Etude d'un code correcteur linéaire pour le canal à effacements de paquets et optimisation par comptage de forêts et calcul modulaire

Antoine Roux
sous la direction de
Michèle Soria et Laurent Frèrebeau

20 novembre 2019

THÊSE
pour obtenir le titre de
Docteur en Sciences mention Informatique

Rapporteurs

Ayoub Otmani	*University of Rouen Normandy*
Christophe Crespelle	*University of Bergen*

Jury de thêse

Ayoub Otmani	*University of Rouen Normandy*
Christophe Crespelle	*University of Bergen*
Michel Habib	*Université Paris Diderot*
Annick Valibouze	*Sorbonne Université*
Laurent Frèrebeau	*Thales Communications and Security*
Hervé Delpeyrat	*Thales Communications and Security*
Michèle Soria	*Sorbonne Université*
Binh-Minh Bui-Xuan	*Sorbonne Université*

Encadrement

Binh-Minh Bui-Xuan	*Sorbonne Université*
Hervé Delpeyrat	*Thales Communications and Security*

Résumé

La transmission fiable de données sur un canal de transmission est un problème récurrent en Informatique. En effet, quel que soit le canal de transmission employé, on observe obligatoirement de la détérioration de l'information transmise, voire sa perte pure et simple. Afin de palier à ce problème, plusieurs solutions ont été apportées, notamment via l'emploi de codes correcteurs.

Dans cette thèse, nous étudions un code correcteur développé en 2014 et 2015 pour l'entreprise Thales durant ma deuxième année de Master en apprentissage. Il s'agit d'un code actuellement utilisé par Thales pour fiabiliser une transmission UDP passant par un dispositif réseau, l'Elips-SD. L'Elips-SD est une diode réseau qu'on place sur une fibre optique et qui garantit physiquement que la transmission est unidirectionnelle. Le cas d'utilisation principal de cette diode est de permettre le suivi de la production d'un site sensible, ou encore de superviser son fonctionnement, tout en garantissant à ce site une protection face aux intrusions extérieures. A l'opposé, un autre cas d'utilisation est la transmission de données depuis un ou plusieurs sites non-sécurisés vers un site sécurisé, dont on souhaite s'assurer qu'aucune information ne pourra par la suite fuiter.

Le code correcteur que nous étudions est un code correcteur linéaire pour le canal à effacements de paquets, qui a reçu la certification OTAN de la Direction Générale des Armées. Nous l'avons babtisé "Fauxtraut", anagramme de "Fast algorithm using Xor to repair altered unidirectionnal transmissions". Afin d'étudier ce code correcteur, de présenter son fonctionnement et ses performances, et les différentes modifications apportées durant cette thèse, nous établissons tout d'abord un état de l'art des codes correcteurs, en nous concentrant principalement sur les codes linéaires non-MDS, tels que les codes LDPC. Puis nous présentons le fonctionnement de Fauxtraut, et analysons son comportement (complexité, consommation mémoire, performances) par la théorie et par des simulations. Enfin, nous présenterons différentes versions de ce code correcteur développées durant cette thèse, qui aboutissent à d'autres cas d'utilisation, tels que la transmission d'information sur un canal unidirectionnel à erreurs ou sur un canal bidirectionnel, à l'image de ce que permet de faire le protocole H-ARQ.

Dans cette partie, nous étudierons notamment le comportement de notre code correcteur via la théorie des graphes : calculer la probabilité de décoder convenablement ou non revient à connaître la probabilité d'apparition de cycles dans le sous-graphe de graphes particuliers, les graphes de Rook et les graphes bipartis complets. Le problème s'énonce simplement et s'avère compliqué, et nous espérons qu'il saura intéresser des chercheurs du domaine. Nous présentons une méthode permettant de calculer exactement cette probabilité pour de petits graphes (qui aboutit à un certain nombre de formules closes), et une fonction tendant asymptotiquement vers cette probabilité pour de plus grands graphes.

Nous étudierons aussi la manière de paramétrer automatiquement notre code correcteur par le calcul modulaire et la combinatoire, utilisant la fonction de Landau, qui retourne un

ensemble de nombres entiers dont la somme est fixée et le plus commun multiple est maximal.

Dans une dernière partie, nous présentons un travail effectué durant cette thèse ayant conduit à une publication dans la revue Theoretical Computer Science. Il concerne un problème non-polynomial de la théorie des graphes : le couplage maximal dans les graphes temporels. Cet article propose notamment deux algorithmes de complexité polynomiale : un algorithme de 2-approximation et un algorithme de kernelisation pour ce problème. L'algorithme de 2-approximation peut notamment être utilisé de manière incrémentale : arêtes du flot de liens nous parviennent les unes après les autres, et on construit la 2-approximation au fur et à mesure de leur arrivée.

Abstract

Reliably transmitting information over a transmission channel is a recurrent problem in Informatic Sciences. Whatever may be the channel used to transmit information, we automatically observe erasure of this information, or pure loss. Different solutions can be used to solve this problem, using forward error correction codes is one of them.

In this thesis, we study a corrector code developped in 2014 and 2015 for Thales society during my second year of master of apprenticeship. It is currently used to ensure the reliability of a transmission based on the UDP protocole, and passing by a network diode, Elips-SD. Elip-SD is an optical diode that can be plugged on an optical fiber to physically ensure that the transmission is unidirectional. The main usecase of such a diode is to enable supervising a critical site, while ensuring that no information can be transmitted to this site. At the opposite, another usecase is the transmission from one or multiple unsecured emitters to one secured receiver who wants to ensure that no information can be robbed.

The corrector code that we present is a linear corrector code for the binary erasure channel using packets, that obtained the NATO certification from the DGA ("Direction Générale de Armées" in French). We named it Fauxtraut, for "Fast algorithm using Xor to repair altered unidirectional transmissions". In order to study this code, presenting how it works, its performance and the modifications we added during this thesis, we first establish a state of the art of forward error correction, focusing on non-MDS linear codes such as LDPC codes. Then we present Fauxtraut behavior, and analyse it theorically and with simulations. Finally, we present different versions of this code that were developped during this thesis, leading to other usecases such as transmitting reliable information that can be altered instead of being erased, or on a bidirectionnal channel, such as the H-ARQ protocole, and different results on the number of cycles in particular graphs.

In the last part, we present results that we obtained during this thesis and that finally lead to an article in the Technical Computer Science. It concerns a non-polynomial problema of Graphs theorie : maximum matching in temporal graphs. In this article, we propose two algorithms with polynomial complexity : a 2-approximation algorithm and a kernelisation algorithm for this problema.

Remerciements

Je tiens tout d'abord à remercier mes deux encadrants, Binh-Minh Bui-Xuan et Hervé Delpeyrat, ainsi que mes deux directeurs de thèse, Michèle Soria et Laurent Frèrebeau, qui ont su me guider depuis ma deuxième année de Master et durant cette thèse et sans lesquels celle-ci n'aurait pas été possible.

Je remercie chaleureusement Christophe Crespelle et Ayoub Otmani d'avoir accepté de rapporter cette thèse, et pour leurs remarques pertinentes et bienveillantes qui m'ont permis de parfaire ce manuscrit. Je remercie aussi Madame Annick Valibouze et Monsieur Michel Habib d'avoir accepté de faire partie de mon jury de thèse et d'assister à ma soutenance.

Merci aussi à l'entreprise Thales, et tout particulièrement à l'équipe SLS de Thales Communications and Security, qui a rendu cette thèse possible, et m'a accompagné durant la fin de mon Master. Je remercie notamment Pascal, Nicolas, Renaud, François, Alice, les Benjamin, Laurent, Manu, Hugo, Thibaut, Pierre-Arthur, Wandrille, Aurélien, Assia, Sebastien.

Merci à toute l'équipe APR qui m'a accueilli chaleureusement et avec laquelle il a été très agréable de travailler : Emmanuel, Pascal, Maryse, Annick, Romain, Frédéric, Jean-Pierre, les (nombreux) Antoine, Cristoph, Maximilien, Thuy, Ha-Duong.

Un grand merci aussi aux nombreux thésards, post-doc' et stagiaires que j'ai pu rencontrer durant ces 3 années, Ghiles, Vincent, Alice, Steven, Boubacar, Yi-Ting, les Ma(t)thieu, Marie, Clément, Hugo, Alexandre, Sandou, Raphael, Martin, Pierre, Julien et bien d'autres.

J'embrasse chaleureusement mes amis et à famille, et mon père Ivan auquel je dédie cette thèse.

Table des matières

Liste des symboles

3GPP 3rd Generation Partnership Project

AL-FEC Application-Level Forward Error Correction

ANSSI Agence Nationale des Systèmes d'Information

BCH Bose, Ray-Chaudhuri et Hocquenghem (codes)

BEC Binary Erasure Channel

BSC Binary Symmetric Channel

CD Confidentiel Défense

CRC Cyclic Redundancy Code

DGA Direction Générale des Armées

DVB Digital Video Broadcasting

Elips-SD Elips Security-Diode

Fauxtraut Fast Algorithm Using Xor To Repair Altered Unidirectionnal Transmission

GF Galois Field

HARQ Hybrid Automatic Repeat Request

IETF Internet Engineering Task Force

IP Internet Protocol

LDPC Low Density Parity Check

LT Luby Transform

MDS Maximum Distance Separable

MIT Massachusetts Institute of Technology

ML Maximum Likelihood

OEIS On-Line Encyclopedia of Integer Sequences

OSI Open Systems Interconnection

OTAN Organisation du Traité de l'atlantique Nord

PoC Proof of Concept

PPCM Plus petit commun multiple

PRNG Pseudo Random Number Generator

RFC Request for Comments

RSD Robust Soliton Distribution

SD Secret Défense

TCP Transmission Control Protocol

TCS Thales Communications Security

TSD Très Secret Défense

UDP User Datagram Protocol

XOR Exclusive Or

Table des figures

Liste des tableaux

Introduction

L'information en elle-même est immatérielle et doit être stockée sur un support d'information afin d'être manipulée. Il existe de nombreux types de supports d'information : papier, tablette d'argile, ADN, CD-rom, disque dur, ayant pour point commun de se détériorer avec le temps. Pour conserver l'information, une technique consiste à ajouter de l'information supplémentaire sur le support. Se pose alors la question : une tablette d'argile n'étant pas extensible, comment ajouter le moins d'information supplémentaire tout en assurant la plus longue longévité ? Grâce à la théorie des *codes correcteurs*, nous savons aujourd'hui que recopier toute la tablette n'est ni la seule ni la meilleure solution.

Dans la législation concernant la gestion de l'information classifiée au sein de la défense Française, il existe trois niveaux de confidentialité définis par le "Code de la Défense" : Confidentiel Défense, Secret Défense et Très-Secret Défense.

Très-Secret-Défense (TSD) : Il est "réservé aux informations ou supports protégés dont la divulgation est de nature à nuire très gravement à la Défense nationale et qui concernent les priorités gouvernementales en matière de défense."

Secret-Défense (SD) : Il est "réservé aux informations ou supports protégés dont la divulgation est de nature à nuire gravement à la Défense nationale".

Confidentiel-Défense (CD) : Il est "réservé aux informations ou supports protégés dont la divulgation est de nature à nuire à la Défense nationale ou pourrait conduire à la découverte d'un secret de la Défense nationale classé au niveau Très Secret-Défense ou Secret-Défense".

Un document "CD" peut se voir attribuer le statut "SD", et un document "SD" peut se voir attribuer le statut "TSD". Lorsqu'il est jugé nécessaire pour un document "CD" de passer au niveau de confidentialité "SD", celui-ci est transmis à une autre autorité, qui devra se charger de sa protection (notamment via, par exemple, l'utilisation d'une cage de Faraday). La transmission de ce document, d'un lieu (au sens géographique) de niveau de confidentialité "CD" à un lieu de niveau de confidentialité "SD" est critique : on peut faire "entrer" de l'information dans le lieu "SD", mais aucune information "SD" ne doit en sortir.

Afin de s'assurer qu'aucune information "SD" ne fuite, il est possible d'utiliser un pare-feu, un support physique (comme une clé USB) sécurisé, ou de le transmettre sur un canal de transmission unidirectionnel. C'est cette dernière option qui est explorée via l'Elips-SD de Thales. L'Elips-SD est une diode, à savoir un dispositif laissant passer la lumière dans un sens et jamais dans l'autre. Les documents confidentiels sont transmis sur un canal rendu unidirectionnel par l'utilisation d'une fibre optique sur laquelle on a préalablement installé la diode.

Les différentes méthodes et mesures clées permettant d'assurer la sécurité de systèmes d'informations cruciaux tels que des sites industriels, sont étudiées dans ce document [6] rédigé par l'. L'ANSSI est l'autorité nationale en matière de sécurité et de défense des systèmes d'information, créée par décret en 2009 et placé sous l'autorité du Premiers Ministre. Le principe de fonctionnement d'une diode réseau, et de ses deux guichets (émetteur et récepteur) est détaillée de ce document de l'ANSSI [7]. Les cas d'utilisation de l'Elips-SD ne sont pas uniquement militaires : on pourra l'utiliser, par exemple, pour récupérer de l'information d'une centrale nucléaire (comme la température du cœur du réacteur en temps-réel, ou de la vidéo-surveillance), tout en s'assurant qu'il est impossible d'envoyer de l'information dans cette centrale nucléaire (notamment pour modifier des paramètres nécessaires à son bon fonctionnement). Il faut aussi noter que dans le cas d'utilisation d'une diode, nous forçons volontairement l'unidirectionnalité de la transmission pour des raisons précises, mais il existe de nombreux cas d'utilisation pour lesquelles l'unidirectionnalité est subie. Å

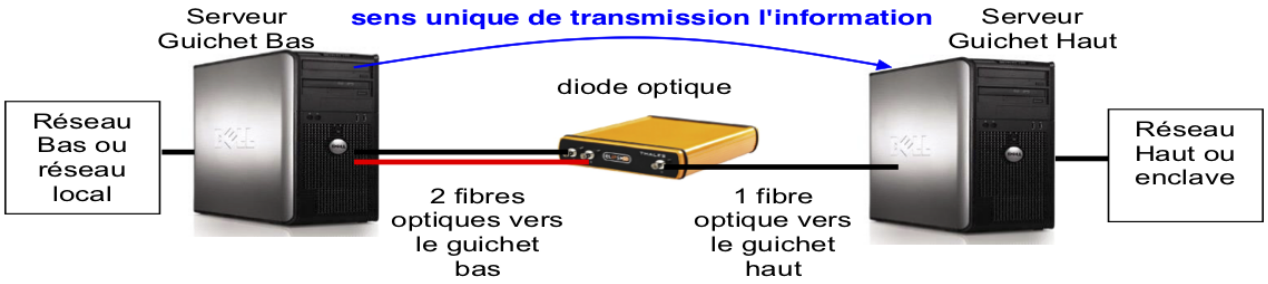


FIGURE 1 – Transmission depuis un système non-sécurisé vers un système sécurisé via la diode Elips-SD. Le cas d'utilisation est par exemple l'envoi d'informations depuis un théâtre d'opération vers un QG sécurisé, tout en s'assurant que ce QG ne pourra pas subir de vol d'information.

Cette diode, qui a notamment reçu les certifications Secret OTAN et Secret Défense Français, est un dispositif performant dont le principe impose un certain nombre de contraintes. En effet, elle interdit physiquement le retour d'information du récepteur à l'émetteur, et il n'est donc pas possible d'utiliser de système d'acquittement et de non-acquittement de paquets. En d'autres

termes, il est impossible d'utiliser des protocoles certifiant la transmission sûre d'un paquet, tels que TCP . Le protocole de transmission de la diode repose sur l'utilisation des paquets UDP , un protocole de transmission unidirectionnel sans acquittement. La diode peut soutenir des débits élevés de l'ordre du gigabit/seconde, et le bruit sur la fibre est relativement faible : la qualité de la transmission au niveau du signal est bonne, et la perte d'information provient le plus souvent d'une perte de paquets coté récepteur.

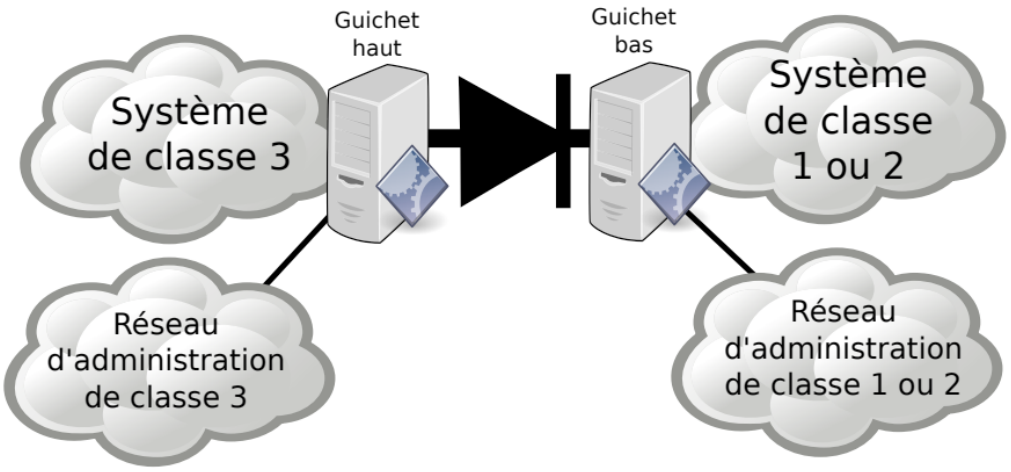


FIGURE 2 – Transmission depuis un système sécurisé vers un système non-sécurisé via une diode réseau. Le cas d'utilisation est par exemple l'envoi d'informations depuis un QG sécurisé à des récepteurs en opérations extérieures, tout en s'assurant qu'aucune information malveillante ne peut être transmise au QG.

C'est au niveau de la couche logicielle que la reconstitution de l'information s'effectue, via l'utilisation d'une famille de codes correcteurs que nous présentons et analysons dans cette thèse. Il s'agit de codes à effacements de paquets, linéaires et systématiques, que nous avons développés chez Thales Communication Security (TCS) durant mon année d'apprentissage en Master 2, et qui se sont avérés performants dans le cas d'une transmission via la diode Elips-SD. Nous l'avons informellement surnommé Fauxtraut , pour "Fast Algorithm Using Xor To Repair Altered Unidirectional Transmissions". C'est un code relativement simple s'apparentant à un entrelaceur de paquets, qui se comporte comme un code rectangulaire pour certaines longueurs et dimensions.

Le manuscrit que nous présentons est construit de la manière suivante : dans un premier temps, nous présentons un état de l'art des codes correcteurs, en nous concentrant sur les codes à effacements linéaires, notamment les codes LDPC et les codes sans rendement. Nous décrivons les différentes métriques permettant de comparer ces différents codes, que ce soit au niveau de leur efficacité de correction, leur complexité de calcul, ou encore leur consommation mémoire. Nous décrirons les différentes méthodes utilisées pour étudier ces codes correcteurs, regroupées en deux catégories : méthodes analytiques et simulations.

Dans un second temps, nous présenterons le code correcteur Fauxtraut, qui a évolué durant ces 3 années de thèse. Nous nous intéresserons à son fonctionnement, aux différentes métriques qui lui sont associées, telles que sa distance minimale, sa complexité en calcul et sa consommation mémoire. Ce code correcteur pouvant être adapté à différents taux d'effacements, en

contrepartie d'un temps de calcul plus ou moins important, nous présenterons les différentes manières de l'utiliser et de le paramétrer. Nous confronterons ces analyses numériques à des simulations en conditions réelles d'utilisation.

Dans un troisième temps, nous présenterons les différentes recherches menées durant cette thèse, concernant notamment la théorie des graphes et le calcul modulaire. Il s'avère que la probabilité de décoder ou non grâce à notre code correcteur est équivalente à la probabilité que le sous graphe d'un graphe biparti soit une forêt : si un sous-graphe contient un cycle, on appelle ce sous-graphe un "Stopping-set" du graphe de Tanner associé à ce code. Ainsi, nous présenterons un certain nombre de résultats concernant la théorie des graphes, notamment les formules closes calculant le nombre de sous-graphes de taille e d'un graphe de Rook qui contiennent au moins un cycle (le nombre de Stopping-sets de taille e de notre code). Nous proposerons aussi des bornes inférieure et supérieure pour de plus grandes valeurs de e, les formules closes devenant de plus en plus difficiles à trouver à mesure que e progresse.

Ce code nous impose de choisir un ensemble de nombres premiers entre eux : la dimension maximale du code dépend de la somme des éléments de cet ensemble, sa longueur maximale dépend du *plus petit commun multiple* de cet ensemble, et les temps de calcul de l'encodage et du décodage sont proportionnels au cardinal de cet ensemble . Ainsi, nous présenterons l'algorithme de Marc Deléglise, Jean-Louis Nicolas et Paul Zimmerman, permettant de calculer rapidement l'ensemble dont le PPCM est maximal pour une somme donnée. Nous présenterons comment il est possible d'utiliser la fonction de Landau pour paramétrer automatiquement notre code correcteur, ainsi que les limitations qu'elle présente.

Ce code reposant sur l'utilisation de nombres premiers entre eux, nous présenterons aussi une autre version de celui-ci adaptée aux transmissions à erreurs, reposant sur l'utilisation du *Théorème des Restes Chinois* afin de déterminer les indices des paquets ayant subi une erreur, avant de les décoder.

Nous avons de plus adapté ce code au protocole de transmission HARQ [16], qui consiste à utiliser un protocole de transmission fiable, type *TCP* (reposant sur un retour d'information du récepteur à l'émetteur) couplé à un code correcteur. Nous présenterons une méthode permettant de calculer le plus petit ensemble de paquets à redemander parmi les paquets que nous ne parvenons pas à décoder, qui une fois reçus nous permettent de reprendre le décodage. Cela garantit au récepteur, dans le cadre d'un protocole de type HARQ utilisant ce code, de ne redemander que la plus petite quantité d'information lui permettant de terminer le décodage. Cette méthode repose notamment sur le calcul d'une forêt couvrante sur le graphe de Tanner de notre code.

Enfin, toujours concernant la théorie des graphes, nous présenterons un problème lié aux graphes temporels : le γ-couplage maximal. Un graphe temporel est un graphe dont les arêtes peuvent apparaître ou disparaître au cours du temps. Ainsi, les arêtes sont de la forme (u, v, t), avec u et v les sommets du graphes, et t l'instant auquel cette arête a existé. Lorsqu'il existe dans un tel graphe une suite de γ arêtes consécutives $(u, v, t), (u, v, t + 1), \ldots, (u, v, t + \gamma - 1)$, on dit alors qu'il existe une γ-arête à l'instant t. Le problème du couplage maximal consiste alors à trouver le plus grand sous-ensemble de γ-arêtes du graphe temporel tel qu'aucune de ces gamma-arêtes ne se chevauchent. Un cas d'application de ce problème que l'on pourrait rencontrer dans la vie réelle est par exemple celui d'une entreprise : les employés sont présents par intermittence, et peuvent donc travailler en binôme certains jours et d'autres non. Supposons qu'il faille γ jours consécutifs pour qu'une tâche soit réalisée par un binôme, et que les membres du binôme ne puissent se accomplir qu'une tâche à la fois : quel est le nombre maximal de tâches

que l'entreprise peut accomplir ? C'est un problème NP-complet difficile à résoudre de manière optimale en temps raisonnable, et nous proposons donc deux algorithmes : un algorithme de 2-approximation et un algorithme de kernelisation, tous deux ayant une complexité quasi-linéaire. Le premier permet de trouver un γ-couplage dont la taille est au pire la moitié du γ-couplage maximal du graphe. Le deuxième calcule une instance du graphe temporel plus petite (ou de même taille) que le graphe temporel original, et qui contient à coup-sûr un γ-matching maximal du graphe original. Nous proposons une implémentation de ces deux algorithmes, ainsi que des simulations de ces derniers sur trois types de graphes temporels : le graphe temporel "Rollernet" et le graphe temporel "Enron", bien connus dans ce domaine de recherche, ainsi que des graphes générés avec une méthode "maison", dans laquelle nous simulons le déplacement de particules sur une surface. Ces recherches ont notamment abouti à une publication dans le journal Theoretical Computer Science en 2019.

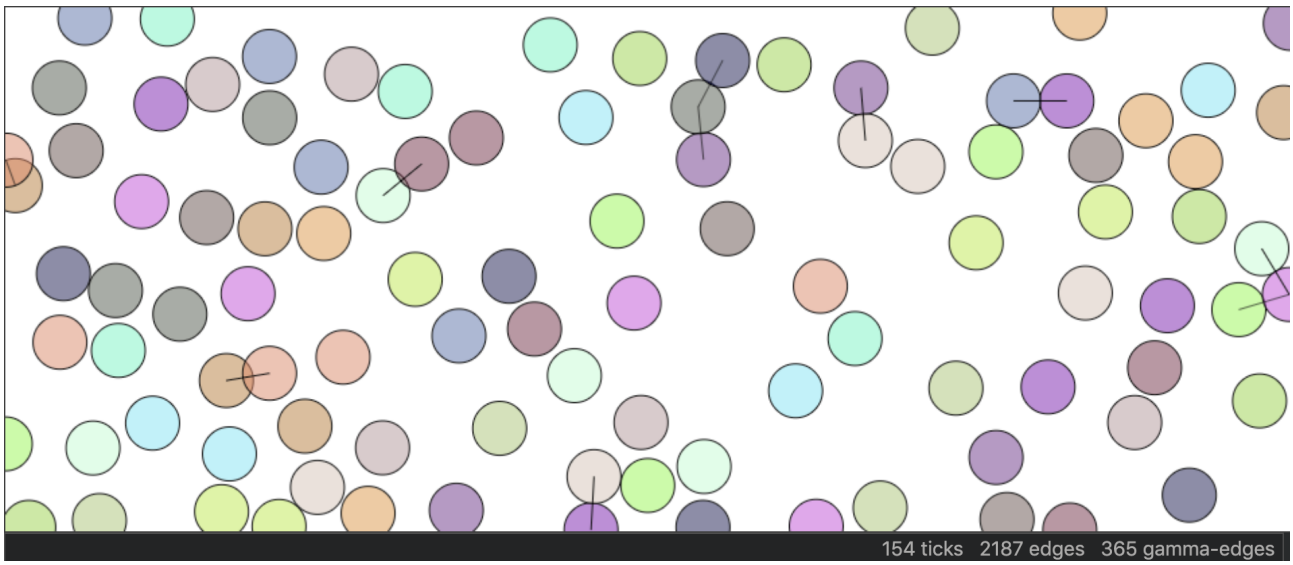


FIGURE 3 – Nous proposons un outil permettant de générer des graphes temporels aléatoires, en simulant le déplacement de particules sur un plan. Les particules sont regroupées en groupes, et il existe une arête entre deux groupes u et v à l'instant t si deux particules de ces deux groupes sont en contact à cet instant. Cet outil est disponible à l'url suivante : https ://antoinedimitriroux.github.io

Chapitre 1

Transmissions et codes correcteurs

1.1 Généralités

1.1.1 Canal de communication

Un canal de communication est un support permettant la transmission d'une certaine quantité d'information, depuis une *source* (ou *émetteur*) vers *une destination* (ou *récepteur*). Il peut être physique, comme un fil électrique ou une fibre optique, ou logique, comme lors d'une transmission par paquets sur un canal multiplexé. Un canal transmet des symboles (ou lettres) regroupés en messages (ou mots).

Un message constitué de k symboles est dit de *taille* k. L'ensemble des symboles à disposition d'un canal est appelé un *alphabet*. Quand celui-ci est fini, on parle alors de canal *discret*, en opposition aux canaux *continus*. En ingénierie réseau, on utilise plutôt les termes de *paquets* et de *blocs de paquets* plutôt que *symboles* et *mots*.

On peut définir un canal de communication comme une application dont les entrées ont leurs valeurs dans un alphabet A et les sorties leurs valeurs dans un alphabet B. Cette application est munie d'une loi de transition déterminant la probabilité d'obtenir le symbole Y à la sortie du canal en fonction de l'entrée X, notée $P(Y|X)$. C'est l'observation du canal qui nous permet d'obtenir cette loi.

Definition 1. *La loi de transition d'un canal à valeurs d'entrée dans A et de sortie dans B est l'ensemble des lois de probabilité $P(Y|X)$ pour tout couple de symboles $(X, Y) \in A * B$.*

Shannon introduit la notion d'*entropie* de l'information, qui représente intuitivement la quantité d'information contenue ou délivrée par une source. Du point de vue du récepteur, plus l'émetteur émet d'informations différentes, plus l'entropie de la source est importante : source qui enverrait toujours le même symbole a alors une entropie nulle. L'entropie d'une source d'information est égale au nombre de bits qu'il faut pour coder cette information.

Definition 2. *On appelle entropie de X, notée $H(X)$, la fonction mesurant l'incertitude sur la variable X à valeurs dans l'alphabet $A = \{x_i\}_{i=0\dots n}$ qui s'exprime de la manière suivante :*

$$H(X) = -\sum_{i=0}^{n} P(X = x_i) \log_2 \big(P(X = x_i)\big),$$

Un symbole X délivré par une source (par exemple un canal de communication) qui émet des bits $\in \{0, 1\}$ de manière équiprobable a donc une entropie

$$H(X) = -\frac{1}{2} * \log_2(\frac{1}{2}) - \frac{1}{2} * \log_2(\frac{1}{2}) = 1$$

et un symbole X délivrée par une source (par exemple un canal de communication) qui émet des octets $\in \{0, ..., 255\}$ de façon équiprobable

$$H(X) = -256 * \frac{1}{256} * \log_2(\frac{1}{256}) = 8.$$

En français, l'alphabet contient 27 lettres, et une lettre tirée au hasard dans un texte devrait avoir une entropie de $\log_2(27) = 4.75$ si les lettres apparaissaient de manière équiprobable. Cependant, le e apparaît bien plus souvent que le k par exemple, et on obtient une entropie d'environ 4.

Le but du récepteur est de déterminer la valeur du message X envoyé en fonction de la valeur du message Y reçu. On considère alors la notion d'*information mutuelle* de deux sources d'information, qui représente la quantité d'information qu'apporte une source sur l'autre.

Definition 3. *L'information mutuelle de deux sources d'information X et Y notée $I(X, Y)$ est la quantité d'information que l'on peut obtenir sur X en connaissant Y. Elle vaut*

$$I(X, Y) = H(X) - H(X|Y)$$

Dans le pire cas, le canal de communication est totalement bruité, les valeurs de X et Y sont totalement décorrélées, et on a $H(X|Y) = H(X)$. L'information mutuelle $I(X, Y)$ est nulle, et il est impossible de transmettre quoi que ce soit sur le canal. Dans le cas d'un canal de communication parfait, la valeur de X est entièrement déterminée par la valeur de Y, on a $H(X|Y) = 0$, et donc $I(X, Y) = H(X)$. En général on se retrouve dans une situation intermédiaire, et il devient alors intéressant de définir la *capacité d'un canal.*

Definition 4. *On appelle la capacité C d'un canal la valeur*

$$C = max_X\{I(X, Y)\},$$

max_X étant le maximum parmi toutes les lois de probabilité définies sur A.

Un canal parfait a une capacité maximale $C = max_X(H(X))$, et le pire canal a une capacité nulle, $C = max_X(0) = 0$.

Un canal de transmission n'est capable de transmettre qu'un nombre limité de symboles par unité de temps. On parle alors du débit du canal.

Definition 5. *Le débit D d'un canal est le nombre de symboles $N(T)$ transitant sur ce canal pendant une durée T, et vaut*

$$D = N(T)/T$$

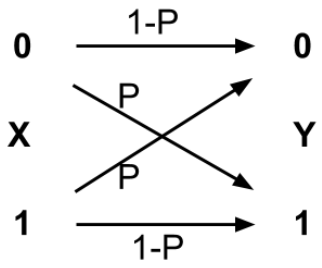


FIGURE 1.1 – Canal binaire symétrique.

Les notions présentées précédemment nous permettent d'étudier un canal de communication simple : *le canal binaire symétrique*, aussi noté BSC . Il prend en entrée et en sortie des symboles ayant leurs valeurs dans $\{0, 1\}$, et est caractérisé par un paramètre p correspondant à la probabilité d'erreur d'un symbole (ou bit) sur ce canal. Sa loi de transition est dite *symétrique*, ce qui signifie que la probabilité d'erreur est la même que X vaille à ou 1 :

$$\begin{aligned} P(Y=0|X=0) &= 1-p \\ P(Y=0|X=1) &= p \\ P(Y=1|X=1) &= 1-p \\ P(Y=1|X=0) &= p \end{aligned}$$

Si la probabilité d'erreur d'un canal BSC est nulle, sa capacité C_{BSC} vaut alors 1, ce qui est aussi le cas quand la probabilité d'erreur vaut 1 (on sait que tous les bits ont été erronés, et on les inverse). Si la probabilité d'erreur est de 0.5, la capacité du canal est nulle : c'est le pire cas possible.

1.1.2 Canal binaire à effacements

Le modèle du canal binaire à effacements (CBE) ou "Binary Erasure Channel" (BEC) en anglais est introduit par Peter Elias en 1955 dans [45]. Les symboles qui transitent sur ce type de canal ne subissent pas d'erreurs mais sont tout simplement effacés. Si la probabilité d'effacement vaut p, on dit que le canal est de paramètre p. Pour modéliser ce canal, on ajoute souvent le symbole $\emptyset$ à l'alphabet des symboles de sortie. Tout comme le canal binaire symétrique, il suit une loi de transition :

$$\begin{aligned} P(Y=0|X=0) &= 1-p \\ P(Y=1|X=0) &= 0 \\ P(Y=1|X=1) &= 1-p \\ P(Y=0|X=1) &= 0 \\ P(Y=\emptyset|X=0) &= p \\ P(Y=\emptyset|X=1) &= p \end{aligned}$$

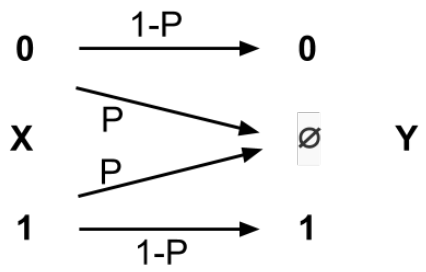


FIGURE 1.2 – Canal binaire à effacements

La capacité C_{CBE} de ce canal vaut $C_{CBE} = max_X((1-p) * H(X)) = 1-p$. On remarque que pour une probabilité d'erreur fixée, la capacité du canal binaire à effacements est meilleure que celle du canal binaire symétrique, égale à $(1-H(p))$, puisque $\forall p \in [0,1], p \leq H(p)$. En effet sur un CBE l'emplacement des effacements est connu. Il reste à corriger les effacements, alors que sur un canal binaire symétrique les erreurs sont à des positions inconnues, ce qui rend la correction plus difficile.

1.1.3 Canal à effacements de paquets

En général, les effacements ne concernent pas seulement quelques bits, mais plutôt des ensembles de bits, pouvant atteindre plusieurs Giga-octets. Il convient alors de présenter un nouveau un modèle de canal, le *canal à effacements de paquets*. Sur ce canal, on ne considère plus des mots de 1 unique bit, mais des groupements de bits : des *paquets*.

Sur un tel canal, soit les paquets sont transmis sans altération, soit ils sont totalement effacés. L'intégrité du paquet est le plus souvent garantie par un système de détection d'erreur sur l'ensemble de celui-ci grâce à un CRC. Un CRC est notamment utilisé pour vérifier l'intégrité des paquets des protocoles TCP et UDP, tels que décrit dans [47], ou encore le CRC (pour Cyclic Redundancy Code) du protocole Ethernet décrit dans [1]. Le calcul d'un CRC est comparable au calcul d'une fonction de hashage, tel que la fonction de hashage MD5 inventée par Ronald Rivest en 1991 dans [50]. Ce modèle de canal permet de décrire le comportement de nombreux systèmes dans lesquels de l'information peut être sujette à effacements. Les paquets peuvent être effacés pour deux raisons principales : soit ils sont totalement effacés, soit ils ont subi trop d'erreurs pour qu'on puisse retrouver l'information originale, et on les considère comme effacés en choisissant des les ignorer. Dans le protocole UDP par exemple, un paquet subissant une altération sur un unique bit est considéré comme erroné.

Exemple 1 : transmissions unidirectionnelles par paquets Les systèmes de communication modernes suivent une organisation en couches (modèle OSI). Une couche a pour but de fournir un services aux couches supérieures, en fonction d'un protocole particulier. Au sein de ces couches, les données sont regroupées au sein de paquets, qui sont alors traités indépendamment les uns des autres. Ces paquets peuvent être altérés ou effacés durant leur transmission.

— on dit d'un paquet qu'il est effacé lorsqu'il n'arrive pas à destination, par exemple à cause d'erreurs de routage sur le réseau. Un paquet peut aussi être effacé à cause de l'encombrement d'un nœud du système (le buffer d'un routeur est plein par exemple),

— un paquet est altéré si un ou plusieurs bits de ses subissent une modification. Ces altérations peuvent provenir de perturbations sur le signal transmis sur le canal. Si aucun code correcteur n'est utilisé pour réparer, il est alors considéré comme erroné.

Le protocole *UDP* (pour *User Datagram Protocol*) est un des principaux protocoles de télécommunication utilisés par Internet. Il fait partie de la couche transport du modèle OSI, quatrième couche de ce modèle, comme *TCP*. Il a été défini par David P. Reed et est détaillé dans la RFC 768. Le rôle de ce protocole est de permettre la transmission de données (sous forme de paquets) de manière simple entre deux entités, chacune étant définie par une adresse IP et un numéro de port. L'intégrité des données est assurée par une somme de contrôle. Un paquet UDP est constitué d'un entête (ou header en anglais), et de ses données. L'entête contient le numéro de port source, le numéro de port destinataire, la longueur des données, et une somme de contrôle. La *somme de contrôle* (ou *checksum* en Anglais) est un nombre qu'on ajoute à un message à transmettre permettant au récepteur de vérifier que le message envoyé et le message reçu sont bien les mêmes. C'est une forme de contrôle par *redondance*, qui permet de détecter une ou plusieurs erreurs susceptibles d'apparaître dans un message, mais pas de connaître leurs positions dans ce message, et donc de les corriger. Une somme de contrôle étant généralement plus petite que le message qu'elle accompagne, l'ensemble des valeurs qu'elle peut prendre est aussi bien plus petit que celui des messages. Par conséquent, plusieurs messages partagent la même somme de contrôle, et des erreurs de type "faux-positif" peuvent advenir, ce qui fera passer un message erroné pour un message valide aux yeux du récepteur. La somme de contrôle inclut également les adresses IP de la source et de la destination. Pour être transmis et acheminé au destinataire sur le réseau, un paquet UDP doit être encapsulé dans un paquet IP. La figure 1.3 représente l'entête d'un paquet UDP encapsule dans un paquet IP.

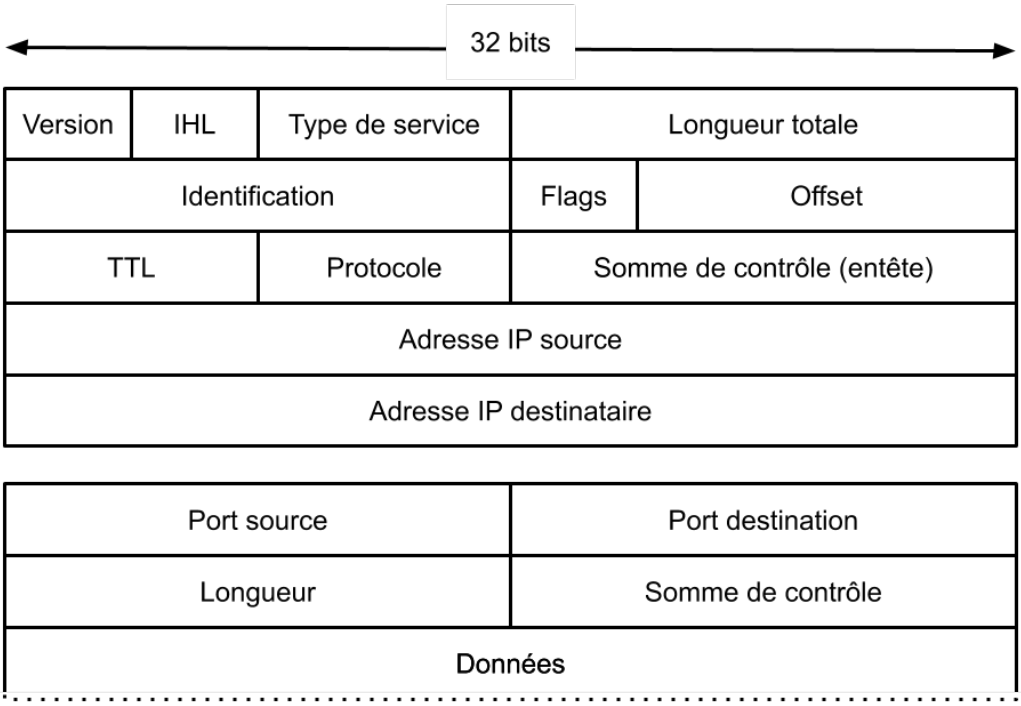


Figure 1.3 – Paquet UDP encapsulé dans un paquet IP

Le protocole *TCP* (pour *Transmission Control Protocol*) décrit dans [48], développé en 1973 puis adopté par Arpanet en 1983, est un protocole de transport fiable documenté dans la RFC 7931 de l'IETF. C'est un protocole de transmission en mode *connecté*, ce qui signifie qu'il repose sur l'établissement d'une connexion (*handshacking*) entre les deux entités préalablement à l'envoi de toute donnée utile. Il permet la transmission fiable d'informations sur un canal de transmission bruité, en permettant au destinataire de demander la retransmission d'éventuels paquets erronés ou effacés, via notamment l'envoi de paquets *ACK* ou *NACK*. Dans le modèle OSI, il correspond à la couche transport, intermédiaire de la couche réseau et de la couche session.

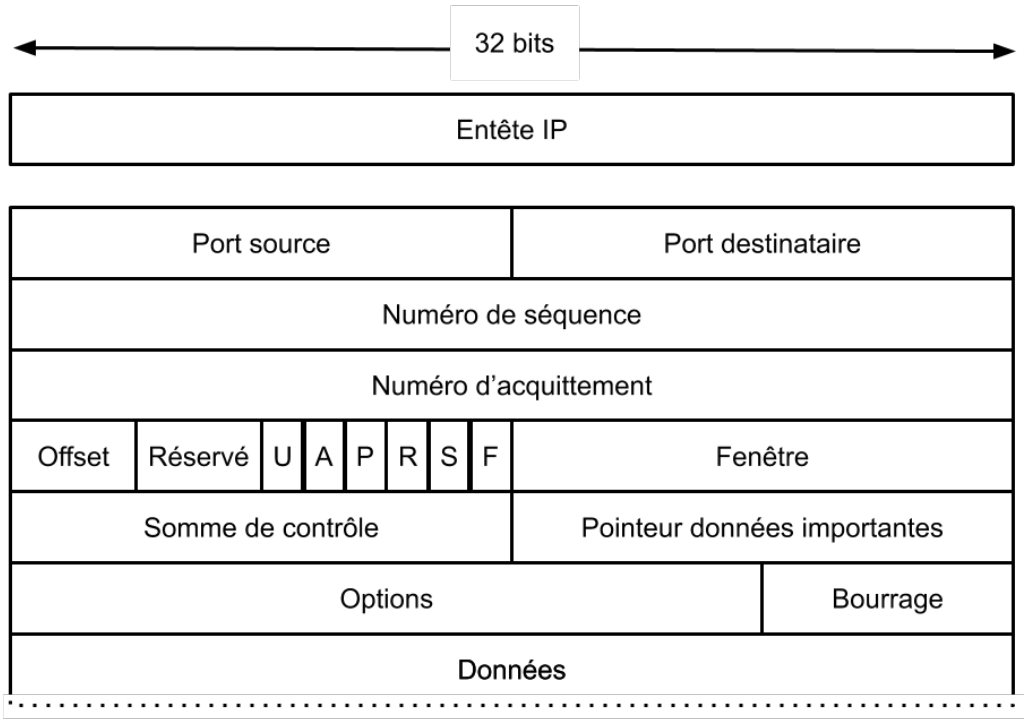


FIGURE 1.4 – Entete d'un paquet TCP. Un checksum de 16 bits permet la détection d'erreurs.

Le protocole que nous utilisons pour transmettre de l'information via la diode Elips-SD est donc le protocole UDP. Le protocole TCP ne peut absolument pas être utilisé pour fiabiliser une transmission transitant par une diode réseau, puisqu'il repose sur l'envoi par le récepteur de paquets à l'émetteur, ce que la diode interdit. D'autres raisons peuvent interdire l'emploi d'un tel protocole :

— Le canal de retour peut ne pas exister (c'est le cas dans les liaisons satellitaires unidirectionnelles [11]).
— Dans le cas d'une transmission multicast ou broadcast [10], le nombre de destinataires peut poser problème. Chaque récepteur subit des pertes différentes, et retransmettre séparément à chacun d'eux reviendrait à retransmettre toute l'information.
— Si le temps de latence de la transmission est important, il peut être contraignant d'utiliser le protocole TCP : il se peut en effet qu'on ait à redemander de nombreuses fois un paquet avant de le recevoir. Dans certains cas, la perte d'un paquet n'est pas nécessairement

problématique, mais dans le cas de données chiffrées, un seul paquet effacé conduit généralement à une grande quantité d'information non-exploitable. Dans le cadre de futures transmissions spatiales, le temps de latence sera très important, et il faudra sûrement avoir recours à des codes correcteurs pour réduire le nombre d'itérations nécessaires au protocole TCP : cette problématique est notamment étudiée dans [11].

Exemple 2 : distribution de données Dans le cas ou un émetteur unique souhaite distribuer de l'information à de nombreux destinataires, tout en ne sachant pas quand ceux-ci seront disponibles, il est possible d'adopter la solution du *broadcast disk*, ou *disque de diffusion à grande échelle* en français [5]. Le principe est de transmettre en boucle l'information afin que des récepteurs subissant beaucoup de pertes, ou ayant à se déconnecter, puissent reconstituer l'information. Ce type de solution a cependant un coût de réception important, correspondant à la quantité d'information reçue en double. Un récepteur qui n'aura pas reçu le dernier paquet émis devra attendre que l'émetteur ait émis tous les autres paquets avant de l'obtenir. Ce coût pour les récepteurs peut être réduit grâce à l'emploi d'un code correcteur. L'ajout de redondance permet alors au récepteur d'exploiter le contenu diffusé dès qu'il aura reçu assez d'information. Dans ce cas de figure, les codes à effacements sans rendement utilisés comme une "fontaine numérique", [12] que nous décrivons par la suite, peuvent s'avérer performants et adaptés.

Exemple 3 : stockage de données distribué Dans le cas du stockage distribué, les données peuvent être stockées à plusieurs endroits différents. Cette répartition a plusieurs objectifs :

- garantir que les données sont disponibles, même en cas de panne de certaines unités de stockage et minimiser la l'information totale nécessaire à leur émission ([62]).
- diminuer le temps nécessaire à la récupération des données, en demandant à plusieurs unités de stockage de les transmettre séparément.

La méthode la plus simple consiste à dupliquer les données sur plusieurs supports de données. Cependant, ceci nécessite une importante consommation mémoire au total, pour une qualité de service assez faible. Supposons par exemple qu'on dispose d'une base de donnée d'1 teraoctet, et de 10 sites de stockages. Une solution sans code correcteur consisterait à répliquer la base de donnée entière sur les 10 sites, ce qui représentera 10 teraoctets au total. La solution avec code à effacements consiste à appliquer un code correcteur sur cette base de donnée, puis à répartir l'information obtenue sur les 10 sites. La quantité d'information totale est alors nécessairement supérieure à 1 teraoctet, mais peut être inférieure à 10 teraoctets, tout en restant tolérante à la panne d'un ou plusieurs de ces sites (suivant la taille et la dimension du code utilisé).

1.2 Codes correcteurs

Nous savons grâce aux travaux de Claude Shannon parus en 1948 [53] qu'il est possible de transmettre de l'information sur un canal bruité via l'utilisation d'un codage adéquat. Dans un premier temps, nous entendrons par *erreurs* les altérations ou effacements subis par les messages transmis. Le code que nous présentons dans cette thèse est un code à *effacements*, par la suite nous nous concentrerons donc sur les *effacements* uniquement.

1.2.1 Définitions

Le code correcteur que nous présentons est un code à effacements linéaire par bloc non-MDS systématique. Dans un premier temps, il convient de rappeler certaines notions et définitions inhérentes à la théorie des codes correcteurs.

Definition 6. *On appelle corps fini (ou corps de Galois) un ensemble d'éléments pour lequel sont définies l'addition, la soustraction, la multiplication et la division. Pour tout entier premier p et entier m, il existe un unique corps fini à $q = p^m$ éléments, qu'on note $GF(q)$ ou $\mathbb{F}_q$.*

L'addition est notée $+$ et la multiplication $*$, et dans $GF(2) = \{0, 1\}$, on aura :

$$\begin{aligned} 0+0 &= 0 \\ 0+1 &= 1 \\ 1+0 &= 1 \\ 1+1 &= 0 \\ 0*0 &= 0 \\ 0*1 &= 0 \\ 1*0 &= 0 \\ 1*1 &= 1 \end{aligned}$$

Un corps fini possède un élément nul pour l'addition, et un élément nul pour la multiplication, respectivement 0 et 1. Ainsi,

$$\forall X \in GF(q), \begin{cases} X + 0 = X \\ X * 1 = X \end{cases}$$

L'addition et la multiplication sont des opérateurs commutatifs :

$$\forall X \in GF(q), \forall Y \in GF(q), \begin{cases} X + Y = Y + X \\ X * Y = Y * X \end{cases}$$

Ils sont aussi transitifs :

$$\forall X \in GF(q), \forall Y \in GF(q), \forall Z \in GF(q), \begin{cases} (X + Y) + Z = X + (Y + Z) \\ (X * Y) * Z = X * (Y * Z) \end{cases}$$

La multiplication est distributive sur l'addition :

$$\forall X \in GF(q), \forall Y \in GF(q), \forall Z \in GF(q) \quad \{ \; X * (Y + Z) = (X * Y) + (X * Z)$$

Un code correcteur $\mathcal{C}$ est un ensemble de mots de longueur n, chaque mot étant une suite de n symboles appartenant à un corps fini $GF(q)$ (qu'on peut aussi appeler *alphabet*). Ces mots sont obtenus grâce à une application ϕ, que l'on applique à un ensemble de mots sources plus courts à k symboles, $k \leq n$. Le code correcteur que nous présentons est un code linéaire systématique par bloc. Cette famille de codes correcteurs possède une structure d'espace vectoriel, ce qui en fait des codes pratiques à utiliser.

Definition 7. *On appelle code $\mathcal{C}$ de dimension k et de longueur n un sous-espace vectoriel de dimension k de l'espace vectoriel $\mathbb{F}_q^n$. On dit du code $\mathcal{C}$ qu'il est un $[n,k]-$code, et on appelle ses éléments les mots du code $\mathcal{C}$.*

On appelle mot de longueur n sur l'alphabet $\mathcal{A}$ une suite de n symboles appartenant à l'alphabet $\mathcal{A}$. L'architecture des ordinateurs reposant sur le bit, nous utiliserons la plupart du temps le corps $\mathcal{F}_2^m$. Dans le cas particulier de $\mathbb{F}_2^n$, l'addition et la soustraction reviennent à l'opération XOR bit à bit. On appelle symboles (ou lettres) les éléments du corps fini sur lequel travaille le code. Ces éléments forment les mots du code. L'ensemble des symboles peut être appelé un alphabet.

Definition 8. *On appelle rendement d'un $[n,k]-$code $\mathcal{C}$ de dimension k et de longueur n le rapport $R = k/n$.*

Plus ce ratio est proche de 0, plus le code possède de redondance, et inversement, plus il est proche de 1, moins il en possède.

Definition 9. *On appelle encodage une application ϕ de l'ensemble $GF(q^k)$ dans $GF(q^n)$, qui permet d'obtenir les mots de longueur n du code à partir des mots sources de longueur k.*

Les codes correcteurs se différencient suivant leur fonction d'encodage ϕ, ayant des complexités calculatoires différentes, et offrant au code une capacité de correction plus ou moins grande.

Definition 10. *On appelle matrice génératrice d'un code la matrice de l'application linéaire ϕ de l'encodage $\mathbb{F}_q^k$ dans $\mathbb{F}_q^n$ dont l'image est le code.*

Les lignes de la matrice génératrice $G \in \mathcal{M}(\mathbb{F}_q)_{k,n}$ forment une base de l'espace vectoriel du code. Soit X un élément de $\mathbb{F}_q^k$ et G la matrice génératrice d'un $[n,k]-$*code*. X est encodé en un mot de code $Y \in \mathbb{F}_q^n$ comme ceci : $Y = XG$. Lorsque le mot source se retrouve dans le mot encodé, on parle alors de code systématique. La matrice génératrice d'un code systématique contient alors la matrice identité. Si le code est systématique de dimension k et de longueur n, et que par conséquent sa matrice génératrice $G_{systmatique}$ contient la matrice identité Id_k, on peut alors noter $G_{systmatique} = [Id_k|C]$.

Definition 11. *Soit $\mathcal{C}$ un $[n,k]-$code. On dit que $H \in \mathcal{M}(\mathbb{F}_q)_{(n-k),n}$ est une matrice de parité de $\mathcal{C}$ si et seulement si*

$$\forall X \in \mathcal{C}, HX = 0 \text{ et } H \text{ est de rang plein.}$$

L'application correspondant à cette matrice a comme noyau les éléments du code. On peut obtenir la matrice de parité H d'un code systématique à partir de sa matrice génératrice G de la façon suivante :

$$H \quad = \quad [-H^T|Id_{n-k}].$$

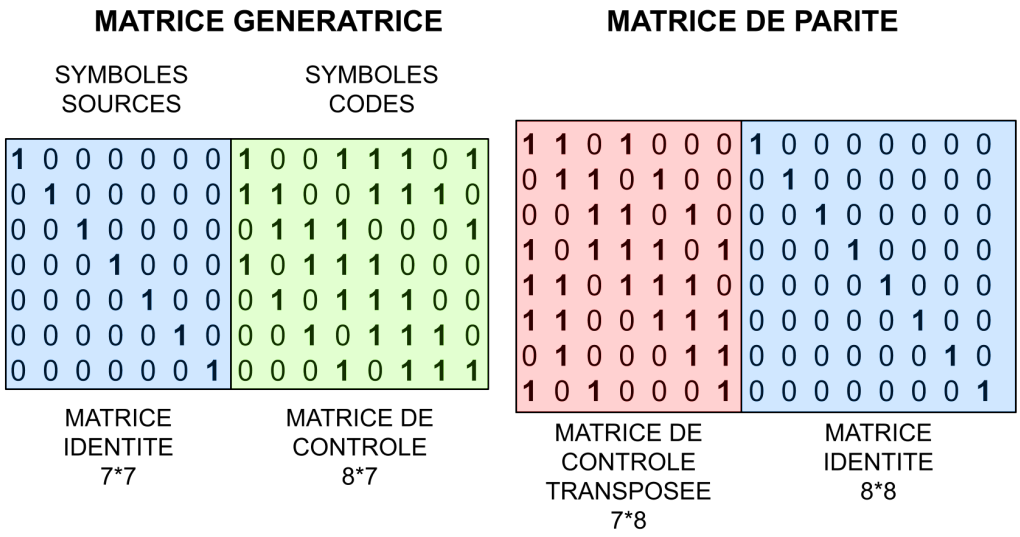


FIGURE 1.5 – La matrice de parité d'un code est obtenue à partir de sa matrice génératrice sous-forme systématique.

En effet, soit $Y \in \mathcal{C}$ un mot du code $\mathcal{C}$:

$$\begin{aligned} HY^T &= H(XG)^T \\ &= HG^T X^T \\ &= \{Id_k|C\}^T X^T \\ &= -C^T + C^T X^T = 0. \end{aligned}$$

Le décodage d'un code correcteur ayant pour but de retrouver le mot X le plus probablement envoyé (et donc le plus proche du mot reçu Y), il est nécessaire d'introduire une notion de distance entre mots. Cette distance, appelée distance de Hamming, découle du poids de Hamming. Quand l'alphabet est le corps $\mathbb{F}_2$, le poids de Hamming d'un mot est alors le nombre de 1 dans ce mot (le nombre de bits égaux à 1 de ce mot).

Definition 12. *Le poids de Hamming d'un mot X est le nombre de symboles de X différents du symbole nul. La distance de Hamming entre deux mots X et Y est le nombre de positions auxquelles les symboles de ces mots diffèrent. Ainsi, la distance entre deux mots X et Y est égale au poids de Hamming de $X - Y$.*

La notion de distance de Hamming nous amène à nous intéresser à la distance séparant les mots d'un code, et plus particulièrement à la plus petite distance possible entre deux mots du code. On appelle cette distance la distance minimale du code, souvent notée d_min. Dans le cas des codes linéaires, cette distance est aussi le minimum des poids des mots non-nuls du code.

Un $[n, k] - code$ de distance minimale d sera noté $[n, k, d] - code$. Les capacités de correction d'un code linéaire dépendent fortement de sa distance minimale. Celle-ci nous donne une borne sur le nombre de correction garanties par le code. Un code correcteur de distance minimale d_{min} permet de corriger un mot si $d_{min} \geq 2t + e + 1$, avec t le nombre d'erreurs et e le nombre d'effacements subis par ce mot. Cette distance minimale possède une borne, appelée borne de Singleton [55], qui dépend de la longueur et de la dimension du code : $d_{min} \leq n - k + 1$. Les

codes dont la distance minimale atteint cette borne, appelés des codes MDS, pour Maximum Distance Separable, possèdent une capacité de correction optimale.

Definition 13. *Un $[n, k, d]-$ code est dit MDS (pour Maximum Distance Separable) si et seulement si*

$$n - k = d - 1$$

On dit que ce code atteint la borne de Singleton.

Exemple : code à répétitions Pour récapituler les définitions énoncées précédemment, appliquons les à un code correcteur simple : le code à répétition. Supposons qu'on souhaite transmettre des mots (ou messages) de longueur $k = 1$ seul symbole, avec des symboles appartenant à $\mathbb{F}_2^3$, en envoyant chaque symbole 2 fois. Les mots transmis sont alors de longueur $n = 2k = 2$, et le code est alors un $[2, 1]-$*code* linéaire, de dimension $k = 1$ et de longueur $n = 2$. L'espace vectoriel contenant tous les mots de départ est $\mathbb{F}_2^3$, et l'espace vectoriel contenant tous les mots du code est $\mathbb{F}_2^6$:

$\mathbb{F}_2^3$	$\longrightarrow$	$\mathbb{F}_2^6$
000	$\longrightarrow$	000000
001	$\longrightarrow$	001001
010	$\longrightarrow$	010010
011	$\longrightarrow$	011011
100	$\longrightarrow$	100100
101	$\longrightarrow$	101101
110	$\longrightarrow$	110110
111	$\longrightarrow$	111111

$\mathbb{F}_2^3 \longrightarrow \mathbb{F}_2^6$
$a \longrightarrow aa$
$b \longrightarrow bb$
$c \longrightarrow cc$
$d \longrightarrow dd$
$e \longrightarrow ee$
$f \longrightarrow ff$
$g \longrightarrow gg$
$h \longrightarrow hh$

Le ratio R de notre code est de $R = k/n = 1/2$, ce qui signifie que nous avons recours à une redondance de 1 sur 2. Les distances entre tous les mots du code sont les suivantes :

	000 000	001 001	010 010	011 011	100 100	101 101	110 110	111 111
000 000	0	2	2	2	2	2	2	2
001 001	2	0	2	2	2	2	2	2
010 010	2	2	0	2	2	2	2	2
011 011	2	2	2	0	2	2	2	2
100 100	2	2	2	2	0	2	2	2
101 101	2	2	2	2	2	0	2	2
110 110	2	2	2	2	2	2	0	2
111 111	2	2	2	2	2	2	2	0

TABLE 1.1 – Distance de Hamming entre deux mots du code à répétition

La distance minimale d_{min} de ce code est 2, car pour toute paire (X, Y) de deux mots différents du code, le poids de Hamming de $X - Y$ est de 2. En effet, comme on peut le voir

dans la table [1.1] les mots **011 011** et **101 101** ont deux symboles qui diffèrent, leur distance de Hamming est donc de 2. Elle correspond aussi au poids le plus faible parmi les mots non-nuls du code, or tous les mots de notre code ont un poids de 2, à l'exception de **000 000**. Ce code est donc un $[2, 1, 2] - code$. Il n'atteint pas la borne de Singleton et est donc non-MDS, car $n-k \neq d-1$. Ce code à répétition saura décoder convenablement un mot X transmis et devenu le mot Y, si X a subi t erreurs et e effacements, avec $2 \geq 2t + e + 1$. Il saura donc décoder un mot ayant subi un unique effacement, mais ne saura pas décoder un mot ayant subi une erreur, ou plus de un effacement.

La matrice génératrice G de ce code est : $\binom{1}{1}$, et sa matrice de parité $\mathcal{M}$ est (1). Le code à répétition est un code systématique, car chaque mot source se retrouve dans le mot code lui correspondant.

1.2.2 Codes à effacements

Cette thèse présente un code correcteur pour le canal à effacements tel que présenté dans la figure [1.2]. Dans un premier temps nous présentons le principe général sur lequel reposent l'utilisation et le fonctionnement de ces codes. Prenons l'exemple suivant : nous souhaitons transmettre k symboles sur un canal à effacements dont la probabilité d'effacements vaut p.

Definition 14. *On appelle probabilité d'effacements p la probabilité qu'un symbole transmis sur un canal soit effacé.*

Afin de fiabiliser convenablement cette transmission, nous devons choisir correcteur dont le rendement R est inférieur à la capacité $1 - p$ du canal.

Tout d'abord, nous allons encoder les k symboles du message sources en $n \geq k$ symboles codes. Ces n symboles peuvent alors être transmis. $p * n$ symboles vont en moyenne être effacés, et le récepteur disposera alors de $(1 - p) * n$ paquets.

Si le récepteur a à sa disposition k paquets ou plus, il a une chance de décoder les symboles effacés et de reconstruire le message original. A contratio, le décodage échoue automatiquement si le récepteur dispose de moins de k paquets.

1.2.3 Codage par paquets

Les codes à effacements travaillent généralement sur les paquets fournis d'un protocole de transmission. Les codes corrigeant des altérations de données s'exécutent sur des petites quantités d'information (quelques milliers de bits), les codes à effacements travaillent sur des mots pouvant atteindre plusieurs mégaoctets. En général, on les utilise juste en dessous de la couche application, là où les unités de données transmises sont grandes (paquets IP). On les appelle donc souvent codes correcteurs de niveau applicatif (ou codes AL-FEC) pour "Application-Level Forward Error Correction". Ils protéger des ensembles de données divisées en paquets. Le code correcteur et la manière dont le code organise et encode les données est appelé un code AL-FEC.

Nous avons vu précédemment que ces paquets ont des tailles qui peuvent atteindre plusieurs milliers d'octets, et que rien ne garantit qu'ils aient tous la même taille. Dans le cas de paquets ayant différentes tailles, on pourra se rapporter au cas des paquets de taille constante en utilisant du padding (remplissage des paquets avec des 0 pour leur faire atteindre la taille désirée). Néanmoins, cette méthode impose d'avoir recours à de l'information inutile, ce que

nous chercherons à éviter : idéalement, on préférera travailler directement sur des paquets ayant la taille adéquat. A partir de maintenant nous considérerons que les paquets d'un même bloc ont tous la même taille.

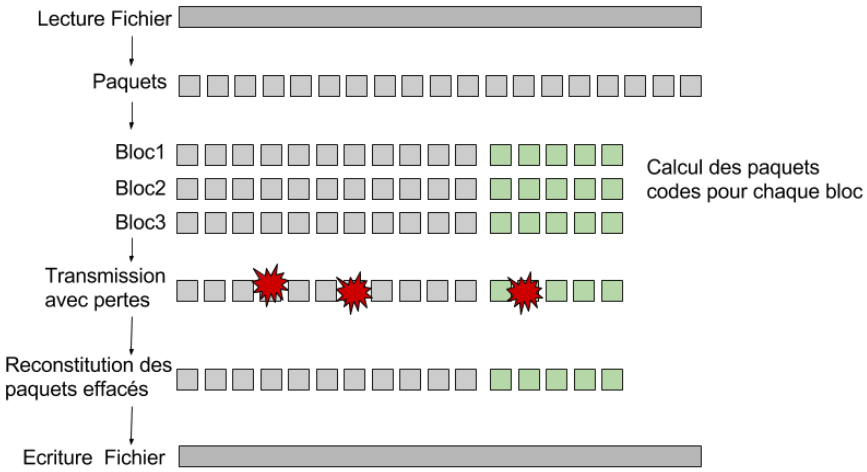


FIGURE 1.6 – Fonctionnement d'un code par bloc. Les paquets sont réunis en blocs qui sont encodés indépendamment les uns des autres.

