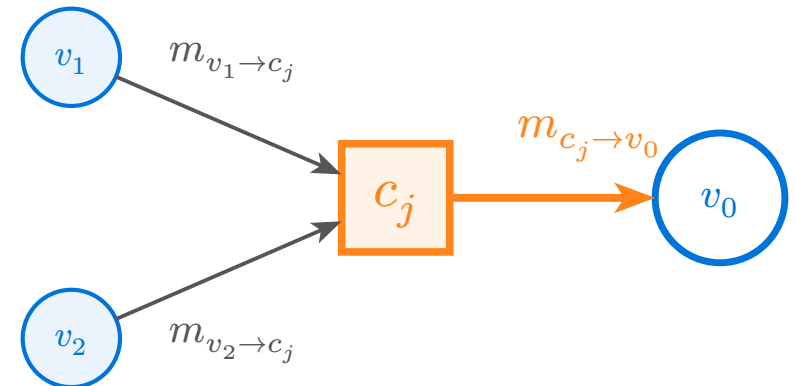


$$m_{v_j \to c_i} = L_{\text{canal}}$$

Échange ○

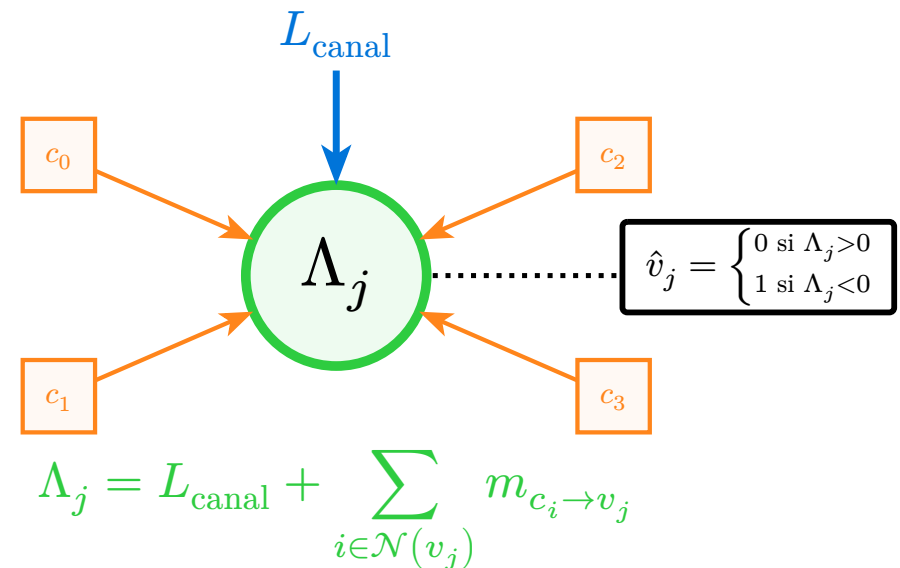


Échange □



Itérations
 $i = 1, \dots, I_{\text{max}}$

Décision Finale

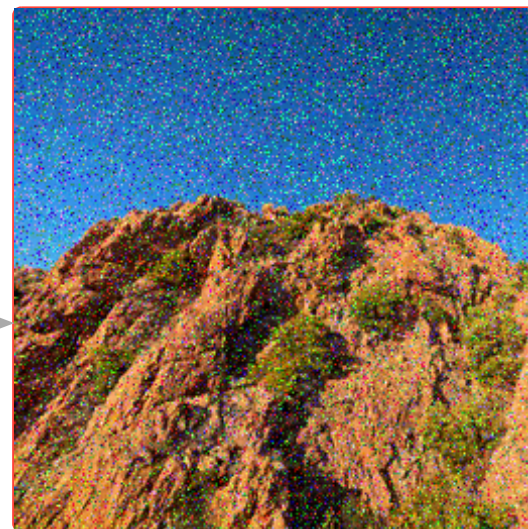


Transmission d'image

Originale



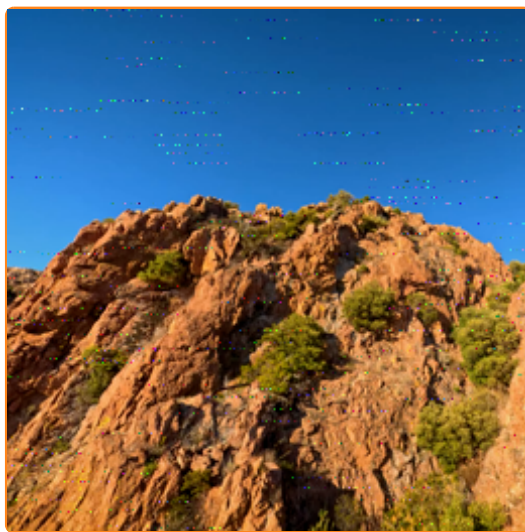
Reçue (Bruité)



AWGN (2.3 dB)



$$R = \frac{1}{2}$$



$$R = \frac{2}{3}$$



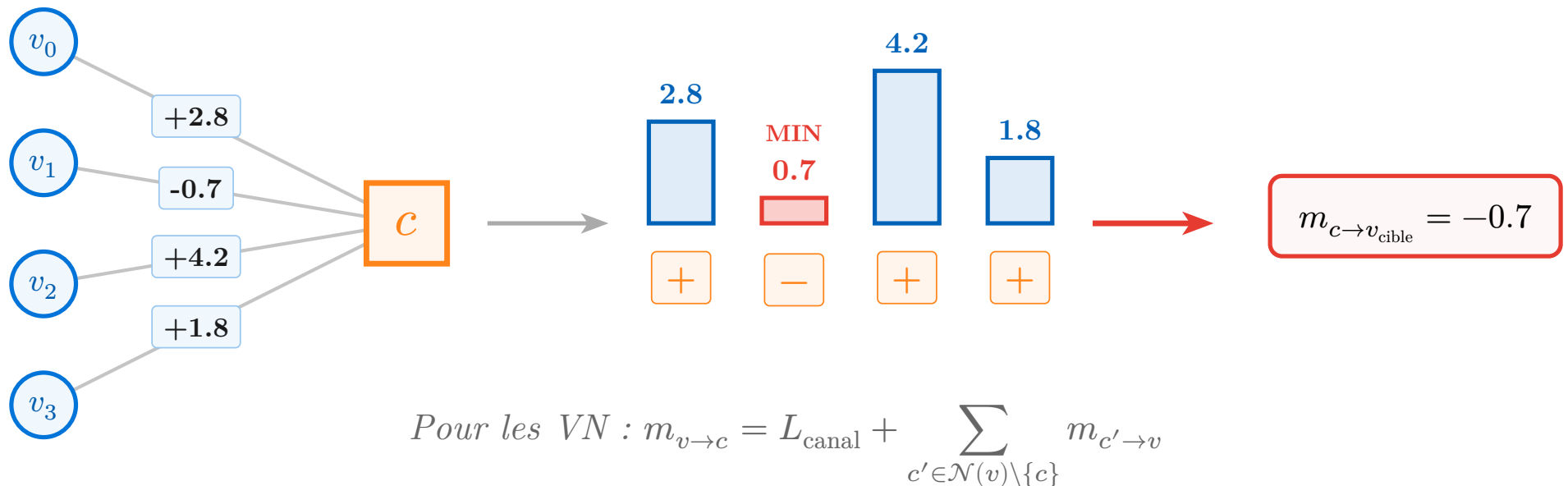
$$R = \frac{3}{4}$$

Avantage Matériel

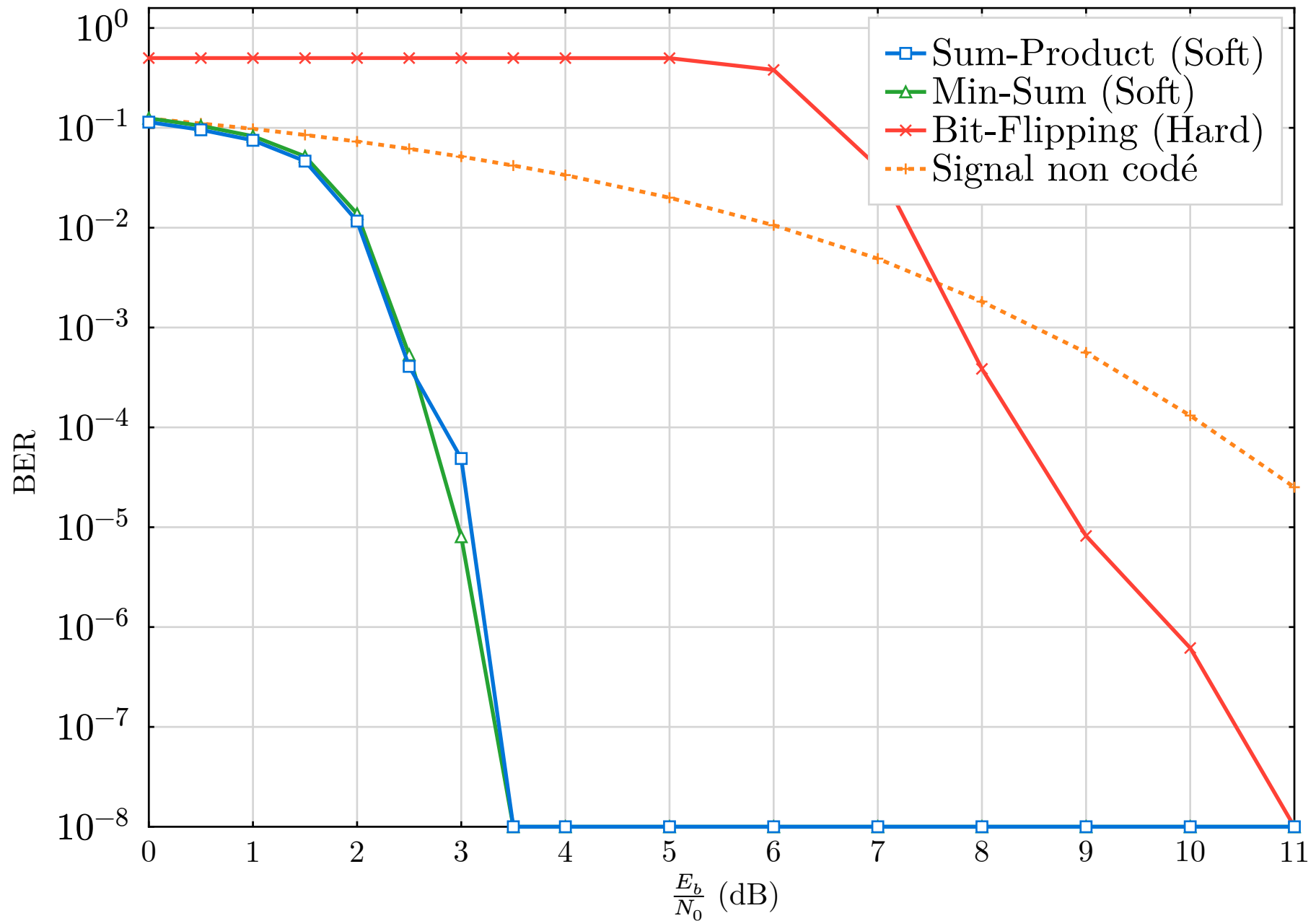
- **Comparateurs** pour le minimum
- **XOR** pour le produit des signes

Mise à jour des CN

$$m_{c \rightarrow v} = \prod_{u \in \mathcal{N}(c) \setminus \{v\}} \text{sgn}(m_{u \rightarrow c}) \times \min_{u \in \mathcal{N}(c) \setminus \{v\}} |m_{u \rightarrow c}|$$



Waterfall : LDPC (3, 9) $n = 1296$, $k = 864$, $R = \frac{2}{3}$



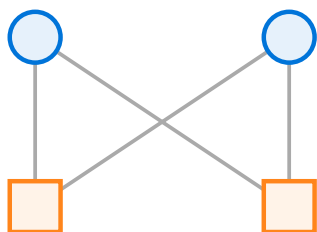
La Topologie de H : Le Girth

Définition : Le Girth (La Maille)

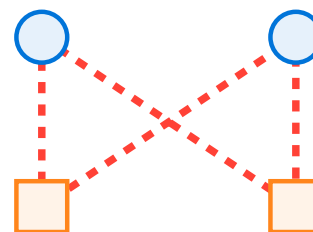
Longueur du plus court cycle dans le graphe de Tanner

- Le girth est **pair**
- La valeur minimale est $g = 4$.

Girth élevé \Rightarrow Meilleure diffusion de l'information.

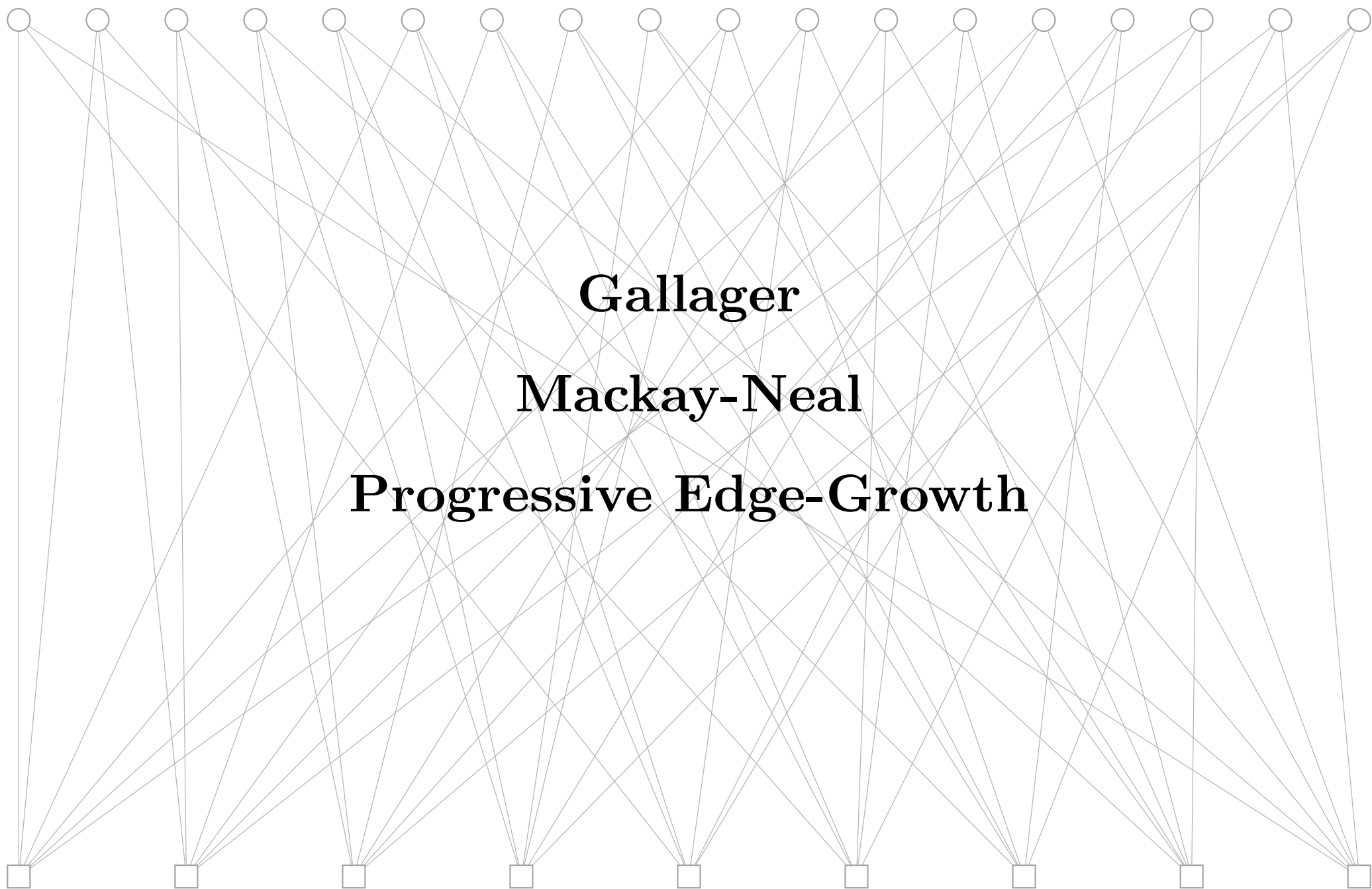


Graphe de Tanner



4-Cycle

Méthode de génération de H



Conclusion

QC-LDPC Encodage

FPGA

Test Réels

Annexe

Théorie derrière la définition des codes linaires

Poser les notations algébriques etc...

Métriques : BER et FER

Bit Error Rate (BER)

$$\text{BER} = \frac{\text{Nombre de bits incorrects reçus}}{\text{Nombre total de bits transmis}}$$

Frame Error Rate (FER)

$$\text{FER} = \frac{\text{Nombre de trames avec au moins 1 bit incorrect}}{\text{Nombre total de trames}}$$

Courbe Waterfall : BER en fonction de E_b/N_0 (dB) — représente les performances

$$\text{FER} \geq \text{BER}$$

Une seule erreur de bit invalide toute la trame

- E_b/N_0 en dB = $10 \log_{10}(E_b/N_0)$

Décodage par Maximum de Vraisemblance (ML)

Décodeur ML — Canal AWGN + BPSK

$$\hat{\mathbf{c}} = \arg \max_{\mathbf{c} \in \mathcal{C}} P(\mathbf{r} \mid \mathbf{c}) = \arg \min_{\mathbf{c} \in \mathcal{C}} d_H(\mathbf{r}, \mathbf{c})$$

Minimiser la distance de Hamming au mot de code le plus proche.

Décodage par Syndrome (Cosets)

- Calculer $\mathbf{s} = \mathbf{H}\mathbf{r}^\top$
- Chercher \mathbf{e} de poids minimal tel que $\mathbf{H}\mathbf{e}^\top = \mathbf{s}$
- Table précalculée de 2^{n-k} entrées \rightarrow faisable seulement si $n - k$ petit

Complexité — NP-Difficile (Berlekamp, 1978)

Pour \mathbf{H} quelconque : $\mathcal{O}(2^k)$ candidats à tester.

