UMONS
Université de Mons

Mémoire

Codes détecteurs et correcteurs d'erreurs : découverte et expérimentation

Silon Christophe
christophe.silon@student.umons.ac.be
Directeur de mémoire : Quoitin Bruno

Master en Sciences Informatiques

2022-2023

Remerciements

Je tiens à exprimer ma profonde gratitude envers mon directeur de mémoire Bruno Quoitin, pour ses conseils et son engagement tout au long de la réalisation de ce mémoire. Je le remercie sincèrement d'avoir consacré du temps à la lecture attentive de mon mémoire, d'avoir apporté des suggestions précieuses pour son amélioration. Ses conseils éclairés et sa capacité à me guider dans la bonne direction ont été essentiels pour mener à bien ce projet.

Mes remerciements s'étendent également à toutes les personnes qui ont contribué, de près ou de loin, à la réalisation de ce travail. Vos conseils, discussions et suggestions ont enrichi ma réflexion et ont contribué à façonner ce mémoire.

Sommaire

1 Introduction

Il est essentiel de pouvoir transmettre ou stocker des informations sans que celles-ci ne soient corrompues. Quel que soit le canal ou le support considéré, celui-ci ne peut jamais être sûr, que ce soit au niveau d'une erreur lors de la transmission physique ou bien encore lors de l'altération du message par un des tiers plus ou moins bien intentionnés. Il est alors nécessaire de pouvoir définir des codes permettant de transformer l'information, de telle sorte à pouvoir soit détecter les erreurs présentes dans une information soit les corriger. Ces codes doivent permettre de coder et décoder rapidement l'information, ainsi que d'éviter une trop grande taille de l'information à la suite d'une transformation.

Ce mémoire a pour but de définir certains codes permettant de pouvoir détecter et/ou de corriger les erreurs. Les codes permettant de sécuriser l'information ne sont pas couverts. La première section couvre en partie l'état de l'art de la théories des codes ainsi que l'état de l'art sur des codes tels que le code à bit de parité, les codes cycliques (CRC), la somme de contrôle Internet et les codes de Hamming.

La seconde section fournit un raisonnement permettant de choisir une structure adaptée à la représentation des informations, elle propose également des implémentations possibles en Java pour les différents codes parcourus dans la première section.

Une application a été conçue pour pouvoir tester les différents codes détaillés. Différentes fonctionnalités sont fournies pour pouvoir y parvenir, tels que la possibilité de générer un message aléatoire, corrompre ce message de manière aléatoire selon un modèle d'erreur, encoder un message, décoder un message et enfin de pouvoir générer un graphe montrant l'évolution du taux de détection et/ou de correction des différents codes. Les choix architecturaux ainsi qu'une démonstration de l'application sont fournis dans la troisième section.

2 État de l'art

2.1 Théorie des codes

Le bruit est un signal électrique ou électromagnétique indésirable qui dégrade la qualité des signaux et des données. Le bruit se produit dans les systèmes numériques et analogiques et peut affecter les fichiers et les communications de tous types.

Le bruit va donc altérer certains bits lors de la transmission d'une information via un réseau de télécommunication lors de la lecture/écriture d'une information sur un moyen de stockage. Cette notion de bruit a poussé les scientifiques à développer des algorithmes permettant de coder l'information afin de pouvoir détecter et dans certains cas corriger les erreurs pouvant altérer une information.

Nous allons définir le taux d'erreurs comme étant la probabilité qu'un bit reçu soit différent du bit émis. Ce taux d'erreurs va dépendre du moyen utilisé pour effectuer une communication. La figure 1 reprend les différents ordres de grandeur des taux d'erreurs en fonction du moyen de communication/stockage utilisé.

Ligne	Taux d'erreurs
Disquette	10^{-9} : à 5 Mo/s, 3 bits erronés par minute
CD-ROM optique	10^{-5} : 7 ko erronés sur un CD de 700 Mo
DAT audio	10^{-5} : à 48 kHz, deux erreurs par seconde
Disques Blu-ray	$< 2 \times 10^{-4}$: environ 1.6 Mo erronés pour 32 Go
Hard Disk Drives (HDD)	$> 10^{-4}$: environ 10^6 erreurs pour 10 Go
Technologie Flash	10^{-5}
Solid State Drives (SSD)	$< 10^{-5}$
Mémoires à semi-conducteurs	$< 10^{-9}$
Liaison téléphonique	entre 10^{-4} et 10^{-7}
Télécommande infrarouge	10^{-12}
Communication par fibre optique	10^{-9}
Satellite	10^{-6} (Voyager), 10^{-11} (TDMA)
ADSL	10^{-3} à 10^{-9}
Réseau informatique	10^{-12}

Figure 1: Ordre de grandeur du taux d'erreurs [3].

Shannon introduit en 1948 le théorème du codage de canal [8], permettant de transmettre un message à un destinataire en passant par un canal non fiable. Le message transmis est codé, c'est-à-dire que ce message va contenir de la redondance de telle sorte à permettre la détection et/ou la correction des erreurs.

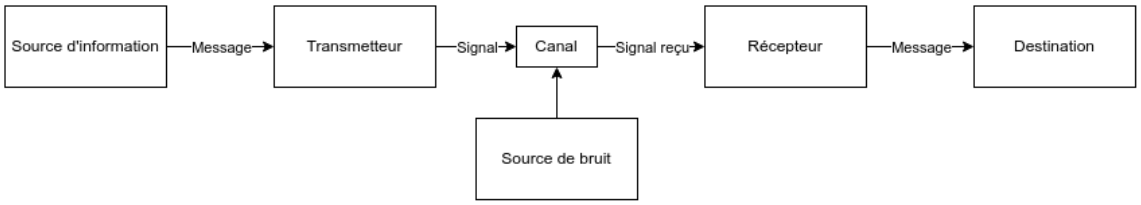


Figure 2: Diagramme de l'émission d'un message dans un système de communication[8].

Nous allons supposer dans la suite de l'article que nous allons passer par un canal binaire, où chaque symbole transmis dans le canal est un bit. L'émetteur du message m transmettra dans le canal des symboles d'un alphabet $V = \{0, 1\}$.

Lors du codage d'un message m, le message est découpé en blocs de taille k. Chaque bloc M_i est ensuite traité individuellement de manière séquentielle.

Exemple de découpage d'un message en bloc à la figure 3.

Bloc	Valeur
M_1	01110100
M_2	11010110
M_3	10011101

Figure 3: Découpage d'un message $m = 011101001101011010011101$ de taille 24 par blocs de taille $k = 8$.

Chaque bloc obtenu après le découpage est codé grâce à une fonction de codage ϕ en ajoutant r symboles de redondance d'informations afin de pouvoir détecter voire corriger les erreurs.

Donc à chaque bloc (appelé mot) de taille k à transmettre $s = [s_1, ..., s_k]$, nous allons transmettre le bloc codé (appelé mot codé) $\phi(s) = [c_1, ..., c_k, ..., c_n]$. Le bloc codé $\phi(s)$ comportera alors $n = k + r$ symboles. L'ensemble $C_\phi = \{\phi(s) : s \in V^k\}$, image par ϕ de V^k, est appelé $code(n, k)$ [3].

Le rendement R d'un $code(n, k)$ est défini comme le taux de bits d'informations par rapport aux bits codés :

$$R = \frac{k}{k+r} = \frac{k}{n}$$

Par exemple, pour un bloc de taille $k = 8$ et un nombre de bits redondants $r = 2$, ce qui nous donne la taille totale du bloc codé $n = 8 + 2 = 10$, nous obtenons un rendement de :

$R = \frac{8}{10} = 0.8$

Un code(n, k) est *systématique* si une partie du mot codé coïncide avec l'entièreté du message. Le code est alors également *séparable* car on peut extraire directement tous les symboles d'informations du mot codé. Souvent, les k symboles d'informations sont placés dans l'ordre au début des mots codés.

Par exemple, fixons un code(8, 5) qui place les bits du message au 3 premières positions et aux 2 dernières positions du mot codé, les bits de redondances vont contenir la somme de tous les bits du message. pour un mot à transmettre $s = [\mathbf{11111}]$, on obtient le mot codé $\phi(s) = [\mathbf{111}10\mathbf{11}]$, la fonction de codage est systématique car l'entièreté du mot à transmettre coïncide avec une partie du mot codé.

Par contre, si ce même code utilisait le complément à un sur le mot codé, nous aurions obtenu $\phi(s) = [00001000]$, alors la fonction de codage n'est pas systématique car elle requiert d'appliquer une transformation sur le mot codé afin de retrouver le mot d'origine.

L'intérêt des codes systématiques est de permettre un codage et un décodage rapide afin de transmettre les informations avec un coût le plus faible possible.

2.1.1 Décoder les messages

Pour pouvoir décoder un message, il est nécessaire que la fonction de codage ϕ soit injective, c'est-à-dire que tout mot code y soit obtenu par le codage de au plus un mot x, tel que $\phi(x) = y$, la figure 4 montre un exemple visuel d'un tel type de fonction.

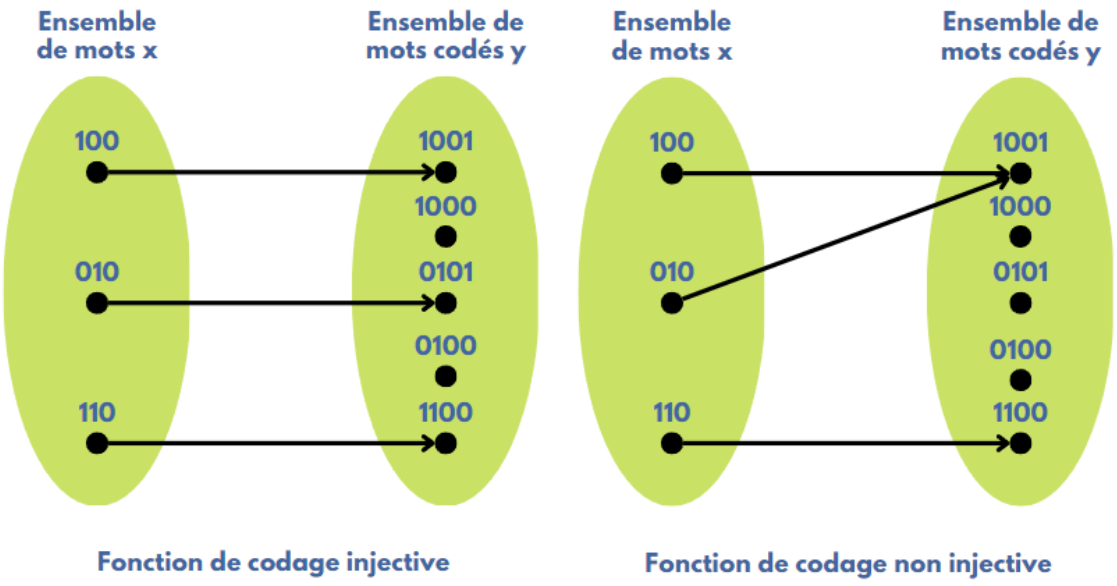


Figure 4: Exemple de fonction de codage injective et non injective.

La détection d'erreurs se base sur le fait que le mot codé reçu $\overline{C_\phi}$ a été altéré par le canal, c'est-à-dire un mot ayant au moins un bit altéré, le mot appartient donc à $\overline{C_\phi} = V^n \setminus C_\phi$. Il existe deux types de corrections :

1. La correction directe : n'est possible uniquement dans le cas où la fonction de codage possède les caractéristiques nécessaires pour corriger un message et le nombre de bits altérés ne doit également pas être trop élevé (ce nombre dépend de la fonction de codage utilisée).

2. La correction par retransmission : s'effectue dans le cas où le mot altéré n'a pas pu être corrigé directement, une requête ARQ (Automatic Repeat Request) est envoyée à l'émetteur du message afin de demander une retransmission du message.

Le choix de la fonction de codage va être influencé par plusieurs facteurs :

1. Le rendement : doit être maximisé afin d'optimiser l'utilisation du réseau et réduire le taux d'erreur.

2. La probabilité de détecter l'erreur : doit être la plus élevée possible afin d'éviter de transmettre des messages absurdes.

3. La possibilité de corriger l'erreur : dans les cas où il n'est pas possible de demander une retransmission de l'information, il va être nécessaire de pouvoir corriger l'erreur.

4. La complexité en temps de l'algorithme : il faut que le codage et le décodage de l'information soit rapide afin de ne brider ni la vitesse de lecture/écriture ni le débit.

5. La complexité en espace de l'algorithme : si l'espace mémoire est limité, il est important de s'assurer que l'algorithme utilisé ne consomme pas une mémoire trop importante.

6. La nature du moyen de transmission ou de stockage des données : les erreurs peuvent survenir de différentes manières.

   - De manière aléatoire (isolée), chaque bit à une probabilité égal d'être altéré.
   - En rafale, pour une séquence contiguë de m bits, le premier et le dernier bit sont erronés. Chaque séquence d'erreurs considérée d'erreurs en rafales est séparée par au moins $m + 1$ bits non erronés.

2.1.2 Probabilité d'erreurs lors de la transmission

Nous allons noter p la probabilité pour chaque bit d'être altéré. Nous faisons l'hypothèse que chaque bit a une même probabilité p d'être altéré. La probabilité qu'il y ait au moins une erreur dans un mot codé va dépendre de sa longueur n, donc le fait de rajouter de la redondance dans le message augmente le risque d'erreur lors de la transmission. Ce qui implique que plus le rendement est faible, plus la probabilité pour qu'il y ait au moins une erreur dans un mot codé est élevée.

Nous faisons l'hypothèse que les erreurs sur les bits sont indépendantes et donc la probabilité qu'un bit à une position donnée soit altéré est toujours identique. Nous pouvons donc calculer la probabilité pour qu'il y ait au moins un bit altéré dans un mot codé avec la formule :

$$p_{error}(n, p) = 1 - (1 - p)^n$$

2.2 Bit de parité

Un code de bit de parité est un code systématique ajoutant un bit de parité à la fin d'un message. Si la somme du nombre de bits valant 1 pour un mot codé est pair, on parle dans ce cas d'un codage à parité paire, sinon il s'agit d'un codage à parité impaire. Par convention, le codage à parité paire est utilisé, dans la suite de cet article, nous utiliserons le codage à parité paire. Lorsque l'on code à parité paire un mot $m = (s_1, ..., s_k)$, on obtient un mode codé $\phi(m) = (s_1, ..., s_k, c_{k+1})$ où $c_{k+1} = (\sum_{i=1}^{k} s_i) \bmod 2$. Ce bit de parité prendra donc la valeur 0 dans le cas où le nombre de bits valant 1 du mot codé est pair et la valeur 1 si ce nombre est impair.

La figure 5 montre un exemple de codage à bit de parité pour des mots de longueur $k = 8$ pour le message $m = 01110100\ 11010110\ 10011101$, une fois codé, nous obtenons $\phi(m) = 011101000\ 110101101\ 100111011$.

Mots M_i du message	Mots codés $\phi(M_i)$
$M_1 = 01110100$	$\phi(M_1) = 011101000$
$M_2 = 11010110$	$\phi(M_2) = 110101101$
$M_3 = 10011101$	$\phi(M_3) = 100111011$

Figure 5: Codage à bit de parité.

Avec cette fonction de codage, nous obtenons toujours un nombre pair de bits valant 1 pour un mot codé. L'erreur est alors détectée si le nombre de bits valant 1 du mot codé est impair. La fonction ne permet alors de détecter l'erreur que lorsque le nombre de bits altérés d'un mot codé est impair. La figure 6 illustre ce comportement.

Mots codés $\phi(M_i)$	Mots codés altérés $\overline{\phi(M_i)}$	Nb erreurs	Erreur détectée?
$\phi(M_1) = 011101000$	$\overline{\phi(M_1)} = 011\mathbf{0}01000$	1	Oui
$\phi(M_2) = 110101101$	$\overline{\phi(M_2)} = 110\mathbf{0}01\mathbf{0}01$	2	Non
$\phi(M_3) = 100111011$	$\overline{\phi(M_3)} = 1\mathbf{1}011\mathbf{0}01\mathbf{0}$	3	Oui
$\phi(M_4) = 011101001$	$\overline{\phi(M_3)} = \mathbf{11}00101\mathbf{1}1$	3	Non

Figure 6: Détection de l'erreur avec le code à bit de parité.

Cette fonction permet uniquement de détecter (dans certains cas) les erreurs mais ne permet pas de les corriger.

2.2.1 Probabilité de détecter l'erreur

Pour détecter une erreur lors de l'utilisation du code à bit de parité, nous devons dans un premier temps calculer la somme des probabilités d'avoir un nombre impair de bits altérés. Il faut donc calculer la probabilité qu'il y ait exactement k bits altérés. Pour calculer cette probabilité, il est nécessaire d'utiliser la loi binomiale.

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

La figure 7 illustre la probabilité que k bits soient altérés lorsqu'un mot est de longueur $n = 16$ et étant donné la probabilité $p = 0.01$ pour chaque bit d'être altéré.

Valeur de k	Probabilité d'erreur
0	0,851457
1	0,137609
2	0,010425
3	0,000491
4	0,000016
...	...

Figure 7: Exemple de calcul de la probabilité d'erreur avec n = 16 et p = 0.01.

Maintenant que nous connaissons la probabilité d'avoir exactement k erreurs, nous pouvons calculer la probabilité de ne pas avoir d'erreur ou de détecter l'erreur :

$$1 - \sum_{i=1}^{\lfloor n/2 \rfloor} P(X = 2i)$$

Pour l'exemple précédent à la figure 7, nous obtenons une probabilité de détecter l'erreur de :
1 - (0,010425 + 0,000016 + ...) = 0,989559
Soit 98.96% de chance de ne pas avoir d'erreur ou alors de la détecter, ce qui nous donne 1.04% de chance de ne pas détecter l'erreur.

La probabilité d'avoir une erreur non détectée est donc trop élevée, ce type de codage est intéressant pour introduire le codage et le décodage de l'information, mais il ne convient pas à une utilisation sur des données réelles.

2.3 Codes cycliques (CRC)

Les codes CRC (Cyclic Redundancy Code) sont des codes systématiques représentant les mots de taille n sous formes d'un polynôme de degré $n-1$. La position k d'un bit dans un mot donné représente un monôme de degré k. Un mot binaire $s = [s_{m-1}...s_0] \in \{0,1\}^m$ est représenté par un polynôme :

$$P_s(X) = \sum_{i=0}^{m-1} u_i X^i$$

Par exemple, pour un mot binaire donné $u = [110101]$, nous obtenons le polynôme $P_s(X) = X^5 + X^4 + 0X^3 + X^2 + 0X + 1 = X^5 + X^4 + X^2 + 1$

2.3.1 Corps fini

Le coefficient de chaque monôme utilisé dans ce code fait partie du corps fini $\mathbb{F}_2$ contenant deux éléments $0, 1$. Les corps fini notés $\mathbb{F}_n$ sont également appelés corps

de Galois notés $GF(n)$. Pour n'importe quel nombre premier, n, il existe un corps fini $\mathbb{F}_n$ qui contient n éléments tels que $\mathbb{F}_n = 0, 1, ..., n-1$. Le corps fini $\mathbb{F}_2$ est le corps possédant le plus petit nombre possible d'éléments. Les opérations sur des corps fini sont implémentés en utilisant des opérations binaires, rendant chaque opération sur ce corps très rapide à être exécutée. L'addition et la soustraction sont le "ou exclusif" (XOR), la multiplication le "et" (AND) et la division le "or" (OR) :

Opération				
Multiplication	$0 \times 0 = 0$	$0 \times 1 = 0$	$1 \times 0 = 0$	$1 \times 1 = 1$
Addition	$0 + 0 = 0$	$0 + 1 = 1$	$1 + 0 = 1$	$1 + 1 = 0$
Soustraction	$0 - 0 = 0$	$0 - 1 = 1$	$1 - 0 = 1$	$1 - 1 = 0$
Division	/	$0/1 = 1$	/	$1/1 = 1$

La division d'un élément par un autre valant 0 est non définie dans cette structure.

Théorème de la division euclidienne des polynômes dans $\mathbb{F}_2$: soient $P_1(X)$ et $P_2(X)$ deux polynômes à coefficients dans $\mathbb{F}_2$, avec $P_2(X)$ non nul, il existe un unique couple (Q, R) tel que $P_1(X) = P_2(X) \times Q + R$ et le degré de R est strictement plus petit que celui de $P_2(X)$. Q est appelé le quotient et R le reste.

Un polynôme $P_1(X)$ est divisible par un second polynôme $P_2(X)$ si le degré du premier est supérieur ou égal au degré du second.

Exemple de division de d'un polynôme par un autre :

						Diviseur
$+X^7$	$+X^5$	$+X^4$	$+X^3$		$+1$	$X^3 + X + 1$
$+X^7$	$+X^5$	$+X^4$				$X^4 + 1$
			$+X^3$		$+1$	**Quotient**
			$+X^3$	$+X$	$+1$	
				$+X$	**Reste**	

Le code CRC se base sur le reste de la division euclidienne d'un polynôme par un autre. Le premier polynôme P_s représente le mot source binaire $s = [s_{k-1}...s_0]$, le second représente un *polynôme générateur* P_g de degré r :

$$P_g(X) = X^r + \sum_{i=0}^{r-1} g_i X^i$$

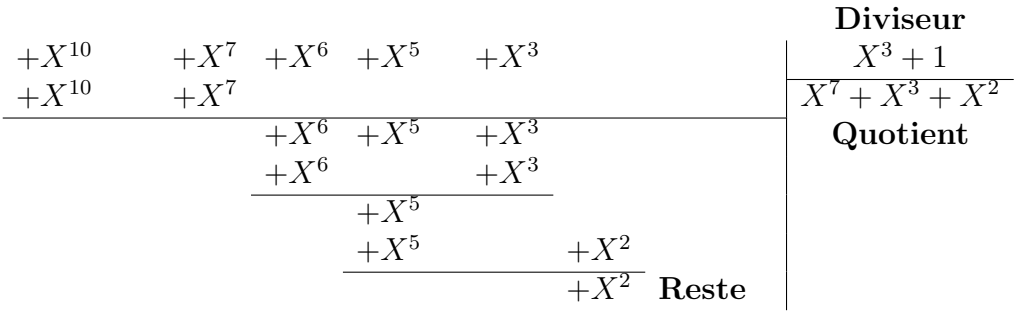
Le mot s est codé par le mot de code binaire $c = [c_{n-1}...c_0] = [s_{k-1}...s_0 c_{r-1}...c_0]$ où $[c_{r-1}...c_0]$ est la représentation du reste de la division euclidienne de $X^r.P_s$ par P_g [3], nous obtenons un polynôme vérifiant :

$$P_c = P_s.X^r + (P_s.X^r mod P_g)$$

Le mot codé est donc un multiple du polynôme générateur P_g. Il est alors nécessaire pour l'émetteur et le récepteur du message de définir le polynôme générateur utilisé. Le récepteur détectera l'erreur si le polynôme reçu P'_c est divisible par P_g sans qu'il n'y ait de reste de la division.