1.2.4 Équivalence symboles-paquets

Les codes de niveau applicatif protégeant ce type de flux doivent permettre au récepteur de récupérer les effacements de paquets. Nous avons vu que les codes à effacements permettent justement de récupérer la valeur de symboles effacés. Il convient donc de trouver une manière de faire correspondre paquets et symboles. Plusieurs propositions peuvent être envisagées :

— Chaque paquet est considéré comme un symbole unique (c'est cette méthode que nous utilisons dans notre code correcteur). Avec certains codes, utiliser des symboles de grande taille peut s'avérer coûteux en calculs, les calculs à effectuer devenant de plus en plus compliqués à mesure que les symboles deviennent grands.

— On peut utiliser un tableau pour encoder les paquets lignes par lignes : chaque paquets est rangé dans une colonne, et on encode le tableau ligne par ligne, en s'assurant que chaque paquet est convenablement positionné. Cela revient alors à utiliser en parallèle plusieurs instances du code correcteur.

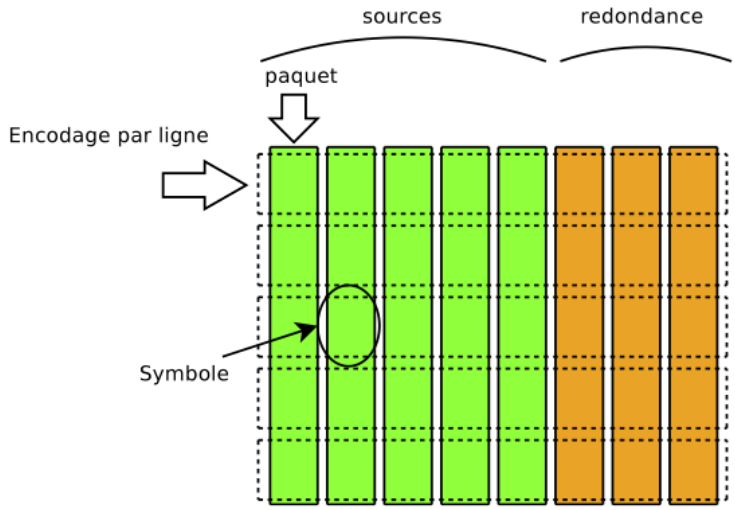


FIGURE 1.7 – Exemple de l'utilisation d'un tableau pour l'encodage par paquets

.

On peut voir les codes reposant sur le corps fini $\mathcal{F}_2^k$ (comme les codes LDPC binaires) de différentes façons. Les opérations sur de tels corps étant des XOR, on peut considérer soit qu'il s'agit d'un unique code sur $\mathcal{F}_2^k$ (première solution), soit qu'il s'agit de k instances d'un code, travaillant sur $\mathcal{F}_2$, exécutées en parallèle (deuxième solution).

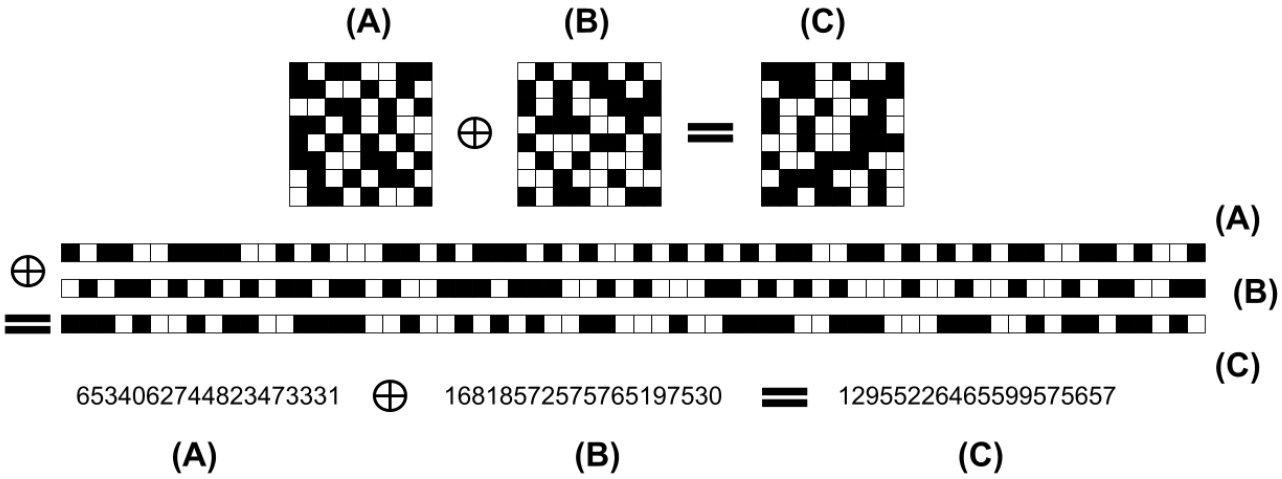


FIGURE 1.8 – Exemple de l'opérateur XOR bit-à-bit sur le corps $\mathcal{F}_2^{64}$: $C = A \oplus B$ représenté de différentes manières. Le i-ème bit de C vaut 1 si et seulement si les i-èmes bits de A et B diffèrent.

.

En utilisant un code avec une taille de symbole et une longueur fixées, on utilisera la deuxième solution. Pour un Reed-Solomon travaillant sur $GF(256)$, on pourra encoder des paquets par bloc de 255. Pour cela on peut placer les paquets dans un tableau dont ayant des colonnes de largeur 8 bits, puis on encoder ligne par ligne avec un code Reed-Solomon.

1.3 Evaluation des codes correcteurs

Pour évaluer un code correcteur, il convient de décrire les métriques sur lesquelles reposent ces évaluations, ainsi que les méthodes à utiliser. Un grand nombre de travaux concernant les codes à effacements ont vu le jour, présentant de nouveaux codes et de nouveaux algorithmes de décodage. Cependant il est en général difficile de comparer deux solutions. En effet, les résultats présentés ne reposent pas toujours sur le même type de simulation, voire n'utilisent pas les mêmes métriques, ce qui peut rendre compliquées leurs comparaisons.

On peut séparer ces méthodes dans deux grands ensembles : les méthodes basées sur de nombreuses simulations, et les méthodes analytiques. Les méthodes analytiques nécessitent de trouver un bon modèle pour le système, puis de résoudre le problème posé. Les méthodes basées sur des simulations se bases sur un ensemble de résultats expérimentaux, qui vont nous permettre de se figurer le comportement de notre système.

Nous allons donc présenter les différentes types d'échec du décodage des codes à effacements, puis les métriques qui leurs sont associées et les techniques permettant de les obtenir.

1.3.1 Probabilité d'erreur de décodage

Sur le canal à effacements, on s'intéressera souvent à la probabilité qu'au moins un symbole effacé n'ait pas pu être décodé. Ce sera notamment le cas lors de la transmission de données cryptées, qui sont inutilisables lorsqu'on ne dispose pas de toute l'information. A contrario, lorsqu'il est possible d'utiliser une partie de l'information source, on pourra considérer la probabilité d'effacement symbole par symbole après le décodage : certains cas d'utilisation ne nous imposent pas de disposer de toute l'information, mais seulement d'une partie (par exemple pour la transmission de vidéo, on peut tolérer que quelques images ne parviennent pas au récepteur).

On exprime souvent ces probabilités en fonction de la probabilité d'effacements du canal, en fixant le rendement du code. Cependant, on peut aussi les comparer au taux d'effacements du canal : ce taux correspond au nombre exact de paquets effacés lors d'une transmission, alors que la probabilité d'effacements correspond à une probabilité que chaque symbole soit effacés, indépendemment des autres.

On utilise ces métriques pour évaluer le système, dans le cas où la fiabilité du système est primordiale et que la probabilité d'effacements sur le canal. Nous pouvons distinguer deux principaux types de probabilités d'erreur.

Probabilité d'erreur bloc

La probabilité que le décodage échoue (qu'il reste au moins un symbole non-décodé) est appelée probabilité d'erreur "bloc". Elle correspond à la probabilité que le décodage n'ait pas terminé étant donné les symboles reçus. On s'intéressera à cette probabilité quand on transmettra des données cryptées (ce qui est souvent le cas lors de l'emploi de l'Elips-SD), car si des symboles n'ont pas pu être décodés, l'ensemble du fichier est inutilisable.

Probabilité d'erreur symbole

Comme dit précédemment, un échec du décodage n'interdit pas nécessairement l'utilisation de l'information décodée ou reçue. Dans le cas d'un code systématique par exemple, on peut avoir reçu une partie des symboles sources. De plus certains décodeurs, tels que les décodeurs

itératifs (dont le code que nous présentons fait partie), permettent un décodage des symboles progressif, à mesure que de nouveaux symboles sont reçus. Même si le décodage aboutit à un échec et qu'il reste des symboles non-décodés, on pourra exploiter une partie des symboles reçus ou décodés.

1.3.2 Métriques associées à l'efficacité d'un code correcteur

Capacité de correction

En théorie, on cherche à utiliser un code le plus proche possible de la capacité du canal de transmission. On dit d'un code atteignant la capacité du canal que c'est un code parfait. Sur le canal à effacements, un code parfait est un $[n, k] - code$ qui de corrige à coup-sûr $n - k$ effacements (on dit alors qu'il atteint la borne de Singleton). Une première manière d'évaluer un code (afin de le comparer à d'autres codes) est de calculer la distance qui le sépare d'un code parfait. Cette grandeur, la distance à la capacité, notée Δ, est la différence entre la capacité du code, C_{code}, et la capacité du canal, C_{canal}.

Sur binaire à effacements, la capacité du canal est égale à $C_{canal} = 1-p$, où p est la probabilité d'effacement sur le canal. La capacité du code C_{code}, égale à $1-P^*$ est la probabilité d'effacement maximum que le code va pouvoir corriger. On en déduit donc la distance à la capacité :

Definition 15. *Soit un canal à effacements de capacité $C_{canal} = 1 - p$, où p est la probabilité d'effacement sur le canal, et un code de capacité $1 - P^*$. On appelle distance à la capacité la valeur*

$$\Delta = C_{code} - C_{canal} = (1 - P^*) - (1 - p) = p - P^*$$

Plus cette valeur tend vers 0, plus le code s'approche d'un code parfait.

Inefficacité d'un code

Jusqu'à présent nous avons présenté les méthodes reposant sur la comparaison entre la probabilité d'erreur et la quantité d'information effacée (ou reçue). Pour cela nous présentons le *ratio d'inefficacité de décodage*, défini comme le rapport entre la le nombre moyen de symboles nécessaire au décodage et le nombre de symboles sources initialement dans notre bloc à encoder (la dimension du code, k).

Definition 16. *Soit un code à effacements de dimension k nécessitant $Nb_{symboles}$ pour que le décodage réussisse. On appelle inefficacité du code la valeur*

$$Ineff = \frac{Nb_{symboles}}{k}$$

On cherchera un code ayant le ratio d'efficacité le plus bas possible, un code parfait ayant un ratio d'inefficacité de 1. A partir de ratio on pourra calculer le surcout de la transmission (*overhead* en anglais), ϵ : il correspond à la distance entre le ratio d'inefficacité du code et celui d'un code parfait. Le ratio d'inefficacité d'un code parfait étant égal à 1, le surcout s'exprime alors :

Definition 17. *Soit un code à effacements ayant un ratio d'inefficacité $Ineff$. On appelle surcout du code la la différence entre le ratio d'inefficacité du code et le ratio d'inefficacité d'un code parfait (valant 1), qui vaut :*

$$\epsilon = Ineff - 1.$$

Le décodage aboutira à un succès à partir de $k * (1 + \epsilon)$ symboles reçus en moyenne. On pourra s'intéresser au nombre de symboles supplémentaires nécessaires au décodage, $(k * \epsilon)$: plus cette valeur est proche de 0 plus le code se comporte comme un code parfait.

1.3.3 Terminologie associée aux probabilités d'erreur

La probabilité d'erreur de décodage d'un code parfait dépend uniquement de la dimension du code (k) et du nombre de symboles reçus. La zone de succès et la zone d'échec du décodage sont donc facilement distinguables : quand on a reçu assez de paquets $(> k)$, le décodage aboutit à un succès, sinon à un échec. Dans le cas général (lorsqu'on utilise un code non-MDS), on ne dispose pas d'une telle frontière, et on peut découper en trois zones les courbes représentant la probabilité d'erreur de décodage en fonction de la probabilité d'effacement du canal :

— si la probabilité d'effacement du canal est supérieure à la capacité du code. Même pour un code MDS il est impossible de les décoder.

— Lorsque le nombre d'effacements est assez faible, la probabilité d'échec du décodage est alors elle aussi très faible : on parle alors de zone "plancher" ("error floor" en anglais). Les meilleurs codes correcteurs ont une grande zone plancher (les codes MDS voyant leur zone plancher s'étendre de 0 effacements à $n - k$ effacements).

— La zone intermédiaire, se situant entre ces deux zones, qu'on appelle la zone de "waterfall". Ici, la probabilité d'échec du décodage évolue rapidement.

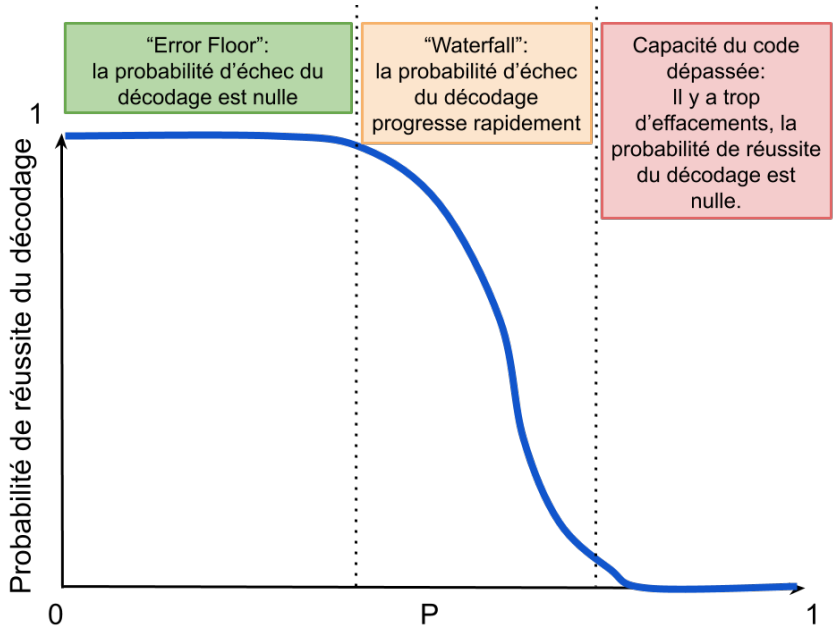


FIGURE 1.9 – Courbe représentant la probabilité d'erreur de décodage d'un code non-MDS en fonction de la probabilité d'effacements

.

On notera que ces courbes sont souvent appelées "courbes en S" ou encore "courbes sigmoïdes". On retrouve très souvent ce genre de courbes dans les problèmes de propagation de

l'information, ou d'un virus au sein d'une population par exemple.

1.3.4 Complexité algorithmique

Nous avons présenté précédemment le principe général d'un code correcteur, reposant sur l'utilisation de fonction mathématiques, l'*encodage* et le *décodage*. Ces fonctions mathématiques doivent être calculées par un ordinateur, et sont plus ou moins "compliquées" à calculer, suivant le code utilisé. Afin de comparer la simplicité ou la difficulté à calculer ces fonctions, il convient d'introduire la notion de *complexité algorithmique*. La complexité d'un algorithme va déterminer son utilisation concrète. Si un code correcteur a un débit trop faible (du à une complexité trop importante), il ralentira l'ensemble du système, en devenant un goulot d'étranglement (*bottleneck* en anglais). Ce sera d'autant plus vrai que le matériel utilisé disposera de peu de puissance de calculs ou de mémoire.

Les codes correcteurs peuvent aussi bien être utilisés au niveau matériel qu'au niveau logiciel. Le code correcteur que nous présentons faisant partie des codes à effacements de niveau applicatif (décrits précédemment comme AL-FEC), nous nous concentrerons sur les points critiques de la complexité pour son implémentation.

Complexité théorique

Pour estimer le coût d'un algorithme, on peut étudier sa complexité, qui représente le nombre d'opérations effectuées en fonction de la taille des données en entrée. Pour cela, on décompose l'algorithme en opérations de bases, comme les multiplications ou les additions, et on va chercher à savoir combien de ces opérations seront exécutées en moyenne. La fonction f, représentant le nombre de calculs nécessaires au déroulement d'un algorithme est un grand O de g si :

$$\lim_{n \longrightarrow \infty} sup \left| \frac{f(n)}{g(n)} \right| < \infty$$

On dit alors que f est dominée par g. On compare alors cette fonction de coût d'un algorithme avec des fonctions usuelles lorsque la taille des entrées n progresse. Ainsi on dira d'un algorithme qu'il a un coût constant si $f(n) \in O(1)$ (accéder à l'indice d'un tableau), logarithmique si $f(n) \in \log(n)$ (recherche dans une liste triée) linéaire si $f(n) \in O(n)$ (recherche du maximum d'un tableau), quadratique (tri d'une liste dans le pire des cas) si $f(n) \in O(n^2)$, ou encore exponentiel si $f(n) \in O(a^n), a > 1$ (problème du voyageur de commerce).

La complexité d'un algorithme nous indique comment va évoluer le temps de calcul de l'algorithme en fonction de la taille de l'entrée, sans tenir compte du coût des opérations de base. Elle représente donc une évaluation du coût de l'algorithme, mais sera souvent insuffisante, et il sera intéressant de réaliser une étude basée sur des simulations afin de réellement comparer les temps de calcul de deux algorithmes.

Vitesse et débit

Une autre manière, plus pratique, d'évaluer le temps nécessaire au déroulement d'un algorithme, est d'effectuer de nombreuses simulations sur plusieurs jeux de données. Dans le cas d'un code correcteur, on cherchera donc à évaluer la progression du temps d'encodage et de

décodage, en fonction de la taille des blocs, de la taille des paquets ou encore du nombre d'effacements, afin d'obtenir le débit, moyen, minimum ou maximum du code. Une fois ce débit obtenu, on saura si le code est utilisable en pratique sur notre canal de transmission.

Lorsqu'on souhaite comparer deux algorithmes (en ayant fixé les mêmes paramètres, comme la longueur et dimension du code, taille des symboles, et les paramètres du canal), on doit s'assurer d'effectuer les simulations sur du matériel identique.

Ces métriques ont l'avantage d'être simples à interpréter. De plus, elles permettent de se figurer en pratique la complexité d'un algorithme. Elles nécessitent néanmoins de disposer de l'implémentation de ces algorithmes, alors que leur analyse théorique nécessite uniquement une analyse de l'algorithme (en étudiant son pseudo-code par exemple).

Consommation mémoire

En plus d'étudier la complexité d'un algorithme en terme de temps, il peut être intéressant d'étudier la quantité mémoire qu'il nécessite. Cette complexité représente la quantité d'information maximale nécessaire au déroulement de l'algorithme. Même si les machines actuelles disposent de beaucoup de mémoire, et que celle-ci augmente rapidement, les codes AL-FEC travaillent sur des quantités de données potentiellement être très importantes (pouvant atteindre plusieurs Gigaoctets). De plus, sur du matériel embarqué (tels que les terminaux mobiles ou l'internet des objets), la mémoire disponible peut parfois s'avérer faible.

1.3.5 Méthodes d'évaluation des performances d'un code

Comme on l'a vu précédemment, on peut étudier les métriques de performances d'un code, ou d'un algorithme en général de deux façons : de manière analytique, ou bien à partir de simulations.

Méthodes analytiques

Ces méthodes nous imposent de modéliser l'algorithme à étudier. Dans le cas d'un code correcteur, nous devrons modéliser son encodage et son décodage. De plus ces méthodes dépendent notamment du type de code étudié. Pour certains codes (par exemple les codes MDS), des arguments théoriques d'algèbre linéaire sont suffisent souvent à déterminer le comportement d'un code.

Dans le cas des codes LDPC, des outils statistiques (reposant sur des simulations) sont souvent nécessaires. Ainsi, ces méthodes sont souvent insuffisantes pour étudier les performances d'un code en condition réelles d'utilisation.

Modèle de canal et ordonnancement des symboles

On a vu précédemment que l'ordre d'émission et la position des symboles effacés dans le mot transmis a un impact sur l'issue du décodage. Dans le cas des codes non-MDS en particulier, tous les symboles ne sont pas nécessairement équivalents : pour un nombre d'effacements fixés, deux répartitions d'effacements différentes pourront mener à des décodages différents, réussite ou échec. Cette répartition des paquets effacés va notamment dépendre de deux éléments : l'ordonnancement des symboles lors de la transmission, et le modèle de perte du canal. Nous présentons ici deux types de modèles de pertes sur le canal :

— les modèles de pertes indépendantes : la probabilité qu'un symbole soit effacés ne dépend pas des autres symboles. On nomme souvent ces modèles des modèles sans-mémoire.
— Les modèles dans lesquels les pertes peuvent être dépendantes les unes des autres, qu'on appelle les modèles Markoviens. Pour ces modèles, la probabilité d'effacements du canal peut évoluer avec le temps. Un modèle de ce type relativement simple est tout simplement un modèle à deux états : "mauvais" et "bons", auxquels on associe des probabilités de transition. Ce type de modèles est particulièrement utile pour modéliser les pertes *impulsives* (*impulsive losses* en anglais, ou encore pertes en *rafale*). Il s'avère que le code correcteur que nous présentons est particulièrement adapté aux pertes impulsives, et que le canal que nous avons à fiabiliser (passant par une diode réseau) subit en général ce type de pertes.

FIGURE 1.10 – Exemple de perte impulsive de 8 paquets consécutifs. Ce type de pertes est souvent inhérent au canal de transmission. Certains codes sont plus ou moins adaptés à corriger des pertes impulsives.
.

L'émetteur peut choisir entre trois modes de transmission :
— Le mode le plus simple est la transmission dans l'ordre séquentiel : on transmet les symboles dans l'ordre dans lequel ils sont entrés dans l'encodeur. Dans le cas d'un code systématique, on transmettra généralement les codes sources avant les symboles de redondance. Le code que nous présentons est un code systématique, et nous transmettons les symboles de manière séquentielle (nous transmettons les symboles sources avant les symboles de parité).

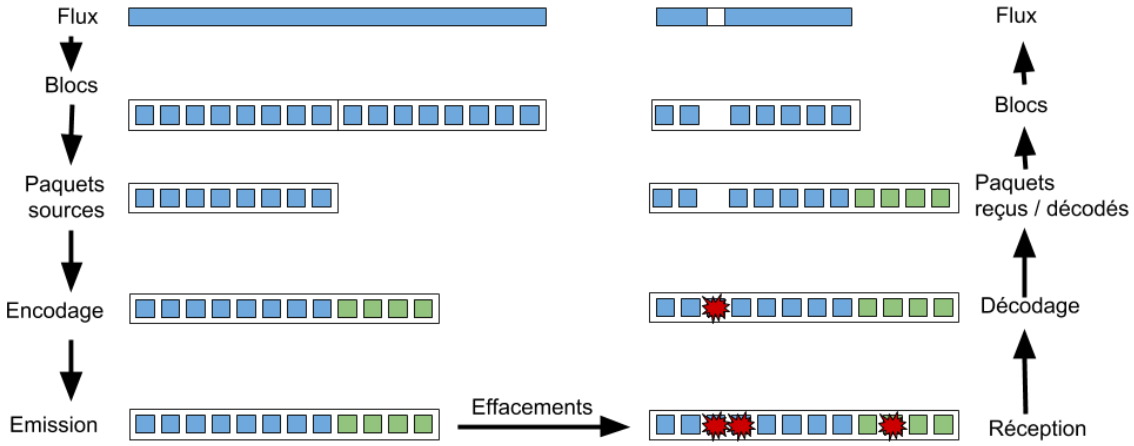


FIGURE 1.11 – Code par bloc systématique et transmission séquentielle. L'information est réunie en paquets, répartis en blocs de tailles constantes. Chaque bloc est ensuite encodé séparément pour produire des paquets codes.

— On peut transmettre les symboles dans un ordre aléatoire. Ainsi, le modèle de perte du canal peut alors être négligé, puisque les pertes peuvent alors être vues comme des pertes indépendantes. On utilise souvent ce type de transmission lorsque les pertes se font de manière impulsives lorsque le code utilisé y est sensible. Cette technique nécessite cependant de conserver les paquets à transmettre un certain temps, ce qui peut s'avérer problématique si on dispose de peu de mémoire, ou qu'on souhaite éviter de conserver de l'information sur un support trop longtemps.
— Enfin, on peut permuter les symboles de façon déterministe. On appelle cette opération un entrelacement. On peut par exemple décider d'envoyer tous les paquets dont l'indice est congru à 0 modulo 5, puis à 1, puis 2, ... Dans ce cas, une perte impulsive de 5 symboles consécutifs se retrouvera dispersée, et potentiellement décodable [44].

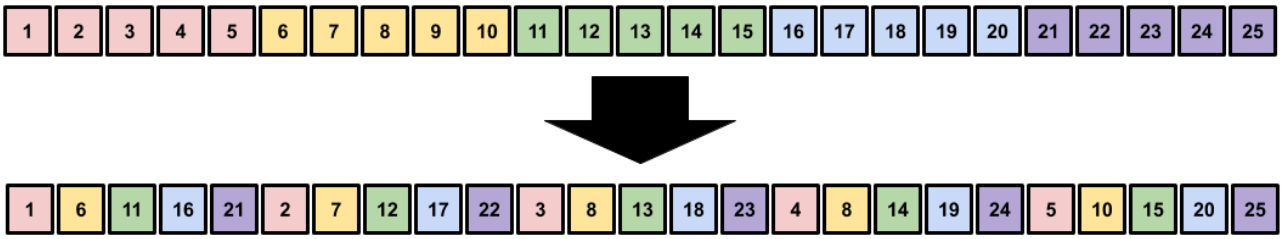


FIGURE 1.12 – Fonctionnement d'un entrelacement : on remarque que deux paquets initialement consécutifs sont maintenant distants de 5 paquets.

Simulations de type Monte-Carlo

Pour évaluer les performances d'un système dans des conditions particulières, on peut avoir recours à des techniques basées sur des simulations. En particulier la dimension k, la longueur n du code et la taille des symboles doivent être fixées. On cherchera à effectuer un grand nombre de simulations afin d'estimer le plus précisément possible le comportement du code étudié en moyenne.

Pour ce faire, on aura notamment recours à la méthode de Monte Carlo. Il s'agit d'une technique permettant de calculer une grandeur en fonction de processus aléatoires. Ces méthodes, introduites en 1949 Nicholas Metropolis and S. Ulam dans [38] reposent sur l'exécution d'un grand nombre d'expériences et nécessitent de nombreux calculs. Elles reposent sur l'idée selon laquelle un grand nombre d'expériences permet de se figurer le comportement moyen d'un algorithme en moyenne.

Afin d'estimer g_{reelle} la valeur moyenne d'une fonction g de variable aléatoire X, on réalise N simulations pour N points tirés au hasard parmi les valeurs que peut prendre X. On utilise souvent la loi des grands nombres. Soit $g_{empirique}(N)$ la moyenne empirique obtenue à partir de N tirages aléatoires indépendants, elle s'écrit :

$$g_{empirique}(N) = \sum_{i=0}^{i=N} \frac{g(X_i)}{N}$$

La loi des grands nombres nous dit que :

$$\forall \epsilon > 0, \lim_{N \longrightarrow \infty} P(|g_{empirique}(N) - g_{reelle}| \geq \epsilon) = 0$$

Ainsi, en augmentant le nombre de mesures de la fonction g on diminue l'écart entre la moyenne empirique $g_{empirique}$ et la valeur moyenne, g_{reelle}.

La méthode de Monte Carlo nous permet donc d'évaluer les performances d'un code. Supposons que l'on dispose d'un code correcteur, et qu'on souhaite connaître la probabilité d'échec du décodage pour un nombre d'effacements fixé n_e. On va réaliser N expériences, pour lesquels on va essayer de décoder n_e effacements. En fonction du nombre A de réussites, et du nombre $B = N - A$ d'échecs, on pourra déterminer la moyenne $g_{empirique} = A/N$, qui représentera alors la probabilité de réussite du décodage. Plus on aura réalisé d'expériences, plus cette probabilité s'approchera de la probabilité réelle, g_{reelle}. Afin d'obtenir une bonne précision, il nous faudra cependant réaliser un grand nombre de simulations, ce qui pourra engendrer un cout important.

Exemple de méthode pratique de simulation de Monte Carlo

Nous présentons ici un exemple de simulation de type Monte Carlo, permettant d'étudier probabilité d'erreur bloc d'un $[n, k]$ code de longueur k et de dimension n sur un canal à effacements de paramètre p avec une précision $\epsilon \leq 10^{-2}$. Tout d'abord, calculons le nombre d'expériences nécessaires $N_{experiences}$ afin de s'assurer de la précision de nos tests :

$$\epsilon \leq 10^{-2} \longrightarrow N_{experiences} > 10^{-(-2)+2} = 10^4.$$

Nous aurons donc à effectuer plus de $N_{experiences} = 10^4$ pour obtenir une précision de $\epsilon \leq 10^{-2}$. Nous décidons de choisir comme modèle de pertes le modèle *sans mémoire*, c'est-à-dire que chaque symbole transmis a une probabilité p de se voir effacé, et celle-ci n'a aucun impact sur la probabilité qu'un autre symbole soit effacé.

Les 10^4 expériences que nous allons réaliser consisteront en cette suite d'étapes :

- à partir du mot X de longueur k, l'émetteur calcule le mot $Y = \{Y_0, Y_1, Y_2, ..., Y_{n-1}\}$ de longueur n.
- pour chaque symbole $Y_i, 0 \leq i < n$, l'émetteur tire au hasard une variable aléatoire MC_i, valant 1 avec une probabilité $1 - p$ et 0 avec une probabilité p.
- chaque symbole Y_i est ensuite "transmis" si $MC_i = 1$, ou "non-transmis" (considéré comme effacé par le récepteur) si $MC_i = 0$. Nous mettons des guillemets autour de *transmis* car il n'est pas nécessaire de transmettre les symboles, puisque nous simulons les effacements, et donc le canal de transmission.
- enfin, le récepteur essaie de reconstruire le mot X en appliquant l'algorithme de décodage. Si celui-ci aboutit sur un succès, le récepteur incrémente un compteur N_{succes}, initialisée à 0 au départ.

Finalement, le récepteur n'aura plus qu'à calculer $g_{empirique}(10^4) = N_{succes}/10^4$, qui sera alors très proche de la valeur g_{reelle} représentant la probabilité que le décodage aboutisse à un succès. Précisons qu'en pratique, il n'est pas nécessaire pour l'émetteur de calculer le mot Y (il n'enverra que des symboles ayant pour valeur 0, voir uniquement leurs indices), et que dans la plupart des cas, le récepteur n'a pas à décoder réellement le mot X. En effet, il s'agit ici d'une simulation, et nous n'avons pas réellement besoin de *décoder* le mot Y : il est souvent possible de savoir si le décodage aboutit sur un échec ou sur une réussite, sans avoir à effectuer tous les calculs (c'est le cas pour le code correcteur que nous présentons).

Conclusion

Nous avons présenté des méthodes et des métriques permettant d'évaluer et de comparer des codes à effacements. Ces métriques dépendent fortement du cas d'utilisation et du matériel employé.

Les outils présentés dans cette partie doivent nous permettre de répondre à cette question : quel est le meilleur code entre deux codes A et B ? Le choix est simple lorsque nous ne nous intéressons qu'à une seule caractéristique (la consommation mémoire par exemple). Cependant, il nous faut souvent tenir compte de plusieurs critères, parfois interdépendant interdépendants, ce qui rend ce choix difficile. Si on souhaite la meilleure capacité de correction, on choisira d'utiliser les codes MDS, car ils sont des codes parfaits, et que tout code non-MDS aura une capacité de correction inférieure, pour une dimension et une longueur de code fixées. A contrario, lorsqu'on tient compte de la complexité de l'encodage ou du décodage, ces codes deviennent difficilement utilisables pour de grandes dimensions, notamment sur du matériel disposant d'une faible puissance de calcul. Un compromis doit alors être trouvé entre la capacité de correction et une complexité inférieure. Dans ce cas, les codes LDPC peuvent alors représenter une solution. En général, il n'existe pas de code adapté à tous les cas d'utilisation, et on sera souvent amenés à faire un compromis lors du choix de tel ou tel code correcteur, voire à combiner plusieurs codes, en cherchant à utiliser leurs qualités respectives au mieux.

1.4 Codes MDS

Dans la section précédente, nous avons discuté des codes MDS et des codes non-MDS, en précisant que les premiers étaient des codes parfaits, avec une capacité de correction optimale, mais une complexité importante, notamment au niveau du décodage. Cette thèse consistant à présenter et à analyser un code correcteur non-MDS, nous ne décrirons pas l'ensemble des codes MDS, mais en expliquerons le principe fondamental et nous concentrerons sur un de ces codes : les codes de Reed-Solomon.

Les codes MDS, permettant de décoder les k symboles sources d'un mot en n'ayant reçu tout sens-ensemble de k symboles ou plus parmi les n symboles transmis reposent sur le fait qu'il est possible de déterminer l'équation d'un polynôme de degré k à partir de $k+1$ valeurs de ce polynôme. Ainsi, en considérant les k symboles sources d'un mot comme les k valeurs d'un polynôme f, $f(0) = X_0$, $f(1) = X_1$, $\ldots$, $f(k-1) = X_{k-1}$, on peut déterminer l'équation du polynôme f, puis calculer de nouvelles valeurs pour ce polynôme et transmettre les symboles obtenus. Le récepteur qui aura reçu k symboles ou plus pourra alors déterminer la même équation, et recalculer les valeurs de $X_0 = f(0), X_1 = f(1), \ldots, X_{k-1} = f(k-1)$.

Les codes Reed-Solomon, introduits par Irvine Reed et Gustave Solomon en 1960 [49] et appartenant à la classe des codes parfait, ou codes MDS, ont popularisés par leur utilisation dans les communications spatiales, notamment avec les sondes Voyager, ou encore dans les CD. Ils travaillent sur des corps finis, et sont basés sur le sur-échantillonnage d'un polynôme ainsi sur le fait qu'il est possible de déterminer l'équation d'un polynôme de degré t en connaissant $t+1$ de ses valeurs.

La figure 1.13 ci-dessous représente le principe des codes de Reed-Solomon pour le canal à effacements :

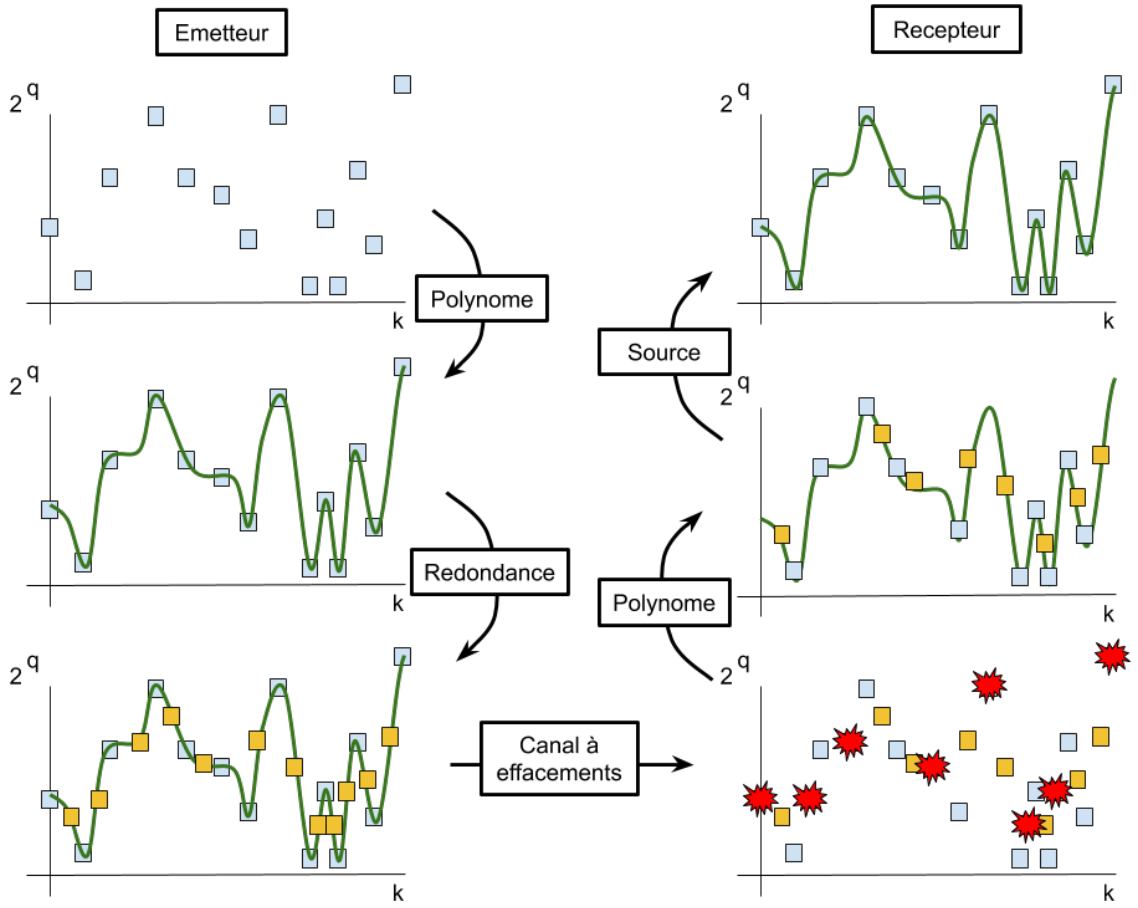


FIGURE 1.13 – Principe des codes de MDS pour le canal à effacements : on peut déterminer l'équation d'un polynôme de degré $k-1$ avec n'importe quel ensemble de k points. On calcule ensuite de nouvelles valeurs de ce polynôme jusqu'à en obtenir n, qu'on transmet. Le récepteur déterminera le même polynôme s'il a reçu k symboles ou plus, et saura alors calculer la valeur des k symboles sources.

Un code Reed-Solomon a une longueur maximale limitée par la taille du corps fini sur lequel il travaille. Un corps de taille q permet de construire des codes Reed-Solomon de taille $q-1$ au maximum. En choisissant comme corps fini $GF(2^p)$, la longueur maximale du code est alors $n_{max} = 2^p - 1$. Pour une longueur de code en particulier, on peut choisir un corps fini assez grand. Néanmoins en pratique, on essaiera d'utiliser un corps fini relativement petit, puisque le coût des opérations sur le corps fini progresse rapidement avec sa taille. On considère donc généralement des corps finis à 2^4, 2^8, et parfois 2^{16} éléments si l'on dispose d'une puissance de calculs importante, mais rarement plus.

On dit de ces codes qu'ils sont des codes parfaits. En 1969, Massey présente un algorithme de décodage basé sur un autre algorithme, introduit auparavant par Berlekamp. L'algorithme de décodage, l'algorithme de Berlekamp-Massey, permet de décoder avec une complexité $O(n \log_2(n))$ sur le canal à erreurs (tel que le canal binaire symétrique présenté précédemment).

1.4.1 Codes Reed-Solomon pour le canal à effacements

Les codes Reed-Solomon sont aussi capables de corriger les effacements. En effet, pour transmettre sur ce type canaux, on multiplie le polynôme correspondant à nos données sources par un polynôme générateur propre au code. Alors que sur le canal à effacements, on évalue en n points différents le polynôme représentant les données source. Ces deux méthodes sont en réalité équivalentes, et on notera que celle que nous utilisons sur le canal à effacements est en celle présentée par Reed et Solomon dans leur publication originale.

Des travaux ont mis en évidence la possibilité de décoder les codes Reed Solomon sur le canal à effacements, notamment avec des transformées de Fourier. Grace à cette technique, la complexité peut être réduite à $O(nlogn)$, ce qui permet d"atteindre des débits de 10Mbps sur des codes de longueur 10000.

1.4.2 Matrices de Vandermonde

Ces ensembles de codes, standards IETF, utilisent une matrice génératrice particulière : la matrice de Vandermonde. Une matrice de Vandermonde V de taille $m \times n$ peut être construite à partir de α, où α est un élément générateur de ce corps fini.

Definition 18. *Soit α un élément générateur d'un corps fini. Soit m et n deux nombres entiers positifs. La matrice de Vandermonde de taille $m * n$ a la forme suivante :*

$$V = \{V_{i,j}\}_{0 \leq i \leq m-1, 0 \leq j \leq n-1} \text{ avec } V_{i,j} = \alpha^{i*j}$$

1	1	1	1	...	1
1	a	a^2	a^3	...	a^{n-1}
1	a^2	a^4	a^6	...	$a^{2(n-1)}$
1	a^m	a^{2m}	a^{3m}	...	$a^{(n-1)*(m-1)}$

TABLE 1.2 – Matrice de Vandermonde

Toute sous-matrice carrée d'une matrice de Vandermonde est inversible, et c'est pour cela que les codes construits sur de cette manière sont des codes MDS. Pour construire un code Reed-Solomon de longueur n et de dimension k, on doit tout d'abord sélectionner un corps fini de taille supérieure ou égale à n, puis construire une matrice de Vandermonde G de taille $n \times k$. Cette matrice est la matrice génératrice du code, qui ne contient pas nécessairement la matrice identité (le code n'est donc pas nécessairement systématique). On pourra néanmoins transformer cette matrice pour y faire apparaître la matrice identité, afin d'obtenir un code systématique.

Encodage

Pour encoder un mot de taille k dont chaque symbole appartient au corps fini, on multiplie par la matrice génératrice : $Y = GX$. On obtient alors un vecteur de longueur n, le mot code qu'on va transmettre sur le canal. Pour des symboles appartenant à $GF(2^8)$, le mot X de k octets sera encodé en un mot Y de n octets, avec $k \leq n \leq 2^8 - 1$.

Décodage

Une fois reçu, le mot peut contenir des effacements à corriger. En général, on n'aura pas à corriger les effacements survenus sur les symboles de redondance, puisque l'information à retrouver est l'information source.

Soit $Y^{'} = \{y_0^{'}, y_1^{'}, ..., y_{n-1}^{'}\}$ ou $y_i^{'} \in GFq \cup E$ le mot de code reçu. Le décodage à pour but de retrouver X en connaissant Y. Supposons que t symboles de Y aient été reçus correctement. Le décodage consiste alors à résoudre un système linéaire de t équations (symboles reçus) à k inconnues (symboles source). Si le nombre de symboles reçus est suffisant, c'est à dire $t \geq k$, alors on sélectionne k éléments de $Y^{'}$, qui forment le mot $Y_{dec}^{'}$ ainsi que les lignes de G correspondantes, qui forment G_{dec} alors la matrice de décodage. On obtient la relation suivante :

$$Y_{dec}^{'} = G_{dec}X$$

Etant donné que G_{dec} est une sous-matrice carrée d'une matrice de Vandermonde, elle est inversible, et on peut retrouver X en inversant la matrice G_{dec}, puis en la multipliant par $Y_{dec}^{'}$:

$$X = G_{dec}^{-1}Y_{dec}^{'}$$

A contrario, si le nombre d'effacements est trop important ($t < k$), le système n'admet pas de solution unique, et le décodage est impossible.

Conclusion

Les codes MDS, dont font partie les codes de Reed-Solomon sont une catégorie de codes correcteurs très utilisée dans le cadre de la fiabilisation des transmissions, puisqu'ils apportent une garantie quant à la probabilité de corriger totalement un bloc de paquets encodés en fonction du nombre d'effacements subis. Leur utilisation est donc particulièrement adaptée aux canaux de transmission dont on connaît précisément la probabilité d'effacements p et que celle-ci fluctue peu avec le temps : la probabilité d'échec du décodage est nulle tant que le récepteur reçoit k symboles ou plus, donc un code convenablement paramétré pour un canal saura décoder à coup-sûr les effacements sur ce canal. On peut citer d'autres codes MDS, tels que les codes BCH [9] par exemple.

Leur complexité en décodage progressant rapidement avec la dimension k du code, ils sont en pratique utilisés pour de petites dimensions, de l'ordre de 2^8 symboles, ce qui peut être contraignant dans le cas d'une transmission subissant de longues pertes impulsives.

Le code que nous avons développé et que nous présenterons par la suite n'est pas un code MDS, et appartient donc à la catégorie des codes non-MDS, tels que les codes LDPC, ou les codes sans rendement, que nous présentons dans la section suivante.

1.5 Codes LDPC

A la différence des codes MDS que nous venons de présenter, il existe de nombreux codes non-MDS, tels que les codes LDPC (pour "Low Density Parity Check"), très utilisés dans l'industrie. Ces codes ont été découverts par Robert G. Gallager, et présentés dans sa thèse [25], publiée en 1963 dans le MIT Press, [26].

Ces codes n'ont pas été utilisés pendant une longue période, et ont été redécouverts 30 ans plus tard par MacKay en 1996 [36] Ces codes sont appelés ainsi car ils ont la particularité d'avoir une matrice de parité possédant une faible densité.

La matrice de parité d'un code LDPC a une majorité de 0 et peu de 1 (d'ou le "Low Density"). Cette propriété nous permet d'utiliser des algorithmes particuliers, basés sur le principe de propagation de croyance ("Belief propagation" en anglais). Ces algorithmes ont une complexité linéaire en la longueur du code, ce qui en fait des algorithmes rapides même sur de grands mots, à la différence des codes MDS : on peut les utiliser sur de grandes quantités de données.

De plus, ils peuvent être adaptés aux canaux à effacements [33]. La capacité de correction de ces codes dépend du poids des mots du code ainsi que leur distance minimale. En général, le poids des mots est grand, mais il peut exister des mots de poids plus faible, ce qui fait que la distance minimale du code est elle aussi faible. Lorsqu'on utilise un décodeur itératif, la capacité de correction d'un code LDPC sera limitées par la longueur des cycles dans le graphe de Tanner du code. On peut utiliser les codes LDPC de façon systématique si on envoie les symboles sources, ou non-systématique si on décide de ne pas le faire. Les codes LDPC ont de plus donné naissance à d'autres codes correcteurs s'en inspirant, tels que les codes Tornados [12], les codes Raptors [54] ou les codes LT [31].

Nous allons maintenant présenter les codes LDPC, en présentant dans un premier temps les moyens à notre disposition pour les représenter : leur graphe de Tanner ou leur matrice de parité.

1.5.1 Représentation des codes LDPC

Graphe de Tanner

On peut représenter les codes LDPC de deux manières : le graphe biparti et la matrice de parité. Ces deux représentations sont équivalentes, et on choisira celle qu'on voudra en fonction du problème à résoudre.

Definition 19. *Un graphe est dit biparti s'il existe deux ensemble de sommets U et V et un ensemble d'arêtes, tels que chaque arête relie un nœud de U à un nœud de V.*

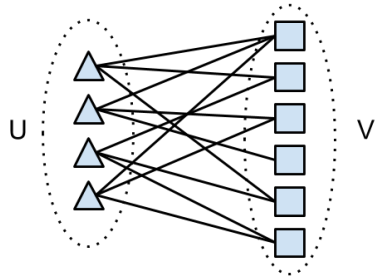


FIGURE 1.14 – Graphe biparti. Les nœuds de l'ensemble U ont leur voisinage dans V et réciproquement.

On peut donc représenter un code LDPC grace à un graphe biparti : les noeuds de gauche (V, noeuds variables) représentent les symboles du code (sources et de redondance), et les noeuds

de droite (C, noeuds contraintes) représentent les contraintes. Un mot du code est valide si pour chaque noeud contrainte, tous les symboles qui sont reliés somment à 0.

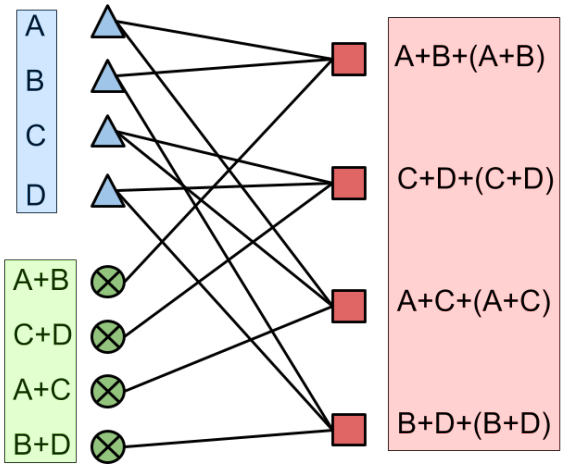


FIGURE 1.15 – Graphe de Tanner d'un code LDPC. Les sommets A, B, C, D en bleu correspondent aux 4 symboles sources, les symboles $A+B, C+D, A+C, B+D$ en vert correspondent aux symboles de redondance calculés lors de l'encodage, et les symboles en rouge correspondent aux nœuds contraintes.

La représentation des codes LDPC par un graphe est présentée en 1981 par R. Michael Tanner dans [57]. Cette représentation sous forme de graphe est utile lorsqu'on souhaite se représenter le principe de passage de messages ("message passing"), sur lequel est basé le décodage itératif. Elle permet de plus d'utiliser la théorie des graphes pour construire et étudier les codes LDPC. En effet nous verrons que les cycles apparaissant dans ces graphes et leur taille, ont un grand impact sur les capacités de correction de ces codes.

Matrice de parité

On peut représenter un graphe biparti à l'aide d'une matrice binaire, aussi appelée *matrice de parité* ou *matrice d'adjacence*. Chaque colonne correspond à un nœud de C et chaque ligne à un nœud de V. L'élément (i, j) de la matrice est non nul si et seulement si il existe une arête reliant le nœud d'indice j de C au nœud d'indice i de V. La matrice de parité définit des équations du système linéaire associé au code (et donc les calculs à effectuer pour obtenir les paquets de redondance). La matrice possédant peu de 1, chaque symbole de redondance sera le résultat de la combinaison linéaire de relativement peu de symboles sources.

Definition 20. *La matrice de parité d'un code LDPC de longueur n et de dimension k sur $\mathcal{F}_q$ est une matrice de rang plein à n colonnes et $n-k$ lignes à éléments dans $\mathcal{F}_q$.*

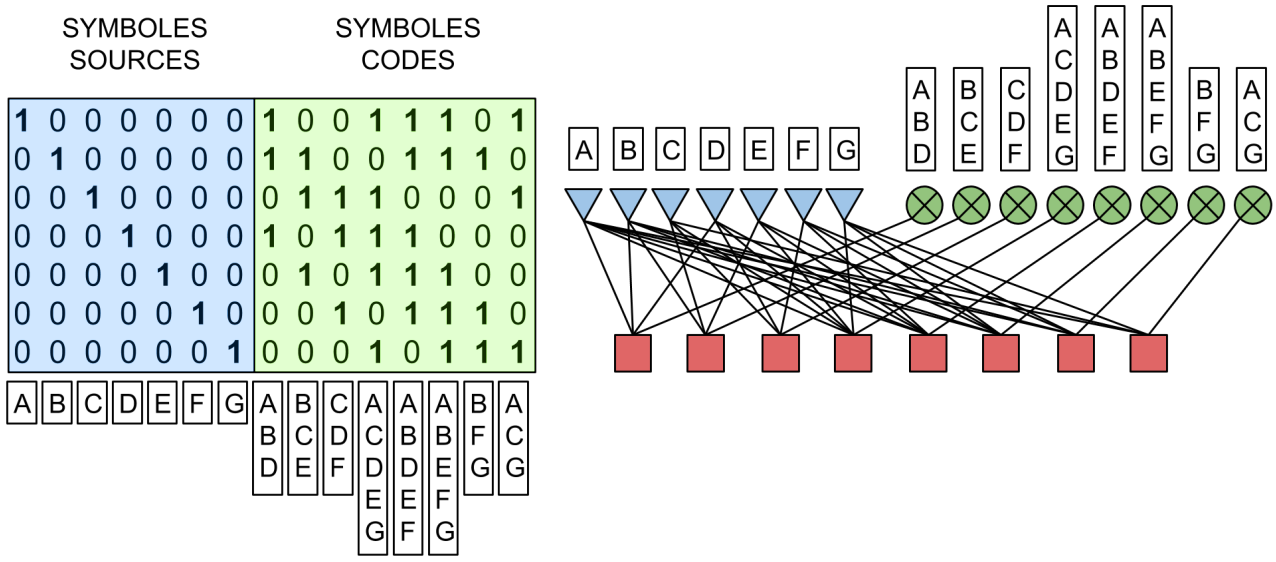


FIGURE 1.16 – Matrice d'un code LDPC et son graphe de Tanner associé.

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

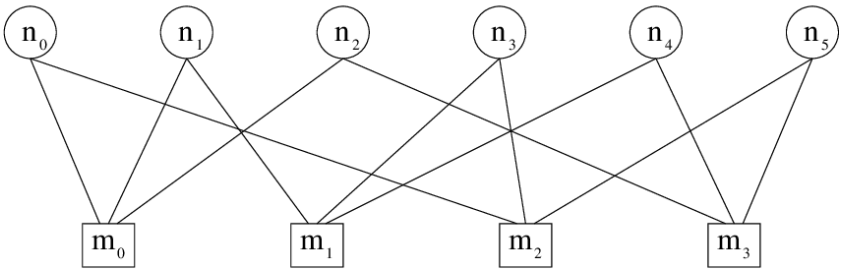


FIGURE 1.17 – Matrice de parité et graphe de Tanner

x_0	$+x_1$	$+x_2$				$=0$
	x_1		$+x_3$	$+x_4$		$=0$
x_0			$+x_3$		$+x_5$	$=0$
		x_2		$+x_4$	$+x_5$	$=0$

TABLE 1.3 – Le système d'équations associé à la matrice H

Pour un code systématique, chaque colonne de la matrice de parité correspond à un symbole. Les k symboles sources sont représentés par les k premières colonnes, et les $n-k$ symboles de redondance correspondent aux $n-k$ dernières colonnes. On appelle $H1$ la partie de la matrice constituée des k premières colonnes, et $H2$ celle constituée des $n-k$ dernières.

Codes LDPC réguliers et irréguliers

Les performances d'un code LDPC, telles qu'on les a présentées précédemment, dépendent grandement du nombre de 1 que contient chaque ligne et chaque colonne. On appelle ces quantités les *degrés* de chaque colonne et de chaque ligne, notés respectivement d_c et d_l. Quand chaque ligne a le même degré, noté d_l, et chaque colonne le même degré, noté d_c, on parle alors de code LDPC *régulier*, en opposition aux codes LDPC *irréguliers*, qui peuvent s'avérer plus performants que les codes LDPC réguliers dans certaines configurations, comme démontré dans [34]. Dans le cas d'un code LDPC régulier, par conservation du nombre total d'entrées non-nulles, on obtient la relation suivante :

$$d_c * n = d_l * (n-k)$$

1.5.2 Encodage des codes LDPC

Pour encoder avec un code LDPC, comme pour tout code linéaire, on multiplie le mot source par sa matrice génératrice. On obtient alors le mot du code désiré. Cette méthode présente cependant un certain nombre d'inconvénients :

- la matrice génératrice doit être connue. Si ce n'est pas le cas, il faut la calculer par une élimination de Gauss sur la matrice de parité, qui est assez coûteuse. On peut effectuer cette élimination en amont, afin qu'elle n'impacte pas le décodage. Cependant, lorsque le code est généré à la volée, on ne peut pas effectuer ce pré-calcul.
- la multiplication par la matrice génératrice (qui est dense à l'inverse de la matrice de parité) est coûteuse en nombre d'opérations (complexité en $O((n-k)n)$).

Afin de répondre à ces inconvénients, deux méthodes peuvent être adoptées : la première consiste à transformer la matrice de parité, afin d'obtenir dans celle-ci une sous-matrice quasiment triangulaire, sur la partie $H2$ de la matrice. La résolution de ce système linéaire plus petit, suivie d'une étape de substitution ("backward substitution"), permet de réduire le coût de l'encodage. La seconde approche considère des codes LDPC dont la matrice possède dès le départ une sous-matrice quasiment triangulaire dans $H2$, afin d'obtenir un encodage de complexité linéaire.

1.5.3 Décodage des codes LDPC

Dans le cas du canal à effacements, le décodage d'un code LDPC consiste à résoudre un système linéaire : les variables sont alors les symboles effacés, et les constantes les symboles reçus. Plusieurs solutions peuvent être adoptées. On peut utiliser un *décodeur itératif* de faible complexité, ou bien utiliser un *décodeur par maximum de vraisemblance*.

Décodage itératif

Le décodage itératif a une complexité linéaire, mais est sous-optimal quand on s'intéresse à la capacité de correction. Il permet néanmoins d'obtenir de relativement bonnes capacités de correction.

Son principe de ces décodeurs est le suivant : dans un graphe où les noeuds correspondent aux symboles sources et de redondance, l'algorithme fait passer l'information le long des arêtes du graphe, et modifie au fur et à mesure la valeur des nœuds suivant l'information délivrée par les arêtes.

Après plusieurs itérations, l'algorithme converge vers les bonnes valeurs des symboles effacés.

Les noeuds variables peuvent prendre trois valeurs lorsqu'on utilise ces codes sur le canal binaire à effacements : $\{0, 1, \emptyset\}$.

Lorsqu'un noeud contrôle est relié à deux messages $\emptyset$, il ne peut rien faire pour le moment. Il retourne alors aux noeuds variables qui lui sont connectés la valeur $\emptyset$.

Si un nœud contrôle reçoit un seul message $\emptyset$, il peut en déduire une nouvelle valeur pour ce nœud, cette valeur étant la somme de tous les autres messages reçus auquel il est connecté (en effet la somme des valeurs des nœuds variables adjacents à un nœud de contrôle doit être égale à zéro).

L'algorithme s'arrête quand plus aucun noeud contrainte n'est relié à un unique noeud variable vallant $\emptyset$. Ce cas de figure peut correspondre à une réussite du décodage (plus aucun noeud variable ne vaut $\emptyset$), ou un échec (tous les noeuds contraintes susceptibles de permettre le décodage d'un noeud variable est relié à deux noeuds variables ou plus).

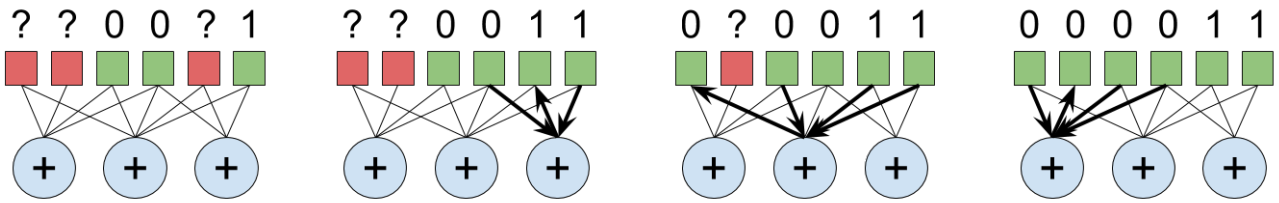


FIGURE 1.18 – Exemple de décodage itératif : le 3ème nœud contrôle permet de réparer le 5ème nœud variable, puis le 2ème nœud contrôle le 1er nœud variable, et enfin le 1er nœud contrôle le deuxième nœud variable.

Avec ce type d'algorithme, on peut réparer au maximum $n - k$ symboles. Le décodage de chaque symbole $\emptyset$ nécessite de sommer $d_r - 1$ symboles. On aura donc au plus $(n - k) * (d_r - 1)$ sommes à effectuer pour le décodage. La complexité de ces décodages est donc linéaire en la dimension du code. Même si le décodage itératif termine sur un succès dans un grand nombre de situations, il se peut qu'il ne puisse pas résoudre le système, même lorsqu'une solution existe. Ce type de décodage est donc sous-optimal, en comparaison avec les décodage par maximum de vraisemblance que nous présentons par la suite. On rencontre cette situation lorsqu'apparait dans notre graphe un "stopping-set", que nous décrivons par la suite. C'est le prix à payer pour obtenir des algorithmes de complexité linéaire.

Décodage par Maximum de Vraisemblance

Le décodage par maximum de vraisemblance (ou décodage ML , pour "Maximum Likelihood") sont une deuxième classe de solutions.

Leur principe est de retourner le mot de code le plus proche (au sens de la distance de Hamming) du mot reçu. Ce mot est le mot qui a le plus de chance d'être le mot initialement

transmis. Ces décodages permettent d'exploiter au maximum les capacités de correction du code, et d'atteindre (ce qui est démontré dans [40] et [46]).

A l'inverse des décodages itératifs, cette optimalité en terme de capacité de correction se paye en temps de calcul. Dans le cas du canal à effacements, les décodeurs par maximum de vraisemblance trouvent toujours la solution au système d'équation associé au code si elle existe.

Stopping Sets

Un "stopping-set" est un ensemble d'effacements qui met le décodeur itératif d'un code LDPC en situation de blocage.

Definition 21. *Soit un code LDPC et son graphe de Tanner. Soit $\mathcal{V}$ un ensemble de nœuds variables, et soit $\mathcal{S}$ un ensemble de nœuds contrôles adjacents à $\mathcal{V}$. On dira que $\mathcal{S}$ est le voisinage de $\mathcal{V}$. L'ensemble $\mathcal{V}$ est un stopping-set de du graphe de Tanner si tous les nœuds de $\mathcal{S}$ sont connectés à au moins 2 nœuds de $\mathcal{V}$.*

Intuitivement, un stopping-set correspond à un ensemble de nœuds variables, tel qu'aucun nœud variable de cet ensemble ne peut être réparé, puisque pour chaque nœud contrôle permettant de le réparer il existe un autre nœud variable effacé empêchant son décodage. Supposons que tous les symboles d'un stopping-set aient subi un effacement : aucun de ces symboles ne peut être reconstruit, car les nœuds contraintes auxquels ils sont adjacents sont incapables de résoudre leur valeur.

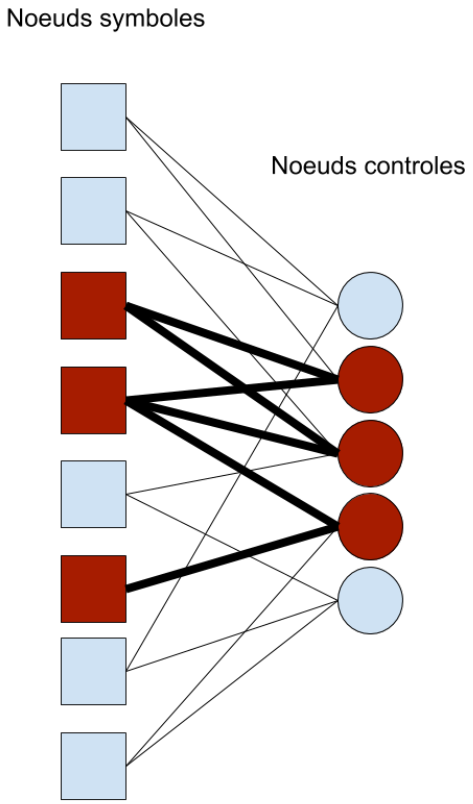


FIGURE 1.19 – Stopping-set empêchant le décodage d'un mot par un code LDPC

Les stopping-set représentent une limitation importante de la capacité de correction des codes à décodage itératif : on souhaitera donc développer des codes correcteurs possédant le moins de stopping-sets possible, et ayant la taille la plus grande possible. La taille des stopping-set d'un code est directement liée à la distance minimale de ce code : en effet, la distance minimale du code nous indique le nombre minimum de paquets effacés réparés à coup sûr. Par conséquent, il ne peut pas exister de stopping-set de taille inférieure à la distance minimale d'un code. De nombreux travaux portent sur le calcul du nombre et de la taille de ces stopping-sets, tels que [18] ou [24].