- $k = 648 : 2^{648} \approx 10^{195}$ — hors de portée
- Solution : exploiter la **structure creuse** de $\mathbf{H} \rightarrow$ BP itératif

Codes LDPC Irréguliers

Distribution de Degrés (polynômes sur les arêtes)

$$\lambda(x) = \sum_i \lambda_i x^{i-1}, \quad \rho(x) = \sum_j \rho_j x^{j-1}$$

λ_i = fraction d'arêtes reliées à des VN de degré i , idem pour ρ_j et les CN.

Avantages

- Approchent la limite de Shannon de plus près
- VN de degré 2 : accélèrent la convergence initiale du BP

Optimisation

Density Evolution : calcule analytiquement le seuil de décodage en fonction de $(\lambda, \rho) \rightarrow$ optimiser numériquement

- Contrainte : $n \sum_i \lambda_i / i = |E| = m \sum_j \rho_j / j$
- DVB-S2 ($n = 64800$) : degrés variables entre 2 et 13 selon le rendement cible

Encodage Efficace : Richardson-Urbanke

Problème : Forme systématique $\rightarrow G$ dense $\rightarrow \mathcal{O}(n^2)$ opérations.

Forme ALT : Approximate Lower Triangular

Par permutation des lignes/colonnes de H :

$$H = \begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix}$$

où T est **triangulaire inférieure** de taille $(m - g) \times (m - g)$, $g = \text{gap}$ (très petit).

Encodage en $\mathcal{O}(n + g^2)$

Résolution en 2 phases :

1. Calculer $p_1 \in \mathbb{F}_2^g$ par élimination sur système de taille g
2. Calculer $p_2 \in \mathbb{F}_2^{m-g}$ par back-substitution sur T (triangulaire)

- En pratique (QC-LDPC / 5G / DVB-S2) : encodage par **registres à décalage** $\rightarrow \mathcal{O}(n)$

Canal AWGN

Modèle du Canal — BPSK sur AWGN

$$y_i = x_i + n_i, \quad x_i \in \{+1, -1\}, \quad n_i \sim \mathcal{N}(0, \sigma^2)$$

Mapping BPSK : bit 0 $\mapsto +1$, bit 1 $\mapsto -1$

Rapport Signal sur Bruit

$$\frac{E_b}{N_0} = \frac{1}{2R\sigma^2}$$

$R = k/n$: rendement du code. Exprimé en dB : $10 \log_{10}(E_b/N_0)$.

LLR Initial sur Canal AWGN

$$L_{\text{canal}(y_i)} = \ln \frac{P(v_i = 0 \mid y_i)}{P(v_i = 1 \mid y_i)} = \frac{2y_i}{\sigma^2}$$

Le LLR est **proportionnel** à la valeur reçue y_i : signe = décision, valeur absolue = confiance.

Construction de Gallager (1962)

Principe

Empiler w_c sous-matrices de taille $(m/w_c) \times n$:

$$H = \begin{bmatrix} H_1 \\ H_2 \\ \vdots \\ H_{w_c} \end{bmatrix}$$

Algorithme :

- H_1 : blocs réguliers de w_r uns consécutifs (colonnes disjointes)
- H_2, \dots, H_{w_c} : copies de H_1 avec colonnes **permutées aléatoirement**

Avantage

Simple à construire.

Garantit w_c et w_r exacts.

Limite

Cycles de longueur 4 fréquents.

Aucun contrôle du girth.

Construction de MacKay-Neal (1996)

Principe

Construction **aléatoire** de \mathbf{H} avec évitement actif des 4-cycles.

Algorithme — pour chaque arête à placer (v_j, c_i) :

1. Vérifier l'absence de 4-cycle : $\nexists(j', i')$ tel que $H_{i,j'} = H_{i',j} = H_{i',j'} = 1$
2. Si conflit : **rejeter** c_i et tirer un autre nœud de contrôle
3. Sinon : ajouter l'arête

Résultat

- Garanti **sans 4-cycles** par construction \rightarrow girth ≥ 6
- Performances nettement supérieures à Gallager

Progressive Edge-Growth (Hu et al., 2005)

Idée

Construire les arêtes **une par une** en maximisant le **girth local** à chaque étape.

Algorithme — pour relier v_j à un nouveau CN :

1. BFS depuis v_j dans le graphe courant
2. S'arrêter quand tous les CN ne sont plus accessibles à un nouveau niveau
3. Relier v_j au CN de plus faible degré **non encore atteint** par le BFS

Avantage

Maximise le girth global.
Surpasse Gallager et MacKay.
Standard de référence.

Complexité

$\mathcal{O}(n \cdot w_c \cdot m)$
Coût en construction seulement,
pas en décodage.

Limite de Shannon : Canal AWGN

Capacité du Canal AWGN (Shannon, 1948)

$$C = \frac{1}{2} \log_2(1 + \text{SNR}) \quad [\text{bit} / \text{utilisation}]$$

Il existe un code de rendement $R < C$ avec $\text{BER} \rightarrow 0$ quand $n \rightarrow \infty$.

- Pour $R = \frac{1}{2}$: limite à $E_b/N_0 \approx 0.19$ dB
- **Bit-Flipping** : 5–6 dB de la limite
- **Sum-Product** : 1–1.5 dB de la limite

Pourquoi de grands blocs ?

Loi des grands nombres : pour n grand, le bruit moyen par bit converge vers σ^2 . Plus n est grand, plus on s'approche de la limite — au prix de la latence.

DVB-S2 : $n = 64800 \rightarrow$ à 0.5 dB de Shannon pour $R = \frac{1}{2}$.

Girth : Impact sur la Convergence du BP

BP Exact \rightarrow Graphe = Arbre

Sans cycle, le BP est **exact** et converge en au plus diamètre(graphe) itérations.

Problème des Cycles Courts

Un cycle de longueur $2l$: après l itérations, un message revient à son point de départ. \rightarrow **Violation de l'indépendance** des messages \rightarrow BP sous-optimal.

- **Girth = 4** : deux VN partagent deux CN \rightarrow corrélation immédiate, convergence vers solution incorrecte
- **Girth = 6** : premier retour après 3 itérations \rightarrow stable pour quelques itérations

$\text{girth} \geq 2I_{\max}$ pour que les cycles n'affectent pas les I_{\max} premières itérations.

QC-LDPC : Codes Quasi-Cycliques

Structure de H — Blocs Circulants

$$H = \begin{bmatrix} \Pi^{p_{0,0}} & \dots & \Pi^{p_{0,n_b-1}} \\ \vdots & \ddots & \vdots \\ \Pi^{p_{m_b-1,0}} & \dots & \Pi^{p_{m_b-1,n_b-1}} \end{bmatrix}$$

Π^p = identité décalée de p lignes ($p = -1 \rightarrow$ matrice nulle)

Avantages

- **Stockage** : seulement la matrice des exposants $p_{i,j}$ (taille $m_b \times n_b$)
 - **Encodage** : registres à décalage $\rightarrow \mathcal{O}(n)$ simple en FPGA
 - **Décodage** : Z noeuds traités en parallèle par bloc circulant
-
- DVB-S2 : $Z = 360$, $m_b \times n_b = 45 \times 90 \rightarrow n = 64800$

Convergence du BP : Comportement et Critères

Hypothèse du BP

Messages entrants en un nœud supposés **indépendants**. → Exact sur arbre, **approché** sur graphe à grand girth.

Critères d'Arrêt

- **Syndrome nul** : $H\hat{c}^\top = \mathbf{0} \rightarrow$ succès, on s'arrête
 - I_{\max} **itérations** atteint \rightarrow échec \rightarrow paquet perdu (compte dans le FER)
-
- En pratique : $I_{\max} = 50\text{--}200$ selon le code et le SNR
 - Oscillations possibles autour d'un **pseudo-mot de code** (trapping set)
 - Plus le SNR est élevé, moins d'itérations nécessaires

Analyse théorique : **Density Evolution** — calcule le seuil exact de convergence en fonction de E_b/N_0 .

CN Update : Formalisme probabiliste

Soit $(V_u)_{u \in \mathcal{N}(c)}$ une famille de variables aléatoires mutuellement indépendantes à valeurs dans \mathbb{F}_2 . On cherche à déterminer la loi du message $R_{c \rightarrow v}(b)$ envoyé par le nœud de contrôle c .

Conditionnement de l'événement de parité

Le message $R_{c \rightarrow v}(b)$ correspond à la probabilité conditionnelle :

$$R_{c \rightarrow v}(b) = P \left(\bigoplus_{u \in \mathcal{N}(c)} V_u = 0 \mid V_v = b \right)$$

Par linéarité du XOR, cette condition est équivalente à :

$$P \left(\bigoplus_{u \in \mathcal{N}(c) \setminus \{v\}} V_u = b \right)$$

Par le théorème des probabilités totales appliqué au système complet d'événements associé aux configurations $x \in \mathbb{F}_2^{d_c-1}$ des voisins :

$$R_{c \rightarrow v}(b) = \sum_{\substack{x \in \mathbb{F}_2^{d_c-1} \\ \bigoplus_{u \in \mathcal{N}(c) \setminus \{v\}} x_u = b}} \prod_{u \in \mathcal{N}(c) \setminus \{v\}} P(V_u = x_u)$$

CN Update (1) : Probabilités

En utilisant les messages entrants du graphe, on définit la probabilité locale $P_{u \rightarrow c}(x_u) = P(V_u = x_u)$.

Le message sortant devient :

$$R_{c \rightarrow v}(b) = \sum_{\substack{x \in \mathbb{F}_2^{d_c-1} \\ \bigoplus_{u \in \mathcal{N}(c) \setminus \{v\}} x_u = b}} \prod_{u \in \mathcal{N}(c) \setminus \{v\}} P_{u \rightarrow c}(x_u)$$

Exacte mais complexité est exponentielle en d_c . On utilise alors une transformation pour simplifier le calcul (Lemme de Gallager).

CN Update (2) : Lemme de Gallager

Lemme de Gallager – Soient (X_1, \dots, X_n) des variables de Bernoulli indépendantes sur \mathbb{F}_2 .

$$P\left(\bigoplus_{i=1}^n X_i = 0\right) = \frac{1}{2} \left(1 + \prod_{i=1}^n E[(-1)^{X_i}]\right)$$

On notes :

$$E[(-1)^{V_u}] = \delta_{u \rightarrow c} = P(V_u = 0) - P(V_u = 1) = 1 - 2P_{u \rightarrow c}(1)$$

On en déduit les probabilités marginales conditionnelles pour le nœud c :

$$R_{c \rightarrow v}(0) = \frac{1}{2} \left(1 + \prod_{u \in \mathcal{N}(c) \setminus \{v\}} \delta_{u \rightarrow c}\right), \quad R_{c \rightarrow v}(1) = \frac{1}{2} \left(1 - \prod_{u \in \mathcal{N}(c) \setminus \{v\}} \delta_{u \rightarrow c}\right)$$

CN Update (3) : Passage aux LLR

Log-Likelihood Ratio (LLR)

Le message LLR entrant $m_{u \rightarrow c}$ au nœud de contrôle est défini par :

$$m_{u \rightarrow c} = \ln \left(\frac{P(V_u = 0)}{P(V_u = 1)} \right)$$

$\delta_{u \rightarrow c}$ s'exprime alors :

$$\delta_{u \rightarrow c} = \tanh \left(\frac{m_{u \rightarrow c}}{2} \right)$$

On en déduit le LLR du message sortant :

$$m_{c \rightarrow v} = \ln \left(\frac{R_{c \rightarrow v}(0)}{R_{c \rightarrow v}(1)} \right) = 2 \tanh^{-1} \left(\prod_{u \in \mathcal{N}(c) \setminus \{v\}} \tanh \left(\frac{m_{u \rightarrow c}}{2} \right) \right)$$

CN Update (4) : Algorithmes

Sum-Product – Mise à jour :

$$m_{c \rightarrow v} = 2 \tanh^{-1} \left(\prod_{u \in \mathcal{N}(c) \setminus \{v\}} \tanh \left(\frac{m_{u \rightarrow c}}{2} \right) \right)$$

Min-Sum – Approximation :

$$m_{c \rightarrow v} \approx \left(\prod_{u \in \mathcal{N}(c) \setminus \{v\}} \operatorname{sgn}(m_{u \rightarrow c}) \right) \times \min_{u \in \mathcal{N}(c) \setminus \{v\}} |m_{u \rightarrow c}|$$

Bit-Flipping : Choix du Seuil

- Chaque v_j reçoit de ses w_c voisins c_i un verdict $f_i \in \{0, 1\}$.
- Il retourne son bit si **trop d'équations échouent**.

Stratégies de Seuil

| Seuil | Règle | Remarque |
|-------------------|-------------------------------|--|
| Majorité stricte | $k_j > w_c/2$ | Standard Gallager-A |
| Seuil fixe b | $k_j \geq b$ (ex. $b = w_c$) | Gallager-B : plus conservateur |
| Tous insatisfaits | $k_j = w_c$ | Très conservateur, peu de faux retournements |

- **Gallager-A** : retourner si **toutes** les contraintes sont violées ($k_j = w_c$) — évite les oscillations
- **Gallager-B** : retourner si **au moins** $b < w_c$ contraintes violées — plus agressif

Que Se Passe-t-il en Cas d'Échec ?

Échec du Décodeur

Après I_{\max} itérations : syndrome $s \neq 0 \rightarrow$ **décodage échoué**.

Deux scénarios :

- La trame est **perdue** (FPV, diffusion) \rightarrow acceptable
- La trame doit **arriver** (fichier, protocole) \rightarrow retransmission nécessaire

ARQ — Automatic Repeat reQuest

Le récepteur envoie un **NACK** (Not Acknowledged) : l'émetteur retransmet.

- **Stop-and-Wait** : simple mais inefficace
- **Hybrid ARQ (HARQ)** : combine ARQ + codage — standard LTE/5G

HARQ Type II — Chase Combining

Le récepteur **combine** les LLR des deux transmissions avant de redécoder :

$$L_{\text{total}(y_i)} = L_{\text{canal}}^1(y_i) + L_{\text{canal}}^2(y_i)$$

Limites du Modèle AWGN

Ce que AWGN suppose

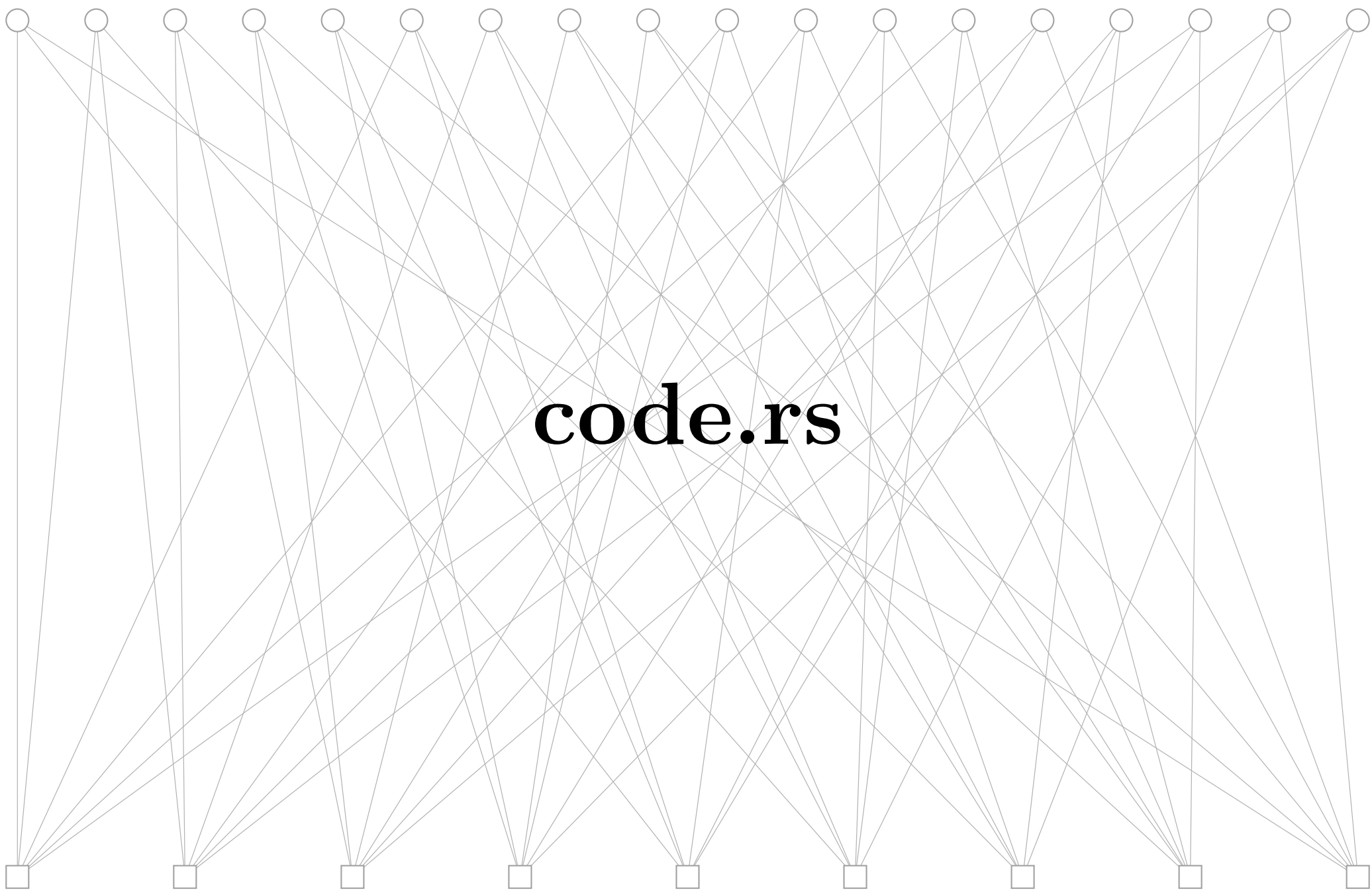
- Bruit **blanc** : indépendant d'un symbole à l'autre
- Distribution **gaussienne** stationnaire
- Canal **sans mémoire** : chaque bit perturbé indépendamment