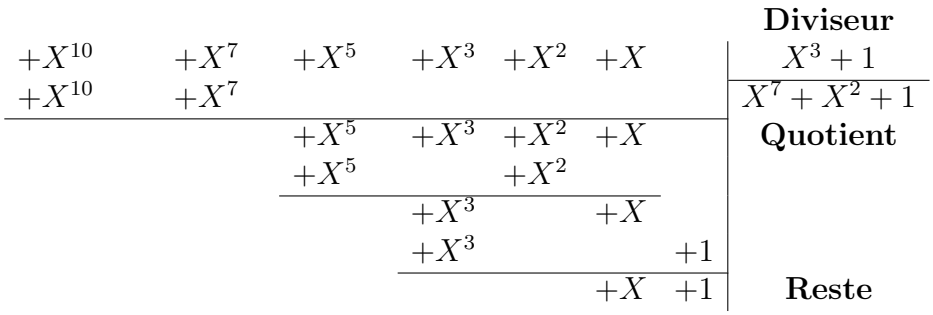
Exemple de codage pour le message 10011101 et un polynôme générateur donné $P_g = X^3 + 1$ de degré $r = 3$:

Le message source $s = 10011101$ est équivalent au polynôme $P_s = X^7 + X^4 + X^3 + X^2 + 1$, $P_s.X^r = (X^7 + X^4 + X^3 + X^2 + 1) * X^3 = X^{10} + X^7 + X^6 + X^5 + X^3$, nous pouvons à présent calculer le reste $(P_s.X^r mod P_g)$:



Nous obtenons $[c_{r-1}...c_0] = [100]$, le message codé vaut alors $c = [10011101100]$.

Exemple de détection d'erreur pour le message codé de l'exemple précédent 10011101100. Le message après transmission a été altéré comme suit $P_e(X) = X^6 + X$, $c' = 1001\mathbf{0}10111\mathbf{1}0$, $P_c(X)' = X^{10} + X^7 + X^5 + X^3 + X^2 + X$. Calculons le reste $(P_c(X)' mod P_g)$:



Le reste différent de zéro indique une erreur.

En plus des propriétés intéressantes des CRC que nous verrons par la suite, le codage et le décodage d'un message est très rapide, ce qui rend ces codes particulièrement intéressants à haut débit. Les CRC sont des codes systématiques, ce qui explique pourquoi l'extraction du message dans le mot codé est rapide.

2.3.2 Propriétés des générateurs

Si le polynôme générateur possède plus d'un coefficient non nul, alors le code peut détecter toutes les erreurs portant sur un seul bit, car le reste de la division euclidienne entre $P_c(X)'$ et $P_g(X)$ sur un bit d'erreur unique ne peut pas être nul. Réciproquement si le polynôme générateur ne possède qu'un seul coefficient non nul $P_g(X) = X^r$, il ne détecte pas l'erreur $P_e(X) = X^{r+i}$ où $i \geq 0$.

Tous les polynômes générateurs ayant la forme $P_g(X) = X^r + \sum_{i=1}^{r-1} g_i X^i + 1$ peuvent détecter une erreur portant sur $r-1$ bits consécutifs. Par exemple, le polynôme générateur $X^8 + X^7 + X^5 + 1$ peut détecter $8 - 1 = 7$ bits altérés consécutifs. Une séquence d'erreurs portant sur au plus $r-1$ bits consécutifs commençant au $i^{ième}$ bit du mot est associée à un polynôme $P_e = X^i Q$ avec Q de degré strictement inférieur à r, Q n'est donc pas un multiple de P_g. Et comme $g_0 = 1$, P_g est premier avec X [3]. Cette propriété est particulièrement intéressant, elle nous permet de comprendre pourquoi tous les CRC existant possèdent le monôme +1 dans leur polynôme générateur, elle permet de maximiser le nombre d'erreurs consécutives détectables, ils peuvent donc tous détecter $r-1$ erreurs consécutives. Elle justifie également l'utilisation des CRC dans le cas où les erreurs peuvent survenir en rafales.

Si le polynôme générateur possède $(X+1)$ comme l'un de ses facteurs, alors le polynôme du mot codé (qui est un multiple de ce polynôme générateur P_g) a également $(X+1)$ comme un de ses facteurs. Tous les polynômes ayant $(X+1)$ comme un de leurs facteurs possèdent un nombre pair de coefficients. C'est démontré en considérant le polynôme $v(X) = (1+X)w(X)$. En substituant $X = 1$, on obtient $v(1) = (1+1)w(1) = 0$, impliquant le fait que $v(X)$ a un nombre pair de coefficients. Donc si le polynôme générateur possède $(X+1)$ comme l'un de ses facteurs, alors tous les mots codés ont un nombre de coefficients non nul pair et toutes les erreurs portant sur un nombre impair de bits sont détectées [5].

Par exemple, pour le polynôme générateur $P_g(X) = X^6 + X^4 + X + 1 = (X+1)(X^5 + X^4 + 1)$ et le polynôme d'un message source $P_s(X) = X^9 + X^5 + X^4 + X^2$, on obtient le polynôme codé $P_c = X^{15} + X^{11} + X^{10} + X^8 + X^4 + X^3$. Ce polynôme contient 6 coefficients non nuls.

Considérons maintenant un modèle à double erreurs $P_e(X) = X^i + X^j = X^i(1+X^{j-i}$ pour un i donné, $0 \leq i \leq n-2$ et pour un j donné, $i+1 \leq j \leq n-1$. Si $P_g(X)$ ne possède pas X comme un de ses facteurs et si ce polynôme ne divise pas de

manière égale $[1 + X^{j-i}]$ avec $1 \leq j - 1 \leq n - 1$, alors le polynôme générateur détecte tous les modèles à double erreurs [5]. Par exemple, le polynôme générateur $P_g(X) = X^4 + X^2 + X + 1$ possède le monôme $+1$, donc P_g ne possède donc pas X comme un de ses facteurs.

Le choix d'un bon polynôme générateur est donc essentiel afin de maximiser le taux de détection des erreurs. Baicheva et al. [1] montrent que de nombreux polynômes générateurs standardisés offrent soit une capacité de détection d'erreurs inférieure à celle d'autres polynômes générateurs, soit une bonne détection d'erreur pour des longueurs de message particulières.

2.4 Somme de contrôle Internet (RFC 1071)

La somme de contrôle Internet est un protocole utilisé dans les protocoles TCP/UDP/IP pour détecter les erreurs. Le principe est d'ajouter un certain nombre de bits qui représente le complément à un de la somme des différents mots d'un paquet. Il a été conçu pour avoir un bon équilibre entre le coût de calcul et la chance de détecter une erreur avec succès.

La somme de contrôle Internet se base donc sur le complément à un de la somme des mots du segment. Le complément à un est obtenu en inversant tous les bits d'un nombre. Par exemple, pour un mot binaire donné $s = [11010100]$, on obtient le complément à un [00101011].

Lors de l'addition des mots du segment, le but est de garder la somme totale sur 16 bits, donc à chaque fois que l'addition de 2 mots donne un résultat sur 17 bits, le bit de poids le plus fort est reporté et additionné au bit de poids le plus faible. Par exemple, pour l'addition de deux mots binaires sur 4 bits, avec $s_1 = [1001]$, $s_2 = [1010]$, on obtient $s_1 + s_2 = [0100]$.

$$\begin{array}{cccccc} & & 1 & 0 & 0 & 1 \\ + & & 1 & 0 & 1 & 0 \\ \hline & \mathbf{1} & 0 & 0 & 1 & 1 \\ & & & & & \mathbf{1} \\ \hline & & 0 & 1 & 0 & 0 \end{array}$$

Lorsque le destinataire reçoit le message avec la somme de contrôle, il va à son tour calculer la somme de tous les mots du segment, une fois cette somme obtenue, il va l'additionner à la somme de contrôle, si un des bits de cette addition est égale à 0, alors le message a été corrompu lors du transfert.

Exemple de calcul de somme internet cs pour un paquet donné de 48 bits :

$m = 1100111010101100\ 0000100011011100\ 1011010101001111$

$M_1 = 1100111010101100$
$M_2 = 0000100011011100$
$M_3 = 1011010101001111$

m_1			1	1	0	0	1	1	1	0	1	0	1	0	1	1	0	0
m_2	+		0	0	0	0	1	0	0	0	1	1	0	1	1	1	0	0
			1	1	0	1	0	1	1	1	1	0	0	0	1	0	0	0
m_3	+		1	0	1	1	0	1	0	1	0	1	0	0	1	1	1	1
		1	1	0	0	0	1	1	0	0	1	1	0	1	0	1	1	1
																		1
$m_1 + m_2 + m_3$			1	0	0	0	1	1	0	0	1	1	0	1	**1**	**0**	**0**	**0**
`Complément à un`			0	1	1	1	0	0	1	1	0	0	1	0	0	1	1	1

La somme de contrôle internet vaut donc 0111001100100111, l'expéditeur du message m va donc rajouter cette somme à la fin du message. Le récepteur reçoit le message et calcule si celui-ci est corrompu :

m_1			1	1	0	0	1	1	1	0	1	0	1	0	1	1	0	0
m_2	+		0	0	0	0	1	0	0	0	1	1	0	1	1	1	0	0
			1	1	0	1	0	1	1	1	1	0	0	0	1	0	0	0
m_3	+		1	0	1	1	0	1	0	1	0	1	0	0	1	1	1	1
		1	1	0	0	0	1	1	0	0	1	1	0	1	0	1	1	1
																		1
$m_1 + m_2 + m_3$			1	0	0	0	1	1	0	0	1	1	0	1	1	0	0	0
cs	+		0	1	1	1	0	0	1	1	0	0	1	0	0	1	1	1
			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Tous les bits de la somme entre $(m_1 + m_2 + m_3) + cs$ valent 1, le message n'est donc pas corrompu.

Reprenons l'exemple précédent et altérons quelques bits lors de l'envoi du message afin d'illustrer l'erreur, le destinataire reçoit le message et le découpe en mots de 16 bits :

$m_1 = 1100101010101100$
$m_2 = 0000100011011000$
$m_3 = 1011010101001111$
$cs = 0111001100100111$

m_1			1	1	0	0	1	0	1	0	1	0	1	0	1	1	0	0
m_2	+		0	0	0	0	1	0	0	0	1	1	0	1	1	0	0	0
			1	1	0	1	0	0	1	1	1	0	0	0	0	1	0	0
m_3	+		1	0	1	1	0	1	0	1	0	1	0	0	1	1	1	1
		1	1	0	0	0	1	0	0	0	1	1	0	1	0	0	1	1
																		1
$m_1 + m_2 + m_3$			1	0	0	0	1	0	0	0	1	1	0	1	0	1	0	0
cs	+		0	1	1	1	0	0	1	1	0	0	1	0	0	1	1	1
			1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1

Au moins un des bits de la somme entre $(m_1 + m_2 + m_3) + cs$ vaut 0, le message a donc été corrompu.

Le calcul de la somme de contrôle possède plusieurs propriétés mathématiques intéressantes, qui peuvent être exploitées afin d'accélérer le codage et la vérification d'erreur [2].

- **L'associativité et la commutativité** : tant que le message est bien découpé en mots de 16 bits, l'addition de chaque mot peut se faire dans n'importe quel ordre et il peut être divisé en groupe. Par exemple pour l'addition de $m_1 + m_2 + m_3 + m_4$, on pourrait l'effectuer dans l'ordre suivant : $m_1 + (m_3 + m_2) + m_4$.

- **L'addition en parallèle** : si la machine possède des mots dont la taille est un multiple de 16, il est possible d'accélérer le codage en codant en parallèle les mots. Dû au fait que l'addition est associative, au lieu d'additionner les mots un par un, nous allons utiliser l'entièreté de la taille du mot de la machine, en faisant les additions en parallèle. Il faut faire attention que l'addition de chaque mot peut provoquer un bit de débordement et que celui-ci ne doit pas être ajouté à un autre mot, mais doit être ajouté au bit de poids le plus faible de ce mot.

  Exemple d'addition en parallèle sur une machine avec une architecture 32 bits, le message à envoyer est découpé en 4 mots $m_1 + m_2 + m_3 + m_4$, les bits des mots m_1 et m_2 sont mis dans une variable commune de 32 bits et additionnés avec la variable contenant m_3 et m_4. Une fois que tous les mots ont été additionnés, il suffit d'additionner les 16 premiers bits de la variable contenant le résultat avec les 16 derniers bits.

- **Actualisation incrémentielle** : si la somme de contrôle a déjà été calculée, qu'un des mots a changé et qu'il est nécessaire de recalculer cette somme. Pour calculer cette nouvelle somme de contrôle C', il suffit simplement de rajouter à l'ancienne somme de contrôle C la différence entre le nouveau mot m' et le mot qui a changé m. Ce phénomène s'explique par le fait que l'addition est associative.

$$C' = C + (-m) + m' = C + (m' - m)$$

Une implémentation efficace est primordiale pour avoir de bonnes performances.

2.4.1 La détection d'erreurs dans la somme de contrôle internet

La somme de contrôle internet se base donc sur la somme des différents mots du message, pour ne pas détecter l'erreur, il suffit que la somme de chaque colonne reste inchangée. Par exemple, pour un message donné, il suffit que 2 bits à la même position d'un mot différent (dont les valeurs étaient respectivement 0 et 1 ou 1 et 0) soient inversés :
$m_1 = 0100111010101100$
$m_2 = 1000100011011100$
$m_3 = 1011010101001111$
$cs = 0111001100100111$
$m_1 + m_2 + m_3 + cs = 1111111111111111$

La somme de contrôle va permettre de détecter toutes les erreurs en rafale portant sur au plus 15 bits [2]. Sur des données uniformément distribuées, il détecte d'autres types d'erreurs à un taux proportionnel à 1 sur 2^{16}. Sur des données non uniformément distribuées, ses performances peuvent être nettement inférieures. Une étude montre que l'utilisation de cette somme de contrôle sur des données réelles est comparable à des données uniformément distribuées avec une somme de contrôle sur 10 bits [12]. Cela signifie qu'au lieu d'avoir une chance de détecter les autres types d'erreurs avec un taux proportionnel à 1 sur 2^{16}, on aurait plutôt un taux de 1 sur 2^{10}.

Avant de tester la somme de contrôle d'un paquet, à la couche liaison, les CRC vont s'occuper de détecter les erreurs. Si le CRC utilisé ne détecte aucune erreur, cette somme va tester à son tour si une erreur est présente. Une étude de Stone et al. [11] montre qu'entre un paquet sur quelques millions et un paquet sur 10 milliards contiendra une erreur non détectée par cette somme.

Une grande limitation de cette somme de contrôle est le fait que la somme d'un ensemble de valeurs de 16 bits est la même, quel que soit l'ordre dans lequel les additions sont effectuées (commutativité). Donc si pour une certaine raison, deux mots sont inversés dans le message, l'erreur ne sera pas détectée.

2.5 Codes de Hamming

Les codes de Hamming binaires sont des codes systématiques de corrections d'erreurs qui sont utilisés pour détecter et corriger les erreurs dans les données transmises. Ils utilisent des bits de parité supplémentaires pour ajouter de la redondance à la

transmission de données pour permettre la correction d'une erreur éventuelle. De même que pour les codes de bit de parité, les codes de Hamming peuvent avoir un codage à parité paire ou impaire, nous utiliserons dans cet article, le codage à parité paire pour les codes de Hamming. La longueur du code de Hamming dépend du nombre de bits de données et de parité dans le message.

Ces codes sont parfaits et 1-correcteur. C'est-à-dire qu'il n'existe pas d'autre code plus compact ayant la capacité de corriger une erreur, il n'y a donc aucune redondance inutile dans un mot codé. Ils sont cependant capables de détecter l'erreur portant sur plusieurs bits erronés d'un mot codé. Néanmoins, en cas d'erreurs portant sur plusieurs bits, ils ne pourront pas corriger correctement ce mot codé.

Le nombre de bits de redondances r ajouté à un mot dépend du nombre de bits de données k du mot à coder, le mot codé est de longueur n. Pour un k donné, nous avons :

$$\begin{aligned} 2^r &\geq n+1 \\ r &\geq \log_2(n+1) \end{aligned} \tag{1}$$

Nous savons que le rendement d'un code est obtenu avec $\frac{k}{k+r}$, donc pour avoir le code le plus compact possible, il faut maximiser k. Pour obtenir un code parfait (en d'autres termes, avec un k maximisé), il faut respecter l'égalité de l'équation précédente. Ce qui nous donne $r = \log_2(n+1)$. Il n'est en conséquence pas possible d'avoir un code parfait pour n'importe quelle valeur de k ou n.

2.5.1 Codage

Un mot codé $c = [c_n, ..., c_1]$ avec un code de Hamming est tel que tous les bits c_i dont l'indice i est une puissance de 2 (2^l avec $l = 0, 1, 2, ...$) sont les bits de redondances et tous les autres bits sont les bits de données. Pour obtenir la valeur d'un bit de redondance, il est nécessaire d'utiliser la représentation binaire de tous les bits de données valant 1. Le bit de redondance placé à l'indice $i = 2^l$ du mot codé est obtenu avec la somme modulo 2 de tous les bits de poids i des représentations binaires des positions des bits de données valant 1.

Remarque, pour obtenir le bit de redondance, il est par conséquent essentiel de préplacer les bits de données afin d'avoir leur position finale dans le mot codé.

Exemple de codage pour le mot 1001. Ce mot peut être codé en utilisant le code de Hamming parfait (7, 4). Nous avons besoin d'ajouter à ce mot les 3 bits de redondances à chaque position 2^l (avec $l = 0, 1, 2$ dans ce cas-ci), mais ces bits dépendent de la position finale de chaque bit de donnée dans le mot codé. On sait donc que le mot codé aura la forme : 100_1__

Calculons à présent la valeur des bits de redondances à la figure 8, nous avons besoin pour cela de la représentation binaire des positions de chaque bits du message valant 1 :

Position \ Poids	4	2	1
3	0	1	1
7	1	1	1
Bit de parité	1	0	0

Figure 8: Calcul des bits de redondance pour le mot 1001.

Le mot codé vaut donc 1001100.

2.5.2 Décodage

Le principe du décodage est le même pour le codage,
Il est nécessaire de regarder la représentation binaire de chaque position de chaque bit valant 1 dans le mot codé. Tous les bits de redondances d'indice $i = 2^l$ sont vérifiés. Une erreur est détectée si l'un de ces bits de redondances est erroné (parité non respectée), donc si la somme modulo 2 de tous les $(l+1)^{ième}$ bits (en partant de la droite) des représentations binaires des positions des bits du mot codé valant 1. Si la somme vaut 0 pour chaque $l = 0, 1, 2$, alors aucune erreur n'est détectée. Par contre si cette somme vaut 1, alors une erreur est détectée et une correction est nécessaire. Le bit erroné se trouvera à la somme des indices des bits de redondances i qui sont erronés.

Exemple de décodage pour le mot codé 1001100 dans la section 2.5.1. Pour rappel, le mot a été codé avec le code de Hamming (7, 4).

Calculons à présent la valeur des bits de redondances à la figure 9, nous avons besoin pour cela de la représentation binaire des positions de chaque bits du message valant 1 :

Position \ Poids	4	2	1
3	0	1	1
4	1	0	0
7	1	1	1
Bit de parité	0	0	0

Figure 9: Calcul des bits de redondance pour le mot codé 1001100.

Aucune erreur n'a été détectée, on peut donc extraire directement le mot en retirant tous les bits aux positions 2^l (avec $l = 0, 1, 2$ dans ce cas-ci). Le mot vaut donc 1001.

Exemple de décodage portant sur un bit erroné, reprenons le mot codé 1001100 dans la section 2.5.1, modifions un bit pour corrompre le mot codé 1011100. Pour rappel, le mot a été codé avec le code de Hamming (7, 4).

Calculons à présent la valeur des bits de redondances à la figure 10, nous avons besoin pour cela de la représentation binaire des positions de chaque bits du message valant 1.

Position \ Poids	4	2	1
3	0	1	1
4	1	0	0
5	1	0	1
7	1	1	1
Bit de parité	1	0	1

Figure 10: Calcul des bits de redondance pour le mot codé 1011100.

Une erreur est détectée, nous allons donc procéder à la correction de cette erreur. La somme des indices des bits de redondances qui sont erronés vaut $1 + 4 = 5$, nous inversons donc le 5ème bit du mot codé et nous obtenons 1001100. Nous pouvons à présent extraire les bits de données pour retrouver le message original. Une fois ces bits extraits, nous obtenons le message 1001.

Nous savons que les codes de Hamming sont 1-correcteur, faisons un exemple de décodage portant sur deux bits erronés pour voir le comportement de ce code. Reprenons le mot codé 1001100 dans la section 2.5.1, modifions deux bits pour corrompre le mot codé 1011000. Pour rappel, le mot a été codé avec le code de Hamming (7, 4).

Calculons à présent la valeur des bits de redondances à la figure 11, nous avons besoin pour cela de la représentation binaire des positions de chaque bits du message valant 1 :

Position \ Poids	4	2	1
4	1	0	0
5	1	0	1
7	1	1	1
Bit de parité	1	1	0

Figure 11: Calcul des bits de redondance pour le mot codé 1011000.

L'erreur est bien détectée dans ce cas-ci, corrigeons le message. La somme des indices des bits de redondances qui sont erronés vaut $2+4=6$, nous inversons donc le 6ème bit du mot codé et nous obtenons 1111000. On constate que nous avons modifié un bit qui n'était pas erroné, nous nous retrouvons avec 3 bits erronés au lieu de 2. Après l'extraction des bits de redondances, nous obtenons le mot 1110.