Conclusion

Les codes LDPC forment une famille de codes correcteurs fréquemment utilisée dans l'industrie. Les différentes techniques d'encodage et de décodage et leurs complexités relativement faibles en font des codes utilisables sur des dispositifs réseaux disposant de peu de mémoire et de puissance de calcul. Le fait qu'ils soient des codes non-MDS, en opposition avec les codes de Reed-Solomon présentés précédemment est un désavantage, mais leur capacité de correction, bien qu'étant sous-optimale, s'avère bonne en pratique, et est souvent suffisante. Le décodage itératif, qui a une probabilité d'échouer même quand on a reçu une quantité d'information supérieure à la quantité d'information initiale présente l'intérêt d'être relativement simple. L'autre technique de décodage, le décodage par maximum de vraisemblance, présente une probabilité de décodage plus importante, bien que sont coût en temps de calcul soit lui-aussi plus important.

Il existe de nombreuses familles de codes LDPC, plus ou moins adaptés aux différentes répartitions des effacements sur le canal de transmission, qui se distinguent par la répartition des 1 dans leur matrice de parité. On pourra par exemple citer les codes LDPC en escaliers ("staircase") [35], standardisés à l'IETF [51].

Le code correcteur que nous présenterons dans la deuxième partie s'apparente grandement aux codes LDPC à décodage itératif. Nous verrons qu'il n'est pas nécessaire dans le cas de notre code de disposer de sa matrice génératrice ou de sa matrice de parité : les combinaisons linéaires à effectuer pour chaque paquet source sont calculées à la volée.

1.6 Codes sans rendement

Les codes MDS et les codes non-MDS linéaires que nous avons présentés précédemment ont un rendement prédéfini, qui vaut $R = k/n$ (qui représente la quantité d'information source k divisée par la quantité d'information transmise après encodage n). Dans le cas d'un canal de transmission dont on connaît la probabilité d'effacements P, on peut paramétrer ces codes de manière à obtenir des codes ayant un rendement adapté à ce canal. Dans le cas ou on ne connaît pas la probabilité d'effacements du canal, ou bien que celle-ci est très importante, on peut s'intéresser à une autre catégorie de codes correcteurs : les codes sans rendement.

Par code sans rendement, on désigne en réalité des codes dont le rendement peut être aussi petit qu'on le souhaite : la dimension du code k (nombre de paquets sources) est fixée, mais sa longueur n (nombre de paquets transmis après encodage) peut varier, et être potentiellement très grande par rapport à k. Ainsi, le rendement du code $R = k/n$ tend vers 0 à mesure que n progresse, et permet de fiabiliser une transmission dont la probabilité d'effacements est très importante.

Il existe cependant des limites à cette quantité :

— Les symboles produits ces codes sont obtenus par combinaisons linéaires des symboles sources. Ces symboles sources étant en nombre fini, il existe un nombre fini de combinaisons linéaires possibles. Pour un code de dimension k, on aura 2^k combinaisons différentes.
— De façon concrète, les symboles sont identifiés par un nombre entier inclus dans leur header : Les bits disponibles pour représenter ces nombres étant en nombre limité, on ne pourra pas avoir une dimension de code infinie. Les codes Raptors, présentés dans [54], possèdent des identifiants codés sur 16 bits, ce qui nous donne $k = 2^16 = 65536$ paquets sources au maximum.

On peut donc obtenir un nombre extrêmement grand de symboles de redondance, ce nombre est limité. On devrait plutôt décrire ces codes "sans-rendement" par des codes à "rendement très faible", ou bien des codes à "rendement non défini".

Les codes sans rendement pouvant être très performants sur des canaux de transmission subissant beaucoup d'effacements, ils sont souvent soumis à des brevets : c'est notamment le cas des codes Raptor, réputés être les codes à effacements les plus performants, qui sont standardisés et brevetés au sein de l'IETF [32], du 3GPP [2] et du DVB [3].

Ces codes sont eux aussi des codes linéaires : chaque paquet encodé puis transmis est le résultat de combinaisons linéaires d'un ou plusieurs paquets sources. Le nombre de paquets sources à combiner entre eux, ainsi que leurs indices sont choisis aléatoirement pour chaque paquet de redondance.

Du point de vue du récepteur, le décodage peut alors se faire de la même manière que pour le décodage itératif des codes LDPC. Les paquets de redondance Y_i qui ne contiendront qu'un seul paquet source X_j seront considérés par le récepteur comme des paquets sources, et les paquets Y_i résultats de combinaisons de plusieurs paquets permettront au fur et à mesure de leur réception de déterminer des paquets sources non-reçus.

Dans cette partie nous présentons les codes "LT", pour "Luby Transform", qui sont les premiers codes sans rendement à avoir été utilisés dans l'industrie.

1.6.1 Codes LT

En 2002, Michael Luby présente dans [31] une classe de codes sans rendement non-systématiques. Ils permettent le calcul des symboles de redondance à la volée. Ces codes sont baptisés "codes LT" ("Luby Transform") en référence à leur inventeur. Comme dit précédemment, ces codes reposent sur la combinaison de paquets choisis aléatoirement parmi les k paquets sources à transmettre pour obtenir itérativement des paquets de redondance.

Codes à très faible rendement

Comme on vient de le voir, désigner ces codes par le terme "sans rendement" est en réalité un abus de langage. En effet, le rendement d'un code vaut $R = k/n$ et dépend donc de sa dimension et de sa longueur. Pour obtenir un code ayant réellement un rendement nul, il faudrait que sa dimension k soit nulle, ce qui n'aurait pas de sens, ou que sa longueur n soit infinie, ce qui est impossible. Ainsi, le rendement minimal d'un code LT est $R_{min} = \frac{k}{2^k}$, qui mais reste strictement positif.

Encodage des codes LT

L'encodage des codes LT est relativement simple et se fait itérativement : à chaque itération i, l'émetteur calcule un paquet Y_i, résultat de combinaisons linéaires de d_i paquets sources.

A chaque itération i, l'émetteur va réaliser ces 5 étapes pour générer un paquet de redondance Y_i :

- Tout d'abord, il initialise le paquet $Y_i = 0$.
- A l'aide d'un générateur de nombres pseudo-aléatoire ("PRNG"), il choisit un nombre d_i, avec $1 \leq d_i \leq k$. Pour cela, il peut utiliser l'indice de l'itération i comme graine pour le PRNG.
- Avec ce même PRNG, il choisit parmi les k paquets sources, d_i paquets qu'il va combiner dans Y_i en utilisant l'opérateur XOR.
- L'émetteur ajoute alors à Y_i la graine qui lui a permis de déterminer d_i puis les paquets à combiner entre eux (dans le header du paquet par exemple).
- Il transmet Y_i.

L'émetteur peut réaliser autant d'itérations qu'il le souhaite, jusqu'à considérer qu'il a calculé et transmis assez de paquets de redondance pour que le récepteur puisse les décoder et obtenir les k paquets sources.

Décodage des codes LT

Le récepteur doit connaître le PRNG utilisé par l'émetteur pour encoder ses paquets de redondance, ce qui va lui permettre de déterminer les degrés et les indices des paquets ayant servi à calculer les paquets de redondance.

Tout d'abord, le récepteur initialise une liste $L_{sources}$ qui va lui servir à conserver la valeur des paquets sources qu'il va décoder, et une liste $L_{redondance}$ qui va servir à conserver les paquets de redondance.

A chaque itération i il reçoit le paquet Y_i, avec une probabilité $1 - p$. Si il le reçoit il réalise alors les calculs suivants :

- Connaissant i (qui a été ajouté à Y_i par l'émetteur), il calcule d_i à l'aide du même PRNG, ainsi que les d_i indices des paquets ayant servi à calculer Y_i.
- Il ajoute Y_i à la liste $L_{redondance}$. Il ajoute aussi ("à coté de Y_i", pour faire simple), le degré d_i de Y_i et la liste des indices des paquets sources ayant servi à le calculer.
- Si $d_i > 1$, et donc que $Y_i = X_{j_0} \oplus X_{j_1} \oplus \cdots \oplus X_{j_{d-1}}$, il cherche dans la liste $L_{sources}$ s'il dispose d'un ou plusieurs paquets sources ayant servis à calculer Y_i : pour chaque paquet X_j pour lequel c'est le cas, il effectue alors le calcul $Y_i = Y_i \oplus X_j$, diminue le degré d_i de Y_i de 1, et retire j de la liste des indices qui lui est associée.
- Tout paquet de la liste $L_{redondance}$ dont le degré d_i vaut 1 a pour valeur celle d'un paquet source X_j, dont l'indice est le dernier indice restant dans la liste qui lui est associée. On retire alors ce paquet de la liste $L_{redondance}$ et on l'ajoute à la liste $L_{sources}$ (si elle ne le contient pas déjà).

Nous présentons ci-dessous un exemple simple de décodage itératif de cet algorithme, avec $k = 6$ paquets sources. Pour simplifier, nous présentons un exemple dans lequel le récepteur reçoit tous les paquets émis par l'émetteur. L'émetteur va transmettre les paquets $Y_0 = X_1 \oplus X_4$, $Y_1 = X_3 \oplus X_4$, $Y_2 = X_2 \oplus X_3 \oplus X_5$, $Y_3 = X_4$, $Y_4 = X_1 \oplus X_2$, $Y_5 = X_0 \oplus X_5$, $Y_4 = X_1 \oplus X_5$, $Y_7 = X_1$.

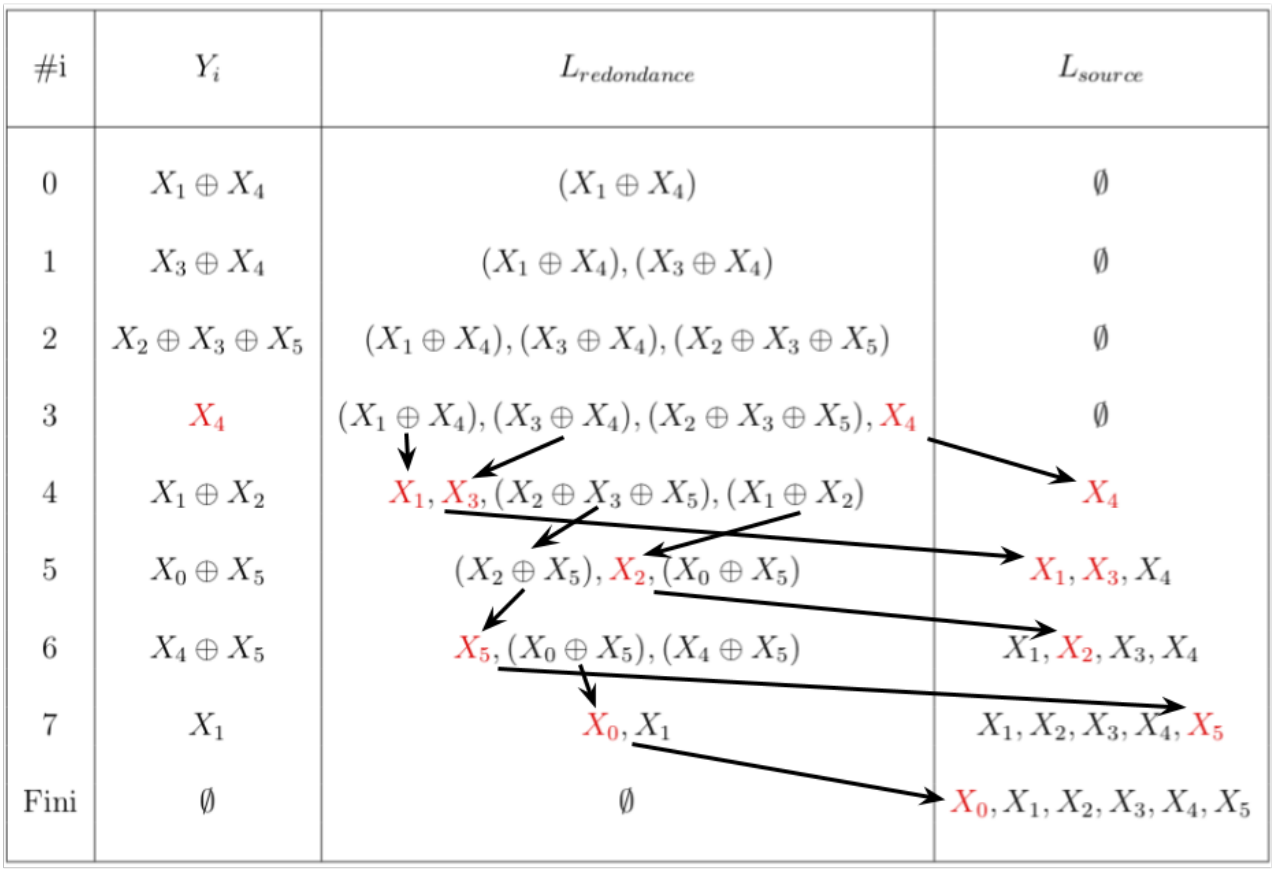


FIGURE 1.20 – Exemple de décodage d'un code LT sur 8 itérations.

L'algorithme de décodage itératif est parvenu à décoder le mot initial de $k = 6$ symboles en 8 itérations. Rappelons qu'ici, le récepteur a pu se contenter de 8 itérations, mais que dans le cas d'une transmission unidirectionnelle, il n'a pas moyen de transmettre cette information à l'émetteur, et que celui-ci est susceptible de transmettre de nombreux paquets par la suite.

Nous avons pris ici un exemple gadget permettant de se représenter le déroulement de l'algorithme de décodage, mais il convient de préciser plusieurs éléments :

— Le hasard a ici bien fait les choses, et nous n'avons jamais reçu deux exemplaires du même paquet de redondance. En effet, il aurait pu arriver que l'émetteur transmette par exemple le paquet $Y_0 = X_1 \oplus X_4$, puis $Y_{17} = X_1 \oplus X_4 = Y_0$. C'est une situation inhérente aux codes LT qu'on souhaite éviter, car recevoir deux exemplaires d'un même paquet est inutile. Sachant que le nombre de combinaisons linéaires qu'on peut obtenir avec k paquets sources vaut 2^k, cette probabilité diminue rapidement donc quand k progresse.

— Le hasard a aussi fait que nous avons reçu des paquets de degré $d_i = 1$, comme les paquets $Y_3 = X_4$ ou $Y_7 = X_1$. Avec un code LT dont les degrés sont choisis totalement au hasard, il se peut qu'on ne reçoive que très peu de paquets de degré 1 voir aucun. Or ce sont ces paquets qui permettent de démarrer l'algorithme de décodage itératif : c'est en recevant $Y_3 = X_4$ qu'à l'itération 5 on peut décoder les paquets X_1 et X_3 et ainsi de suite. Luby apporte une solution à ce problème en proposant une distribution des degrés particulière, la "Robust Soliton Distribution" que nous présentons par la suite.

— A la 8-ième et dernière itération, le récepteur est parvenu à décoder le mot source, et

n'a reçu que les paquets de degré 1 $Y_3 = X_4$ et $Y_7 = X_1$. Du point de vue du récepteur, le code n'est donc pas un code systématique. Cependant, s'il avait eu à attendre plus d'itérations, il aurait pu recevoir les $k = 6$ paquets sources et aurait donc eu à faire à un code systématique.

— Dans cet exemple, la liste $L_{redondance}$ contient au maximum 4 paquets, aux itérations 3 et 4. Dans d'autres cas de figure, il se pourrait que cette liste soit bien plus longue. Ce n'est pas tellement un problème au niveau de la complexité en temps : bien qu'il faille à chaque itération chercher si un paquet de la liste L_{source} se trouve dans un paquet de la liste $L_{redondance}$, des implémentations intelligentes utilisant une table de hashage par exemple permettent d'obtenir des algorithmes dont la complexité progresse faiblement en fonction de la taille de ces listes. Cependant, cela peut finir par poser un problème en termes de consommation mémoire, puisque ces listes peuvent finir par être très longues, surtout quand la dimension du code k, la taille des paquets et le taux d'effacements p sont grands.

Consommation mémoire

La consommation mémoire d'un code LT peut s'avérer importante et poser un problème dans le cas de matériel embarqué par exemple. Du point de vue de l'émetteur tout d'abord, l'encodage se fait de façon statique : durant tout le temps de l'encodage et à chaque itération, il doit disposer de tous les paquets sources, ce qui n'est pas le cas pour les codes LDPC par exemple, qui peuvent encoder un flux d'information. De plus, en termes de sécurité de l'information, ce point peut poser un problème de rémanence : on souhaite ne pas conserver de l'information trop longtemps sur le même support, car ceci peut laisser des traces exploitables pour de potentiels intrus.

Du point de vue du récepteur, on a vu que la liste $L_{redondance}$ peut finir par être très longue, surtout quand k et le taux d'effacements p est grand. Si aucun moyen n'est mis en place du côté du récepteur pour vérifier si un paquet de redondance a déjà été reçu auparavant, la longueur de cette liste est potentiellement infinie. A contrario si le récepteur sait déterminer qu'un paquet Y_j est le résultat des mêmes combinaisons linéaires qu'un paquet Y_i reçu auparavant (faisable avec une table de hachage par exemple), cette liste a alors pour longueur maximale $longueur_{max} = 2^k - k$. On obtient ce cas de figure quand on a reçu tous les paquets de redondance possibles, et aucun paquet source. Sachant qu'il existe 2^k combinaisons linéaires au total, et k paquets de degré 1, on a bien $2^k - k$ paquets de redondance de degré strictement supérieur à 1. Cependant, ce cas de figure a assez peu de chance d'advenir, et à la réception de n'importe quel paquet de degré 1, le récepteur pourra décoder entièrement les k paquets sources.

1.6.2 Distribution des degrés

On a vu que ne pas recevoir de paquets de degré $d_i = 1$ pouvait s'avérer problématique, puisque ceux-ci sont nécessaires au bon déroulement du décodage itératif. Sachant que pour un code LT de degré k, on peut obtenir des paquets de redondance de degré $1 \leq d_i \leq k$, et donc k degrés différents, la probabilité qu'un paquet de redondance soit un paquet de degré 1 vaut $1/p$ si on choisit les degrés de manière uniforme. Pour k grand, on aura alors une faible probabilité d'envoyer (et d'obtenir) des paquets de degré 1.

Afin de répondre à ce problème, Luby propose donc une distribution particulière pour la sélection des degrés, appelée "Ideal Soliton Distribution". Elle dépend d'un paramètre k, nombre

entier qui correspond à la dimension k d'un code LT (et donc du nombre de degrés que peuvent prendre les paquets de redondance d'un code LT de dimension k).

Definition 22. *On appelle "Ideal Soliton Distribution" de paramètre k la fonction de masse P, qui a tout $x \in [1;k] \cap N$ associe la probabilité d'apparition*

$$\begin{cases} P(x=1) = \frac{1}{k} \\ P(x=i) = \frac{1}{i*(i-1)}, i \in [2,...,k] \end{cases}$$

Afin de visualiser la probabilité d'apparition d'un degré suivant la loi de Soliton, nous présentons ci-dessous les deux courbes correspondant à cette loi pour $k = 20$, et $k = 50$.

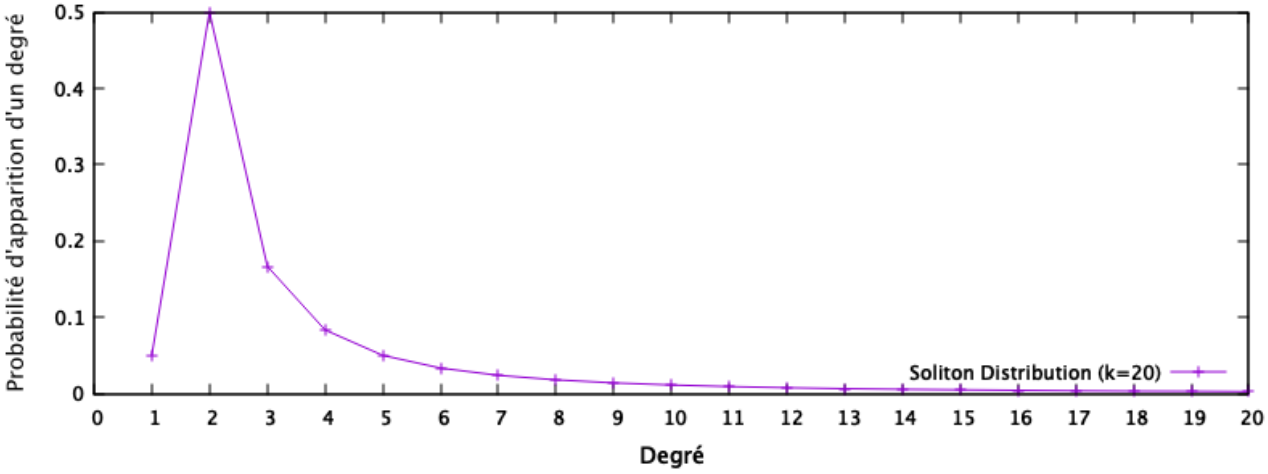


FIGURE 1.21 – Probabilité d'obtenir un degré avec la distribution de Soliton pour k=20

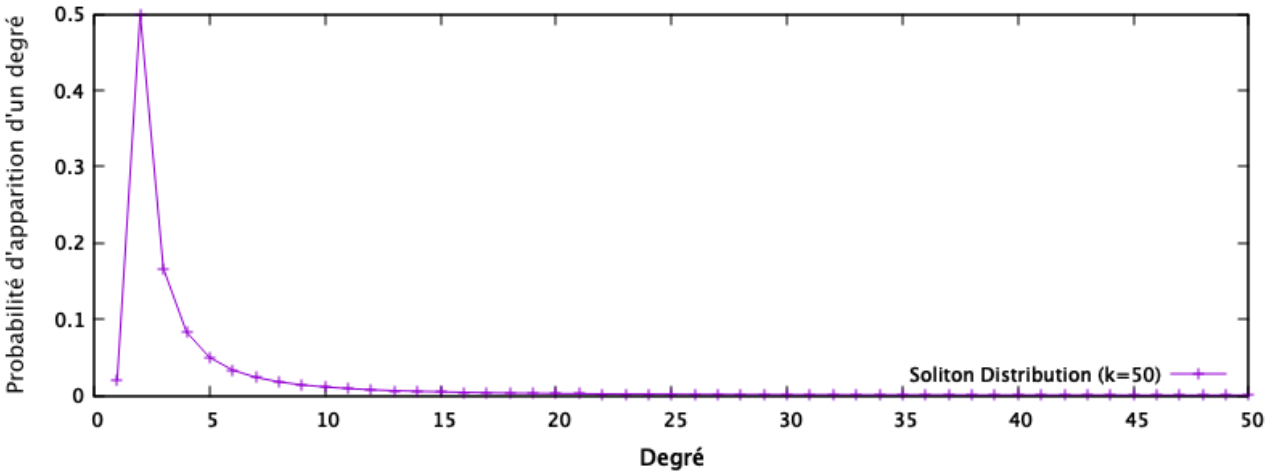


FIGURE 1.22 – Probabilité d'obtenir un degré avec la distribution de Soliton pour k=50

Nous proposons de plus une figure représentant la probabilité d'obtenir un degré inférieur à une certaine valeur. Cette correspond à la somme des valeurs de la figure 1.23.

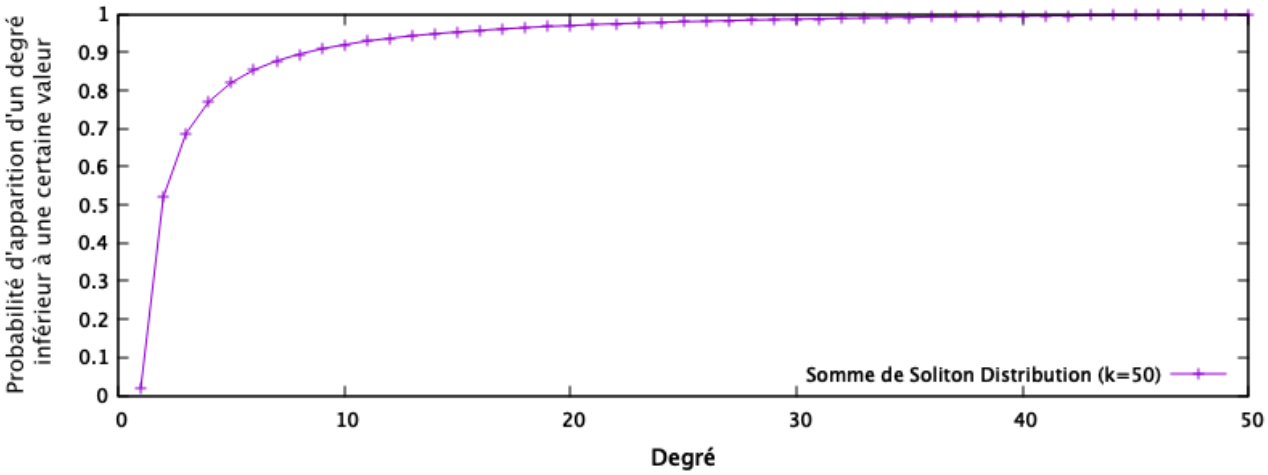


FIGURE 1.23 – Probabilité d'obtenir un degré inférieur à une certaine valeur avec la distribution de Soliton pour k=50. On voit que la probabilité d'obtenir un degré inférieur ou égal à k vaut bien 1.

Cependant, cette distribution de probabilité n'est pas utilisée en pratique pour plusieurs raisons. Tout d'abord pour des grandes dimensions, la probabilité d'obtenir des paquets de degré 1 reste faible. De plus, il y a une probabilité non-nulle qu'un paquet source n'ait été Xoré dans aucun paquet de redondance, et qu'on ne puisse donc le décoder par aucun moyen. En pratique, on utilise une autre loi de distribution, très semblable à la loi Ideal Soliton Distribution, appelée Robust Soliton Distribution.

Pour l'Ideal Soliton Distribution, on remarque un "pic" pour $d = 2$: la probabilité d'obtenir un paquet de degré d est maximale pour $d = 2$. La Robust Soliton Distribution consiste à ajouter un autre pic pour une plus grande valeur de d, qui aura pour effet de transmettre de temps en temps un paquet de redondance dans lequel on aura Xoré beaucoup de paquets sources, en espérant ainsi qu'aucun paquet source n'aura été oublié.

Pour ajouter ce pic au degré $d = M$, on utilise une fonction t définie sur le même intervalle $[1; k]$ que pour l'Ideal Soliton Distribution, et on ajoute ses valeurs à la distribution originale avant de les normaliser pour que leur somme vaille 1. La fonction t requiert deux nouveaux paramètres, $\delta \in R$ et $M \in [1; k]$. On définit alors la valeur $R = M/k$, et on obtient l'équation de t :

$$\begin{cases} t(x = i) = \frac{1}{x*M}, i < M \\ t(x = M) = \frac{ln(R/\delta)}{M} \end{cases}$$

Nous présentons dans la figure 1.24 ci-dessous les valeurs de la fonction t pour $k = 20$ et pour différentes valeurs de M et de δ, et dans la figure 1.25 les probabilités d'obtenir un degré avec la Robust Soliton Distribution pour $M = 5, 10, 15$ et $\delta = 0.1$.

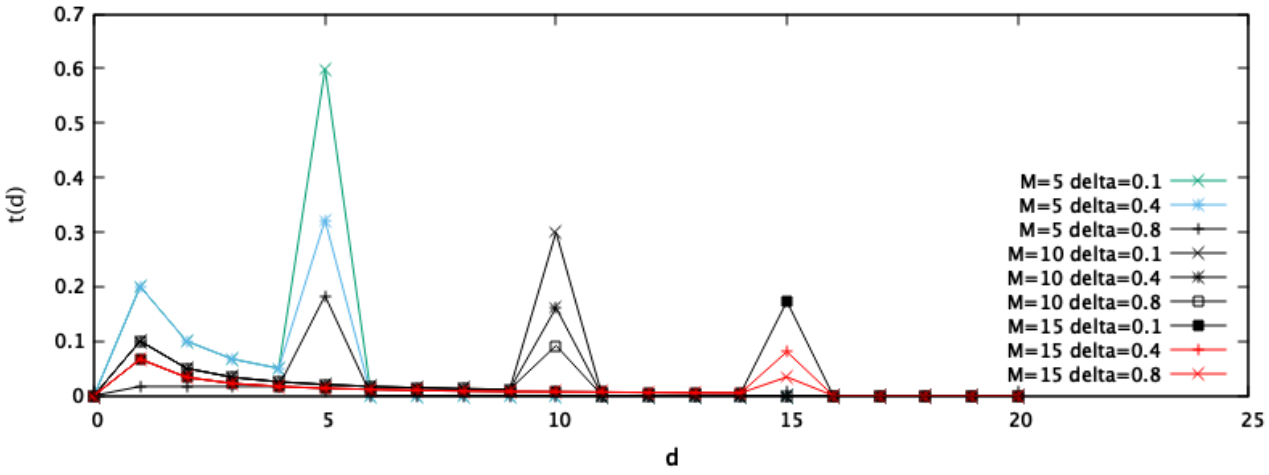


FIGURE 1.24 – Valeurs de la fonction t pour différents paramètres M et δ, avec $k = 20$. Ces valeurs sont à ajouter aux valeurs données par l'Ideal Soliton Distribution, puis à normaliser pour que leur somme vaille 1.

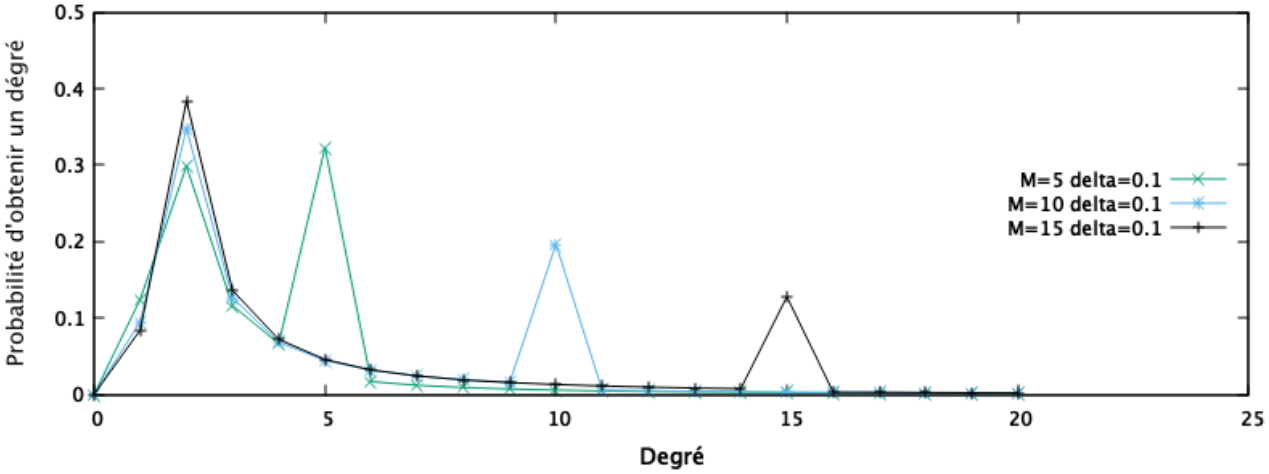


FIGURE 1.25 – Probabilités d'obtenir un degré en utilisant la Robust Soliton Distribution pour $M = 5, 10$ et 15, $\delta = 0.1$ et $k = 20$.

Générateur de nombres pseudo-aléatoire et propriété intellectuelle

Les symboles de redondance sont obtenus en Xorant d symboles sources choisis de manière aléatoire, où d, le degré du symbole calculé, dépend d'une distribution de particulière. On peut utiliser un PRNG (*Pseudo-Random Numbers Generator*, *Générateur de nombres pseudo-aléatoire* en Français) dont la graine sera l'indice du symbole de redondance. Le PRNG prendra donc l'indice du symbole comme graine, puis pourra calculer d, et enfin choisir les indices des paquets sources à combiner pour obtenir ce symbole.

Quel que soit le canal à effacements sur lequel on transmet nos symboles, les codes LT sont asymptotiquement optimaux : même pour une probabilité d'effacements très importante, et po-

tentiellement inconnue par l'émetteur, la transmission de suffisamment de symboles permettra au récepteur de recouvrer toute l'information source. Ceci est cependant vrai à condition que la distribution des degrés soit bien choisie. Ainsi, si on ne transmet que des symboles de très forts degrés, mais jamais de symboles de degré 1, on ne saura jamais décoder un seul symbole source.

Pour cela, Luby propose la distribution dite "Robust Soliton" (RSD). Pour cette distribution, le degré choisi pour encoder un symbole de redondance est en moyenne de $log(k)$. Cette distribution permet au récepteur de décoder à partir de $k + log2(k)/\sqrt{k}$ symboles reçus en moyenne. Lorsque la dimension du code tend vers l'infini, le nombre de symboles nécessaires au décodage tend vers k, ce qui fait que les codes LT tendent asymptotiquement vers des codes parfaits, les codes MDS. Ce n'est cependant pas le cas pour de petites dimensions et longueurs de code, ce qui peut représenter un inconvénient. De plus, on n'utilisera pas ces codes si le canal n'a qu'une très faible probabilité d'effacements, et on leur préférera des codes LDPC construits spécialement pour ce type de canaux.

Conclusion

Les codes sans rendement que nous avons présentés sont particulièrement adaptés aux canaux de transmission dont la probabilité d'effacements n'est pas connue par l'émetteur, ou très importante. Le nombre de symboles de redondance différents pouvant être calculés puis transmis, dépendant du nombre k de paquets sources, peut être très important mais n'est pas infini.

Un cas d'application de ces codes est par exemple le cas du "carrousel" décrit en introduction, qui consiste à devoir distribuer de l'information depuis une source vers un destinataire dont la connexion est intermittente : à chaque fois que le récepteur se connecte, il reçoit de nouveaux symboles de redondance, et pourra au bout d'un certain temps obtenir l'information source, peu importe les instants auxquels la connexion a été établie.

Leur complexité en encodage et en décodage faible en font des codes pratiques pour du matériel disposant de peu de puissance de calcul. Cependant, leur consommation en mémoire progresse avec le taux d'effacements de la transmission et la dimension du code, ce qui peut s'avérer problématique pour le récepteur quand celui-ci dispose de peu de mémoire. De plus, l'émetteur doit disposer durant la transmission de toute l'information source à transmettre, ce qui pose des problèmes de rémanence dans le cas d'information sensible.

Dans le cas ou la probabilité d'effacements p du canal est faible et connue, on préférera un code LDPC ayant le rendement adéquat, voire un code MDS si le cas d'utilisation et le matériel le permettent.

Chapitre 2

Fauxtraut

2.1 Introduction

Dans cette partie, nous présentons donc la solution adoptée par Thales pour fiabiliser la transmission de données via l'Elips-SD. Il s'agit d'un code correcteur, informellement baptisé Fauxtraut pour "Fast Algorithm Using XOR To Repair Altered Unidirectionnal Transmissions".

Ce code correcteur est un code à effacements LDPC irrégulier systématique, développé et implémenté pendant ma deuxième année de Master en apprentissage chez Thales pour fiabiliser la transmission de blocs de paquets UDP via l'Elips-SD. Il a été développé au niveau industriel, intégré dans plusieurs produits et est actuellement utilisé par des clients de Thales. L'améliorer (tout en le modifiant le moins possible), l'étudier théoriquement, le tester, le paramétrer le mieux possible et le comparer avec d'autres codes correcteurs ont fait l'objet de ces trois dernières années de thèse.

Son fonctionnement est le suivant : les paquets de redondance (ou paquets codes) sont obtenus par combinaisons linéaires des paquets sources, via l'opérateur XOR. Les paquets doivent être de taille constante mais quelconque et potentiellement très grande, ainsi que les blocs de paquets que nous encodons. Il n'y a théoriquement pas de limite à la quantité d'information que nous pouvons encoder et décoder avec ce code correcteur, si ce n'est celle disponible sur le support. Le choix des paquets sources à Xorer entre eux pour produire des paquets codes est entièrement déterminé par les indices des paquets sources, modulo un ensemble de nombres premiers entre eux P, sur lequel l'émetteur et le récepteur devront s'être accordés avant la transmission. Ce point précis du partage de la valeur de P n'est pas adressé dans ce manuscrit, et il pourrait être intéressant d'étudier les différentes solutions envisageables permettant à l'émetteur et au récepteur de se passer de cet accord préalable (par exemple, en incluant dans les paquets transmis ce fameux ensemble P).

Il s'agit d'un code LDPC irrégulier, qui peut par conséquent être représenté aussi bien par une matrice de parité que par un graphe de Tanner. Nous verrons d'ailleurs que dans certains cas particuliers, il s'agit d'un code rectangulaire : nous verrons qu'il est possible de représenter son système d'équations par un graphe de Rook, et simplifier son graphe de Tanner.

Nous verrons que l'ensemble de nombres premiers entre eux P détermine totalement les combinaisons linéaires à effectuer pour produire nos paquets codes, et affecte sa complexité calculatoire, sa consommation mémoire, sa longueur, sa dimension, sa distance minimale, et par conséquent sa capacité de correction.

Ce code est un code non-MDS, donc nécessairement sous-optimal en terme de capacité de

correction, et s'avère moins performant que d'autres codes non-MDS existants, dans un cas d'utilisation général, notamment parce qu'il possède une distance minimale faible. Les codes tels que des codes LDPC ou des codes LT ont une meilleure capacité de correction que ce code, mais imposent des contraintes que notre cas d'utilisation nous interdit.

Il présente aussi certains avantages, tels que sa complexité calculatoire linéaire en encodage et en décodage, et est particulièrement performant dans le cas de pertes de paquets impulsives ("Burst losses", décrites précédemment).

Ce code correcteur est un code LDPC irrégulier. Les lignes de la matrice génératrice sont calculées à la volée pour chaque symbole source, en fonction de son indice. Ceci nous permet notamment de ne pas avoir à conserver les paquets sources pour procéder à l'encodage ou au décodage : les paquets sources sont traités les uns après les autres, potentiellement dans le désordre dans les symboles de redondance. De plus, l'algorithme d'encodage et de décodage se faisant à la volée, ils nécessitent une consommation mémoire assez faible : ni le récepteur ni l'émetteur n'ont à conserver les paquets pour l'algorithme d'encodage et décodage, et peuvent se contenter de les transmettre (pour l'émetteur) ou de les utiliser (pour le récepteur). Cela représente un atout non négligeable lorsqu'on désire transmettre de l'information confidentielle. En effet, stocker des données informatiques sur un support d'information produit nécessairement de la rémanence d'information. Stocker de l'information sur un support peut être risqué, puisque même si cette information est finalement supprimée du support, il est possible par certaines techniques d'obtenir des indices sur celle-ci, voire de la recouvrer totalement. Le fait de ne pas avoir à conserver les paquets sources longtemps, quand bien-même ceux-ci seraient cryptés, présente un atout non-négligeable dans notre cas d'utilisation.

Ainsi, dans ce chapitre, nous présentons le fonctionnement de ce code correcteur, notamment via les pseudo-codes des algorithmes d'encodage et de décodage. Nous présenterons des exemples de l'algorithme pour des valeurs concrètes pour faciliter la compréhension et l'appréhension du code par le lecteur. Nous présentons ensuite les analyses théoriques de sa complexité calculatoire, linéaire en la longueur du code (k), sa consommation mémoire linéaire en le nombre de paquets de redondance ($n-k$), ainsi que sa distance minimale, en fonction de l'ensemble P qui le définit.

Nous présenterons une série de simulations, confirmant sa complexité calculatoire et sa consommation mémoire : ces tests sont réalisés sur un large panel d'exemples et de cas d'utilisation (taille des paquets, tailles des blocs traités, nombre de blocs à traiter, probabilité d'effacements sur le canal , etc...). Ce code correcteur a une implémentation relativement simple ce qui présente l'avantage, dans le cadre du développement et de l'intégration de composants logiciels sécurisés, d'être facile à étudier et certifier. La DGA ("Direction Générale des Armées") a rapidement pu certifier à Thales que ce code ne présentait pas de faille de sécurité. Enfin, sa complexité calculatoire et sa consommation mémoire font qu'il peut être exécuté sur du matériel consommant peu d'énergie et disposant de peu de mémoire (par exemple, dispositif réseau embarqué, ou "internet des objets").

Le décodage de ce code étant un décodage itératif, il est nécessairement confronté au problème des Stopping-sets exposé précédemment. Calculer le nombre et la taille des Stopping-sets d'un code LDPC est un problème très étudié de nos jours. Dans le cas de notre code, dont le système d'équations peut être représenté par un graphe de Rook (pour certains paramétrages particuliers), le problème devient alors assez simple à énoncer et élégant, mais soulève néanmoins un certain nombre de questions de combinatoire difficiles, ce qui en fait un problème intéressant. Nous avons établi un certain nombre de formules closes, de bornes supérieures et inférieures, ainsi que de fonctions tendant asymptotiquement vers ce nombre de Stopping-sets,

et espérons que certains de ces résultats pourront être adaptés pour d'autres types de codes correcteurs, voire d'autres cas d'application que la théorie des codes correcteurs.

2.1.1 Cas d'utilisation et motivations

Il convient ici de rappeler le cas d'utilisation du code correcteur que nous allons décrire par la suite. Nous cherchons à fiabiliser une transmission passant par une diode réseau, Elips-SD. Utiliser une diode réseau permet de transmettre de l'information confidentielle d'un site non-sécurisé à site sécurisé, tout en s'assurant que celle-ci ne pourra pas en ressortir. Nous faisons l'hypothèse que l'utilisateur n'a pas besoin de recevoir l'information en temps-réel (il ne s'agit pas d'une communication téléphonique ou radio, mais plutôt d'une transmission de fichiers). Par conséquent, il est tout à fait possible d'utiliser un code par bloc, avec des blocs de potentiellement très grande taille. L'utilisateur peut, de notre point de vue, attendre un peu pour recevoir l'information qui lui a été transmise.

La diode ne perturbe pas le signal, ou très peu, et la transmission est de bonne qualité : il n'y a quasiment pas de perte au niveau binaire. Néanmoins des pertes peuvent advenir au niveau de la couche logicielle, notamment dus à un engorgement du buffer au niveau du récepteur. Ainsi, les pertes se font au niveau des paquets reçus, et plutôt de manière impulsive : quand on perd des paquets, il s'agit souvent de pertes en rafale, les paquets effacés étant souvent consécutifs. Précisons que le taux de perte binaire, le taux de perte paquets, et aucun effet de la diode sur la transmission n'ont fait l'objet d'étude ou de discussion dans ce manuscrit.

Nous n'avons donc pas le choix du protocole de transmission, et avons ainsi à fiabiliser une transmission UDP. En effet, les utilisateurs de l'Elips-SD préfèrent utiliser un protocole simple, sûr et éprouvé. De plus, utiliser des protocoles plus bas-niveaux peut représenter des failles en termes de sécurité, choix que les utilisateurs de la diode préfèrent ne pas avoir à faire. Pendant cette thèse, nous nous sommes donc imposés de fiabiliser un flux de paquets UDP.

2.1.2 Définitions

Soit P un ensemble de nombres premiers entre eux de taille $d >= 1$, $k_{max} = PPCM(P)$, $0 \leq k \leq k_{max}$ et $n = \Sigma(P) + k + 1$, et $len >= 0$, la fonction $PPCM$ correspondant au *plus petit commun multiple* d'un ensemble de nombres entiers passé en argument. Un code Fauxtraut de dimension n et de longueur k est un code LDPC irrégulier systématique, dont les mots sources sont de longueur k et les mots codes de longueur n. Le nombre de symboles de redondance est fixé, mais on dispose d'un degré de liberté sur le nombre de symboles sources d'un bloc, et le choix de P va influer sur le temps de calcul de l'encodage et du décodage, la capacité de réparation, la longueur et le rendement du code.

Soit un mot source X de longueur k et le mot code Y de longueur n (contenant des symboles ou paquets de taille len). Les k premiers symboles du mot code Y sont les k symboles sources de X, et les $(n-k)$ symboles restants sont les symboles de redondance obtenus par combinaisons linéaires des k premiers (c'est en cela que ce code correcteur est systématique).

Soit $Y = \phi(X, P)$ l'application de len^k dans len^n représentant l'application d'encodage de ce code, et qui au mot X de longueur k constitué de symboles de longueur len fait correspondre le mot code Y de longueur n constitué de symboles de longueur len.

L'application ϕ retournera le mot Y suivant le système d'équations ci-dessous (2.1) :

$$Y_i = \begin{cases} X_i & & 0 \leq i < k \\ \bigoplus X_j & j \bmod P_0 = i - k & 0 \leq i - k < P_0 \\ \bigoplus X_j & j \bmod P_1 = i - k - P_0 & 0 \leq i - k - P_0 < P_1 \\ \bigoplus X_j & j \bmod P_2 = i - k - P_0 - P_1 & 0 \leq i - k - P_0 - P_1 < P_2 \\ \dots & & \\ \bigoplus X_j & j \bmod P_d = i - k - P_0 - \dots - P_{d-1} & 0 \leq i - k - P_0 - \dots - P_{d-2} < P_{d-1} \\ \bigoplus X_j & 0 \leq j < k & i = k - P_0 - P_1 - \dots + P_{d-1} \end{cases}$$

FIGURE 2.1 – Le mot $Y = \phi(X, P)$ est obtenu en concaténant les k paquets/symboles du mot/bloc X à P_0, puis P_1, puis P_2 ...paquets de redondance. Le dernier paquet de redondance est le résultat du Xor des k paquets sources.

La première ligne du système présenté en figure 2.1 correspond aux k premiers paquets du bloc encodé, les paquets sources. La deuxième ligne correspond aux P_0 premiers paquets codes, obtenus par combinaisons linéaires de paquets sources, chaque paquet source ayant été une seule fois dans un seul autre paquet code, la troisième ligne correspond aux P_1 paquets code, ...La dernière ligne étant le résultat du Xor des k paquets sources.

Pour différencier les symboles sources des symboles de redondance, et faciliter la compréhension et la lisibilité du code, on introduit la notation $C_{i,j}$ pour représenter les symboles de redondance, avec $i \in P$ et $O \leq j < i$. On peut alors noter

$$C_{i,j} = \bigoplus X_{i*k+j}$$

Chaque symbole $C_{i,j}$ peut être représenté simplement par un système d'équations :

$$\begin{cases} \begin{cases} C_{P_0,0} &= X_0 \oplus X_{P_0} \oplus X_{2*P_0} \oplus X_{3*P_0} \dots \\ C_{P_0,1} &= X_1 \oplus X_{P_0+1} \oplus X_{2*P_0+1} \oplus X_{3*P_0+1} \dots \\ C_{P_0,2} &= X_2 \oplus X_{P_0+2} \oplus X_{2*P_0+2} \oplus X_{3*P_0+2} \dots \\ \dots \end{cases} \\ \begin{cases} C_{P_1,0} &= X_0 \oplus X_{P_1} \oplus X_{2*P_1} \oplus X_{3*P_1} \dots \\ C_{P_1,1} &= X_1 \oplus X_{P_1+1} \oplus X_{2*P_1+1} \oplus X_{3*P_1+1} \dots \\ C_{P_1,2} &= X_2 \oplus X_{P_1+2} \oplus X_{2*P_1+2} \oplus X_{3*P_1+2} \dots \\ C_{P_1,3} &= X_3 \oplus X_{P_1+3} \oplus X_{2*P_1+3} \oplus X_{3*P_1+3} \dots \\ \dots \end{cases} \\ \begin{cases} C_{1,0} &= X_0 \oplus X_1 \oplus X_2 \oplus X_3 \oplus X_4 \dots X_{k-1} \end{cases} \end{cases}$$

FIGURE 2.2 – Système d'équations permettant d'obtenir les paquets/symboles de redondance à partir des paquets/symboles sources. Les paquets de redondance sont ici renommés pour faciliter la compréhension et la lecture.

Ce code peut être représenté par sa matrice génératrice assez simple. Ses k premières lignes forment une matrice identité $k * k$ correspondant aux paquets sources, puis les $n - k$ lignes

suivantes les paquets de redondance. Afin de mieux visualiser la manière de calculer Y à partir de X et P, nous présentons la matrice génératrice d'un code Fauxtraut pour $P = [3, 4, 5]$ et $k = 20$ en figure (2.4). Cette matrice est une matrice de dimensions $k * n = 20 * 33$. La matrice identité $20 * 20$ représente la partie systématique du code (les 20 paquets sources sont envoyés), et la partie inférieure (une matrice 20*13) représente le système de 13 équations du code.

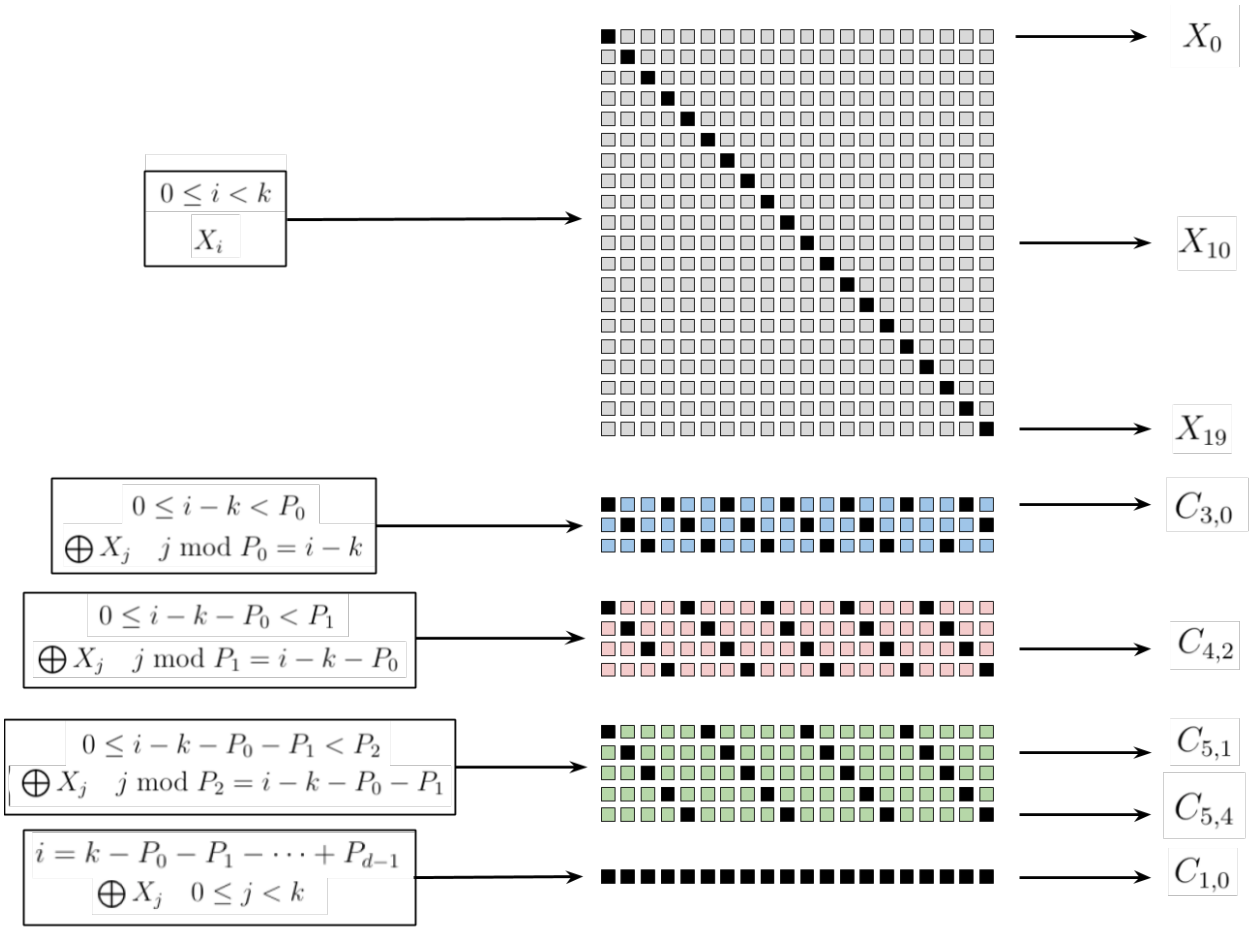


FIGURE 2.3 – Matrice génératrice d'un code Fauxtraut pour $P = [3, 4, 5]$ et $k = 20$. Pour plus de clarté, les paquets codes du mot Y sont renommés suivant l'élément de P qui a servi à le calculer.

La figure 2.4 présente le système d'équations du code correcteur, avec renommage des paquets de redondance de la forme $C_{i,j}$. Le paquet $C_{P_2,4}$ sera le résultat du XOR des paquets sources d'indices 4, $4 + P_2$, $4 + 2 * P_2$, $4 + 3 * P_2$,....

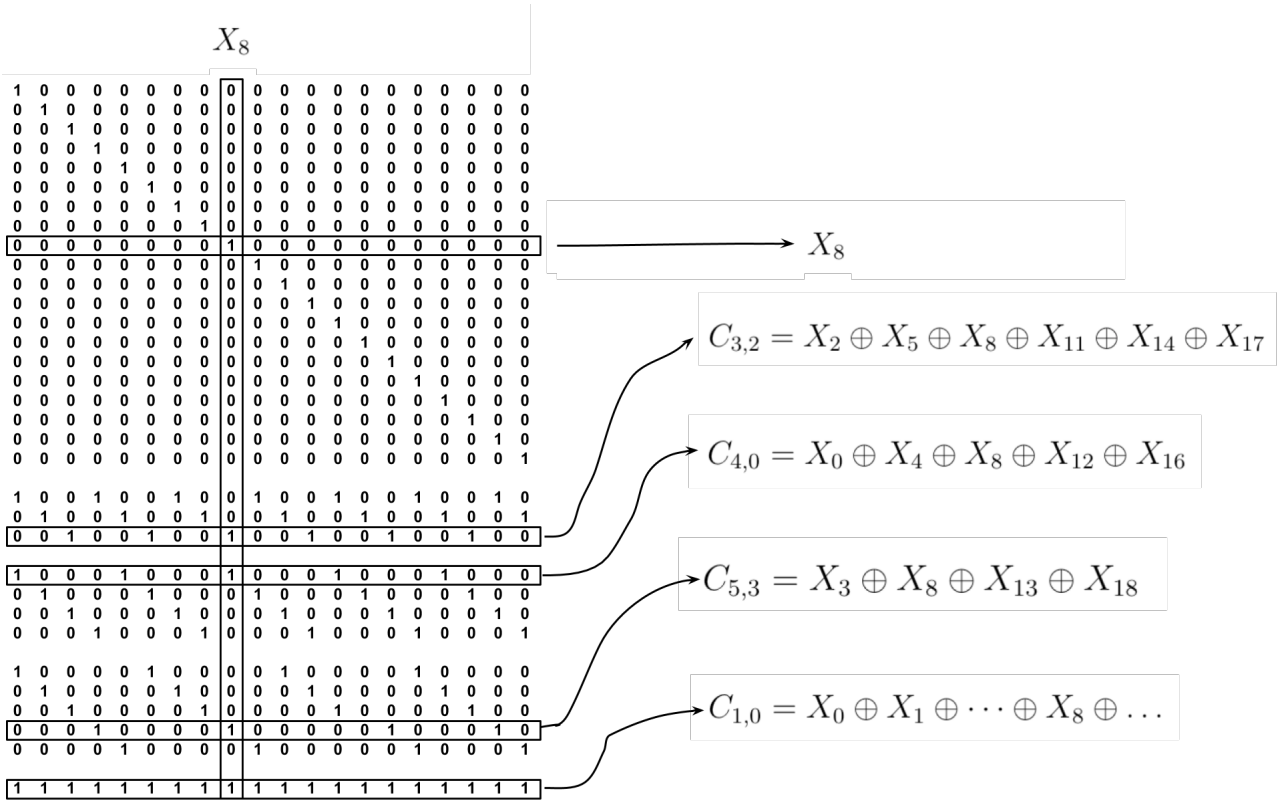


FIGURE 2.4 – Matrice génératrice d'un code Fauxtraut pour $P = [3, 4, 5]$ et $k = 20$. Le paquet X_8 a été XORé dans les paquets $C_{3,2}$, $C_{4,0}$, $C_{5,3}$ et $C_{1,0}$ avant d'être transmis.

Pour plus de clarté, prenons comme dans nos figures $P = [3, 4, 5]$. On a $k_{max} = PPCM(3, 4, 5) = 3 * 4 * 5 = 60$, on peut donc choisir $0 \leq k \leq 60$ symboles sources à envoyer par bloc, et nous ajouterons à chaque bloc $\Sigma(P) + 1 = 3 + 4 + 5 + 1 = 13$ paquets de redondance. Prenons $k = 20$, pour obtenir un code dont le rendement sera $R = 20/33 = 0.606\ldots$. Le premier symbole X_0 doit être Xoré dans les symboles codes $C_{3,0}$, $C_{4,0}$, $C_{5,0}$ et $C_{1,0}$ puis peut être transmis. Le deuxième symbole X_1 est Xoré dans $C_{3,1}$, $C_{4,1}$, $C_{5,1}$ et $C_{1,0}$, et le symbole X_i sera Xoré dans $C_{4,i \mod 4}$, $C_{5,i \mod 5}$, $C_{9,i \mod 9}$ et $C_{1,0}$. A la fin de l'encodage tous les paquets sources ont été transmis et chacun a été Xoré dans $d = 4$ paquets codes. La complexité de l'encodage est donc proportionnelle à $(card(P) + 1) * k = (d + 1) * k$.

Une fois le calcul des paquets codes terminé ceux-ci peuvent être transmis. Il faut que le récepteur puisse distinguer les paquets sources des paquets codes, ce qui est fait en ajoutant cette information au début du buffer des paquets UDP (dans le cas de notre implémentation utilisant les RawPackets, nous mettons ces informations dans le header sur deux champs de 32 bits chacun). On envoie les paquets codes dans l'ordre $C_{3,0}$, $C_{3,1}$, $C_{3,2}$,..., $C_{5,4}$ et enfin $C_{1,0}$.

Le récepteur reçoit les paquets sources au fur et à mesure, et peut commencer à calculer ses paquets codes en effectuant les mêmes calculs que l'émetteur, potentiellement dans le désordre. A la fin de la réception d'un bloc, si celui-ci a subi des effacements, le récepteur peut en réparer un certain nombre, en résolvant itérativement le système d'équations présenté en figure (2.4).

2.2 Pseudo-codes

Nous présentons dans cette section les pseudo-codes de l'encodage et du décodage d'un code Fauxtraut. Comme dit précédemment, l'émetteur et le récepteur doivent s'être accordés sur un ensemble P de nombres premiers entre eux et sur la longueur k des blocs à traiter (ainsi que sur la taille des paquets transmis len). Ces paramètres sont donc requis pour ces deux algorithmes.

Pseudo-code de l'algorithme d'encodage

Algorithm 1 Encodage

Require: P est un ensemble de d nombres premiers entre eux (de préférence trié, bien que ce ne soit pas obligatoire) et X est l'ensemble des k paquets sources de taille len à transmettre (qui peuvent être traités les uns après les autres, en streaming). Cet algorithme d'encodage produit $n - k = \Sigma(P) + 1$ paquets codes de taille len qui vont eux aussi être transmis.

```
1 : function ENCODAGE(X, P)
2 :     C_{1,0} = 0
3 :     for i ← 0 to d do
4 :         for j ← 0 to P_d do
5 :             C_{p_j,(i mod p_j)} = 0
6 :     for X_i in X do
7 :         for j ← 0 to d do
8 :             C_{p_j,(i mod p_j)} = C_{p_j,(i mod p_j)} ⊕ X_i
9 :         TRANSMETTRE(X_i)
10 :    for i ← 0 to d do
11 :        for j ← 0 to P_d do
12 :            TRANSMETTRE(C_{i,j})
13 :
14 :    for i ← 0 to P_0 do
15 :        C_{1,0} = C_{1,0} ⊕ C_{P_0,i}
16 :    TRANSMETTRE(C_{1,0})
```

Les lignes **3** à **5** correspondent à l'étape d'initialisation des $N - K = \Sigma(P) + 1$ paquets codes du bloc, dont les buffers doivent être remis à 0 pour chaque bloc : Chaque paquet code ayant un buffer de len bits, cette étape a une complexité en $\mathcal{O}(\Sigma(P) + 1) * len$. Cette étape ne correspond pas réellement à un calcul, et on pourrait par exemple imaginer disposer de beaucoup d'espace mémoire vierge, et qu'à chaque nouveau bloc à encoder, on choisisse une nouvelle zone mémoire pour les buffers de nos paquets codes. Cette étape est donc nécessaire au procédé d'encodage, mais n'en fait pas réellement partie.

Les lignes **6** à **9** correspondent au traitement d'un paquet source X_i dans les $|P|$ paquets codes qui lui sont associés, avant sa transmission. Cette étape a une complexité en $\mathcal{O}(|P| * k *$

len) : chaque paquet source X_i est XORé dans les $|P|$ paquets codes lui correspondant, et le buffer des paquets sources représente lui aussi *len* bits. On remarque que dans cette boucle, nous ne XORons pas les paquets sources dans le paquet $C_{1,0}$: nous effectuerons ces calculs de façon plus efficace, une fois tous les paquets codes calculés, et les paquets sources transmis.

Une fois les k paquets sources traités et transmis puis tous les paquets codes calculés, l'émetteur peut alors transmettre les $n-k$ paquets codes (ce qui correspond aux lignes **10** à **12** de l'algorithme).

Les lignes **14** et **16** correspondent au calcul du paquet $C_{1,0}$ avant sa transmission, qui correspond au Xor de tous les paquets sources X_i du bloc. On remarque qu'il est possible de réduire le nombre de Xors à effectuer pour calculer $C_{1,0}$: au lieu de Xorer les k paquets sources du bloc, on peut se contenter de ne Xorer que les P_0 paquets codes correspondant à P_0, $C_{P_0,0}$, $C_{P_0,1}$, $\ldots C_{P_0,P_0-1}$. En effet, rappelons que

$$C_{P_0,0} = X_0 \oplus X_{P_0+0} \oplus X_{2*P_0+0} \cdots$$
$$C_{P_0,1} = X_1 \oplus X_{P_0+1} \oplus X_{2*P_0+1} \cdots$$
$$C_{P_0,2} = X_2 \oplus X_{P_0+2} \oplus X_{2*P_0+2} \cdots$$

et que par conséquent, $C_{1,0} = X_0 \oplus X_1 \oplus X_2 \cdots \oplus X_{k-1} = C_{P_0,0} \oplus C_{P_0,1} \cdots \oplus C_{P_0,P_0-1}$.

On voit donc qu'il est possible de calculer $C_{1,0}$ plus rapidement qu'en Xorant les k paquets sources. On remarque aussi que ce calcul peut se faire sur les paquets codes correspondant à P_0 aussi bien qu'à P_1, P_2, ..., et on choisira de préférence de les faire sur les paquets codes du plus petit élément de P. La complexité de cette étape est donc en $\mathcal{O}(min(P) * len)$.

Ce pseudo-code nous permet d'observer que la complexité de l'encodage est majorée par la boucle de traitement des paquets sources : ceux sont deux boucles For imbriquées, qui parcourent le bloc X de k paquets, et qui pour chacun de ces paquets les Xor d fois. Ainsi, la complexité de l'algorithme d'encodage est bien en $\mathcal{O}(d * k * len)$. En effet, il ne faut pas oublier que l'opérateur $\oplus$ est en réalité l'opérateur Xor appliqué à *len* octets, et donc la complexité de l'algorithme est proportionnelle à la taille des paquets transmis.

Pseudo-code de l'algorithme de décodage

Nous présentons ici le pseudo-code de l'algorithme de décodage, que nous avons divisé en deux sous-algorithmes afin de faciliter la lecture et la compréhension. En effet, comme vu précédemment, l'algorithme de décodage consiste à voir les paquets reçus comme les variables d'un système d'équations, et les paquets non-reçus comme les inconnues de ce système. A mesure que le récepteur reçoit des paquets, il effectue les calculs permettant de réduire ce système d'équations : cette étape correspond à notre premier sous-algorithme (*"Initialisation des tableaux et traitement des paquets reçus"*).

Une fois que le récepteur sait qu'il ne recevra plus de paquets pour le bloc actuel, le récepteur peut alors lancer l'algorithme de décodage sur le bloc (ie : résoudre le système d'équations associé à notre code, induit par l'ensemble des paquets effacés). Cette étape correspond alors au deuxième algorithme, l'algorithme de décodage itératif.