Canaux Réels — Ce qu'AWGN ne capture pas

| Phénomène | Impact |
|--------------------------|--|
| Évanouissements (fading) | SNR varie dans le temps |
| Burst d'erreurs | Bits consécutifs corrompus → entrelacement nécessaire |
| Bruit impulsionnel | Pics de bruit ponctuels (moteurs, orages) |
| Erreurs de phase | Synchronisation imparfaite en BPSK |

Solution : Entrelacement

Permuter les bits **avant** l'envoi : les erreurs en rafale deviennent des erreurs isolées pour le décodeur LDPC.



```
use crate::graph::TannerGraph;
use crate::matrix::{DenseMatrixGF2, SparseMatrixGF2};
use crate::{Gf2, LdpcError, Result};
use serde::{Deserialize, Serialize};

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct LdpcParams {
    pub n: usize,
    pub k: usize,
    pub topology: CodeTopology,
    pub generation: GenerationMethod,
    pub seed: Option<u64>,
}

impl LdpcParams {
    pub fn rate(&self) -> f64 {
        self.k as f64 / self.n as f64
    }
    pub fn m(&self) -> usize {
        self.n - self.k
    }

    pub fn validate(&self) -> Result<()> {
        if self.k >= self.n {
            return Err(LdpcError::InvalidParameters("k doit être < n".into()));
        }
        if self.n < 4 {
            return Err(LdpcError::InvalidParameters("n trop petit".into()));
        }
        self.topology.validate(self)
    }
}
```

```

    }
}

// Topologie
#[derive(Debug, Clone, Serialize, Deserialize)]
pub enum CodeTopology {
    // Chaque varnode a degré wc, chaque checknode a degré wr
    // Condition nécessaire  $n * wc == m * wr$ 
    Regular { wc: usize, wr: usize },
    // Potentielle implémentation Irregular (plus tard !)
}

impl CodeTopology {
    fn validate(&self, params: &LdpcParams) -> Result<()> {
        match self {
            CodeTopology::Regular { wc, wr } => {
                if params.n * wc != params.m() * wr {
                    return Err(LdpcError::InvalidParameters(format!(
                        "n*wc ({{}}) != m*wr ({{}}) pour LDPC régulier",
                        params.n * wc,
                        params.m() * wr
                    )));
                }
            }
        }
        Ok(())
    }
}

// Méthode de génération
#[derive(Debug, Clone, Serialize, Deserialize)]
pub enum GenerationMethod {
    //  $H = [H1 \mid H2 \mid \dots \mid Hwc]^T$ , H1 régulière, Hi = permutation de H1
    // Gallager,
    // Ajout de colonnes de poids fixe, rejet si cycle4 créé
}

```

```

    MacKayNeal { max_attempts: usize },
}

// Forme systématique
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct SystematicForm {
    //  $G = [I_k \mid P]$ , dense
    pub g: DenseMatrixGF2,
    // Permutation de colonnes appliquée à  $H \Rightarrow [A \mid I_m]$ 
    pub col_perm: Vec<usize>,
    // Permutation inverse  $\Rightarrow$  reformer le mot de code
    pub col_perm_inv: Vec<usize>,
}

// Structure principale
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct LdpcCode {
    pub params: LdpcParams,
    pub h: SparseMatrixGF2,
    #[serde(skip, default = "default_graph")]
    pub graph: TannerGraph,
    pub systematic_form: Option<SystematicForm>,
}

impl LdpcCode {
    pub fn new(params: LdpcParams) -> Result<Self> {
        params.validate()?;
        let mut rng = build_rng(params.seed);
        let h = match &params.generation {
            GenerationMethod::Gallager => generate_gallager(&params, &mut rng)?,
            GenerationMethod::MacKayNeal { max_attempts } => {
                generate_mackay_neal(&params, *max_attempts, &mut rng)?
            }
        };
        let graph = TannerGraph::from_matrix(&h);
    }
}

```



```

    Ok(Self {
        params,
        h,
        graph,
        systematic_form: None,
    })
}

pub fn from_matrix(h: SparseMatrixGF2, k: usize) -> Result<Self> {
    let n = h.cols;
    let params = LdpcParams {
        n,
        k,
        topology: CodeTopology::Regular { wc: 0, wr: 0 }, // inconnu
        generation: GenerationMethod::Gallager,
        seed: None,
    };
    let graph = TannerGraph::from_matrix(&h);
    Ok(Self {
        params,
        h,
        graph,
        systematic_form: None,
    })
}

// Calcule G par Gauss-Jordan sur H (to cache)
pub fn compute_systematic_form(&mut self) -> Result<()> {
    if self.systematic_form.is_some() {
        return Ok(());
    }

    let mut dense = DenseMatrixGF2::zeros(self.m(), self.n());
    let h_dense = self.h.to_dense();
    for r in 0..self.m() {

```

```

        for c in 0..self.n() {
            dense.set(r, c, h_dense[r][c]);
        }
    }

    let (col_perm, rank) = dense.systematize(self.k());

    // Possibilité d'avoir rg < m donc juste rejeter et recommencer
    if rank < self.m() {
        return Err(LdpcError::SingularMatrix);
    }

    // Création de G^T (taille n,k) pour encodage c = G^T * m
    let mut g_t = DenseMatrixGF2::zeros(self.n(), self.k());

    // I_k (haut)
    for i in 0..self.k() {
        g_t.set(i, i, 1);
    }

    // (bas) Matrice A (extraite des k premières colonnes de la matrice dense pivotée)
    for r in 0..self.m() {
        for c in 0..self.k() {
            g_t.set(self.k() + r, c, dense.get(r, c));
        }
    }

    // Création de la permutation inverse pour réordonner le mot de code
    let mut col_perm_inv = vec![0usize; self.n()];
    for (i, &p) in col_perm.iter().enumerate() {
        col_perm_inv[p] = i;
    }

    self.systematic_form = Some(SystematicForm {
        g: g_t,

```

```

        col_perm,
        col_perm_inv,
    });
    Ok(())
}

pub fn rate(&self) -> f64 {
    self.params.rate()
}
pub fn n(&self) -> usize {
    self.params.n
}
pub fn k(&self) -> usize {
    self.params.k
}
pub fn m(&self) -> usize {
    self.params.m()
}
pub fn girth(&self) -> usize {
    self.graph.girth()
}

pub fn is_codeword(&self, c: &[Gf2]) -> bool {
    self.h.multiply_vec(c).iter().all(|&s| s == 0)
}
}

fn build_rng(seed: Option<u64>) -> impl rand::Rng {
    use rand::SeedableRng;
    rand::rngs::StdRng::seed_from_u64(seed.unwrap_or_else(rand::random))
}

// Gallager
// H divisée en wc sous-matrices de taille (m / wc) * n
// H1 = matrice régulière (ligne contient wr 1)

```

```
// H2..Hwc = permutations aléatoires de colonnes de H1
```

```
fn generate_gallager(params: &LdpcParams, rng: &mut impl rand::Rng) -> Result<SparseMatrixGF2> {  
    let CodeTopology::Regular { wc, wr } = params.topology else {  
        return Err(LdpcError::InvalidParameters(  
            "Gallager nécessite un code régulier".into(),  
        ));  
    };  
    let n = params.n;  
    let m = params.m();  
    if m % wc != 0 {  
        return Err(LdpcError::InvalidParameters(  
            "m doit être divisible par wc".into(),  
        ));  
    }  
    let rows_per_block = m / wc;  
    let mut ones: Vec<(usize, usize)> = Vec::new();  
  
    for r in 0..rows_per_block {  
        for j in 0..wr {  
            ones.push((r, r * wr + j));  
        }  
    }  
    use rand::seq::SliceRandom;  
    for block in 1..wc {  
        let mut perm: Vec<usize> = (0..n).collect();  
        perm.shuffle(rng);  
        for r in 0..rows_per_block {  
            for j in 0..wr {  
                ones.push((block * rows_per_block + r, perm[r * wr + j]));  
            }  
        }  
    }  
  
    for b in 1..wc {
```

```

    let r_source = b * rows_per_block;

    let row_0_cols: Vec<usize> = ones
        .iter()
        .filter(|&&(r, _)| r == 0)
        .map(|&(_, c)| c)
        .collect();

    if let Some(idx) = ones
        .iter()
        .position(|&(r, c)| r == r_source && !row_0_cols.contains(&c))
    {
        ones[idx].0 = 0;
    }
}

Ok(SparseMatrixGF2::from_positions(m, n, ones))
}

// MacKay-Neal
// Ajoute les colonnes une à une avec poids wc
// Rejette colonne créant cycle4 (2 colonnes n'ont qu'un 1 en commun)
fn generate_mackay_neal(
    params: &LdpcParams,
    max_attempts: usize,
    rng: &mut impl rand::Rng,
) -> Result<SparseMatrixGF2> {
    let CodeTopology::Regular { wc, wr } = params.topology else {
        return Err(LdpcError::InvalidParameters(
            "MacKayNeal nécessite régulier".into(),
        ));
    };
    let n = params.n;
    let m = params.m();
    let mut ones: Vec<(usize, usize)> = Vec::new();

```

```

// Suivi du poids de chaque ligne
let mut row_weights = vec![0usize; m];

use rand::seq::SliceRandom;
for col in 0..n {
    let mut placed = false;
    for _attempt in 0..max_attempts {
        let mut available_rows: Vec<usize> = (0..m).filter(|&r| row_weights[r] <
wr).collect();

        if available_rows.len() < wc {
            break; // Plus de lignes dispo
        }

        available_rows.shuffle(rng);
        let candidate = available_rows[..wc].to_vec();

        // Vérifier cycle4
        let mut ok = true;
        let mut c2 = 0;
        while c2 < col && ok {
            let existing: Vec<usize> = ones
                .iter()
                .filter(|&&(_, c)| c == c2)
                .map(|&(r, _)| r)
                .collect();
            let shared = candidate.iter().filter(|r| existing.contains(r)).count();
            if shared >= 2 {
                ok = false;
            }
            c2 += 1;
        }

        if ok {

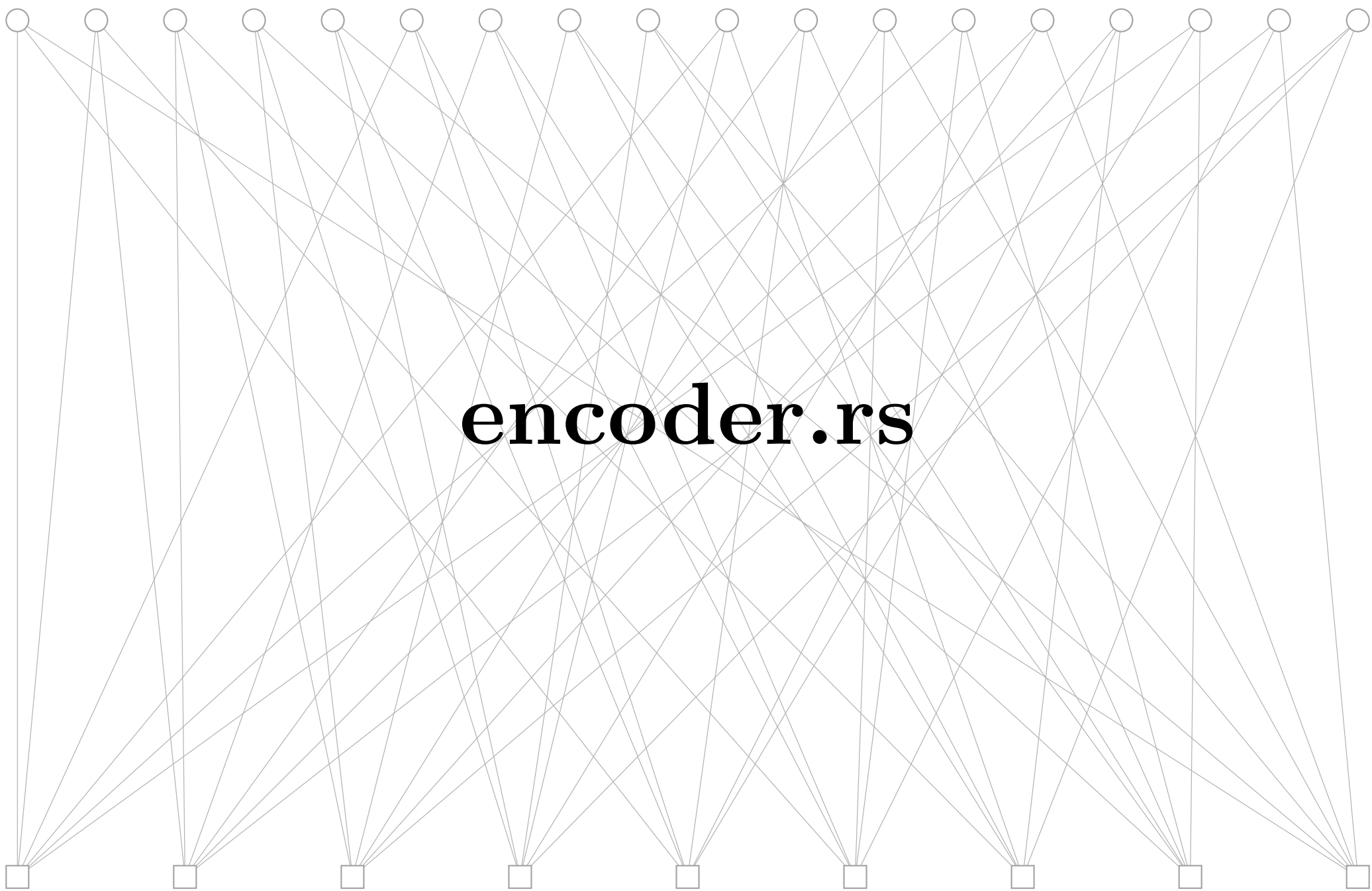
```

```

        for &r in &candidate {
            ones.push((r, col));
            row_weights[r] += 1;
        }
        placed = true;
        break;
    }
}
if !placed {
    return Err(LdpcError::GenerationFailed {
        attempts: max_attempts,
    });
}
}
Ok(SparseMatrixGF2::from_positions(m, n, ones))
}

fn default_graph() -> TannerGraph {
    TannerGraph::from_matrix(&SparseMatrixGF2::zeros(1, 1))
}

```



encoder.rs

```
use crate::code::LdpcCode;
use crate::{BitVec, Gf2, LdpcError, Result};

pub trait Encoder: Send + Sync {
    fn encode(&self, message: &[Gf2]) -> Result<BitVec>;
    fn message_len(&self) -> usize;
    fn codeword_len(&self) -> usize;
    fn extract_message(&self, codeword: &[Gf2]) -> Vec<Gf2>;

    fn check_input(&self, msg: &[Gf2]) -> Result<()> {
        if msg.len() != self.message_len() {
            return Err(LdpcError::DimensionMismatch {
                expected: self.message_len(),
                got: msg.len(),
            });
        }
        Ok(())
    }
}

#[derive(Debug, Clone)]
pub enum EncodingMethod {
    Systematic,
}

pub struct SystematicEncoder {
    k: usize,
    n: usize,
    // g_t: DenseMatrixGF2,
    packed_g_cols: Vec<Vec<u64>>,
}
```

```

    perm_inv: Vec<usize>,
    col_perm: Vec<usize>,
}

impl SystematicEncoder {
    pub fn new(code: &mut LdpcCode) -> Result<Self> {
        code.compute_systematic_form()?;
        let sf = code.systematic_form.as_ref().unwrap();

        let k = code.k();
        let n = code.n();
        let g_t = &sf.g;

        let num_blocks = (n + 63) / 64;

        // Bitpacking
        let mut packed_g_cols = vec![vec![0u64; num_blocks]; k];

        for j in 0..k {
            for i in 0..n {
                if g_t.get(i, j) == 1 {
                    let block_idx = i / 64;
                    let bit_idx = i % 64;
                    packed_g_cols[j][block_idx] |= 1 << bit_idx;
                }
            }
        }

        Ok(Self {
            k,
            n,
            packed_g_cols,
            perm_inv: sf.col_perm_inv.clone(),
            col_perm: sf.col_perm.clone(),
        })
    }
}