2.5.3 Distance de Hamming

Le poids de Hamming de $x = (x_1, ..., x_n) \in V^n$, noté $w(x)$, représente le nombre d'éléments non nuls de x :

$$w(x) = |\{i \in \{1, ..., n\}/x_i \neq 0\}$$

Par exemple, pour $x = 10110101$, il suffit de compter le nombre de bit valant 1 : $w(x) = 5$.

La distance de Hamming, notée $d_H(x, y)$, entre x et y correspond au nombre d'éléments pour lesquelles x et y diffèrent.

$$d_H(x, y) = |\{i \in \{1, ..., n\}/x_i \neq y\}$$

Par exemple, pour $x = 10110101$ et $y = 10101100$, $d_H(x, y) = 3$.

x	10110101
y	10101100

2.5.4 Distance minimale de Hamming

La distance minimale de Hamming pour un code C donné, notée $\delta(C)$, est obtenue en calculant la distance de Hamming minimale entre deux mots codés quelconques c_1 et c_2 différents :

$$\delta(C) = \min_{c_1, c_2 \in C; c_1 \neq c_2} d_H(c_1, c_2)$$

Par exemple, si nous calculons la distance minimale de Hamming du bit de parité pour $k = 2$, nous générons les mots codés de longueur $n = 3$ et nous pouvons ensuite calculer une liste de distance de Hamming entre chaque mot codé. Une fois cette liste calculée, $\delta(C)$ sera alors égal au minimum de cette liste.

m	C(m)
00	000
01	011
10	101
11	110

Dans cet exemple, il est évident que $\delta(C) = 2$.

2.5.5 La détection d'erreurs

Pour une distance de Hamming (d_{min}) donnée, la capacité de détection d'erreurs pour un code est de $\frac{d_{min}-1}{2}$. Les codes de Hamming étant 1-correcteur, ils ne sont pas efficaces contre les erreurs en rafale.

3 Mise en oeuvre des codes

3.1 Choix du langage utilisé

Le langage choisi pour le projet est Java. Il s'agit d'un choix personnel, par préférence. Le but du projet n'est pas d'avoir une exécution la plus rapide possible. Le projet a été développé avec la version 19 de Java. L'avantage est que le Java est un langage connu et utilisé par beaucoup de développeurs. Il existe beaucoup de librairies et d'implémentations de différents codes détecteurs et correcteurs d'erreurs.

3.2 Fonctionnalités

3.2.1 Encodage

Cette fonctionnalité permet d'encoder un message donné pour pouvoir détecter ou corriger des erreurs qui pourraient survenir en raison d'une corruption du message. Cette méthode prend en entrée le message à encoder ainsi que la longueur du message avant l'encodage.

3.2.2 Décodage

Cette fonctionnalité permet de décoder un message s'il n'a pas été corrompu ou si l'erreur a pu être corrigée. Cette méthode prend en entrée le message à décoder ainsi que la longueur du message encodé.

3.2.3 Génération de message aléatoire

Cette fonctionnalité génère un message aléatoire de la longueur spécifiée.

3.2.4 Corruption d'un message

Cette fonctionnalité introduit des erreurs dans le message en fonction du modèle d'erreur. L'utilisateur fournit le message, la probabilité qu'un bit soit corrompu, ainsi que le modèle d'erreur à utiliser.

3.2.5 Génération d'un graphique

Cette fonctionnalité génère un graphique montrant l'évolution du taux de détection/correction d'un code en fonction de l'évolution du taux d'erreur sur un canal. L'utilisateur fournit plusieurs paramètres, notamment le nombre d'itérations, le taux d'erreur, la taille du message, le code à utiliser ainsi que le modèle d'erreur. Le graphique montre une première courbe indiquant le taux de détection des erreurs et une seconde courbe indiquant le taux de correction des erreurs, uniquement si le code peut corriger les erreurs.

3.3 Choix de l'architecture

L'application est conçue selon une architecture modulaire, avec trois packages principaux : `command`, `math` et `service`. Le package `command` sert d'interface entre l'utilisateur et l'application, il contient toutes les classes nécessaires pour appeler les différentes fonctionnalités. Le package `math` contient la structure de données pour stocker et effectuer des opérations binaires sur un entier représentant un message, tandis que le package service implémente les fonctionnalités.

L'application utilise la librairie `apache common-cli` pour analyser les options de ligne de commande transmises par l'utilisateur et pour imprimer des messages d'aide détaillant les options disponibles. Les options dépendent de la fonctionnalité à utiliser et sont initialisées en analysant les arguments fournis par l'utilisateur.

Le coeur de l'application repose sur trois interfaces principales : `Code`, `ErrorChannelModel` et `Command`. L'interface `Code` contient les déclarations de toutes les méthodes abstraites nécessaires au codage et au décodage des messages, tandis que l'interface `ErrorChannelModel` contient une méthode abstraite `corrupt` qui permet de corrompre un message pour une probabilité donnée pour chaque bit d'être altéré. L'interface `Command` est une interface générale qui permet d'exécuter n'importe quelle fonctionnalité en recevant en paramètre un objet de type `CommandLineParameter`.

La classe `CommandLineParameter` est utilisée pour stocker tous les paramètres nécessaires aux différentes fonctionnalités, y compris l'initialisation des interfaces `Code` et `ErrorChannelModel`. Les classes implémentant l'interface `Command` peuvent ainsi simplement faire appel aux interfaces stockées dans l'objet `CommandLineParameter`.

La classe `CommandLineOption` contient la définition de toutes les options différentes du programme, ainsi que plusieurs listes de toutes les options pour chaque fonctionnalité différente.

L'application utilise également trois énumérations : `DetectingCode`, `ErrorChannelModel` et `MainCommand`. Ces énumérations sont associées respectivement aux interfaces `Code`, `ErrorChannelModel` et `Command` et permettent de stocker le nom des différents arguments.

Pour ajouter une nouvelle fonctionnalité à l'application, il suffit d'ajouter une classe qui implémente l'interface `Command`, une énumération dans l'énumération `MainCommand` et un case dans le `switch case` de la classe `Main`. Pour ajouter un nouveau code ou un modèle d'erreur sur un canal, il faut implémenter l'interface correspondante, ajouter une énumération dans l'énumération correspondante et un case dans le `switch case` correspondant de la classe `CommandLineParameter` lors de l'initialisation des paramètres de l'application.

La librairie standard pour générer des nombres aléatoires est la classe `Random`, il existe cependant des librairies proposant de meilleures performances, ainsi qu'une meilleure distribution des données. La classe `SplittableRandom` a été utilisée pour générer les nombres aléatoires du projet, elle permet de générer des nombres aléatoires presque deux fois plus rapidement que la classe `Random` et présente une meilleure distribution statistique des données. Il existe d'autres librairies proposant une légère amélioration au niveau des performances par rapport au `SplittableRandom`, cependant la classe `SplittableRandom` a été choisie car elle est fournie avec Java depuis la version 8.

La librairie `JMathPlot` [6] a été utilisée pour générer les graphes, cette librairie propose différentes classes permettant de représenter les données dans des graphiques à 2 ou 3 dimensions de différentes manières. Pour ce projet, nous utiliserons la classe `Plot2DPanel` afin d'afficher une ou plusieurs lignes polygonales.

La figure 12 montre l'architecture de haut niveau de l'application, l'utilisateur interagit avec l'application via une commande Shell, ce qui va lancer la classe `Main`, celle-ci va s'occuper de faire appel à l'interface `Command` qui se trouve dans le package `Command` afin d'exécuter la fonctionnalité souhaité par l'utilisateur, cette interface va elle-même faire appel à une interface ou une classe du package `Service` qui contient le coeur des fonctionnalités.

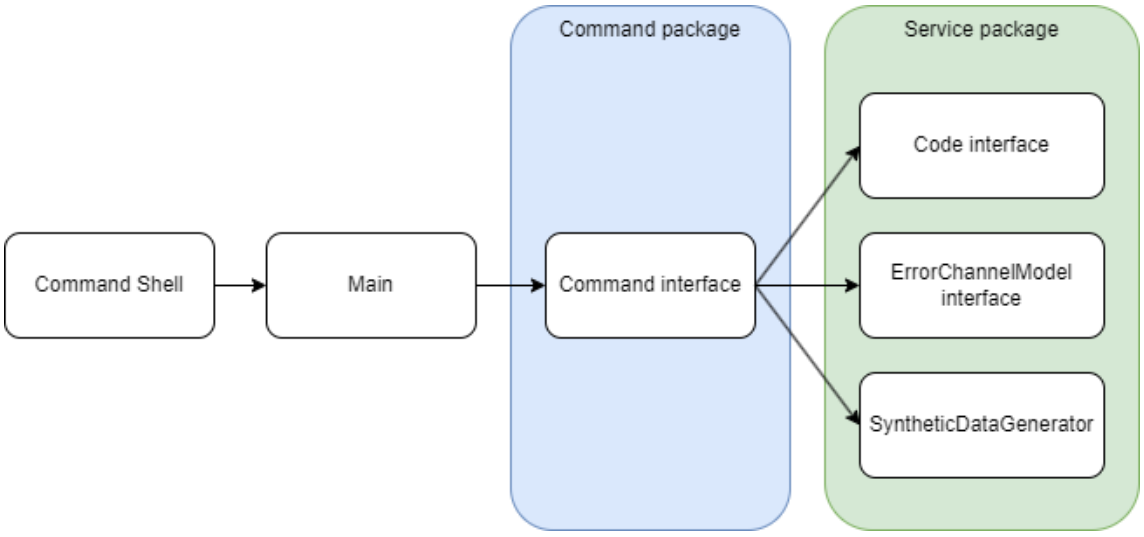


Figure 12: Architecture de haut niveau de l'application.

3.4 Choix de la structure utilisée

Le choix d'une structure de données est primordial afin de pouvoir construire un algorithme permettant de détecter/corriger les erreurs dans un message. Il faut

s'assurer que cette structure de données permette de contenir un nombre arbitraire de bits, il n'est donc pas possible d'utiliser les structures de base, puisque la plus grande structure pouvant contenir un nombre est la structure `long`. Celle-ci ne peut contenir que 64 bits. Il existe d'autres structures permettant de représenter un nombre de taille arbitraire, cependant aucune structure fournie par Java ne permet d'avoir une structure mutable représentant ce genre de nombre.

De nos jours, le hardware permet de stocker ou de transmettre l'ordre du gigabit de données par seconde, il faut par conséquent que cette structure possède des opérations binaires optimisées afin d'avoir un temps de réponse adéquat. Le but de ce travail n'est pas d'atteindre cet objectif mais d'avoir un temps d'exécution raisonnable.

Nous allons considérer l'utilisation de 3 structures de données :

- **StringBuilder** : est une séquence mutable de caractères. Cette structure peut être vue comme un `String` mais dont son contenu peut être modifié. Nous allons l'utiliser en utilisant uniquement les caractères '0' et '1' afin de représenter des messages binaires. L'inconvénient de cette structure est qu'il faut implémenter soi-même les opérations binaires nécessaires aux différentes fonctions de codage.
- **BigInteger** : est un tableau d'entier immuable qui représente un nombre de taille arbitraire. L'avantage est que cette structure possède déjà des opérations binaires optimisées. Cependant, le fait qu'elle soit immuable rend les performances de son utilisation médiocre lorsqu'il est nécessaire de faire appel aux opérations binaires modifiant des bits.
- **BigInt** [4] : est un tableau d'entier immuable qui représente un nombre de taille arbitraire. L'avantage est que cette structure possède déjà des opérations binaires optimisées et qu'elle est mutable.

3.4.1 Métriques considérées

Lors de la comparaison des structures de données décrites précédemment, nous allons utiliser les métriques suivantes :

- **Efficacité en temps** : mesure le temps d'exécution d'une opération, dans notre cas, nous allons mesurer le temps d'exécution des différentes opérations binaires nécessaire à la réalisation des algorithmes permettant d'encoder et de décoder des messages.
- **Efficacité en mémoire** : mesure l'espace occupé par une structure de données, nous allons mesurer en détail l'espace mémoire occupé par les différentes structures considérées.

Le choix de la structure de données va donc se porter sur la structure proposant un compromis entre un temps de réponse faible des opérations binaires, ainsi que l'occupation la plus réduite possible de l'espace en mémoire.

3.4.2 Opérations binaires nécessaires

Voici la liste des opérations binaires qui ont été utilisées lors de l'écriture des algorithmes permettant d'encoder et de décoder des messages pour le code à bit de parité, les codes cycliques (CRC), somme de contrôle Internet (RFC 1071) et pour les codes de Hamming :

- **Addition de nombres binaires** : ajout d'un nombre binaire à un autre nombre binaire.
- **ET binaire** : renvoie un résultat où chaque bit vaut 1 si les bits à cette même position dans les deux opérandes valent 1.

&	0	1
0	0	0
1	0	1

- **Comptage de bits** : compte le nombre de bits valant 1 pour un entier.
- **Test de bit** : teste si un bit à un index donné vaut 1.
- **Insertion d'un bit** : insère un bit à une position donnée, la valeur du bit inséré est fourni en paramètre. Par exemple, pour l'entier binaire 1101, si l'on ajoute le bit '1' à cet entier à l'index $i = 2$, on obtient 11011.
- **Bit de poids fort** : renvoie l'index du bit ayant le poids le plus fort d'un entier. Exemple, pour l'entier binaire 01100101, l'index du bit du poids le plus fort vaut 7.

Index	8	7	6	5	4	3	2	1
Valeur entier	0	1	1	0	0	1	0	1

- **NON binaire** : inverse tous les bits d'un entier. Par exemple, en appliquant un NON binaire sur l'entier 01100101, nous obtenons 10011010.

Entier	0	1	1	0	0	1	0	1
NON(Entier)	1	0	0	1	1	0	1	0

- **Décalage de bits** : décale tous les bits vers la gauche ou vers la droite dépendant de l'opération utilisée. Décaler les bits vers la gauche consiste à insérer i bits à la position 1, où i est un paramètre d'entrée de l'opération. Par exemple, décaler le nombre 1111 de 2 crans vers la gauche nous donne le résultat 111100. Tandis que décaler les bits vers la droite consiste à supprimer les i bits les plus faibles. Par exemple, décaler le nombre 1111 de 2 crans vers la droite nous donne le résultat 11.

- **OU exclusif** : renvoie un résultat où chaque bit vaut 1 si exactement un bit à cette même position dans un des deux opérandes vaut 1.

|	0	1
0	0	1
1	1	0

- **Inversion de bit** : inverse la valeur d'un bit à une position donnée.
- **Set de bit** : met la valeur d'un bit à une position donnée à 1.

3.4.3 Méthodologie

Il est important en exécutant des tests de benchmark en Java de bien chauffer la JVM (Java Virtual Machine). Lorsque du code est exécuté en Java, toutes les classes ne sont pas chargées en mémoire. Donc lorsque l'on fait appel à une fonction, celle-ci n'est pas directement exécutée, mais les différentes classes nécessaires à son fonctionnement sont dans un premier temps chargées en mémoire et le bytecode correspondant à la fonction est interprété. 10 000 itérations sont requises (par défaut) avant qu'un code soit compilé et considéré comme "chaud", ce nombre d'itérations varie en fonction du temps nécessaire à l'exécution d'une fonction.

Lorsqu'une fonction est appelée en boucle de nombreuses fois avec les mêmes paramètres, la JVM peut optimiser (dépend de nombreux paramètres) le code en éliminant le code mort, c'est-à-dire du code appelé dont sa valeur de retour n'est jamais utilisée et que cette méthode n'a pas d'effet secondaire, c'est-à-dire une méthode modifiant l'objet appelant, dans ce cas, si l'objet est utilisé par la suite, l'optimisation n'est pas effectuée. C'est pourquoi, certains tests vont stocker le résultat dans un tableau afin d'éviter cette optimisation de la JVM.

Les tests sont tous initialisés avec des nombres aléatoires afin d'éviter l'effet "constant folding" qui lors de la compilation du code, va remplacer toutes les expressions constantes par sa valeur.

Le benchmark des différentes opérations binaires n'a pas toujours pu être réalisé sur le `StringBuilder`, cette structure ne contient aucune opération binaire directement, il est nécessaire d'implémenter soi-même ces opérations si on en a le besoin. Lors de l'implémentation des codes utilisant le `StringBuilder`, il faut prendre en compte le fait qu'on manipule des caractères et non des entiers, la logique de ces implémentations est donc différente.

Pour chaque benchmark, la fonction testée est exécutée 10 000 fois dans la plupart des cas afin d'effectuer la chauffe de la JVM, ce nombre d'itérations est réduit ou augmenté respectivement si la méthode appelée est lente ou rapide d'exécution.

Une fois la chauffe terminée, les points du graphe sont générés pour former une ligne polygonale. Chaque ligne polygonale est formée de 100 points différents, l'axe des abscisses représente la taille d'un entier en bits, tandis que l'axe des ordonnées représente le temps d'exécution en millisecondes de x exécutions d'une fonction, x dépend de la complexité de la fonction, plus la fonction est complexe et plus la valeur choisie pour x sera faible. L'axe des ordonnées représente donc le temps pour x exécutions et non pas le temps moyen d'exécution, cette représentation a été choisie afin de faciliter la lecture et de ne pas changer régulièrement entre des représentations en millisecondes et en microsecondes.

Il existe des librairies réputées permettant de faire du benchmark de codes (comme `JMH` par exemple). Cependant le résultat d'un benchmark effectué par `JMH` est différent d'un benchmark réalisé soi-même. On pourrait s'attendre à ce que les résultats soient proportionnellement similaires, mais ce n'est pas le cas (dû aux différentes optimisations faites par `JMH` qui va modifier le comportement de la JVM). J'ai préféré exécuter le benchmark moi-même afin d'avoir des temps similaires à l'exécution du programme.

3.4.4 Temps d'exécution

Une application open source a été développée afin d'effectuer les différents benchmark de cette section [10]. Ce benchmark a été réalisé sur un processeur AMD Ryzen 9 5900X 12-Core avec la version 19 de Java.

Regardons maintenant le résultat du benchmark. La figure 13 illustre la première opération que nous allons comparer, il s'agit de l'ajout entre deux entiers.

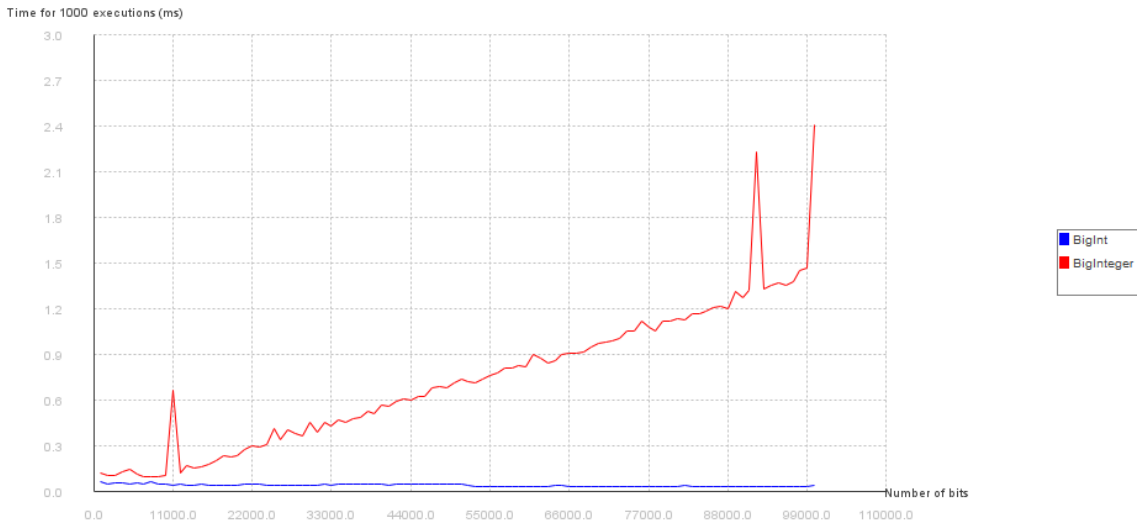


Figure 13: Évolution du temps d'exécution d'un ajout entre deux éléments en fonction de la taille d'un message.

On constate ici que le temps d'exécution est constant pour les deux structures, le `BigInt` est beaucoup plus rapide que le `BigInteger` lorsque la taille des entiers est élevée. Cette différence de temps s'explique par le fait que le `BigInteger` est immuable, c'est-à-dire que lors de l'ajout d'un entier à un autre, au lieu de faire cet ajout immédiatement, le résultat est stocké dans une variable intermédiaire, du temps de calcul non négligeable est donc perdu pour la création d'un nouvel objet contenant le résultat. De manière générale, pour chaque opération binaire qui modifie un entier en entrée, le `BigInteger` aura un temps d'exécution médiocre.

La seconde opération est le ET binaire entre deux entiers, la figure 14 représente cette comparaison.

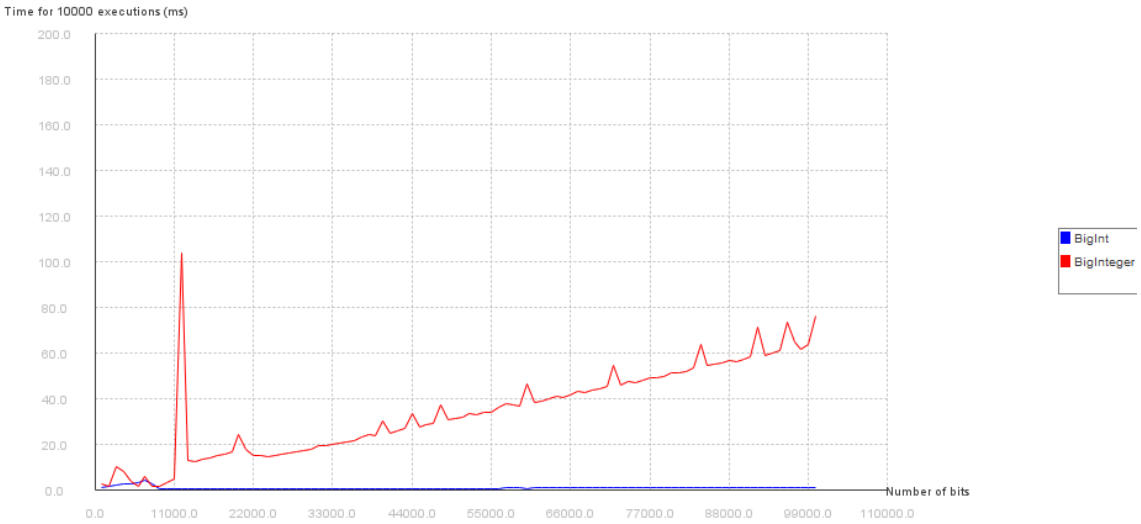


Figure 14: Évolution du temps d'exécution d'un ET binaire entre deux éléments identiques en fonction de la taille d'un message.