Algorithm 2 Initialisation des tableaux et traitement des paquets reçus

Require: P est un ensemble de d nombres premiers entre eux (le même ensemble que celui de l'émetteur). R, RR, # et ## sont des tableaux qu'on va initialiser puis se servir durant l'algorithme.

```
1 : function INITIALISATION                          ▷ A exécuter avant la transmission
2 :     for P_i in P do
3 :         RR[P_i] = 0
4 :         ##[P_i] = 0
5 :         for j ← 0 to P_i do
6 :             R[P_i][j] = 0
7 :             #[P_i][j] = 0

8 : function RECEPTION_SOURCES(X_i)                  ▷ A la réception d'un paquet source
9 :     for P_j in P do
10 :        R[P_j][i mod P_j] = R[P_j][i mod P_j] ⊕ X_i
11 :        #[P_j][i mod P_j] + +
12 :    UTILISER(X_i)

13 : function RECEPTION_CODES(C_{P_i,j})             ▷ A la réception d'un paquet code
14 :    if C_{P_i,j} == C_{1,0} then
15 :
16 :        for P_i in P do
17 :            RR[P_i] = RR[P_i] ⊕ C_{P_i,j}
18 :            ##[P_i] + +
19 :    else
20 :        R[P_i][j] = R[P_i][j] ⊕ C_{P_i,j}
21 :        #[P_i][j] + +
22 :        RR[P_i] = RR[P_i] ⊕ C_{P_i,j}
23 :        ##[P_i] + +
```

Cet algorithme est divisé en 3 sous-fonctions.

La fonction $INITIALISATION()$, dans laquelle on initialise 4 tableaux :

— R, un tableau de $\Sigma(P)$ paquets initialisés à 0 et #, un tableau de $\Sigma(P)$ nombres entiers initialisés à 0.

— RR, un tableau de $|P|$ paquets initialisés à 0 et ##, un tableau de $|P|$ nombres entiers initialisés à 0.

Les tableaux R et RR vont respectivement contenir les paquets que le récepteur va calculer au fur et à mesure qu'il reçoit des paquets sources ou codes : ils correspondent aux nœuds contraintes d'un code LDPC. Les tableaux # et ## vont contenir le nombre de paquets qu'on aura reçus et Xorés dans les cases des tableaux R et RR correspondantes : ils nous serviront lors de l'algorithme de décodage à déterminer rapidement si oui ou non le récepteur peut réparer un paquet effacé avec le paquet qu'il a calculé.

La fonction $RECEPTION_{SOURCES}(X_i)$ correspond aux traitements à effectuer lorsqu'on reçoit un paquet source X_i. Comme on l'a vu précédemment, on va Xorer X_i dans les paquets codes correspondant, $C_{P_j,i \mod P_j}$, au bon endroit dans le tableau R. Afin de se souvenir qu'on

a Xoré ce paquet dans chaque code, on incrémente de 1 les $|P|$ cases de # correspondant. La fonction $UTILISER(X_i)$ (ligne 13) correspond à l'utilisation que va faire le récepteur du paquet X_i, et nous ne faisons pas d'hypothèse sur celle-ci.

La fonction $RECEPTION_{CODES}(C_{P_i,j})$ correspond aux traitements à effectuer lorsqu'on reçoit un paquet code $C_{P_i,j}$. Elle est elle-même divisée en deux sous-cas :

- Si le paquet code reçu est $C_{1,0}$, on le Xore dans les $|P|$ cases du tableau RR, et on incrémente toutes les cases du tableau ##.
- Si le paquet $C_{P_i,j}$ n'est pas le paquet $C_{1,0}$, on le Xore dans la case lui correspondant dans le tableau R, et dans la case du tableau RR correspondant à P_i (et on incrémente les bons compteurs de # et ##).

Le récepteur effectue les mêmes Xor que ceux effectués par l'émetteur à la réception d'un paquet. A la fin de la transmission du bloc

- si il a reçu le paquet code $C_{P_i,j}$ et tous les paquets sources correspondant, la case du tableau $R[P_i][j]$ contiendra un paquet dont le buffer est nul, et la case du tableau $\#[P_i][j]$ contiendra le nombre de Xor effectués par l'émetteur pour calculer $C_{P_i,j}$ (augmenté de 1 puisqu'on Xore aussi le paquet code).
- si la case $\#[P_i][j]$ contient exactement le nombre de Xor effectués par l'émetteur pour calculer $C_{P_i,j}$, le récepteur sait que la case $R[P_i][j]$ contient la valeur du buffer d'un des paquets effacés associé à ce paquet code (source ou code).
- enfin, si $\#[P_i][j]$ contient une valeur inférieure au nombre de Xor effectués pour calculer $C_{P_i,j}$, alors il ne peut pas encore déterminer la valeur d'un paquet effacé : on est face à une équation (du système d'équations associé à notre code) qui a plus d'une inconnue.

Les tableaux RR et ## ont la même utilité, mais permettent eux de réparer des paquets codes $C_{P_i,j}$, ou le paquet code $C_{1,0}$. Le paquet $C_{1,0}$ est le résultat du Xor de tous les k paquets sources du bloc. Mais il est aussi égal au Xor de tous les paquets codes correspondant à P_0, ainsi que de tous les paquets codes correspondant à P_1, puis P_2... Si on a reçu tous les paquets codes correspondant à P_i ($C_{P_i,0}, C_{P_i,1}, \ldots, C_{P_i,P_i-1}$, on pourra recalculer le paquet $C_{1,0}$, et réciproquement, on pourra calculer le paquet $C_{P_i,j}$ si on a reçu les $P_i - 1$ autres paquets codes et le paquet $C_{1,0}$.

A la fin de la réception d'un bloc, le récepteur doit déterminer les paquets sources effacés (qu'il rangera dans un tableau $X_{EFFACES}$) et les paquets codes effacés (qu'il rangera dans un tableau $C_{EFFACES}$). Il dispose en plus des 4 tableaux R, RR, # et ##, qu'il va utiliser lors de l'algorithme de décodage itératif présenté ci-après.

Algorithm 3 Decodage

Require: $X_{EFFACES}$ est l'ensemble des paquets sources effacés, $C_{EFFACES}$ est l'ensemble des paquets codes effacés, P est un tableau de d nombres premiers entre eux, R, RR, $\#$ et $\#\#$ sont les tableaux obtenus après réception et traitement des paquets du bloc

1 : **function** DECODAGE($X_{EFFACES}, C_{EFFACES}, R, \#, RR, \#\#, P$)
2 : OK = true
3 : **while** OK **do** ▷ Décodage itératif
4 : OK = false
5 : **for** (X_i in $X_{EFFACES}$) **do** ▷ Pour chaque paquet source X_i à réparer
6 : **for** (P_j in P) **do** ▷ Pour chaque élément $P_j \in P$
7 : $Objectif = \lfloor k/P_j \rfloor$
8 : **if** ($i \mod P_j < k \mod P_j$) **then**
9 : $Objectif = Objectif + 1$
10 : **if** ($\#[j][i \mod P_j] == Objectif$) **then** ▷ On a reçu assez de paquets
11 : $X_i = R[j][i \mod P_j]$ ▷ On a la valeur de X_i
12 : $X_{EFFACES} = X_{EFFACES} \setminus X_i$ ▷ On retire X_i des paquets à réparer
13 : $RECEPTION_{SOURCES}(X_i)$ ▷ On traite X_i comme si on l'avait reçu
14 : $OK = true$
15 : *break* ▷ Pour ne pas réparer X_i plusieurs fois
16 : **for** ($C_{P_i,j}$ in $C_{EFFACES}$) **do**
17 : **if** ($C_{P_i,j} == C_{1,0}$) **then** ▷ Pour le paquet code $C_{1,0}$
18 : **for** (P_m in P) **do**
19 : **if** ($\#\#[P_m] == P_m$) **then** ▷ On a reçu assez de paquets codes
20 : $C_{1,0} = RR[m]$ ▷ On a la valeur de $C_{1,0}$
21 : $C_{EFFACES} = C_{EFFACES} \setminus C_{1,0}$ ▷ On retire $C_{1,0}$ des paquets à réparer
22 : $RECEPTION_{CODES}(C_{1,0})$ ▷ On traite $C_{1,0}$ comme si on l'avait reçu
23 : $OK = true$
24 : break ▷ Pour ne pas réparer $C_{1,0}$ plusieurs fois
25 : **else** ▷ Pour les autres paquets codes
26 : $Objectif = \lfloor k/P_i \rfloor$
27 : **if** ($j < k \mod P_i$) **then**
28 : $Objectif = Objectif + 1$
29 : **if** ($\#[P_i][j] == Objectif$) **then** ▷ On a reçu assez de paquets sources
30 : $C_{P_i,j} = R[P_i][j]$ ▷ On a la valeur de $C_{P_i,j}$
31 : $C_{EFFACES} = C_{EFFACES} \setminus C_{P_i,j}$ ▷ On retire $C_{P_i,j}$ des paquets à réparer
32 : $RECEPTION_{CODES}(C_{P_i,j})$ ▷ On traite $C_{P_i,j}$ comme si on l'avait reçu
33 : $OK = true$
34 : *break* ▷ On sort de la boucle pour ne pas réparer $C_{P_i,j}$ plusieurs fois
35 : **else**
36 : **if** ($\#\#[P_i] == P_i$) **then** ▷ On a reçu assez de paquets codes
37 : $C_{P_i,j} = RR[P_i]$ ▷ On a la valeur de $C_{P_i,j}$
38 : $C_{EFFACES} = C_{EFFACES} \setminus C_{P_i,j}$ ▷ On retire $C_{P_i,j}$ des paquets à réparer
39 : $RECEPTION_{CODES}(C_{P_i,j})$ ▷ On traite $C_{P_i,j}$ comme si on l'avait reçu
40 : OK = true
41 : break ▷ Pour ne pas réparer $C_{P_i,j}$ plusieurs fois

Cet algorithme est donc un algorithme de décodage itératif, correspondant à la boucle $WHILE$ qui va de la ligne **3** à la ligne **41**. En effet, il prend en entrée une liste de paquets sources ($X_{EFFACES}$) et de paquets codes ($C_{EFFACES}$), et cherche à les réparer itérativement en les retirant de ces listes au fur et à mesure.

En entrant dans la boucle, le booléen OK est remis à $false$ (ligne **4**). Dans cette boucle $WHILE$, on va d'abord chercher à réparer des paquets sources (de la ligne **5** à la ligne **15**), puis des paquets codes (de la ligne **16** à la ligne **41**). Lorsqu'on parvient à réparer au moins un paquet, le booléen OK prend alors la valeur $true$, ce qui permettra de relancer une itération (ligne **3**). Ainsi, si on a réparé un ou plusieurs paquets sources ou codes lors d'une itération, les tableaux R, RR, # et ## ont été modifiés, et on est susceptible de réparer d'autres paquets à la prochaine itération. A contrario, si aucun paquet n'a pu être réparé (soit parce qu'il n'y a plus de paquet à réparer, soit parce qu'on est face à un Stopping-Set), on sortira alors de la boucle. Intéressons-nous plus précisément aux calculs effectués au sein d'une itération :

— Tout d'abord, on parcourt l'ensemble des paquets sources de la liste $X_{EFFACES}$. Pour un paquet X_i de cette liste, on parcourt l'ensemble des $|P|$ compteurs qui lui correspondent dans le tableau #. Pour chacun de ces compteurs, il existe une valeur $Objectif$ correspondant au nombre de paquets qu'il faut avoir reçus pour réparer X_i. Cette valeur vaut $Objectif = \lfloor k/P_j \rfloor$ (ligne **7**) dans le cas ou on a choisi $k = PPCM(P)$, ou doit être augmentée de 1 dans certains cas si on a choisi $k < PPCM(P)$ (lignes **8** et **9**). On remarquera que la valeur $Objectif$ correspond en réalité au degré du nœud contrainte auquel on s'intéresse. Lorsqu'un de ces compteurs atteint la valeur $Objectif$, on sait alors que le paquet correspondant dans le tableau R contient la valeur du paquet X_i. En effet, on a reçu le paquet code et tous les autres paquets sources correspondant à ce nœud contrainte, et le seul paquet manquant est X_i : on a résolu une équation de notre système d'équations. On peut alors utiliser X_i comme un paquet reçu, en appliquant sur celui-ci la fonction $RECEPTION_{SOURCE}(X_i)$, qui va effectuer les Xors et augmenter les compteurs adéquats. Dans le cas ou on est parvenu à réparer le paquet X_i, il faut sortir de la boucle qui parcourt les nœuds contraintes lui correspondant, afin de ne pas réparer plusieurs fois X_i, ce qui poserait des problèmes au niveau des Xors et des compteurs qui lui sont associés.

— Ensuite, qu'on soit parvenu ou non à réparer un ou des paquets sources, on parcourt l'ensemble des paquets codes de la liste $C_{EFFACES}$. Il faut alors distinguer deux cas : le cas du paquet $C_{1,0}$ (des lignes **17** à **24**) et celui des autres paquets codes (des lignes **25** à **41**).

On va pouvoir réparer le paquet $C_{1,0}$ si, pour un élément P_m de P, on a reçu tous les paquets codes correspondant. En effet, $C_{1,0}$ est le résultat du Xor de tous les paquets sources, ainsi que le résultat du Xor de tous les paquets codes correspondant à un élément de P. On se sert des $|P|$ compteurs du tableau ## pour déterminer si on peut réparer $C_{1,0}$, et sa valeur se trouvera à la case correspondante dans RR.

Il existe deux moyens de réparer un paquet code C_{P_i}, j (autre que $C_{1,0}$) : si on a reçu tous les paquets sources correspondant à ce paquet code, la valeur de R correspondant à ce paquet contient la valeur de C_{P_i}, j (on a effectué exactement les mêmes calculs que l'émetteur). Si on a reçu le paquet $C_{1,0}$ et tous les autres paquets codes correspondant à P_i, on a alors la valeur de C_{P_i}, j dans la case de RR lui correspondant. Là-encore, on se sert des valeurs de # et ## pour déterminer si oui ou non on peut réparer $C_{P_i,j}$.

Exemple d'application des algorithmes d'encodage et de décodage

Appliquons les algorithmes d'encodage et de décodage présentés précédemment sur un exemple concret. Prenons $P = [3, 4, 5]$ et $k = 20$, et donnons des valeurs arbitraires aux 20 paquets sources :

$X_0 = 12, X_1 = 1, X_2 = 20, X_3 = 31, X_4 = 51, X_5 = 20, X_6 = 37, X_7 = 2, X_8 = 10, X_9 = 10, X_{10} = 11, X_{11} = 40, X_{12} = 51, X_{13} = 28, X_{14} = 21, X_{15} = 12, X_{16} = 31, X_{17} = 60, X_{18} = 39, X_{19} = 37.$

Nous prenons des petites valeurs pour faciliter la lecture, mais rappelons que les paquets transmis peuvent être de très grande taille (mais constante). Ici, nous prenons des valeurs inférieures à 64, et pourrions donc envoyer des paquets dont le buffer ferait 6 bits.

La première étape de l'encodage consiste à calculer les $3 + 4 + 5 = 12$ paquets codes $C_{P_i,j} \neq C_{1,0}$. Ces calculs sont représentés dans la figure (2.5) ci-dessous :

X_i	$P_0 = 3$	$P_1 = 4$	$P_2 = 5$
X_0	$C_{3,0} = 00 \oplus 12 = 12$	$C_{4,0} = 00 \oplus 12 = 12$	$C_{5,0} = 00 \oplus 12 = 12$
X_1	$C_{3,1} = 00 \oplus 01 = 01$	$C_{4,1} = 00 \oplus 01 = 01$	$C_{5,1} = 00 \oplus 01 = 01$
X_2	$C_{3,2} = 00 \oplus 20 = 20$	$C_{4,2} = 00 \oplus 20 = 20$	$C_{5,2} = 00 \oplus 20 = 20$
X_3	$C_{3,0} = 12 \oplus 31 = 19$	$C_{4,3} = 00 \oplus 31 = 31$	$C_{5,3} = 00 \oplus 31 = 31$
X_4	$C_{3,1} = 01 \oplus 51 = 50$	$C_{4,0} = 12 \oplus 51 = 63$	$C_{5,4} = 00 \oplus 51 = 51$
X_5	$C_{3,2} = 20 \oplus 20 = 00$	$C_{4,1} = 01 \oplus 20 = 21$	$C_{5,0} = 12 \oplus 20 = 24$
X_6	$C_{3,0} = 19 \oplus 37 = 54$	$C_{4,2} = 20 \oplus 37 = 49$	$C_{5,1} = 01 \oplus 37 = 36$
X_7	$C_{3,1} = 50 \oplus 02 = 48$	$C_{4,3} = 31 \oplus 02 = 29$	$C_{5,2} = 20 \oplus 02 = 22$
X_8	$C_{3,2} = 00 \oplus 10 = 10$	$C_{4,0} = 63 \oplus 10 = 53$	$C_{5,3} = 31 \oplus 10 = 21$
X_9	$C_{3,0} = 54 \oplus 10 = 60$	$C_{4,1} = 21 \oplus 10 = 31$	$C_{5,4} = 51 \oplus 10 = 57$
X_{10}	$C_{3,1} = 48 \oplus 11 = 59$	$C_{4,2} = 49 \oplus 11 = 58$	$C_{5,0} = 24 \oplus 11 = 19$
X_{11}	$C_{3,2} = 10 \oplus 40 = 34$	$C_{4,3} = 29 \oplus 40 = 53$	$C_{5,1} = 36 \oplus 40 = 12$
X_{12}	$C_{3,0} = 60 \oplus 51 = 15$	$C_{4,0} = 53 \oplus 51 = 06$	$C_{5,2} = 22 \oplus 51 = 37$
X_{13}	$C_{3,1} = 59 \oplus 28 = 39$	$C_{4,1} = 31 \oplus 28 = 03$	$C_{5,3} = 21 \oplus 28 = 09$
X_{14}	$C_{3,2} = 34 \oplus 21 = 55$	$C_{4,2} = 58 \oplus 21 = 47$	$C_{5,4} = 57 \oplus 21 = 44$
X_{15}	$C_{3,0} = 15 \oplus 12 = 03$	$C_{4,3} = 53 \oplus 12 = 57$	$C_{5,0} = 19 \oplus 12 = 31$
X_{16}	$C_{3,1} = 39 \oplus 31 = 56$	$C_{4,0} = 06 \oplus 31 = 25$	$C_{5,1} = 12 \oplus 31 = 19$
X_{17}	$C_{3,2} = 55 \oplus 60 = 11$	$C_{4,1} = 03 \oplus 60 = 63$	$C_{5,2} = 37 \oplus 60 = 25$
X_{18}	$C_{3,0} = 03 \oplus 39 = 36$	$C_{4,2} = 47 \oplus 39 = 08$	$C_{5,3} = 09 \oplus 39 = 46$
X_{19}	$C_{3,1} = 56 \oplus 37 = 29$	$C_{4,3} = 57 \oplus 37 = 28$	$C_{5,4} = 44 \oplus 37 = 09$

FIGURE 2.5 – Calculs effectués par l'émetteur pour calculer les paquets codes.

A la fin de cette étape, on va pouvoir transmettre les paquets codes calculés, qui valent : $C_{3,0} = 36$, $C_{3,1} = 29$, $C_{3,2} = 11$, $C_{4,0} = 25$, $C_{4,1} = 63$, $C_{4,2} = 8$, $C_{4,3} = 28$, $C_{5,0} = 31$, $C_{5,1} = 19$, $C_{5,2} = 25$, $C_{5,3} = 46$ et $C_{5,4} = 9$. On peut calculer le paquet code $C_{1,0} = C_{3,0} \oplus C_{3,1} \oplus C_{3,2} = 36 \oplus 29 \oplus 11 = 50$. On pourra d'ailleurs vérifier que $C_{1,0} = C_{4,0} \oplus C_{4,1} \oplus C_{4,2} \oplus C_{4,3} = 25 \oplus 63 \oplus 8 \oplus 28 = 50$, que $C_{1,0} = C_{5,0} \oplus C_{5,1} \oplus C_{5,2} \oplus C_{5,3} \oplus C_{5,4} = 31 \oplus 19 \oplus 25 \oplus 46 \oplus 9 = 50$, et que $C_{1,0} = X_0 \oplus X_1 \oplus \cdots \oplus X_{19} = 12 \oplus 1 \oplus \cdots \oplus 37 = 50$.

On peut transmettre le paquet $C_{1,0}$: l'encodage et la transmission de ce bloc sont terminés.

Plaçons-nous maintenant du point de vue du récepteur. Supposons que le récepteur reçoive les paquets dans cet ordre :
X_0,X_1,X_7,X_2,X_5,X_{11},X_6,X_{10},X_8,X_{12},X_{14},X_{15},X_{17},$C_{4,2}$,X_{18},$C_{3,2}$,X_{19},$C_{3,1}$,$C_{4,0}$,$C_{5,2}$,$C_{4,1}$,$C_{4,3}$,$C_{5,3}$,$C_{5,1}$ et $C_{1,0}$. Il lui manque les paquets X_3,X_4,X_9,X_{13},X_{16},$C_{3,0}$,$C_{5,0}$,$C_{5,3}$ et $C_{1,0}$. Le récepteur va effectuer les calculs dans l'ordre présenté dans la figure ci-dessous (2.7)

	R [3][0]	R [3][1]	R [3][2]	R [4][0]	R [4][1]	R [4][2]	R [4][3]	R [5][0]	R [5][1]	R [5][2]	R [5][3]	R [5][4]	RR [3]	RR [4]	RR [5]
$\emptyset$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
X_0=12	12			12				12							
X_1=1		1			1				1						
X_7=2		3					2			2					
X_2=20			20			20				22					
X_5=20			0		21			24							
X_{11}=40			40				42		41						
X_6=37	41					49			12						
X_{10}=11		8				58		19							
X_8=10			34	6							10				
X_{12}=51	26			53						37					
X_{14}=21			55			47						21			
X_{15}=12	22						38	31							
X_{17}=60			11		41					25					
$C_{4,2}$=8						39								8	
X_{18}=39	49					0					45				
$C_{3,2}$=11			0										11		
X_{19}=37		45					3					48			
$C_{3,1}$=29		48											22		
$C_{4,0}$=25				44										17	
$C_{5,2}$=25										0					25
$C_{4,1}$=63					22									46	
$C_{4,3}$=28							31							50	
$C_{5,3}$=46											3				55
$C_{5,1}$=19									31						36
#	5	5	7	4	4	6	5	4	4	5	3	2			
##													2	4	3
OBJECTIF	8	8	7	6	6	6	6	5	5	5	5	5	4	5	6
OK ?	NO	NO	END	NO	NO	END	YES	YES	YES	END	NO	NO	NO	YES	NO

FIGURE 2.6 – Calculs effectués par le récepteur à la réception des paquets

A la fin de la réception du bloc, le récepteur a effectué les calculs nécessaires, et dispose des tableaux R, RR, # et ## qu'il utilisera ensuite pour le décodage. On peut d'ores-et-déjà faire plusieurs remarques : la valeur de #[3][2] vaut sa valeur $OBJECTIF$, et par conséquent on a bien $RR[3][2] = 0$. C'est aussi le cas pour #[4][2] et #[5][2]. On remarque de plus que #[4][3] est égal à sa valeur $OBJECTIF$ moins 1 donc $R[4][3]$ contient la valeur d'un paquet effacé (en l'occurrence le paquet X_3, dont la valeur est effectivement 31). C'est aussi le cas pour ##[4], et $RR[4] = 50$ correspond à la valeur du paquet $C_{1,0}$, qui est bien 50, [5][0] et [5][1].

Le récepteur peut maintenant lancer l'algorithme de décodage itératif.

R [3][0]	R [3][1]	R [3][2]	R [4][0]	R [4][1]	R [4][2]	R [4][3]	R [5][0]	R [5][1]	R [5][2]	R [5][3]	R [5][4]	RR [3]	RR [4]	RR [5]
49	48	0	44	22	0	31	31	31	0	3	48	22	50	36
X_3						X_3				X_3				
	X_4		X_4								X_4			
X_9				X_9							X_9			
	X_{13}			X_{13}						X_{13}				
	X_{16}		X_{16}					X_{16}						
$C_{3,0}$												$C_{3,0}$		
							$C_{5,0}$							$C_{5,0}$
										$C_{5,3}$				$C_{5,3}$
												$C_{1,0}$	$C_{1,0}$	$C_{1,0}$
	X_4		X_4								X_4			
X_9				X_9							X_9			
	X_{13}			X_{13}						X_{13}				
$C_{3,0}$												$C_{3,0}$		
										$C_{5,3}$				$C_{5,3}$
X_9				X_9							X_9			
	X_{13}			X_{13}						X_{13}				

FIGURE 2.7 – Calculs effectués par le récepteur lors du décodage

Afin de simplifier la lecture, nous ne répétons pas les tableaux # et ## à chaque itération, mais ils sont bien modifiés au fur et à mesure de la réparation de paquets (ainsi que les tableaux R et RR). A la première itération, le récepteur peut réparer X_3 avec $R[4][3]$, X_{16} avec $R[5][1]$, $C_{5,0}$ avec $R[5][0]$ et $C_{1,0}$ avec $RR[4]$. A la deuxième itération, on répare X_4 avec $R[4][0]$, $C_{3,0}$ avec $RR[3]$ et $C_{5,3}$ avec $RR[5]$. Enfin à la dernière itération on répare X_9 avec $R[3][0]$ ou $R[5][4]$ et X_{13} avec $R[3][1]$ ou $R[5][3]$.

Le récepteur est ici parvenu à réparer 9 paquets (5 paquets sources et 4 paquets codes), en ayant eu recours à 13 paquets de redondance. Rappelons que notre code correcteur n'est pas un code MDS. S'il s'agissait d'un code MDS (pour Maximum Distance Separable), nous saurions à coup sûr que n'importe quel ensemble de $n - k - 1$ paquets effacés est réparable, ce qui n'est pas le cas pour notre code. En effet, il existe des ensembles de paquets non-réparables plus petits que l'ensemble des paquets de redondance qu'on ajoute au bloc transmis. C'est la une limitation de notre code correcteur, que l'on rencontre avec les codes non-MDS.

Présentons ici un exemple de petit ensemble de paquets non-réparable. Prenons, comme dans l'exemple précédent, $P = [3, 4, 5]$ et $k = 20$, mais supposons maintenant que les 5 paquets X_0, $C_{3,0}$, $C_{4,0}$, $C_{5,0}$ et $C_{1,0}$ ont été effacés. En lançant l'algorithme de décodage itératif, le récepteur se retrouve dès la première itération face à une situation de blocage (stopping-set). Dans cet exemple, nous ne donnons pas de valeur aux paquets effacés, le problème de blocage du décodeur itératif ne dépendant pas des valeurs des paquets, mais uniquement de leurs indices.

R [3][0]	R [3][1]	R [3][2]	R [4][0]	R [4][1]	R [4][2]	R [4][3]	R [5][0]	R [5][1]	R [5][2]	R [5][3]	R [5][4]	RR [3]	RR [4]	RR [5]
X_0 $C_{3,0}$			X_0 $C_{4,0}$				X_0 $C_{5,0}$					$C_{3,0}$ $C_{1,0}$	$C_{4,0}$ $C_{1,0}$	$C_{5,0}$ $C_{1,0}$

FIGURE 2.8 – Situation de blocage du décodage itératif (stopping-set) pour 5 paquets effacés

La figure (2.8) présente donc un stopping-set de taille 5 mettant le décodeur itératif dans un état de blocage : aucun des 5 paquets effacés ne peut être réparé, car pour chaque nœud contrainte susceptible de permettre sa réparation, il existe au moins deux paquets effacés, et on ne peut résoudre aucune équation du système associé à notre code et induit par nos effacements.

Rappelons quelques notions que nous avons présentées dans le premier chapitre de ce manuscrit :

- un code correcteur est muni d'une distance minimale δ. Cette valeur nous indique le nombre minimal de paquets (ou symboles) que le récepteur est sûr de pouvoir réparer, quelle que soit leur position dans le bloc (ou mot) reçu. Ainsi, tant qu'on a subi δ effacements ou moins, le décodage aboutit à sur un succès.
- le nombre de paquets de redondance qu'on ajoute à un bloc $n-k$ est le nombre maximum de paquets effacés qu'on va pouvoir réparer. En effet, supposons qu'un bloc encodé ait subi e effacements, avec $e > n-k$: alors le récepteur n'a reçu que $n-e < k$ paquets, et il doit alors résoudre un système dans lequel il y a plus d'inconnues que d'équations, ce qui est impossible.
- enfin, si le bloc a subi e effacements, avec $\delta < e \leq n-k$, il y a alors une probabilité de décoder les e paquets effacés, qui diminue à mesure que e augmente et s'approche de $n-k$. On se trouve ici dans la zone de Waterfall décrite en figure (1.9).

La probabilité d'échec du décodage itératif en fonction de e (le nombre d'effacements sur un bloc code) progresse alors de façon analogue à la courbe présentée dans la figure ci-dessous :

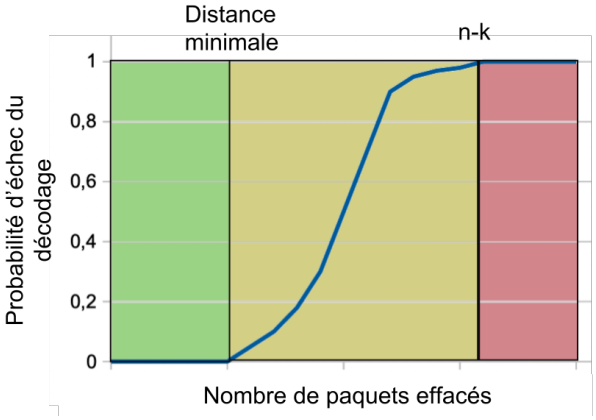


FIGURE 2.9 – La probabilité d'échec du décodage d'un bloc progresse en fonction du nombre d'effacements subis par ce bloc comme une courbe en S.

2.3 Analyse théorique

Nous proposons dans cette section d'étudier les algorithmes d'encodage et de décodage présentés précédemment, afin d'en déduire leur complexité en temps et leur consommation mémoire : nous rappellerons la notion de matrice génératrice du code, qui nous donne directement le nombre de Xors à effectuer pour calculer nos paquets codes, et étudierons leurs pseudo-codes, qui contiennent quelques astuces permettant de réduire cette quantité.

2.3.1 Complexité algorithmique de l'encodage

Ce code correcteur est un code linéaire utilisable sur un flux de paquets, et sa complexité doit donc être proportionnelle à la quantité d'information à encoder. Nous montrons dans cette partie que l'algorithme d'encodage à une complexité et consommation mémoire de l'ordre de $\mathcal{O}(k * len)$ pour P fixé.

Complexité temps de l'encodage

Comme on l'a vu dans le premier chapitre, la phase d'encodage d'un code linéaire (tel que les codes LDPC par exemple) consiste à combiner, notamment via l'utilisation de l'opérateur $\oplus$ XOR des symboles sources, pour obtenir des symboles de redondance. Les combinaisons linéaires à effectuer sont données par la matrice creuse génératrice G du code. Dans le cas des codes systématiques (dont notre code fait partie), la matrice G d'un $[n, k] - code$ contient la sous-matrice identité I_k, qui correspond aux k paquets sources que l'on transmet dans le bloc code, et les $n - k$ autres lignes correspondent aux paquets de redondance qu'on a effectivement calculés par combinaisons linéaires.

Les k lignes de la sous-matrice identité I_k ne correspondent donc pas à des calculs, mais uniquement à l'envoi des k paquets sources du bloc, alors que les $n - k$ autres lignes correspondent à un certain nombre de calculs effectués pour calculer les $n - k$ paquets codes (rappelons qu'on appelle cette sous-matrice la matrice H du code). Dans le cas d'un code LDPC irrégulier dont notre code fait partie, chacune de ces lignes peut contenir un nombre quelconque de 1 : ce nombre de 1 correspond au nombre de paquets sources à Xorer pour obtenir le paquet code correspondant à cette ligne.

La matrice génératrice G de notre code est obtenue grâce à l'ensemble de nombre premiers entre eux P sur lequel l'émetteur et le récepteur doivent s'être accordés au préalable, ainsi que par la dimension du code k. Le système d'équations associé à cette matrice est donné en figure (2.1), et représenté dans la figure (2.4). Le nombre total de 1 dans la matrice H associée à notre code est égal à $((|P| + 1) * k)$: chaque paquet X_i parmi les k paquets sources se retrouve Xoré dans un paquet code pour chaque élément P_j de P, ainsi que dans le paquet code $C_{1,0}$. Pour chaque paquet source, on doit effectuer $|P| + 1$ Xor, et chaque Xor d'un paquet source correspond en réalité à len Xors binaires.

Ainsi, le nombre de Xors binaires de l'algorithme de l'encodage est proportionnel à $\mathcal{O}(k * (|P| + 1) * len)$. Cependant, en observant le pseudo-code de l'encodage, on remarque que le calcul du paquet $C_{1,0}$ ne requiert pas de Xorer les k paquets sources, mais peut se faire en ne Xorant que les P_0 (ou P_1, P_2, ...) paquets codes qu'on aura au préalable calculés (ce qui correspond aux lignes 14 et 15 du pseudo-code de l'encodage). Cette astuce nous permet donc de réduire la complexité de l'encodage : le temps de calcul des $k = \Sigma(P)$ paquets codes $C_{P_i,j}$

(autres que $C_{1,0}$) devient alors proportionnelle à $\mathcal{O}(k * |P| * len)$, et le temps de calcul de $C_{1,0}$ est proportionnel à $\mathcal{O}(P_0 * len)$.

Nous devons ajouter aux calculs des paquets codes la phase d'initialisation de ces paquets, qui revient à mettre $\Sigma(P) + 1$ paquets de len bits à 0, et la complexité de l'initialisation est alors proportionnelle à $\mathcal{O}((\Sigma(P) + 1) * len)$.

L'algorithme d'encodage a donc une complexité en :

$$\mathcal{O}((|P| * k + P_0 + \Sigma(P) + 1) * len) \tag{2.1}$$

Consommation mémoire de l'encodage

La consommation mémoire de l'encodage de notre code correspond à la quantité de mémoire nécessaire au calcul de nos paquets de redondance avant leur émission : elle est proportionnelle à la quantité d'information redondante ajoutée à un bloc encodé. En effet, comme on l'a vu précédemment, le code que nous présentons est utilisable en streaming : les paquets sources n'ont pas à être conservés par l'émetteur, et peuvent être transmis dès que les calculs les concernant ont été effectués. Nous considérons en effet que l'information source à encoder est nécessairement stockée ou reçue par l'émetteur, d'une manière ou d'une autre, et qu'elle n'est donc pas consommée par notre algorithme, mais ne fait que passer dans celui-ci. Ajoutons qu'ici, nous négligeons la quantité de mémoire nécessaire aux différentes variables utilisées par l'encodage : certaines implémentations pourront réutiliser des variables de boucle, et quelques améliorations peuvent être apportées pour réduire cette quantité d'information au strict minimum. Néanmoins, cette quantité d'information nécessaire au déroulement de cet algorithme est négligeable par rapport à la taille des paquets sources d'un bloc. Nous négligeons de plus la quantité mémoire nécessaire aux headers de nos paquets : les headers des paquets (sources et codes) sont en général beaucoup plus petits que leurs buffer, et des implémentations pourront se contenter de réutiliser un seul header, qu'on modifie avant l'envoi d'un paquet.

Ainsi, seuls les paquets codes doivent être conservés en mémoire lors de l'encodage. Cette quantité de mémoire correspond donc aux $\Sigma(P) + 1$ paquets codes d'un bloc, chaque paquet représentant len bits. La consommation mémoire de l'algorithme d'encodage est donc

$$(\Sigma(P) + 1) * len \tag{2.2}$$

.

Cette consommation mémoire relativement faible est à comparer avec la consommation mémoire des codes sans rendement que nous avons présentés dans le premier chapitre, du type Luby-Transform par exemple. Ces codes consistent à générer un très grand nombre de paquets codes, en Xorant un ensemble de paquets sources à l'émission de chaque paquet code. Par conséquent, pour chaque paquet code émis, il faut disposer de l'ensemble des paquets sources, et la consommation mémoire est alors plus importante. Alors que notre code ne consomme qu'une quantité proportionnelle aux $n - k$ paquets de redondance d'un bloc, les codes sans rendement nécessitent une quantité d'information proportionnelle aux k paquets sources d'un bloc.

2.3.2 Complexité algorithmique du décodage

Tout comme pour l'algorithme d'encodage, la complexité du décodage doit elle aussi être proportionnelle à la dimension du bloc pour permettre à notre code de fonctionner sur un flux,

ou sur de très grande quantité d'information (ie : beaucoup de paquets, et potentiellement de grande taille). Dans cette partie, nous montrons que l'algorithme de décodage nécessite légèrement plus de calculs que l'algorithme d'encodage, mais que ceux-ci restent proportionnels à la quantité d'information encodée, pour P fixé.

Complexité temps du décodage

On a vu que l'algorithme de décodage est divisé en deux sous-algorithmes : certains calculs sont effectués durant la réception du bloc, et les calculs correspondant à l'algorithme de décodage sont effectués après la réception de ce bloc. Nous proposons donc ici d'étudier la complexité de ces deux sous-algorithmes indépendamment l'un de l'autre.

Tout comme pour l'émetteur, le récepteur doit initialiser des paquets sur lesquels il va effectuer ses calculs (ainsi que des compteurs). Cette phase d'initialisation, correspondant aux lignes 2 à 8 du pseudo-code, nécessite de mettre $(|P|+\Sigma(P)*len)$ bits à 0 (les nœuds contraintes des tableaux R et RR), ainsi que $(|P| + \Sigma(P))$ nombres entiers (les compteurs des tableaux $\#$ et $\#\#$). Cette phase d'initialisation a donc une complexité de $((|P| + \Sigma(P) * (len + \epsilon))$, ϵ correspondant à la taille en mémoire de nos compteurs (qui peut varier suivant l'implémentation adoptée, mais reste globalement négligeable par rapport à la taille des paquets).

Une fois l'initialisation des tableaux effectuée, les calculs réalisés durant la réception du bloc correspondent aux Xors que le récepteur effectue pour calculer les nœuds contraintes, qui lui serviront à décoder des paquets effacés. Suivant que le paquet reçu est un paquet source X_i, un paquet code C_{P_i}, j ou le paquet code $C_{1,0}$, les calculs à effectuer ne sont pas les mêmes :

- si le paquet reçu est un paquet source X_i, le récepteur Xore ce paquet dans $|P|$ paquets de redondance, les $|P|$ cases du tableau R correspondant à X_i. Ainsi, pour chaque paquet source, on effectue $|P| * len$ Xors binaires. Ces calculs correspondent à la ligne 10 du pseudo-code.
- si le paquet reçu est un paquet code $C_{P_i,j}$ autre que $C_{1,0}$, le récepteur Xore ce paquet dans les deux paquets de redondance lui correspondant : la case adéquate du tableau R et la case adéquate du tableau RR. Ces calculs correspondent respectivement aux lignes 20 et 22 du pseudo-code.
- si le paquet reçu est le paquet code $C_{1,0}$, le récepteur Xore ce paquet dans les $|P|$ nœuds contraintes lui correspondant (les $|P|$ cases de RR).

Chaque paquet Xoré représentant len bits, le récepteur effectuera :

- $(|P| * (k + 1) + \Sigma(P) * 2) * len$ Xors binaires si le bloc n'a subi aucun effacement, puis il n'aura pas à lancer l'algorithme de décodage
- moins de $(|P| * (k + 1) + \Sigma(P) * 2) * len$ si le bloc a subi des effacements, mais il devra ensuite lancer l'algorithme de décodage.

Une fois le bloc reçu, si celui-ci a subi des effacements, le récepteur peut alors lancer l'algorithme de décodage. Supposons que le récepteur aie à corriger e effacements. Dans ce cas, l'algorithme de décodage itératif va traiter chaque paquet réparable comme si celui-ci était un paquet effectivement reçu :

- $|P| * len$ Xors binaires si il s'agit d'un paquet source X_i,
- $2 * len$ Xors binaires si il s'agit d'un paquet code $C_{P_i,j}$ autre que le paquet $C_{1,0}$,
- $|P| * len$ Xors binaires si il s'agit du paquet $C_{1,0}$.

A la fin de l'algorithme de décodage, si celui-ci aboutit à un succès et que le récepteur a décodé les e paquets effacés, il aura donc effectué $(e * max(|P|, 2) * len)$ Xors binaires au maximum. Si le récepteur n'est pas parvenu à décoder tous les paquets effacés, il effectuera moins de calculs.

On sait que pour que le décodage aboutisse à un succès, il faut que $e <= n-k$, par conséquent, l'algorithme de décodage aura au maximum une complexité $((n-k) * max(|P|, 2) * len)$.

On remarque donc qu'il existe deux cas-possibles pour la complexité de l'algorithme de décodage :

— Si le bloc est convenablement décodé, alors le récepteur a Xoré tous les paquets (reçus ou réparés), et a donc effectué $(|P| * (k+1) + \Sigma(P) * 2) * len$ XOR binaires,

— Si le bloc n'est pas convenablement décodé (il reste des paquets effacés non-réparés), alors le récepteur a effectué moins de $(|P| * (k+1) + \Sigma(P) * 2) * len$ XOR binaires.

La complexité de l'algorithme de décodage est donc de $(|P| * (k+1) + \Sigma(P) * 2) * len$ au maximum, et est atteinte quand le bloc est entièrement reçu ou décodé.

Précisons que pour l'algorithme de décodage, nous négligeons les calculs de la valeur $Objectif$ à comparer avec les compteurs de # et ## (lignes 7, 9, 26 et 28 du pseudo-code), la comparaison de cette valeur avec les compteurs (lignes 10, 19, 29 et 36), et la suppression d'un paquet des listes des paquets effacés (lignes 12, 21, 31 et 38). Les calculs de la valeur $Objectif$ sont simples, et peuvent être effectués en avance, et le retrait d'un élément d'une liste (liste chaînée par exemple) est peu coûteuse.

L'algorithme de décodage, en considérant les calculs effectués durant la réception, puis ceux effectués durant le décodage, a donc une complexité en

$$\mathcal{O}((|P| * (k+1) + \Sigma(P) * 2) * len) \tag{2.3}$$

et l'algorithme de décodage, en ne tenant pas compte des calculs effectués pendant la réception, aura une complexité au maximum de

$$\mathcal{O}((n-k) * max(|P|, 2) * len) \tag{2.4}$$

Consommation mémoire du décodage

Tout comme pour l'encodage, la consommation mémoire de l'algorithme de décodage correspond aux paquets dans lesquels le récepteur va Xorer successivement les paquets lors de leur réception ou de leur réparation (les nœuds contraintes des tableaux R et RR et les compteurs associés # et ##). La consommation mémoire est alors de $(\Sigma(P) + |P|) * (len + \epsilon)$ bits. Nous choisissons ici de négliger la quantité d'information utilisée pour les différentes variables nécessaires à l'algorithme. En effet, les listes $X_{EFFACES}$, $C_{EFFACES}$ peuvent par exemple être des listes chaînées ne contenant que les headers des paquets effacés, et leur taille peut donc être négligée en comparaison avec les tableaux R et RR.
La consommation mémoire de l'algorithme de décodage est donc de

$$(\Sigma(P) + |P|) * len) \tag{2.5}$$

Tout comme pour l'émission, nous ne considérons pas ici l'information source : les paquets sources (ainsi que les paquets codes) n'ont pas besoin d'être conservés pour le déroulement de l'algorithme de décodage, et peuvent être utilisés par le récepteur une fois que les calculs ont été effectués.

2.3.3 Complexité linéaire

Comme on vient de le voir, les algorithmes d'encodage et de décodage ont respectivement une complexité algorithmique de $((|P|*k+\Sigma(P)+P_0)*len)$ et $((|P|*(k+1)+\Sigma(P)*2)*len)$, et

ont une consommation mémoire de $((\Sigma(P)+1)*len)$ et de $((\Sigma(P)+|P|)*(len+\epsilon))$. En fixant la valeur de P, qui est un paramètre du code correcteur, et en notant $d = |P|$, $s = \Sigma(P)$ et $P_{min} = min(P)$, on obtient donc des complexités de $((d+s)*k*len)$ et $((s+P_{min})*len+d*k*len)$, et des consommations mémoires de $((s+1)*len)$ et $((s+d)*len)$. Nos algorithmes ont donc bien une complexité linéaire en $(k*len)$ pour P fixé : les complexités de l'encodage et du décodage progresseront principalement avec le cardinal de P, et leur consommations mémoires progresseront principalement avec la somme de P.

2.3.4 Analyse de la capacité de correction

Cette partie est consacrée à l'analyse théorique de la capacité de correction de notre code correcteur. Comme vu précédemment, on peut analyser l'efficacité d'un code correcteur en cherchant à calculer la distance minimale de ce code, notée δ, qui correspond au nombre maximal d'effacements qu'un code peut corriger à coup-sûr (moins 1). Ainsi, pour un code correcteur de distance minimale δ, il existera des combinaisons de δ paquets ou plus qui mettront l'algorithme de décodage en échec. Un code correcteur avec une grande distance minimale aura ainsi une capacité de correction à coup-sûr meilleure qu'un code correcteur ayant une petite distance minimale, et on cherchera donc à construire des codes correcteurs avec la plus grande distance minimale possible. Les codes correcteurs peuvent être rangés dans deux grandes catégories : les codes MDS, qui ont une distance minimale optimale (égale au nombre de paquets de redondance moins 1), et les codes non-MDS, dont le décodage peut échouer même quand on a reçu plus de k paquets.

Rappelons rapidement la notion de code correcteur MDS, sachant que le code que nous présentons n'en fait pas partie. Un code MDS, pour Maximum Distance Separable, est un code de dimension k et de longueur n dont la distance minimale vaut $\delta = n-k-1$. Ces codes permettent donc de décoder tout sous-ensemble d'effacements de taille $\delta = n-k-1$ ou inférieur, et ceci représente une borne indépassable. Un code MDS systématique de dimension k et de longueur n permet de corriger $n-k-1$ effacements à coup-sûr, grâce à l'ajout de $n-k$ paquets, à l'image des codes de Reed-Solomon présentés précédemment.

A contrario, un code non-MDS de dimension k et de longueur n a une distance minimale $\delta < n-k-1$, et il existera des combinaisons de δ paquets qui mettront en échec l'algorithme de décodage. Un code non-MDS tel que le notre peut aboutir à un échec du décodage même en dans le cas ou on aura subi e moins que $n-k$ effacements : on dit alors que ces e effacements forment un Stopping-set dans notre graphe de Tanner. L'étude des Stopping-set n'a de sens que dans le cas des codes non-MDS : un code MDS sera décodé si il a subi moins de $\delta = n-k-1$ effacements, et échouera quand il aura subi $\delta = n-k-1$ effacements ou plus, la notion de Stopping-set pour un code non-MDS n'est donc pas pertinente.
Un Stopping-Set de taille e est un ensemble de e paquets qui met en échec le décodage : c'est une situation qu'on souhaite éviter, puisque même en transmettant $(n-k)$ paquets supplémentaires on se retrouve face à un échec du décodage, en ayant subi moins que $(n-k)$ effacements. Pour un code non-MDS de distance minimale δ, on sait donc qu'il existe par définition des Stopping-set de taille supérieure ou égale à δ et inférieure à $n-k-1$. De nos jours, l'étude du nombre de Stopping-set pour de nombreux codes et leur répartition est un domaine de recherche important, ainsi que la construction de codes correcteurs dont la complexité est faible, et la distance minimale élevée.

Pour les codes correcteurs comme pour de nombreux domaines de l'informatique, il existe

une relation entre l'efficacité d'un algorithme et sa complexité. Dans le cas des codes correcteurs, plus un code a une bonne efficacité (il s'approche d'un code MDS), plus sa complexité est importante. Ainsi, les codes MDS ont une distance minimale optimale, mais une complexité d'encodage et de décodage importante, et les codes non-MDS une capacité de correction moindre, mais une complexité d'encodage et de décodage meilleure. Notre code correcteur fait partie de la deuxième catégorie : nous avons montré précédemment que notre code correcteur a une complexité en encodage et en décodage linéaire. Nous allons maintenant étudier sa capacité de correction, en étudiant sa distance minimale et la répartition de ses Stopping-set, qui évoluent suivant son paramétrage.

Distance minimale

Dans cette partie, nous proposons d'étudier la distance minimale δ de notre code. Celle-ci est relativement faible, ce qui est un des désavantages de notre code correcteur. Rappelons que le code que nous présentons dépend d'un ensemble de nombres premiers entre eux, P sur lequel l'émetteur et le récepteur doivent s'être accordés, et dont on déduit la dimension maximale k_{max} du code, et sa matrice génératrice. Les codes que cet ensemble permet de créer ont une dimension $k \leq PPCM(P)$, avec $PPCM(P)$ le plus petit commun multiple de l'ensemble P. Le nombre de paquets de redondance par bloc $n-k$ est fixé, et on a $n-k = \Sigma(P)+1$.

Ainsi, $P = [3,4,5,11]$ nous permet de produire des codes de dimension $k \leq 660$ et de longueur $n = k + (n-k) = k + \Sigma(P) + 1 = k + 24$.

Afin d'étudier cette distance minimale, il faut que nous distinguions deux cas, qui correspondent à des paramétrages différents de notre code.

— Le cas ou on a choisi comme dimension du code $k = k_{max} = PPCM(P)$. Il correspond aux codes de dimension maximale qu'on peut obtenir pour un ensemble P. Dans ce cas, on aura toujours $\delta = 4$, une distance minimale constante et faible : en utilisant ce paramétrage, il existera des stopping-sets de taille 4, que nous présentons par la suite. En prenant $P = [3,4,5,11]$ et $k = 660$, il existera des combinaisons de 4 paquets qui mettront en échec le décodage.

— Le cas ou on a choisi comme dimension du code $k \leq k_{max}/(P_i)$, qui correspond à un code pour lequel on encode des blocs de paquets sources plus petits que k_{max}. Dans ce cas, la distance minimale augmente et peut dépasser 4. Pour $P = [3,4,5,11]$ et $k = 4*5*11 = 220$ par exemple, on atteindra une distance minimale $\delta = 5$

.

Avant de rentrer dans le détail des démonstrations, présentons quelques exemples d'ensembles de Stopping-sets de taille $e = 4$ qui mettent en échec notre algorithme de décodage pour différents ensembles P, afin de se familiariser avec la notion de distance minimale et de stopping-sets :

— Pour $P = [2,3]$ et $k = 6$, $X_{effaces} = [X_0, X_1, X_2, X_5]$ est un stopping-set.

— Pour $P = [100,101]$ et $k = 10100$, $X_{effaces} = [X_0, X_2, 6, X_{11}, X_{103}]$ est un stopping-set.

— Pour $P = [2,3,5]$ et $k = 30$, $X_{effaces} = [X_0, X_6, X_{10}, X_{16}]$ est un stopping-set.

— Pour $P = [2,3,5,7]$ et $k = 210$, $X_{effaces} = [X_0, X_{14}, X_{105}, X_{119}]$ est un stopping-set.

— Pour $P = [2,3,5,7,11]$ et $k = 2310$ $X_{effaces} = [X_0, X_1, X_{210}, X_{2101}]$ est un stopping-set.

— Pour $P = [2,3,5,7,11,13]$ et $k = 30030$, $X_{effaces} = [X_0, X_1, X_{6930}, X_{23101}]$ est un stopping-set.

Quel que soit P, on trouvera toujours des Stopping-sets de taille 4 quand on aura $k = PPCM(P)$.

Pour $P = [2, 3]$ et $k = 6$, il y a 9 stopping-sets de taille 4, qui sont :

$$[X_0, X_1, X_2, X_5]$$
$$[X_0, X_1, X_3, X_4]$$
$$[X_1, X_2, X_4, X_5]$$
$$[X_0, C_{2,0}, C_{3,0}, C_{1,0}]$$
$$[X_1, C_{2,1}, C_{3,1}, C_{1,0}]$$
$$[X_2, C_{2,0}, C_{3,2}, C_{1,0}]$$
$$[X_3, C_{2,1}, C_{3,0}, C_{1,0}]$$
$$[X_4, C_{2,0}, C_{3,1}, C_{1,0}]$$
$$[X_5, C_{2,1}, C_{3,2}, C_{1,0}]$$

Pour $P = [3, 5]$ et $k = 15$, les 77 stopping-sets de taille 4 sont :

$[X_0, X_1, X_6, X_{10}]$	$[X_0, X_2, X_{11}, X_{12}]$	$[X_0, X_4, X_9, X_{10}]$	$[X_0, X_7, X_{10}, X_{12}]$
$[X_0, X_3, X_5, X_8]$	$[X_0, X_5, X_6, X_{11}]$	$[X_0, X_3, X_{10}, X_{13}]$	$[X_0, X_5, X_9, X_{14}]$
$[X_6, X_2, X_7, X_{11}]$	$[X_6, X_4, X_1, X_9]$	$[X_6, X_7, X_1, X_{12}]$	$[X_6, X_8, X_3, X_{11}]$
$[X_6, X_{13}, X_1, X_3]$	$[X_6, X_{14}, X_9, X_{11}]$	$[X_{12}, X_4, X_7, X_9]$	$[X_{12}, X_8, X_2, X_3]$
$[X_{12}, X_{13}, X_3, X_7]$	$[X_{12}, X_{14}, X_7, X_9,]$	$[X_{10}, X_2, X_5, X_7]$	$[X_{10}, X_8, X_5, X_{13}]$
$[X_{10}, X_{11}, X_1, X_5]$	$[X_{10}, X_{14}, X_4, X_5]$	$[X_1, X_2, X_7, X_{11}]$	$[X_1, X_8, X_{11}, X_{13}]$
$[X_1, X_{14}, X_4, X_{11}]$	$[X_3, X_4, X_9, X_{13}]$	$[X_3, X_{14}, X_8, X_9]$	$[X_7, X_8, X_2, X_{13}]$
$[X_7, X_{14}, X_2, X_4]$	$[X_{13}, X_{14}, X_4, X_8]$		

$[X_0, X_3, C_{5,0}, C_{5,3}]$	$[X_0, X_5, C_{3,0}, C_{3,2}]$	$[X_0, X_6, C_{5,0}, C_{5,1}]$	$[X_0, X_9, C_{5,0}, C_{5,4}]$
$[X_0, X_{10}, C_{3,0}, C_{3,1}]$	$[X_0, X_{12}, C_{5,0}, C_{5,2}]$	$[X_6, X_1, C_{3,0}, C_{3,1}]$	$[X_6, X_3, C_{5,1}, C_{5,3}]$
$[X_6, X_9, C_{5,1}, C_{5,4}]$	$[X_6, X_{11}, C_{3,0}, C_{3,2}]$	$[X_6, X_{12}, C_{5,1}, C_{5,3}]$	$[X_{12}, X_2, C_{3,0}, C_{3,2}]$
$[X_{12}, X_3, C_{5,2}, C_{5,4}]$	$[X_{12}, X_7, C_{3,0}, C_{3,1}]$	$[X_{12}, X_9, C_{5,2}, C_{5,4}]$	$[X_{10}, X_1, C_{5,0}, C_{5,1}]$
$[X_{10}, X_4, C_{5,0}, C_{5,4}]$	$[X_{10}, X_5, C_{3,1}, C_{3,2}]$	$[X_{10}, X_7, C_{5,0}, C_{5,2}]$	$[X_{10}, X_{13}, C_{5,0}, C_{5,3}]$
$[X_3, X_8, C_{3,0}, C_{3,2}]$			
$[X_3, X_9, C_{5,3}, C_{5,4}]$	$[X_3, X_{13}, C_{3,0}, C_{3,1}]$	$[X_1, X_4, C_{5,1}, C_{5,4}]$	$[X_1, X_7, C_{5,1}, C_{5,2}]$
$[X_1, X_{11}, C_{3,1}, C_{3,2}]$	$[X_1, X_{13}, C_{5,1}, C_{5,3}]$	$[X_7, X_2, C_{3,1}, C_{3,2}]$	$[X_7, X_4, C_{5,2}, C_{5,4}]$
$[X_7, X_{13}, C_{5,2}, C_{5,3}]$	$[X_{13}, X_4, C_{5,3}, C_{5,4}]$	$[X_{13}, X_8, C_{3,1}, C_{3,2}]$	

$[X_0, C_{3,0}, C_{5,0}, C_{1,0}]$	$[X_1, C_{3,1}, C_{5,1}, C_{1,0}]$	$[X_2, C_{3,2}, C_{5,2}, C_{1,0}]$	$[X_3, C_{3,0}, C_{5,3}, C_{1,0}]$
$[X_4, C_{3,1}, C_{5,4}, C_{1,0}]$	$[X_5, C_{3,2}, C_{5,0}, C_{1,0}]$	$[X_6, C_{3,0}, C_{5,1}, C_{1,0}]$	$[X_7, C_{3,1}, C_{5,2}, C_{1,0}]$
$[X_8, C_{3,2}, C_{5,3}, C_{1,0}]$	$[X_9, C_{3,0}, C_{5,4}, C_{1,0}]$	$[X_{10}, C_{3,1}, C_{5,0}, C_{1,0}]$	$[X_{11}, C_{3,2}, C_{5,1}, C_{1,0}]$
$[X_{12}, C_{3,0}, C_{5,2}, C_{1,0}]$	$[X_{13}, C_{3,1}, C_{5,3}, C_{1,0}]$	$[X_{14}, C_{3,2}, C_{5,4}, C_{1,0}]$	

Enfin, nous présentons une façon de visualiser les stopping-sets de nos codes, dans le cas ou $|P| = 2$: le graphe de Rook. Dans le cas particulier ou $|P = 2|$, par exemple pour $P = [3, 5]$, on peut représenter notre système d'équations par un graphe de Rook de dimensions $(P_0{+}1, P_1{+}1)$.

Un graphe de Rook de dimension (a, b), noté $R_{a,b}$, peut être vu comme un graphe dont les $a * b$ sommets sont les cases d'un échiquier, et les $b(a+b)/2-ab$ arêtes comme les déplacements que peut effectuer une tour sur cet échiquier.

Ici, nous présentons comment sont répartis les paquets d'un bloc dans le graphe de Rook qui lui correspond en fonction de leurs indices, et des stopping-sets de taille 4. Ce moyen de représenter les relations entre les paquets de notre code n'est par contre pas adaptée dans le cas ou $|P| > 2$, et il nous faut alors des représentations en dimension supérieures.

Un stopping-set de taille e mettant en échec notre algorithme de décodage correspond dans un graphe de Rook à la situation dans laquelle e tours disposées dans ce graphe forment un cycle : chaque tour peut atteindre au moins une autre tour sur sa ligne et sur sa colonne.

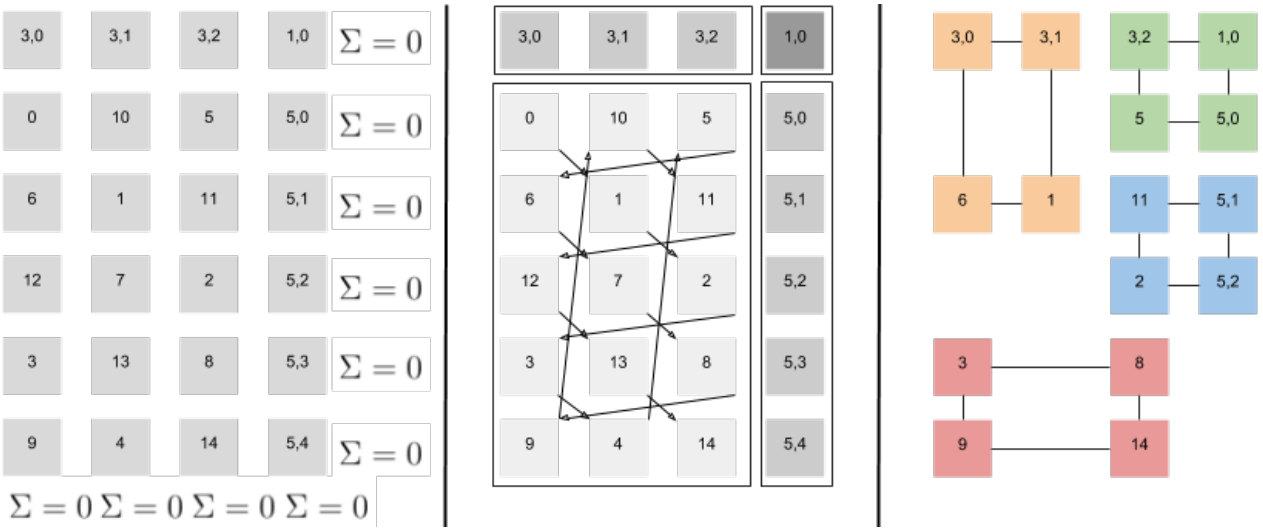


FIGURE 2.10 – Visualisation des stopping-sets de longueur 4 sur un graphe de Rook de dimensions $(4, 6)$, pour $P = [3, 5]$

Cette représentation permet de voir apparaître intuitivement le nombre de stopping-set de longueur 4 dans le cas ou $|P| = 2$: avec cette représentation, on peut décoder un paquet s'il est tout seul sur sa ligne ou sur sa colonne (situation qui correspond à une équation de notre système qu'on peut résoudre, car elle n'a qu'une seule inconnue). Un Stopping-set de taille 4 forme donc un rectangle dans notre graphe de Rook : aucun des 4 paquets n'est seul sur sa ligne ou sur sa colonne. Le nombre de rectangles dans un graphe de Rook de dimensions (a, b) est :

$$\frac{ab(a-1)(b-1)}{4}.$$

On peut le démontrer de la manière suivante :

- On choisit d'abord deux lignes L_1 et L_2, dans lesquelles on placera nos 4 paquets effacés : on a $a(a-1)/2$ manière de choisir L_1 et L_2.
- On choisit ensuite deux colonnes C_1 et C_2, dans lesquelles on placera là aussi nos paquets effacés : on a là aussi $b(b-1)/2$ combinaisons.
- Les intersections des lignes et des colonnes choisies forment un rectangle dont les sommets, correspondant à des paquets effacés, sont alors un Stopping-set de taille 4. Il y a donc $\frac{ab(a-1)(b-1)}{4}$ stopping-sets de taille 4 dans un code généré avec $P = [a-1, b-1]$. Il existe $\binom{4}{a*b}$ sous-graphes de taille 4 dans un graphe de Rook de dimension (a, b), et parmi ces graphes, $\frac{ab(a-1)(b-1)}{4}$ sont un Stopping-set. Par conséquent, la probabilité d'échec du décodage pour 4 effacements et $P = [a-1, b-1]$ est exactement $\frac{\frac{ab(a-1)(b-1)}{4}}{\binom{4}{a*b}}$.

Nous verrons par la suite que dans le cas ou $|P| = 2$ il est possible d'utiliser un graphe de Rook pour calculer le nombre de stopping-sets de taille quelconque e. Nous venons de voir que c'est très simple pour $e = 4$ effacements, mais qu'il s'agit d'un problème combinatoire : quand les dimensions du graphe ou que e est grand, il devient difficile de calculer la valeur exacte du nombre de stopping-set.

2.4 Simulations

Dans cette section nous présentons un certain nombre de simulations ayant pour but de confirmer les analyses théoriques présentées précédemment : les complexités en temps et en mémoire des algorithmes d'encodage et de décodage, et la capacité de correction du code. Elles confirment que la complexité de l'encodage est linéaire en la dimension du code (k), en la taille des paquets (len) et en le nombre d'éléments de l'ensemble P. En effet, rappelons que l'encodage consiste à Xorer chaque paquet de taille source de taille len dans un nombre d de paquets de redondance, d correspondant au cardinal de l'ensemble P.

Ces simulations ont été réalisées suivant le modèle de Monte-Carlo, tel que présenté dans le premier chapitre. Nous réalisons un grand nombre de simulations qui permettent de juger du temps d'exécution de ces algorithmes en fonction des paramètres P, k, len, ainsi qu'en encodant et en envoyant des fichiers de différentes tailles, afin d'avoir à encoder plusieurs blocs successivement.

Elles ont été réalisées sur un ordinateur portable : il s'agit d'un MacBook Pro (Retina 13 pouces) datant de début 2015, ayant un processeur Intel Core i7 cadencé à 3.1GHz, et 16 Go de mémoire vive DDR3 cadencés à 1867MHz.

2.4.1 Simulations de l'encodage

Nous présentons ici les simulations concernant l'encodage uniquement : dans un premier temps, nous réalisons des simulations sur l'encodage et la transmission de fichiers dont la taille varie de $1 megaoctet$ à $5 Gigaoctets$, afin d'étudier le comportement de notre code par bloc. Dans un second temps, nous nous intéresserons à l'encodage d'un seul bloc, en faisant varier les différents paramètres de notre code.