```

```
}  
}
```

```
impl Encoder for SystematicEncoder {  
    // fn encode(&self, message: &[Gf2]) -> Result<BitVec> {  
    //     self.check_input(message)?;  
    //  
    //     let c_perm = self.g_t.multiply_vec(message);  
    //  
    //     // Retablir l'ordre initial des bits selon la permutation de H  
    //     let mut c = vec![0u8; self.n];  
    //     // for (i, &ci) in c_perm.iter().enumerate() {  
    //         //     c[self.perm_inv[i]] = ci;  
    //     // }  
    //  
    //     for i in 0..self.n {  
    //         c[i] = c_perm[self.perm_inv[i]];  
    //     }  
    //  
    //     Ok(c)  
    // }  
    fn encode(&self, message: &[Gf2]) -> Result<BitVec> {  
        self.check_input(message)?;  
  
        let num_blocks = (self.n + 63) / 64;  
        let mut accum = vec![0u64; num_blocks];  
  
        for (j, &bit) in message.iter().enumerate() {  
            if bit == 1 {  
                for b in 0..num_blocks {  
                    accum[b] ^= self.packed_g_cols[j][b];  
                }  
            }  
        }  
    }  
}
```

```

        let mut c = vec![0u8; self.n];
        for i in 0..self.n {
            let block_idx = i / 64;
            let bit_idx = i % 64;
            let val = ((accum[block_idx] >> bit_idx) & 1) as u8;

            c[self.col_perm[i]] = val;
        }

        Ok(c)
    }

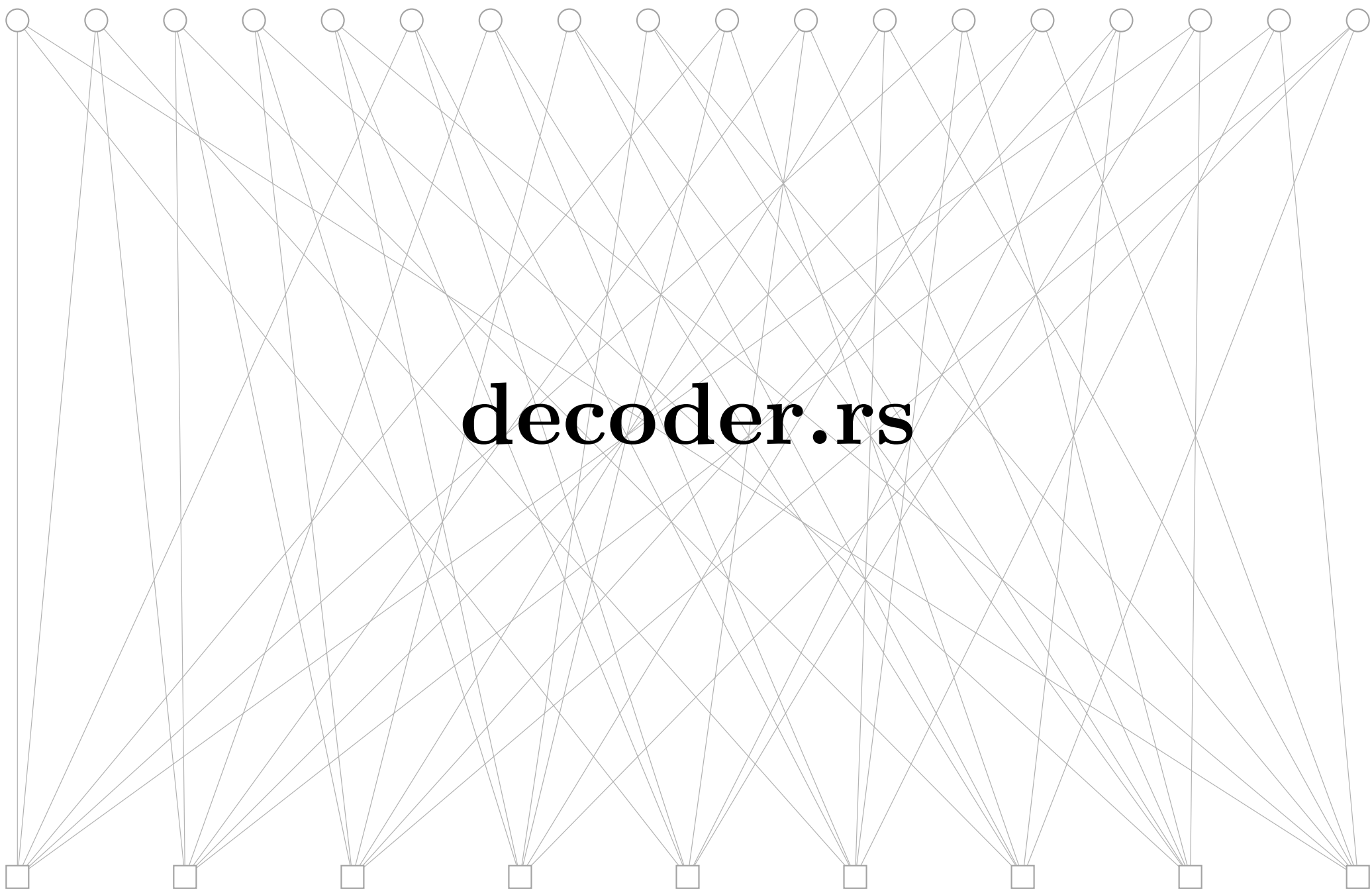
    fn extract_message(&self, codeword: &[Gf2]) -> Vec<Gf2> {
        let mut msg = vec![0u8; self.k];
        for j in 0..self.k {
            msg[j] = codeword[self.col_perm[j]];
        }
        msg
    }

    fn message_len(&self) -> usize {
        self.k
    }

    fn codeword_len(&self) -> usize {
        self.n
    }
}

pub fn build_encoder(code: &mut LdpcCode, method: EncodingMethod) -> Result<Box<dyn Encoder>> {
    match method {
        EncodingMethod::Systematic => Ok(Box::new(SystematicEncoder::new(code)?)),
    }
}

```



decoder.rs

```
use crate::code::LdpcCode;
use crate::graph::TannerGraph;
use crate::matrix::SparseMatrixGF2;
use crate::{BitVec, Gf2, Llr};

// Résultat
#[derive(Debug, Clone)]
pub enum DecoderResult {
    Converged(BitVec),
    MaxIterationsReached(BitVec),
    Failure,
}

impl DecoderResult {
    pub fn codeword(&self) -> Option<&BitVec> {
        match self {
            DecoderResult::Converged(c) | DecoderResult::MaxIterationsReached(c) => Some(c),
            DecoderResult::Failure => None,
        }
    }

    pub fn is_success(&self) -> bool {
        matches!(self, DecoderResult::Converged(_))
    }
}

// Configuration
#[derive(Debug, Clone)]
pub struct DecoderConfig {
    pub max_iterations: usize,
    pub early_stopping: bool,
```

```

}

impl Default for DecoderConfig {
    fn default() -> Self {
        Self {
            max_iterations: 50,
            early_stopping: true,
        }
    }
}

// Trait Decoder
pub trait Decoder: Send + Sync {
    fn decode(&self, channel_llr: &[Llr]) -> DecoderResult;

    fn decode_hard(&self, received: &[Gf2]) -> DecoderResult {
        let llr: Vec<Llr> = received
            .iter()
            .map(|&b| if b == 0 { 1.0 } else { -1.0 })
            .collect();
        self.decode(&llr)
    }
}

// Primitives GF(2) et LLR
#[inline]
pub fn hard_decision(llr: Llr) -> Gf2 {
    if llr >= 0.0 {
        0
    } else {
        1
    }
}

pub fn compute_syndrome(h: &SparseMatrixGF2, c: &[Gf2]) -> Vec<Gf2> {

```

```

    h.multiply_vec(c)
}

#[inline]
fn phi(x: Llr) -> Llr {
    let ax = x.abs().max(1e-10);
    -((ax / 2.0).tanh()).ln()
}

// Mises à jour des noeuds

// Mise à jour Sum-Product du noeud de contrôle
fn check_node_update_sp(incoming: &[Llr], out: &mut [Llr]) {
    let phi_sum: Llr = incoming.iter().map(|&l| phi(l.abs())).sum();
    let sign_prod: Llr = incoming.iter().map(|&l| l.signum()).product();
    for (_j, (&l, r)) in incoming.iter().zip(out.iter_mut()).enumerate() {
        let phi_excl = phi_sum - phi(l.abs());
        let sign_excl = sign_prod * l.signum();
        *r = sign_excl * phi(phi_excl);
    }
}

// Mise à jour Min-Sum avec facteur de normalisation a
// alpha in [0.75, 0.875] compense le biais de Min-Sum brut
fn check_node_update_ms(incoming: &[Llr], out: &mut [Llr], alpha: Llr) {
    let sign_prod: Llr = incoming.iter().map(|&l| l.signum()).product();
    let mut min1 = Llr::INFINITY;
    let mut min2 = Llr::INFINITY;
    let mut min1_idx = 0;
    for (j, &l) in incoming.iter().enumerate() {
        let al = l.abs();
        if al < min1 {
            min2 = min1;
            min1 = al;
            min1_idx = j;
        }
    }
}

```



```

        } else if al < min2 {
            min2 = al;
        }
    }
    for (j, (&l, r)) in incoming.iter().zip(out.iter_mut()).enumerate() {
        let min_excl = if j == min1_idx { min2 } else { min1 };
        let sign_excl = sign_prod * l.signum();
        *r = alpha * sign_excl * min_excl;
    }
}

// Mise à jour du noeud de variable
fn variable_node_update(ch_llr: Llr, incoming_c2v: &[Llr], out: &mut [Llr]) {
    let total: Llr = ch_llr + incoming_c2v.iter().sum::<Llr>();
    for (&r, o) in incoming_c2v.iter().zip(out.iter_mut()) {
        *o = total - r;
    }
}

#[inline]
fn posterior_llr(ch_llr: Llr, c2v_msgs: &[Llr]) -> Llr {
    ch_llr + c2v_msgs.iter().sum::<Llr>()
}

// Messages internes
// Indexés par (check_id, position_dans_liste_voisins) (0(1))
struct Messages {
    v2c: Vec<Vec<Llr>>,
    c2v: Vec<Vec<Llr>>,
}

impl Messages {
    fn new(graph: &TannerGraph) -> Self {
        let v2c = (0..graph.n_chk)
            .map(|c| vec![0.0; graph.chk_degree(c)])

```

```

        .collect();
    let c2v = (0..graph.n_chk)
        .map(|c| vec![0.0; graph.chk_degree(c)])
        .collect();
    Self { v2c, c2v }
}

```

// Table de correspondance, pour chaque (var, check), index dans la liste de voisins
 // Précalculé une fois à construction du décodeur

```

struct EdgeIndex {
    var_pos_in_chk: Vec<Vec<usize>>,
    chk_pos_in_var: Vec<Vec<usize>>,
}

```

```

impl EdgeIndex {
    fn build(graph: &TannerGraph) -> Self {
        let var_pos_in_chk = (0..graph.n_chk)
            .map(|c| {
                graph
                    .chk_neighbors(c)
                    .iter()
                    .map(|&v| graph.var_neighbors(v).iter().position(|&x| x == c).unwrap())
                    .collect()
            })
            .collect();
        let chk_pos_in_var = (0..graph.n_var)
            .map(|v| {
                graph
                    .var_neighbors(v)
                    .iter()
                    .map(|&c| graph.chk_neighbors(c).iter().position(|&x| x == v).unwrap())
                    .collect()
            })
            .collect();
    }
}

```

```

        Self {
            var_pos_in_chk,
            chk_pos_in_var,
        }
    }
}

// Bit-Flipping
pub struct BitFlippingDecoder {
    graph: TannerGraph,
    h: SparseMatrixGF2,
    config: DecoderConfig,
}

impl BitFlippingDecoder {
    pub fn new(code: &LdpcCode, config: DecoderConfig) -> Self {
        Self {
            graph: code.graph.clone(),
            h: code.h.clone(),
            config,
        }
    }
}

impl Decoder for BitFlippingDecoder {
    fn decode(&self, channel_llr: &[Llr]) -> DecoderResult {
        let mut bits: Vec<Gf2> = channel_llr.iter().map(|&l| hard_decision(l)).collect();

        for _iter in 0..self.config.max_iterations {
            let syndrome = compute_syndrome(&self.h, &bits);
            if self.config.early_stopping && syndrome.iter().all(|&s| s == 0) {
                return DecoderResult::Converged(bits);
            }
            let mut unsatisfied = vec![0usize; self.graph.n_var];
            for c in 0..self.graph.n_chk {

```

```

        if syndrome[c] == 1 {
            for &v in self.graph.chk_neighbors(c) {
                unsatisfied[v] += 1;
            }
        }
    }
    let mut flipped = false;
    for v in 0..self.graph.n_var {
        if unsatisfied[v] > self.graph.var_degree(v) / 2 {
            bits[v] ^= 1;
            flipped = true;
        }
    }
    if !flipped {
        break;
    }
}

let synd = compute_syndrome(&self.h, &bits);
if synd.iter().all(|&s| s == 0) {
    DecoderResult::Converged(bits)
} else {
    DecoderResult::MaxIterationsReached(bits)
}
}
}

```

// BP

```

fn bp_decode<F>(
    graph: &TannerGraph,
    h: &SparseMatrixGF2,
    config: &DecoderConfig,
    channel_llr: &[Llr],
    edge_idx: &EdgeIndex,
    check_update: F,

```

```

) -> DecoderResult
where
  F: Fn(&[Llr], &mut [Llr]),
{
  let mut msgs = Messages::new(graph);

  // Init
  for c in 0..graph.n_chk {
    for (j, &v) in graph.chk_neighbors(c).iter().enumerate() {
      msgs.v2c[c][j] = channel_llr[v];
    }
  }

  for _iter in 0..config.max_iterations {
    // Maj des checknodes
    for c in 0..graph.n_chk {
      let v2c = msgs.v2c[c].clone();
      check_update(&v2c, &mut msgs.c2v[c]);
    }

    // Maj des varnodes
    for v in 0..graph.n_var {
      let neighbors = graph.var_neighbors(v);
      // Rassembler les c2v entrants sur ce varnode
      let incoming: Vec<Llr> = neighbors
        .iter()
        .enumerate()
        .map(|(i, &c)| {
          let j = edge_idx.chk_pos_in_var[v][i];
          msgs.c2v[c][j]
        })
        .collect();
      let mut new_v2c = vec![0.0; neighbors.len()];
      variable_node_update(channel_llr[v], &incoming, &mut new_v2c);
      for (i, &c) in neighbors.iter().enumerate() {

```

```

        let j = edge_idx.chk_pos_in_var[v][i];
        msgs.v2c[c][j] = new_v2c[i];
    }
}

// Hard décision + arrêt
if config.early_stopping {
    let bits = make_decision(graph, &msgs, channel_llr, edge_idx);
    if compute_syndrome(h, &bits).iter().all(|&s| s == 0) {
        return DecoderResult::Converged(bits);
    }
}

let bits = make_decision(graph, &msgs, channel_llr, edge_idx);
let synd = compute_syndrome(h, &bits);
if synd.iter().all(|&s| s == 0) {
    DecoderResult::Converged(bits)
} else {
    DecoderResult::MaxIterationsReached(bits)
}
}

fn make_decision(
    graph: &TannerGraph,
    msgs: &Messages,
    channel_llr: &[Llr],
    edge_idx: &EdgeIndex,
) -> Vec<Gf2> {
    (0..graph.n_var)
        .map(|v| {
            let incoming: Vec<Llr> = graph
                .var_neighbors(v)
                .iter()
                .enumerate()

```

```

        .map(|(i, &c)| {
            let j = edge_idx.chk_pos_in_var[v][i];
            msgs.c2v[c][j]
        })
        .collect();
    hard_decision(posterior_llr(channel_llr[v], &incoming))
})
.collect()
}

```

// Sum-Product

```

pub struct SumProductDecoder {
    graph: TannerGraph,
    h: SparseMatrixGF2,
    config: DecoderConfig,
    edge_idx: EdgeIndex,
}

```

```

impl SumProductDecoder {
    pub fn new(code: &LdpcCode, config: DecoderConfig) -> Self {
        let edge_idx = EdgeIndex::build(&code.graph);
        Self {
            graph: code.graph.clone(),
            h: code.h.clone(),
            config,
            edge_idx,
        }
    }
}

```

```

impl Decoder for SumProductDecoder {
    fn decode(&self, channel_llr: &[Llr]) -> DecoderResult {
        bp_decode(
            &self.graph,
            &self.h,

```

```

        &self.config,
        channel_llr,
        &self.edge_idx,
        |incoming, out| check_node_update_sp(incoming, out),
    )
}
}

// Min-Sum
pub struct MinSumDecoder {
    graph: TannerGraph,
    h: SparseMatrixGF2,
    config: DecoderConfig,
    scaling_factor: Llr,
    edge_idx: EdgeIndex,
}

impl MinSumDecoder {
    pub fn new(code: &LdpcCode, config: DecoderConfig, scaling_factor: Llr) -> Self {
        let edge_idx = EdgeIndex::build(&code.graph);
        Self {
            graph: code.graph.clone(),
            h: code.h.clone(),
            config,
            scaling_factor,
            edge_idx,
        }
    }
}

impl Decoder for MinSumDecoder {
    fn decode(&self, channel_llr: &[Llr]) -> DecoderResult {
        let alpha = self.scaling_factor;
        bp_decode(
            &self.graph,