La troisième opération comparée à la figure 15 compte le nombre de bit valant 1 pour un entier.

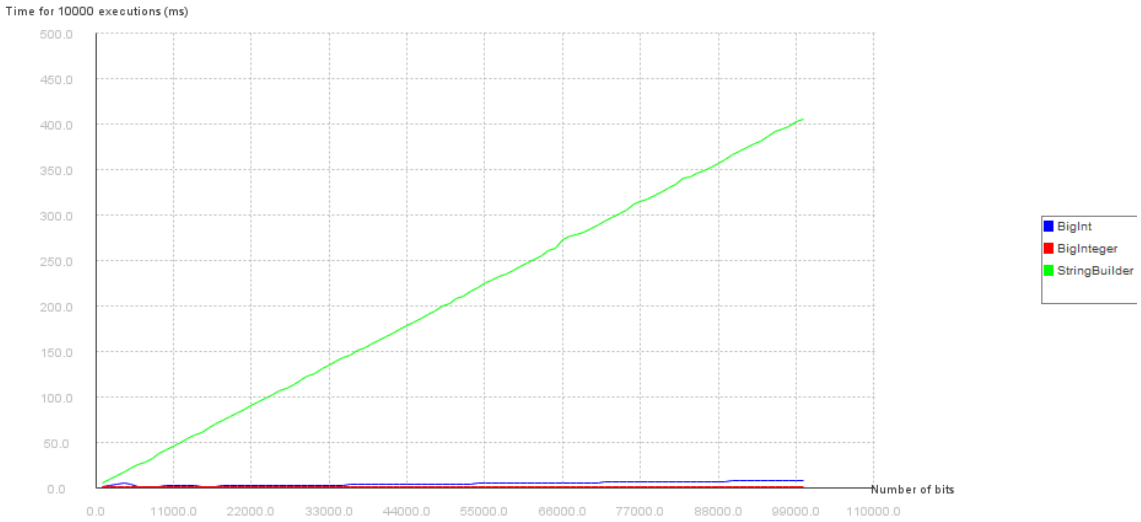


Figure 15: Évolution du temps d'exécution d'une fonction comptant le nombre de bits valant 1 d'un message en fonction de sa taille.

On remarque que le `BigInteger` offre de très bonnes performances dû au fait que la structure de données contient une variable `bitCountPlusOne` qui renseigne le nombre de bits valant 1 contenu dans la structure + 1.

La quatrième opération va tester si un bit à un index donné vaut 1, la figure 16

illustre la comparaison de cet opération.

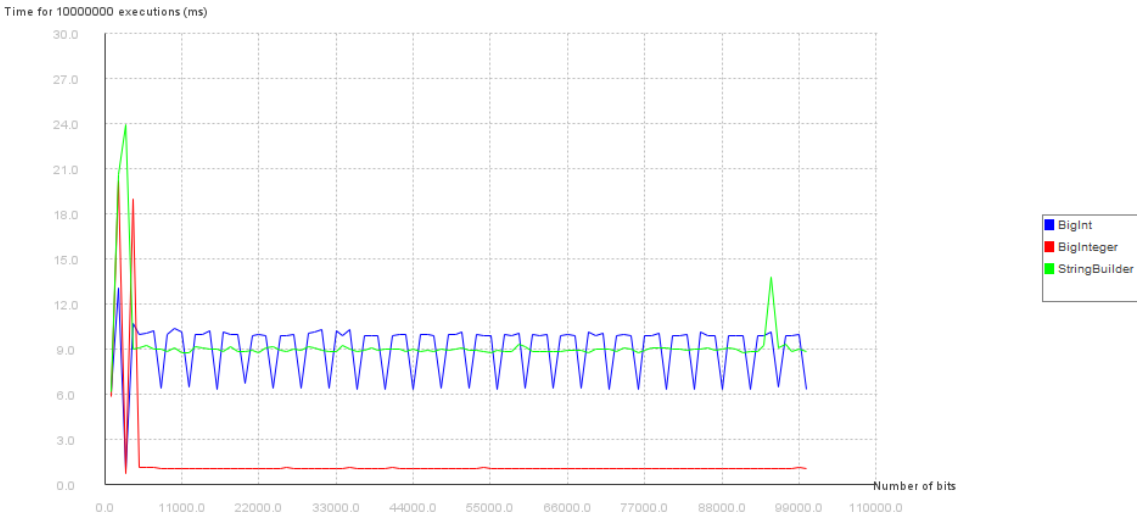


Figure 16: Évolution du temps d'exécution d'un test de bit en fonction de la taille d'un message.

La cinquième opération va insérer un bit à une position donnée, la figure 17 montre la comparaison de cet opération.

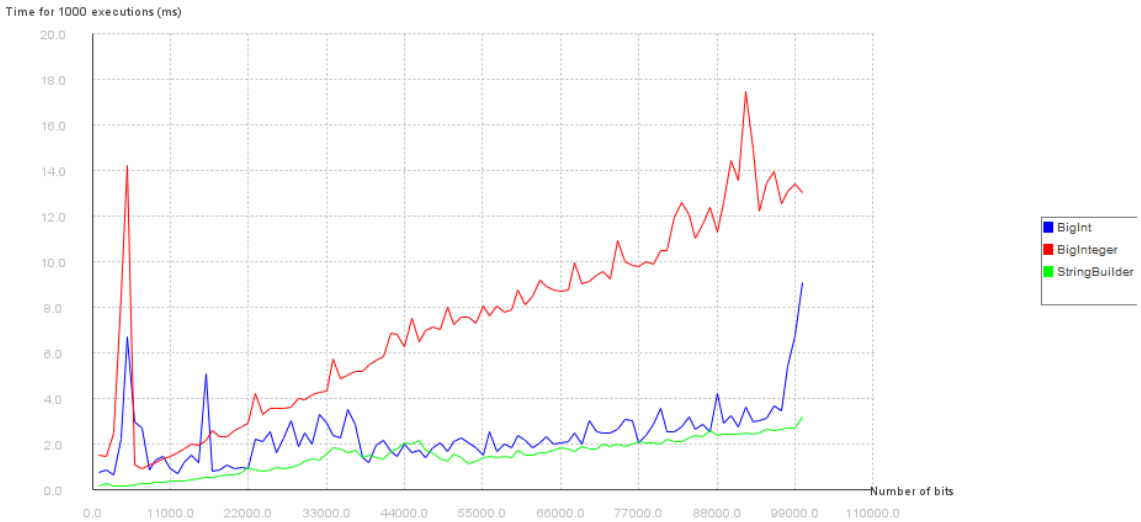


Figure 17: Évolution du temps d'exécution d'une fonction insérant un bit à une position donnée en fonction de la taille d'un message.

La sixième opération comparée à la figure 18 permet d'obtenir l'index du bit ayant le poids le plus fort d'un entier.

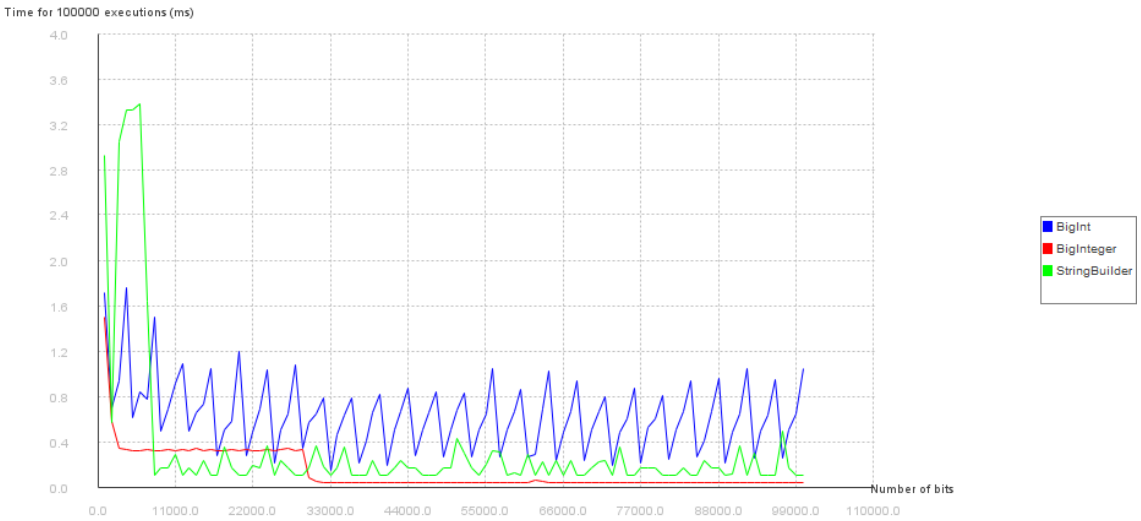


Figure 18: Évolution du temps d'exécution d'une fonction donnant la position du bit de poids fort en fonction de la taille d'un message.

La septième opération permet d'obtenir un entier dont tous les bits sont inversés La figure 19 montre la comparaison de cet opération.

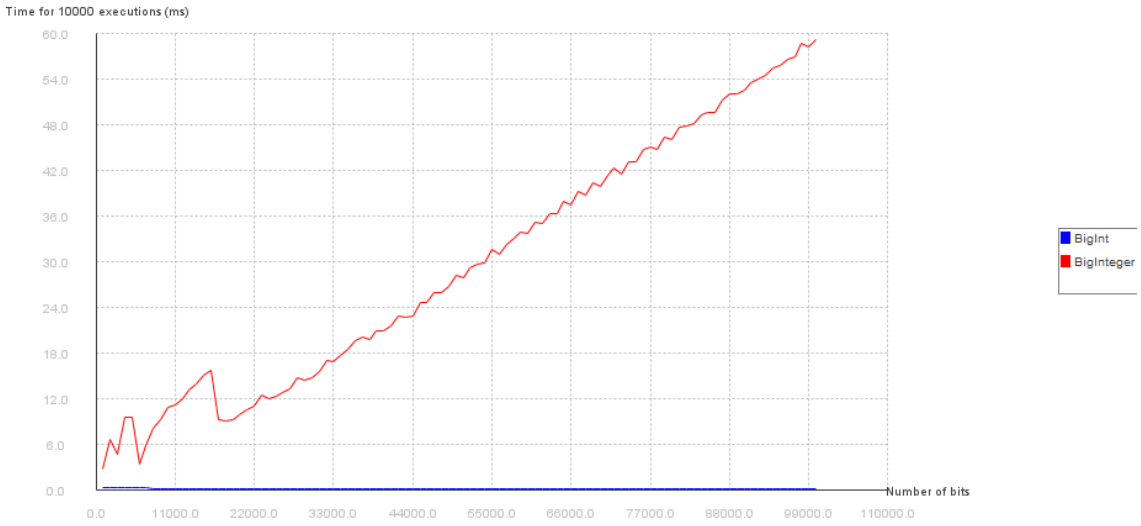


Figure 19: Évolution du temps d'exécution d'un NON binaire en fonction de la taille d'un message.

La huitième opération permet de décaler tous les bits d'un entier de x bits vers la gauche, où x est un paramètre de l'opération. Plus la valeur x est élevée, plus le temps d'exécution sera élevé, il a été choisi de définir $x = 16$ pour une première comparaison illustrée à la figure 20, car pour le codage en utilisant la somme de contrôle Internet (RFC 1071), nous devons décaler les bits du message de 16 bits

vers la gauche. Pour une seconde comparaison illustrée à la figure 21, x a été défini à la valeur 32, car certains CRC de degré 32 sont couramment utilisés dans les réseaux de télécommunications.

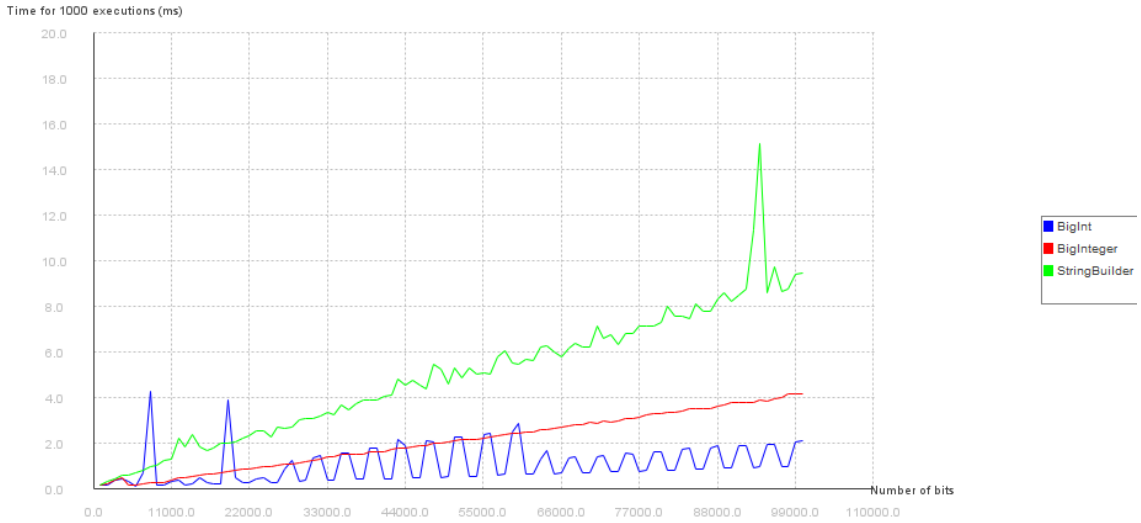


Figure 20: Évolution du temps d'exécution d'une fonction décalant tous les bits de 16 bits à gauche en fonction de la taille d'un message.

Le temps d'exécution pour le `BigInt` est sous forme d'une courbe sinusoïdale, car dépendamment de la taille du message, il ne doit pas allouer un nouveau tableau d'entier pour contenir le résultat dans le cas où $n/32 = (n+16)/32$, où n est la valeur du bit de poids fort, nous nous trouvons dans le creux de cette courbe dans ce cas là. Dans le cas contraire, ses performances sont similaires à celle du `BigInteger` qui va dans tous les cas allouer un nouveau tableau d'entier pour contenir le résultat.

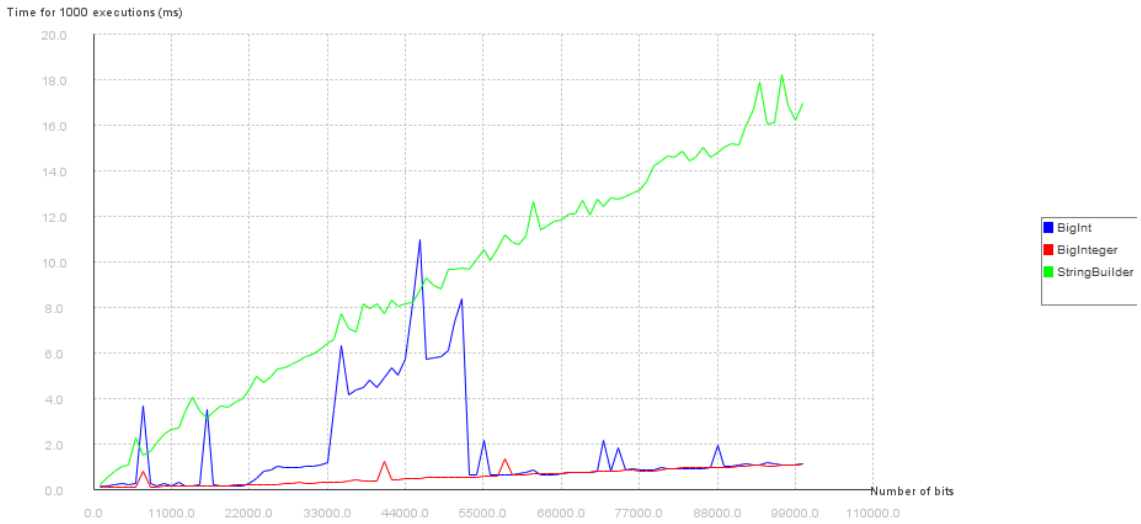


Figure 21: Évolution du temps d'exécution d'une fonction décalant tous les bits de 32 bits à gauche en fonction de la taille d'un message.

Cette fois-ci, le temps d'exécution du `BigInt` est similaire lorsque la taille du message est supérieure à 55 000 bits, car il doit allouer un nouveau tableau d'entier dans tous les cas comme pour le BigInteger.

La neuvième opération décale tous les bits d'un entier de x bits vers la droite, où x est un paramètre de l'opération. La figure 22 représente la comparaison de cette opération, dans ce cas-ci, $x = 16$.

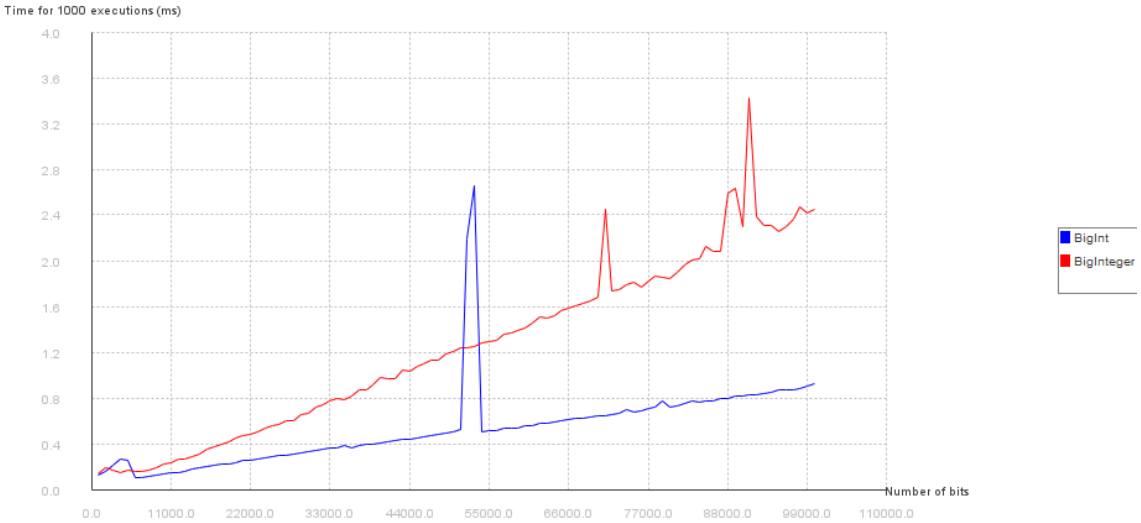


Figure 22: Évolution du temps d'exécution d'une fonction décalant tous les bits de 16 bits à droite en fonction de la taille d'un message.

La dixième opération illustrée à la figure 23 est le OU exclusif entre deux entiers.

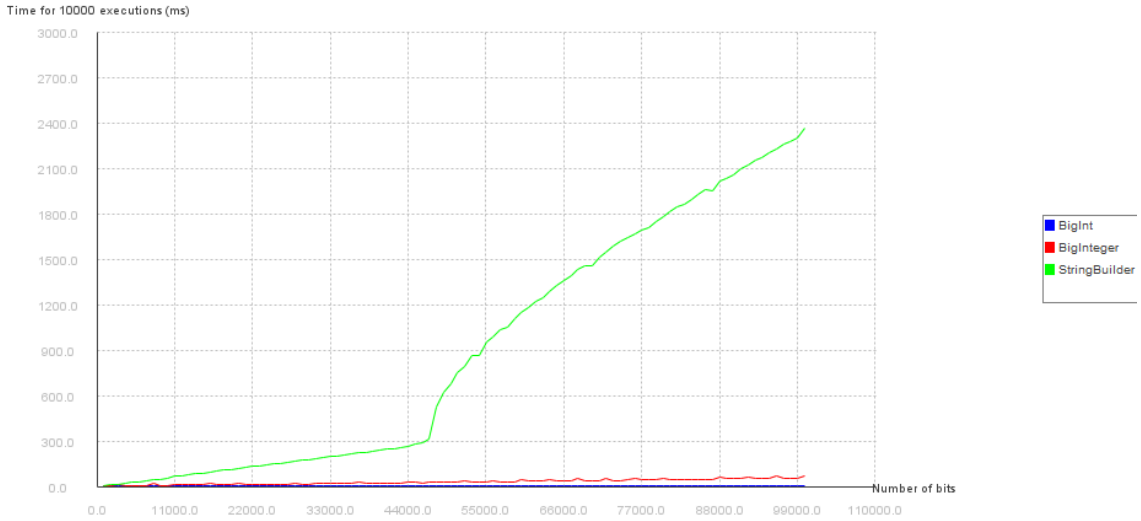


Figure 23: Évolution du temps d'exécution d'un XOR binaire entre deux messages identiques en fonction de leur taille.

La onzième opération est une inversion de bit à un index donné. La figure 24 montre cette comparaison.

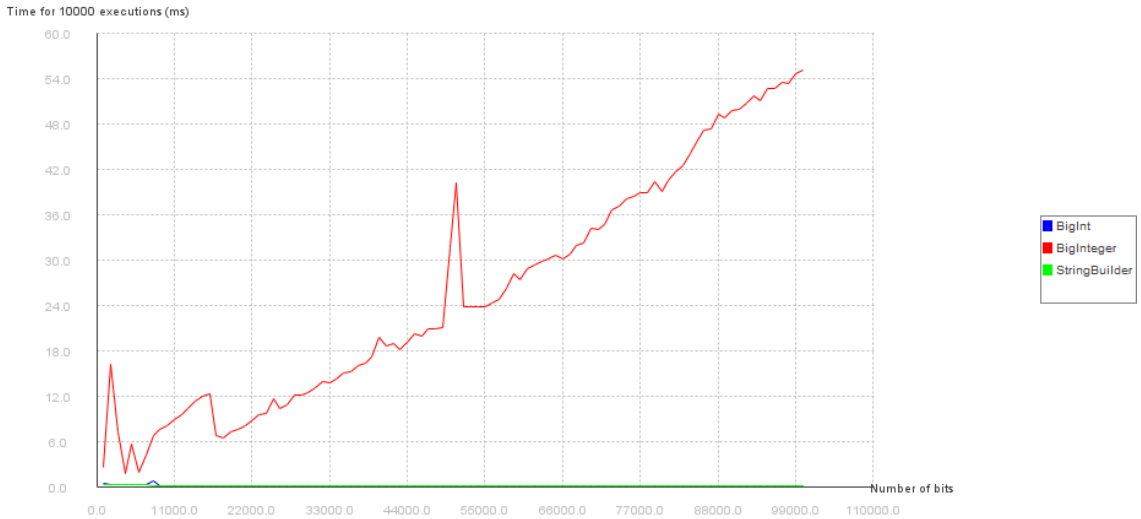


Figure 24: Évolution du temps d'exécution d'une inversion du bit à l'index le plus élevé en fonction de la taille d'un message.