Complexité de l'encodage : taille de fichiers et nombre de blocs

Nous présentons les simulations de l'encodage en conditions réelles d'utilisation de notre code par bloc : nous encodons et transmettons bloc par bloc des fichiers de différentes tailles, en faisant varier le cardinal de P et en fixant la taille des blocs k et des paquets len. Ces simulations permettent de confirmer que le code correcteur est bien un code par bloc linéaire, et que par conséquent, encoder b blocs prend b fois plus de temps qu'encoder 1 bloc. Les tailles des fichiers encodés et transmis vont du mégaoctet à plusieurs gigaoctets. De plus, elles confirment que le temps de calcul de l'encodage dépend bien du cardinal de P.

Ces simulations ont été réalisées de deux manières :

— en envoyant les paquets (sources et de redondance), afin de juger du temps total d'exécution de l'émission, et comparer le temps requis par l'algorithme d'encodage et le reste du procédé d'émission (voir figure 2.12).

— sans envoyer les paquets (sources et de redondance), ce qui nous permet de juger uniquement le temps requis par l'algorithme d'encodage, indépendamment du reste du procédé d'émission (voir figure 2.11).

Ceci nous permet de comparer le temps d'exécution de l'algorithme d'encodage par rapport au temps total nécessaire à l'encodage puis la transmission d'un bloc, notamment dans la figure (2.13). Nous effectuons plusieurs simulations en fonction des différents paramètres de notre code correcteur : la taille du fichier à transmettre (qui varie de 1mo à 5Go), la dimension du code ou la longueur du code.

	$d=3$	$d=4$	$d=5$	$d=6$	$d=7$	$d=8$	$d=9$	$d=10$	$d=11$	$d=12$
1mo	0.007	0.010	0.013	0.016	0.019	0.021	0.023	0.024	0.029	0.030
2mo	0.015	0.020	0.025	0.031	0.036	0.041	0.045	0.051	0.055	0.061
5mo	0.040	0.052	0.069	0.087	0.098	0.120	0.113	0.128	0.139	0.152
10mo	0.077	0.104	0.130	0.154	0.181	0.210	0.231	0.258	0.294	0.352
20mo	0.170	0.208	0.260	0.311	0.352	0.404	0.466	0.528	0.572	0.606
50mo	0.397	0.514	0.714	0.766	0.950	1.030	1.190	1.285	1.409	1.584
100mo	0.772	1.252	1.432	1.527	1.776	2.103	2.295	2.558	3.258	3.215
200mo	1.539	2.312	2.574	3.152	3.547	4.617	4.640	5.008	5.560	6.364
500mo	3.958	5.242	6.395	7.694	9.163	9.998	11.407	12.909	14.145	15.658
1Go	7.865	10.627	12.937	15.075	18.876	20.470	22.935	29.046	29.607	32.073
2Go	16.994	22.470	27.993	32.839	41.131	44.375	48.287	53.178	58.725	62.073
5Go	40.320	53.316	65.842	81.093	95.301	108.276	122.183	141.671	156.640	167.611

FIGURE 2.11 – Temps d'exécution de l'encodage pour un bloc de $k = 1000$ paquets, chaque paquet étant de taille $len = 1500octets$, pour différents ensembles P et différentes tailles de fichiers. L'ensemble P est $[7, 11, 13, 17, 23, 29, 31, 32, 37, 41, 43, 45]$, et la colonne $d = 5$ signifie qu'on a encodé chaque bloc avec les 5 premiers éléments de P, $[7, 11, 13, 17, 23]$. Les temps ici sont en secondes et ne représentent que le temps de l'encodage, ils n'incluent pas le temps d'émission des paquets.

La figure (2.11) représente le temps d'exécution de l'algorithme d'encodage en ne considérant que les calculs effectués pour l'encodage. Nous fixons $k = 1000$ paquets, $len = 1500octets$ et faisons varier P et la quantité d'information totale à transmettre. Ainsi, un fichier de $10mo$ représente un total de 6667 paquets, qui vont être réunis en 6 blocs de 1000 paquets et 1 bloc de 667 paquets, et un fichier de $1Go$ représente un total de 666667 paquets réunis en 666 blocs de 1000 paquets et 1 bloc de 667 paquets. On remarque bien que le temps d'exécution de l'algorithme est environ 100 fois supérieur pour l'encodage d'un fichier de $1Go$ que pour l'encodage d'un fichier de $10mo$, et ce quel que soit le tableau P utilisé. De plus, la figure 2.11 nous permet d'observer que pour une taille de fichier fixée (et donc un nombre de paquets et de blocs fixé), le temps d'exécution progresse avec le cardinal de P. Ceci s'explique par le fait que pour $d = 3$ par exemple, chaque paquet source est Xoré $d = 3$ fois, alors que pour $d = 12$, chaque paquet source est Xoré $d = 12$ fois. Ces observations confirment bien que la complexité de l'algorithme d'encodage est proportionnelle à d et à la taille du fichier qu'on souhaite encoder, et que donc sa complexité est en $\mathcal{O}(d * k * n_{blocs})$.

	$d=3$	$d=4$	$d=5$	$d=6$	$d=7$	$d=8$	$d=9$	$d=10$	$d=11$	$d=12$
1mo	0.019	0.016	0.021	0.028	0.029	0.031	0.032	0.033	0.040	0.038
2mo	0.031	0.037	0.044	0.048	0.055	0.060	0.061	0.070	0.081	0.083
5mo	0.084	0.098	0.114	0.144	0.149	0.196	0.164	0.180	0.188	0.200
10mo	0.154	0.185	0.219	0.238	0.270	0.307	0.328	0.359	0.413	0.504
20mo	0.365	0.366	0.425	0.491	0.533	0.597	0.654	0.749	0.777	0.820
50mo	0.813	0.953	1.197	1.192	1.482	1.471	1.713	1.792	1.978	2.156
100mo	1.597	2.376	2.549	2.485	2.717	3.161	3.285	3.571	4.609	4.487
200mo	3.084	4.679	4.299	5.036	5.428	7.406	6.724	6.980	7.697	8.892
500mo	8.168	9.392	10.879	12.216	14.000	14.797	16.307	18.412	19.749	21.491
1Go	16.103	19.715	21.404	23.700	29.970	30.136	32.778	43.305	41.548	44.418
2Go	35.934	42.349	49.083	53.674	65.783	67.333	70.091	74.756	81.281	84.101
5Go	83.921	98.082	110.314	132.482	148.442	163.024	178.543	205.293	224.673	234.988

FIGURE 2.12 – Temps d'exécution de l'encodage pour un bloc de $k = 1000$ paquets, chaque paquet étant de taille $len = 1500 octets$, pour différents ensembles P et différentes tailles de fichiers. L'ensemble P est $[7, 11, 13, 17, 23, 29, 31, 32, 37, 41, 43, 45]$, et la colonne $d = 5$ signifie qu'on a encodé notre bloc avec le sous-ensemble à 5 éléments $P = [7, 11, 13, 17, 23]$. Les temps ici sont en secondes et représentent le temps de l'encodage et de l'émission.

La figure (2.12) représente le temps d'exécution de l'algorithme d'encodage en tenant compte du temps d'émission des paquets. On remarque qu'ici aussi, le temps d'exécution total progresse linéairement avec le nombre de blocs à encoder (et donc la taille du fichier à transmettre). De plus, on remarque que pour une taille de fichier fixée (et donc un nombre de blocs fixé), le temps d'exécution de l'encodage est là aussi quasiment proportionnel au cardinal de P : encoder les blocs d'un fichier de $5Go$ avec $P = [7, 11, 13, 17, 23, 29, 31, 32, 37, 41, 45]$ prendra $167,611/40,320 = 4,14$ fois plus de temps que de les encoder avec $P = [7, 11, 13]$.

Les débits atteignables varient donc selon le cardinal de P : pour $P = [7, 11, 13]$, on atteint un débit de $60 mo/sec$, alors que pour $P = [7, 11, 13, 17, 23, 29]$ on atteint un débit inférieur, d'environ $40 mo/secs$.

	$d=3$	$d=4$	$d=5$	$d=6$	$d=7$	$d=8$	$d=9$	$d=10$	$d=11$	$d=12$
1mo	0,368	0,625	0,619	0,571	0,655	0,677	0,718	0,727	0,725	0,789
2mo	0,483	0,540	0,568	0,645	0,654	0,683	0,737	0,728	0,679	0,734
5mo	0,476	0,530	0,605	0,604	0,657	0,612	0,689	0,711	0,739	0,76
10mo	0,5	0,562	0,593	0,647	0,670	0,684	0,704	0,718	0,711	0,698
20mo	0,465	0,568	0,611	0,6334	0,660	0,676	0,712	0,704	0,736	0,739
50mo	0,488	0,539	0,596	0,6426	0,641	0,700	0,694	0,717	0,712	0,734
100mo	0,483	0,526	0,561	0,6144	0,653	0,665	0,698	0,716	0,706	0,716
200mo	0,499	0,494	0,598	0,6258	0,653	0,623	0,690	0,717	0,722	0,715
500mo	0,484	0,558	0,587	0,6298	0,6545	0,675	0,699	0,701	0,716	0,728
1Go	0,488	0,539	0,604	0,6360	0,629	0,679	0,699	0,670	0,712	0,722
2Go	0,472	0,530	0,570	0,6118	0,625	0,659	0,688	0,711	0,722	0,738
5Go	0,480	0,543	0,596	0,6121	0,642	0,66	0,684	0,690	0,697	0,713

FIGURE 2.13 – Ratio entre le temps d'encodage et le temps total (encodage + emission) pour un bloc de $k = 1000$ paquets, chaque paquet étant de taille $len = 1500 octets$, pour différents ensembles P et différentes tailles de fichiers. L'ensemble P est $[7, 11, 13, 17, 23, 29, 31, 32, 37, 41, 43, 45]$, et la colonne $d = 5$ signifie qu'on a encodé notre bloc avec le sous-ensemble à 5 éléments $P = [7, 11, 13, 17, 23]$.

La figure 2.13 présente le ratio entre le temps d'encodage d'un bloc et le temps total de son encodage et de son émission. Ces simulations ont été réalisées sur des fichiers de différentes tailles, mais on observe que la taille d'un fichier ne joue que très peu sur ce ratio. En effet, chaque bloc étant traité de la même manière, il n'y a pas de raison que ce ratio varie en fonction du nombre de blocs. A contrario, en faisant varier le cardinal de P, on observe que le temps d'encodage prend de plus en plus le pas sur le temps d'émission : ceci s'explique par le fait que le nombre de Xors à effectuer pour chaque paquet augmente avec d. Cependant, le nombre de paquets de redondance à transmettre progresse lui aussi avec d, ainsi que le temps d'émission, mais moins vite que le temps d'encodage.

Complexité de l'encodage : P, k et len

Comme dit précédemment, le temps d'exécution de l'algorithme d'encodage dépend du cardinal de P, de la taille des blocs à encoder k et de la taille des paquets constituant ces blocs len. Ces simulations démontrent que le temps nécessaire à l'encodage est proportionnel à k et len, et quasiment proportionnel à $|P|$. Ici, nous présentons des simulations dans lesquelles nous faisons varier les paramètres P, k et len afin d'observer leur impact sur le temps d'exécution de l'encodage d'un bloc seul.

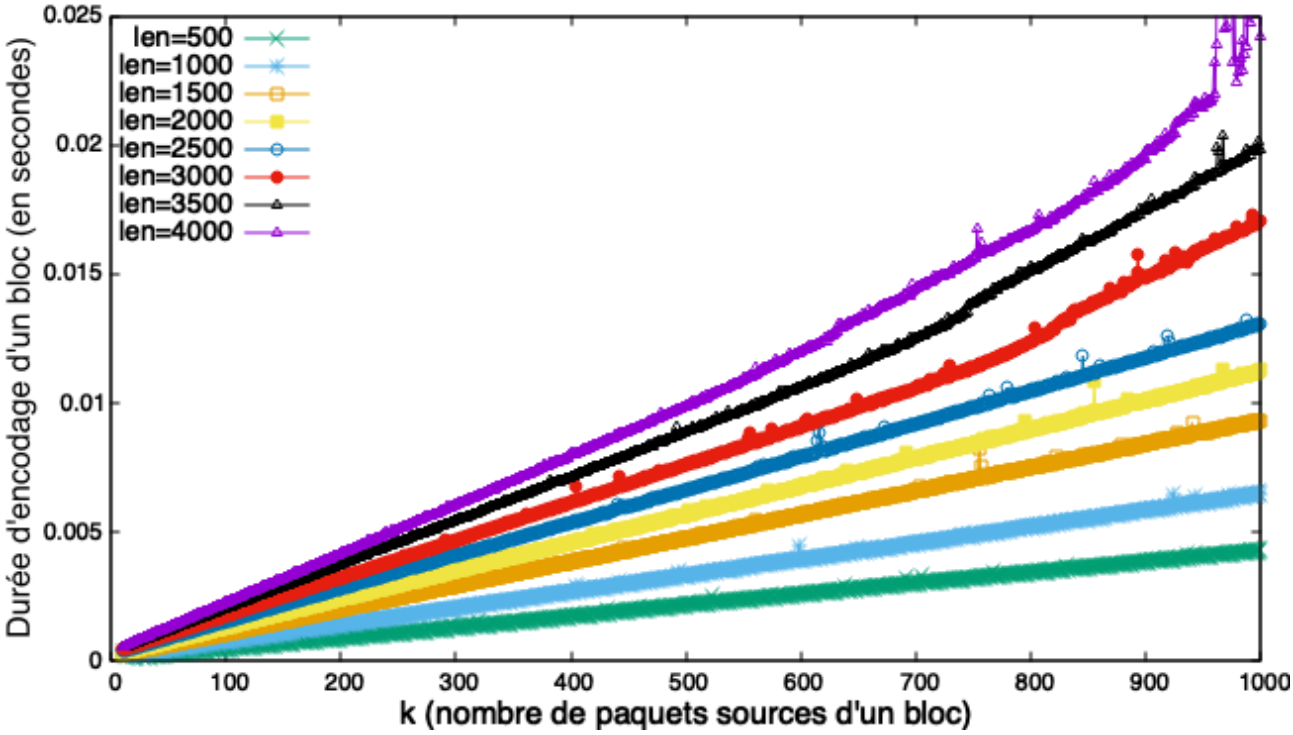


FIGURE 2.14 – Temps de calcul en secondes de l'encodage pour $P = [7, 11, 13]$ en fonction de k, pour différentes valeurs de len.

Tout d'abord intéressons-nous aux impacts de la taille des paquets len et la taille des blocs k (la dimension du code). La figure (2.14) représente les temps de calcul nécessaires à l'encodage pour $P = [7, 11, 13]$ et $n - k = 7 + 11 + 13 + 1 = 32$, en faisant varier la taille des paquets len qui prend les valeurs $500octets$, $1000octets$, $1500octets$, $2000octets$, $2500octets$, $3000octets$, $3500octets$ et $4000octets$, et k, qui prend des valeurs comprises entre 10 et 1000. Pour chaque courbe chaque point correspond à 1000 simulations dont on a calculé la moyenne.

On remarque que pour des paquets de taille len fixée, le temps de calcul est bien proportionnel au nombre de paquets sources d'un bloc k. On remarque aussi que pour k fixé le temps de calcul est quasiment proportionnel à len. Ceci confirme bien les observations faites dans les figures précédentes (2.12) et (2.11).

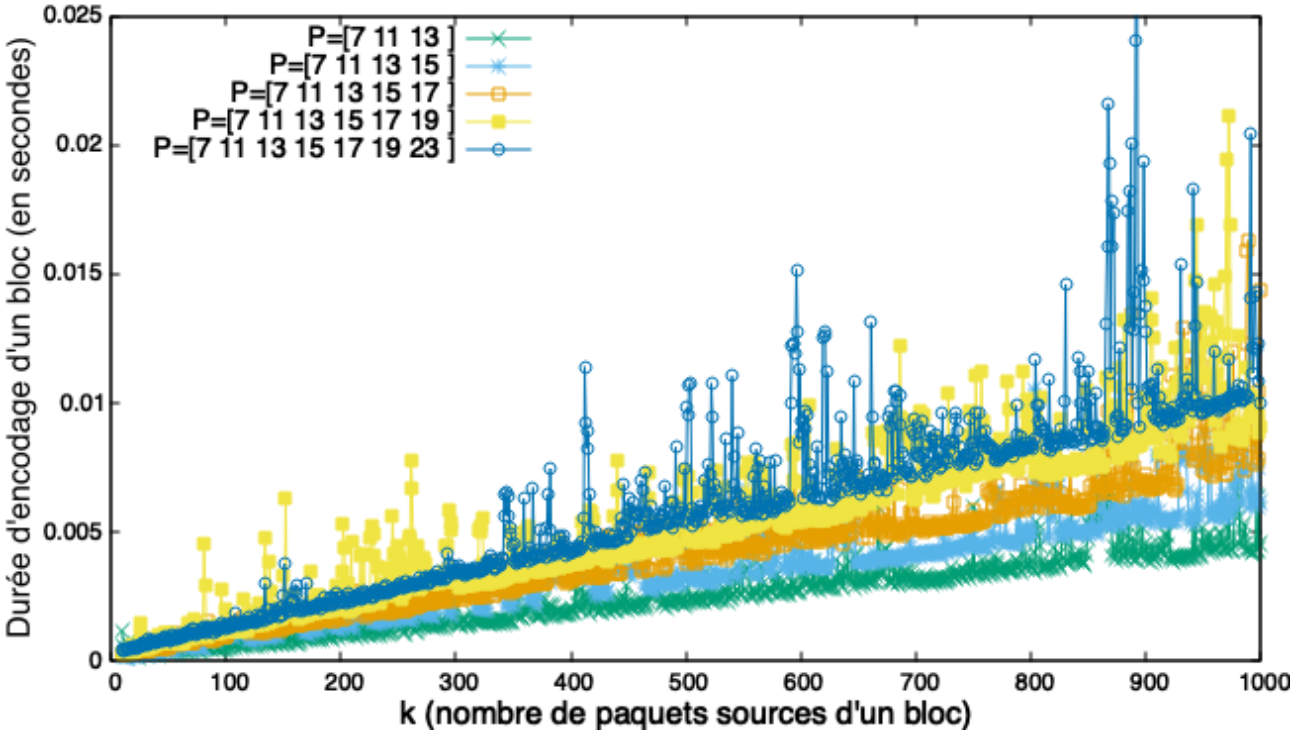


FIGURE 2.15 – Temps de calcul en secondes de l'encodage pour $k = 1000$ en fonction de k, pour différentes valeurs de P.

La figure (2.17) représente les temps de calcul de l'encodage pour des paquets de taille $len = 1000 octets$, en faisant varier le cardinal de P de 3 à 7 et la taille des blocs encodés k. On remarque que pour un P fixé, le temps de calcul de l'encodage est proportionnel à k, et que pour k fixé, le temps de calcul est alors quasiment proportionnel à $|P|$.

2.4.2 Simulations du décodage

Tout comme pour la partie précédente, nous présentons ici de nombreuses simulations, permettant cette fois-ci d'observer les temps de calculs nécessaires au décodage, en fonction de P, k, len, ou encore de la probabilité d'effacements du canal. Nous avons vu précédemment que le décodage était divisé en deux étapes : les calculs effectués à la réception des paquets, et les calculs effectués par l'algorithme de décodage itératif : les simulations que nous présentons ici concernent cette deuxième étape.

Afin de comparer des codes de différents paramètre k, n et P, nous avons choisi de présenter les différents temps d'exécution en fonction de la probabilité d'effacements sur le canal, p. Ainsi, nous pouvons superposer plusieurs courbes correspondant à des codes différents et observer l'influence du paramétrage de ces codes sur différentes métriques : pour un code de longueur n et une probabilité de pertes p, on aura environ $e = p * n$ paquets effacés, et donc $e = p * n$ à réparer par le décodage.

Rappelons l'équation représentant le temps de calcul du décodage en fonction des différents paramètres de notre code P, k et len :

$$\mathcal{O}((n-k) * max(|P|, 2) * len) \tag{2.6}$$

On sait qu'il est impossible lors de l'algorithme de décodage itératif de décoder plus de $n-k$ paquets. On peut cependant réécrire cette équation, pour faire apparaître $e \leq n-k$, ce qui donne alors

$$\mathcal{O}(e * max(|P|, 2) * len) \tag{2.7}$$

ou bien la réécrire en utilisant le taux de perte du canal p, dont va dépendre le nombre de paquets effacés à corriger $e = p * n$:

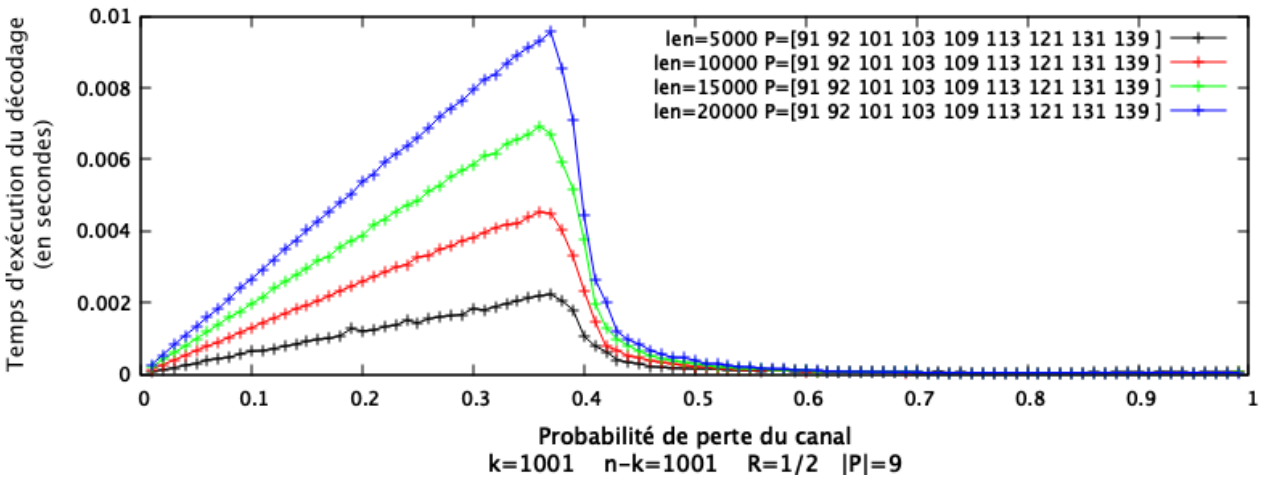
$$\mathcal{O}(p * n * max(|P|, 2) * len) \tag{2.8}$$

Enfin, nous pouvons préciser encore plus cette équation, en y faisant apparaître distinctement le nombre de paquets sources k et le nombre de paquets codes $n-k$ d'un bloc :

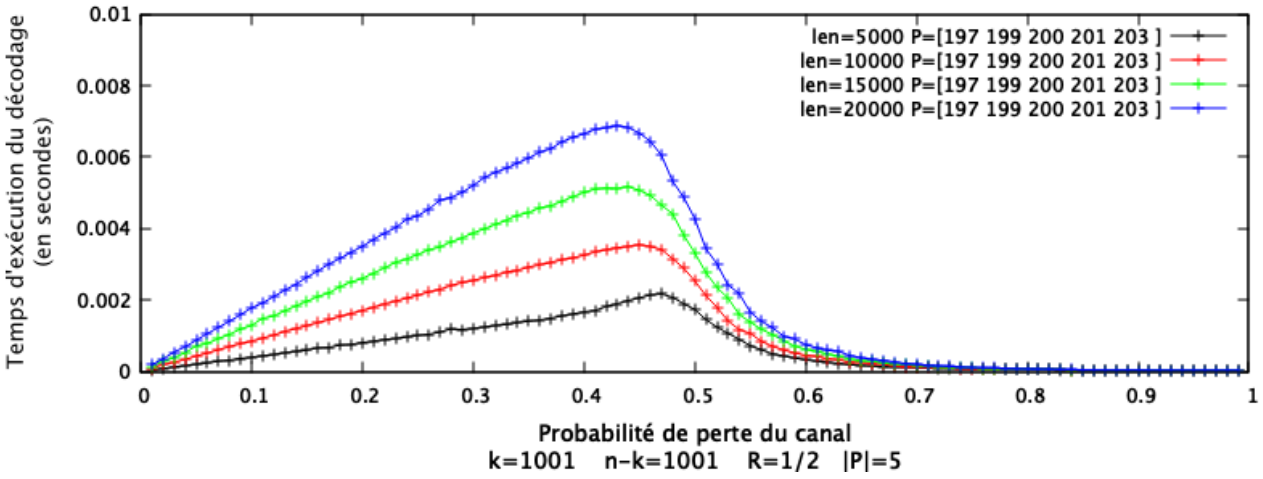
$$\mathcal{O}(p * ((k+1) * |P| + (n-k) * 2) * len) \tag{2.9}$$

Complexité du décodage : influence de la taille des paquets *len*

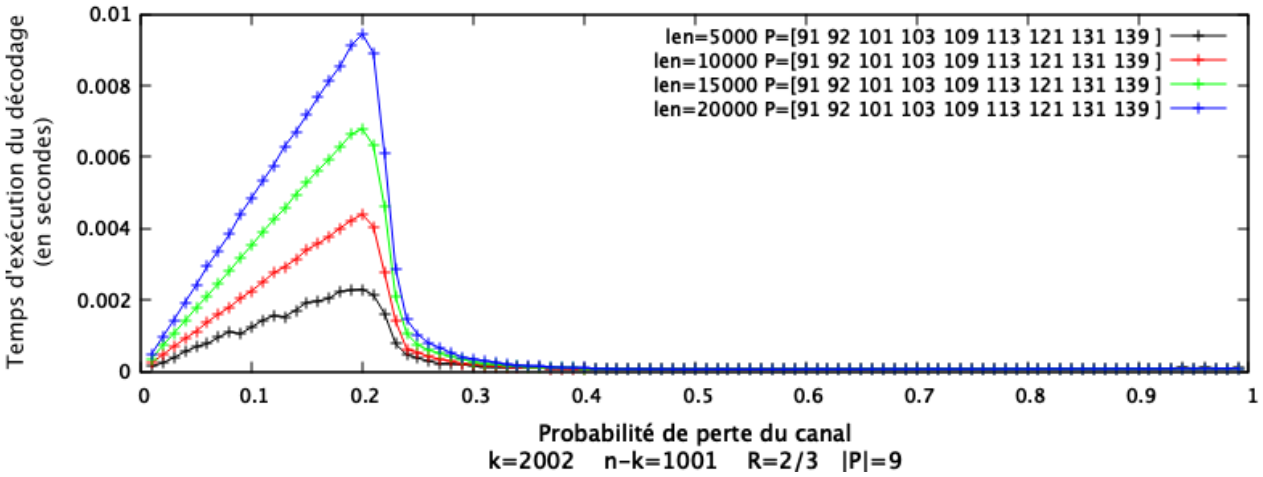
Dans un premier temps, nous présentons des simulations pour le décodage pour différents paramétrages de notre code, en faisant varier la taille des paquets *len*, qui a une influence directe sur le temps d'exécution du décodage.



Cette première figure représente donc les temps nécessaires au décodage pour un code de dimension 1001 et de longueur 2002, avec $P = [91, 92, 101, 103, 109, 113, 121, 131, 139]$, pour lesquels on a fait varier la taille des paquets *len*, prenant des valeurs parmi $[5000, 10000, 15000, 20000]$ octets. On observe ici que le temps de calcul augment linéairement avec la taille des paquets : en effet, chaque fois qu'on Xore un paquet, on effectue en réalité *len* Xors binaires. Par conséquent, le temps de calcul du décodage sera proportionnel à la taille *len* des paquets. La figure ci-dessous représente le même type de simulations que la figure précédente, mais pour $P = [197, 199, 200, 201, 203]$. On remarque là-aussi que le temps de calcul du décodage est bien proportionnel à la taille des paquets *len*.



Enfin, nous présentons une dernière simulation basée sur le même principe que pour les figures précédentes, mais pour une dimension $k = 2002$, pour obtenir un code de rendement $R = 2002/3003 = 2/3$. Là-aussi, le temps de calcul est proportionnel à *len* pour un nombre d'effacements $e = p * n$ fixé.

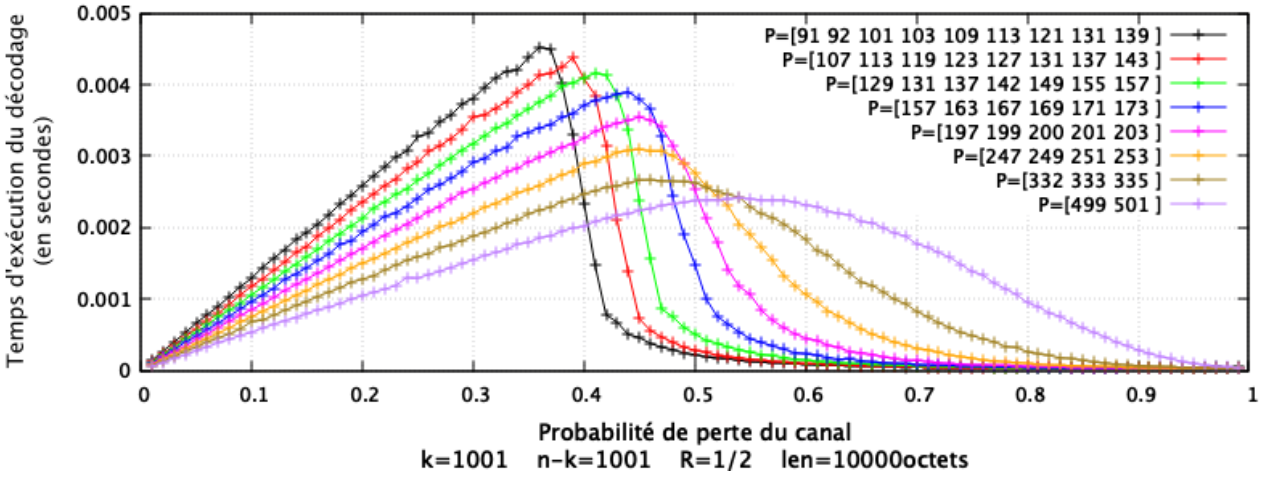


Ces trois figures combinées nous permettent donc d'observer que le temps de calcul du décodage est bien proportionnel à la taille des paquets, toutes choses égales par ailleurs : pour un même paramétrage de notre code (P, k et $n-k$ fixés), et pour un taux de perte p, on aura à réparer $e = p*n$ paquets, et plus ces paquets auront une taille *len* importante, plus le décodage prendra du temps. Le temps de calcul du décodage progresse donc bien linéairement avec la taille des paquets qui constituent nos blocs.

Complexité du décodage : influence du paramètre P

Les simulations que nous présentons ci-dessous permettent d'observer l'influence du paramètre P, et notamment de son cardinal $|P|$ sur le temps d'exécution de l'algorithme de décodage. Rappelons que P est un ensemble de nombres premiers entre eux que partagent l'émetteur et le récepteur, et donc on déduit la matrice génératrice de notre code, et donc les combinaisons linéaires à effectuer pour encoder ou décoder nos blocs. Ces simulations ont été réalisées de la manière suivante :

— Chaque courbe correspond à un paramétrage du code, pour des paquets de taille $len = 10000 octets$ et une longueur $n - k = 1001$.
— Pour chaque courbe, nous choisissons un ensemble P dont la somme vaut toujours 1001, et pour lesquels nous avons bien vérifié qu'ils ne contiennent que des nombres premiers entre eux.

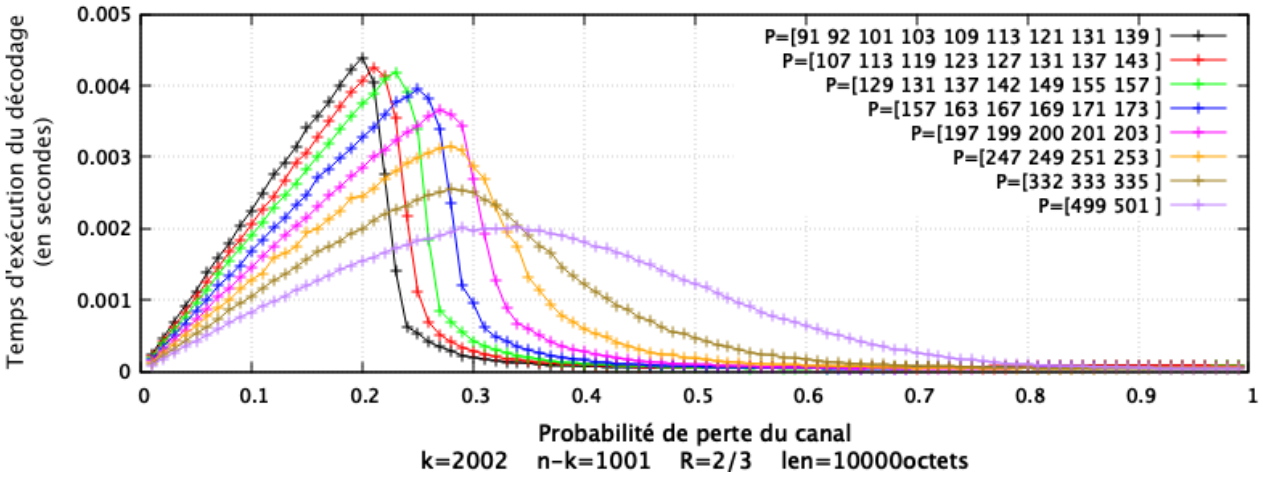


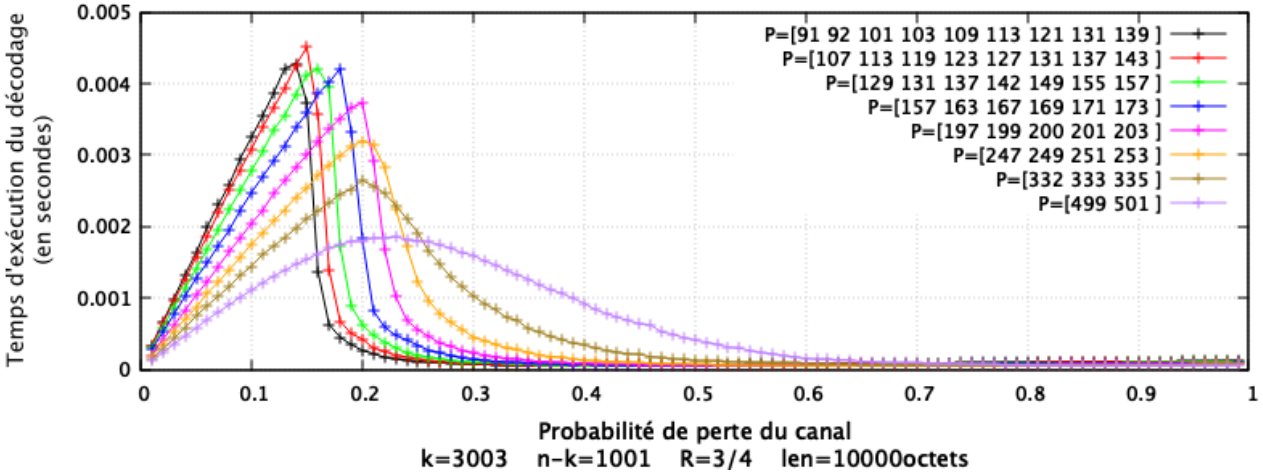
La figure ci-dessus représente donc des simulations de décodage pour $k = 1001$, $n - k = 1001$, $n = 2002$, et donc un rendement $R = k/n = 1001/2002 = 1/2$. Elle se lit de la manière suivante : pour une probabilité de perte $p = 0.2$, des paquets de taille $len = 10000 octets$ et un paramétrage de notre code $P = [332, 333, 335]$ et $k = 1001$, le décodage de $e = p * n$ prend environ $1.2 millisecondes$.

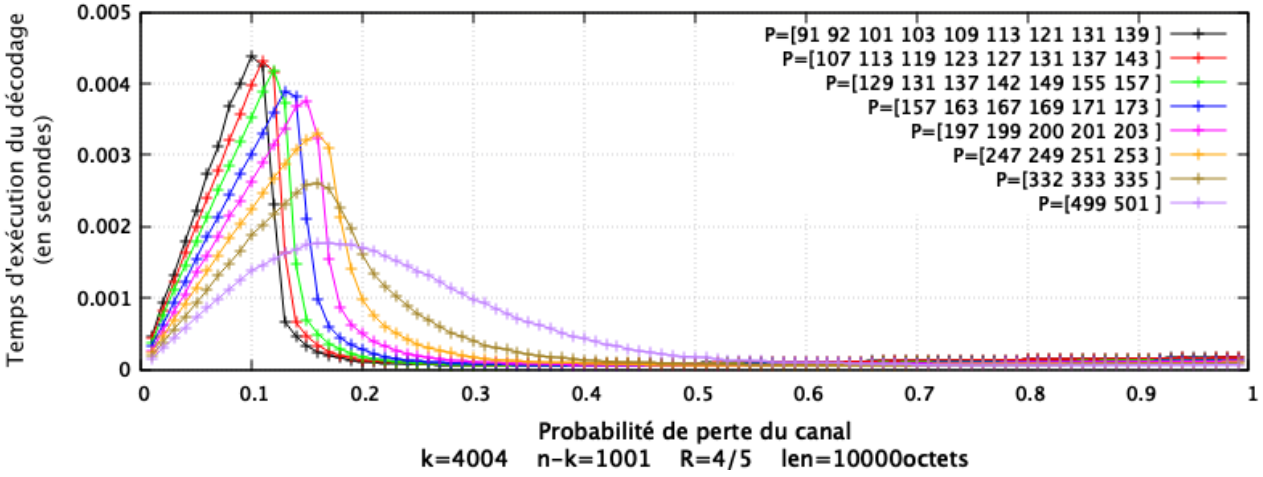
On peut remarquer que le temps de calcul progresse avec le cardinal de P : en effet, pour chaque source réparé, on doit alors Xorer celui-ci dans les $|P| + 1$ nœuds contraintes lui correspondant.

On peut aussi observer que le temps de calcul progresse avec le nombre d'effacements à corriger, tant que l'algorithme itératif parvient à corriger tous les paquets effacés. Une fois que le taux de perte devient trop important pour que l'algorithme de décodage parvienne à réparer des paquets, le temps de calcul diminue alors rapidement : ceci arrive quand le taux de pertes dépasse 0.37 pour $P = [91, 92, 101, 103, 109, 113, 121, 131, 139]$, et quand il dépasse 0.45 pour $P = [499, 501]$. On pourrait croire ici que la capacité de correction du code correspondant à $P = [91, 92, 101, 103, 109, 113, 121, 131, 139]$ est moins bonne que pour $[499, 501]$ (puisque le temps de calcul pour $P = [499, 501]$ reste grand pour des valeurs de p supérieures à 0.5), mais nous verrons par la suite qu'il n'en est rien, et que c'est même l'inverse.

Les figures suivantes représentent les mêmes simulations que la figure précédente, pour lesquels nous avons fait varier k et donc le rendement du code R, qui prend des valeurs parmi $R \in [2/3, 3/4, 4/5, 5/6]$: elles permettent de confirmer les observations faites pour la première courbe (rendement $R = 1/2$). En effet, on remarque que le temps de calcul du décodage ne dépend pas de k la dimension du code, mais de $e = p * n$, le nombre approximatif d'effacements du bloc. Ainsi, pour chaque paramétrage (k, n) de nos codes, on observe que le temps de calcul est maximal quand p s'approche de la valeur $(n - k)/n$: ceci correspondant bien à la situation ou le bloc a subi à peu près autant d'effacements que le nombre de paquets de redondance. Notre code correcteur n'étant pas un code MDS, cette situation arrive en réalité quand p est légèrement plus petit que $(n - k)/n$.

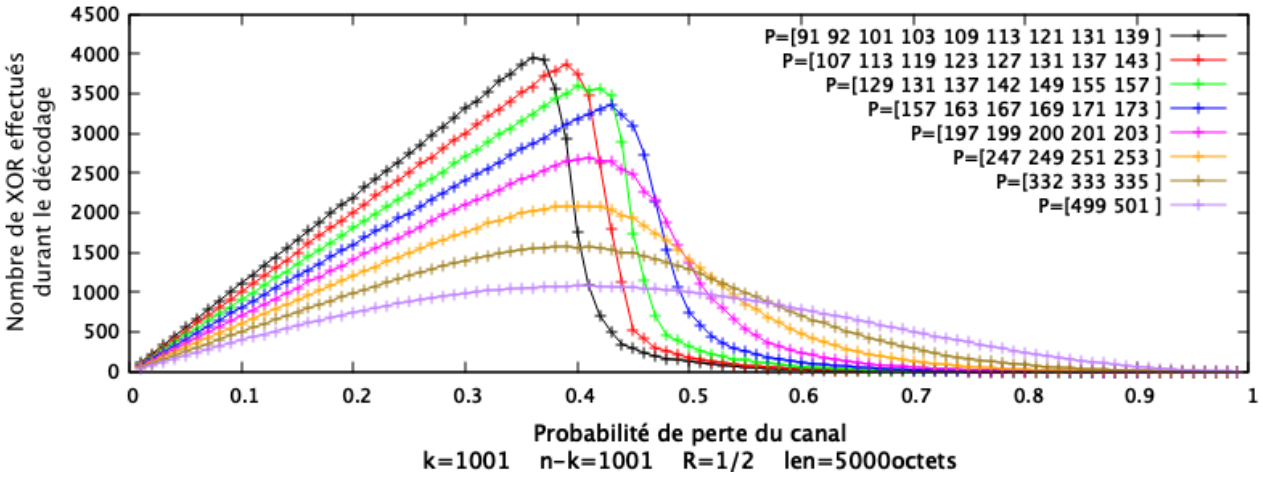






Les calculs effectués par le décodage de notre code consistant majoritairement à Xorer des paquets entre eux, nous présentons de plus une courbe sur laquelle figure le nombre de Xors moyens effectués lors du décodage d'un bloc : suivant la puissance de calcul de la machine utilisée pour décoder un bloc, les Xors prendront plus ou moins de temps, alors que leur nombre

restera le même. Il est donc intéressant de présenter le nombre Xors moyen à effectuer lors du décodage. Ces simulations correspondent aux mêmes simulations effectuées pour les courbes précédentes, mais ne représentent donc pas le temps de calcul, mais le nombre de Xors effectués au cours du décodage.

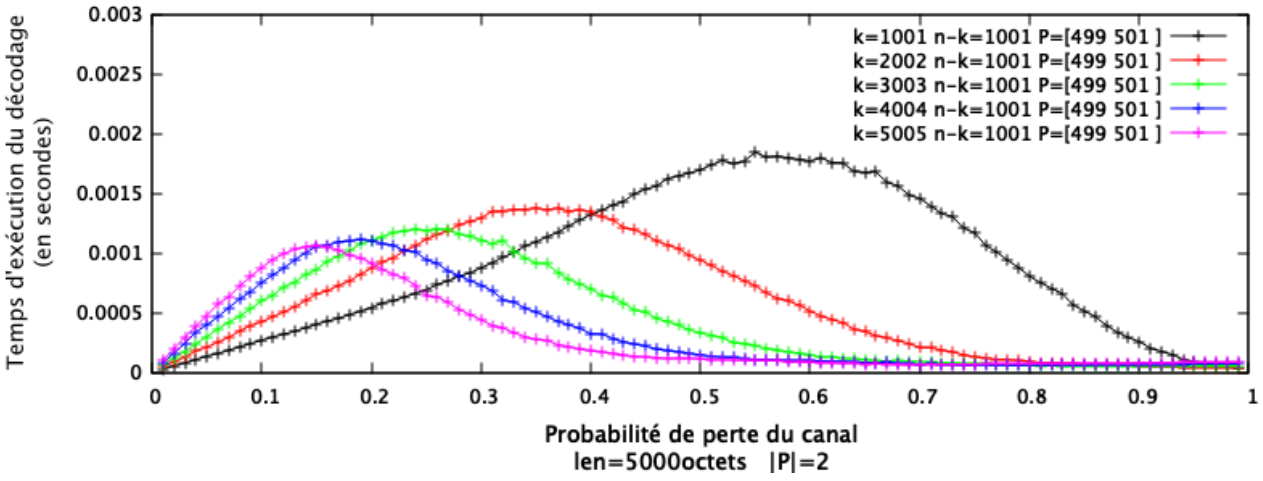


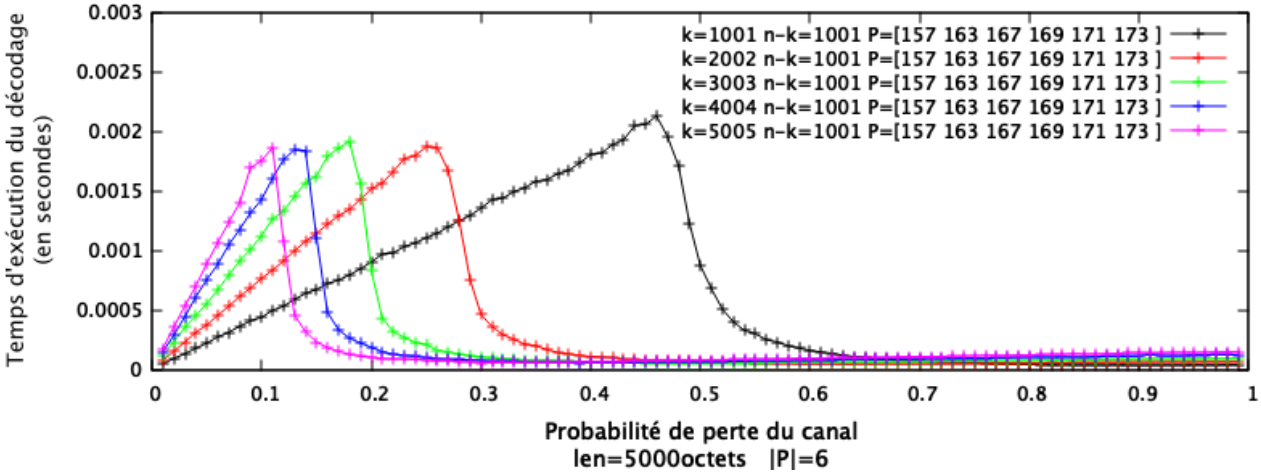
On remarque que cette figure représente des courbes qui se superposent presque exactement avec les courbes de la figure () : le temps de calcul est donc bien proportionnel au nombre de Xors effectués lors du décodage (modulo les calculs nécessaires à l'algorithme, tels que le parcourt de listes, le retrait d'éléments d'une liste, ou l'incrémentation de compteurs).

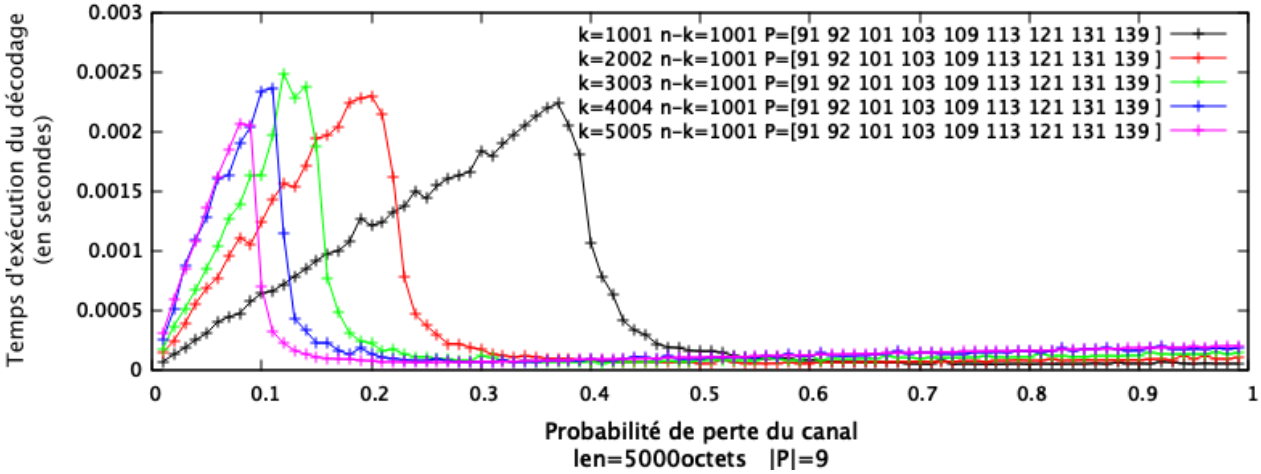
Complexité du décodage : influence des paramètres k et n

Dans cette partie, nous présentons des simulations permettant d'observer l'influence de la longueur n de notre code sur la durée du décodage. Rappelons que le paramètre n représente le nombre total de paquets (sources et codes) d'un bloc envoyé par l'émetteur, et que $n = k + (n-k) = k + PPCM(P) + 1$: notre code nous impose de fixer le nombre de paquets codes $n - k = \Sigma(P) + 1$, et pour faire varier n la taille de nos blocs, nous devons donc faire varier k le nombre de paquets sources.

Comme pour les simulations présentées précédemment, nous fixons tous les autres paramètres du code, et faisons donc varier n, en faisant en réalité varier k.







Analyses des résultats

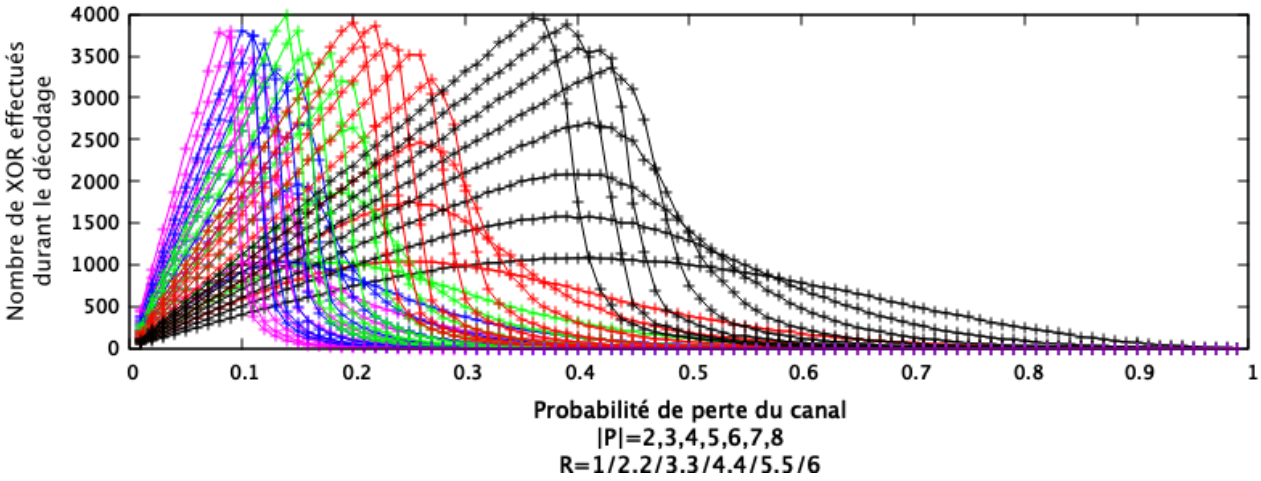


FIGURE 2.16 –

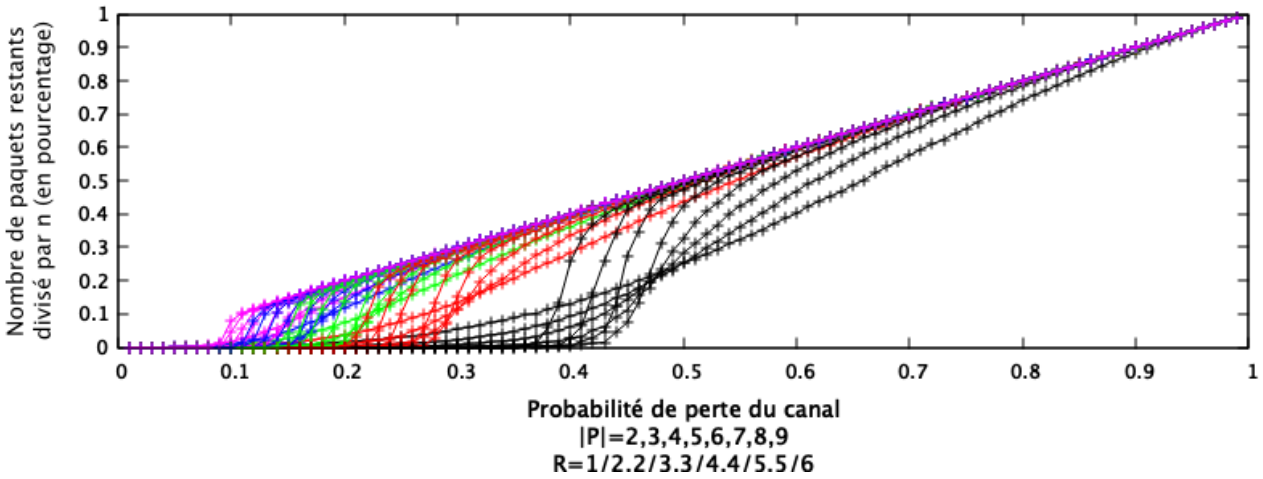


FIGURE 2.17 –

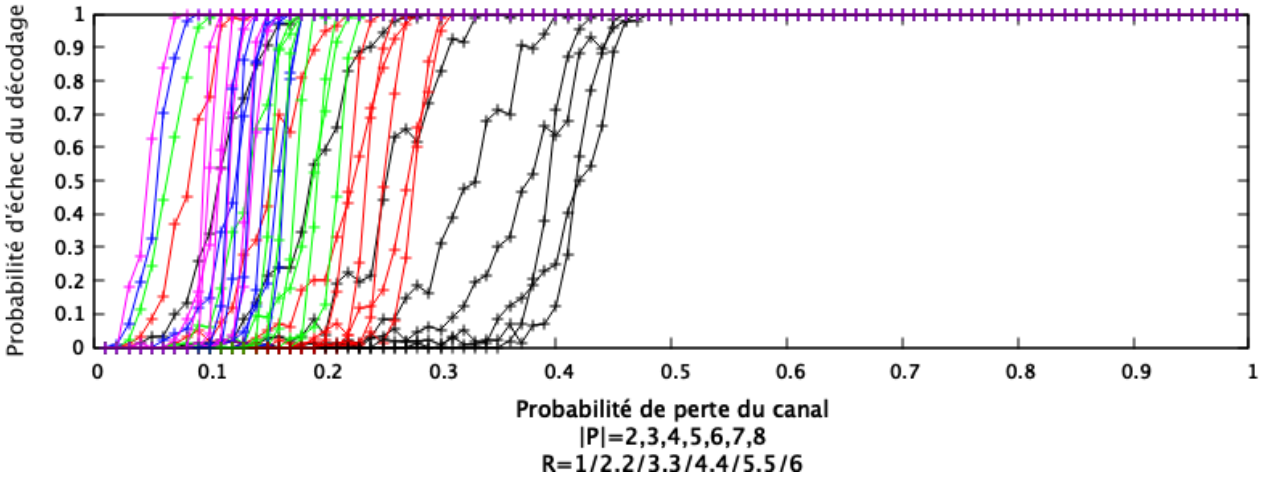


FIGURE 2.18 –

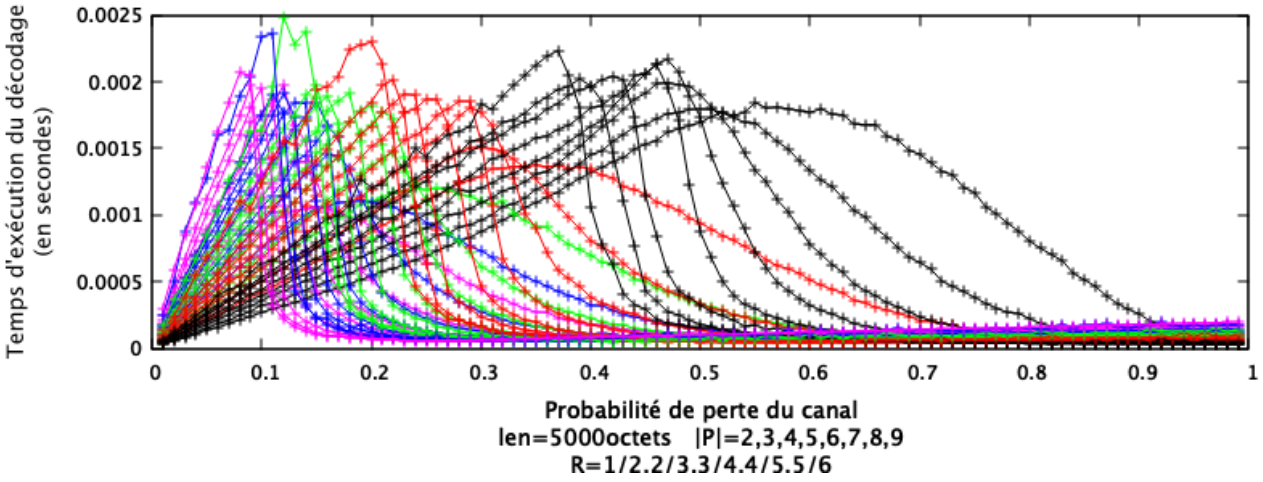


FIGURE 2.19 – Temps d'exécution du décodage en fonction de la probabilité d'effacements pour différents P

2.4.3 Capacité de correction

Comme dit dans le chapitre 1, la capacité de réparation d'un code LDPC pour le canal à effacements (et de tous les codes linéaires de manière générale) dépend grandement de la matrice génératrice du code, et notamment des degrés (colonnes et lignes, d_c et d_l) de cette matrice. Or, le code que nous présentons est un code LDPC irrégulier, dont la matrice génératrice dépend d'un ensemble de d nombres premiers entre eux P. Le cardinal de P va directement nous donner le degré par colonne de la matrice génératrice, qui vaudra $d_c = d + 2$. En effet, d_c correspond au nombre de paquets sources ou de redondance qui vont contenir l'information d'un paquet source. Or, comme nous l'avons vu précédemment, chaque paquet source est Xoré dans $d + 1$ paquets de redondance différents, puis est envoyé. Par conséquent, pour chaque paquet source, on va transmettre $d + 2$ paquets contenant l'information du paquet source, et permettant potentiellement de retrouver sa valeur lors du décodage. L'ensemble P et la longueur du code k nous indique aussi le degré par ligne de notre matrice génératrice : ce degré va varier suivant les lignes de la matrice génératrice, ce qui fait de notre code un code LDPC irrégulier (en opposition aux codes LDPC réguliers, pour lesquels ce degré est constant). Pour un paquet code $C_{P_i,j}$, on aura Xoré k/P_i paquets sources, et on aura donc d_r qui variera suivant la ligne de la matrice, avec $d_r = k/P_i$, P_i étant l'élément de P ayant servi à générer le paquet $C_{P_i,j}$.

Les figures suivantes (2.20), (2.21), (2.22), (2.23), (2.24), (2.25), et (2.26), présentent les probabilités de décoder totalement un bloc encodé suivant différents ensembles P. Ici, nous avons choisi des codes tels quel $k = (n - k) = \Sigma(P) + 1$, qui ont donc tous le même rendement $R = k/n = k/(k + (n - k)) = k/(2k) = 1/2$. Ils peuvent s'interpréter de cette manière : en abscisse, nous faisons varier le nombre d'effacements d'un bloc de 0 à $(n - k)$ inclus. Il n'est en effet pas utile d'étudier la probabilité de décodage d'un bloc quand $e > (n - k)$ puisque celle-ci vaut alors 0.

Ces simulations ont été réalisées suivant la méthode de Monte-Carlo : chaque courbe est le résultat de 1000000 tests consistant en le tirage d'un nombre e (le nombre d'effacements du bloc), puis en l'effacement de e paquets choisis uniformément dans le bloc, puis décodage du bloc. Ainsi, pour chaque valeur de $0 \leq e \leq (n - k)$, on obtient un ratio $\in [0, 1]$ en divisant le nombre de succès du décodage par le nombre d'essais effectués pour chaque e, qui tend vers la probabilité de décoder un bloc avec e effacements.

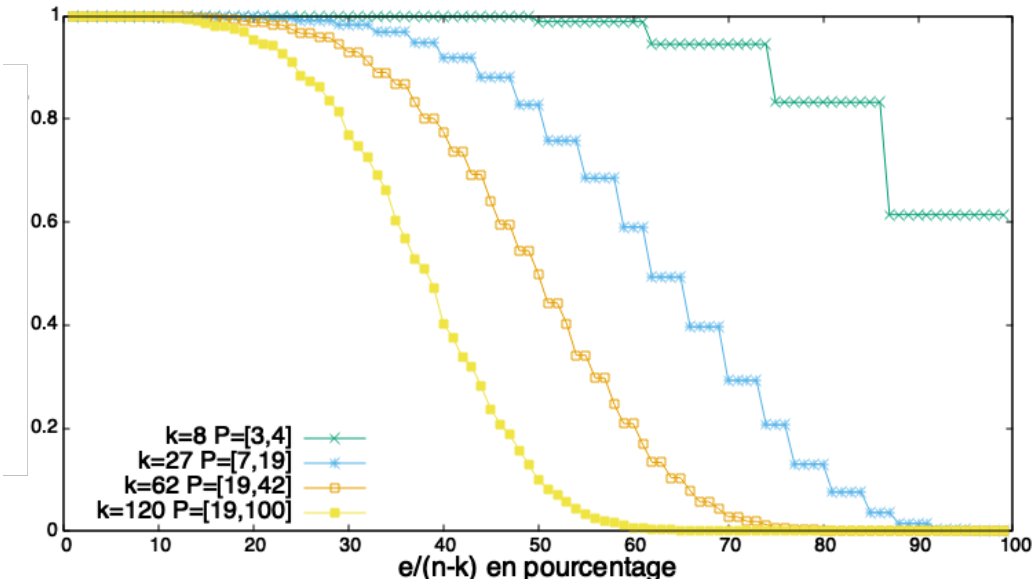


FIGURE 2.20 – Probabilité de décoder totalement un bloc en fonction du nombre d'effacements pour différents P, avec $|P| = 2$ et $k = n - k = \Sigma(P) + 1$

Il est intéressant d'étudier la figure (2.20), pour laquelle on a utilisé des ensembles P de cardinal 2 : en effet, on voit que la courbe jaune (correspondant à $k = 120$ et $P = [19, 100]$) diminue plus rapidement que la courbe verte (correspondant à $k = 8$ et $P = [3, 4]$). On remarque que les courbes jaune, orange, bleue et verte correspondent à des codes dont la capacité de correction progresse, et apparemment, un code a une meilleure capacité de correction quand les éléments de P sont proches.

Ceci peut s'expliquer de la manière suivante. Supposons qu'on ait choisi $P = [2, 1001]$ et $k = P_0 * P_1 = 2 * 1001 = 2002$, qu'on ait subi un certain nombre d'effacements, et qu'on lance l'algorithme de décodage. Les deux nœuds contraintes correspondant à $C_{2,0}$ et $C_{2,1}$ sont le résultat du Xor de 1002 paquets chacun, alors que les nœuds contraintes correspondant à $C_{1001,0}$, $C_{1001,1}$, $C_{1001,2}$,...sont le résultat du Xor de 2 paquets. Par conséquent, la probabilité de décoder des paquets avec les nœuds contraintes de $C_{2,0}$ ou de $C_{2,1}$ est beaucoup plus faible que de décoder avec un nœud contrainte avec $C_{1001,i}$.

A contrario, en prenant $P = [501, 502]$ et $k = 2002 \leq P_0 * P_1$, on aura environ la même probabilité de décoder un paquet avec un nœud contrainte de $C_{501,i}$ ou de $C_{502,i}$: les éléments de P ont ici la même efficacité.