```



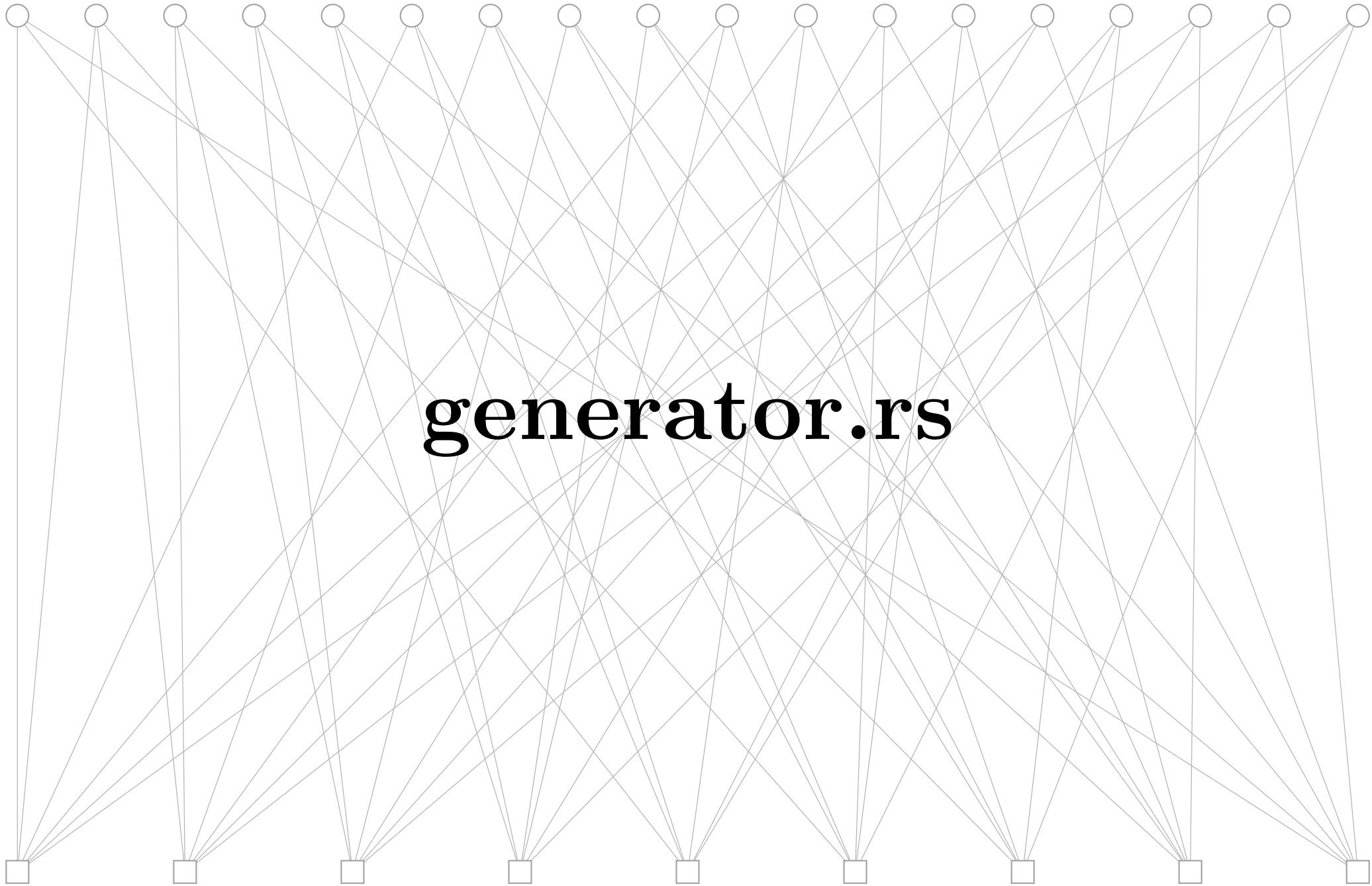
```

        &self.h,
        &self.config,
        channel_llr,
        &self.edge_idx,
        move |incoming, out| check_node_update_ms(incoming, out, alpha),
    )
}
}

// Factory
#[derive(Debug, Clone)]
pub enum DecoderMethod {
    BitFlipping,
    SumProduct,
    MinSum { scaling_factor: Llr },
}

pub fn build_decoder(
    code: &LdpcCode,
    method: DecoderMethod,
    config: DecoderConfig,
) -> Box<dyn Decoder> {
    match method {
        DecoderMethod::BitFlipping => Box::new(BitFlippingDecoder::new(code, config)),
        DecoderMethod::SumProduct => Box::new(SumProductDecoder::new(code, config)),
        DecoderMethod::MinSum { scaling_factor } => {
            Box::new(MinSumDecoder::new(code, config, scaling_factor))
        }
    }
}
}

```



generator.rs

```
use clap::Parser;
use indicatif::{ProgressBar, ProgressStyle};
use ldpc::code::{CodeTopology, GenerationMethod, LdpcCode, LdpcParams};
use rayon::prelude::*;
use std::fs::File;
use std::io::Write;
use std::sync::{
    atomic::{AtomicBool, AtomicU64, Ordering},
    Arc,
};
use std::time::{Duration, Instant};

#[derive(Parser, Debug)]
#[command(author, version, about = "Générateur LDPC Haute Performance")]
struct Args {
    #[arg(short, long)]
    n: usize,
    #[arg(short, long)]
    k: usize,
    #[arg(long, default_value_t = 3)]
    wc: usize,
    #[arg(long, default_value_t = 6)]
    wr: usize,
    #[arg(short, long, default_value = "mackay")]
    method: String,
}

fn main() -> ldpc::Result<()> {
    let args = Args::parse();
    let m = args.n - args.k;
```

```

if (args.n * args.wc) % m != 0 || (args.n * args.wc) / m != args.wr {
    println!("Erreur : Parametres impossibles (n*wc != m*wr)");
    std::process::exit(1);
}

let found = Arc::new(AtomicBool::new(false));
let attempts = Arc::new(AtomicU64::new(0));
let start_global = Instant::now();

let pb = ProgressBar::new_spinner();
pb.enable_steady_tick(Duration::from_millis(80));
pb.set_style(
    ProgressStyle::default_spinner()
        .template(
            "{spinner:.cyan} [{elapsed_precise}] {msg} | Tentatives: {pos} | {per_sec} mats/
s",
        )
        .unwrap(),
);
pb.set_message("Recherche d'une matrice valide...");

let method = if args.method == "mackay" {
    GenerationMethod::MacKayNeal { max_attempts: 1000 }
} else {
    GenerationMethod::Gallager
};

let pb_clone = pb.clone();
let found_clone = Arc::clone(&found);
let attempts_clone = Arc::clone(&attempts);

let result = (0..num_cpus::get())
    .into_par_iter()
    .map(|_| {

```

```

while !found_clone.load(Ordering::Relaxed) {
    let current_attempt = attempts_clone.fetch_add(1, Ordering::SeqCst);
    pb_clone.set_position(current_attempt);

    let params = LdpcParams {
        n: args.n,
        k: args.k,
        topology: CodeTopology::Regular {
            wc: args.wc,
            wr: args.wr,
        },
        generation: method.clone(),
        seed: Some(rand::random()),
    };

    if let Ok(mut code) = LdpcCode::new(params) {
        // Cette etape est la plus longue (Gauss-Jordan)
        if code.compute_systematic_form().is_ok() {
            if !found_clone.swap(true, Ordering::SeqCst) {
                return Some(code);
            }
        }
    }
}

None
}))
.find_any(|res| res.is_some())
.flatten();

if let Some(code) = result {
    pb.finish_with_message(format!("Succes en {:.2?}", start_global.elapsed()));

    let filename = format!(
        "cache_ldpc_{}_n{}_k{}.bin",
        args.method.to_uppercase(),

```

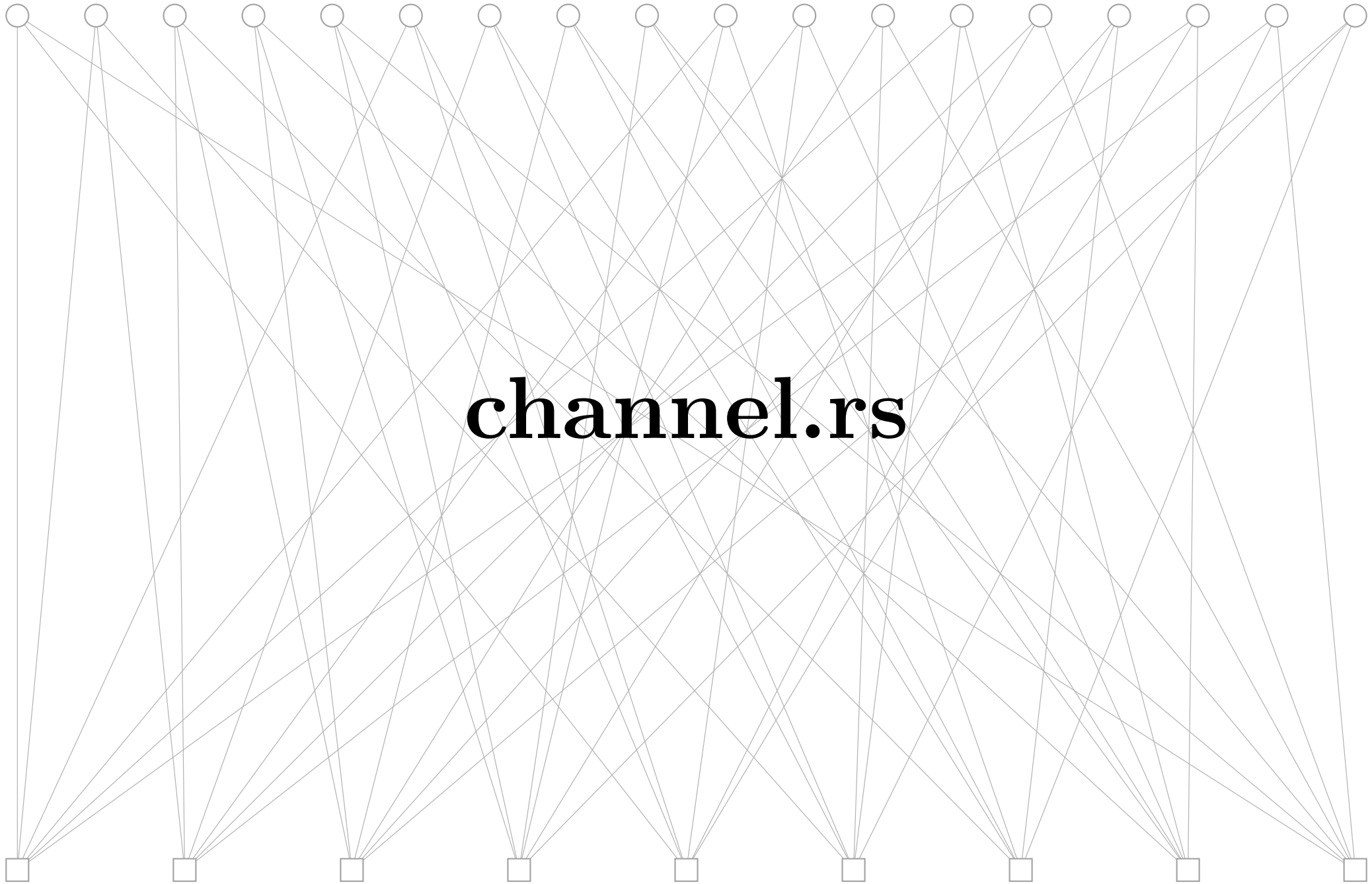
```

        args.n,
        args.k
    );
    let encoded = bincode::serialize(&code).expect("Erreur serialisation");
    let mut file = File::create(&filename).expect("Erreur creation");
    file.write_all(&encoded).expect("Erreur ecriture");

    println!("\nFichier genere : {}", filename);
    println!(
        "Girth : {} | Densite : {:.2}%",
        code.girth(),
        code.h.density() * 100.0
    );
} else {
    pb.abandon_with_message("Recherche arrete.");
}

Ok(())
}

```



channel.rs

```
use crate::{Gf2, LdpcError, Llr, Result};

// Trait Channel
pub trait Channel: Send + Sync {
    fn transmit(&self, codeword: &[Gf2], rng: &mut impl rand::Rng) -> Vec<Llr>;
    fn capacity(&self) -> f64;
}

// Canal AWGN
// BPSK mod 0 -> +1.0, 1 -> -1.0
// Signal recu  $y = x + n$ ,  $n \sim N(0, \sigma^2)$ 
// LLR optimal  $L(y) = 2y/\sigma^2$ 
//  $\sigma^2 = N_0/2 = 1/(2 * R * SNR_{lin})$ 

#[derive(Debug, Clone)]
pub struct AwgnChannel {
    pub snr_db: f64,
    sigma: f64,
}

impl AwgnChannel {
    pub fn new(snr_db: f64, code_rate: f64) -> Result<Self> {
        if !(0.0..1.0).contains(&code_rate) {
            return Err(LdpcError::OutOfRange("code_rate ∈ ]0, 1[".into()));
        }
        let snr_lin = 10.0_f64.powf(snr_db / 10.0);
        let sigma = (1.0 / (2.0 * code_rate * snr_lin)).sqrt();
        Ok(Self { snr_db, sigma })
    }
}
```



```

pub fn sigma(&self) -> f64 {
    self.sigma
}
pub fn snr_linear(&self) -> f64 {
    10.0_f64.powf(self.snr_db / 10.0)
}

#[inline]
pub fn llr_from_received(y: f64, sigma: f64) -> Llr {
    2.0 * y / (sigma * sigma)
}
}

impl Channel for AwgnChannel {
    fn transmit(&self, codeword: &[Gf2], rng: &mut impl rand::Rng) -> Vec<Llr> {
        use rand_distr::{Distribution, Normal};
        let normal = Normal::new(0.0, self.sigma).unwrap();
        codeword
            .iter()
            .map(|&b| {
                let x = if b == 0 { 1.0_f64 } else { -1.0_f64 };
                let y = x + normal.sample(rng);
                Self::llr_from_received(y, self.sigma)
            })
            .collect()
    }
}

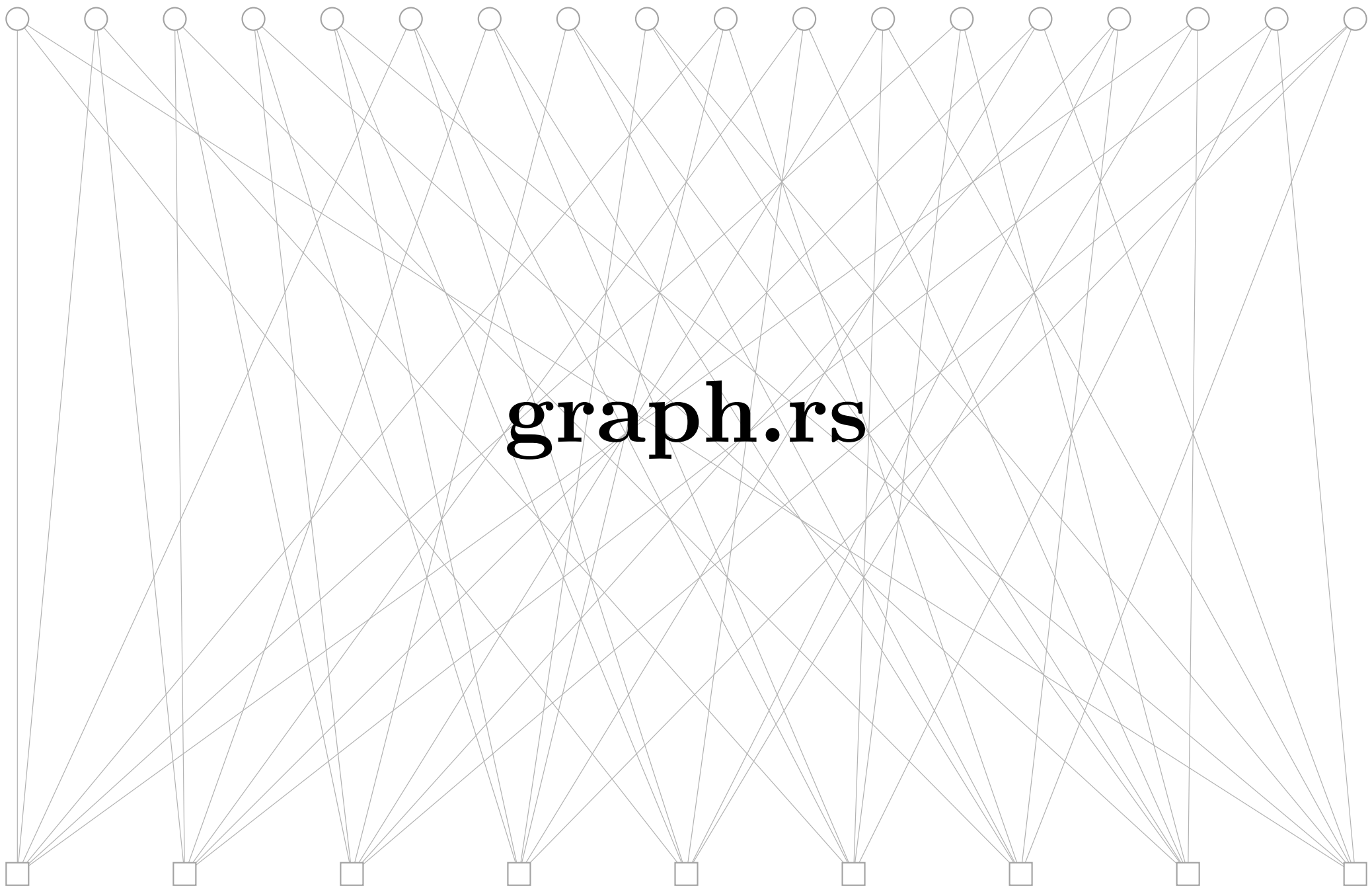
fn capacity(&self) -> f64 {
    // Capacité BPSK-AWGN Monte-Carlo
    use rand_distr::{Distribution, Normal};
    let mut rng = rand::thread_rng();
    let normal = Normal::new(0.0, self.sigma).unwrap();
    let n_samples = 10_000usize;
    let mut sum = 0.0f64;
    for _ in 0..n_samples {

```

```

        let n: f64 = normal.sample(&mut rng);
        let y = 1.0 + n; // bit 0 transmis (x=+1)
        let llr = Self::llr_from_received(y, self.sigma);
        // I = 1 - E[log2(1 + exp(-2y/sig^2))]
        sum += (1.0 + (-llr).exp()).log2();
    }
    1.0 - sum / n_samples as f64
}

```



graph.rs

```
use crate::matrix::SparseMatrixGF2;
use std::collections::VecDeque;

// Graphe de Tanner
#[derive(Debug, Clone)]
pub struct TannerGraph {
    pub n_var: usize,
    pub n_chk: usize,
    var_to_chk: Vec<Vec<usize>>,
    chk_to_var: Vec<Vec<usize>>,
}

impl TannerGraph {
    pub fn from_matrix(h: &SparseMatrixGF2) -> Self {
        let n_var = h.cols;
        let n_chk = h.rows;
        let chk_to_var: Vec<Vec<usize>> = (0..n_chk).map(|c|
h.row_neighbors(c).to_vec()).collect();
        let mut var_to_chk = vec![vec![]; n_var];
        for c in 0..n_chk {
            for &v in &chk_to_var[c] {
                var_to_chk[v].push(c);
            }
        }
        Self {
            n_var,
            n_chk,
            var_to_chk,
            chk_to_var,
        }
    }
}
```

```

}

pub fn var_neighbors(&self, v: usize) -> &[usize] {
    &self.var_to_chk[v]
}
pub fn chk_neighbors(&self, c: usize) -> &[usize] {
    &self.chk_to_var[c]
}
pub fn var_degree(&self, v: usize) -> usize {
    self.var_to_chk[v].len()
}
pub fn chk_degree(&self, c: usize) -> usize {
    self.chk_to_var[c].len()
}

// Calcule le girth par BFS depuis chaque noeud de variable
pub fn girth(&self) -> usize {
    let mut min_girth = usize::MAX;
    for start in 0..self.n_var {
        if let Some(g) = self.bfs_girth_from_var(start) {
            min_girth = min_girth.min(g);
            if min_girth == 4 {
                return 4;
            } // minimum
        }
    }
    min_girth
}

// Détection cycles-4, 2 varnodes partagent >= check-nodes
pub fn has_4_cycle(&self) -> bool {
    for v1 in 0..self.n_var {
        for v2 in (v1 + 1)..self.n_var {
            let common = self.var_to_chk[v1]
                .iter()