Le temps d'exécution très élevé de cette opération pour le `BigInteger` ne nous permet pas de voir correctement la comparaison du temps d'exécution de cette

opération pour le `BigInt` et le StringBuilder, regardons maintenant les résultats à la figure 25 sans utiliser le `BigInteger`.

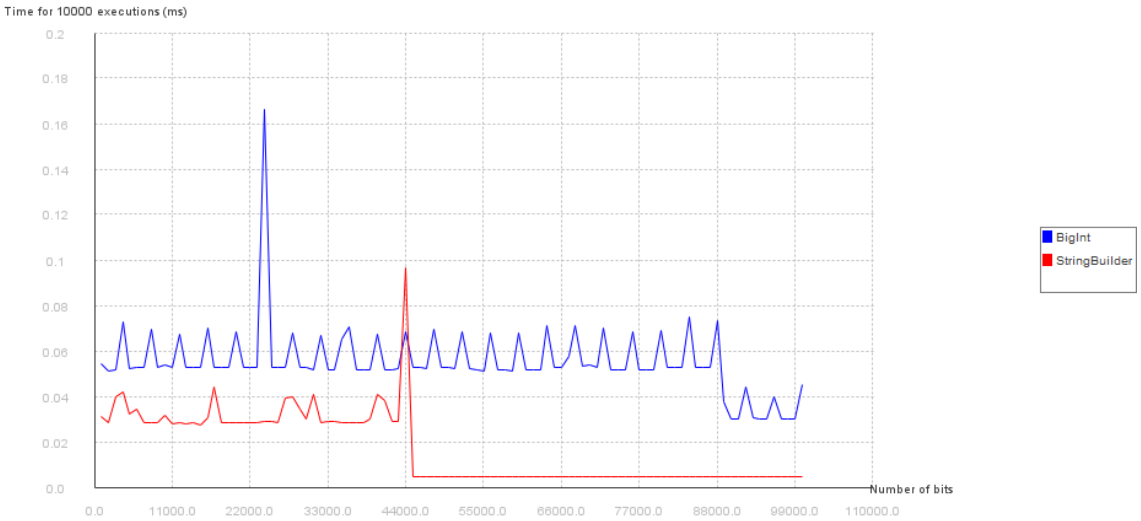


Figure 25: Évolution du temps d'exécution d'une inversion du bit à l'index le plus élevé entre le `StringBuilder` et le `BigInt` en fonction de leur taille.

La douzième opération illustrée à la figure 26 est la modification d'un bit à un index donné.

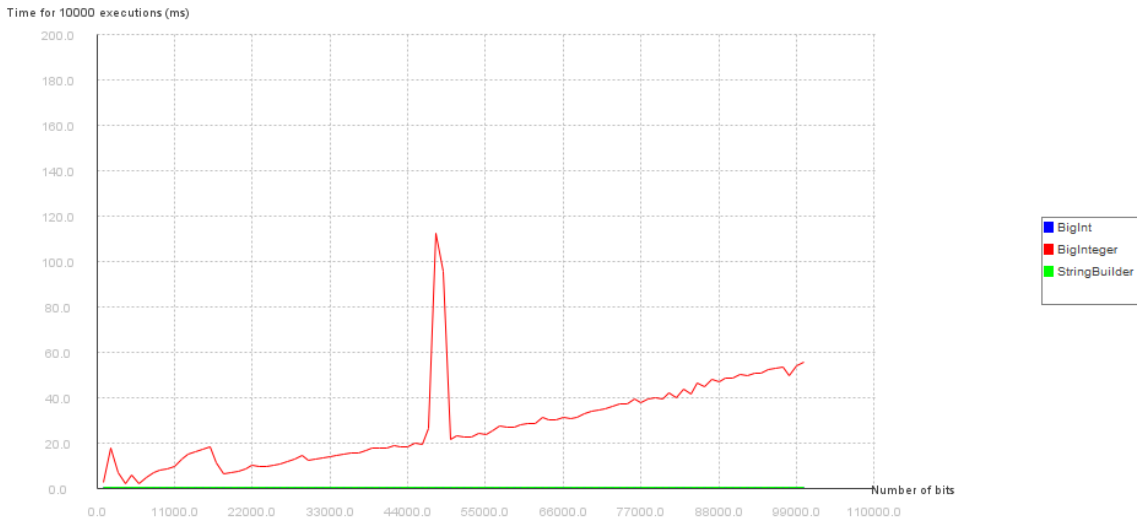


Figure 26: Évolution du temps d'exécution de la modification d'un bit à un en fonction de la taille d'un message.

À nouveau, le temps d'exécution très élevé de cette opération pour le `BigInteger` ne nous permet pas de voir correctement la comparaison du temps d'exécution de cette

opération pour le `BigInt` et le `StringBuilder`, regardons maintenant les résultats à la figure 27 sans utiliser le `BigInteger`.

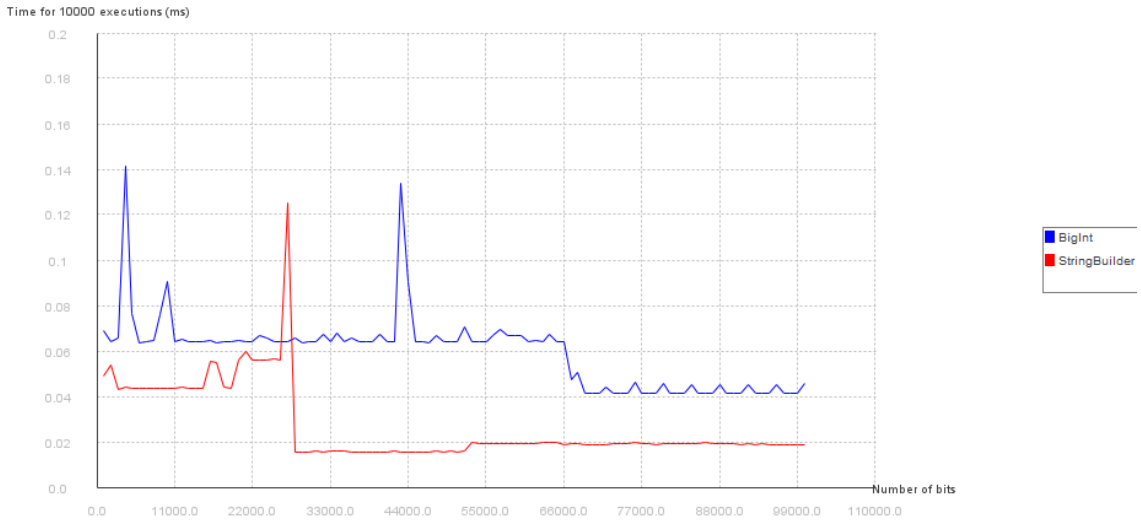


Figure 27: Évolution du temps d'exécution de la modification d'un bit à un en fonction de la taille d'un message.

Maintenant que nous avons vu le temps d'exécution des différentes opérations binaires, nous pouvons plus aisément comprendre le temps d'exécution des différents algorithmes de codage en fonction de la structure de données utilisée.

La première fonction de codage que nous allons analyser est le code à bit de parité, l'implémentation utilisée fait appel aux opérations binaires suivantes :

1. Décalage de bits vers la gauche.
2. Compter les bits valant 1.
3. Ajout d'un nombre à un autre.

La figure 28 montre le temps d'exécution d'une fonction encodant des messages avec le code à bit de parité. L'ajout d'un nombre à un autre est quasiment instantané puisqu'il s'agit d'ajouter 1 au message dans un cas et 0 dans l'autre. Les courbes obtenues sont donc expliquées par les performances des opérations comptant le nombre de bits qui valent 1 et le décalage de bits de 1 cran vers la gauche.

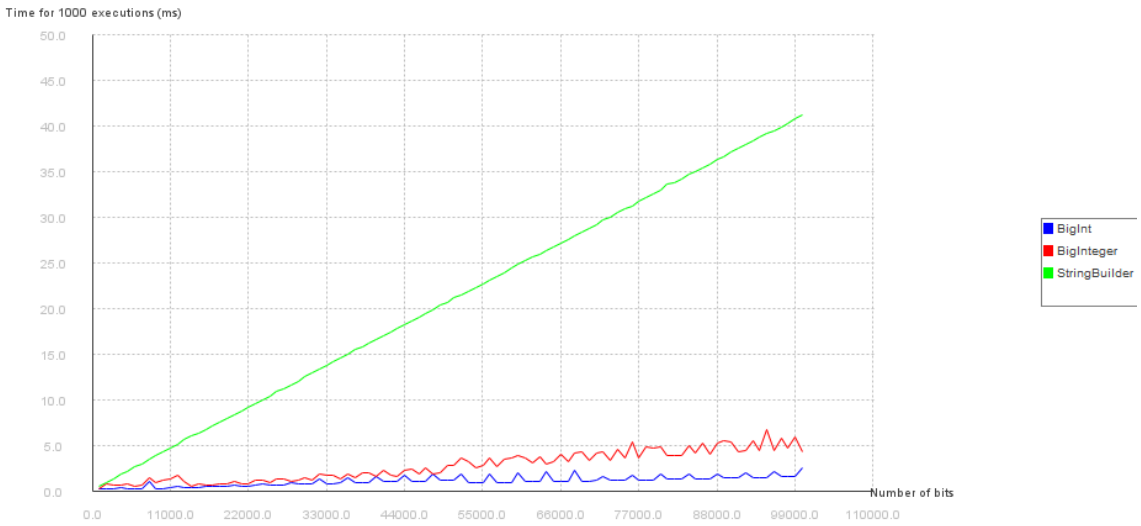


Figure 28: Évolution du temps d'exécution d'une méthode encodant un message avec le code de bit de parité en fonction de sa taille.

La seconde fonction de codage analysée est le code cyclique (CRC), l'implémentation utilisée fait appel aux opérations binaires suivantes :

1. Décalage de bits vers la gauche.
2. Rechercher l'index du bit ayant le poids le plus fort.
3. Tester la valeur d'un bit (pour le `BigInt` et le StringBuilder).
4. Inverser un bit (pour le `BigInt` et le StringBuilder).
5. OU exclusif (pour le BigInteger).

Les opérations utilisées ne sont pas les mêmes pour le `BigInteger` car cette structure de données est immuable, il faut alors effectuer le moins d'opérations possible afin d'éviter de copier l'entier un nombre trop important de fois. La figure 29 illustre le temps d'exécution d'une fonction encodant des messages avec le CRC32 étant l'un des polynômes les plus utilisés $g(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$. Le choix de ce polynôme va influencer les performances de codage d'un message car la taille de la somme de contrôle à ajouter au message dépend du degré du polynôme générateur.

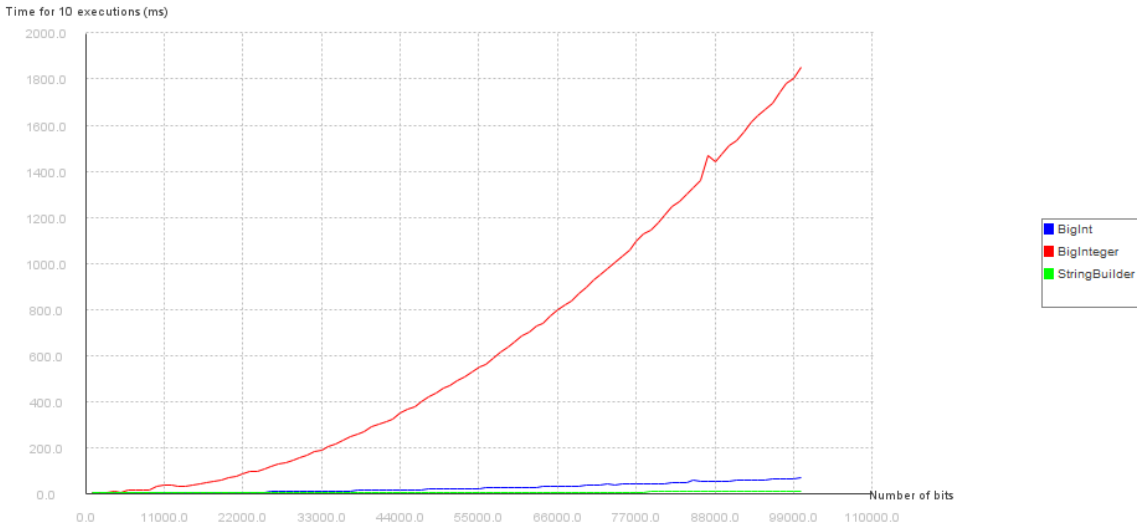


Figure 29: Évolution du temps d'exécution d'une méthode encodant un message avec le polynôme générateur CRC-32 en fonction de sa taille.

On s'attendait à voir un temps d'exécution pour le `BigInt` plus rapide que celui du StringBuilder. Cependant, dans les opérations utilisées pour l'implémentation des CRC, seule l'opération du `StringBuilder` décalant les bits vers la gauche est en réalité plus lente que celle du BigInt, cette opération n'est effectuée qu'une seule fois dans tout l'algorithme. Contrairement aux autres opérations qui sont appelés un nombre important de fois, ce qui explique pourquoi le temps d'exécution pour le `StringBuilder` est plus rapide que celui du BigInt.

La troisième fonction de codage est la somme de contrôle internet (RFC 1071), l'implémentation utilisée fait appel aux opérations binaires suivantes :

1. Décalage de bits vers la gauche.
2. Ajout d'un nombre à un autre.
3. Non binaire.
4. Et binaire.

La figure 30 montre le temps d'exécution d'une fonction encodant des messages avec la somme de contrôle internet.

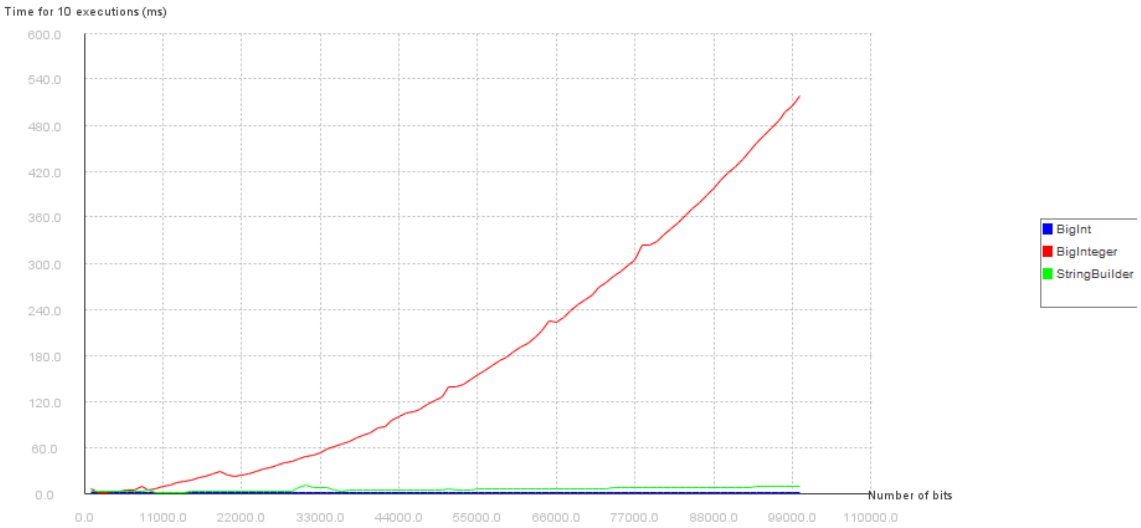


Figure 30: Évolution du temps d'exécution d'une méthode encodant un message avec la somme de contrôle Internet (RFC 1071) en fonction de sa taille.

Les temps d'exécution pour le `BigInt` et le `StringBuilder` sont semblables, leur implémentation est similaire, ils utilisent tous les deux des `long` pour effectuer les additions. La différence est qu'il est nécessaire de couper le `StringBuilder` en bloc de longueur 16 dans un premier temps et dans un second temps de transformer le `String` de longueur 16 en entier. Alors que le `BigInt` possède dans un structure interne un tableau d'`int`, ce qui nous permet de directement faire les additions de chaque `int` entre eux et de profiter des propriétés mathématiques permettant de faire l'addition en parallèle que nous avons vu dans la section 2.4.

La quatrième fonction de codage est le code de Hamming, l'implémentation utilisée utilise les opérations binaires suivantes :

1. Rechercher l'index du bit ayant le poids le plus fort.
2. Tester la valeur d'un bit.
3. Insérer un bit à un index donné.
4. Modifier la valeur d'un bit à un.

La figure 31 montre le temps d'exécution d'une fonction encodant des messages avec les codes de Hamming.

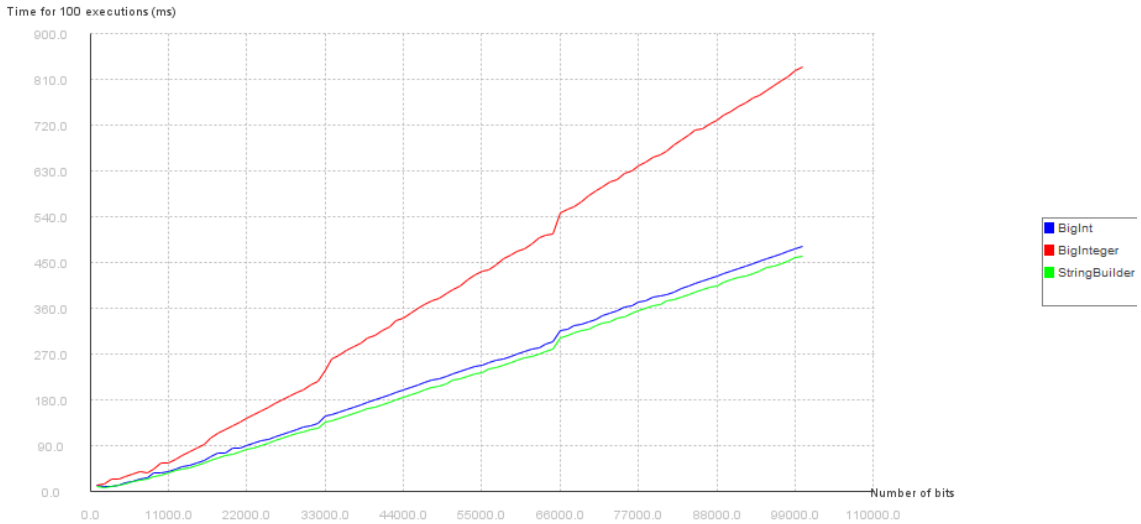


Figure 31: Évolution du temps d'exécution d'une méthode encodant un message avec le code de Hamming en fonction de sa taille.

On constate que les temps d'exécution pour le `BigInt` et le `StringBuilder` sont presque identiques, ce qui n'a rien d'étonnant lorsque l'on regarde les temps d'exécution pour les différentes opérations aux figures 18, 16, 17 et 26, on observe des temps similaires. Le `BigInteger` est quant à lui plus lent dû au fait qu'il perd du temps à créer des copies lors de l'insertion de bit à une position donnée, ainsi que lors de la modification d'un bit donné.

Les benchmarks nous permettent d'en conclure que le `BigInteger` n'est pas du tout adapté pour les algorithmes manipulant les bits d'un entier en mémoire. Le choix entre le `StringBuilder` et le `BigInt` n'est pas évident non plus. Si l'objectif est d'avoir le temps d'exécution le plus rapide possible, alors il faut alors utiliser l'une ou l'autre structure en fonction du codage à utiliser.

3.4.5 Espace mémoire

La complexité des opérations des différentes structures n'est pas le seul critère, il est également intéressant de s'intéresser à la taille en mémoire occupée par les structures. Regardons maintenant plus en détail l'espace mémoire occupée par un `StringBuilder` en java, la figure 32 nous montre les détails concernant cet espace mémoire utilisé.

OFF	SZ	TYPE	DESCRIPTION	VALUE
0	8		(object header: mark)	0x1
8	4		(object header: class)	0x5b9f0
12	4	int	count	6
16	1	byte	coder	0
17	1	boolean	maybeLatin1	false
18	2		(alignment/padding gap)	
20	4	byte[]	StringBuilder.value	[109, 101, 109, 111, 114, ...]

Figure 32: Espace mémoire utilisée par le `StringBuilder` contenant **memory** en java.

On constate ici que 24 bytes sont utilisés pour représenter un `StringBuilder`. Si on regarde en détail le contenu de cet objet, on constate 2 bytes utilisés par `(alignment/padding gap)`. En java, la taille en byte des objets doit être un multiple de 8 (dans la plupart des systèmes) afin que chaque objet soit aligné, cet alignement permet d'accéder à la mémoire plus rapidement. Donc si la taille d'un objet n'est pas sur 8 bytes, le système va ajouter des bytes d'alignement.

Dans le `StringBuilder`, on constate également que `StringBuilder.value` est représenté sur 4 bytes, en réalité cette zone mémoire ne contient que le pointeur vers un tableau de bytes, il est donc nécessaire de calculer la taille d'un tableau de bytes pour obtenir la taille totale occupée par un `String`. L'espace mémoire utilisée par un tableau de bytes est observable à la figure 33.

OFF	SZ	TYPE	DESCRIPTION	VALUE
0	8		(object header: mark)	0x1
8	4		(object header: class)	0x21f0
12	4		(array length)	22
16	22	byte	[B.<elements>	N/A
38	2		(object alignment gap)	

Figure 33: Espace mémoire utilisée par le tableau de bytes contenant **memory** en java.