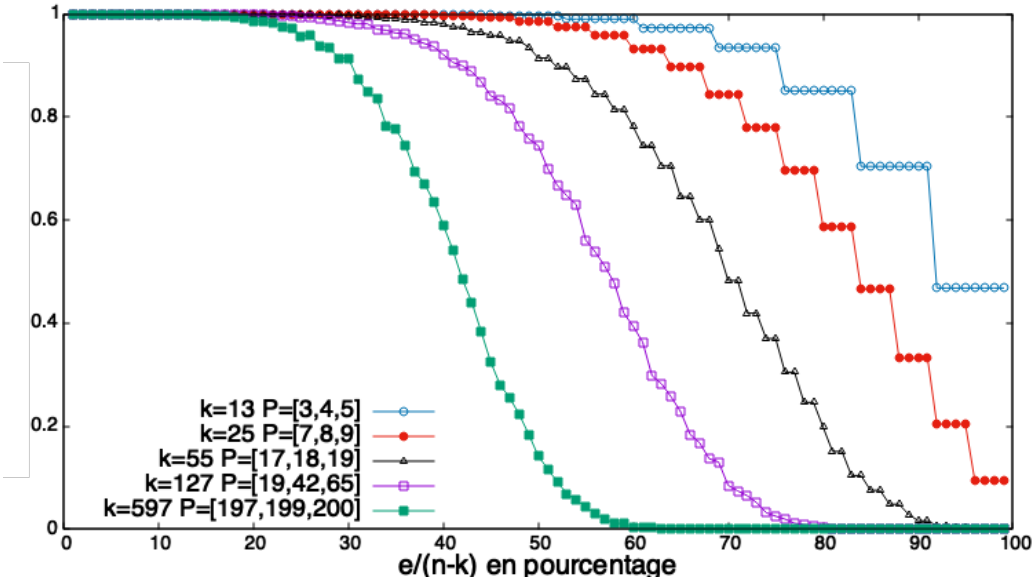


FIGURE 2.21 – Probabilité de décoder totalement un bloc en fonction du nombre d'effacements pour différents P, avec $|P| = 3$ et $k = n - k = \Sigma(P) + 1$

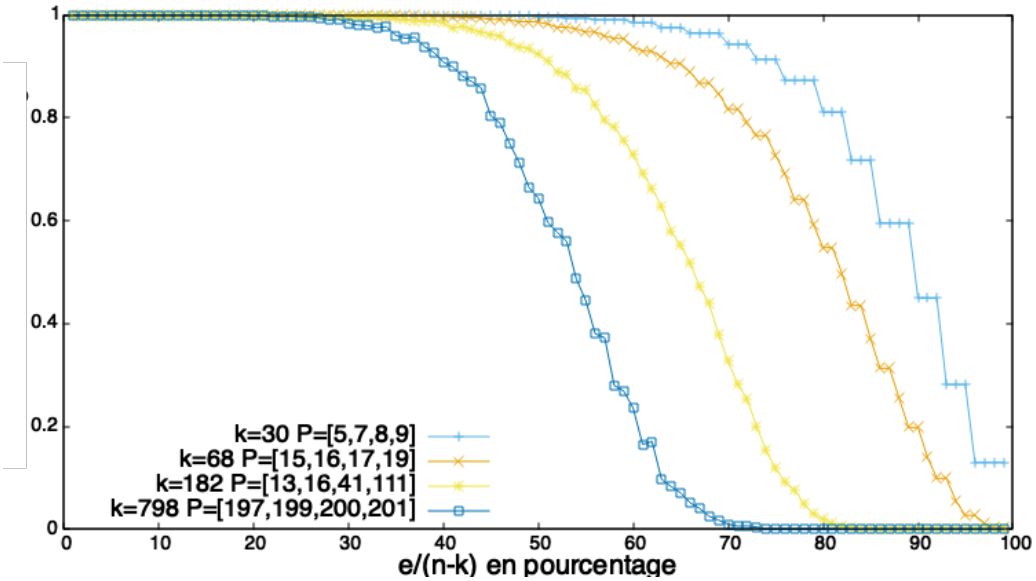


FIGURE 2.22 – Probabilité de décoder totalement un bloc en fonction du nombre d'effacements pour différents P, avec $|P| = 4$ et $k = n - k = \Sigma(P) + 1$

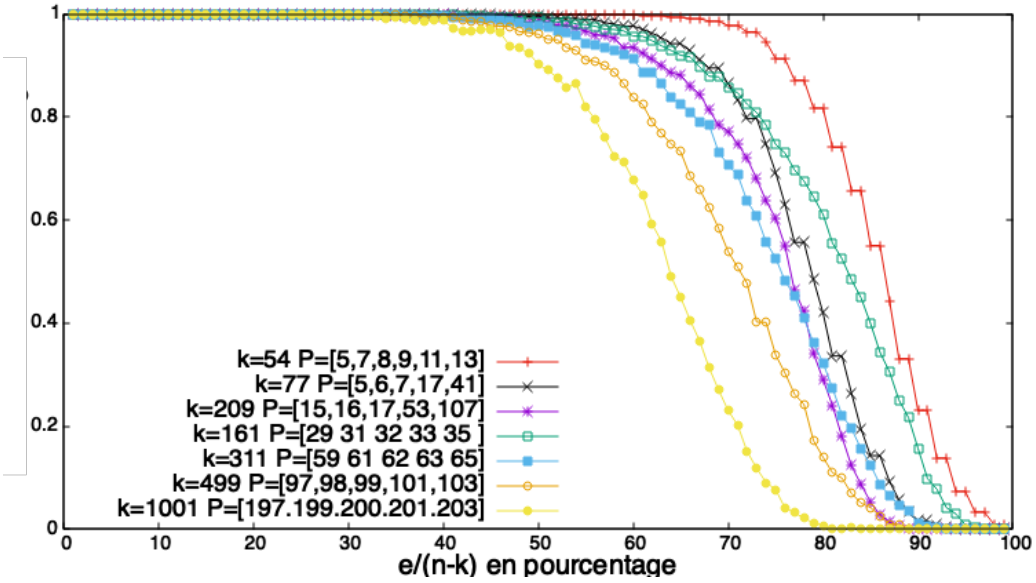


FIGURE 2.23 – Probabilité de décoder totalement un bloc en fonction du nombre d'effacements pour différents P, avec $|P| = 5$ et $k = n - k = \Sigma(P) + 1$

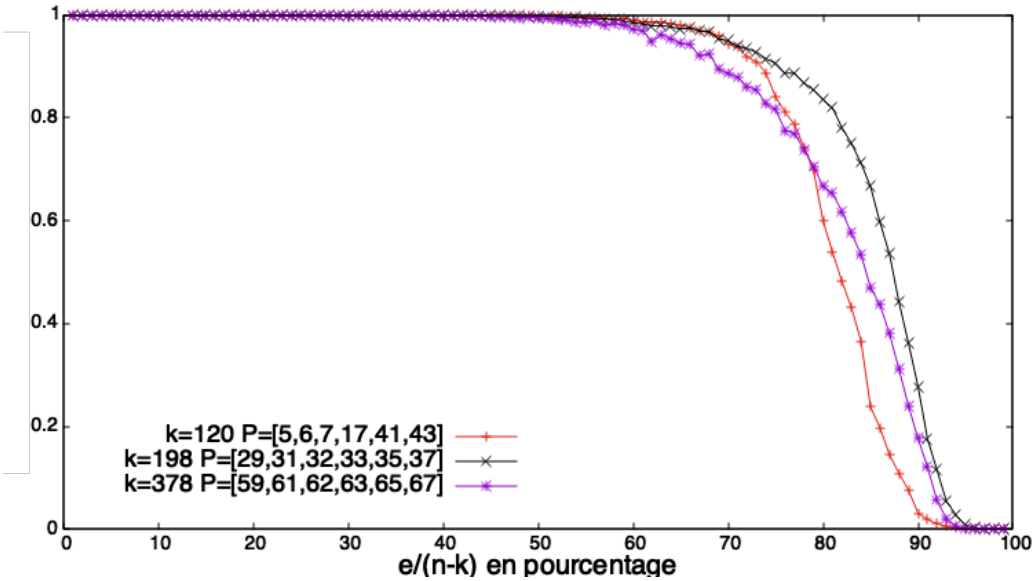


FIGURE 2.24 – Probabilité de décoder totalement un bloc en fonction du nombre d'effacements pour différents P, avec $|P| = 6$ et $k = n - k = \Sigma(P) + 1$.

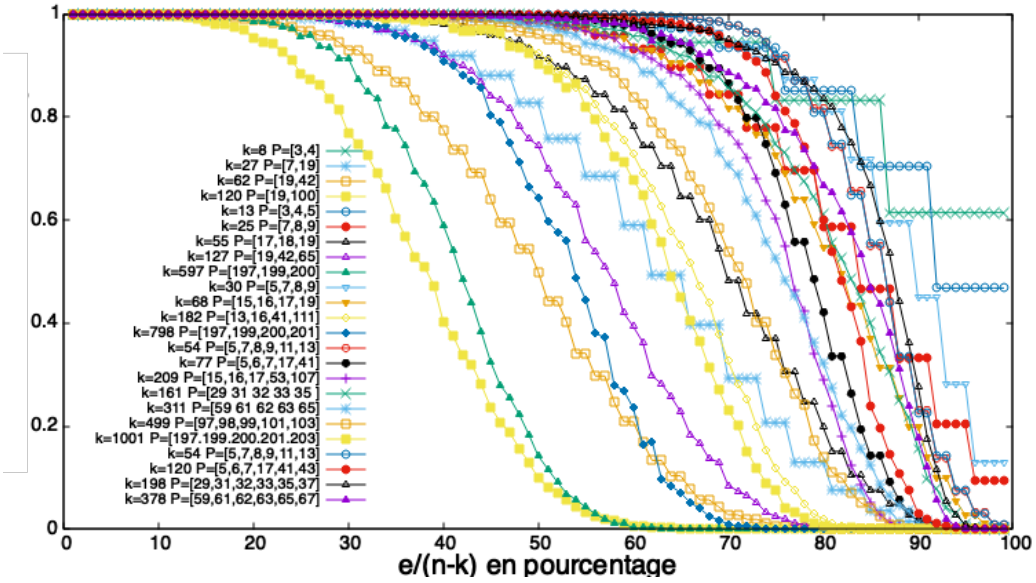


FIGURE 2.25 – Probabilité de décoder totalement un bloc en fonction du nombre d'effacements pour différents P, avec $2 \leq |P| \leq 6$ et $k = n - k = \Sigma(P) + 1$

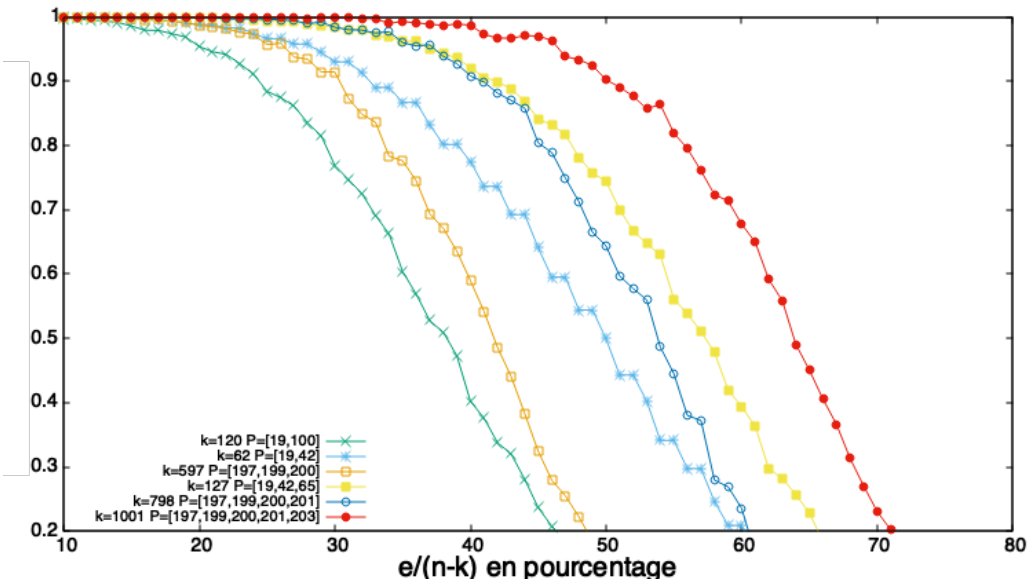


FIGURE 2.26 – Probabilité de décoder totalement un bloc en fonction du nombre d'effacements pour différents P ayant une mauvaise capacité de correction, avec $2 \leq |P| \leq 6$ et $k = n - k = \Sigma(P) + 1$. Les échelles ont été modifiées.

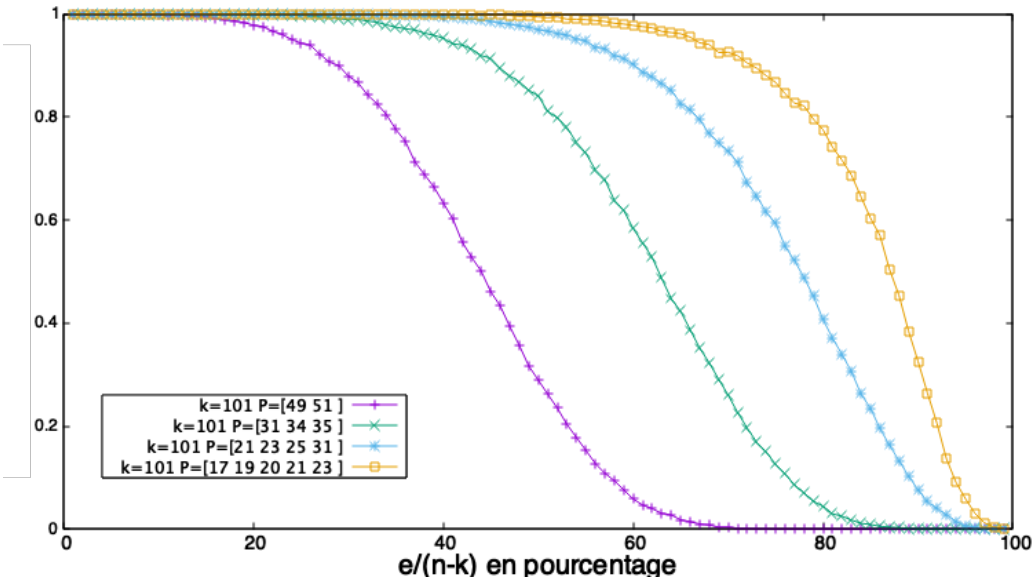


FIGURE 2.27 – Probabilité de décoder totalement un bloc en fonction du nombre d'effacements pour différents P, avec $k = n - k = \Sigma(P) + 1 = 101$.

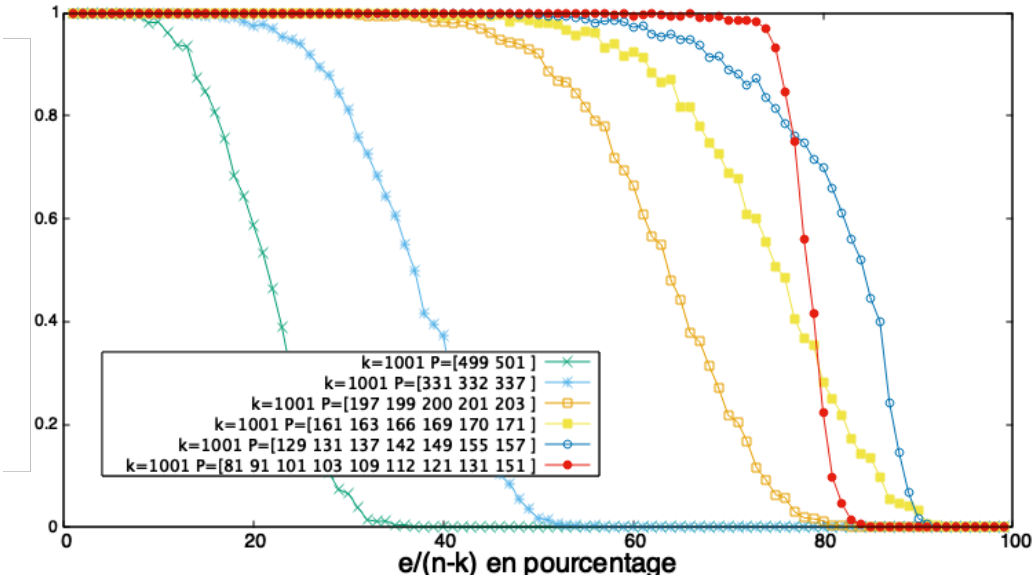


FIGURE 2.28 – Probabilité de décoder totalement un bloc en fonction du nombre d'effacements pour différents P, avec $k = n - k = \Sigma(P) + 1 = 1001$.

Afin de visualiser l'effet du cardinal de P sur l'efficacité du décodage, nous présentons dans les figures (2.28) et (2.27). Elles représentent des simulations pour lesquelles nous fixons $\Sigma(P) = 101$ ou $\Sigma(P) = 1001$ et de dimension et de longueur $k = n - k = 101$ (ou 1001), afin d'obtenir un code de rendement $R = 1/2$ pour différentes valeurs de P. Nous nous sommes assurés que chaque P utilisé contient bien des nombres premiers entre eux deux à deux. En termes de capacité de correction, la probabilité de décoder tous les paquets effacés est de quasiment 100 lorsqu'on a reçu au moins $k + 70\%(n - k)$) paquets pour certains paramétrages. C'est le cas pour $k = 1001$ et $P = [81, 91, 101, 103, 109, 112, 121, 131, 151]$, visible sur la figure

2.28 : le décodage aboutit lorsqu'on a reçu au moins 1300 paquets parmi les 2002 paquets transmis.

Ces figures nous permettent de remarquer que d'une manière générale, plus le cardinal de P est grand, plus le décodage est efficace. Il semble donc exister une relation entre le cardinal de P et la capacité de correction du code. Cependant, on atteint une sorte de plafond d'efficacité à partir d'une certaine valeur de $|P|$: quand $|P|$ dépasse cette valeur, l'efficacité du décodage progresse peu. Sachant que la complexité de l'encodage et du décodage sont proportionnelles à $|P|$, il peut donc être intéressant de choisir une relativement petite valeur de $|P|$, ce qui réduira le nombre de calculs à effectuer, pour une efficacité presque égale à des valeurs de $|P|$ plus grande.

2.5 Canal à erreurs et théorème des restes Chinois

Jusqu'à présent, nous avons présenté notre code correcteur, dans le contexte d'une transmission unidirectionnelle sur un canal à effacements. Les paquets transmis ne peuvent pas parvenir au récepteur en mauvais état : ceux qui auraient subi des erreurs sont tout simplement considérés comme effacés (ce qui est rendu possible par l'utilisation du Checksum de nos paquets, des paquets UDP).

Dans cette partie, nous présentons une version modifiée de notre code correcteur, permettant de fiabiliser une transmission, non plus sur un canal à effacements, mais sur un canal à erreurs. Nous présentons cette version modifiée de manière "Proof of Concept" : en effet, le cas d'utilisation de notre code correcteur n'étant pas du tout le canal à erreurs mais bien le canal à effacements, nous ne nous sommes donc pas immensément concentrés sur cette version du code. Elle repose néanmoins sur une utilisation habile du théorème des restes Chinois afin de déterminer les indices des paquets erronés avant de les réparer.

2.5.1 Adaptation au canal à erreurs

Les calculs effectués par l'émetteur et le récepteur sont exactement les mêmes que pour la version du code que nous avons présentée précédemment, pour ce qui est des combinaisons linéaires des paquets sources entre eux pour obtenir les paquets de redondance. Le code nécessite là encore le paramètre P, qui nous indique le nombre fixe de paquets de redondance $n - k$, et la dimension maximale du code k.

Du point de vue de l'émetteur, absolument rien ne change, à l'exception du fait qu'il n'ajoute pas de Checksum dans les paquets qu'il transmet.

Du point de vue du récepteur, les calculs sont aussi exactement les mêmes à la réception de chaque paquet. La différence se fait alors lors de l'algorithme de décodage. En effet, le récepteur ne sait pas si le paquet qu'il reçoit est endommagé ou non, et ne peut plus utiliser un système de compteurs comme pour la version de l'algorithme adaptée aux effacements. Ainsi, dans cette version de l'algorithme, le récepteur va avoir à parcourir le tableau contenant les valeurs qu'il a Xorées au fur et à mesure de la réception des paquets.

Intuitivement, si le tableau contient des 0 dans toutes les cases, cela signifie qu'aucun paquet n'a subi d'erreurs (les valeurs de tous les paquets se compensent, et chaque case du tableau vaut alors bien 0).

Supposons maintenant qu'un unique paquet ait subi une erreur : toutes les cases du tableau vaudront 0, à l'exception des cases correspondant à ce paquet, qui elles vaudront exactement l'erreur qu'a subie ce paquet. Le récepteur doit alors trouver ces cases, et en fonction de leur position dans le tableau, pourra déterminer dans certaines conditions l'indice du paquet erroné, puis le corriger.

La figure ci-dessous présente un exemple de tableau obtenu par le récepteur à la fin de la réception.

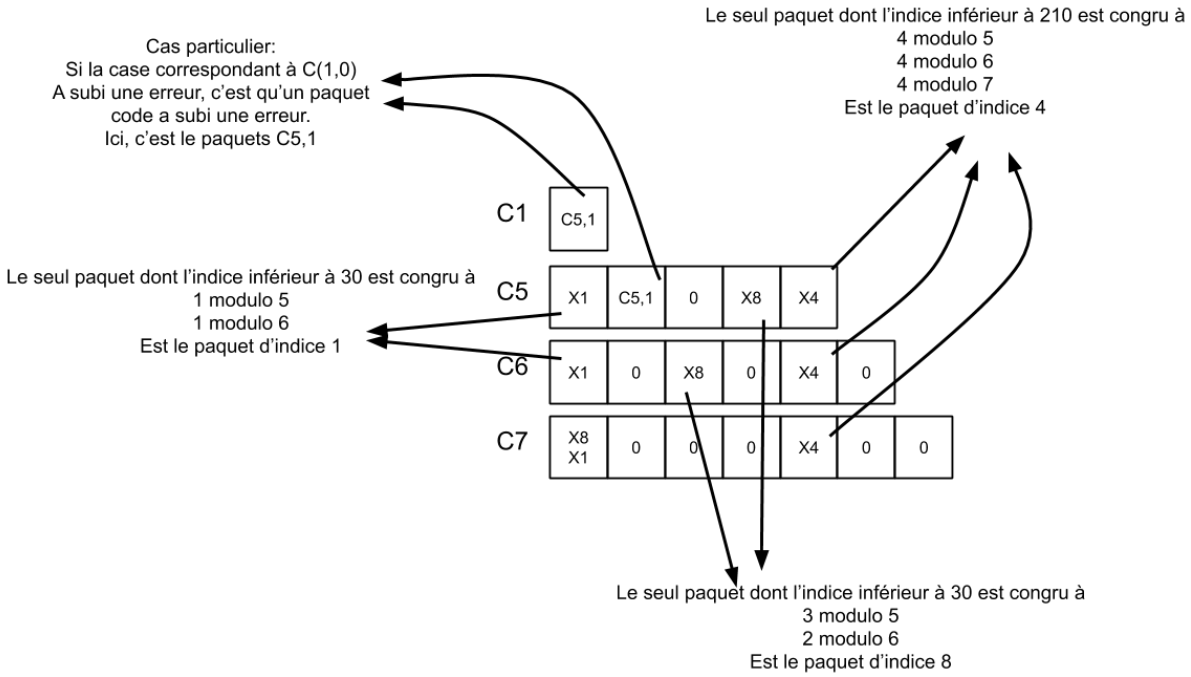


FIGURE 2.29 – Dans cet exemple, les cases correspondant aux paquets $C_{5,3}$ et $C_{6,2}$ contiennent la même erreur, et le théorème des restes Chinois nous indique que cette erreur est arrivée sur le paquet X_8 (de même pour les paquets X_1 et X_4.

L'ensemble P vaut ici $P = [5, 6, 7]$, et $k = 30$. Dans cet exemple, les paquets X_1, X_4, X_8 et $C_{5,1}$ ont chacun subi une erreur. Dans ce tableau, certaines cases valent 0, et d'autres ont une valeur non-nulle. Le récepteur va chercher à trouver des cases appartenant à différentes lignes du tableau ayant la même valeur, et appliquera alors le théorème des restes Chinois pour obtenir l'indice du paquet ayant subi cette erreur. Dans notre exemple, le récepteur va observer que les cases $C_{5,3}$ et $C_{6,2}$ ont subi la même erreur, et en déduire que c'est le paquet X_8 qui a subi cette erreur : 8 est le seul nombre entier compris entre 0 et 29 inclus qui soit congrus à 3 modulo 5 et 2 modulo 6. Il n'a alors plus qu'à modifier la valeur du paquet X_8 en fonction de cette erreur, en Xorant l'erreur lui correspondant dans ce dernier et dans toutes les cases du tableau correspondant. Il Xorera donc cette erreur dans les cases $C_{5,3}$, $C_{6,2}$ et $C_{7,1}$. On remarque d'ailleurs que pour le moment, la case $C_{7,1}$ contient une erreur qui est en réalité le Xor des erreurs correspondant à X_1 et X_8 : en Xorant l'erreur correspondant à X_8 dans la case $C_{7,1}$, il va alors apparaître l'erreur correspondant au paquet X_1. On voit donc ici apparaître le fonctionnement d'un décodage itératif, reposant sur la détection et la correction d'erreurs de paquets, que l'on va traiter de la même manière que si on les avait reçus, en espérant pouvoir en détecter et corriger d'autres à la prochaine itération du décodage.

La case du tableau correspondant à $C_{1,0}$ est un cas particulier : elle permet de déterminer si des paquets codes ont subi une erreur. En effet, dans notre exemple, le paquet $C_{5,1}$ a subi une erreur, erreur que l'on retrouve donc dans notre tableau dans les cases correspondant à $C_{5,1}$ et $C_{1,0}$.

2.5.2 Limitations de cette méthode

On peut observer un certain nombre de limitations de cette méthode, qui font que concrètement, elle peut être difficile à mettre en place, voire dangereuse. Tout d'abord, il faut choisir une dimension k de notre code assez petite pour qu'on puisse déterminer l'indice d'un paquet avec peu de cases du tableau. En effet, supposons qu'on trouve la même erreur dans deux cases différentes du tableau, mais que k soit supérieur au produit des P_i et P_j correspondant à ces cases : on ne peut pas alors déterminer l'indice du paquet ayant subi cette erreur, puisque plusieurs paquets d'indices différents correspondent à ces deux cases. Ainsi, la solution est alors de diminuer la dimension du code, ou d'utiliser un tableau P ayant un plus grand cardinal pour éviter de tomber dans cette situation.

De plus, un problème peut survenir quand, par exemple, deux paquets ont subi la même erreur. En effet, il est alors possible que les paquets aient été Xoré dans une même case du tableau, et que par conséquent cette case vaille 0, ce qui sera interprété comme une case valide. Avec cette méthode, on peut donc aboutir à des faux-positifs. On remarquera toutefois que c'est aussi le cas lorsqu'on utilise le Checksum d'un paquet UDP, qui peut avoir subi des erreurs, tout en ayant le même Checksum à l'origine. Enfin, la capacité de détection et de correction de ce code est nécessairement moins bonne que dans sa version uniquement correctrice, puisque dans cette dernière, un paquet est décodable si un nœud contrainte le permet, alors qu'ici, il faut plusieurs (au moins 2) nœuds contraintes disponibles pour déterminer puis réparer un paquet erroné : sachant qu'il faut au moins de nœuds contraintes pour détecter et réparer un paquet, sa capacité de détection et de correction est globalement au moins deux fois moins bonne que pour la version uniquement correction. Dans le cas d'utilisation de l'Elips-SD, nous n'aurons pas recours à ce code détecteur-correcteur, car nous nous imposons d'utiliser des paquets UDP qui disposent d'un Checksum. De plus, elle est moins performante en terme de capacité de correction, et l'algorithme de décodage a une complexité plus importante, alors que le calcul d'un Checksum se fait très rapidement. Néanmoins, la détection des paquets erronés repose sur une utilisation élégante du théorème des restes Chinois, et nous désirions présenter cette méthode en tant que "Proof of Concept".

Chapitre 3

Optimisation, graphes et calcul modulaire

Nous avons jusqu'ici présenté le fonctionnement de notre code correcteur, dont les combinaisons linéaires permettant d'obtenir les paquets de redondance sont dictées par l'ensemble P, un ensemble de nombres premiers entre eux. Nous avons présenté des simulations permettant de se figurer l'impact de P sur la complexité, la capacité de correction, la consommation mémoire des algorithmes d'encodage et de décodage. Cependant, le choix des nombres premiers entre eux constituant P s'est jusqu'ici fait de façon relativement subjective. Nous présenterons donc dans cette partie une manière automatique de choisir de bons candidats pour P. La méthode que nous proposons prend en entrée le nombre de paquets de redondance du code $n-k+1$, et retourne l'ensemble P dont la somme vaut $n-k$, et le $PPCM$ est maximisé (ce qui nous donne alors un grand degré de liberté sur le nombre de paquets sources par bloc, et une bonne capacité de correction). Cette méthode repose avant tout sur la fonction de Landau, qui pour $n \in \mathcal{N}$ nous retourne l'ensemble P, tel que $\Sigma(P) = n$ et $PPCM(P)$ est maximal. A titre d'exemple, $Landau(49) = [4, 5, 7, 9, 11, 13], PPCM([4, 5, 7, 9, 11, 13]) = 180180$: il n'existe pas d'ensemble dont la somme vaille 49 et dont le $PPCM$ soit supérieur à 180180. Rappelons que le nombre de paquets sources k que l'on peut encoder pour P fixé doit être inférieur ou égal au $PPCM$ de P, ainsi, en prenant $P = [4, 5, 7, 9, 11, 13]$, on obtient $n - k = \Sigma(P) + 1$ paquets de redondance à partir de $0 < k \leq 180180$ paquets sources. Pour calculer rapidement ces ensembles optimaux P pour de grandes valeurs de $n - k$, nous utilisons un algorithme proposé par Marc Deléglise, Jean-Louis Nicolas et Paul Zimmerman proposent en 2008 dans [17]. Il permet de calculer cette fonction pour $n - k$ allant jusqu'à plusieurs millions de milliards.
Toujours en rapport avec le calcul modulaire et le fait que nos combinaisons linéaires sont dictées par un ensemble de nombres premiers entre eux, nous présenterons une version modifiée de notre code correcteur, fonctionnant sur un canal à erreurs. En effet, jusqu'à présent, nous n'avons fait que considérer le cas du canal à effacement, dans lequel le récepteur sait quel paquet est erroné grâce au Checksum contenu dans chaque paquet (paquet qu'il considère alors comme effacé). Cette version modifiée de notre code correcteur repose exactement sur les mêmes combinaisons linéaires que présentées jusqu'à présent : l'émetteur effectue exactement les mêmes calculs. C'est le cas aussi pour le récepteur à la réception de chaque paquet. La différence se fait au niveau de l'algorithme de décodage, qui utilise habilement le Théorème des Restes Chinois pour déterminer quel paquet est erroné, puis le réparer.

Nous avons aussi vu qu'il était possible de représenter le décodage itératif d'un code correcteur linéaire par un graphe, son graphe de Tanner : le décodage d'un tel code aboutit sur un

succès lorsque son graphe de Tanner ne contient pas de cycle, et sur un échec le cas échéant. Dans le cas ou il existe un cycle dans le graphe de Tanner d'un code correcteur, on parle alors de Stopping-set, qui peuvent être de différentes tailles. Il est alors intéressant, dans le but d'étudier la probabilité de décoder totalement un bloc de paquets, de connaître le nombre de Stopping-set de différentes tailles que l'on peut obtenir dans un graphe de Tanner. En effet, supposons que, après transmission, e paquets aient été effacés : quelle est la probabilité que parmi ces e paquets, il en existe un sous-ensemble qui forme un Stopping-set dans le graphe de Tanner ? Dans un premier temps, nous présenterons une manière de représenter le graphe de Tanner de notre code d'une manière plus claire et lisible, pour un paramétrage particulier de notre code (le cas ou le paramètre du code $P = [P_0, P_1]$ ne possède que 2 éléments). En effet, nous verrons qu'il est possible de représenter notre système d'équations par un graphe de Rook de dimensions $(P_0 + 1, P_1 + 1)$. Nous étudierons la probabilité d'apparition d'un Stopping-set dans un graphe de Tanner de différentes manières : pour un petit nombre d'effacements et un paramétrage particulier de notre code correcteur, nous présentons un algorithme, proposé par Rebecca Stones dans [56], exponentiel en mémoire et permettant de compter le nombre de sous-graphes de taille fixée formant un ou plusieurs Stopping-set dans le graphe de Tanner. Nous proposons de plus un certain nombre de formules closes permettant de calculer, pour un petit nombre d'effacements mais un paramétrage de notre code quelconque, obtenus en interpolant les résultats retournés par l'algorithme de Stones. Pour un plus grand nombre d'effacements, nous proposerons une borne inférieure et supérieure pour cette probabilité. Dans un second temps, nous présenterons une version modifiée de notre protocole de transmission, qui utilisera en plus de notre code correcteur un système d'acquittements et de non-acquittements, tel que le font les protocoles de type HARQ (pour Hybrid automatic repeat request). Ce type de protocoles repose, comme le protocole TCP par exemple, sur le fait que le récepteur peut demander à l'émetteur la retransmission de paquets. Ainsi, on peut se demander l'intérêt d'utiliser un code correcteur en plus de ce système d'acquittements et de non-acquittements : cela s'avère très utile dans le cas de transmission ayant beaucoup de "ping", et un certain nombre de pertes, et permet de réduire le temps total de transmission, en réduisant le nombre d'itérations entre l'émetteur et le récepteur.

3.1 PPCM maximal d'un ensemble dont la somme est fixée

Comme on l'a vu précédemment, le code correcteur que nous présentons nécessite que l'émetteur et le récepteur se soient mis d'accord sur un ensemble de nombres premiers entre eux P, dont on va déduire sa matrice génératrice et les combinaisons linéaires produisant les paquets de redondance, ainsi que du paramètre k correspondant à la taille des mots (ou blocs) que l'on va encoder. Le cardinal de P nous donne le degré par colonne de la matrice, d_c, et P et k nous donnent le degré par ligne d_l : d_c est fixé, et vaut $|P| + 2$, alors que d_l peut varier, en fonction des valeurs de P et de k. Ainsi, pour $P = [3, 4, 5]$ et $k = PPCM(3, 4, 5) = 3 * 4 * 5 = 60$, on aura $d_c = |P| + 2 = 3 + 2 = 5$, ce qui signifie que pour chaque paquet source X_i, on va envoyer 5 paquets susceptibles de permettre de retrouver X_i lors du décodage : le paquet X_i lui-même, le paquet code $C_{1,0}$ (qui est le résultat du Xor de tous les paquets sources), et les paquets codes $C_{3,i \mod 3}$, $C_{4,i \mod 4}$ et $C_{5,i \mod 5}$ dans lesquels on a Xoré X_i.

Dans la partie précédente, les simulations effectuées sur différents paramétrages de notre

code nous ont faits nous rendre compte que le cardinal de P avait une influence sur la capacité de correction ainsi que les temps de calcul de l'encodage et du décodage (visible sur la figure 2.27 par exemple : pour $P = [49, 51]$, $P = [31, 34, 35]$, $P = [21, 23, 25, 31]$ ou encore $P = [17, 19, 20, 21, 23]$, dont les sommes valent toutes $\Sigma(P) = 101$, et $PPCM(P) >= 101$, on peut choisir $k = 101$ et $n - k = 101$, et obtenir des codes de mêmes dimensions et longueurs, et les comparer du point de vue de leur efficacité. On remarque bien que la capacité de correction du code obtenu avec $P = [17, 19, 20, 21, 23]$ est meilleure que pour $P = [49, 51]$.

Nous proposons ici une méthode permettant de déterminer un bon ensemble P en fonction de la dimension k ou de longueur n souhaitée, utilisant la fonction de Landau. Supposons que l'on souhaite utiliser un code de dimension disposant de $n - k = 11$ paquets de redondance par bloc. Il nous faut trouver un ensemble $P = [P_0, P_1, ...]$ dont la somme vaut 10 et le plus petit commun multiple sera le plus grand possible. En l'occurrence, l'ensemble P dont la somme vaut 10 et dont le $PPCM$ est maximal est $P = [2, 3, 5]$, avec un cardinal de 3 et un $PPCM$ égal à $2 * 3 * 5 = 30$. Ainsi, pour $n - k = 11$ fixé, on ne pourra pas produire de codes ayant une dimension k supérieure à 30. La fonction de Landau que nous présentons plus en détail par la suite, souvent notée g, prend en argument un nombre entier positif x, et retourne le plus grand nombre entier $y = g(x)$ tel que $y = PPCM(P)$ et $x = \Sigma(P)$. On peut donc se servir de cette fonction pour déterminer la longueur maximale d'un code dont on aura fixé le nombre de paquets de redondance $n - k$.

Cette fonction progresse asymptotiquement comme $\sqrt{x \ln x}$, ce qui nous permet d'obtenir des codes à très forts rendements $R = k/n$ dont la dimension k est très proche de sa longueur n, ce qui est intéressant dans le cas d'une transmission avec un très faible taux d'effacements.

Dans cette partie, nous allons donc présenter la fonction de Landau, ainsi qu'un algorithme proposé par Marc Deléglise, Jean-Louis Nicolas, Paul Zimmermann en 2008 dans [17] permettant de calculer cette dernière pour de très grandes valeurs. Nous verrons de plus qu'il n'est pas toujours possible d'utiliser cette fonction pour paramétrer convenablement notre code, mais qu'elle nous donne au moins le rendement maximal de notre code pour une dimension fixée.

3.1.1 Fonction de Landau

Dans cette partie, nous présentons donc la fonction de Landau, dont nous nous servons pour obtenir la longueur maximum n de notre code pour une dimension k fixée.

Cette fonction permet donc de calculer le plus grand $PPCM$ d'un ensemble de nombres entiers P dont la somme vaut x, ce qui revient à calculer le plus grand $PPCM$ de toute partition de x. De manière équivalente, elle permet de calculer le plus grand ordre d'un élément d'un groupe symétrique S_n. Elle tient son nom de Edmund Landau, qui démontra qu'elle tend asymptotiquement vers $e^{\sqrt{x \ln x)}}$:

$$\lim_{x \longrightarrow \infty} \frac{\ln g(x)}{\sqrt{x \ln x}} = 1 \tag{3.1}$$

La figure ci-dessous représente le logarithme népérien de la fonction de Landau et la fonction $f(x) = \sqrt{x * \ln x}$ pour $1 \leq x \leq 1000$, et permet d'observer qu'elles sont quasiment superposées.

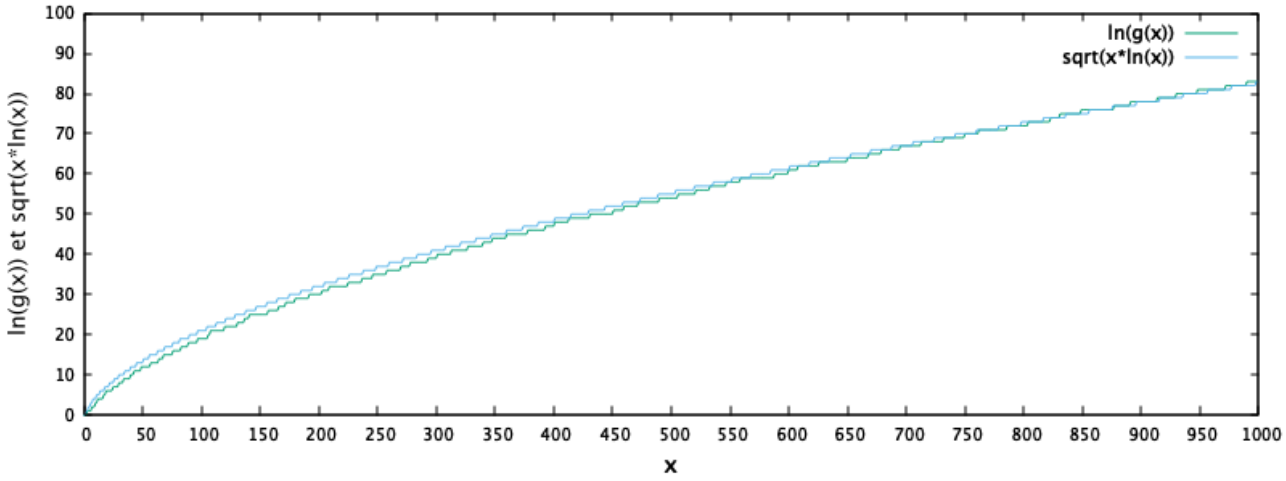


FIGURE 3.1 – La fonction de Landau tend asymptotiquement vers $e^{\sqrt{(x \ln x)}}$

Afin de se familiariser avec cette fonction, présentons tout d'abord certaines de ses valeurs pour différents ordres de grandeur. A titre d'exemples,

- $g(7) = 12$, avec $P = [3, 4]$,
- $g(8) = 15$, avec $P = [3, 5]$,
- $g(9) = 20$, avec $P = [4, 5]$,
- $g(10) = 30$, avec $P = [2, 3, 5]$,
- $g(120) = 5.35422888 * 10^9$, avec $P = [5, 7, 9, 11, 13, 16, 17, 19, 23]$,
- $g(299) = 1.79967741245412128 * 10^{17}$, avec $P = [5, 7, 8, 9, 11, 13, 17, 19, 23, 29, 31, 37, 43, 47]$.

Le tableau 3.1 ci-dessous présente les valeurs de g et les ensembles P associés pour $0 \leq k \leq 75$. On remarque que cette fonction est évidemment croissante, mais pas strictement, et qu'il existe des valeurs $g(x) = g(y), x \neq y$, comme par exemple $g(44) = g(45) = g(46) = 60060$. Nous verrons par la suite que ceci peut poser problème, et que nous ne pourrons pas utiliser les valeurs retournées pour $g(45)$ et $g(46)$, car les tableaux correspondant à ces valeurs contiennent des 1. En pratique, il est possible d'utiliser ces valeurs, mais le 1 contenu dans les ensembles P obtenus ne produira qu'un seul paquet de redondance, pour un grand nombre de Xors (k Xors au total, calculs qui peuvent être effectués plus rapidement lors de nos algorithmes, en Xorant non pas k paquets sources entre eux, mais par exemple les P_0 paquets codes correspondant à P_0). Nous observons aussi que plus x est grand, plus nous avons de chance d'obtenir un ensemble contenant un ou plusieurs 1. Ainsi, pour un x fixé (rappelons que $x = n - k - 1$ représente le nombre de paquets de redondance par bloc), nous choisirons en réalité le premier nombre $y \geq x$ tel que l'ensemble P correspondant ne contient pas de 1.

x	g(x)	P	x	g(x)	P	x	g(x)	P
1	1	[1]	26	1260	[1, 4, 5, 7, 9]	51	180180	[1, 1, 4, 5, 7, 9, 11, 13]
2	2	[2]	27	1540	[4, 5, 7, 11]	52	180180	[1, 1, 1, 4, 5, 7, 9, 11, 13]
3	3	[3]	28	2310	[2, 3, 5, 7, 11]	53	360360	[5, 7, 8, 9, 11, 13]
4	4	[4]	29	2520	[5, 7, 8, 9]	54	360360	[1, 5, 7, 8, 9, 11, 13]
5	6	[2, 3]	30	4620	[3, 4, 5, 7, 11]	55	360360	[1, 1, 5, 7, 8, 9, 11, 13]
6	6	[1, 2, 3]	31	4620	[1, 3, 4, 5, 7, 11]	56	360360	[1, 1, 1, 5, 7, 8, 9, 11, 13]
7	12	[3, 4]	32	5460	[3, 4, 5, 7, 13]	57	471240	[5, 7, 8, 9, 11, 17]
8	15	[3, 5]	33	5460	[1, 3, 4, 5, 7, 13]	58	510510	[2, 3, 5, 7, 11, 13, 17]
9	20	[4, 5]	34	9240	[3, 5, 7, 8, 11]	59	556920	[5, 7, 8, 9, 13, 17]
10	30	[2, 3, 5]	35	9240	[1, 3, 5, 7, 8, 11]	60	1021020	[3, 4, 5, 7, 11, 13, 17]
11	30	[1, 2, 3, 5]	36	13860	[4, 5, 7, 9, 11]	61	1021020	[1, 3, 4, 5, 7, 11, 13, 17]
12	60	[3, 4, 5]	37	13860	[1, 4, 5, 7, 9, 11]	62	1141140	[3, 4, 5, 7, 11, 13, 19]
13	60	[1, 3, 4, 5]	38	16380	[4, 5, 7, 9, 13]	63	1141140	[1, 3, 4, 5, 7, 11, 13, 19]
14	84	[3, 4, 7]	39	16380	[1, 4, 5, 7, 9, 13]	64	2042040	[3, 5, 7, 8, 11, 13, 17]
15	105	[3, 5, 7]	40	27720	[5, 7, 8, 9, 11]	65	2042040	[1, 3, 5, 7, 8, 11, 13, 17]
16	140	[4, 5, 7]	41	30030	[2, 3, 5, 7, 11, 13]	66	3063060	[4, 5, 7, 9, 11, 13, 17]
17	210	[2, 3, 5, 7]	42	32760	[5, 7, 8, 9, 13]	67	3063060	[1, 4, 5, 7, 9, 11, 13, 17]
18	210	[1, 2, 3, 5, 7]	43	60060	[3, 4, 5, 7, 11, 13]	68	3423420	[4, 5, 7, 9, 11, 13, 19]
19	420	[3, 4, 5, 7]	44	60060	[1, 3, 4, 5, 7, 11, 13]	69	3423420	[1, 4, 5, 7, 9, 11, 13, 19]
20	420	[1, 3, 4, 5, 7]	45	60060	[1, 1, 3, 4, 5, 7, 11, 13]	70	6126120	[5, 7, 8, 9, 11, 13, 17]
21	420	[1, 1, 3, 4, 5, 7]	46	60060	[1, 1, 1, 3, 4, 5, 7, 11, 13]	71	6126120	[1, 5, 7, 8, 9, 11, 13, 17]
22	420	[1, 1, 1, 3, 4, 5, 7]	47	120120	[3, 5, 7, 8, 11, 13]	72	6846840	[5, 7, 8, 9, 11, 13, 19]
23	840	[3, 5, 7, 8]	48	120120	[1, 3, 5, 7, 8, 11, 13]	73	6846840	[1, 5, 7, 8, 9, 11, 13, 19]
24	840	[1, 3, 5, 7, 8]	49	180180	[4, 5, 7, 9, 11, 13]	74	6846840	[1, 1, 5, 7, 8, 9, 11, 13, 19]
25	1260	[4, 5, 7, 9]	50	180180	[1, 4, 5, 7, 9, 11, 13]	75	6846840	[1, 1, 1, 5, 7, 8, 9, 11, 13, 19]

TABLE 3.1 – Valeurs de g(x) et l'ensemble P associé pour $1 \leq x \leq 75$

A titre indicatif, $g(1000) =$
33243053827815621110079681643591870945206608017128168697894154852872668087678431117
38720162875054843603263700192999897717625028578801443857666338780583560014730857850
36452761475533959916212260000637813162472420368873512757561006927322557340130752050
50344641658984673177374409382084445412293731171272130479701451356488716711192824510
69025376877101365308799106711467776702204839016567186747724796678518384096663404717
39960278666666634210169107891641005373821214992686268222379440509273649600 0.
Le tableau 3.2 représente les valeurs de $g(n)$ pour $1000 \leq n \leq 160000$, avec un pas de 1000, en notation scientifique.

x	g(x)	x	g(x)	x	g(x)	x	g(x)
1000	$1,516 * 10^{36}$	41000	$2,125 * 10^{122}$	81000	$3,232 * 10^{133}$	121000	$3,490 * 10^{141}$
2000	$5,301 * 10^{54}$	42000	$7,438 * 10^{122}$	82000	$6,141 * 10^{133}$	122000	$3,555 * 10^{141}$
3000	$8,089 * 10^{68}$	43000	$1,488 * 10^{123}$	83000	$1,045 * 10^{134}$	123000	$6,631 * 10^{141}$
4000	$2,230 * 10^{81}$	44000	$2,294 * 10^{123}$	84000	$1,568 * 10^{134}$	124000	$1,105 * 10^{142}$
5000	$1,221 * 10^{92}$	45000	$4,589 * 10^{123}$	85000	$3,135 * 10^{134}$	125000	$1,689 * 10^{142}$
6000	$5,126 * 10^{96}$	46000	$1,116 * 10^{124}$	86000	$7,838 * 10^{134}$	126000	$3,377 * 10^{142}$
7000	$4,952 * 10^{99}$	47000	$2,715 * 10^{124}$	87000	$1,568 * 10^{135}$	127000	$3,508 * 10^{142}$
8000	$1,535 * 10^{101}$	48000	$5,430 * 10^{124}$	88000	$1,632 * 10^{135}$	128000	$6,754 * 10^{142}$
9000	$8,904 * 10^{102}$	49000	$1,629 * 10^{125}$	89000	$3,135 * 10^{135}$	129000	$7,017 * 10^{142}$
10000	$1,336 * 10^{104}$	50000	$3,258 * 10^{125}$	90000	$3,264 * 10^{135}$	130000	$9,946 * 10^{142}$
11000	$1,647 * 10^{105}$	51000	$5,430 * 10^{125}$	91000	$5,957 * 10^{135}$	131000	$1,902 * 10^{143}$
12000	$1,351 * 10^{106}$	52000	$9,231 * 10^{125}$	92000	$9,928 * 10^{135}$	132000	$2,009 * 10^{143}$
13000	$2,026 * 10^{107}$	53000	$2,145 * 10^{126}$	93000	$1,489 * 10^{136}$	133000	$3,804 * 10^{143}$
14000	$1,114 * 10^{108}$	54000	$4,290 * 10^{126}$	94000	$2,978 * 10^{136}$	134000	$3,943 * 10^{143}$
15000	$8,712 * 10^{108}$	55000	$1,287 * 10^{127}$	95000	$3,229 * 10^{136}$	135000	$7,227 * 10^{143}$
16000	$9,583 * 10^{109}$	56000	$2,574 * 10^{127}$	96000	$5,957 * 10^{136}$	136000	$1,205 * 10^{144}$
17000	$4,095 * 10^{110}$	57000	$4,290 * 10^{127}$	97000	$8,073 * 10^{136}$	137000	$1,807 * 10^{144}$
18000	$2,252 * 10^{111}$	58000	$6,760 * 10^{127}$	98000	$1,615 * 10^{137}$	138000	$3,614 * 10^{144}$
19000	$9,008 * 10^{111}$	59000	$1,287 * 10^{128}$	99000	$1,769 * 10^{137}$	139000	$3,816 * 10^{144}$
20000	$2,928 * 10^{112}$	60000	$2,188 * 10^{128}$	100000	$3,229 * 10^{137}$	140000	$7,227 * 10^{144}$
21000	$2,387 * 10^{113}$	61000	$4,375 * 10^{128}$	101000	$3,539 * 10^{137}$	141000	$7,633 * 10^{144}$
22000	$4,774 * 10^{113}$	62000	$1,094 * 10^{129}$	102000	$7,520 * 10^{137}$	142000	$1,084 * 10^{145}$
23000	$3,103 * 10^{114}$	63000	$2,188 * 10^{129}$	103000	$1,534 * 10^{138}$	143000	$1,145 * 10^{145}$
24000	$6,207 * 10^{114}$	64000	$3,560 * 10^{129}$	104000	$3,068 * 10^{138}$	144000	$2,168 * 10^{145}$
25000	$2,817 * 10^{115}$	65000	$6,053 * 10^{129}$	105000	$3,194 * 10^{138}$	145000	$2,290 * 10^{145}$
26000	$6,103 * 10^{115}$	66000	$1,211 * 10^{130}$	106000	$6,135 * 10^{138}$	146000	$4,298 * 10^{145}$
27000	$3,662 * 10^{116}$	67000	$1,816 * 10^{130}$	107000	$8,154 * 10^{138}$	147000	$4,298 * 10^{145}$
28000	$8,591 * 10^{116}$	68000	$3,632 * 10^{130}$	108000	$1,631 * 10^{139}$	148000	$8,167 * 10^{145}$
29000	$2,563 * 10^{117}$	69000	$9,079 * 10^{130}$	109000	$1,728 * 10^{139}$	149000	$2,042 * 10^{146}$
30000	$1,117 * 10^{118}$	70000	$1,816 * 10^{131}$	110000	$3,261 * 10^{139}$	150000	$4,083 * 10^{146}$
31000	$2,234 * 10^{118}$	71000	$1,947 * 10^{131}$	111000	$3,455 * 10^{139}$	151000	$4,083 * 10^{146}$
32000	$6,701 * 10^{118}$	72000	$4,753 * 10^{131}$	112000	$6,197 * 10^{139}$	152000	$8,167 * 10^{146}$
33000	$1,564 * 10^{119}$	73000	$9,506 * 10^{131}$	113000	$1,033 * 10^{140}$	153000	$8,167 * 10^{146}$
34000	$3,127 * 10^{119}$	74000	$1,077 * 10^{132}$	114000	$1,609 * 10^{140}$	154000	$8,167 * 10^{146}$
35000	$1,497 * 10^{120}$	75000	$1,616 * 10^{132}$	115000	$3,219 * 10^{140}$	155000	$1,225 * 10^{147}$
36000	$2,993 * 10^{120}$	76000	$3,232 * 10^{132}$	116000	$3,344 * 10^{140}$	156000	$1,633 * 10^{147}$
37000	$1,048 * 10^{121}$	77000	$8,080 * 10^{132}$	117000	$6,437 * 10^{140}$	157000	$2,450 * 10^{147}$
38000	$1,571 * 10^{121}$	78000	$1,616 * 10^{133}$	118000	$8,724 * 10^{140}$	158000	$2,450 * 10^{147}$
39000	$3,330 * 10^{121}$	79000	$1,616 * 10^{133}$	119000	$1,745 * 10^{141}$	159000	$2,450 * 10^{147}$
40000	$1,063 * 10^{122}$	80000	$3,232 * 10^{133}$	120000	$1,777 * 10^{141}$	160000	$3,131 * 10^{147}$

TABLE 3.2 – Valeurs (arrondies pour faciliter la lecture) de $g(x)$ et l'ensemble P associé pour $1000 \leq x \leq 160000$ et x multiple de 1000

Algorithme calculant la fonction de Landau

Dans cette partie nous présentons deux algorithmes permettant de calculer la fonction de Landau. Marc Deléglise, Jean-Louis Nicolas et Paul Zimmerman proposent en 2008 dans [17] un algorithme permettant de calculer la fonction de Landau pour les nombres entiers $0 \leq n \leq N$ pour N de l'ordre de 10^6. Tout d'abord, nous présentons un algorithme simple permettant de calculer $g(n)$. Cet algorithme a une complexité en temps théorique de $\mathcal{O}(N^{3/2}/\sqrt{\log N})$ et une consommation mémoire de $\mathcal{O}(N)$ entiers de taille $\exp(\mathcal{O}(\sqrt{N * \log N}))$. En 2008, lors de la parution de l'article [17], calculer $g(10^5)$ prenait $13heures$ et consommait $337mo$ sur un Pentium 4 cadencé à $3Ghz$.

Algorithm 4 Algorithme calculant la fonction de Landau $g(n)$ pour $0 \leq n \leq N$.

Require: Cet algorithme prend en entrée le nombre entier positif N, et utilise les variables locales a, k, n, p, p_{max} (nombres entiers) et g un tableau de $N+1$ nombres entiers qui contiendra les valeurs de $g(n)$ pour $0 \leq n \leq N$

```
1 : function LANDAU(N)
2 :     for n ← 0 to N do
3 :         g[n] = 0                ▷ On initialise le tableau g qui contiendra les valeurs de Landau
4 :     p_max = ⌊1.328 * sqrt(N * log(N))⌋
5 :     p = 2
6 :     while p ≤ p_max do
7 :         for n ← N to p do
8 :             k = 1
9 :             while p^k ≤ n do
10 :                a = p^k * g[n − p^k]
11 :                if g[n] < a then
12 :                    g[n] = a
13 :                k = k + 1
14 :        p = nextPrime(p)        ▷ Fonction qui retourne le prochain nombre premier
        return g
```

Nous avons vu précédemment que la fonction de Landau nous est utile pour paramétrer notre code correcteur. Cependant, connaître $g(n)$ ne nous suffit pas : il nous faut disposer de l'ensemble de nombres entiers qui correspond à cette valeur. Une méthode très simple permettant d'obtenir cet ensemble en fonction de la valeur retournée par cet algorithme $g(n)$ est tout simplement de calculer sa décomposition en facteurs premiers. En effet, celle-ci est unique, et il existe donc un unique ensemble pour chaque valeur de $g(n)$. Une fois la décomposition en facteurs premiers de $g(n)$, on leur ajoutera le nombre de 1 adéquat afin que la somme de cet ensemble vaille bien n.

Utilisations et limitations de la fonction de Landau pour le paramétrage de notre code

On a vu dans la partie précédente qu'on pouvait se servir de la fonction de Landau, calculable par l'algorithme proposé par Marc Deléglise, Jean-Louis Nicolas et Paul Zimmermann, pour paramétrer notre code et trouver un bon ensemble P en fonction de la dimension k souhaitée, ou du nombre de paquets de redondance $n-k$ souhaitée.

— En fixant le nombre de paquets de redondance $n-k$ de notre code, on peut calculer $g(n-k-1)$ qui nous retourne l'ensemble P correspondant, dont le $PPCM$ nous indiquera la dimension k maximale qu'on pourra obtenir pour n. Très concrètement, si on souhaite ajouter $n-k=30$ paquets de redondance pour encoder un bloc, on calculera $g(30-1) = g(29)$ (puisqu'on ajoute le paquet $C_{1,0}$ à chaque bloc), qui vaut $g(29) = 2520$ pour $P = [5, 7, 8, 9]$. On sait alors qu'un bloc pourra contenir au maximum $k = 2520$ paquets sources.

— A contrario, si on souhaite fixer la dimension k de notre code et obtenir l'ensemble P permettant de fiabiliser des blocs de k paquets sources et dont la somme est minimale, on calculera $g^{-1}(k)-1$. Ainsi, si on souhaite fiabiliser des blocs de $k = 10^5 = 100000$ paquets sources, on utilisera l'ensemble $P = [3, 5, 7, 8, 11, 13]$, dont la somme vaut $\Sigma P = 47$: il s'agit de l'ensemble P dont $PPCM(P) = 120120$ est supérieur ou égal à $k = 100000$ et dont la somme est minimale. On ajoutera donc $n - k = 47 + 1 = 48$ paquets de redondance à chaque bloc de $k = 100000$ paquets sources.

Cette méthode présente cependant un problème inhérent à la fonction de Landau : les ensembles P obtenus avec cet algorithme peuvent contenir des 1. C'est par exemple le cas pour $g(56)$, qui nous retourne $P = [1, 1, 1, 5, 7, 8, 9, 11, 13]$, ou $g(75)$ qui nous retourne $P = [1, 1, 1, 5, 7, 8, 9, 11, 13, 19]$.

Expliquons pourquoi obtenir des ensembles P contenant des 1 est un problème :

— Soit P_i un élément de l'ensemble P. On a vu que P_i nous permet de calculer P_i paquets de redondance, et que le temps de calcul de ces P_i paquets ne dépend pas de P_i mais uniquement de k (et de la taille des paquets). Ainsi, que P_i vaille 2, 100 ou 1000, le temps de calcul des 2, 100 ou 1000 paquets de redondance correspondant sera le même (chaque paquet source étant Xoré une fois dans un paquet de redondance pour chaque P_i). Si $P_i = 1$, on va alors Xorer les k paquets sources pour obtenir un unique paquet de redondance $C_{1,0}$.

— Quel que soit l'ensemble P choisi pour paramétrer notre code, nous calculons toujours un paquet $C_{1,0}$, en effectuant beaucoup moins que k Xors : nous Xorons les P_i paquets codes correspondant à P_i, $C_{P_i,0}$, $C_{P_i,1}$, $C_{P_i,2}$, ..., C_{P_i,P_i-1}. Il est donc dommage de dépenser du temps de calcul pour calculer $C_{1,0}$.

— Pour l'ensemble $P = [1, 1, 1, 5, 7, 8, 9, 11, 13]$, notre algorithme d'encodage va calculer séparément les 3 paquets de redondance correspondant à $P_0 = 1$, $P_1 = 1$ et $P_2 = 1$, alors que leurs valeurs seront égales. Là-aussi, il est dommage de dépenser 3 fois plus de temps de calcul pour calculer $C_{1,0}$, alors qu'on pourrait ne le calculer qu'une seule fois, et de façon bien plus rapide.

Le tableau 3.3 présenté par la suite permet de visualiser en vert les ensembles P ne contenant aucun 1, et en rouge ceux contenant un 1 ou plus.

x	g(x)	P	x	g(x)	P	x	g(x)	P
1	1	1	26	1260	1, 4, 5, 7, 9	51	180180	1, 1, 4, 5, 7, 9, 11, 13
2	2	2	27	1540	4, 5, 7, 11	52	180180	1, 1, 1, 4, 5, 7, 9, 11, 13
3	3	3	28	2310	2, 3, 5, 7, 11	53	360360	5, 7, 8, 9, 11, 13
4	4	4	29	2520	5, 7, 8, 9	54	360360	1, 5, 7, 8, 9, 11, 13
5	6	2, 3	30	4620	3, 4, 5, 7, 11	55	360360	1, 1, 5, 7, 8, 9, 11, 13
6	6	1, 2, 3	31	4620	1, 3, 4, 5, 7, 11	56	360360	1, 1, 1, 5, 7, 8, 9, 11, 13
7	12	3, 4	32	5460	3, 4, 5, 7, 13	57	471240	5, 7, 8, 9, 11, 17
8	15	3, 5	33	5460	1, 3, 4, 5, 7, 13	58	510510	2, 3, 5, 7, 11, 13, 17
9	20	4, 5	34	9240	3, 5, 7, 8, 11	59	556920	5, 7, 8, 9, 13, 17
10	30	2, 3, 5	35	9240	1, 3, 5, 7, 8, 11	60	1021020	3, 4, 5, 7, 11, 13, 17
11	30	1, 2, 3, 5	36	13860	4, 5, 7, 9, 11	61	1021020	1, 3, 4, 5, 7, 11, 13, 17
12	60	3, 4, 5	37	13860	1, 4, 5, 7, 9, 11	62	1141140	3, 4, 5, 7, 11, 13, 19
13	60	1, 3, 4, 5	38	16380	4, 5, 7, 9, 13	63	1141140	1, 3, 4, 5, 7, 11, 13, 19
14	84	3, 4, 7	39	16380	1, 4, 5, 7, 9, 13	64	2042040	3, 5, 7, 8, 11, 13, 17
15	105	3, 5, 7	40	27720	5, 7, 8, 9, 11	65	2042040	1, 3, 5, 7, 8, 11, 13, 17
16	140	4, 5, 7	41	30030	2, 3, 5, 7, 11, 13	66	3063060	4, 5, 7, 9, 11, 13, 17
17	210	2, 3, 5, 7	42	32760	5, 7, 8, 9, 13	67	3063060	1, 4, 5, 7, 9, 11, 13, 17
18	210	1, 2, 3, 5, 7	43	60060	3, 4, 5, 7, 11, 13	68	3423420	4, 5, 7, 9, 11, 13, 19
19	420	3, 4, 5, 7	44	60060	1, 3, 4, 5, 7, 11, 13	69	3423420	1, 4, 5, 7, 9, 11, 13, 19
20	420	1, 3, 4, 5, 7	45	60060	1, 1, 3, 4, 5, 7, 11, 13	70	6126120	5, 7, 8, 9, 11, 13, 17
21	420	1, 1, 3, 4, 5, 7	46	60060	1, 1, 1, 3, 4, 5, 7, 11, 13	71	6126120	1, 5, 7, 8, 9, 11, 13, 17
22	420	1, 1, 1, 3, 4, 5, 7	47	120120	3, 5, 7, 8, 11, 13	72	6846840	5, 7, 8, 9, 11, 13, 19
23	840	3, 5, 7, 8	48	120120	1, 3, 5, 7, 8, 11, 13	73	6846840	1, 5, 7, 8, 9, 11, 13, 19
24	840	1, 3, 5, 7, 8	49	180180	4, 5, 7, 9, 11, 13	74	6846840	1, 1, 5, 7, 8, 9, 11, 13, 19
25	1260	4, 5, 7, 9	50	180180	1, 4, 5, 7, 9, 11, 13	75	6846840	1, 1, 1, 5, 7, 8, 9, 11, 13, 19

TABLE 3.3 – Valeurs de g(x) et l'ensemble P associé pour $1 \leq x \leq 75$. Les ensembles P contenant un ou plusieurs 1 sont en rouge, ceux ne contenant aucun 1 sont en vert. On cherchera à éviter les ensembles contenant des 1, car ils correspondent à de "mauvais" paramétrages de notre code.

3.1.2 Simulations de décodage avec la méthode de Landau

Nous présentons dans cette partie des simulations, afin d'observer la capacité de correction de notre code en fonction des différents P que nous obtenons grâce à notre méthode utilisant la fonction de Landau.