```

```

        .filter(|c| self.var_to_chk[v2].contains(c))
        .count();
        if common >= 2 {
            return true;
        }
    }
}
false
}

pub fn local_girth_from_var(&self, v: usize) -> usize {
    self.bfs_girth_from_var(v).unwrap_or(usize::MAX)
}

// retourne la longueur du court cycle passant par ce noeud (None si pas cycle)
fn bfs_girth_from_var(&self, start: usize) -> Option<usize> {
    let mut dist_var = vec![usize::MAX; self.n_var];
    let mut dist_chk = vec![usize::MAX; self.n_chk];
    dist_var[start] = 0;

    // File (is_var, index, distance, parent_index)
    let mut queue: VecDeque<(bool, usize, usize, usize)> = VecDeque::new();
    queue.push_back((true, start, 0, usize::MAX));
    let mut shortest = None;

    while let Some((is_var, node, dist, parent)) = queue.pop_front() {
        if is_var {
            for &c in self.var_neighbors(node) {
                if c == parent {
                    continue;
                } // aller retour immédiat impossible
                if dist_chk[c] == usize::MAX {
                    dist_chk[c] = dist + 1;
                    queue.push_back((false, c, dist + 1, node));
                } else {

```

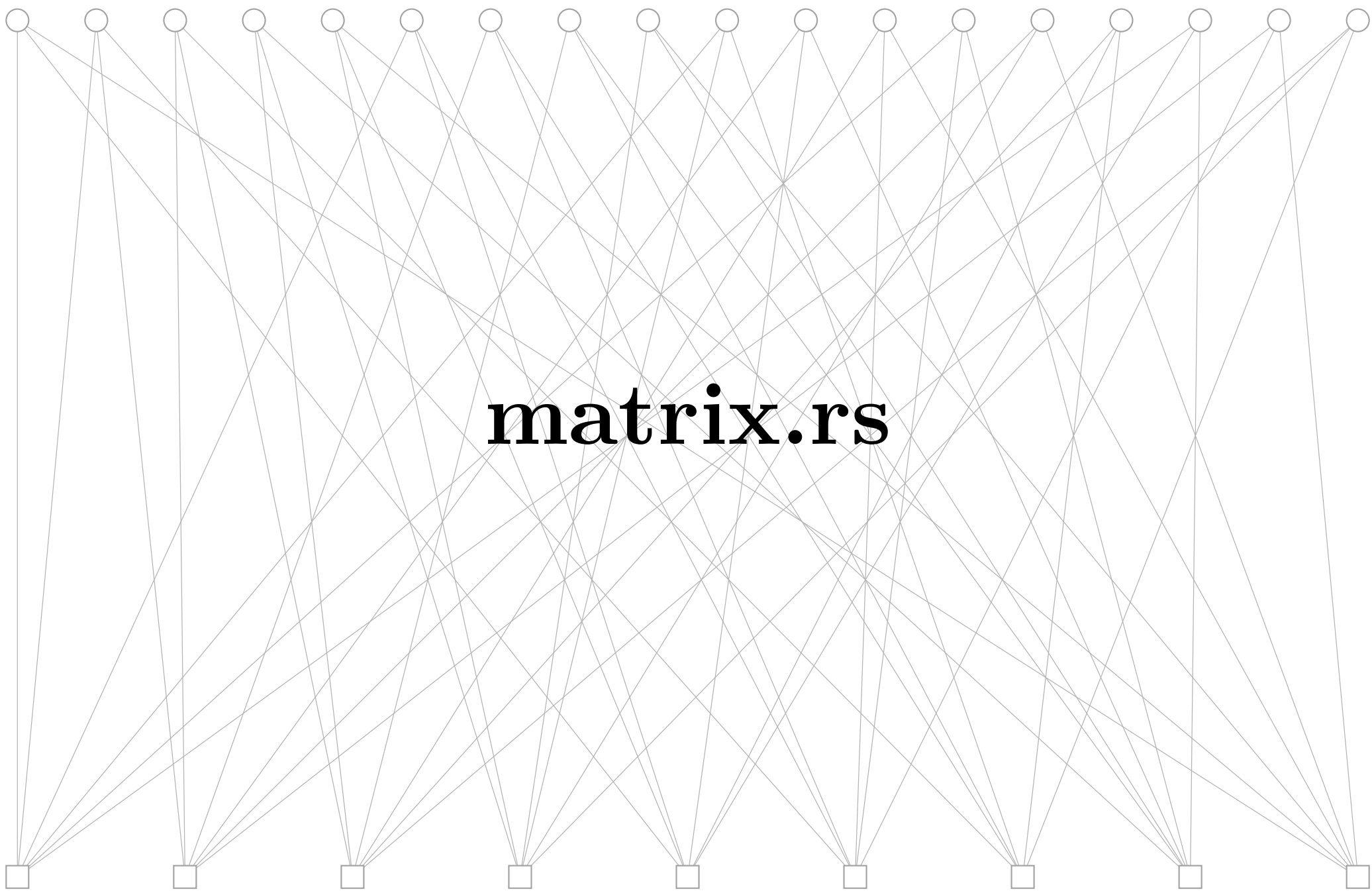
```

        let cycle_len = dist + 1 + dist_chk[c];
        shortest = Some(shortest.map_or(cycle_len, |s: usize| s.min(cycle_len)));
    }
}
} else {
    for &v in self.chk_neighbors(node) {
        if v == parent {
            continue;
        } // aller retour immédiat impossible
        if v == start {
            let cycle_len = dist + 1;
            shortest = Some(shortest.map_or(cycle_len, |s: usize| s.min(cycle_len)));
            continue;
        }
        if dist_var[v] == usize::MAX {
            dist_var[v] = dist + 1;
            queue.push_back((true, v, dist + 1, node));
        }
    }
}
}
shortest
}

pub fn var_degree_distribution(&self) -> Vec<f64> {
    let max_deg = self.var_to_chk.iter().map(|v| v.len()).max().unwrap_or(0);
    let mut counts = vec![0usize; max_deg + 1];
    for v in 0..self.n_var {
        counts[self.var_degree(v)] += 1;
    }
    counts
        .iter()
        .map(|&c| c as f64 / self.n_var as f64)
        .collect()
}

```

```
pub fn is_regular(&self) -> bool {  
    let d0 = self.var_degree(0);  
    let c0 = self.chk_degree(0);  
    self.var_to_chk.iter().all(|v| v.len() == d0)  
        && self.chk_to_var.iter().all(|c| c.len() == c0)  
}  
}
```

matrix.rs

```
use crate::Gf2;
use serde::{Deserialize, Serialize};

// Matrice creuse format CSR + CSC
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct SparseMatrixGF2 {
    pub rows: usize,
    pub cols: usize,
    // CSR accès ligne i : col_idx[row_ptr[i]..row_ptr[i+1]]
    row_ptr: Vec<usize>,
    col_idx: Vec<usize>,
    // CSC accès col j : row_idx[col_ptr[j]..col_ptr[j+1]]
    col_ptr: Vec<usize>,
    row_idx: Vec<usize>,
}

impl SparseMatrixGF2 {
    pub fn zeros(rows: usize, cols: usize) -> Self {
        Self {
            rows,
            cols,
            row_ptr: vec![0; rows + 1],
            col_idx: vec![],
            col_ptr: vec![0; cols + 1],
            row_idx: vec![],
        }
    }
}

// Depuis une liste de (row, col) indiquant les positions des 1s
// Trie les entrées et construit CSR + CSC en un seul passage
```

```

pub fn from_positions(rows: usize, cols: usize, mut ones: Vec<(usize, usize)>) -> Self {
    // Construction CSR
    ones.sort_unstable();
    let mut row_ptr = vec![0usize; rows + 1];
    let mut col_idx = Vec::with_capacity(ones.len());
    for &(r, c) in &ones {
        row_ptr[r + 1] += 1;
        col_idx.push(c);
    }
    for i in 0..rows {
        row_ptr[i + 1] += row_ptr[i];
    }
    // Construction CSC
    let mut col_sorted = ones.clone();
    col_sorted.sort_unstable_by_key(|&(r, c)| (c, r));
    let mut col_ptr = vec![0usize; cols + 1];
    let mut row_idx = Vec::with_capacity(ones.len());
    for &(r, c) in &col_sorted {
        col_ptr[c + 1] += 1;
        row_idx.push(r);
    }
    for j in 0..cols {
        col_ptr[j + 1] += col_ptr[j];
    }
    Self {
        rows,
        cols,
        row_ptr,
        col_idx,
        col_ptr,
        row_idx,
    }
}

pub fn from_dense(dense: &[Vec<Gf2>]) -> Self {

```

```

let rows = dense.len();
let cols = if rows > 0 { dense[0].len() } else { 0 };
let ones: Vec<(usize, usize)> = dense
    .iter()
    .enumerate()
    .flat_map(|(r, row)| {
        row.iter()
            .enumerate()
            .filter(|(_, &v)| v == 1)
            .map(move |(c, _)| (r, c))
    })
    .collect();
Self::from_positions(rows, cols, ones)
}

pub fn get(&self, row: usize, col: usize) -> Gf2 {
    let slice = self.row_neighbors(row);
    if slice.binary_search(&col).is_ok() {
        1
    } else {
        0
    }
}

// Indices des colonnes où la ligne row vaut 1
pub fn row_neighbors(&self, row: usize) -> &[usize] {
    &self.col_idx[self.row_ptr[row]..self.row_ptr[row + 1]]
}

// Indices des lignes où la colonne col vaut 1
pub fn col_neighbors(&self, col: usize) -> &[usize] {
    &self.row_idx[self.col_ptr[col]..self.col_ptr[col + 1]]
}

pub fn row_weight(&self, row: usize) -> usize {

```

```

        self.row_ptr[row + 1] - self.row_ptr[row]
    }

    pub fn col_weight(&self, col: usize) -> usize {
        self.col_ptr[col + 1] - self.col_ptr[col]
    }

    pub fn nnz(&self) -> usize {
        self.col_idx.len()
    }

    pub fn density(&self) -> f64 {
        self.nnz() as f64 / (self.rows * self.cols) as f64
    }

    // Produit H * x mod 2 (syndrome : s = H * c^T)
    pub fn multiply_vec(&self, x: &[Gf2]) -> Vec<Gf2> {
        (0..self.rows)
            .map(|r| {
                self.row_neighbors(r)
                    .iter()
                    .map(|&c| x[c])
                    .fold(0u8, |acc, b| acc ^ b)
            })
            .collect()
    }

    pub fn transpose(&self) -> Self {
        Self {
            rows: self.cols,
            cols: self.rows,
            row_ptr: self.col_ptr.clone(),
            col_idx: self.row_idx.clone(),
            col_ptr: self.row_ptr.clone(),
            row_idx: self.col_idx.clone(),
        }
    }

```

```

    }
}

// Vérifie si deux colonnes partagent >= 2 positions de 1 -> cycle-4 détecté
pub fn columns_share_two_ones(&self, c1: usize, c2: usize) -> bool {
    let n1 = self.col_neighbors(c1);
    let n2 = self.col_neighbors(c2);
    let mut common = 0usize;
    let (mut i, mut j) = (0, 0);
    while i < n1.len() && j < n2.len() {
        match n1[i].cmp(&n2[j]) {
            std::cmp::Ordering::Less => i += 1,
            std::cmp::Ordering::Greater => j += 1,
            std::cmp::Ordering::Equal => {
                common += 1;
                if common >= 2 {
                    return true;
                }
                i += 1;
                j += 1;
            }
        }
    }
    false
}

pub fn to_dense(&self) -> Vec<Vec<Gf2>> {
    let mut out = vec![vec![0u8; self.cols]; self.rows];
    for r in 0..self.rows {
        for &c in self.row_neighbors(r) {
            out[r][c] = 1;
        }
    }
    out
}

```

```

}

// Matrice dense
// Utilisée pour G et Gauss-Jordan
// G = [I | P], P dense

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct DenseMatrixGF2 {
    pub rows: usize,
    pub cols: usize,
    data: Vec<Vec<Gf2>>,
}

impl DenseMatrixGF2 {
    pub fn zeros(rows: usize, cols: usize) -> Self {
        Self {
            rows,
            cols,
            data: vec![vec![0u8; cols]; rows],
        }
    }

    pub fn identity(n: usize) -> Self {
        let mut m = Self::zeros(n, n);
        for i in 0..n {
            m.data[i][i] = 1;
        }
        m
    }

    pub fn get(&self, row: usize, col: usize) -> Gf2 {
        self.data[row][col]
    }

    pub fn set(&mut self, row: usize, col: usize, val: Gf2) {
        self.data[row][col] = val;
    }
}

```

```

}

pub fn row_add(&mut self, dst: usize, src: usize) {
    for j in 0..self.cols {
        self.data[dst][j] ^= self.data[src][j];
    }
}

pub fn row_swap(&mut self, r1: usize, r2: usize) {
    self.data.swap(r1, r2);
}

pub fn col_swap(&mut self, c1: usize, c2: usize) {
    for r in 0..self.rows {
        let tmp = self.data[r][c1];
        self.data[r][c1] = self.data[r][c2];
        self.data[r][c2] = tmp;
    }
}

pub fn multiply_vec(&self, x: &[Gf2]) -> Vec<Gf2> {
    self.data
        .iter()
        .map(|row| {
            row.iter()
                .zip(x.iter())
                .fold(0u8, |acc, (&a, &b)| acc ^ (a & b))
        })
        .collect()
}

pub fn into_sparse(self) -> SparseMatrixGF2 {
    SparseMatrixGF2::from_dense(&self.data)
}

```



```

pub fn systematize(&mut self, k: usize) -> (Vec<usize>, usize) {
    let m = self.rows;
    let n = self.cols;
    let mut col_perm: Vec<usize> = (0..n).collect();
    let mut rank = 0;

    for i in 0..m {
        // Placer le pivot ligne i à la colonne target_c (former I_m à droite)
        let target_c = k + i;
        let mut pivot_found = false;

        // pivot, chercher en premier dans les colonnes cibles restantes
        // et après dans les colonnes de données (0..k)
        for c_search in (target_c..n).chain(0..k) {
            for r_search in i..m {
                if self.data[r_search][c_search] == 1 {
                    // pivot à la position (i, target_c)
                    self.row_swap(i, r_search);
                    if c_search != target_c {
                        self.col_swap(target_c, c_search);
                        col_perm.swap(target_c, c_search);
                    }
                    pivot_found = true;
                    break;
                }
            }
            if pivot_found {
                break;
            }
        }

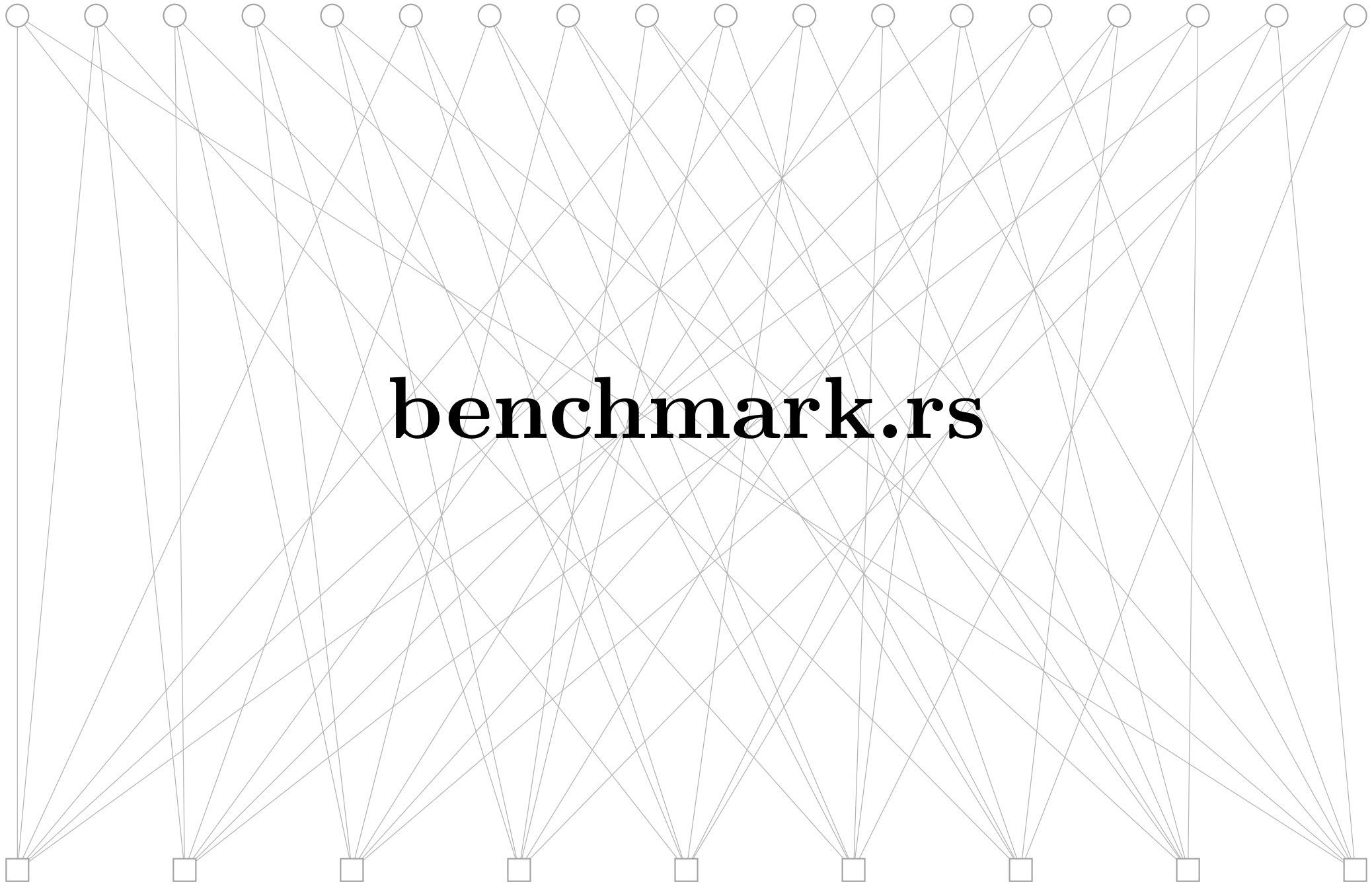
        if pivot_found {
            rank += 1;
            // Elimination dans toutes les autres lignes => forcer la colonne à 0 (sauf le
            // pivot à 1)