On s'attend à voir une taille de 6 bytes si un caractère est représenté sur un byte en

Java. Cependant, 22 bytes sont utilisés pour représenter la chaîne **memory**. Si on regarde plus en détail dans le constructeur du `StringBuilder`, on remarque qu'il alloue une taille plus grande (taille de la chaîne + 16 bytes) en prévision d'un ajout éventuel de plusieurs caractères. La taille de celui-ci va donc dépendre du nombre de caractères dans le `StringBuilder`. Le tableau de byte contient donc 16 bytes de méta-données et ensuite $n + 16$ bytes de données, où n représente le nombre de caractères dans le tableau, suivi d'un nombre x entre 0 et 7 bytes pour aligner les données.

Un `StringBuilder` contient donc au total $24 + 16 + n + 16 + x$ bytes. Cette structure de données n'est pas optimisée pour travailler avec des messages binaires du fait que chaque bit est représenté par un byte. Pour un message de 2048 bits par exemple, il faudrait 2104 bytes, soit 16832 bits en mémoire, pour pouvoir être représenté. Cependant, cette structure possède l'avantage de pouvoir représenter un nombre très élevé de caractères. La taille maximale d'un `StringBuilder` est limitée par 2 facteurs, soit par la taille maximale du tas disponible, soit par une taille de 2.147.483.647($2^{31} - 1$) caractères. Cette limite est basée sur le fait que nous ne considérons que le stockage des caractères '0' et '1'. Il est évidemment possible de stocker n'importe quelle valeur dans chaque caractère et seulement considérer les bits contenus dans chaque byte, ce qui rendrait l'utilisation du `StringBuilder` complètement insensé.

Passons maintenant à l'espace mémoire occupée par un `BigInteger` en java illustré à la figure 34.

OFF	SZ	TYPE	DESCRIPTION	VALUE
0	8		(object header: mark)	0x9
8	4		(object header: class)	0x8f2f0
12	4	int	BigInteger.signum	1
16	4	int	BigInteger.bitCountPlusOne	0
20	4	int	BigInteger.bitLengthPlusOne	0
24	4	int	BigInteger.lowestSetBitPlusTwo	0
28	4	int	BigInteger.firstNonzeroIntNumPlusTwo	0
32	4	int[]	BigInteger.mag	[-2147483648, 0, 0]
36	4		(object alignment gap)	

Figure 34: Espace mémoire utilisée par un `BigInteger` contenant $2^{95}(1 << 95)$ en java.

Cet objet contient également un nombre fixe de 40 bytes, on constate que la valeur du message est contenue dans un tableau d'entier, la taille de celui-ci devrait être identique à la taille du tableau de bytes 33, à l'exception que chaque entier (`int`)

dans le tableau est représenté sur 4 bytes au lieu de 1 byte. Observons son contenu en mémoire à la figure 35.

OFF	SZ	TYPE	DESCRIPTION	VALUE
0	8		(object header: mark)	0x1
8	4		(object header: class)	0x25b0
12	4		(array length)	3
16	12	int	[I.<elements>	N/A
28	4		(object alignment gap)	

Figure 35: Espace mémoire utilisée par le tableau d'entier représentant $2^{95}(1 << 95)$ en java.

Comme pour le tableau de bytes 33, 16 bytes fixes sont utilisées pour les méta-données du tableau ensuite chaque entier utilise 4 bytes de mémoire. Cette fois-ci, pour représenter un nombre contenant 96 bits, on a besoin de seulement 12 bytes au lieu de 96 bytes utilisés par un String. Si le tableau contient un nombre impaire d'entiers, alors il sera nécessaire d'ajouter 4 bytes pour aligner les données.

Un `BigInteger` contient donc au total $40 + 16 + \lceil n/32 \rceil * 4 + x$ bytes. Où n est égal au nombre de bits du message et x vaut 4 si $\lceil n/32 \rceil \% 2 = 1$, sinon x vaut 0. La taille d'un `BigInteger` est comprise entre $-2^{Integer.MAX\_VALUE}$ (exclus) et $+2^{Integer.MAX\_VALUE}$ (exclus) où $Integer.MAX\_VALUE = 2.147.483.647$. Chaque bit du message est réellement stocké sur un bit, ce qui rend cette structure très intéressante pour représenter des messages d'une taille arbitraire.

Passons maintenant à l'espace mémoire occupée par un `BigInt` [4] en java 36 :

OFF	SZ	TYPE	DESCRIPTION	VALUE
0	8		(object header: mark)	0x1
8	4		(object header: class)	0x010ac
12	4	int	BigInt.sign	1
16	4	int	BigInt.len	3
20	4	int	BigInt.dig	[0, 0, -2147483648, 0]

Figure 36: Espace mémoire utilisée par un `BigInt` contenant $2^{95}(1 << 95)$ en java.

Cet objet contient un nombre fixe de 24 bytes, comme pour le BigInteger, sa valeur est stockée dans un tableau d'entier. Cependant on constate que lors de la représentation de $2^{95}(1 << 95)$, le tableau utilisé pour représenter ce nombre, utilise un entier en plus que pour la même représentation en utilisant un BigInteger. Ce qui s'explique par la définition des opérations dans la classe du BigInt. Si on regarde plus en détail ces opérations, on constate qu'il n'y aura qu'au pire un entier

non nécessaire utilisé pour représenter un message d'une taille arbitraire, ce qui est au final négligeable.

Un `BigInt` contient donc au total $24 + 16 + \lceil n/32 \rceil * 4 + 4 + x$ bytes. Où n est égal au nombre de bits du message et x vaut 4 si $\lceil n/32 \rceil \% 2 = 1$, sinon x vaut 0. Pour connaître sa taille maximale, il faut s'intéresser à la taille maximale d'un tableau en Java, cette taille maximale est constante et ne dépend pas de l'objet, cette taille est de $2,147,483,647$ maximum. Le `BigInt` peut donc contenir un message contenant maximum $2.147.483.647 * 32 = 68.719.476.704$ bits.

Un `BigInt` va donc utiliser moins de mémoire qu'un `BigInteger` pour un même message. Il est donc plus intéressant d'utiliser le `BigInt` que le `BigInteger` pour l'aspect mémoire.

Cependant, si la taille des messages est faible, pour toutes structures présentées ci-dessus, ces structures possèdent un coût en mémoire non négligeable. Il est donc intéressant d'utiliser les `bytes` (8 bits), `int` (32 bits) ou les `long` (64 bits) si la taille des messages est restreint.

3.5 Implémentation des codes

Cette section présente les algorithmes implémentés dans l'application [9] permettant de coder et de décoder les messages, l'objectif est de fournir une compréhension approfondie des algorithmes implémentés. Il est important de noter qu'il existe d'autres algorithmes plus optimisés dans la littérature.

3.5.1 Bit de parité

Comme nous l'avons vu à la section 2.2, le code à bit de parité va simplement ajouter un bit de parité à la fin du message, voici une implémentation possible permettant d'encoder un message :

```
1  public void encode(BigInt message) {
2      message.shiftLeft(1);
3      if (message.getBitCount() % 2 != 0) {
4          message.add(1);
5      }
6  }
```

La fonction `encode` va simplement décaler les bits du message de 1 cran vers la gauche et va ensuite compter le nombre de bit valant 1, si ce nombre est impaire, alors on va ajouter 1 au message, sinon il ne faut rien ajouter.

```
1  public boolean isCorrupted(BigInt message) {
2      return message.getBitCount() % 2 != 0;
3  }
```

La fonction `isCorrupted` va renvoyer vrai si le nombre de bit du message codé valant 1 est impair.

Voici une implémentation possible permettant de décoder un message :

```
1  public void decode(BigInt encodedMessage) {
2      if (isCorrupted(encodedMessage)) {
3          throw new RuntimeException("Message is corrupted");
4      }
5      encodedMessage.shiftRight(1);
6  }
```

La fonction `decode` va vérifier l'intégrité du message. Si il est corrompu, une exception est lancée, si il ne l'est pas, il suffit simplement de décaler les bits de 1 cran vers la droite pour décoder le message.

3.5.2 Codes cycliques (CRC)

L'implémentation réalisée par la suite est une implémentation simple des CRC que nous avons parcourus dans la section 2.3, il existe d'autres algorithmes plus optimisés tels que le calcul via des tables de correspondance (look-up tables) [7]. Il est important de noter que selon les standards, les résultats de ces algorithmes diffèrent. Voici une implémentation simple possible permettant d'encoder un message :

```
1  public static BigInt getPolynomialArithmeticModulo2(BigInt dividend,
2                                                       BigInt divisor) {
3      BigInt remainder = new BigInt(dividend);
4      int remainderLeftMostSetBit = remainder.getLeftMostSetBit();
5      int divisorLeftMostSetBit = divisor.getLeftMostSetBit();
6
7      int i;
8      while (remainderLeftMostSetBit >= divisorLeftMostSetBit) {
9          i = 0;
10         while (i < divisorLeftMostSetBit) {
11             if (divisor.testBit(divisorLeftMostSetBit - 1 - i)) {
12                 remainder.flipBit(remainderLeftMostSetBit - 1 - i);
13             }
14             i++;
15         }
16         remainderLeftMostSetBit = remainder.getLeftMostSetBit();
17     }
18     return remainder;
19 }
20
21 public void encode(BigInt message, BigInt generatorPolynomial) {
22     message.shiftLeft(generatorPolynomial.getLeftMostSetBit() - 1);
23     message.add(getPolynomialArithmeticModulo2(message,
24                 generatorPolynomial));
25 }
```

La fonction `getPolynomialArithmeticModulo2` va calculer le reste de la division euclidienne du message (dont les bits ont été au préalable décalés de r bits vers la gauche, où r est le degré du polynôme générateur) par le polynôme générateur $P_s.X^r mod P_g$. Dans un premier temps, cette fonction va créer une copie du message, afin de ne pas l'altérer.

La boucle `while` à la ligne 9, va effectuer un ou exclusif entre le message et le polynôme générateur dont ses bits ont été décalés vers la gauche, de sorte que son bit de poids le plus élevé soit aligné avec le bit de poids le plus élevé du message. La méthode `getLeftMostSetBit` va retourner un indice compris entre 1 et 32 du bit du poids le plus élevé, les fonctions `testBit` et `flipBit` quant à elles, utilisent

des indices compris entre 0 et 31, c'est pourquoi il est nécessaire de retirer 1 aux paramètres passés à ces fonctions.

La boucle `while` à la ligne 7, va s'assurer que les ou exclusifs soient exécutés jusqu'à ce que l'indice bit de poids fort du polynôme générateur soit supérieur à l'indice du bit de poids fort du reste de cette division euclidienne.

Au lieu de tester chaque bit du polynôme générateur et d'effectuer une inversion de bit (`flipBit`), il est possible de faire un décalage des bits de ce polynôme afin d'aligner le bit ayant le plus grand poids avec le bit ayant le plus grand poids du message et ensuite d'effectuer un ou exclusif entre le message et ce polynôme. Cependant, la fonction prend presque 4 fois plus de temps à s'effectuer sur un message ayant une longueur de 100 000 bits et un polynôme générateur de degré 32.

La fonction encode va tout simplement décaler les bits du message de r bits vers la gauche ($P_s.X^r$) et ensuite y ajouter le reste de la division euclidienne entre celui-ci et le polynôme générateur ($P_s.X^r + P_s.X^r mod P_g$).

```
1  public boolean isCorrupted(BigInt encodedMessage,
2                             BigInt generatorPolynomial) {
3      return !getPolynomialArithmeticModulo2(encodedMessage,
4              generatorPolynomial).isZero();
5  }
```

La fonction `isCorrupted` va vérifier si le reste entre la division euclidienne entre le message encodé et le polynôme générateur vaut 0 ou non. Si le reste vaut 0, alors le message n'est pas corrompu, sinon il l'est.

Voici une implémentation possible permettant de décoder un message :

```
1  public void decode(BigInt encodedMessage) {
2      if (isCorrupted(encodedMessage)) {
3          throw new RuntimeException("Message is corrupted");
4      }
5      encodedMessage.shiftRight(generatorPolynomial.getLeftMostSetBit() - 1);
6  }
```

La fonction `decode` va vérifier l'intégrité du message. Si il est corrompu, une exception est lancée, si il ne l'est pas, il suffit simplement de décaler les bits d'un nombre égal au degré du polynôme générateur crans vers la droite pour décoder le message.

3.5.3 Somme de contrôle Internet (RFC 1071)

Nous avons vu à la section 2.4 que le but de la somme de contrôle Internet est de calculer le complément à un de la somme des mots de taille 16 du segment. Voici une implémentation possible permettant de coder un message :

```
1  private BigInt getSumOfWords(BigInt message) {
2      long result = 0;
3      for (int i = 0; i < message.getDig().length; i++) {
4          result += message.getDig()[i];
5      }
6      while (BitUtil.leftMostSetBit(result) > 16) {
7          result = (result >> 16) + (result & 0xffff);
8      }
9      return new BigInt(result);
10 }
11
12 public BigInt getChecksum(BigInt message) {
13     BigInt sumOfWords = getSumOfWords(message);
14     sumOfWords.not();
15     sumOfWords.and(new BigInt(0xffff));
16     return sumOfWords;
17 }
18
19 public void encode(BigInt message) {
20     BigInt checksum = getChecksum(message);
21     message.shiftLeft(16);
22     message.add(checksum);
23 }
```

Regardons en détail le fonctionnement de `getSumOfWords`. À la ligne 3 et 4, celle-ci va faire la somme de chaque entier contenu dans le tableau d'entier représentant le message, cette somme sera stockée dans un long (64 bits). Il n'est pas nécessaire de découper les entiers de 32 bits en entier de 16 bits grâce à la propriété de l'addition en parallèle vu à la section 2.4, accélérant le calcul et facilitant la lisibilité du code.

Le résultat obtenu aux lignes 3 et 4 peut être stocké sur plus de 16 bits, il faut donc reporter les bits dont l'index i est supérieur à 16 à un nouvel index $i' = i - 16$ et l'ajouter au bit se trouvant à l'index i'. La ligne 6 et 7 vont s'occuper de ce report, tant que le bit de poids fort du résultat de la somme est supérieur à 16, nous allons extraire les 16 bits de poids faible grâce à un ET binaire entre le long et le nombre `0xffff` qui représente un entier dont les 16 bits de poids faible sont tous mis à 1 (cette extraction correspond à `result & 0xffff`). Les 16 bits extraits sont ajoutés au résultat dont les bits sont décalés de 16 crans vers la droite (`resultat >> 16`). À la fin de cette boucle, la variable `result` contient la somme de tous les mots du message.

La fonction `getChecksum` quant à elle, va simplement inverser tous les bits de la somme des mots (ligne 14). Du fait que la somme des mots est représentée par des entiers de 32 bits, il est important d'effectuer un ET binaire entre `0xffff` afin de

s'assurer d'uniquement recevoir le résultat des 16 derniers bits.

La fonction `encode` va simplement décaler le message de 16 bits vers la gauche et y ajouter la somme de contrôle de celui-ci.

```
1  public boolean isCorrupted(BigInt encodedMessage) {
2      BigInt decodedMessage = new BigInt(encodedMessage);
3      decodedMessage.shiftRight(16);
4
5      BigInt encodedMessageChecksum = new BigInt(new BigInt(0xffff));
6      encodedMessageChecksum.and(encodedMessage);
7
8      return !getChecksum(decodedMessage).equals(encodedMessageChecksum);
9  }
```

La fonction `isCorrupted` va vérifier le contenu de la somme de contrôle dans le message codé, celle-ci se trouve dans les 16 derniers bits du message. Il est donc nécessaire de couper le message en 2, d'extraire le message et la somme de contrôle pour faire la vérification. Les lignes 2 et 3 vont créer une copie du message encodé et va décaler cette copie de 16 crans vers la droite afin d'extraire le message, une fois le message extrait, on peut calculer sa somme de contrôle avec la fonction vue précédemment `getChecksum`. Nous devons donc extraire la somme de contrôle contenue dans le message encodé afin de le comparer à la somme de contrôle que nous allons calculer. Les lignes 5 et 6 vont s'occuper de cette extraction de la somme de contrôle, en effectuant un et binaire entre un `BigInt` contenant les bits d'indices 1 à 16 valant 1 (`0xffff`) et entre le message encodé. Une fois la somme de contrôle extraite, il faut simplement vérifier l'égalité de ces deux sommes de contrôle. Si ces deux sommes correspondent, alors le message est intègre, sinon il est corrompu.

Voici une implémentation possible permettant de décoder un message :

```
1  public void decode(BigInt encodedMessage) {
2      if (isCorrupted(encodedMessage)) {
3          throw new RuntimeException("Message is corrupted");
4      }
5      encodedMessage.shiftRight(16);
6  }
```

La fonction `decode` va vérifier l'intégrité du message. Si il est corrompu, une exception est lancée, si il ne l'est pas, il suffit simplement de décaler les bits de 16 crans vers la droite pour décoder le message.

3.5.4 Codes de Hamming

La section 2.5 vu précédemment décrit le fonctionnement théorique des codes de Hamming, voici une implémentation possible permettant d'encoder un message :

```java
public static int numberOfRedundancyBitsToAdd(int messageLength) {
    int r = 0;
    int power = 1;
    while (power < (messageLength + r + 1)) {
        r++;
        power *= 2;
    }
    return r;
}

public void encode(BigInt message, int k) {
    int numberOfRedundancyBitsToAdd = numberOfRedundancyBitsToAdd(k);
    for (int i = 0; i < numberOfRedundancyBitsToAdd; i++) {
        message.insertBit((int) (Math.pow(2.d, i)) - 1, false);
    }

    int leftMostSetBit = message.getLeftMostSetBit();

    for (int i = 0; i < numberOfRedundancyBitsToAdd; i++) {
        int bitPosition = 1 << i;
        int numberOfOneForBitPosition = 0;

        for (int j = 2; j < leftMostSetBit; j++) {
            if (message.testBit(j)) {
                if ((bitPosition & (j + 1)) != 0) {
                    numberOfOneForBitPosition++;
                }
            }
        }
        if (numberOfOneForBitPosition % 2 == 1) {
            message.setBit(bitPosition - 1);
        }
    }
}
```

Dans la section 2.5, nous avons vu la formule permettant de trouver le nombre de bits de redondances r :

$$2^r \geq n + 1 \tag{2}$$

Nous savons que $n = k + r$, où n est la taille du message codé et k la taille du bloc. La fonction `numberOfRedundancyBitsToAdd` va donc initialiser r à 0 et ensuite, de manière itérative, incrémenter r jusqu'à ce que $2^r \geq k + r + 1$.

La fonction `encode` va encoder le message en deux étapes. Dans un premier temps, on va calculer r afin de pré-placer tous les bits de redondances aux indices valant une puissance de 2. La fonction permettant d'insérer un bit (`insertBit`) du `BigInt` décrite à la section 3.4.2 va nous permettre d'insérer tous les bits de redondances. Par exemple, pour un message $m = 1101$ donné, on utilise le code de Hamming(7,4), on doit donc ajouter 3 bits de redondances, regardons la valeur du message après chaque insertion de bit :

Valeur de i	Indice du bit de redondance	Valeur du message
0	1	11010
1	2	110100
2	4	1100100

Les lignes 12 et 13 vont s'occuper de faire l'insertion des bits de redondances, une fois ces bits ajoutés (initialisés à 0), il faut donc pour chacun de ces bits, calculer si sa valeur doit être modifiée à 1. Dans la boucle for à la ligne 19, on va regarder pour un indice de bit de redondance donné r_i, tous les bits du message valant 1 et dont la représentation binaire de leur indice contient le bit d'indice r_i valant 1, si ce bit vaut 1, la variable `numberOfOneForBitPosition` est incrémentée.

La boucle à la ligne 23 va simplement itérer sur les bits du message dont l'indice commence par 2, il est inutile de regarder les bits aux indices 0 et 1 puisque ceux-ci sont des bits de redondances ajoutés au préalable. Il faut noter que la variable `bitPosition` contient l'indice du bit de redondance dont sa valeur est une puissance de 2, alors que la méthode `testBit` et `setBit` utilise des indices commençant à 0, il faut donc faire attention à la comparaison entre cette variable et la variable j qui représente l'indice - 1 d'une position d'un bit du message.

Une fois le message parcouru pour un indice donné, il faut simplement regarder si la variable `numberOfOneForBitPosition` est impaire ou non. Si c'est le cas, il faut alors modifier le bit de redondance à l'indice r_i en mettant sa valeur à 1.

Voici une implémentation possible permettant de décoder un message :

```
1  public static int numberOfRedundancyBitsAdded(int encodedMessageLength) {
2      return BitUtil.binLog(encodedMessageLength + 1);
3  }
4
5  public void decode(BigInt encodedMessage, int n) {
6      int leftMostSetBit = encodedMessage.getLeftMostSetBit();
7      int numberOfRedundancyBitsAdded = numberOfRedundancyBitsAdded(n);
8
9      int errorBitPosition = 0;
```

```
10
11      for (int i = 0; i < numberOfRedundancyBitsAdded; i++) {
12          int bitPosition = 1 << i;
13          int numberOfOneForBitPosition = 0;
14
15          for (int j = 2; j < leftMostSetBit; j++) {
16              if (encodedMessage.testBit(j)) {
17                  if ((bitPosition & (j + 1)) != 0 &&
18                       BitUtil.isNotPowerOfTwo(j + 1)) {
19                      numberOfOneForBitPosition++;
20                  }
21              }
22          }
23
24          boolean isBitSet = encodedMessage.testBit(bitPosition - 1);
25          if (isBitSet != (numberOfOneForBitPosition % 2 == 1)) {
26              errorBitPosition += bitPosition;
27          }
28      }
29
30      if (errorBitPosition != 0) {
31          encodedMessage.flipBit(errorBitPosition - 1);
32      }
33
34      int indexToRemove = 1 << (numberOfRedundancyBitsAdded - 1);
35      for (int i = 0; i < numberOfRedundancyBitsAdded; i++) {
36          encodedMessage.removeBit(indexToRemove - 1);
37          indexToRemove >>= 1;
38      }
39  }
```

La fonction `numberOfRedundancyBitsAdded` permet de calculer le nombre de bits de redondances contenu dans le message. Nous avons vu dans la section 2.5 que $r = log_2(n + 1)$, la fonction `BitUtil.binLog()` permet de calculer le logarithme binaire d'un entier. Il faut noter que cette fonction ne fonctionne que pour des codes de Hamming parfaits, c'est-à-dire des codes tels que $n + 1 = 2^r$, avec $r \geq 2$.