La figure 3.3 ci-dessous présente la probabilité d'échec du décodage en fonction de la probabilité d'effacements, pour différents ensembles P retournés par notre méthode utilisant la fonction de Landau. Pour toutes ces simulations, nous avons fixé $k = 1000$ paquets sources par bloc.

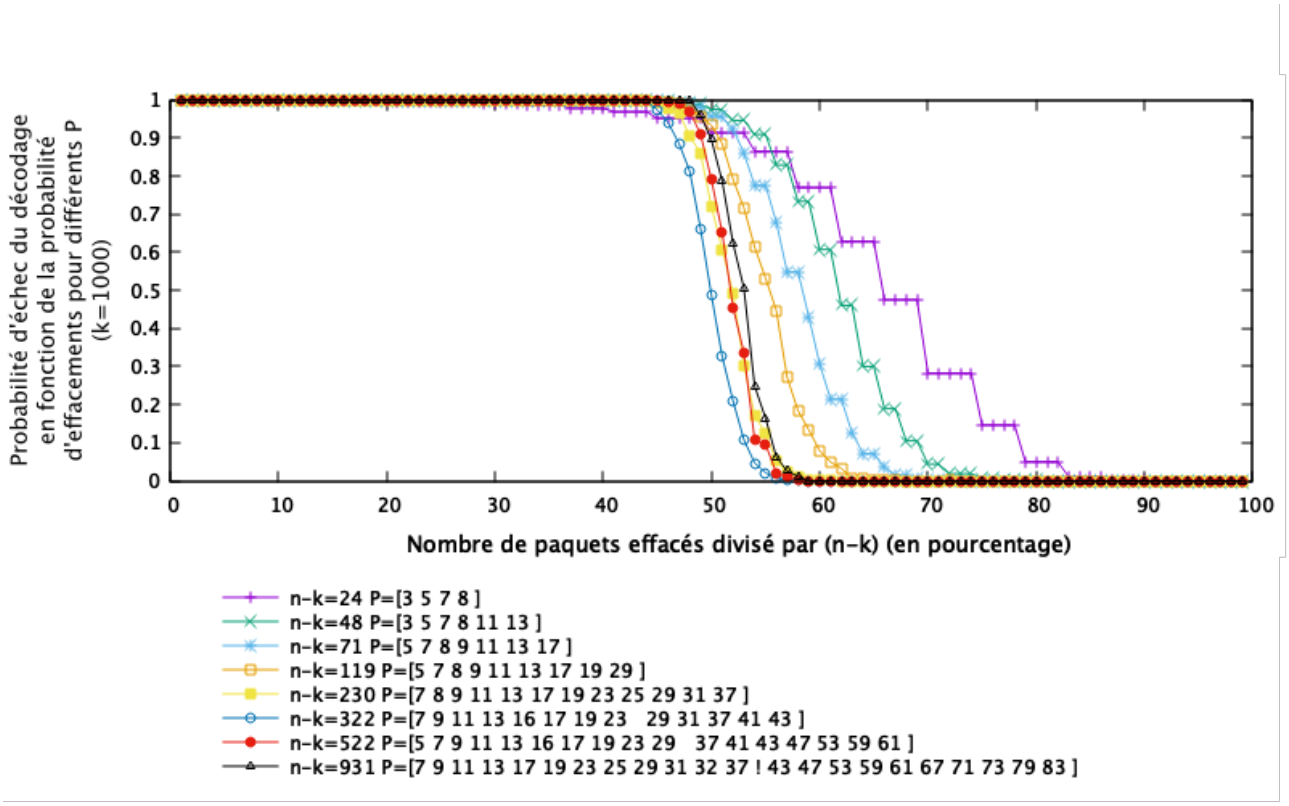


FIGURE 3.2 – Probabilité d'échec du décodage en fonction de la probabilité d'effacements sur le canal, pour différents ensembles P retournés par notre méthode utilisant la fonction de Landau. Ici, toutes les simulations ont été réalisézs avec $k = 1000$

On remarque notamment sur cette figure que plus le nombre de paquets de redondance $(n - k)$ pour lequel nous calculons P est grand, plus la courbe dévisse vite. Ceci peut sembler être un désavantage, mais en réalité ça n'en est pas un. Si on compare par exemple la courbe correspondant à $n - k = 24$ et la courbe correspondant à $n - k = 931$, on remarque que pour une probabilité d'effacements de 70%, la première présente une probabilité de décodage d'environ 50%, alors que la deuxième présente une probabilité de décodage nulle. Cependant, si on compare ces deux courbes pour une probabilité d'effacements de 50%, on remarque que la première présente une probabilité d'environ 90%, alors que la deuxième présente une probabilité de décodage de 100%. C'est donc en réalité la courbe correspondant à $n - k = 931$ qui est la plus performante, car elle nous garantit plus clairement un décodage réussit, si on a bien choisi le paramètre P.

La courbe ci-dessous représente le cardinal de P retourné par notre méthode utilisant la fonction de Landau, pour des valeurs allant de 0 à 10000.

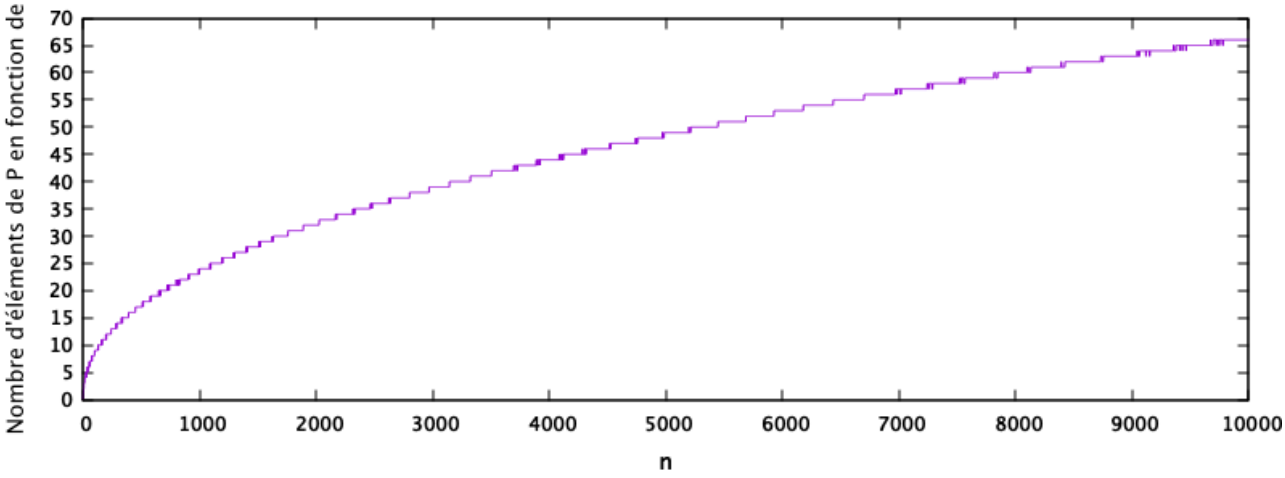


Figure 3.3 – Cardinal des ensembles P retournés par notre méthode utilisant la fonction de Landau, pour n allant de 0 à 10000.

Sur cette figure, on remarque que le cardinal de P progresse avec n, atteignant 66 pour $n = 10000$. Ceci pose un problème, puisque, comme on l'a vu précédemment, le temps de calcul de l'encodage et du décodage est proportionnel au cardinal de P. Par conséquent, notre méthode nous retourne ici des ensembles P ayant un très grand, voire trop grand cardinal, pour que nous voulions les utiliser en pratique.

3.1.3 PPCM maximal d'un ensemble dont la somme et le cardinal sont fixés : algorithme brute force

Nous venons de voir que la méthode utilisant la fonction de Landau permettait d'obtenir des ensembles P optimaux pour maximiser le rendement de notre code, mais qu'en termes de capacité de correction, ils s'avéraient loin d'être optimaux, puisque leur cardinal devient rapidement trop grand. Nous présentons ici une autre méthode permettant de trouver des ensembles P dont la somme et le cardinal sont fixés, et leur $PPCM$ maximal. Pour ce faire, nous parcourons tout simplement tous les ensembles dont la somme et le cardinal sont fixés, et conservons celui ayant le plus grand $PPCM$. Nous n'avons pas trouvé d'autres méthodes que la méthode "brute force", consistant à chercher tous les ensembles P de cardinal *card*, dont la somme vaut *sum*, et à conserver celui dont le $PPCM$ est maximisé, et il serait très intéressant de trouver une méthode plus rapide permettant de trouver ces ensembles.

Le tableau 3.4 présente ces ensembles P pour $9 \leq \Sigma(P) \leq 50$ et $5 \leq |P| \leq 9$. Dans ce tableau, les ensembles en rouge contiennent des 1, et les ensembles en jaune contiennent au moins deux nombres qui ne sont pas premiers entre eux (ils ne sont pas de bons candidats en tant que paramètre P). Les ensembles en vert ne possèdent pas de 1, et leur produit est égal à leur $PPCM$, ce qui en fait donc de bons candidats en tant que paramètre P. Nous n'affichons pas sur ce tableau les valeurs pour $|P| = 2$, $|P| = 3$ et $|P = 4|$ par manque de place, et parce qu'il est très simple de calculer ces ensembles pour $|P| = 2$ et $|P = 3|$ (un peu moins pour $|P| = 4$.

$\sum$P	|P| = 5	|P| = 6	|P| = 7	|P| = 8	|P| = 9
9	1 1 2 2 3	1 1 1 1 2 3	1 1 1 1 1 1 3	1 1 1 1 1 1 1 2	1 1 1 1 1 1 1 1 1
10	1 1 1 3 4	1 1 1 2 2 3	1 1 1 1 1 2 3	1 1 1 1 1 1 1 3	1 1 1 1 1 1 1 1 2
11	1 1 1 3 5	1 1 1 1 3 4	1 1 1 1 2 2 3	1 1 1 1 1 1 2 3	1 1 1 1 1 1 1 1 3
12	1 1 2 3 5	1 1 1 1 3 5	1 1 1 1 1 3 4	1 1 1 1 1 2 2 3	1 1 1 1 1 1 1 2 3
13	1 2 2 3 5	1 1 1 2 3 5	1 1 1 1 1 3 5	1 1 1 1 1 1 3 4	1 1 1 1 1 1 2 2 3
14	1 1 3 4 5	1 1 2 2 3 5	1 1 1 1 2 3 5	1 1 1 1 1 1 3 5	1 1 1 1 1 1 1 3 4
15	1 2 3 4 5	1 1 1 3 4 5	1 1 1 2 2 3 5	1 1 1 1 1 2 3 5	1 1 1 1 1 1 1 3 5
16	1 1 3 4 7	1 1 2 3 4 5	1 1 1 1 3 4 5	1 1 1 1 2 2 3 5	1 1 1 1 1 1 2 3 5
17	1 1 3 5 7	1 1 1 3 4 7	1 1 1 2 3 4 5	1 1 1 1 1 3 4 5	1 1 1 1 1 2 2 3 5
18	1 2 3 5 7	1 1 1 3 5 7	1 1 1 1 3 4 7	1 1 1 1 2 3 4 5	1 1 1 1 1 1 3 4 5
19	2 2 3 5 7	1 1 2 3 5 7	1 1 1 1 3 5 7	1 1 1 1 1 3 4 7	1 1 1 1 1 2 3 4 5
20	1 3 4 5 7	1 2 2 3 5 7	1 1 1 2 3 5 7	1 1 1 1 1 3 5 7	1 1 1 1 1 1 3 4 7
21	2 3 4 5 7	1 1 3 4 5 7	1 1 2 2 3 5 7	1 1 1 1 2 3 5 7	1 1 1 1 1 1 3 5 7
22	3 3 4 5 7	1 2 3 4 5 7	1 1 1 3 4 5 7	1 1 1 2 2 3 5 7	1 1 1 1 1 2 3 5 7
23	1 4 5 6 7	1 3 3 4 5 7	1 1 2 3 4 5 7	1 1 1 1 3 4 5 7	1 1 1 1 2 2 3 5 7
24	1 3 5 7 8	1 1 4 5 6 7	1 1 3 3 4 5 7	1 1 1 2 3 4 5 7	1 1 1 1 1 3 4 5 7
25	2 3 5 7 8	1 1 3 5 7 8	1 1 1 4 5 6 7	1 1 1 3 3 4 5 7	1 1 1 1 2 3 4 5 7
26	1 4 5 7 9	1 2 3 5 7 8	1 1 1 3 5 7 8	1 1 1 1 4 5 6 7	1 1 1 1 3 3 4 5 7
27	2 4 5 7 9	1 1 4 5 7 9	1 1 2 3 5 7 8	1 1 1 1 3 5 7 8	1 1 1 1 1 4 5 6 7
28	2 3 5 7 11	1 2 4 5 7 9	1 1 1 4 5 7 9	1 1 1 2 3 5 7 8	1 1 1 1 1 3 5 7 8
29	2 4 5 7 11	1 2 3 5 7 11	1 1 2 4 5 7 9	1 1 1 1 4 5 7 9	1 1 1 1 2 3 5 7 8
30	3 4 5 7 11	2 2 3 5 7 11	1 1 2 3 5 7 11	1 1 1 2 4 5 7 9	1 1 1 1 1 4 5 7 9
31	2 5 7 8 9	1 3 4 5 7 11	1 2 2 3 5 7 11	1 1 1 2 3 5 7 11	1 1 1 1 2 4 5 7 9
32	3 4 5 7 13	2 3 4 5 7 11	1 1 3 4 5 7 11	1 1 2 2 3 5 7 11	1 1 1 1 2 3 5 7 11
33	4 5 6 7 11	1 3 4 5 7 13	1 2 3 4 5 7 11	1 1 1 3 4 5 7 11	1 1 1 2 2 3 5 7 11
34	3 5 7 8 11	2 3 4 5 7 13	1 1 3 4 5 7 13	1 1 2 3 4 5 7 11	1 1 1 1 3 4 5 7 11
35	4 5 6 7 13	1 3 5 7 8 11	1 2 3 4 5 7 13	1 1 1 3 4 5 7 13	1 1 1 2 3 4 5 7 11
36	4 5 7 9 11	2 3 5 7 8 11	1 1 3 5 7 8 11	1 1 2 3 4 5 7 13	1 1 1 1 3 4 5 7 13
37	5 6 7 8 11	1 4 5 7 9 11	1 2 3 5 7 8 11	1 1 1 3 5 7 8 11	1 1 1 2 3 4 5 7 13
38	4 5 7 9 13	2 4 5 7 9 11	1 1 4 5 7 9 11	1 1 2 3 5 7 8 11	1 1 1 1 3 5 7 8 11
39	3 5 7 11 13	1 4 5 7 9 13	1 2 4 5 7 9 11	1 1 1 4 5 7 9 11	1 1 1 2 3 5 7 8 11
40	5 7 8 9 11	2 4 5 7 9 13	1 1 4 5 7 9 13	1 1 2 4 5 7 9 11	1 1 1 1 4 5 7 9 11
41	4 7 9 10 11	2 3 5 7 11 13	1 2 4 5 7 9 13	1 1 1 4 5 7 9 13	1 1 1 2 4 5 7 9 11
42	5 7 8 9 13	2 5 7 8 9 11	1 2 3 5 7 11 13	1 1 2 4 5 7 9 13	1 1 1 1 4 5 7 9 13
43	3 5 7 11 17	3 4 5 7 11 13	2 2 3 5 7 11 13	1 1 2 3 5 7 11 13	1 1 1 2 4 5 7 9 13
44	5 7 8 11 13	2 5 7 8 9 13	1 3 4 5 7 11 13	1 2 2 3 5 7 11 13	1 1 1 2 3 5 7 11 13
45	5 7 9 11 13	1 5 7 8 11 13	2 3 4 5 7 11 13	1 1 3 4 5 7 11 13	1 1 2 2 3 5 7 11 13
46	5 8 9 11 13	4 5 6 7 11 13	3 3 4 5 7 11 13	1 2 3 4 5 7 11 13	1 1 1 3 4 5 7 11 13
47	7 8 9 10 13	3 5 7 8 11 13	1 4 5 6 7 11 13	1 3 3 4 5 7 11 13	1 1 2 3 4 5 7 11 13
48	7 8 9 11 13	3 4 7 10 11 13	1 3 5 7 8 11 13	1 1 4 5 6 7 11 13	1 1 3 3 4 5 7 11 13
49	5 7 9 11 17	4 5 7 9 11 13	2 3 5 7 8 11 13	1 1 3 5 7 8 11 13	1 1 1 4 5 6 7 11 13
50	7 9 10 11 13	5 6 7 8 11 13	1 4 5 7 9 11 13	1 2 3 5 7 8 11 13	1 1 1 3 5 7 8 11 13

TABLE 3.4 – Ensembles P dont la somme et le cardinal sont fixés, et dont le $PPCM$ est maximal. Les ensembles en rouge contiennent un 1 (et ne sont pas optimaux pour comme paramètre P), les ensembles en jaune contiennent deux nombres qui ne sont pas premiers entre eux (ce qui est sous-optimal pour le paramètre P), et les ensembles en vert sont de bons candidats pour P.

Le tableau ci-dessous3.5 présente ces ensembles P pour $50 \leq \Sigma(P) \leq 125$ et $6 \leq |P| \leq 9$. Les couleurs rouge, jaune et vert ont la même signification que pour le tableau 3.4.

$\sum$P	|P| = 6	|P| = 7	|P| = 8	|P| = 9
50	5 6 7 8 11 13	1 4 5 7 9 11 13	1 2 3 5 7 8 11 13	1 1 1 3 5 7 8 11 13
55	4 5 7 9 13 17	2 5 7 8 9 11 13	1 1 5 7 8 9 11 13	1 1 4 4 5 7 9 11 13
60	5 8 9 11 13 14	3 4 5 7 11 13 17	2 2 3 5 7 11 13 17	1 1 2 3 5 7 11 13 17
65	7 8 9 11 13 17	4 5 6 7 11 13 19	1 3 5 7 8 11 13 17	1 2 3 4 5 7 11 13 19
70	5 7 9 13 17 19	5 7 8 9 11 13 17	2 4 5 7 9 11 13 19	1 1 4 5 7 9 11 13 19
75	7 9 11 13 16 19	7 8 9 10 11 13 17	3 5 7 8 9 11 13 19	1 2 5 7 8 9 11 13 19
80	7 9 11 13 17 23	5 7 9 11 13 16 19	2 5 7 9 11 13 16 17	1 3 4 5 7 11 13 17 19
85	9 11 13 16 17 19	5 7 9 11 13 17 23	4 5 7 9 11 13 17 19	2 3 5 7 8 11 13 17 19
90	7 11 13 17 19 23	7 8 11 13 15 17 19	4 7 9 10 11 13 17 19	1 5 7 8 9 11 13 17 19
100	9 11 13 19 23 25	8 9 11 13 17 19 23	7 8 9 10 11 13 19 23	3 5 7 9 11 13 16 17 19
105	11 13 17 20 21 23	7 11 13 15 17 19 23	5 8 9 11 13 17 19 23	4 5 6 7 11 13 17 19 23
110	11 13 17 21 23 25	9 11 13 16 17 19 25	5 7 9 11 13 17 19 29	1 7 9 10 11 13 17 19 23
115	13 17 19 21 22 23	7 11 13 17 19 23 25	7 9 11 13 16 17 19 23	4 5 9 11 13 14 17 19 23
120	11 17 19 21 23 29	11 13 16 17 19 21 23	8 11 13 14 15 17 19 23	5 7 9 11 13 16 17 19 23
125	16 17 19 21 23 29	9 11 13 17 19 25 31	8 9 11 13 17 19 23 25	7 9 10 11 13 16 17 19 23

TABLE 3.5 – Ensembles P dont la somme et le cardinal sont fixés, et dont le $PPCM$ est maximal.

On remarque qu'ici, les ensembles retournés ne contiennent pas uniquement des puissances de nombres premiers (à la différence de ceux retournés avec la méthode s'appuyant sur la fonction de Landau), mais peuvent contenir des nombres composés. Par exemple, l'ensemble retourné pour $|P| = 5$ et $\Sigma(P) = 125$ est $P = [16, 17, 19, 21, 23, 29]$, avec $21 = 3 * 7$.

Il peut aussi être intéressant de comparer le PPCM des ensembles retournés avec la valeur maximale qu'il pourrait atteindre. En effet, le PPCM d'un ensemble de nombres entiers est toujours inférieur ou égal à son produit, et un ensemble de *card* nombres réels dont la somme vaut *somme* aura comme produit $(somme/card)^{card}$. Concrètement, en cherchant l'ensemble à 5 éléments dont la somme vaut 100, le meilleur ensemble que l'on puisse trouver aura comme PPCM maximum le produit de l'ensemble $[20, 20, 20, 20, 20]$, qui vaut 3200000. Notre algorithme nous retourne comme meilleur ensemble $[17, 19, 20, 21, 23]$, dont le PPCM vaut 3120180, qui divisé par 3200000 vaut environ 0.97505625. On peut intuitivement penser qu'il s'agit d'un bon candidat comme paramètre P. Nous présenterons plus loin des simulations afin de comparer quels sont les meilleurs ensembles P retournés par cette méthode en terme de capacité de correction. Le tableau 3.6 présente ces ratios pour différents ensembles P, avec $3 \leq |P| \leq 9$ et $10 \leq \Sigma(P) \leq 125$.

$\sum$P	|P| = 3	|P| = 4	|P| = 5	|P| = 6	|P| = 7	|P| = 8	|P| = 9
10	0.81	0.384	0.375	0.2799	0.4941	0.5033	0.7748
15	0.84	0.4247	0.2469	0.2457	0.1446	0.1963	0.1511
20	0.945	0.336	0.4101	0.153	0.1351	0.0688	0.0635
25	0.8553	0.8257	0.2688	0.1605	0.0566	0.0461	0.0426
30	0.99	0.4171	0.5941	0.1478	0.0869	0.0322	0.0248
35	0.8595	0.9457	0.3248	0.2345	0.0698	0.0406	0.0227
40	0.9745	0.9009	0.8459	0.1865	0.0823	0.0354	0.0204
45	0.9822	0.8405	0.7628	0.2249	0.1323	0.0599	0.0153
50	0.9849	0.8961	0.9009	0.3586	0.1899	0.0515	0.0238
55	0.9517	0.9601	0.6792	0.4693	0.1949	0.0722	0.0151
60	0.9975	0.9123	0.8792	0.3603	0.3003	0.0509	0.0196
65	0.9807	0.9727	0.7683	0.7579	0.1917	0.1075	0.0213
70	0.9957	0.9401	0.9492	0.5245	0.6126	0.0996	0.0328
75	0.9936	0.9617	0.9123	0.7179	0.3779	0.1147	0.0353
80	0.9966	0.975	0.961	0.6269	0.5377	0.1225	0.0559
85	0.9881	0.9901	0.8633	0.8228	0.4524	0.3583	0.0648
90	0.9988	0.9291	0.9435	0.6528	0.668	0.2268	0.1163
95	0.9908	0.9867	0.8903	0.8517	0.4384	0.3982	0.0866
100	0.9979	0.9582	0.975	0.656	0.6299	0.2642	0.0901
105	0.9967	0.9861	0.9549	0.8175	0.6528	0.4342	0.1114
110	0.9982	0.9761	0.9793	0.773	0.7027	0.3302	0.1099
115	0.9935	0.9947	0.9249	0.8999	0.5755	0.5873	0.1474
120	0.9993	0.9558	0.9675	0.7776	0.8203	0.3481	0.402
125	0.9897	0.9954	0.9363	0.8853	0.5564	0.5382	0.2784

TABLE 3.6 – Ratio entre le PPCM des ensembles P retournés par notre algorithme et l'optimal théorique, qui vaut $(\Sigma(P)/|P|)^{|P|}$

Le tableau ci-dessus 3.6 est intéressant, puisqu'il permet de s'apercevoir que plus $\Sigma(P)$ est grand, ou plus $|P|$ est petit, plus le ratio $PPCM(P)/optimum$ tend vers 1. Ceci s'explique par le fait que plus $\Sigma(P)$ est grand, plus il existe de nombres entiers proches de $\Sigma(P)/|P|$ premiers entre eux. Aussi, plus le cardinal de P est petit, plus $\Sigma(P)/|P|$ est grand, ce qui revient au même. Pour un grand cardinal, il nous faut trouver un ensemble dont la somme est suffisamment grande pour que cet ensemble soit vert (c'est-à-dire utilisable en tant que paramètre P).

Algorithme brute-force

Il existe plusieurs algorithmes permettant de trouver tous les ensembles d'entiers positifs dont le cardinal vaut *card* et la somme vaut *sum*. Nous présentons ici le pseudo-code d'un de ces algorithmes (c'est l'algorithme que nous avons utilisé pour obtenir nos ensembles).

Algorithm 5 Algorithme brute-force retournant l'ensemble de cardinal *card* et dont la somme vaut *sum* dont le $PPCM$ est maximal

Require: Cet algorithme prend en entrée le nombre entier positif *card* et le nombre entier *sum* et retourne l'ensemble P de cardinal *card* et dont la somme vaut *sum* dont le $PPCM$ est maximal

```
1 : function BRUTEFORCE(card,sum)
2 :     PPCM_best = 0
3 :     A_best = Array[card]
4 :     A = Array[card]
5 :     for i ← 0 to card − 1 do
6 :         A[i] ← 0
7 :     A[1] ← n
8 :     k ← 1
9 :     while k ≠ 0 do
10 :        x ← A[k − 1] + 1
11 :        y ← A[k] − 1
12 :        k ← k − 1
13 :        while x ≤ y AND k ≤ card − 1 do
14 :            A[k] ← x
15 :            y ← y − x
16 :            k ← k + 1
17 :        A[k] = x + y
18 :        if PPCM(A) ≥ PPCM_best then
19 :            PPCM_best ← PPCM(A)
20 :            A_best ← A
        return A
```

3.1.4 Simulations de décodage avec la méthode brute-force

Nous présentons ici des simulations permettant d'observer la capacité de correction de notre code pour différents paramètres P retournés par l'algorithme brute-force que nous venons de présenter. Nous proposons plusieurs simulations :

- Pour les 3 premières, nous avons fixé $n - k = 126$ paquets de redondance, et faisons varier le cardinal de P. La première a une dimension $k = n - k = 126$, la deuxième $k = 4(n - k) = 504$, la deuxième $k = 16(n - k) = 2016$.
- Pour la 4ème, nous avons fixé $P = [29, 31, 32, 33]$ $(n - k = 126)$, et faisons varier le nombre de paquets sources, $k \in [126, 252, 378, 504, 630, 756, 882, 1008]$ pour obtenir des codes de rendement différents.
- Pour la dernière, nous présentons différents des simulations pour différents ensembles P et différentes dimensions k, qui donnent de bonnes capacités de correction.

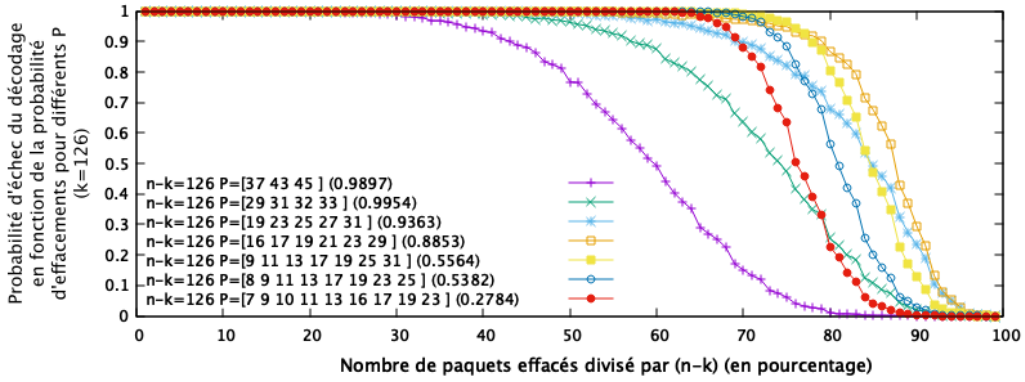


FIGURE 3.4 – Probabilité d'échec du décodage en fonction du nombre d'effacements divisé par (n-k) ($k = 126$)

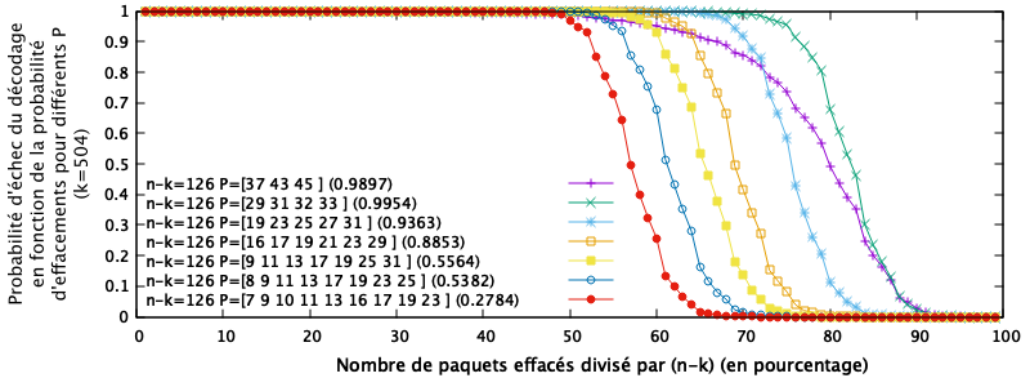


FIGURE 3.5 – Probabilité d'échec du décodage en fonction du nombre d'effacements divisé par (n-k) ($k = 504$)

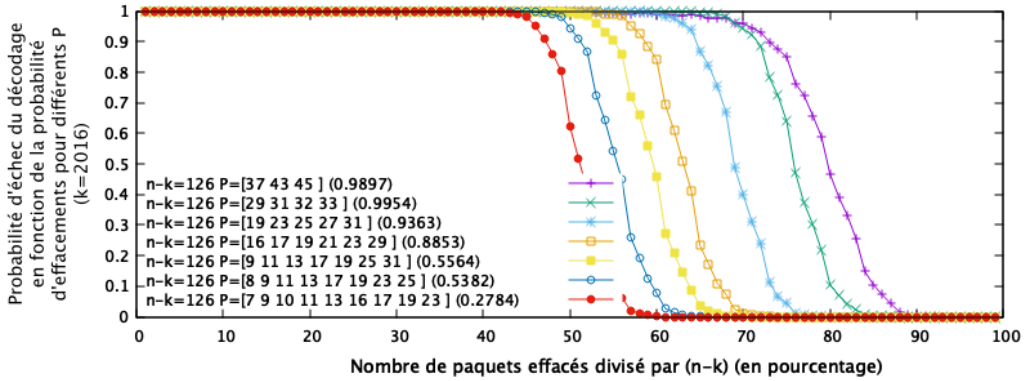


FIGURE 3.6 – Probabilité d'échec du décodage en fonction du nombre d'effacements divisé par (n-k) ($k = 2016$)

La figure 3.7 ci-dessous présente des simulations effectuées en fixant $P = [29, 31, 32, 33]$ et en faisant varie k le nombre de paquets sources (et donc le rendement du code).

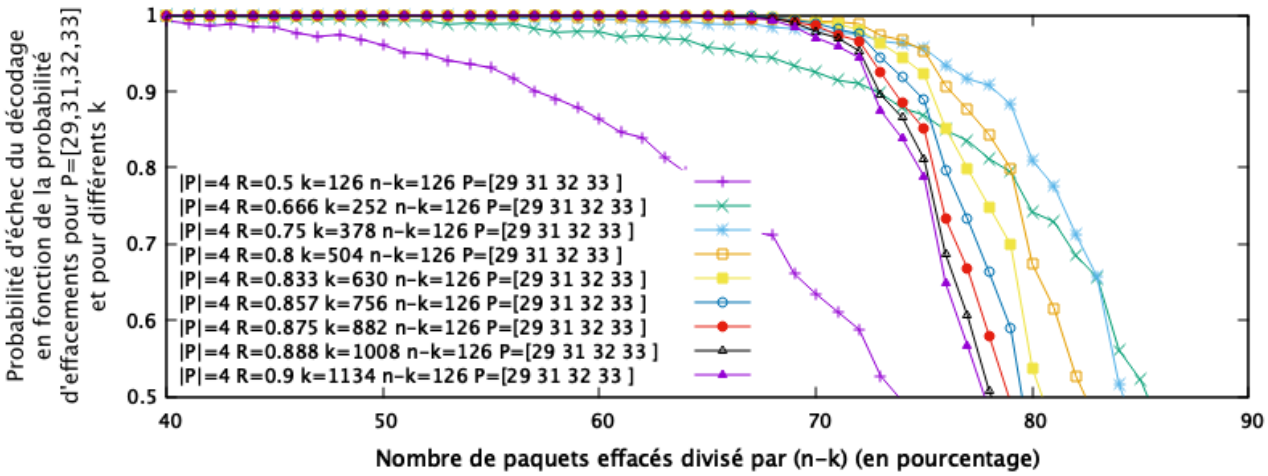


FIGURE 3.7 – Probabilité d'échec du décodage en fonction du nombre d'effacements divisé par (n-k)

On remarque que la moins bonne capacité de correction est atteinte pour $k = n - k$ (donc un rendement $R = 1/2$), puis vient $k = 2 * (n - k)$ ($R = 2/3$), mais la meilleure capacité de correction est atteinte pour $k = 3 * (n - k)$ ($R = 3/4$). Ainsi, il semble que la capacité de correction ne progresse pas automatiquement avec le rendement du code, mais qu'il existe un optimum qu'il faut déterminer.

Enfin, en utilisant notre méthode, nous avons réalisés de nombreux tests, pour différents P retournés par notre méthode, et différents k (nous faisons donc varier le rendement de notre code), et avons conservés ceux qui avaient une bonne capacité de correction. La figure 3.8 présente ces simulations : nous avons centré les probabilités d'effacements entre 40% et 80%, et les probabilités de décodage entre 90% et 100% pour plus de clarté. On remarque que pour certaines d'entre elles ($P = [23, 29, 31, 33]$ et $k = 571$, ou encore $P = [11, 17, 19, 21, 23, 25]$ et $k = 206$), la probabilité de réussite du décodage est pratiquement de 100%, même lorsqu'on atteint une probabilité d'effacements de 65%.

Encore une fois, notre code n'étant pas un code MDS, il est normal que la probabilité de réussite du décodage soit inférieure à 100%, même quand on a reçu plus de k paquets. Cette méthode nous permet néanmoins d'obtenir de bonnes capacité de correction, avec une probabilité de décoder proche de 100% quand on a reçu $k + (n - k) * 30\%$.

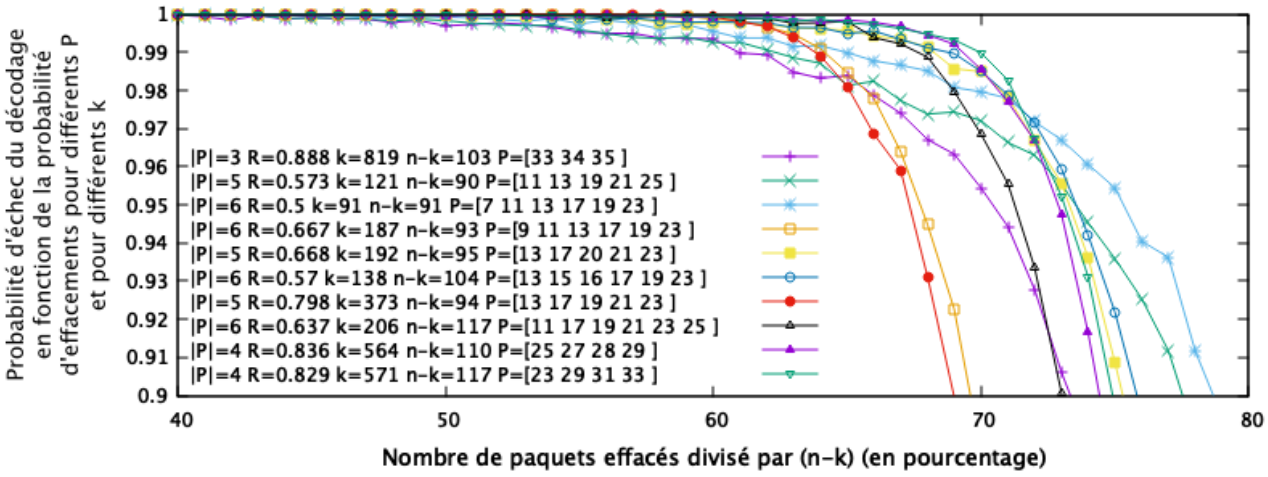


FIGURE 3.8 – Probabilité d'échec du décodage en fonction du nombre d'effacements divisé par (n-k)

3.1.5 Conclusion

Nous avons présenté ici une première méthode reposant sur le calcul de la fonction de Landau permettant de calculer des ensembles P nous permettant de maximiser le rendement de notre code pour un nombre de paquets de redondance $n-k$ fixé, ou à l'inverse pour un nombre de paquets sources k fixé. C'est intéressant lorsqu'on a très peu de pertes sur le canal de transmission, et qu'on souhaite donc ajouter le moins de paquets de redondance possible tout en fiabilisant le plus grand nombre de paquets sources. Cependant, pour de grandes dimensions, cet algorithme nous retourne des ensembles dont le cardinal est très grand, ce qui augmente considérablement le nombre de calculs à effectuer lors de l'encodage et du décodage. De plus, la capacité de correction résultant de ces paramètres P est sous-optimale (là-aussi, parce que le cardinal de P devient rapidement trop grand). Cette méthode peut donc être utilisée dans certains cas d'utilisation mais pas dans d'autres.

Nous avons de plus présenté une autre méthode, basée sur un algorithme brute-force, nous retournant des ensembles P dont nous fixons le cardinal et la somme, et dont le $PPCM$ est maximal. Ces ensembles sont de bons candidats en tant que paramètre P pour notre code, et nous permettent dans certains cas (pour des valeurs de k adéquats) d'obtenir de bonnes capacités de correction. Sur le site, de l'OEIS , les suites $A129649$ et $A129650$ correspondent aux $PPCM$ des ensembles retournés par notre algorithme, pour $|P| = 4$ et $|P| = 5$.

Il serait très intéressant de trouver un algorithme plus rapide que l'algorithme brute-force pour obtenir ces ensembles, mais nous n'y sommes pas parvenus.

3.2 Enumération de forêts dans les graphes bipartis et capacité de correction

Dans cette partie, nous étudions notre code correcteur du point de vue de la théorie des graphes. Dans un premier temps, nous présentons une manière plus lisible de représenter notre code en réarrangeant son système d'équations, pour le représenter par un graphe de Rook. De cette façon, nous présentons ensuite un algorithme, proposé par Rebecca Stones dans [56] permettant de calculer exactement la probabilité de décoder ou non un ensemble d'effacements, en comptant le nombre de sous-graphes de taille e dans le graphe de Tanner d'un graphe de Rook qui ne contiennent pas de cycle (ceux sont donc des forêts). En utilisant les résultats retournés par cet algorithme que nous interpolerons, nous proposons un certain nombre de formules closes retournant le nombre exact de forêts de taille e dans nos graphes de Tanner : un graphe de Tanner ne contenant pas de cycle correspond alors à une configuration d'effacements totalement décodable. Ainsi, en connaissant le nombre de forêts de taille e dans un graphe de Tanner, on peut déterminer la probabilité exacte de décoder totalement e paquets effacés, en comparant ce nombre au nombre total de sous-graphes de taille e dans ce graphe de Tanner.

3.2.1 Système d'équations et graphe de Rook

On a vu jusqu'ici qu'il était possible de représenter le décodage de notre code, comme tout code linéaire, par son graphe de Tanner. La figure 3.9 ci-dessous représente le graphe de Tanner de notre code ayant pour paramètre $P = [2, 3]$ et $k = 12$.

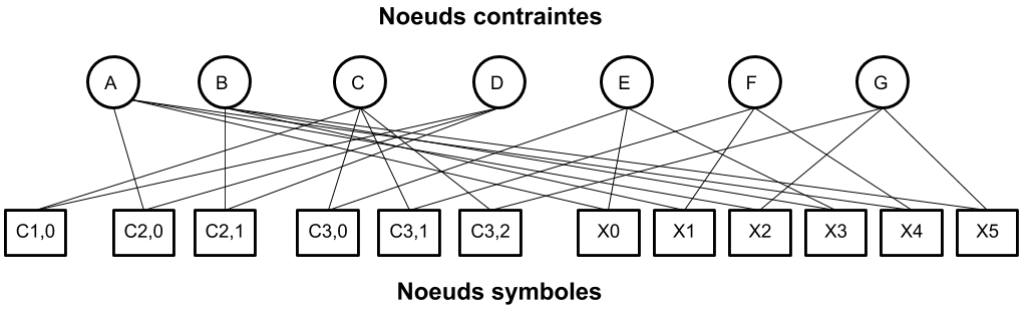


FIGURE 3.9 – Graphe de Tanner de notre code pour P=[2,3] et k=6

On remarque que même pour des valeurs de P relativement petites, et P n'ayant que deux éléments, le graphe de Tanner devient rapidement difficile à lire. Dans le cas ou P ne contient que deux éléments, il est cependant possible de représenter son graphe de Tanner de manière plus lisible.

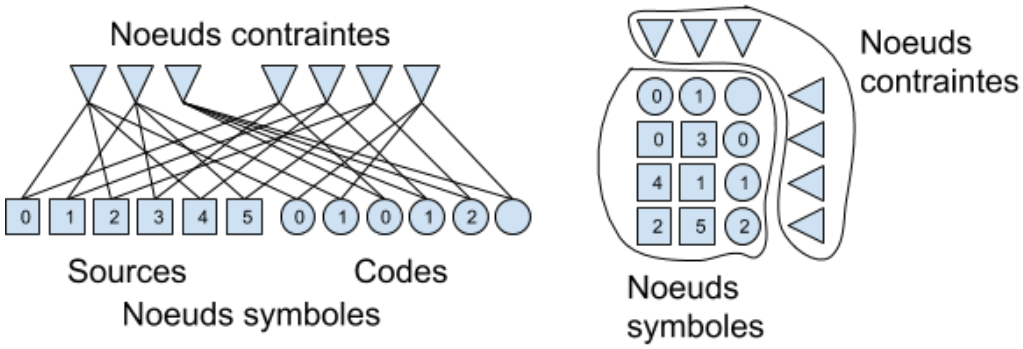


FIGURE 3.10 – Graphe de Tanner réarrangé de notre code pour P=[2,3] et k=6

Ces deux graphes sont le même graphe de Tanner correspondant à $P = [2, 3]$ et $k = 6$. Le graphe de gauche est le graphe de Tanner tel que nous l'avons présenté jusqu'à présent. Celui de droite a été réarrangé, et nous avons enlevé les arêtrs pour plus de clareté. Les noeuds symboles sources sont des représentés par des carrés, les noeuds symboles codes par des ronds, et les noeuds contraintes par des triangles.

On peut même représenter ce graphe de Tanner de manière encore plus simplifiée, en retirant du graphe les noeuds contraintes, comme nous le faisons dans la figure 3.11. Ainsi, pour qu'un paquet soit décodable à une itération donnée de l'algorithme de décodage, il suffit qu'il soit seul sur sa ligne ou sur sa colonne.

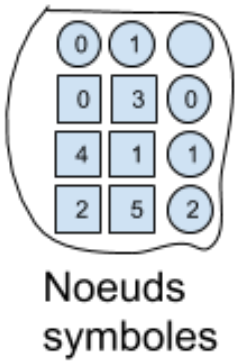


FIGURE 3.11 – Graphe de Tanner réarrangé de notre code pour P=[2,3] et k=6, auquel on a enlevé les noeuds contraintes. Un paquet est décodable si il est seul sur sa ligne ou sa colonne.

Lors du décodage itératif, un paquet (noeud symbole) est décodable si il est relié à un noeud contrainte n'ayant qu'un seul voisin. Par conséquent avec cette représentation, un paquet est décodable si il est le seul paquet sur sa ligne ou sur sa colonne. Les figures 3.12 et 3.13 représentent respectivement un graphe de Tanner possédant un cycle et un graphe de Tanner sans cycle : on voit apparaître ou non un "cycle" dans le graphe réarrangé correspondant. Nous mettons le terme "cycle" entre guillemets car en réalité, 3 sommets sur la même ligne (ou la même colonne) forment un cycle, et sont pourtant décodables. Quand nous parlons de "cycle" dans le graphe de Rook, nous parlons donc en réalité de cycle dans son graphe de Tanner.

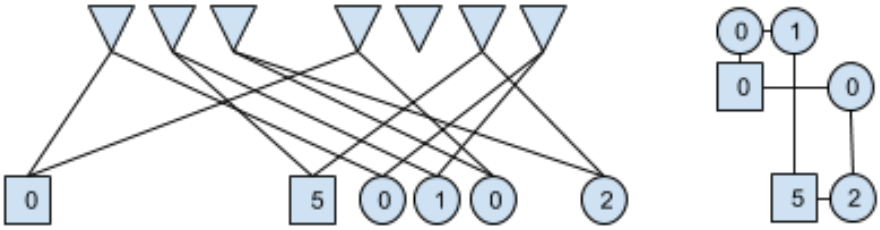


FIGURE 3.12 – Le graphe de Tanner contient un cycle, les paquets réarrangés en graphe de Rook forment eux aussi un "cycle" et ne peuvent pas être décodés.

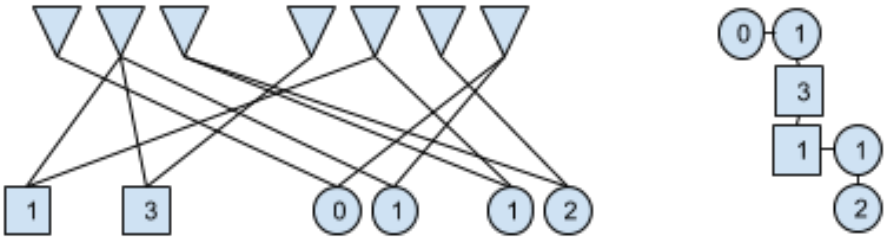


FIGURE 3.13 – Le graphe de Tanner ne contient pas de cycle, les paquets réarrangés en graphe de Rook ne forment pas de "cycle" et sont donc décodables : on va décoder $C_{2,0}$ et $C_{3,2}$, puis X_0 et $C_{3,1}$, et enfin X_3 et X_1.

Dans le cas ou on a comme paramètre $|P| = 2$ et $k = P_0 * P_1$ on peut donc représenter notre système d'équations par un graphe de Rook. Pour $P = [P_0, P_1]$, on devra prendre considérer le graphe de Rook de dimensions $(m = P_0 + 1, m = P_1 + 1)$. Dans la prochaine section nous nous servons de cette représentation pour représenter le graphe de Tanner de notre code là-aussi de façon plus lisible.

Graphe de Rook et probabilité exacte d'erreur du décodage - démonstrations formelles pour $e <= 6$

On a vu précédemment qu'on pouvait représenter le système d'équations de notre code, quand celui-ci est paramétré avec P possédant deux éléments, par un graphe de Rook. On va ici présenter une façon de calculer la probabilité exacte de décoder totalement e effacements, pour $P = [P_0, P_1]$ et $k = P_0 * P_1$ fixé.

Pour ce faire, nous allons devoir dénombrer l'ensemble des sous-graphes d'un graphe de taille e de Rook de dimension $(m = P_0 + 1, n = P_1 + 1)$ qui contiennent au moins un "cycle" (là-encore, le terme cycle n'est pas exact, il faut plutôt parler de cycle dans son graphe de Tanner).

Notons $Forets(m, n, e)$ le nombre de sous-graphes de taille e d'un graphe de Rook de dimensions (m, n) ne contenant pas de "cycle" (ceux sont donc des forêts décodables), et $Cycles(m, n, e)$ le nombre de sous-graphes de taille e d'un graphe de Rook de dimensions (m, n) contenant au moins un cycles (donc des graphes non-décodables).

Remarquons que l'ensemble des sous-graphes d'un graphe de Rook de dimension (m, n) a pour cardinal $\binom{m*n}{e}$, et que par conséquent $Forets(m, n, e) + Cycles(m, n, e) = \binom{m*n}{e}$.

Si les effacements se font de manière équiprobable, la probabilité d'échec du décodage en fonction du nombre d'effacements e sera exactement de
$\mathcal{P}_{echec}(m, n, e) = Cycles(m, n, e)/\binom{mn}{e}$.

Dans un premier temps, essayons de calculer cette valeur pour des petites valeurs de e, allant de 0 à 6.

Demonstration pour e<4

Le graphe de Tanner d'unn graphe de Rook étant un graphe biparti, il ne possède pas de cycle de longueur inférieure à 4. C'est d'ailleurs pour cette raison que notre code a une distance minimale $\delta = 4$. Par conséquent, il n'existe pas d'ensemble de 0, 1, 2, ou 3 paquets qui ne soit pas décodable.

Ainsi, on peut dire que

$$Cycles(m, n, e \leq 3) = 0$$
$$Forets(m, n, e \leq 3) = \binom{m*n}{e}$$

Démonstration pour e=4

Dans le cas ou $e = 4$, les sous-graphes "cycliques" de taille 4 d'un graphe de Rook de dimensions (m, n) forment un rectangle.

Ainsi, il faut compter le nombre de rectangles présent dans ce graphe de Rook.

Pour produire un rectangle dans ce graphe de Rook, il faut choisir 2 lignes aléatoirement parmi les n lignes du graphe de Rook (il y en a $n * (n - 1)/2$ possibilités), et 2 colonnes aléatoirement parmis les n colonnes du graphe de Rook (il y en a $m * (m - 1)/2$.

On obtient alors

$$Cycles(m, n, 4) = m * n * (m - 1) * (n - 1)/4$$
$$Forets(m, n, 4) = \binom{m*n}{4} - m * n * (m - 1) * (n - 1)/4$$

Démonstration pour e=5

Dans le cas ou $e = 5$, les graphes "cycliques" de taille 5 d'un graphe de Rook de dimensions (m, n) forment un rectangle, auquel on aurait ajouté un sommet n'importe ou qui, lui, sera décodable. La figure 3.14 représente quelques exemples de graphes de taille 5 contenant un cycle de taille 4.

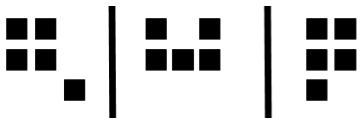


FIGURE 3.14 – Exemples de sous-graphes de taille 5 contenant un cycle.

Ainsi, pour calculer ce nombre, il nous suffit de reprendre la valeur
$Cycles(m, n, 4) = m * n * (m - 1) * (n - 1)/4$, et de la multiplier par le nombre de sommets restants, $m * n - 4$.

On obtient alors

$$Cycles(m,n,5) = m*n*(m-1)*(n-1)/4*(m*n-4)$$
$$Forets(m,n,5) = \binom{m*n}{5} - m*n*(m-1)*(n-1)/4*(m*n-4)$$

Démonstration pour e=6

Pour $e = 6$, le calcul est légèrement plus complexe.

La figure 3.15 ci-dessous représente les différents types de graphes "cycliques" de taille 6 d'un graphe de Rook.

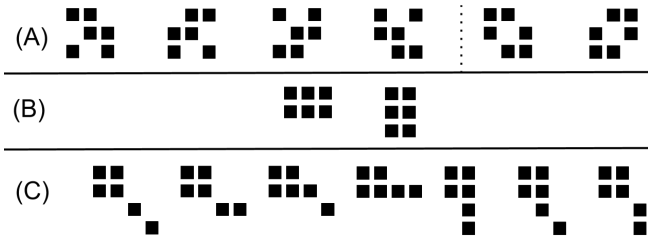


FIGURE 3.15 – Les différentes manières d'obtenir un sous-graphe de taille 6 contenant un cycle.

En étudiant séparément les 3 cas **(A)**, **(B)** et **(C)** de la figure 3.15, on peut déterminer $Cycles(m,n,6)$:

- **(A)** nous devons choisir 3 lignes et 3 colonnes parmi les n lignes et m colonnes, ce qui nous donne $m(m-1)(m-2)n(n-1)(n-2)/36$, puis multiplier par les 6 différentes façons d'agencer les effacements à l'intersection de ces 3 lignes et 3 colonnes. On obtient donc $m(m-1)(m-2)n(n-1)(n-2)/6$ pour les graphes de la catégorie **(A)**.
- **(B)** Nous devons choisir 3 lignes et 2 colonnes, ou bien 2 lignes et 3 colonnes, ce qui nous donne $m(m-1)n(n-1)(m+n-4)/12$.
- **(C)** Pour la catégorie **(C)**, nous partons du fait qu'il existe un "cycle" de taille 4 auquel nous ajoutons 2 sommets n'importe ou, à l'exception des combinaisons qui nous donnent un graphe de catégorie **(B)**.
  Ainsi, on a $mn(m-1)(n-1)/4$ cycles de taille 4, qu'on doit multiplier par le nombre de combinaisons de 2 symboles n'importe ou sur les cases restantes $(mn-4)(mn-5)/2$, auquel on retranche les $(m+n-2)$ couples qui nous donneraient un graphe de catégorie **(B)**. On obtient alors $mn(m-1)(n-1)/4*((mn-4)(mn-5)-(m+n-2))$.

En additionnant les 3 équations que nous avons obtenues pour les 3 différentes catégories **(A)**, **(B)** et **(C)** puis en les factorisant, on obtient le polynôme suivant :

$$Cycles(m,n,6) = 1/24*mn(m-1)(n-1)(3m^2n^2-23mn-12m-12n+92)$$
$$Forets(m,n,6) = \binom{mn}{6} - 1/24*mn(m-1)(n-1)(3m^2n^2-23mn-12m-12n+92)$$

3.2.2 Graphe de Rook et probabilité exacte d'erreur du décodage - Algorithme de Stones

Dans la partie précédente, on a calculé le nombre de sous-graphes de taille inférieure ou égale à 6 d'un graphe de Rook de dimension (m,n) contenant au moins un cycle (ou réciproquement

n'en contenant aucun). Cette méthode est néanmoins fastidieuse à mesure que e devient grand, puisque de nombreux cas particuliers susceptibles d'être comptés plusieurs fois apparaissent. Nous présentons ici un algorithme proposé par Rebecca Stones dans [56] qui permet de calculer les valeurs de $Forets(m, n, e)$, ayant une complexité exponentielle en mémoire. Cet algorithme nous permet néanmoins de calculer ces valeurs pour m, n et e allant jusqu'à 40 environ.

Dans cette partie, nous allons donc expliquer cet algorithme, et la manière dont nous nous sommes servis de ces résultats pour calculer un certain nombre de formules closes pour $Cycles(m, n, e)$ pour de petites valeurs de e, en interpolant ces résultats.

Tout d'abord, présentons les premières valeurs que l'on obtient avec cet algorithme.

m	n	e=	0	1	2	3	4	5	6	7	8	9	10
1	1		1	1									
1	2		1	2	1								
1	3		1	3	3	1							
1	4		1	4	6	4	1						
1	5		1	5	10	10	5	1					
1	6		1	6	15	20	15	6	1				
2	2		1	4	6	4							
2	3		1	6	15	20	12						
2	4		1	8	28	56	64	32					
2	5		1	10	45	120	200	192	80				
2	6		1	12	66	220	480	672	544	192			
3	3		1	9	36	84	117	81					
3	4		1	12	66	220	477	648	432				
3	5		1	15	105	455	1335	2673	3375	2025			
3	6		1	18	153	816	3015	7938	14499	16524	8748		
4	4		1	16	120	560	1784	3936	5632	4096			
4	5		1	20	190	1140	4785	14544	31520	44800	32000		
4	6		1	24	276	2024	10536	40704	117376	244224	331776	221184	
5	5		1	25	300	2300	12550	51030	155900	347500	515625	390625	
5	6		1	30	435	4060	27255	138606	544525	1641000	3645000	5400000	4050000
6	6		1	36	630	7140	58680	369792	1834992	7210080	22083840	50388480	77262336

TABLE 3.7 – Valeurs de $Cycles(m, n, e)$ retournées par l'algorithme de Stones

Le tableau 3.7 se lit de la manière suivante : pour $m = 3$ et $n = 6$, il existe 14499 sous-graphes de taille $e = 6$ d'un graphe de Rook de dimensions (m, n) qui sont des "forêts".

Confrontation de nos équations et des résultats retournés par l'algorithme Stones
Nous avons donc obtenu les équations suivantes :

$$Cycles(m, n, 4) = 1/4 * mn(m-1)(n-1)$$
$$Cycles(m, n, 5) = 1/4 * mn(m-1)(n-1)(mn-4)$$
$$Cycles(m, n, 6) = 1/24 * mn(m-1)(n-1)(3m^2n^2 - 23mn - 12m - 12n + 92)$$

Vérifions que les résultats retournés par l'algorithme coïncident bien avec ces équations : le tableau ci-dessous 3.8 contient les valeurs des polynômes que nous venons de donner pour $1 \leq m \leq 6$, $1 \leq n \leq 6$ et $4 \leq e \leq 6$.

m	n	e=	4 Forets	5 Forets	6 Forets	4 Cycles	5 Cycles	6 Cycles
1		1	0	0	0	0	0	0
1		2	0	0	0	0	0	0
1		3	0	0	0	0	0	0
1		4	1	0	0	0	0	0
1		5	5	1	0	0	0	0
1		6	15	6	1	0	0	0
2		2	0	0	0	1	0	0
2		3	12	0	0	3	6	1
2		4	64	32	0	6	24	28
2		5	200	192	80	10	60	130
2		6	480	672	544	15	120	380
3		3	117	81	0	9	45	84
3		4	477	648	432	18	144	492
3		5	1335	2673	3375	30	330	1630
3		6	3015	7938	14499	45	630	4065
4		4	1784	3936	5632	36	432	2376
4		5	4785	14544	31520	60	960	7240
4		6	10536	40704	117376	90	1800	17220
5		5	12550	51030	155900	100	2100	21200
5		6	27255	138606	544525	150	3900	49250
6		6	58680	369792	1834992	225	7200	112800

TABLE 3.8 – Valeurs de $Forets(m, n, e)$ et de $Cycles(m, n, e)$ pour $1 \leq m \leq 6$, $1 \leq n \leq 6$ et $4 \leq e \leq 6$. $Forets(m, n, e) + Cycles(m, n, e) = \binom{mn}{e}$

Le tableau 3.8 ci-dessus montre que nos équations sont en adéquation avec les résultats retournés par l'algorithme de Stones. De plus, pour des petites valeurs de m et n, nous avons énuméré tous les sous-graphes des graphes de Rook de dimensions (m, n), afin de les comparer avec ces résultats.

Nous sommes cependant confrontés à un problème : nous aimerions disposer des formules closes de $Forets(m, n, e)$ (ou de $Cycles(m, n, e)$) pour de plus grandes valeurs de e, mais trouver ces équations "à la main" en étudiant tous les cas possibles devient rapidement compliqué. Nous avons donc entrepris la démarche suivante : en disposant d'assez de résultats retournés par l'algorithme, nous souhaiterions les interpoler afin d'en déduire le polynôme à deux variables m et n passant par tous ces points. Une fois un tel polynôme obtenu, nous pourrons vérifier qu'il s'agit de la bonne équation en l'appliquant à des valeurs que nous n'avons pas interpolées, et en vérifiant qu'elles coïncident bien avec les valeurs retournées par l'algorithme.

Présentons notre méthode sur un exemple : supposons que l'on ne connaisse pas l'équation $Cycles(m, n, 6)$, et qu'on souhaite la trouver en interpolant les résultats retournés par l'algorithme.

Dans un premier temps, nous allons chercher les équations de $Cycles(0, n, 6)$, $Cycles(1, n, 6)$, $Cycles(2, n, 6)$, $Cycles(3, n, 6)$, $Cycles(4, n, 6)$, ..., ce qui s'avère relativement simple.

Une fois ces équations obtenues, nous les alignons selon les degrés de leurs monômes, puis allons interpoler les coefficients associés à chacun de ces monômes :

$$\begin{aligned}
Cycles(0, n, 6) &= a(0) * n^4 + b(0) * n^3 + c(0) * n^2 + d(0) * n^1 + e(0) * n^0 \\
Cycles(1, n, 6) &= a(1) * n^4 + b(1) * n^3 + c(1) * n^2 + d(1) * n^1 + e(1) * n^0 \\
Cycles(2, n, 6) &= a(2) * n^4 + b(2) * n^3 + c(2) * n^2 + d(2) * n^1 + e(2) * n^0 \\
Cycles(3, n, 6) &= a(3) * n^4 + b(3) * n^3 + c(3) * n^2 + d(3) * n^1 + e(3) * n^0 \\
Cycles(4, n, 6) &= a(4) * n^4 + b(4) * n^3 + c(4) * n^2 + d(4) * n^1 + e(4) * n^0 \\
Cycles(5, n, 6) &= a(5) * n^4 + b(5) * n^3 + c(5) * n^2 + d(5) * n^1 + e(5) * n^0 \\
Cycles(6, n, 6) &= a(6) * n^4 + b(6) * n^3 + c(6) * n^2 + d(6) * n^1 + e(6) * n^0
\end{aligned}$$

Les différents coefficients, qui ici ont des valeurs "en dur", sont les valeurs de polynômes de variable m. Par exemple les valeurs $a(0)$, $a(1)$, $a(2)$, $a(3)$, ...sont les valeurs d'un polynôme $a(m)$, dont on va chercher l'équation.

Nous allons donc chercher les équations des polynômes passant par ces coefficients, afin de remplacer ces polynômes à une seule variable par un unique polynôme à deux variables.

Appliquons donc cette méthode sur les valeurs de l'algorithme : on cherche à calculer l'équation de $Cycles(m, n, 6)$ en connaissant ses valeurs pour plusieurs combinaisons de m et n. On va d'abord cherche l'équation du polynôme $Cycles(1, n, 6)$

Pour trouver l'équation de $Cycles(1, n, 6)$, on va interpoler les valeurs

$$\begin{gathered}
Cycles(1, 7, 6) = 0, \\
Cycles(1, 8, 6) = 0, \\
Cycles(1, 9, 6) = 0, \\
Cycles(1, 10, 6) = 0, \\
Cycles(1, 11, 6) = 0, \\
Cycles(1, n, 6) = 0n^4 + 0n^3 + 0n^2 + 0n + 0.
\end{gathered}$$

Pour trouver l'équation de $Cycles(2, n, 6)$, on interpole

$$\begin{gathered} Cycles(2,6,6) = 380, \\ Cycles(2,7,6) = 875, \\ Cycles(2,8,6) = 1736, \\ Cycles(2,9,6) = 3108, \\ Cycles(2,10,6) = 5160, \\ Cycles(2,n,6) = n^4 - \tfrac{35}{6}n^3 + \tfrac{21}{2}n^2 - \tfrac{17}{3}n. \end{gathered}$$

Pour trouver l'équation de $Cycles(3, n, 6)$, on interpole

$$\begin{gathered} Cycles(3,7,6) = 1630, \\ Cycles(3,8,6) = 4065, \\ Cycles(3,9,6) = 8526, \\ Cycles(3,10,6) = 15904, \\ Cycles(1,11,6) = 27252, \\ Cycles(3,n,6) = \tfrac{27}{4}n^4 - 27n^3 + \tfrac{137}{4}n^2 - 14n \end{gathered}$$

Pour trouver l'équation de $Cycles(4, n, 6)$, on interpole

$$\begin{gathered} Cycles(4,3,6) = 1630, \\ Cycles(4,4,6) = 7240, \\ Cycles(4,5,6) = 21200, \\ Cycles(4,6,6) = 17220, \\ Cycles(4,7,6) = 98530 \\ Cycles(4,n,6) = 24n^4 - 76n^3 + 74n^2 - 22n. \end{gathered}$$

Pour trouver l'équation de $Cycles(5, n, 6)$, on interpole

$$\begin{gathered} Cycles(5,2,6) = 380, \\ Cycles(5,3,6) = 4065, \\ Cycles(5,4,6) = 17220, \\ Cycles(5,5,6) = 49250, \\ Cycles(5,6,6) = 112800, \\ Cycles(5,n,6) = \tfrac{125}{2}n^4 - \tfrac{505}{3}n^3 + \tfrac{265}{2}n^2 - \tfrac{80}{3}n. \end{gathered}$$

Nous avons obtenu les équations $Cycles(m, n, 6)$ pour $1 \leq m \leq 5$, et on cherche maintenant l'équation du polynôme à deux variables $Cycles(m, n, 6)$ pour tout m et pour tout n :

$$\begin{gathered} Cycles(1,n,6) = 0n^4 + 0n^3 + 0n^2 + 0n + 0. \\ Cycles(2,n,6) = n^4 - \tfrac{35}{6}n^3 + \tfrac{21}{2}n^2 - \tfrac{17}{3}n. \\ Cycles(3,n,6) = \tfrac{27}{4}n^4 - 27n^3 + \tfrac{137}{4}n^2 - 14n. \\ Cycles(4,n,6) = 24n^4 - 76n^3 + 74n^2 - 22n. \\ Cycles(5,n,6) = \tfrac{125}{2}n^4 - \tfrac{505}{3}n^3 + \tfrac{265}{2}n^2 - \tfrac{80}{3}n. \\ Cycles(m,n,6) = a(m)n^4 + b(m)n^3 + c(m)n^2 + d(m)n^1 + e(m). \end{gathered}$$

Nous allons maintenant interpoler les coefficients de ces polynômes. Les coefficients des monômes de degré 0, valent tous 0, donc $e(m) = 0$.