```

```

        for r in 0..m {
            if r != i && self.data[r][target_c] == 1 {
                self.row_add(r, i);
            }
        }
    }
    (col_perm, rank)
}

```



benchmark.rs

```
use crate::channel::{AwgnChannel, Channel};
use crate::code::{CodeTopology, GenerationMethod, LdpcCode, LdpcParams};
use crate::decoder::{build_decoder, DecoderConfig, DecoderMethod};
use crate::encoder::{build_encoder, EncodingMethod};
use crate::Result as LdpcResult;
use rand::Rng;
use rayon::prelude::*;
use std::fs::File;
use std::io::{Read, Write};
use std::path::Path;
use std::time::Instant;

pub fn run_simulation(mut code: LdpcCode) -> LdpcResult<()> {
    println!("[*] Étape 1 : Construction Mathématique et Graphe de Tanner");

    println!(
        "    - Dimensions : n={}, k={}, m={} (Taux R={:.3})",
        code.n(),
        code.k(),
        code.m(),
        code.rate()
    );

    let methode_nom = match code.params.generation {
        GenerationMethod::MacKayNeal { .. } => "MacKay-Neal",
        GenerationMethod::Gallager => "Gallager",
    };

    println!("    - Topologie : Régulier via {}", methode_nom);
    println!("    - Densité H : {:.2}%", code.h.density() * 100.0);
}
```

```

println!("    - Girth      : {}", code.girth());
println!(
    "    - Cycles-4    : {}",
    if code.graph.has_4_cycle() {
        "Présents (Problématique)"
    } else {
        "Aucun (Optimal)"
    }
);

println!("\n[*] Étape 2 : Instanciation de l'Encodeur (SIMD Bit-Packing)");
let start_enc = Instant::now();
let encoder = build_encoder(&mut code, EncodingMethod::Systematic?);
println!("    - Encodeur prêt en {:.2?}", start_enc.elapsed());

println!("\n[*] Étape 3 : Instanciation des Décodeurs sur Graphe");
let config = DecoderConfig {
    max_iterations: 50,
    early_stopping: true,
};

let dec_sp = build_decoder(&code, DecoderMethod::SumProduct, config.clone());
let dec_ms = build_decoder(
    &code,
    DecoderMethod::MinSum {
        scaling_factor: 0.8,
    },
    config.clone(),
);
let dec_bf = build_decoder(&code, DecoderMethod::BitFlipping, config.clone());
println!("    - Moteurs prêts : Sum-Product, Min-Sum ( $\alpha=0.8$ ), Bit-Flipping");

let snr_range = [1.0, 1.25, 1.5, 1.75, 2.0, 2.25, 2.5, 3.0, 3.5, 4.0];
let n_trials = 1000;

```

```
println!(
    "\n[*] Étape 4 : Simulation sur Canal AWGN ({ trames par SNR, Multi-threadé)",
    n_trials
);
println!("{:-<115}", "");
println!(
    "{:>8} | {:>9} || {:>10} | {:>10} || {:>10} | {:>10} || {:>10} | {:>10} |",
    "SNR (dB)",
    "Capacité",
    "FER (SP)",
    "BER (SP)",
    "FER (MS)",
    "BER (MS)",
    "FER (BF)",
    "BER (BF)"
);
println!("{:-<115}", "");

for &snr in &snr_range {
    let channel = AwgnChannel::new(snr, code.rate());

    let results: Vec<_> = (0..n_trials)
        .into_par_iter()
        .map(|_| {
            let mut rng = rand::thread_rng();

            let message: Vec<u8> = (0..code.k()).map(|_| rng.gen:::<u8>() & 1).collect();
            let codeword = encoder.encode(&message).unwrap();
            let received_llr = channel.transmit(&codeword, &mut rng);

            let mut t_sp_f = 0;
            let mut t_sp_b = 0;
            let mut t_ms_f = 0;
            let mut t_ms_b = 0;
            let mut t_bf_f = 0;

```

```
let mut t_bf_b = 0;

// SP
let res_sp = dec_sp.decode(&received_llr);
if let Some(decoded) = res_sp.codeword() {
    let errs = count_bit_errors(&codeword, decoded);
    if errs > 0 {
        t_sp_f = 1;
        t_sp_b = errs;
    }
} else {
    t_sp_f = 1;
    t_sp_b = code.n();
}

// MS
let res_ms = dec_ms.decode(&received_llr);
if let Some(decoded) = res_ms.codeword() {
    let errs = count_bit_errors(&codeword, decoded);
    if errs > 0 {
        t_ms_f = 1;
        t_ms_b = errs;
    }
} else {
    t_ms_f = 1;
    t_ms_b = code.n();
}

// BF
let res_bf = dec_bf.decode(&received_llr);
if let Some(decoded) = res_bf.codeword() {
    let errs = count_bit_errors(&codeword, decoded);
    if errs > 0 {
        t_bf_f = 1;
        t_bf_b = errs;
    }
}
```

```

        }
    } else {
        t_bf_f = 1;
        t_bf_b = code.n();
    }

    (t_sp_f, t_sp_b, t_ms_f, t_ms_b, t_bf_f, t_bf_b)
})
.collect();

// Agrégation
let mut err_sp_frames = 0;
let mut err_sp_bits = 0;
let mut err_ms_frames = 0;
let mut err_ms_bits = 0;
let mut err_bf_frames = 0;
let mut err_bf_bits = 0;

for res in results {
    err_sp_frames += res.0;
    err_sp_bits += res.1;
    err_ms_frames += res.2;
    err_ms_bits += res.3;
    err_bf_frames += res.4;
    err_bf_bits += res.5;
}

let total_bits = (n_trials * code.n()) as f64;
let total_frames = n_trials as f64;

println!(
    "{:>8.2} | {:>9.4} || {:>9.2}% | {:>9.2}% || {:>9.2}% | {:>9.2}% || {:>9.2}% |
{:>9.2}% |",
    snr, channel.capacity(),
    (err_sp_frames as f64 / total_frames) * 100.0, (err_sp_bits as f64 / total_bits) *

```



```

100.0,
    (err_ms_frames as f64 / total_frames) * 100.0, (err_ms_bits as f64 / total_bits) *
100.0,
    (err_bf_frames as f64 / total_frames) * 100.0, (err_bf_bits as f64 / total_bits) *
100.0
    );
}
println!("{: <115}", "");
Ok(())
}

#[inline]
fn count_bit_errors(transmitted: &[u8], decoded: &[u8]) -> usize {
    transmitted
        .iter()
        .zip(decoded.iter())
        .filter(|(a, b)| a != b)
        .count()
}

// pub fn generate_valid_code(
//     n: usize,
//     k: usize,
//     wc: usize,
//     wr: usize,
//     generation_method: GenerationMethod,
// ) -> LdpcResult<LdpcCode> {
//     let mut attempt = 0;
//     let start_gen = Instant::now();
//     loop {
//         attempt += 1;
//         let params = LdpcParams {
//             n,
//             k,
//             topology: CodeTopology::Regular { wc, wr },

```

```

//          generation: generation_method.clone(),
//          seed: Some(rand::random()),
//      };
//
//      if let Ok(mut code) = LdpcCode::new(params) {
//          if code.compute_systematic_form().is_ok() {
//              if attempt > 1 {
//                  println!(
//                      "      -> Matrice inversible obtenue après {} tentatives.",
//                      attempt
//                  );
//              }
//              println!("      - Génération : Terminée en {:.2?}", start_gen.elapsed());
//              return Ok(code);
//          }
//      }
//  }
// }

```

```

pub fn get_or_generate_cached_code(
    n: usize,
    k: usize,
    wc: usize,
    wr: usize,
    generation_method: GenerationMethod,
) -> LdpcResult<LdpcCode> {
    let method_str = match generation_method {
        GenerationMethod::MacKayNeal { .. } => "MN",
        GenerationMethod::Gallager => "GAL",
    };
    let cache_filename = format!("cache_ldpc_{}_n{}_k{}.bin", method_str, n, k);
    let path = Path::new(&cache_filename);

    // Chargement
    if path.exists() {

```

```

let start_load = Instant::now();
let mut file = File::open(&path).expect("Impossible d'ouvrir le fichier de cache");
let mut buffer = Vec::new();
file.read_to_end(&mut buffer).unwrap();

let mut code: LdpcCode = bincode::deserialize(&buffer).expect(
    "Erreur de désérialisation du cache LDPC. Supprimez le fichier .bin et réessayez.",
);

// Reconstruction du Graphe de Tanner
code.graph = crate::graph::TannerGraph::from_matrix(&code.h);

println!(
    "    -> Matrice chargée depuis le cache en {:.2?}",
    start_load.elapsed()
);
return Ok(code);
}

// Génération
println!("    -> Aucun cache trouvé. Génération en cours ...");
let mut attempt = 0;
let start_gen = Instant::now();

loop {
    attempt += 1;
    let params = LdpcParams {
        n,
        k,
        topology: CodeTopology::Regular { wc, wr },
        generation: generation_method.clone(),
        seed: Some(rand::random()),
    };

    if let Ok(mut code) = LdpcCode::new(params) {

```

```

if code.compute_systematic_form().is_ok() {
    println!(
        "    - Génération et Pivot de Gauss terminés en {:.2?}",
        start_gen.elapsed()
    );

    // sauvegarde cache
    let encoded = bincode::serialize(&code).expect("Échec de la sérialisation");
    let mut file =
        File::create(&path).expect("Impossible de créer le fichier de cache");
    file.write_all(&encoded)
        .expect("Impossible d'écrire sur le disque");
    println!("    -> Matrice sauvegardée ({})", cache_filename);

    return Ok(code);
}
}
}
}

```



benchmark2.rs

benchmark2.rs

```
use crate::benchmark::get_or_generate_cached_code;
use crate::channel::{AwgnChannel, Channel};
use crate::code::GenerationMethod;
use crate::decoder::{build_decoder, DecoderConfig, DecoderMethod};
use crate::encoder::{build_encoder, EncodingMethod};
use crate::Result as LdpcResult;

use indicatif::{ProgressBar, ProgressStyle};
use rand::Rng;
use rayon::prelude::*;
use std::fs::OpenOptions;
use std::io::Write;
use std::time::Instant;

#[derive(Clone)]
pub struct CodeScenario {
    pub n: usize,
    pub k: usize,
    pub wc: usize,
    pub wr: usize,
    pub method: GenerationMethod,
    pub name: String,
}

pub struct CampaignConfig {
    pub scenarios: Vec<CodeScenario>,
    pub snr_range: Vec<f64>,
    pub n_trials: usize,
    pub max_iterations: usize,
    pub output_csv: String,
```

```

    pub export_graph: bool,
}

pub fn run_massive_campaign(config: CampaignConfig) -> LdpcResult<()> {
    println!("TEST");
    println!("Fichier de sortie : {}", config.output_csv);
    println!("Scenarios a tester: {}", config.scenarios.len());
    println!("SNRs par scenario : {}", config.snr_range.len());
    println!("Trames par point : {}\n", config.n_trials);

    let file_exists = std::path::Path::new(&config.output_csv).exists();
    let mut file = OpenOptions::new()
        .create(true)
        .append(true)
        .open(&config.output_csv)
        .expect("Impossible d'ouvrir le fichier CSV");

    if !file_exists {
        writeln!(
            file,
            "Scenario,N,K,Rate,Wc,Wr,Method,Girth,Density_pct,SNR_dB,Capacity,Decoder,FER_pct,BER_pct,Frames,Time_ms"
        ).unwrap();
    }

    let total_steps = (config.scenarios.len() * config.snr_range.len()) as u64;
    let pb = ProgressBar::new(total_steps);
    pb.set_style(
        ProgressStyle::default_bar()
            .template("{spinner:.cyan} [{elapsed_precise}] [{bar:40.green/blue}] {pos}/{len}"
            ({eta}) - {msg}")
            .unwrap()
            .progress_chars("=>-"),
    );
};

```

```

for scenario in config.scenarios {
    pb.set_message(format!("Generation Matrice: {}", scenario.name));

    let mut code = get_or_generate_cached_code(
        scenario.n,
        scenario.k,
        scenario.wc,
        scenario.wr,
        scenario.method.clone(),
    );

    if config.export_graph {
        let dot_filename = format!("{}_tanner.dot", scenario.name);
        let mut dot_file = std::fs::File::create(&dot_filename).expect("Erreur
creation .dot");

        writeln!(dot_file, "graph TannerGraph {{").unwrap();
        writeln!(dot_file, "    rankdir=TB;").unwrap();
        writeln!(dot_file, "    nodesep=0.5;").unwrap();
        writeln!(dot_file, "    ranksep=2.0;").unwrap();

        writeln!(dot_file, "    node [style=filled, fontname=\"Arial\"];").unwrap();

        writeln!(dot_file, "    {{ rank=same; }).unwrap();
        for v in 0..code.n() {
            writeln!(
                dot_file,
                "        v{{ [shape=circle, fillcolor=lightblue, label=\"v{{\\\"};",
                v, v
            )
            .unwrap();
        }
        writeln!(dot_file, "    }}").unwrap();

        writeln!(dot_file, "    {{ rank=same; }).unwrap();

```



```

for c in 0..code.m() {
  writeln!(
    dot_file,
    "          c{} [shape=square, fillcolor=lightcoral, label=\"c{}\\\"];",
    c, c
  )
  .unwrap();
}
writeln!(dot_file, "    }}").unwrap();

for c in 0..code.m() {
  for &v in code.graph.chk_neighbors(c) {
    writeln!(dot_file, "      v{} -- c{};", v, c).unwrap();
  }
}
writeln!(dot_file, "}}").unwrap();
}

let rate = code.rate();
let girth = code.girth();
let density = code.h.density() * 100.0;
let method_str = match scenario.method {
  GenerationMethod::Gallager => "Gallager",
  GenerationMethod::MacKayNeal { .. } => "MacKayNeal",
};

pb.set_message(format!("Initialisation DSP: {}", scenario.name));
let encoder = build_encoder(&mut code, EncodingMethod::Systematic?);

let dec_config = DecoderConfig {
  max_iterations: config.max_iterations,
  early_stopping: true,
};
let dec_sp = build_decoder(&code, DecoderMethod::SumProduct, dec_config.clone());
let dec_ms = build_decoder(

```

```

        &code,
        DecoderMethod::MinSum {
            scaling_factor: 0.8,
        },
        dec_config.clone(),
    );
    let dec_bf = build_decoder(&code, DecoderMethod::BitFlipping, dec_config);

    for &snr in &config.snr_range {
        pb.set_message(format!("Test: {} | SNR: {:.2} dB", scenario.name, snr));
        let channel = AwgnChannel::new(snr, rate)?;
        let cap = channel.capacity();

        let start_snr_time = Instant::now();

        let results: Vec<_> = (0..config.n_trials)
            .into_par_iter()
            .map(|_| {
                let mut rng = rand::thread_rng();
                let message: Vec<u8> = (0..scenario.k).map(|_| rng.gen::<u8>()) &
1).collect();

                let codeword = encoder.encode(&message).unwrap();
                let rx_llr = channel.transmit(&codeword, &mut rng);

                let eval_decoder = |decoder: &dyn crate::decoder::Decoder| -> (usize, usize)
{
                    if let Some(decoded) = decoder.decode(&rx_llr).codeword() {
                        let errs = codeword
                            .iter()
                            .zip(decoded.iter())
                            .filter(|(a, b)| a != b)
                            .count();
                        (if errs > 0 { 1 } else { 0 }, errs)
                    } else {
                        (1, scenario.n)
                    }
                }
            })
    }

```

```

    }
};

let (sp_f, sp_b) = eval_decoder(&*dec_sp);
let (ms_f, ms_b) = eval_decoder(&*dec_ms);
let (bf_f, bf_b) = eval_decoder(&*dec_bf);

(sp_f, sp_b, ms_f, ms_b, bf_f, bf_b)
})
.collect();

let mut sp_err = (0, 0);
let mut ms_err = (0, 0);
let mut bf_err = (0, 0);
for r in results {
    sp_err.0 += r.0;
    sp_err.1 += r.1;
    ms_err.0 += r.2;
    ms_err.1 += r.3;
    bf_err.0 += r.4;
    bf_err.1 += r.5;
}

let elapsed_ms = start_snr_time.elapsed().as_millis();
let total_f = config.n_trials as f64;
let total_b = (config.n_trials * scenario.n) as f64;

let mut write_csv_line = |dec_name: &str, f_err: usize, b_err: usize| {
    writeln!(
        file,
        "{} {}, {}, {:.3}, {}, {}, {}, {}, {:.4}, {:.2}, {:.4}, {}, {:.4}, {:.4e}, {}, {}",
        scenario.name,
        scenario.n,
        scenario.k,
        rate,