La fonction `decode` va décoder le message en trois étapes. Elle va premièrement vérifier si chaque bit de redondance possède une valeur cohérente, dans un second temps, elle va corriger un bit du message si nécessaire et enfin, elle va retirer tous les bits de redondances.

La boucle for à la ligne 11, va pour chaque bit de redondance, compter le nombre d'index de bits du message possédant le bit à l'index `bitPosition` valant 1. Pour

mieux comprendre cette phrase, voyons un exemple complet pour un passage d'une boucle :

Pour un message donné $m = 1010010$, lors du deuxième passage dans la boucle, avec $i = 1$, `bitPosition` obtient alors la valeur $1 << 1 = 2$, la variable `numberOfOneForBitPosition` est initialisée à 0. La boucle `for` à la ligne 15 va effectuer un passage complet sur chaque bit du message. Pour rappel, les deux premiers bits du message sont des bits de redondances, c'est pourquoi la variable j est initialisée à 2. Si un bit du message codé vaut un (`bitPosition` & (j + 1) != 0) et que cet index n'est pas une puissance de deux (`BitUtil.isNotPowerOfTwo(j + 1)`) alors la variable `numberOfOneForBitPosition` est incrémentée de un, il est nécessaire de vérifier que cet index n'est pas une puissance de 2, car les bits situés à ces index sont des bits de redondances. Il ne faut donc pas prendre en compte ces bits.

Une fois tous les bits du message parcourus, il est alors nécessaire de vérifier la validité du bit de redondance à l'index `bitPosition`. Si la variable `numberOfOneForBitPosition` est impaire, alors le bit de redondance doit être égal à 1.

Maintenant que la boucle à la ligne 11 est terminée, tous les bits de redondances ont été parcourus. On peut alors modifier le bit du message corrompu si nécessaire. La variable `errorBitPosition` va nous renseigner la position du bit erronée, si cette variable est égale à 0, alors aucune erreur n'a été détectée dans le message.

La dernière étape consiste à retirer tous les bits de redondances du message. Il est primordial de retirer tous les bits de redondances en commençant par les bits de redondances ayant le poids le plus fort en priorité, c'est pourquoi la variable `indexToRemove` est initialisée dans un premier temps avec le bit de redondance ayant le poids le plus fort. Après chaque passage dans la boucle, cet index est mis à jour en décalant les bits de cet index de 1 cran vers la droite, nous permettant de réduire la puissance de l'exposant de cet index (2^r, avec $r \geq 0$).

4 Démonstration de l'application

Nous allons dans un premier temps voir un exemple d'exécution pour chaque fonctionnalité décrite dans la section 3.2, et dans un second temps nous allons comparer les différents taux de détections/corrections des erreurs en fonction du code utilisé, ainsi que des paramètres fournis à ces codes. Les différentes options pour chaque fonctionnalité sont renseignées dans le fichier `readme.md` de l'application.

4.1 Exemple d'utilisation des fonctionnalités

4.1.1 Encoder un message

Voici un exemple permettant d'encoder un message en utilisant les codes cycliques (CRC), avec le polynôme générateur $P_g = x^2 + 1$ et le message $m = 101110$.

```
cd-codes encode -C CRC -M 101110 -GP 101
```

Les paramètres passés en entrée de l'application permettent :

- **-C** (code) : fonction de codage utilisée pour la fonctionnalité demandée, dans ce cas-ci, CRC fait référence aux codes cycliques (CRC) vus à la section 2.3.
- **-M** (message) : le message qui doit être encodé par l'application, seule la représentation binaire est autorisée.
- **-GP** (generator polynomial) : le polynôme générateur qui doit être utilisé pour encoder le message, seule la représentation binaire est autorisée.

La console renvoie la valeur '10111011' correspondant au message codé.

4.1.2 Décoder un message

Voici un exemple permettant de décoder un message en utilisant les codes cycliques (CRC), avec le polynôme générateur $P_g = x^2 + 1$ et le message codé $m = 10111011$.

```
cd-codes decode -C CRC -M 10111011 -GP 101
```

La console renvoie la valeur '101110' correspondant au message décodé.

4.1.3 Générer un message

Voici un exemple permettant de générer un message de 10 bits.

```
cd-codes generateMessage -MBS 10
```

Le paramètre `MBS` (message bit size) permet de renseigner la taille du message, si celui-ci est omis, la taille d'un message par défaut est initialisée à 8 bits. La console renvoie la valeur '0010010110' correspondant au message généré.

4.1.4 Corrompre un message

Voici un exemple permettant de corrompre le message $m = 0010010110$.

```
cd-codes corruptMessage -M 0010010110 -E burstError -BL 4 -P 0.2
```

Les paramètres passés en entrée de l'application permettent :

- **-E** (error) : le modèle d'erreur utilisé pour altérer le message, différents modèles sont proposés par l'application :
  - **constantError** : de manière aléatoire (isolée), chaque bit a une probabilité p égale d'être altéré, ce modèle est choisi par défaut si l'option -E est omise.
  - **burstError** : en rafale, pour une séquence contiguë de m bits, le premier et le dernier bit sont erronés, chaque bit entre le deuxième et l'avant-dernier bit a une probabilité p d'être altéré.
- **-BL** (burst length) : lorsque le modèle `burstError` est choisi par l'utilisateur, cette option permet de définir la taille de la rafale.
- **-P** (probability) : la probabilité pour chaque bit d'être altéré.

La commande va donc corrompre le message $m = 0010010110$ avec une probabilité pour chaque bit d'être altéré est de 20%, le modèle d'erreur correspond à un modèle d'erreur en rafale, avec une longueur de la rafale de 4 bits. Une fois la commande lancée, la console renvoie la valeur '1011010110' correspondant au message corrompu.

4.1.5 Graphe montrant le taux de détection/correction des erreurs

La fonctionnalité `generateErrorDetectingRateGraph` permet de générer un graphe montrant l'évolution du taux de détection et de correction (si le code le permet) des erreurs en fonction de la probabilité pour chaque bit d'être altéré pour un code donné, un affichage console est également généré afin de permettre à l'utilisateur de récupérer les données.

L'application va générer s points (via le paramètre **-S** initialisé par défaut à 50) sur un graphe en deux dimensions, l'axe des abscisses représente la probabilité pour chaque bit d'être altéré, l'axe des ordonnées représente le taux de détection/correction des erreurs pour un code donné. L'utilisateur fournit deux paramètres permettant de définir la plage de probabilité avec laquelle le code est testé, ceux-ci sont renseignés via **-MinP** (minimum probability) et **-MaxP** (maximum probability). Avant de générer les points, l'application calcule l'écart de probabilité nécessaire entre la génération de chaque point.

Par exemple, pour les paramètres `-S` 5, `-MinP` 0.01, `-MaxP` 0.1, l'écart de probabilité sera (0.1 - 0.01) / (5 - 1) = 0.0225, ce qui nous donne une probabilité pour chaque bit d'être altéré pour la génération de chaque point de :

Point	1	2	3	4	5
Probabilité	0.01	0.0325	0.055	0.0775	0.1

Une fois l'écart de probabilité obtenu, l'application peut calculer le taux de détection et de correction d'une erreur. Pour ce faire, l'application va itérer i fois (via le paramètre `-I` initialisé par défaut à 10 000) sur une boucle, cette boucle va effectuer les opérations suivantes dans l'ordre :

1. Génère un message de taille k (via le paramètre `-MBS` message bit size).
2. Encode le message généré en utilisant la fonction `encode` du code passé en paramètre (via le paramètre `-C`).
3. Corrompt le message avec une certaine probabilité passée en paramètre et avec un modèle d'erreur passé en paramètre (via le paramètre `-E` vu précédemment).
4. Vérifie si le message a bien été corrompu, si il ne l'est pas, les étapes qui suivent ne sont pas effectuées.
5. Vérifie si la fonction `isCorrupted` du code passé en paramètre arrive à correctement détecter l'erreur, une variable est mise à jour si l'erreur du message est correctement détectée.
6. Si la fonction de codage passée en paramètre permet de corriger les erreurs, alors la fonction `decode` de celle-ci est appelée, le message décodé est ensuite comparé au message initialement généré, une variable est mise à jour si le message est correctement corrigé.

Une fois les i itérations terminées, la fonction peut calculer le taux de détection/correction d'erreur. Voici un exemple à la figure 37, de génération d'un graphe montrant l'évolution du taux de détection d'erreurs en fonction de la probabilité pour chaque bit d'être altéré.

```
cd-codes generateErrorDetectingRateGraph -C parityBit
```

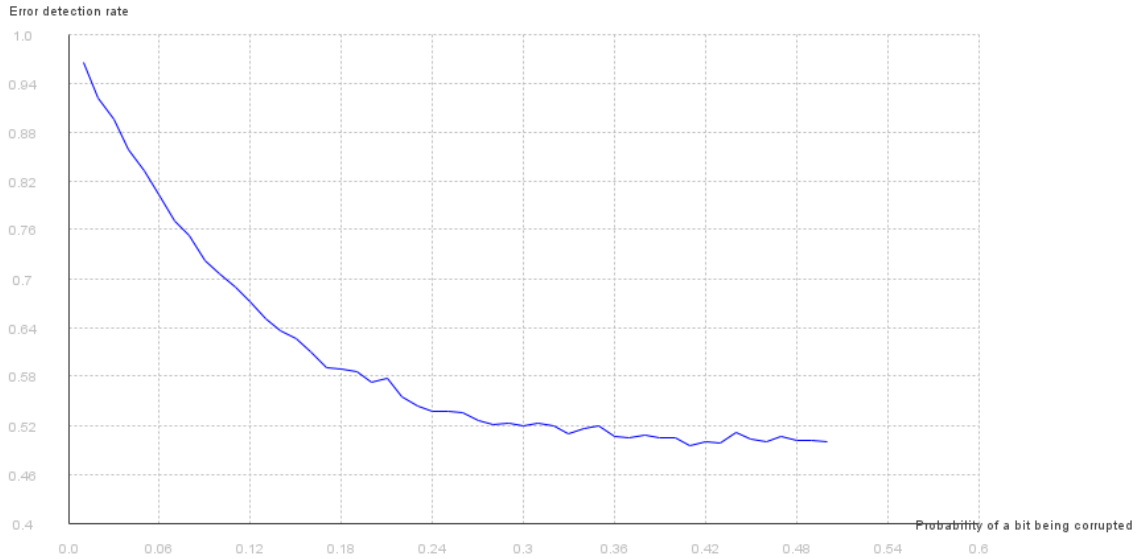


Figure 37: Évolution du taux de détection d'erreurs en fonction de la probabilité pour chaque bit d'être altéré.

La console renvoie également 2 tableaux de nombres décimaux, le premier tableau contient la liste des probabilités pour chaque bit d'être altéré allant de 0.01 à 0.5, le second tableau contient la liste des taux de détections des erreurs par le code à bit de parité, si le code pouvait corriger les erreurs, alors un troisième tableau aurait été généré contenant la liste des taux de correction des erreurs.

La console affiche :
$[0.01, 0.02, 0.03, ..., 0.5]$
$[0.94, 0.90, 0.85, ..., 0.5]$

4.2 Comparaison du taux de détection/correction des erreurs

4.2.1 Bit de parité

Nous constatons à la figure 37 que lorsque la probabilité pour chaque bit d'être altéré est trop élevée, le taux de détection d'erreurs est de 50%, ce qui est tout à fait logique, puisque dans la section 2.2, nous avons vu que le code à bit de parité détecte un nombre impair d'erreur uniquement. Regardons à présent comment évolue le taux de détection des erreurs en fonction de la taille des messages. Dans les figures 38 et 39, le taux d'erreur utilisé correspond au taux d'erreur présent dans de l'ADSL (10^{-3} à 10^{-9}), la taille des messages est de respectivement 8 bits et 1000 bits, on constate que l'augmentation de la taille des messages réduit considérablement les performances du code à bit de parité.

```
cd-codes generateErrorDetectingRateGraph -C parityBit
-MaxP 0.001 -MinP 0.000001 -I 100000 -MinBY 0.67 -MaxBY 1
```

Les paramètres passés en entrée de l'application permettent :

- **-MinP** (minimum probability) : la probabilité minimum pour chaque bit d'être altéré, ce paramètre va de pair avec le paramètre `-MaxP`, les deux paramètres combinés permettent d'obtenir la plage de probabilité avec laquelle une fonction de codage sera testée.
- **-MaxP** (maximum probability) : la probabilité maximum pour chaque bit d'être altéré.
- **-I** (iterations) : le nombre d'itérations par point généré sur le graphe.
- **-MinBY** (minimum bound y) : la valeur minimale sur l'axe des ordonnées. Ce paramètre va de pair avec le paramètre `-MaxBY`, si un des deux paramètres est spécifié, alors l'échelle sur l'axe des ordonnées sera définie par ces paramètres, sinon cette échelle est définie automatiquement par l'application.
- **-MaxBY** (maximum bound y) : la valeur maximale sur l'axe des ordonnées.

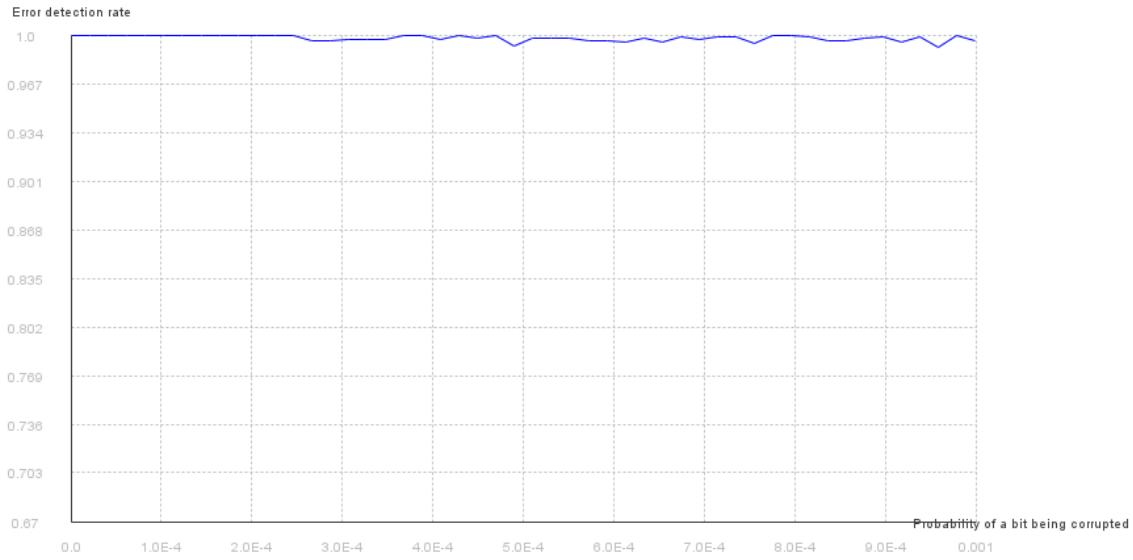


Figure 38: Évolution du taux de détection d'erreurs en fonction de la probabilité pour chaque bit d'être altéré.

```
cd-codes generateErrorDetectingRateGraph -C parityBit
-MaxP 0.001 -MinP 0.000001 -MBS 1000 -MinBY 0.67 -MaxBY 1
```

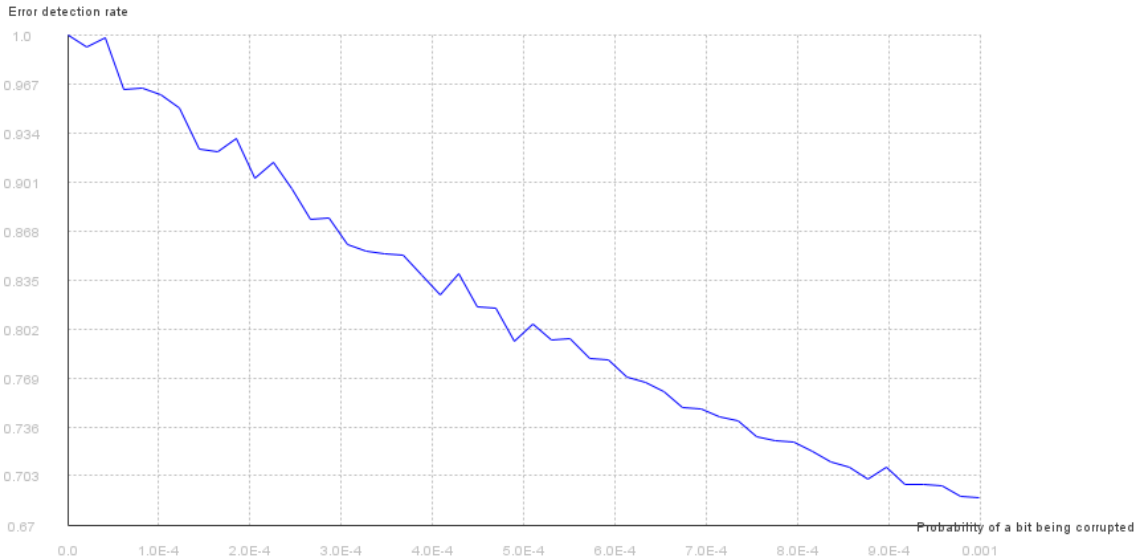


Figure 39: Évolution du taux de détection d'erreurs en fonction de la probabilité pour chaque bit d'être altéré.

4.2.2 Codes cycliques (CRC)

Passons maintenant aux codes cycliques (CRC). Dans la figure 40, le taux d'erreur utilisé correspond au taux d'erreur présent dans de l'ADSL (10^{-3} à 10^{-9}), les messages non codés ont une taille de 1000 bits, le modèle d'erreur est en rafale, la taille de la rafale porte sur 2 bits consécutifs, le polynôme générateur utilisé est $P_g = x^3$, dans un second exemple, le graphe à la figure 41 possède exactement les mêmes configurations, à l'exception du polynôme générateur qui est $P_g = x^3 + 1$. Ces figures mettent en valeur la propriété vue à la section 2.3 nous indiquant qu'un polynôme générateur possédant le monôme +1 peut détecter toutes erreurs en rafale portant sur $r - 1$ bits, où r est le degré du polynôme générateur, c'est pourquoi il possède un meilleur taux de détection des erreurs.

```
cd-codes generateErrorDetectingRateGraph -C CRC -MBS 1000
  -GP 1000 -MaxP 0.001 -MinP 0.000001 -E burstError -BL 2
  -I 100000 -MinBY 0.999 -MaxBY 1
```

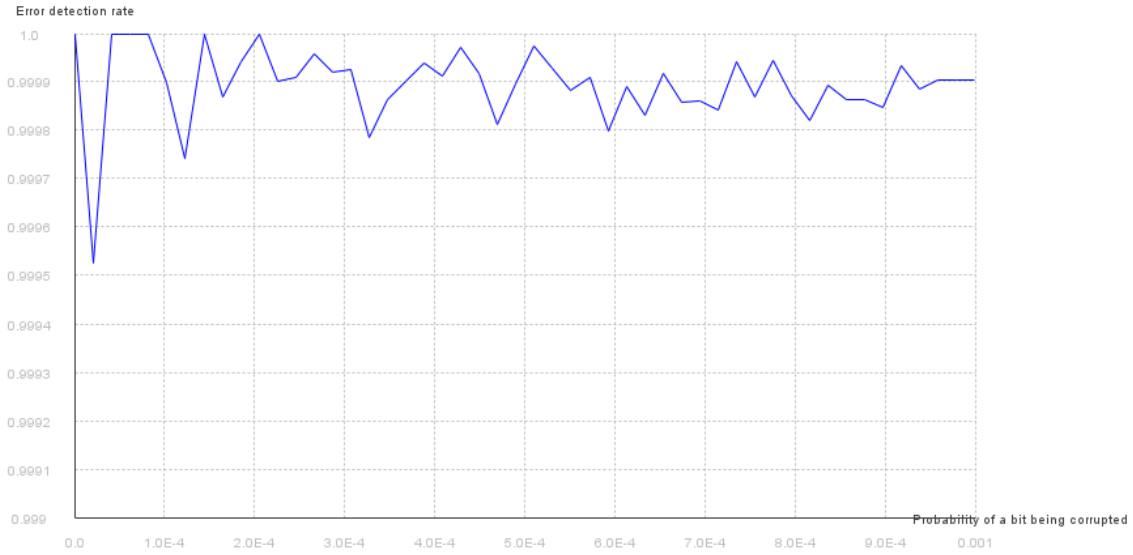


Figure 40: Évolution du taux de détection d'erreurs des CRC en fonction de la probabilité pour chaque bit d'être altéré ne détectant pas les erreurs en rafale.

```
cd-codes generateErrorDetectingRateGraph -C CRC -MBS 1000
  -GP 1001 -MaxP 0.001 -MinP 0.000001 -E burstError -BL 2
  -I 100000 -MinBY 0.999 -MaxBY 1
```

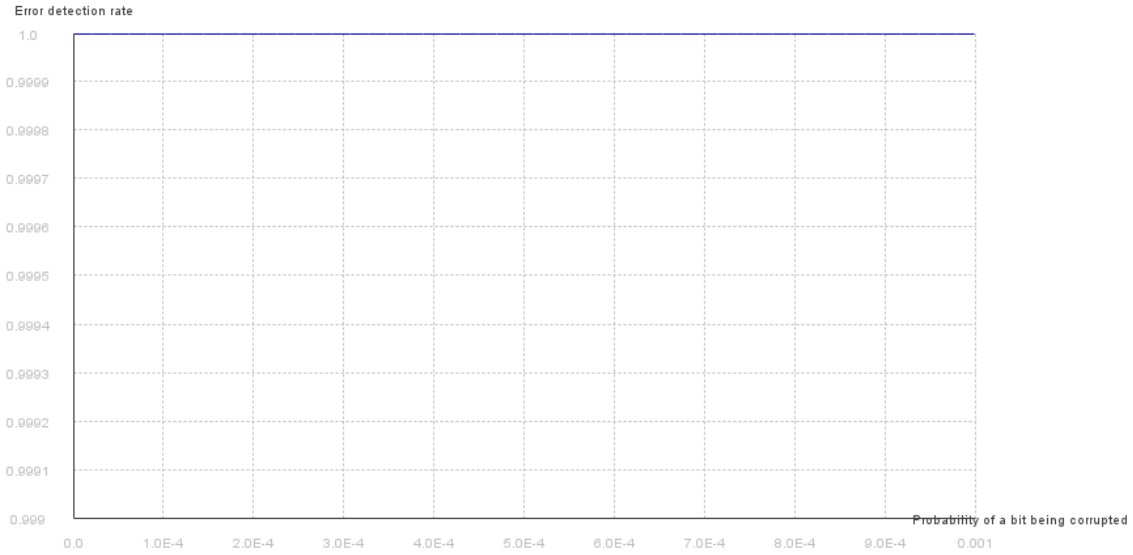


Figure 41: Évolution du taux de détection d'erreurs des CRC en fonction de la probabilité pour chaque bit d'être altéré détectant maximum 2 erreurs en rafale.