```

```

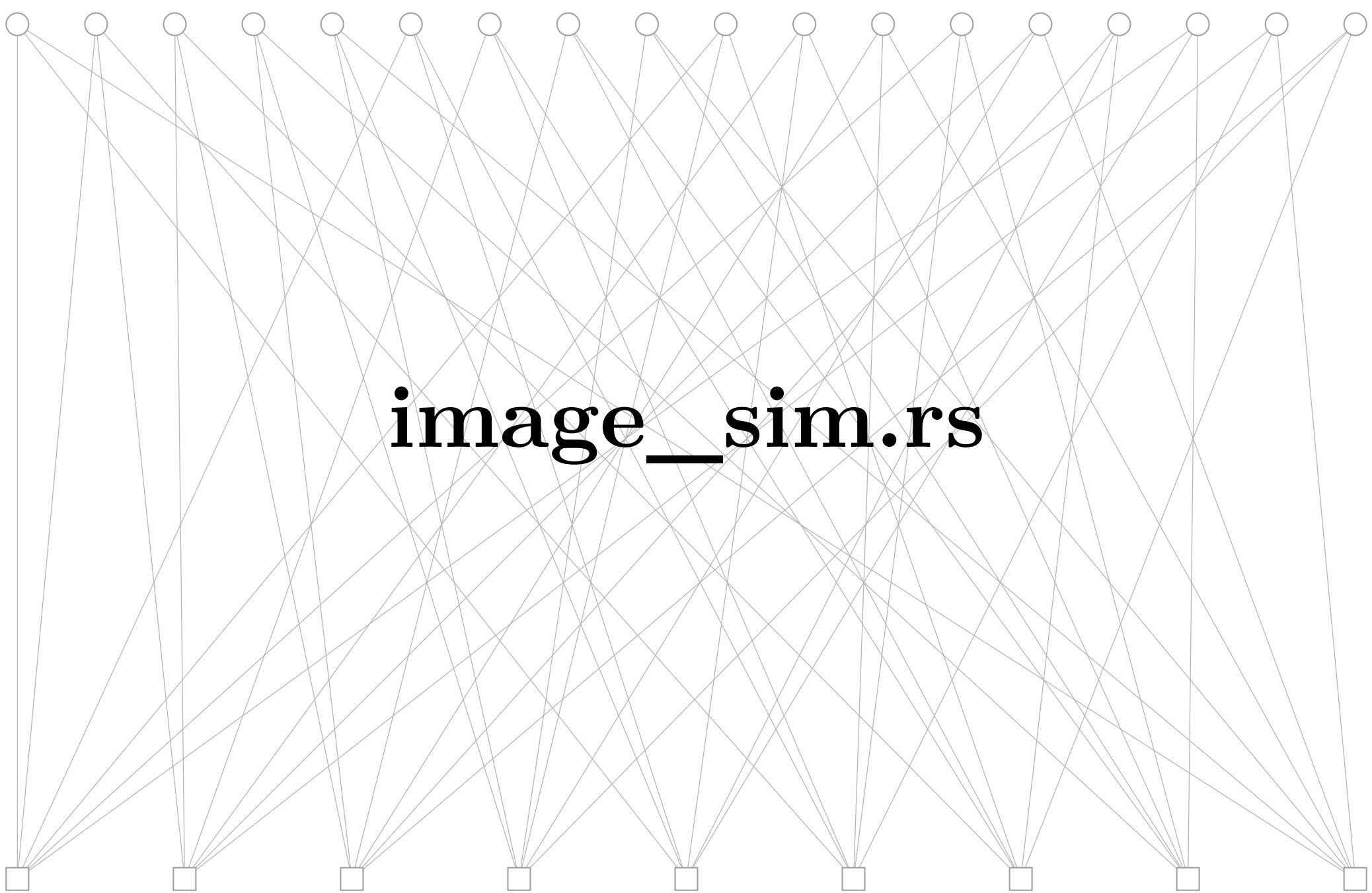
        scenario.wc,
        scenario.wr,
        method_str,
        girth,
        density,
        snr,
        cap,
        dec_name,
        (f_err as f64 / total_f) * 100.0,
        (b_err as f64 / total_b) * 100.0,
        config.n_trials,
        elapsed_ms
    )
    .unwrap();
};

write_csv_line("SumProduct", sp_err.0, sp_err.1);
write_csv_line("MinSum_0.8", ms_err.0, ms_err.1);
write_csv_line("BitFlipping", bf_err.0, bf_err.1);

file.flush().unwrap();
pb.inc(1);
}
}

pb.finish_with_message("Campagne terminee avec succes !");
Ok(())
}

```



image_sim.rs

```
use crate::{
    channel::{AwgnChannel, Channel},
    decoder::Decoder,
    encoder::Encoder,
    Gf2, Result,
};
use image::{ImageBuffer, Rgb};
use indicatif::{ProgressBar, ProgressStyle};
use rayon::prelude::*;
use std::time::Instant;

// Convertit un tableau d'octets en un flux de bits
pub fn bytes_to_bits(bytes: &[u8]) -> Vec<Gf2> {
    let mut bits = Vec::with_capacity(bytes.len() * 8);
    for &byte in bytes {
        for i in (0..8).rev() {
            bits.push((byte >> i) & 1);
        }
    }
    bits
}

// Convertit un flux de bits en tableau d'octets
pub fn bits_to_bytes(bits: &[Gf2]) -> Vec<u8> {
    let mut bytes = Vec::with_capacity(bits.len() / 8);
    for chunk in bits.chunks(8) {
        let mut byte = 0u8;
        for (i, &bit) in chunk.iter().enumerate() {
            byte |= bit << (7 - i);
        }
    }
}
```

```

        bytes.push(byte);
    }
    bytes
}

// Transmet une image à travers le canal avec codage LDPC
// pub fn transmit_image(
//     input_path: &str,
//     noisy_out_path: &str,
//     decoded_out_path: &str,
//     encoder: &dyn Encoder,
//     decoder: &dyn Decoder,
//     channel: &AwgnChannel,
// ) -> Result<()> {
//     println!("[*] Chargement de l'image : {}", input_path);
//     let img = image::open(input_path)
//         .expect("Erreur de chargement de l'image")
//         .to_rgb8();
//     let (width, height) = img.dimensions();
//     let raw_bytes = img.into_raw();
//
//     let mut bits = bytes_to_bits(&raw_bytes);
//     let original_bit_len = bits.len();
//
//     // Padding
//     let k = encoder.message_len();
//     let remainder = bits.len() % k;
//     if remainder != 0 {
//         bits.resize(bits.len() + (k - remainder), 0);
//     }
//
//     let num_blocks = bits.len() / k;
//     println!("    - Taille: {}x{} pixels", width, height);
//     println!("    - Blocs à transmettre (k={}): {}", k, num_blocks);
//
//

```

```

//      let mut rng = rand::rngs::StdRng::seed_from_u64(42);
//
//      let mut noisy_bits = Vec::with_capacity(num_blocks * k);
//      let mut decoded_bits = Vec::with_capacity(num_blocks * k);
//
//      let mut frame_errors = 0;
//
//      println!("[*] Transmission et Décodage en cours...");
//
//      for (i, block) in bits.chunks(k).enumerate() {
//          if i % 100 == 0 && i > 0 {
//              println!("      - Progression: {} / {} blocs...", i, num_blocks);
//          }
//
//          let codeword = encoder.encode(block)?;
//
//          let rx_llr = channel.transmit(&codeword, &mut rng);
//
//          // Sans correction LDPC
//          let hard_codeword: Vec<Gf2> = rx_llr
//              .iter()
//              .map(|&l| if l < 0.0 { 1 } else { 0 })
//              .collect();
//          let noisy_block = encoder.extract_message(&hard_codeword);
//          noisy_bits.extend_from_slice(&noisy_block);
//
//          // Décodage LDPC
//          let res = decoder.decode(&rx_llr);
//          if let Some(decoded_codeword) = res.codeword() {
//              let decoded_msg = encoder.extract_message(decoded_codeword);
//              decoded_bits.extend_from_slice(&decoded_msg);
//
//              if decoded_msg != block {
//                  frame_errors += 1;
//              }
//          }
//      }

```



```

//      } else {
//          decoded_bits.extend_from_slice(&noisy_block);
//          frame_errors += 1;
//      }
//  }
//
//  println!(
//      "[*] Transmission terminée. FER : {:.2}%",
//      (frame_errors as f64 / num_blocks as f64) * 100.0
//  );
//
//  // Suppression du padding
//  noisy_bits.truncate(original_bit_len);
//  decoded_bits.truncate(original_bit_len);
//
//  // Reconstitution des images
//  let noisy_bytes = bits_to_bytes(&noisy_bits);
//  let decoded_bytes = bits_to_bytes(&decoded_bits);
//
//  let noisy_img = ImageBuffer::<Rgb<u8>, _>::from_raw(width, height, noisy_bytes)
//      .expect("Erreur de reconstruction de l'image bruitée");
//  noisy_img.save(noisy_out_path).unwrap();
//
//  let decoded_img = ImageBuffer::<Rgb<u8>, _>::from_raw(width, height, decoded_bytes)
//      .expect("Erreur de reconstruction de l'image décodée");
//  decoded_img.save(decoded_out_path).unwrap();
//
//  println!(
//      "[*] Images sauvegardées : {} et {}",
//      noisy_out_path, decoded_out_path
//  );
//  Ok(())
// }
pub fn transmit_image(
    input_path: &str,

```

```

noisy_out_path: &str,
decoded_out_path: &str,
encoder: &dyn Encoder,
decoder: &dyn Decoder,
channel: &AwgnChannel,
) -> Result<()> {
    println!("{}", "Chargement de l'image : {}", input_path);
    let img = image::open(input_path)
        .expect("Erreur : Impossible de trouver ou lire l'image")
        .to_rgb8();
    let (width, height) = img.dimensions();
    let raw_bytes = img.into_raw();

    let mut bits = bytes_to_bits(&raw_bytes);
    let original_bit_len = bits.len();

    let k = encoder.message_len();
    let remainder = bits.len() % k;
    if remainder != 0 {
        bits.resize(bits.len() + (k - remainder), 0);
    }

    let num_blocks = bits.len() / k;
    println!("{}", "Résolution : {}x{} pixels", width, height);
    println!(
        "{} - Poids total : {:.2} Mo",
        raw_bytes.len() as f64 / 1_048_576.0
    );
    println!("{}", "Découpage en : {} blocs de {} bits\n", num_blocks, k);

    let start_time = Instant::now();

    let pb = ProgressBar::new(num_blocks as u64);
    pb.set_style(
        ProgressStyle::default_bar()

```

```

        .template("{spinner:.green} [{elapsed_precise}] [{bar:40.cyan/blue}] {pos}/{len}"
blocs ({eta}))")
        .unwrap()
        .progress_chars("=>-")
    );

let results: Vec<_> = bits
    .par_chunks(k)
    .map(|block| {
        let mut rng = rand::thread_rng();
        let codeword = encoder.encode(block).unwrap();
        let rx_llr = channel.transmit(&codeword, &mut rng);
        let hard_codeword: Vec<Gf2> = rx_llr
            .iter()
            .map(|&l| if l < 0.0 { 1 } else { 0 })
            .collect();
        let noisy_block = encoder.extract_message(&hard_codeword);
        let res = decoder.decode(&rx_llr);

        let (decoded_block, has_error) = if let Some(decoded_codeword) = res.codeword() {
            let decoded_msg = encoder.extract_message(decoded_codeword);
            let err = if decoded_msg != block { 1 } else { 0 };
            (decoded_msg, err)
        } else {
            (noisy_block.clone(), 1)
        };
        pb.inc(1);
        (noisy_block, decoded_block, has_error)
    })
    .collect();

pb.finish_with_message("Décodage terminé !");

let mut noisy_bits = Vec::with_capacity(num_blocks * k);
let mut decoded_bits = Vec::with_capacity(num_blocks * k);

```

```

let mut frame_errors = 0;

for (n_block, d_block, err) in results {
    noisy_bits.extend_from_slice(&n_block);
    decoded_bits.extend_from_slice(&d_block);
    frame_errors += err;
}

let duration = start_time.elapsed();
let fer = (frame_errors as f64 / num_blocks as f64) * 100.0;
println!("[*] Transmission terminée en {:.2?}", duration);
println!("    - Taux d'Erreur Trame (FER) : {:.2}%", fer);

noisy_bits.truncate(original_bit_len);
decoded_bits.truncate(original_bit_len);

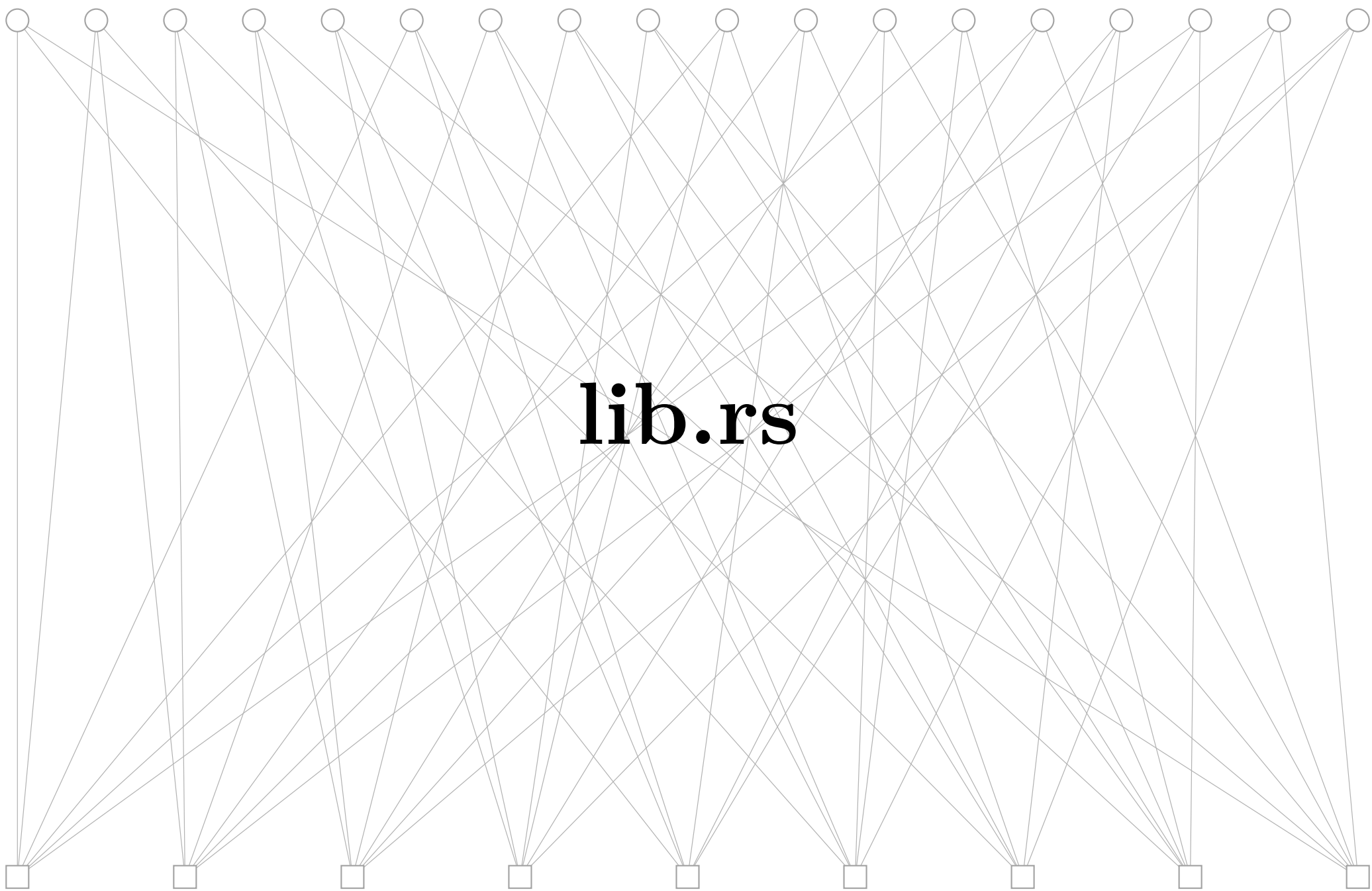
let noisy_bytes = bits_to_bytes(&noisy_bits);
let decoded_bytes = bits_to_bytes(&decoded_bits);

let noisy_img = ImageBuffer::<Rgb<u8>, _>::from_raw(width, height, noisy_bytes)
    .expect("Erreur de reconstruction de l'image bruitée");
noisy_img.save(noisy_out_path).unwrap();

let decoded_img = ImageBuffer::<Rgb<u8>, _>::from_raw(width, height, decoded_bytes)
    .expect("Erreur de reconstruction de l'image décodée");
decoded_img.save(decoded_out_path).unwrap();

println!(
    "[*] Succès ! Images sauvegardées : {} et {}",
    noisy_out_path, decoded_out_path
);
Ok(())
}

```



```
pub mod benchmark;
pub mod benchmark2;
pub mod channel;
pub mod code;
pub mod decoder;
pub mod encoder;
pub mod graph;
pub mod image_sim;
pub mod matrix;

pub type Gf2 = u8;
pub type Llr = f64;
pub type BitVec = Vec<Gf2>;

#[derive(Debug, thiserror::Error)]
pub enum LdpcError {
    #[error("Paramètres invalides : {0}")]
    InvalidParameters(String),

    #[error("Matrice singulière : impossible d'inverser")]
    SingularMatrix,

    #[error("Génération échouée après {attempts} tentatives")]
    GenerationFailed { attempts: usize },

    #[error("Dimension incorrecte : attendu {expected}, reçu {got}")]
    DimensionMismatch { expected: usize, got: usize },

    #[error("Le vecteur fourni n'est pas un mot de code valide")]
    InvalidCodeword,
```

```
#[error("Paramètre hors plage : {0}")]
    OutOfRange(String),
}

pub type Result<T> = std::result::Result<T, LdpcError>;

pub use channel::Channel;
pub use code::{CodeTopology, GenerationMethod, LdpcCode, LdpcParams};
pub use decoder::{Decoder, DecoderConfig, DecoderMethod, DecoderResult};
pub use encoder::{Encoder, EncodingMethod};
```