4.2.3 Somme de contrôle Internet (RFC 1071)

Le taux d'erreur utilisé à la figure 42 correspond au taux d'erreur présent dans de l'ADSL (10^{-3} à 10^{-9}), les messages non codés ont une taille de 992 bits. Pour rappel, la somme de contrôle Internet codent des messages dont leur taille est un multiple de 16.

```
cd-codes generateErrorDetectingRateGraph -C internetChecksum -MBS 992
-MaxP 0.001 -MinP 0.000001 -S 100 -MinBY 0.5 -MaxBY 1
```

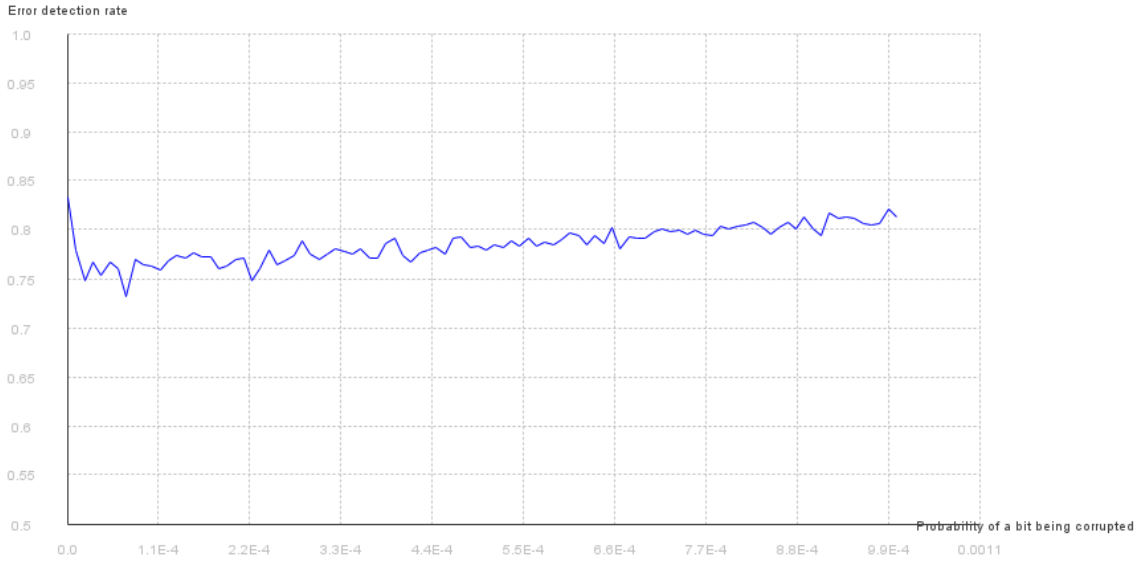


Figure 42: Évolution du taux de détection d'erreurs de la somme de contrôle Internet (RFC 1071) en fonction de la probabilité pour chaque bit d'être altéré.

Nous avons vu à la section 2.4 que la somme de contrôle Internet (RFC 1071) détecte les erreurs en rafale portant sur au plus 15 bits, regardons deux exemples illustrant cette propriété. La figure 43 montre un cas où la taille de la rafale porte sur 15 bits, tandis que la taille de la rafale sur la figure 44 porte sur 17 bits, nous constatons que les performances de la fonction de codage sont médiocres et la ligne polygonale représentée dans la figure est fortement similaire à la ligne polygonale de la figure 42 lorsque la taille de la rafale porte sur 17 bits. Cependant le taux de détection d'erreurs n'est pas parfait lorsque la rafale porte sur 15 bits, ce qui est dû à des rafales multiples portant sur les mêmes indices de bit de blocs différents si nous coupons le message en bloc de 16 bits.

```
cd-codes generateErrorDetectingRateGraph -C internetChecksum -MBS 992
-E burstError -BL 15 -MaxP 0.001 -MinP 0.000001 -S 100
-MinBY 0.5 -MaxBY 1
```

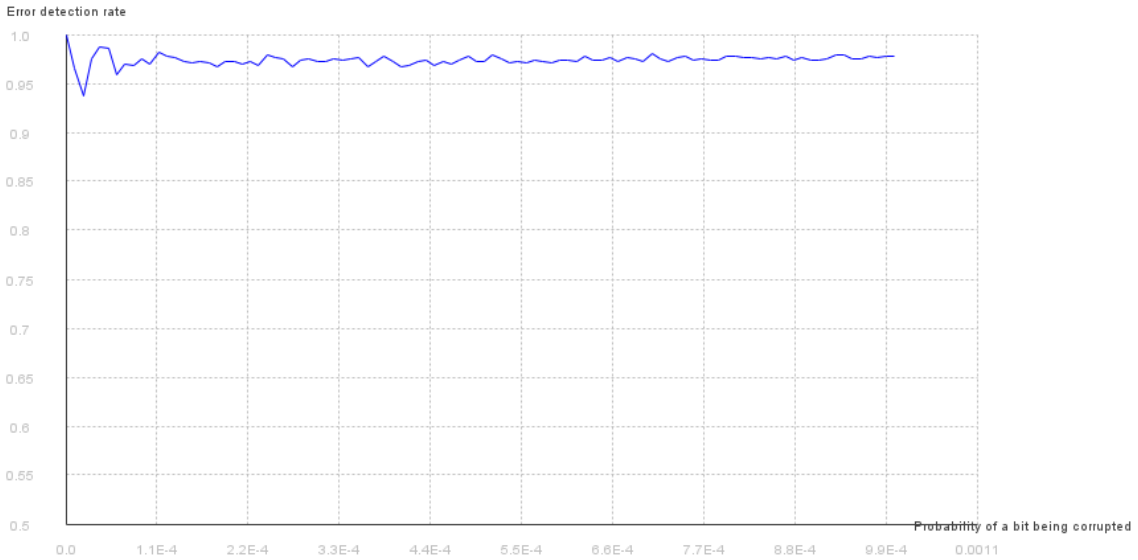


Figure 43: Évolution du taux de détection d'erreurs de la somme de contrôle Internet (RFC 1071) en fonction de la probabilité pour chaque bit d'être altéré.

```
cd-codes generateErrorDetectingRateGraph -C internetChecksum -MBS 992
-E burstError -BL 17 -MaxP 0.001 -MinP 0.000001 -S 100
-MinBY 0.5 -MaxBY 1
```

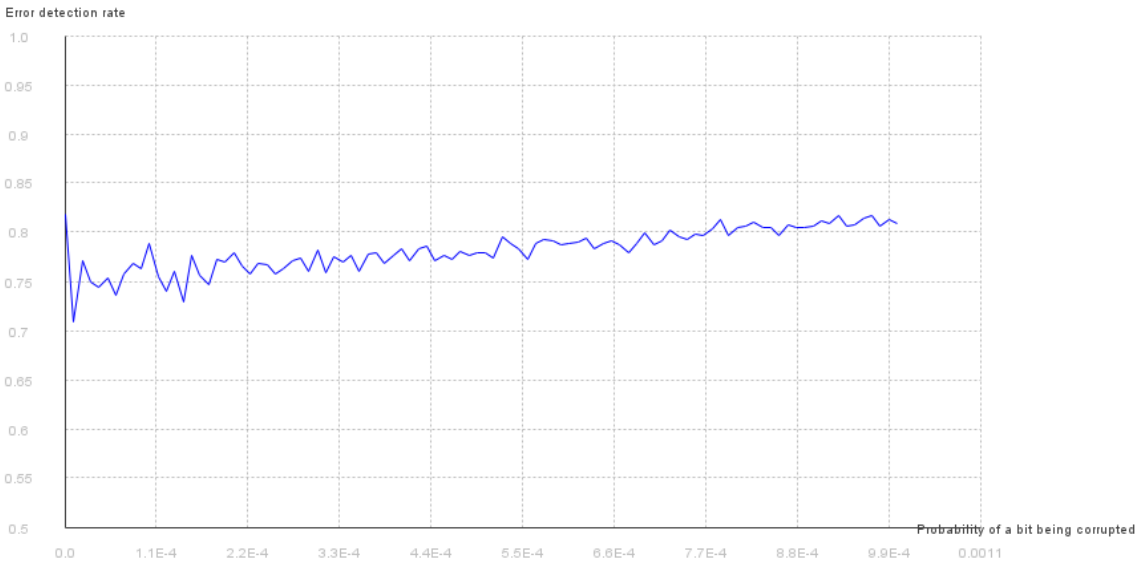


Figure 44: Évolution du taux de détection d'erreurs de la somme de contrôle Internet (RFC 1071) en fonction de la probabilité pour chaque bit d'être altéré.

4.2.4 Codes de Hamming

Nous avons vu également dans la section 2.5, que les codes de Hamming ne sont pas efficaces contre les erreurs en rafale puisqu'ils sont 1-correcteur, les figures 45 et 46 permettent de se rendre compte de ce fait, le taux d'erreurs utilisé correspond à l'ADSL (10^{-3} à 10^{-9}), le code utilisé est Hamming(247, 255). Si le premier bit altéré est le dernier bit du message, alors il n'y a pas d'autres bits qui peuvent être erronés. Ce cas de figure explique le fait que le taux de correction ne soit pas constamment à zéro.

```
cd-codes generateErrorDetectingRateGraph -C hamming
  -MBS 247 -MaxP 0.001 -MinP 0.000001 -MinBY 0 -MaxBY 1
```

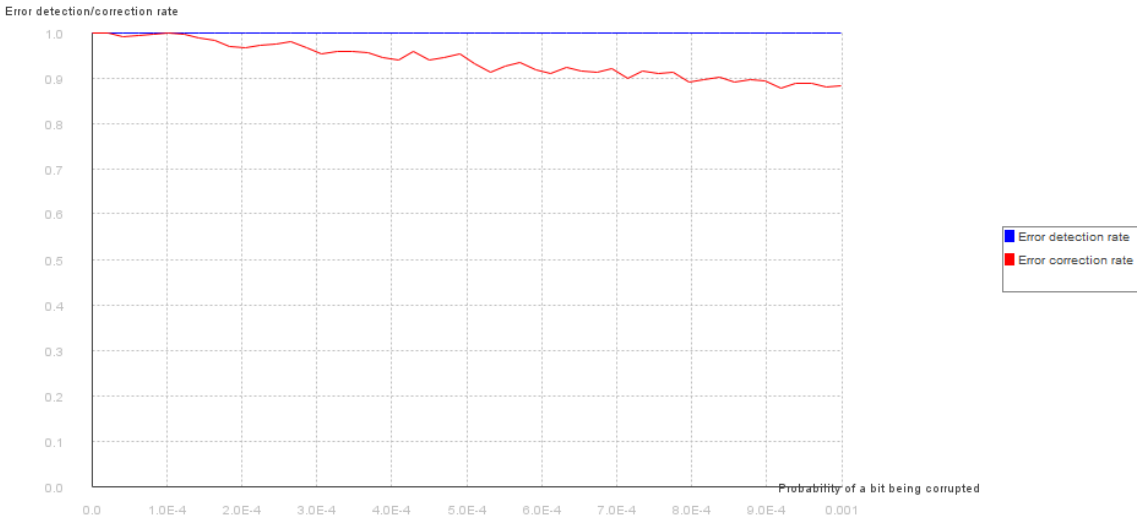


Figure 45: Évolution du taux de détection d'erreurs du code de Hamming(247, 255) en fonction de la probabilité pour chaque bit d'être altéré.

```
cd-codes generateErrorDetectingRateGraph -C hamming
  -MBS 247 -MaxP 0.001 -MinP 0.000001 -E burstError
```

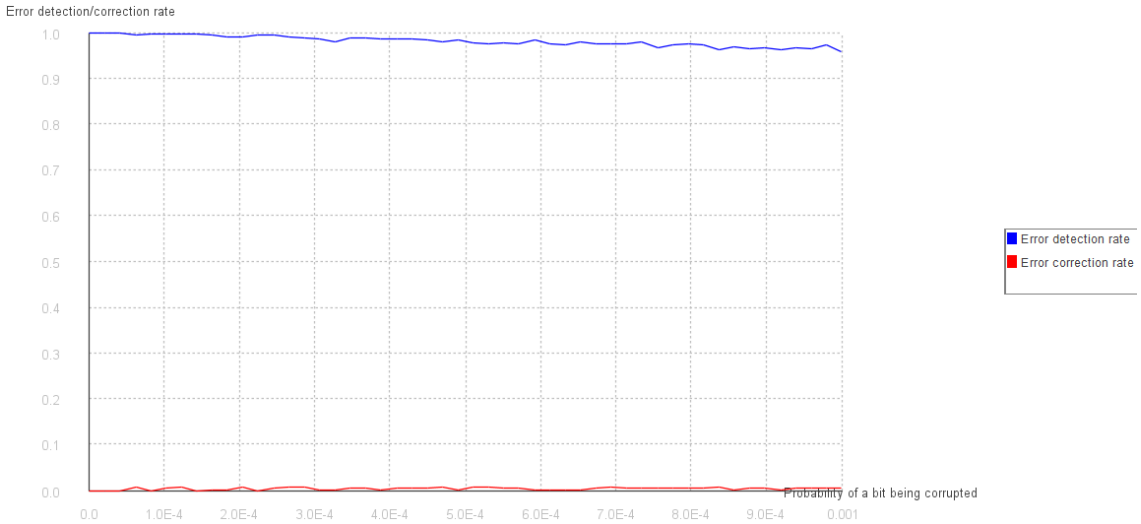


Figure 46: Évolution du taux de détection d'erreurs du code de Hamming(247, 255) en fonction de la probabilité pour chaque bit d'être altéré, avec un modèle d'erreurs en rafale.

Regardons à présent à la figure 47 ce qui se passe lorsque nous diminuons la taille d'un message. On constate que le taux de détection et de correction des erreurs est bien meilleur lorsque la taille des messages diminue, il faut noter que la figure ne possède pas les mêmes échelles que pour les figures 45 et 46 afin de pouvoir distinguer plus aisément que les deux taux affichés ne sont pas parfaits.

```
cd-codes generateErrorDetectingRateGraph -C hamming
  -MBS 4 -MaxP 0.001 -MinP 0.000001 -I 100000
```

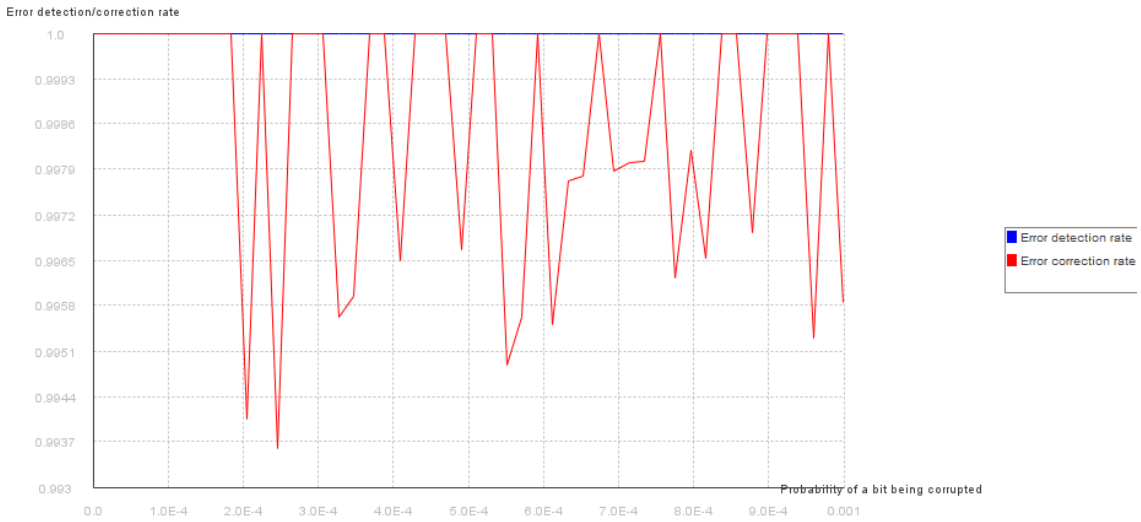


Figure 47: Évolution du taux de détection d'erreurs du code de Hamming(4, 7) en fonction de la probabilité pour chaque bit d'être altéré.

En comparant tous les taux de détections et de corrections des erreurs selon la fonction de codage utilisée, nous constatons globalement que la taille des messages va influencer les différents taux. Les codes cycliques (CRC) sont les seuls à avoir un taux de détection de l'erreur parfait pour des messages de grande taille (1000 dans les exemples vus précédemment), même si il est possible en pratique qu'il y ait des erreurs, il faut prendre en compte le fait que les tests n'ont pas été effectués en lançant des milliards d'itérations, il faut donc prendre du recul concernant les résultats. De plus, les codes cycliques (CRC) gèrent les erreurs en rafale portant sur un nombre plus important de bits selon le polynôme générateur utilisé, puisque la taille des rafales qu'ils peuvent gérer dépend du degré du polynôme générateur utilisé (sachant que les polynômes générateurs les plus utilisés sont de degré 32).

5 Conclusion

Dans ce mémoire, nous avons vu pourquoi il était nécessaire de pouvoir construire des codes permettant de transformer l'information afin de détecter et/ou corriger les erreurs, que ceux-ci doivent avoir un rendement le plus élevé possible et qu'ils doivent encoder et décoder de l'information rapidement.

Nous avons vu le code a bit de parité, celui-ci permet d'introduire le concept de code détecteur, mais il fournit en pratique une capacité de détection médiocre par rapport à d'autres codes. Nous avons ensuite vu les codes cycliques (CRC), lorsque le polynôme générateur est bien choisi, il offre une très bonne capacité de détection, il permet également de détecter les erreurs qui surviennent en rafale. Nous avons également vu la somme de contrôle Internet (RFC 1071), celle-ci est calculée très rapidement grâce à des propriétés mathématiques intéressantes, elle détecte également les erreurs en rafale, cependant, nous avons vu que sa capacité de détection n'est pas suffisante pour se permettre d'utiliser uniquement ce code lors de l'encodage et du décodage de l'information, mais ce code est utilisé en combinaison avec les codes cycliques (CRC). Nous avons enfin vu les codes de Hamming, ceux-ci permettent de corriger une erreur, cependant ils sont 1-correcteur et par conséquent ils sont inefficaces face aux erreurs en rafale.

Par après, nous avons vu que la structure de données en Java `BigInt` est plus appropriée que le `BigInteger` ainsi que le `StringBuilder` pour stocker les informations. Nous avons vu une implémentation possible en Java pour ces différents codes.

Nous avons finalement vu les fonctionnalités d'une application fournie permettant de tester ces différents codes, ainsi qu'une démonstration de l'application avec des exemples montrant certaines propriétés.

De multiples améliorations peuvent être apportées afin de compléter ce mémoire et d'améliorer l'application fournie, les améliorations suivantes peuvent être prises en considération :

1. Une implémentation hardware pour les fonctions de codage : la littérature contient de nombreux articles proposant des implémentations hardware de ces fonctions, rendant leur exécution plus rapide qu'une implémentation software (si leur fonctionnement est similaire).

2. Utiliser un autre langage pour l'application : l'application a été développée en Java, ce langage apporte des avantages mais il ne permet pas d'avoir les meilleures performances. Un langage permettant plus d'optimisation tel que le C ou le C++ peut être envisagé.

3. Optimisation des implémentations : les différentes fonctions de codages implémentées peuvent être améliorées, notamment pour les CRC qui possède une implémentation très simple, il existe d'autres algorithmes proposés dans la littérature plus optimisés permettant d'atteindre le même résultat.

4. Une implémentation pour d'autres fonctions de codage : ce mémoire et l'application ne décrivent que quatre fonctions de codage, il existe d'autres fonctions de codages utilisées qui sont intéressantes telles que les codes de Reed-Solomon par exemple.

6 Bibliographie

References

[1] Tsonka Baicheva and Faiza Sallam. "CRC codes for error control". In: *Mathematica Balkanica* 21 (2007), pp. 377–387.

[2] Robert Braden, David Borman, and Craig Partridge. "Computing the internet checksum". In: *ACM SIGCOMM Computer Communication Review* 19.2 (1989), pp. 86–94.

[3] Jean-Guillaume Dumas et al. *Théorie des codes-3e éd.: Compression, cryptage, correction*. Dunod, 2018.

[4] Simon Klein. *Github Huldra bigint project*. `https://github.com/bwakell/Huldra`. Accessed online on March 23, 2023.

[5] Tenkasi V Ramabadran and Sunil S Gaitonde. "A tutorial on CRC computations". In: *IEEE micro* 8.4 (1988), pp. 62–75.

[6] Yann Richet. *JMathPlot: interactive 2D and 3D plots*. `https://github.com/yannrichet/jmathplot`. Accessed online on May 27, 2023.

[7] Dilip V. Sarwate. "Computation of cyclic redundancy checks via table lookup". In: *Communications of the ACM* 31.8 (1988), pp. 1008–1013.

[8] Claude Elwood Shannon. "A mathematical theory of communication". In: *The Bell system technical journal* 27.3 (1948), pp. 379–423.

[9] Christophe Silon. *Detecting and correcting codes*. `https://github.com/Sun8642/detecting_and_correcting_codes`. Accessed online on May 27, 2023.

[10] Christophe Silon. *Detecting and correcting codes: benchmark*. `https://github.com/Sun8642/detecting_and_correcting_codes_performances`. Accessed online on May 27, 2023.

[11] Jonathan Stone and Craig Partridge. "When the CRC and TCP checksum disagree". In: *ACM SIGCOMM computer communication review* 30.4 (2000), pp. 309–319.

[12] Jonathan Stone et al. "Performance of checksums and CRCs over real data". In: *IEEE/ACM Transactions on Networking* 6.5 (1998), pp. 529–543.