Pour les coefficients des monômes de degré 1, on doit interpoler les valeurs
$d(1) = 0$, $d(2) = -\frac{17}{3}$, $d(3) = -14$, $d(4) = -22$, $d(5) = -\frac{80}{3}$
ce qui nous donne le polynôme $d(m) = \frac{1}{2}m^3 - \frac{13}{3}m^2 + \frac{23}{6}m$.

Pour les coefficients des monômes de degré 2, on doit interpoler les valeurs
$c(1) = 0$, $c(2) = \frac{21}{2}$, $c(3) = \frac{137}{4}$, $c(4) = 74$, $c(5) = \frac{265}{2}$
ce qui nous donne le polynôme $c(m) = \frac{11}{24}m^3 + \frac{31}{8}m^2 - \frac{13}{3}m$.

Pour les coefficients des monômes de degré 3, on doit interpoler les valeurs
$b(1) = 0$, $b(2) = -\frac{35}{6}$, $b(3) = -27$, $b(4) = -76$, $b(5) = -\frac{505}{3}$
ce qui nous donne le polynôme $b(m) = -\frac{1}{8}m^4 - \frac{5}{6}m^3 + \frac{11}{24}m^2 + \frac{1}{2}m$.

Pour les coefficients des monômes de degré 4, on doit interpoler les valeurs
$a(1) = 0$, $a(2) = 1$, $a(3) = \frac{27}{4}$, $a(4) = 24$ et $a(5) = \frac{125}{2}$
ce qui nous donne le polynôme $a(m) = \frac{1}{8}m^4 - \frac{1}{8}m^3$.

En combinant tous ces résultats, on obtient au final :

$$\begin{aligned} Cycles(m,n,6) \quad &= (1/2m^3 - 13/3m^2 + 23/6m)n + \\ &(11/24m^3 + 31/8m^2 - 13/3m)n^2 + \\ &(-1/8m^4 - 5/6m^3 + 11/24m^2 + 1/2m)n^3 + \\ &(1/8m^4 - 1/8m^3)n^4 \end{aligned}$$

qui une fois factorisé nous donne
$Cycles(m,n,6) = \frac{1}{24}mn(m-1)(n-1)(3m^2n^2 - 23mn - 12m - 12n + 92)$.

Étonnamment, on peut remarquer qu'ici, on a toujours $d(m) + c(m) + b(n) + a(n) = 0$. En effet, on a
$d(1) + c(1) + b(1) + a(1) = 0$
$d(2) + c(2) + b(2) + a(2) = -\frac{17}{3} + \frac{21}{2} - \frac{35}{6} + 1 = 0$
$d(3) + c(3) + b(3) + a(3) = -14 + \frac{137}{4} - 27 + \frac{27}{4} = 0$
$d(4) + c(4) + b(4) + a(4) = -22 + 74 - 76 + 24 = 0$
$d(5) + c(5) + b(5) + a(5) = -\frac{80}{3} + \frac{265}{2} - \frac{505}{3} + \frac{125}{2} = 0$.
Nous nous sommes rendus compte de ceci tardivement, et cette propriété n'a pas été étudiée ou exploitée durant cette thèse. Nous ne savons d'ailleurs pas si c'est toujours le cas.

En partant des valeurs retournées par l'algorithme et en les interpolant, on est parvenu à retrouver l'équation de $Cycles(m,n,6)$. Nous avons ici détaillé à la mon tous les calculs, mais pour obtenir ces équations pour de plus grandes valeurs de e nous avons utilisé une solution automatique. Dans la section suivante nous présentons ces équations pour $7 \leq e \leq 11$.

3.2.3 Calculs exacts et formules closes par interpolation

Nous proposons ici les formules closes pour $Cycles(m,n,e)$ pour $4 \leq e \leq 11$. Elles ont été obtenues avec la méthode présentée dans la section précédente. Nous les avons obtenues automatiquement, en utilisant un interpolateur codé en Java par nos soins, utilisant la classe des BigIntegers. Les coefficients pouvant être des nombres rationnels, nous avons créé une classe BigRationnal, ayant comme numérateur et dénominateur des BigInteger.

$$
\begin{aligned}
Cycles(m,n,4) =& \frac{1}{4}mn(m-1)(n-1)\\
Cycles(m,n,5) =& \frac{1}{4}mn(m-1)(n-1)(mn-4)\\
Cycles(m,n,6) =& \frac{1}{24}mn(m-1)(n-1)(3m^2n^2-23mn-12m-12n+92)\\
Cycles(m,n,7) =& \frac{1}{24}mn(m-1)(n-1)\\
&(m^3n^3-11m^2n^2-12m^2n-12mn^2+58mn+120m+120n-408)\\
Cycles(m,n,8) =& \frac{1}{96}mn(m-1)(n-1)\\
&(m^4n^4-14m^3n^3-24m^3n^2-24m^2n^3\\
&+52m^2n^2+399m^2n+180m^2+399mn^2\\
&+123mn-3900m+180n^2-3900n+8628)\\
Cycles(m,n,9) =& \frac{1}{1440}mn(m-1)(n-1)\\
&(3m^5n^5-50m^4n^4-120m^4n^3-120m^3n^4\\
&-40m^3n^3+2385m^3n^2+2700m^3n+2385m^2n^3\\
&+10925m^2n^2-28620m^2n-50400m^2+2700mn^3-28620mn^2\\
&-154108mn+478240m-50400n^2+478240n-806240)\\
Cycles(m,n,10) =& \frac{1}{2880}mn(m-1)(n-1)\\
&(m^6n^6-19m^5n^5-60m^5n^4-60m^4n^5\\
&-160m^4n^4+1185m^4n^3+2700m^4n^2+1185m^3n^4\\
&+11923m^3n^3-2337m^3n^2-79092m^3n-30240m^3+2700m^2n^4-2337m^2n^3\\
&-273373m^2n^2-34852m^2n+1380960m^2-79092mn^3-34852mn^2\\
&+4371692mn-8336160m-30240n^3+1380960n^2-8336160n+11611872)\\
Cycles(m,n,11) =& \frac{1}{21060}mn(m-1)(n-1)\\
&(m^7n^7-21m^6n^6-84m^6n^5-84m^5n^6\\
&-406m^5n^5+1365m^5n^4+6300m^5n^3+1365m^4n^5\\
&+25571m^4n^4+47481m^4n^3-200844m^4n^2-211680m^4n+6300m^3n^5+47481m^3n^4\\
&-593495m^3n^3-2417954m^3n^2+4402440m^3n+6350400m^3-200844m^2n^4-2417954m^2n^3\\
&+13528102m^2n^2+43539944m^2n-122915520m^2-211680mn^4+4402440mn^3+43539944mn^2\\
&-375739960mn+551315520m+6350400n^3-122915520n^2+551315520n-665058240)
\end{aligned}
$$

On peut remarquer que ces polynômes sont de la forme
$Cycles(m,n,e) = \frac{1}{4(e-3)!}mn(m-1)(n-1)((mn)^{e-4}+O((mn)^{e-5}))$

3.2.4 Comparaisons entre théorie et simulations

Dans cette partie, nous confrontons les résultats présentés précédemment avec des simulations.

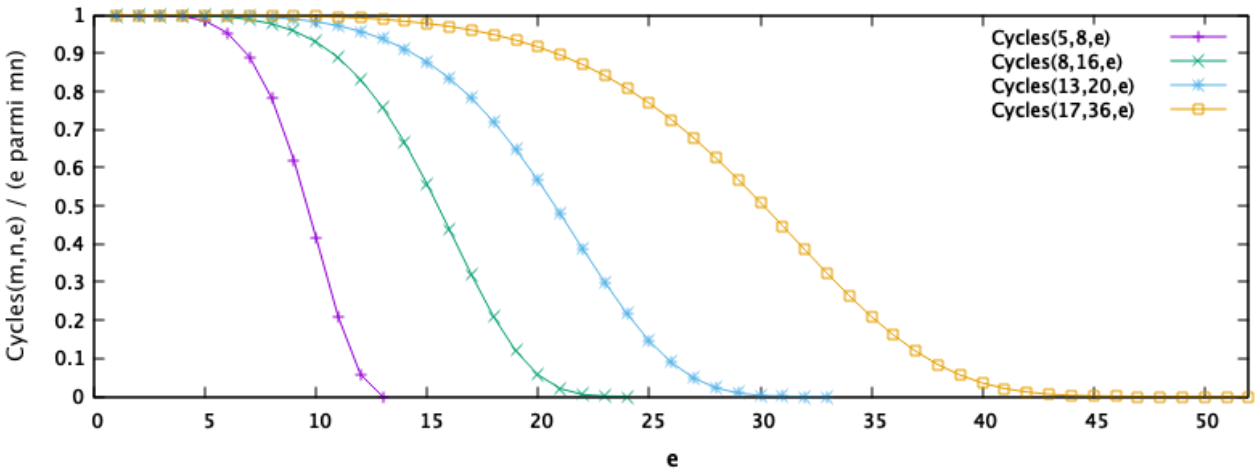


FIGURE 3.16 – Résultats obtenus en utilisant $1 - Cycles(m,n,e)/\binom{mn}{e}$. Les valeurs de $Cycles(m,n,e)$ sont celles retournées par l'algorithme de Rebecca Stones.

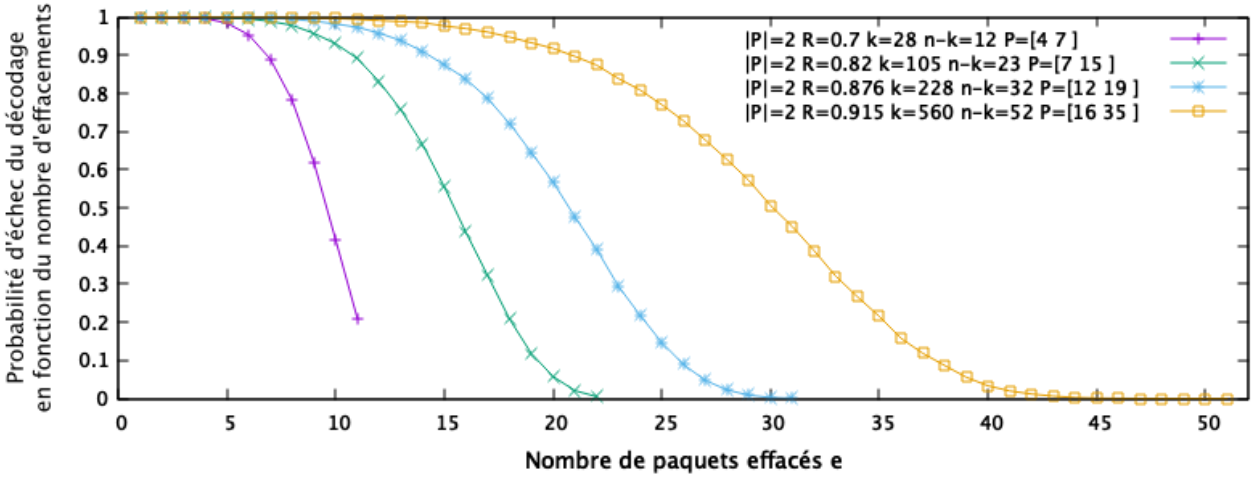


FIGURE 3.17 – Résultats obtenus par simulations

Les courbes se superposent parfaitement, ce qui tend à confirmer que cette méthode est la bonne pour estimer la capacité de correction du code. Avec les équations polynomiales de $Cycles(m,n,e \leq 11)$ nous pouvons calculer la probabilité d'échec du décodage précisément même pour de très grandes dimensions. Cependant en pratique nous n'utiliserons pas comme paramètre un ensemble à 2 éléments car les capacités de correction sont bien meilleures quand $|P| > 2$, plutôt de l'ordre de 5 ou 6. Dans la section suivante nous présentons un protocole de transmission basé sur l'utilisation d'un paramètre P à deux élements.

3.3 Forêts couvrantes et application aux transmissions bidirectionnelles

Dans cette partie, nous étudions la probabilité de pouvoir décoder totalement un ensemble d'effacements ou non, tout en estimant le nombre de paquets restant à décoder après décodage si celui-ci n'a pas abouti, là-encore dans le cas ou $|P| = 2$. Pour ce faire, nous allons modifier les graphes de Tanner représentant l'algorithme de décodage, en leur appliquant une contraction d'arêtes. Rappelons que pour $P = [P_0, P_1]$ on peut représenter notre système d'équations (ou notre algorithme de décodage) par un graphe de Rook de dimsnsions $(m = P_0 + 1, n = P_1 + 1)$. Dans cette partie, nous allons laisser de coté la notation $P = [P_0, P_1]$ pour nous concentrer sur les dimensions du graphe de Rook, que nous noterons donc (m, n) : il faudra donc nous souvenir que nous travaillons sur un graphe de Rook de dimensions (m, n), obtenu avec $P = [P_0 = m - 1, P_1 = n - 1]$. On peut d'ailleurs voir que dans les figures 3.16 et 3.17, nous comparons bien les résultats obtenus avec l'algorithme de Rebecca Stones pour $Cycles(5, 8, e)$ avec les performances du code obtenu pour $P = [4, 7]$, ou encore $Cycles(17, 36, e)$ avec $P = [16, 35]$.

Si on a choisi $P = [P_0, P_1] = [m - 1, n - 1]$, cette opération nous donne alors un nouveau graphe, le sous-graphe d'un graphe biparti complet de taille $m + n$ (graphe biparti complet $K_{m,n}$), que nous appellerons graphe de Tanner contracté. En effet, rappelons que nous choisissons un ensemble P de nombres premiers entre eux, ici $P = [P_0, P_1]$, et que le graphe de Rook que nous obtenons est un graphe de Rook de dimensions $(m + 1, n + 1)$: nous avons $m + n + 2$ noeuds contraintes dans notre graphe de Tanner.

Pour contracter notre graphe de Tanner, nous procédons de la manière suivante : les sommets du graphe contractés seront tous les noeuds contraintes de notre graphe de Tanner. On peut les ranger en deux ensembles H et D : H les noeuds contraintes du "haut" (rangés horizontalement), et D les noeuds contraintes de "droite" (rangés verticalement). Il existe une arête entre La figure 3.18 représente l'étape de contraction d'arête que nous appliquons sur notre graphe de Tanner pour obtenir ce graphe de Tanner contracté.

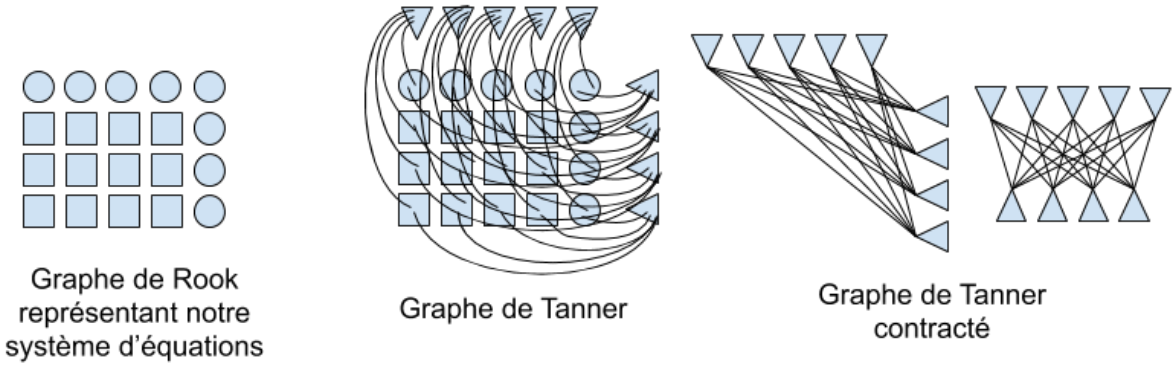


FIGURE 3.18 – En appliquant une contraction d'arête sur les noeuds contraintes de nos graphes de Tanner, on obtient le sous-graphe d'un graphe biparti complet. Les sommets carrés correspondent à des paquets sources, les sommets ronds à des paquets codes, et les sommets triangles les noeuds contraintes du graphe de Tanner. Dans le graphe contracté, il n'y a plus que des noeuds contraintes (triangles), reliés entre eux si il existe un chemin entre eux dans le graphe de Tanner non-contracté.

Il existe un cycle dans ce graphe si et seulement si il existe un cycle dans le graphe de Tanner original, donc la présence d'un cycle dans ce graphe produit elle aussi un échec du décodage. Les figures ci-dessous 3.19 et3.20 représentent les deux situations possibles : la première lorsque le graphe de Tanner contracté contient un cycle, et la seconde lorsqu'il ne contient pas de cycle. Ces situations sont équivalentes à l'échec ou la réussite du décodage : le graphe de Tanner (non contracté) contient ou non un cycle.

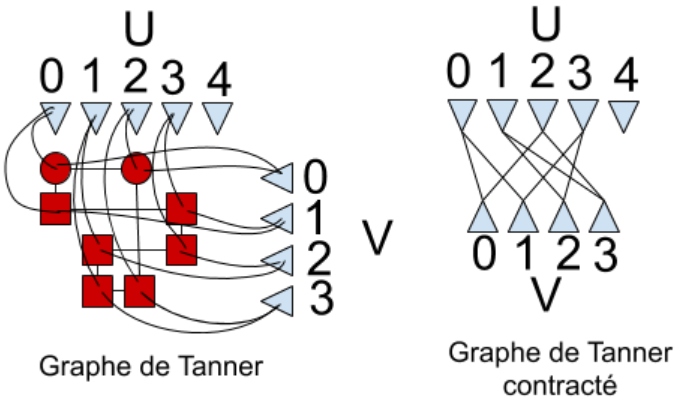


FIGURE 3.19 – Le graphe de Tanner contracté induit par notre de graphe de Tanner contient un cycle, le décodage échoue. Le sommet 0 de U est relié au sommet 0 de l'ensemble V car le sommet rond en haut à gauche (correspondant à un paquet code) a été effacé, et qu'il relie ces deux sommets.

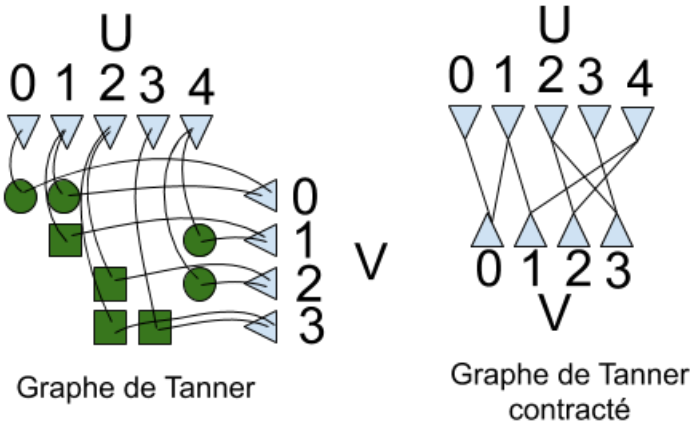


FIGURE 3.20 – Le graphe de Tanner contracté induit par notre de graphe de Tanner ne contient pas de cycle, le décodage fonctionne. A la première itération, on va pouvoir décoder en se servant des sommets contraintes 0 et 3 de U, puis 0 et 3 de V, puis 1 et 2 de U, et enfin 1 et 2 de V.

Nous nous servons de cette représentation de nos graphes de Tanner afin d'adapter notre code au protocole de type HARQ, qui, en plus d'utiliser un code correcteur pour fiabiliser la transmission, autorise le récepteur à demander la retransmission de paquets non-décodés. Ainsi, si le récepteur n'est pas parvenu à décoder un certain nombre de e paquets, il n'a pas à demander

la transmission de tous ces e paquets, mais uniquement un sous-ensemble. Ce sous-ensemble, bien choisi, peut alors s'avérer petit par rapport à e : on choisira le plus petit sous-ensemble de ces e paquets tel que si on reçoit cet ensemble, on peut décoder les e paquets totalement. On verra aussi que lorsque l'émetteur aura à retransmettre ces paquets, il pourra en plus ajouter y des paquets de redondance pour s'assurer que le récepteur saura les décoder si ils ont été effacés. Ainsi, notre protocole de transmission devient un protocole itératif, pour lequel chaque itération apporte de nouveaux paquets, susceptibles de permettre le décodage de l'itération précédente, jusqu'à remonter à la première itération.

Pour trouver le plus petit nombre d'arêtes permettant de supprimer tous les cycles du graphe, nous allons calculer une forêt couvrante de poids maximal, puis retirer ses arêtes du graphe. Les arêtes restantes formeront alors l'ensemble d'arêtes de coût minimal tel que si on retire cet ensemble du graphe, on obtient un graphe sans cycle (on obtient la forêt couvrant de poids maximal). Rappelons que les paquets effacés que nous souhaitons décoder sont les arêtes de ce graphe. Pour enlever des arêtes, nous allons simplement demander la retransmission de ces paquets : nous modifions donc notre cas d'utilisation.

Nous proposons ici de calculer la forêt couvrante de poids maximal de notre graphe, sans préciser ce que nous entendons par "poids". Le poids d'une arête (et donc d'un paquet à retransmettre) peut par exemple correspondre au coût de retransmission d'un paquet. Pour le cas d'utilisation qui nous concerne, à savoir la transmission de données via une diode réseau, nous pourrions donner à tous les paquets le même poids, et toutes les forêts couvrantes auraient alors le même poids.

Cependant pour d'autres cas d'utilisation, on pourrait aussi donner des poids plus importants aux paquets codes qu'aux paquets sources, si on souhaite demander en priorité la retransmission de ces derniers. On pourrait à l'inverse préférer la retransmission de paquets codes à celle de paquets sources, si les paquets sources contiennent de l'information confidentielle dont on souhaite qu'elle transite le moins possible sur le canal de transmission.

On verra qu'il est tout à fait possible d'utiliser ce protocole de transmission, reposant sur le calcul d'une forêt couvrante de poids maximal, pour un paramétrage P ayant plus de deux éléments : on cherchera là-aussi une forêt couvrante de poids maximal dans son graphe de Tanner contracté, exactement de la même manière que pour $|P| = 2$. Cependant, on verra que dans le cas ou $|P| = 2$, il est possible de prédire la quantité d'information moyenne à redemander à chaque itération. Pour cela, nous calculerons la taille moyenne d'une forêt couvrante de nos graphes de Tanner, qui dépend de son nombre de composantes connexes, en appliquant deux théorèmes énoncés par Kalugin [27] et Saltykov [52].

Précisons néanmoins un dernier point : nous cherchons ici à calculer une forêt couvrante de poids maximal, et d'enlever ses arêtes du graphe. Ainsi, on obtient un autre graphe, dont les arêtes correspondent aux paquets dont on va demander la retransmission (et qui du coup auront un poids total faible, correspondant à un coût de retransmission total faible). En pratique, le coût de retransmission est le même quel que soit le paquet retransmis, et cette notion de poids ou de coût peut finalement prêter à confusion. Le principe reste le même si on donne à tous les paquets le même poids, et la notion de forêt couvrante de poids maximal n'a alors plus vraiment de sens : on cherchera uniquement une forêt couvrante.

Récapitulons donc : le récepteur se retrouve avec un graphe de Tanner contracté qui contient des cycles (et n'est donc pas une forêt), dont les sommets sont les noeuds contraintes, et les arêtes les paquets effacés qu'il n'est pas parvenu à décoder.

Il cherche à transformer ce graphe en une forêt, en supprimant certaines arêtes (ce qui se

fait concrètement en demandant la retransmission de paquets). Il va d'abord chercher la forêt couvrante (et de poids maximal si les arêtes/paquets ont des coûts différents), dont les arêtes seront les paquets qu'il ne va pas redemander. Il va alors demander la transmission de toutes les arêtes du graphe contracté qui ne sont pas dans la forêt couvrante. En les recevant, il pourra les retirer du graphe contracté, qui sera alors une forêt, correspondant à un graphe de Tanner contracté décodable.

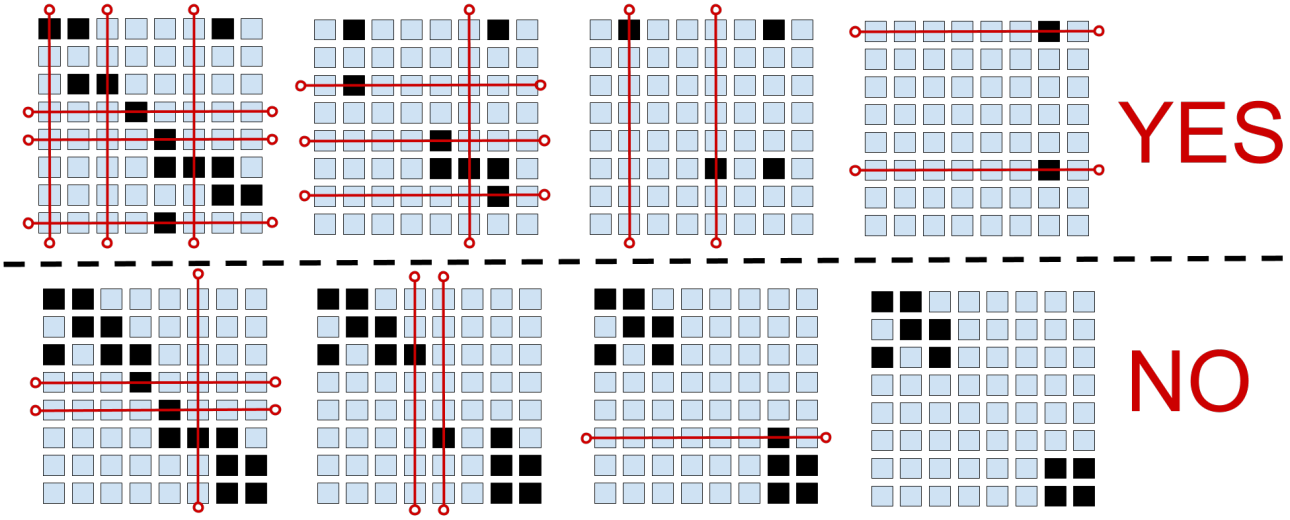


FIGURE 3.21 – Visualisation du décodage itératif : en haut, le décodage aboutit à un succès après 4 itérations car le graphe ne contient pas de cycle. En bas, le décodage échoue car le graphe contient 2 cycles : le récepteur devrait demander la retransmission d'au minimum 2 paquets pour que le décodage puisse reprendre et terminer sur un succès : un paquet parmi les 6 paquets formant un cycle en haut à gauche, et un parmi les 4 paquets formant un cycle en bas à droite. Il n'existe pas d'ensemble de moins de 2 paquets qui casse tous les cycles du graphe. Pour faciliter la lecture, nous considérons ici que tous les paquets sont équivalents (on oublie qu'il y a des paquets sources et codes), et on les représente tous par des carrés

.

3.3.1 Graphes bipartis, Forets couvrantes et retransmission minimale

Supposons que nous ayons une mauvaise configuration qui nous donne un graphe de Tanner contracté G, et qu'on puisse demander la retransmission de certaines de ses arêtes (ou paquets).

Comme on l'a évoqué, on peut ajouter un coût à chacun de nos paquets, qui représenterait son coût de retransmission, bien que concrètement nous ne le ferons pas dans notre cas d'utilisation. Le coût total d'une retransmission est alors la somme des coûts des paquets retransmis.

En calculant une forêt couvrante de poids maximal, qu'on notera $Foret_{max}$, on va pouvoir calculer le plus petit ensemble d'arêtes à retirer de G pour obtenir un graphe G acyclique, qu'on va noter $Retransmission_{min}$. Le coût total de la retransmission de $Retransmission_{min}$ sera alors la somme des coûts de toutes ses arêtes. La figure 3.23 récapitule les différents graphes que nous venons de présenter : le graphe (1) est le graphe de Rook correspondant à nos effacements.

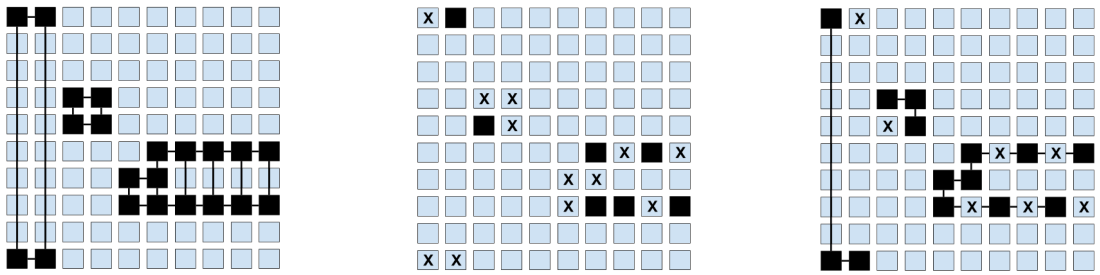


FIGURE 3.22 – Le graphe de gauche est le graphe original G. Le graphe du milieu est le graphe correspondant à $Retransmission_{min}$. Le graphe de droite est la la forêt couvrante maximale $Foret_{max}$

Pour calculer la forêt couvrante maximale $Foret_{max}$, on a recours à l'algorithme de Kruskal : on commence par trier les arêtes selon leur coût, puis on construit itérativement notre $Foret_{max}$ en y ajoutant l'arête de plus petit coût si elle ne forme pas de cycle dans le graphe. Cet algorithme bien connu a une complexité en $O(E \log(E)$, E étant le nombre d'arêtes de G.

Algorithm 6 Algorithme de Kruskal retournant une forêt de poids maximal.

Require: Algorithme calculant une forêt de poids maximale dans le graphe G

```
1 : function Kruskal(G = (V, E))
2 :     Foret_max ← (V, E_max = ∅)
3 :     Trier les arêtes de E selon leur coût dans l'ordre croissant.
4 :     while E n'est pas vide do
5 :         a ← La première arête de E
6 :         E = E \ a
7 :         if a ne crée pas de cycles dans Foret_max then
8 :             E_max = E_max ∪ a
    return Foret_max
```

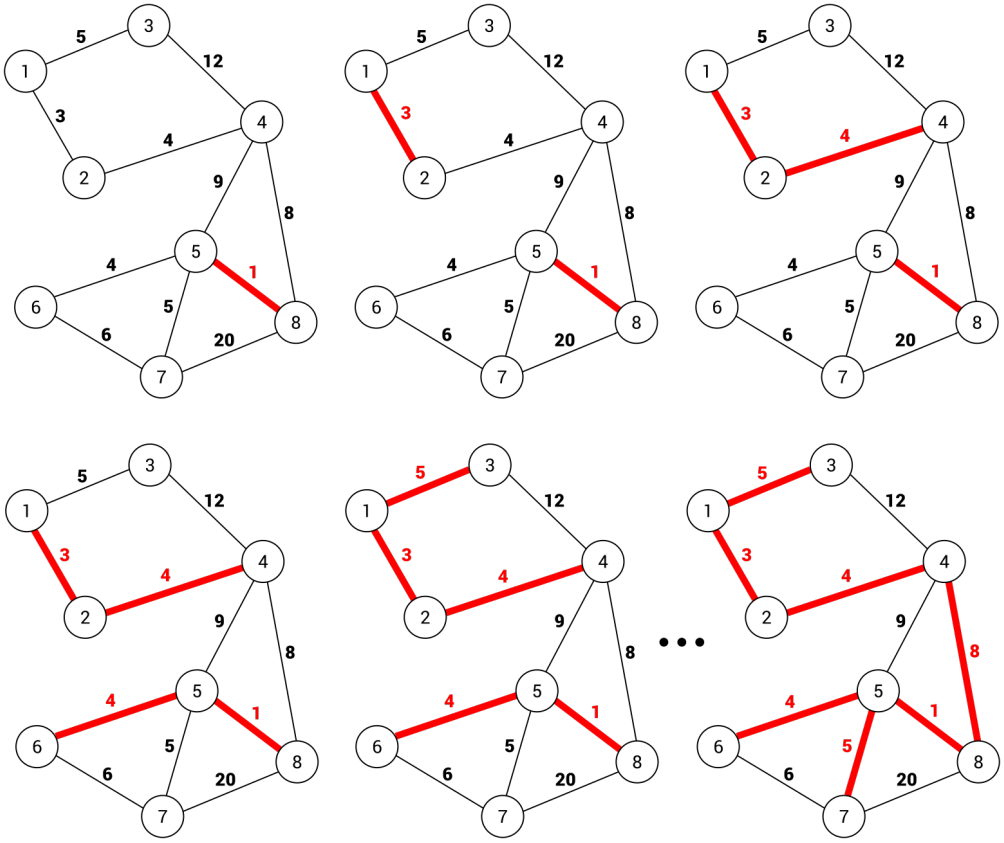


FIGURE 3.23 – Exemple de l'exécution de l'algorithme de Kruskal pour trouver sa forêt couvrante de poids minimal. En effectuant un prétraitement sur le graphe (inversant les poids des arêtes), on peut calculer une forêt de poids maximal.

Nous allons nous poser deux questions :

— En connaissant uniquement les dimensions (m, n) de notre graphe de Rook correspondant à un blocage du décodage, et du nombre d'effacements e (effacements se réalisant de manière équiprobable), est-il possible d'estimer la taille moyenne de $Retransmission_{min}$ A contrario, est-il possible d'estimer la taille de la forêt couvrante maximale de notre graphe en fonction de m, n et e.

— Quelle est la probabilité qu'on n'ait aucun paquet à retransmettre ? Nous avons déjà abordé la question dans la section précédente, notamment avec l'algorithme de Stones.

Taille moyenne de $Foret_{max}$ et $Retransmission_{min}$

Pour tout graphe G à e sommets constitué de N_{cc} composantes connexes, toute forêt couvrante de G a exactement $(e - N_{cc})$ arêtes.

Soit G un graphe de Rook de dimensions (m, n) correspondant à un système d'équations que l'on souhaite résoudre (ou des paquets que l'on souhaite décoder).

Le graphe de Tanner contracté de G, que l'on notera T est un graphe biparti dont les sommets sont séparés en 2 sous-ensembles $V = V_L \cup V_R$. On notera N_L le nombre de sommets de V_L et N_R celui de V_R. Concrètement, ces valeurs correspondent respectivement aux lignes (V_L) et aux colonnes (V_R) dans notre graphe G qui contiennent au moins un paquet.
En notant N_{cc} le nombre de composantes connexes de $T = (V, E)$, toute forêt couvrante de T $Foret_{max} = (V_{max}, E_{max})$ aura donc $|E_{max}| = N_L + N_R - N_{cc}$ arêtes.

$$|E_{max}| = N_L + N_R - N_{cc} \tag{3.2}$$

avec N_L et N_R le nombre de sommets de V_L et V_R, et N_{cc} le nombre de composantes connexes.
Estimer la taille d'une forêt couvrante de T nécessite donc d'estimer le nombre de composantes connexes en moyenne dans T, ainsi que les valeurs N_L et N_R, en fonction de m, n et e.

Rappelons qu'on a $|E| = e = N_L + N_R$ arêtes dans G, qui correspondent à nos paquets effacés. $Retransmission_{min} = (V_{min}, E_{min})$ contenant toutes les arêtes de G qui ne sont pas, $E_{min} = E \setminus E_{max}$, on obtient naturellement le nombre d'arêtes de $Retransmission_{min}$,

$$|E_{min}| = |E| - [E_{max}| = e - (N_L + +N_R - N_{cc}) = e - N_L - N_R + N_{cc} \tag{3.3}$$

avec e le nombre d'arètes, N_L et N_R le nombre de sommets de V_L et V_R, et N_{cc} le nombre de composantes connexes.

La figure ci-dessous 3.24 représente différents graphes G correspondant à des échecs de décodage, et mettent en évidence cette dernière équation. Les sommets en vert correspondent aux sommets de $Foret_{max}$, et vérifient bien, et les sommets en jaune correspondent aux sommets de $Retransmission_{min}$.

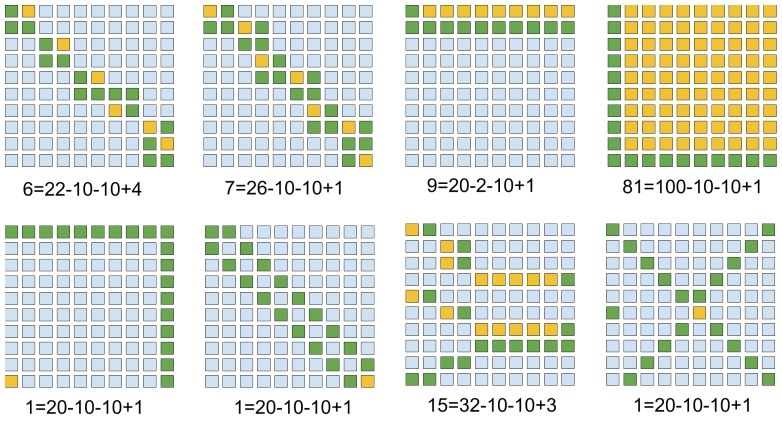


FIGURE 3.24 – Des graphes de Rook correspondant à des échecs du décodage (leur graphe de Tanner contracté contient un cycle) et les $Foret_{max}$ (en vert) et $Retransmission_{min}$ (en jaune) associées. Rappelons qu'ici, les paquets sont des sommets, mais que dans le graphe contracté leur correspondant, il s'agirait d'arêtes. Les paquets sont ici tous représentés par des carrés, mais il y a bien des paquets sources et codes.

On remarque que leur cardinal correspond bien aux équations :

$$|E_{max}| = N_L + N_R - N_{cc} \tag{3.4}$$

$$|E_{min}| = |E| - [E_{max}| = e - [E_{max}| = e - (N_L + N_R - N_{cc}) = e - N_L - N_R + N_{cc} \tag{3.5}$$

Pour le graphe tout en haut à gauche par exemple, on a bien $e = 22$ sommets dans G (donc $|E| = 22$ arêtes dans T), $N_L = 10$ lignes occupées par un sommet (donc N_L sommets dans V_L), $N_R = 10$ colonnes occupées par un sommet (donc N_R sommets dans V_R), et $N_{cc} = 4$ composantes connexes, ce qui nous donne une $Retransmission_{min}$ de taille $e - N_L - N_R + N_{cc} = 22 - 10 - 10 + 4 = 6$.

Cette équation permet donc de calculer exactement $Foret_{max}$ pour un graphe de Tanner contracté T donné. On peut donc estimer, quand on a subi e effacements, le nombre minimal d'arêtes à retransmettre pour que le décodage termine $|E_{min} = e - m - n + N_{cc}|$ si on parvient à estimer le nombre moyen de sommets de V_L N_L, le nombre moyen de sommets de V_R N_R et le nombre moyen de composantes connexes N_{cc} du graphe G. Nous allons présenter les espérances de ces valeurs en fonction de m, n et e.

Etude de l'espérance de N_L et de N_R

Tout d'abord, pour N_L (ou N_R), la probabilité que la ligne L_i (ou la colonne C_j) de notre graphe de Rook ne contienne pas d'effacements pour e effacements équirépartis est :

$$\Pr(L_i = 0|e) = \frac{\binom{mn-n-1}{e}}{\binom{mn}{e}} \tag{3.6}$$

Ainsi, on peut déterminer la probabilité que le nombre de lignes N_L (ou le nombre de colonnes N_R) vaille k :

$$\Pr(N_L = k|e) = \binom{m+1}{k} \left(1 - \frac{\binom{mn-n-1}{e}}{\binom{mn}{n_e}}\right)^k \left(\frac{\binom{mn-n-1}{e}}{\binom{mn}{e}}\right)^{m+1-k} \tag{3.7}$$

Enfin, on obtient

$$\mathbb{E}(N_C|e) = \sum_{j=0}^{m} \mathbb{E}(C_j|e) = (m+1) * \left(1 - \frac{\binom{mn-n-1}{e}}{\binom{mn}{e}}\right) \tag{3.8}$$

$$\mathbb{E}(N_R|e) = (n+1) * \left(1 - \frac{\binom{mn-m-1}{e}}{\binom{mn}{e}}\right) \tag{3.9}$$

La figure 3.25 ci-dessous représente la progression de N_L ou N_R pour $m = n = 10$ en fonction de e. Il ne s'agit pas d'une simulation mais simplement de l'équation de $\mathbb{E}(N_C|e)$ appliquée à $0 \leq e$.

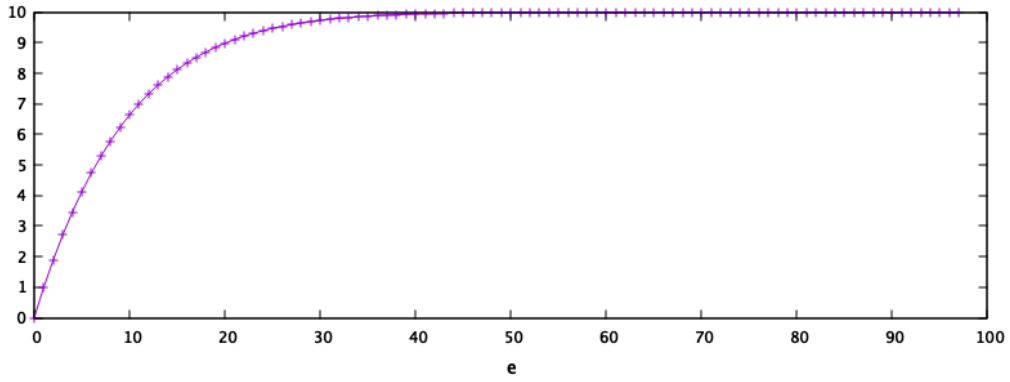


FIGURE 3.25 – Le nombre moyen de sommets de V_L dans le graphe de Tanner contracté, pour $m = n = 10$.

Etude de N_{cc} par les théorèmes de Kalugin et Saltykov

Pour estimer N_{cc}, nous avons recours à deux théorèmes proposés par Kalugin et Saltykov dans [27] et [52].

Theorem 1 (Kalugin ('94))**.** Pour $n, m, e \to \infty$,

— Si $\frac{e^2}{(n+1)(m+1)} \to 0$, alors :

$$\Pr(N_{cc} = n + m + 2 - e) \to 1 \tag{3.10}$$

— Si $e = \Theta(n)$, $e = \Theta(m)$, et $\frac{e^2}{(n+1)(m+1)} \to c \in]0; 1[$, alors :

$$\forall k \in [\![0; e]\!], P(N_{cc} = n + m + 2 - e + k) \to \frac{\lambda^k exp(-\lambda)}{k!}, avec \lambda := -\frac{ln(1-c) + c}{2} \tag{3.11}$$

Theorem 2 (Saltykov ('95))**.** *Pour* $(n + m) \to \infty$*, si* $n_e = (n + 1) * ln(n + m) + O(n)$*, alors :*

$$\Pr(N_{cc} = 1|e) \to 1 \tag{3.12}$$

La figure 3.27 ci-dessous, une simulation, représente le nombre moyen de composantes connexes (N_{cc}), pour $m = 8$ et $n = 8$, en fonction du nombre d'effacements e.

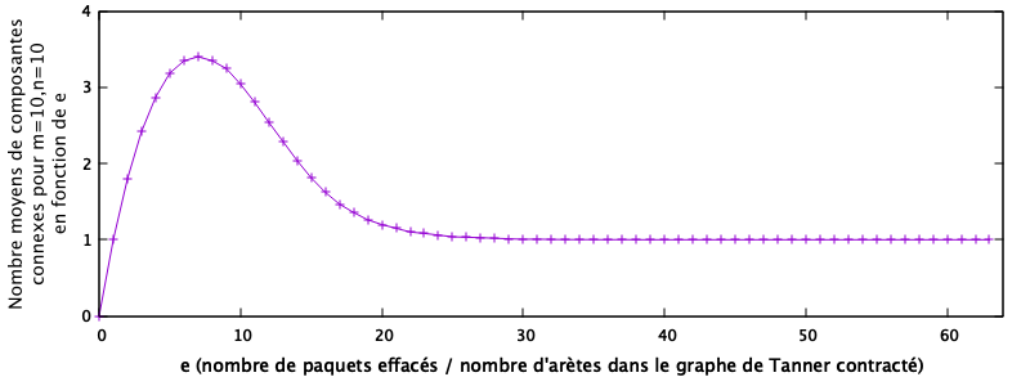


FIGURE 3.26 – Nombre moyen de composantes connexes dans le graphe de Tanner contracté T en fonction du nombre d'effacements e pour $m = 8$ et $n = 8$.

La figure 3.27 est intéressante, et nous proposons de faire quelques remarques. Tout d'abord, il est normal que pour $e = 0$ effacements on ait bien $N_{cc} = 0$, et pour $e = 1$ on a $N_{cc} = 1$. Puis le nombre moyen de composantes connexes N_{cc} progresse, à mesure que e progresse, jusqu'à atteindre un pallier vers $e = 8$. Cette situation correspond au moment ou les effacements sont tous sur des lignes ou des colonnes séparés (il y a beaucoup de lignes et de colonnes et peu d'effacements, donc N_{cc} est proche de e. Une fois ce palier dépassé, le nombre moyen de composantes connexes N_{cc} se met à diminuer : à mesure qu'on ajoute des effacements (et donc des arêtes) dans notre graphe, on le rend de plus en plus connexe (et N_{cc} tend alors vers 1. Enfin, quand $e \geq (m-1)(n-1)+1$, on ne peut plus avoir qu'une seule composante, et on a alors $N_{cc} = 1$.

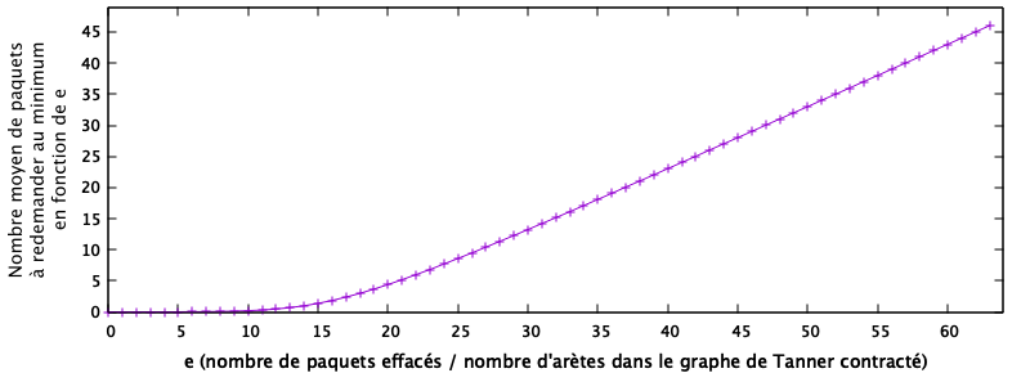


FIGURE 3.27 – Simulations du nombre moyen de paquets $|E_{min}|$ à redemander pour débloquer le décodage en fonction du nombre d'effacements e pour $m = n = 8$

3.3.2 HARQ - Hybrid Automatic Repeat reQuest

Dans cette partie, nous nous appuyons sur les résultats présentés précédemment pour proposer un protocole de transmission adapté aux transmissions bidirectionnelles. Durant ma thèse, j'ai eu l'occasion de travailler avec Pierre Meyer, à l'époque en Master 1 à l'Ecole Normale de Lyon. Nous avons proposé un article, qui n'a pas été publié, mais est disponible en ligne [39].

Nous décrivons et simulons un protocole de transmission basé sur le code correcteur Fauxtraut présenté précédemment, que nous utilisons sur une transmission qui cette fois-ci dispose d'un canal retour : le récepteur peut demander la retransmission de certains paquets. C'est ce que proposent les protocoles de type HARQ, qui consistent à ajouter des paquets de redondance lors de la transmission d'un ensemble de paquets, en plus de l'utilisation d'acquittements et de non-acquittements permettant la retransmission de paquets (à la manière des protocoles de type ARQ comme TCP). Ainsi, avant de demander la retransmission de paquets effacés, le récepteur va d'abord essayer de les décoder. Le protocole HARQ est utilisé sur le réseau mobile, notamment les réseaux 3G+ et les réseaux LTE.

Ce type de protocoles associant un code correcteur à une transmission avec canal retour, est notamment utile lorsque le ping de la transmission est grand, et qu'un grand nombre de réémissions est nécessaire, ce qui devient contraignant et allonge la durée totale de la transmission (le temps total de la transmission augmentant du ping et du nombre d'itérations). La figure 3.28 ci-dessous présente le fonctionnement d'un protocole de type HARQ. A l'itération 0 l'émetteur transmet 3 paquets de redondance calculés grâce à un code correcteur en plus des 6 paquets sources. Le récepteur, qui a subi 4 effacements sur les 9 paquets attendus (2 sur les paquets sources et 2 sur les paquets de redondance) parvient à décoder un paquet source, mais pas le deuxième. Il redemande alors la réémission de ce dernier. A l'itération 1, l'émetteur transmet le paquet source, et un autre paquet de redondance. Le récepteur, qui a subi 1 effacement, peut décoder le paquet effacé, et la transmission est terminée en deux itérations. De nombreuses analyses plus ou moins anciennes des protocoles de type HARQ ont été réalisées, comme dans [13], [64] ou encore [30].

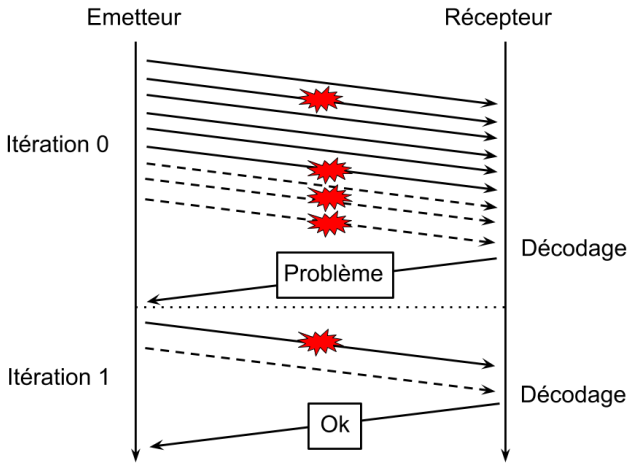


FIGURE 3.28 – Transmission avec acquittements de 6 paquets, avec utilisation d'un code correcteur à chaque itération : 2 itérations ont été nécessaires. La transmission des paquets sources est représentée par des traits pleins, et la transmission des paquets de redondance en pointillés.

Dans l'exemple proposé dans la figure 3.28, nous ne montrons que deux itérations : il est tout à fait possible que la transmission se déroule sur beaucoup plus d'itérations.

Un des principaux avantages d'un protocole de type HARQ est que, si il y a effectivement des effacements qui se produisent lors de la transmission, il y a une chance non-nulle qu'à la première itération, le récepteur parvienne à les décoder, et dispose donc de l'information source

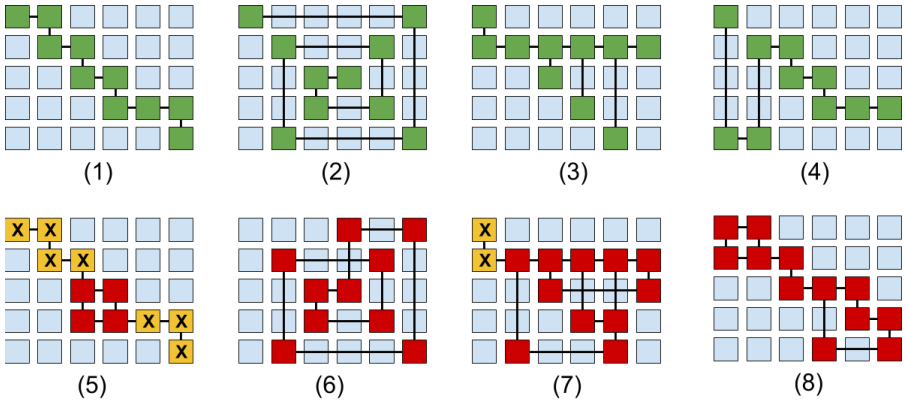


FIGURE 3.29 – Configurations d'effacements : *Bonnes configurations* (1,2,3,4) sans Stopping-set, donc totalement décodables. *Mauvaises configurations d'effacements* (5,6,7,8) contenant des Stopping-sets (en rouge) ne sont pas totalement décodables : les paquets jaunes le sont, mais pas les paquets rouges car ils sont sur un cycle. Tous les paquets sont représentés par des carrés, mais là encore on a toujours des paquets sources et des paquets codes.

.

sans avoir recours à une retransmission. Dans le cas d'un protocole ARQ, qui n'utiliserait donc pas de code correcteur, l'effacement d'un seul paquet à une itération impose le récepteur à demander la réémission de ce paquet. D'une manière générale, un protocole HARQ permet donc de réduire le nombre d'itérations nécessaires, au prix de la transmission de paquets de redondance, qui pourront s'avérer inutiles lorsqu'on n'aura aucun effacement à corriger.

3.3.3 Protocole HARQ et Fauxtraut

Comme nous l'avons fait jusqu'à présent, supposons que l'expéditeur souhaite transmettre k paquets au destinataire, k étant un paramètre du protocole. Les paquets sont là-encore des paquets de type UDP, c'est à dire qu'ils disposent d'un Checksum permettant au récepteur de vérifier si le paquet lui est parvenu en bon état, sinon il est considéré comme effacé : le code correcteur est exactement le même que celui présenté jusqu'ici.

Le principe général du protocole que nous présentons ici est le suivant : l'émetteur calcule des paquets de redondance qu'il va transmettre en plus des k paquets sources, exactement de la même manière que celle présentée jusqu'ici. Si des paquets sources ont été effacés, le récepteur va tout d'abord essayer de les décoder. Supposons qu'à la fin de l'algorithme de décodage, il reste un ensemble E de d paquets non-décodés (sources et/ou codes) : le récepteur va alors demander la retransmission de certains de ces paquets. Ce qui est alors intéressant, c'est que le récepteur ne va pas avoir à redemander la retransmission des d paquets qu'il n'est pas parvenu à décoder, mais uniquement un sous-ensemble de ces paquets, le plus petit possible.

En effet, on a vu que l'algorithme de décodage pouvait aboutir à un échec lorsqu'il existe un cycle dans le graphe de Tanner associé à notre code, et qu'à contrario, le décodage se faisait sans problème en l'absence de cycles. La figure 3.29 présente différents graphes acycliques ou cycliques.

Le protocole que nous présentons ici repose sur l'idée que le récepteur va chercher à demander le plus petit nombre de paquets, tels qu'en les recevant, il pourra supprimer les cycles du graphe

de Tanner, et reprendre le décodage de l'itération précédente. Or comme nous l'avons vu, il est possible de calculer rapidement le plus petit ensemble de paquets à redemander, en calculant une forêt couvrante dans le graphe de Tanner de notre code, et de ne considérer que les paquets qui n'appartiennent pas à cette forêt.

Rappelons cependant que notre code correcteur nécessite comme paramètre P, un ensemble de nombres premiers entre eux, qui détermine la dimension maximale $k \leq PPCM(P)$ et la longueur $N = \Sigma(P)$ du code, et a un impact sur sa complexité, sa consommation mémoire et sa capacité de correction. Jusqu'à présent, P pouvait être de cardinal quelconque, mais dans cette partie nous ne considérerons que des ensembles P à deux éléments : la capacité de correction est moins optimale que pour un ensemble P ayant plus d'éléments, mais cette limitation nous permet d'étudier son graphe de Tanner de manière intéressante.

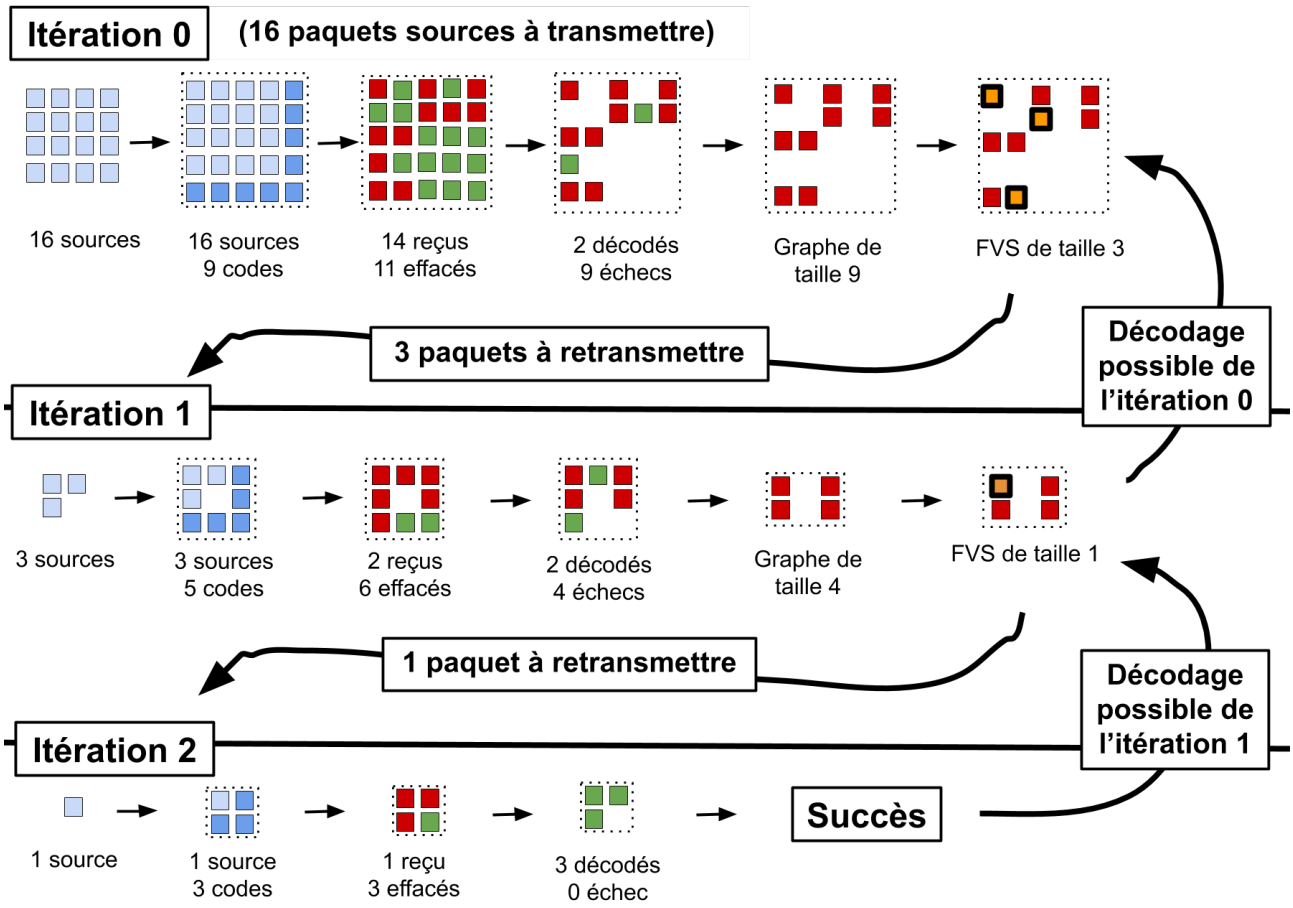


FIGURE 3.30 – Principe général de notre protocole HARQ. Lorque le récepteur ne parvient pas à décoder un bloc, il n'a pas besoin de redemander tous les paquets non-décodés, mais peut se contenter de ne demander que ceux qui "cassent" les cycles. Une fois ceux-ci reçus, il pourra alors décoder tous les autres. Lorsque l'émetteur retransmet des paquets, il recalcule des paquets codes puis les transmet.

La figure 3.30 présente donc le fonctionnement de ce protocole. Lorsqu'à une itération i, on n'est pas parvenu à décoder tous les paquets, on demande la retransmission de ces paquets, qui vont eux-mêmes être protégés par un code correcteur. Lorsqu'on sera parvenu à une certaine

itération j à recevoir ou décoder tous les paquets, on pourra alors utiliser ceux-ci pour décoder des paquets de l'itération $(j-1)$, puis ceux-là nous serviront à décoder l'itération $(j-2)$, ...et ainsi de suite jusqu'à remonter à l'itération 0 d'origine.

A chaque itération, on encode donc les k paquets redemandés par le récepteur à l'itération précédente, en les encodant avec notre code. A chaque itération, il faut donc choisir deux nouveaux éléments pour P en fonction du nombre k de paquets à retransmettre. Notre solution ici a été de prendre $m = n = \sqrt{(k)}\rceil$, afin d'avoir $k \leq mn$. Jusqu'à présent, nous imposions à P de ne contenir que des nombres premiers entre eux, mais ici nous encodons nos paquets avec un code rectangulaire. Si nous désirions réellement utiliser notre code Fauxtraut pour encoder nos paquets, il nous faudrait choisir par exemple 2 diviseurs de k, ou deux nombres m et n premiers entre eux, proches l'un de l'autre, et dont le produit est supérieur à k.

3.3.4 Résultats expérimentaux

Le principal intérêt des protocoles de type HARQ, est de réduire le nombre d'itérations nécessaires à la bonne réception d'un ensemble de paquets en comparaison avec des protocoles de type ARQ. Nous espérons donc que notre protocole va nous permettre de gagner un certain nombre d'itérations, en comparaison avec un protocole de type ARQ, n'utilisant donc pas de codes correcteurs en plus des acquittements et non-acquittement pour garantir l'acheminement de l'information.

Nous proposons ici de comparer notre protocole de transmission avec le protocole TCP sur deux critères principaux.

— Le premier est le nombre d'itérations nécessaires à la bonne transmission d'un ensemble de paquets.

— Le deuxième est le nombre total de paquets que l'émetteur va avoir à transmettre pour que la transmission puisse aboutir. Nous espérons que ce nombre pour notre méthode soit le plus proche possible de celui pour la solution sans code correcteur. En effet, supposons tout simplement que nous utilisions notre protocole sur une transmission ne subissant aucun effacement, on a alors envoyé trop de paquets par rapport à ce qui était nécessaire. Nous verrons que ces valeurs se rapprochent à mesure que le taux d'effacements augmente.

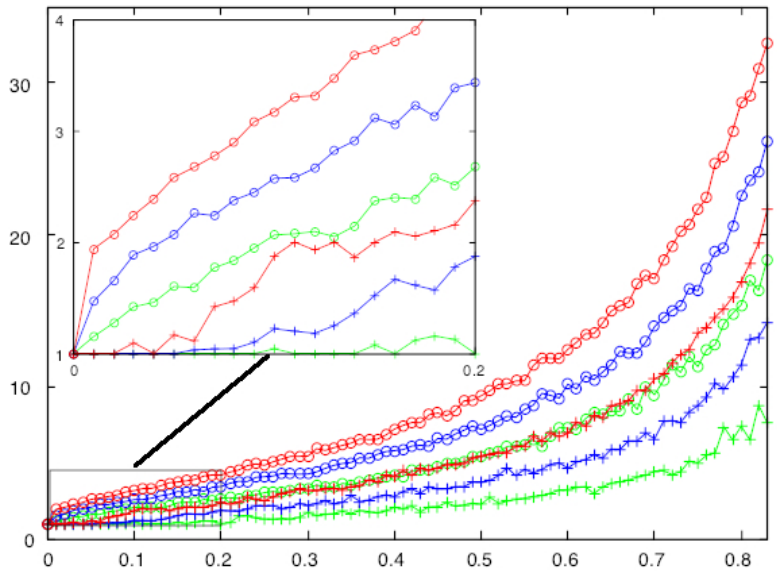


FIGURE 3.31 – Nombre moyen d'itérations nécessaires à la transmission de $K = 16$ paquets (rouge), $k = 64$ paquets (vert) et $k = 256$ paquets (bleu) en fonction du taux d'effacements. Les lignes avec des cercles représentent les itérations nécessaires à TCP, et les lignes avec des croix représentent celles pour notre solution.

La figure ci-dessus 3.31 montre qu'on gagne un certain nombre d'itérations en utilisant notre méthode. Pour un taux d'effacement même très faible, un protocole sans code correcteur nécessite très souvent une deuxième itération, alors que le notre permet de s'en passer. De plus, le gain en terme d'itérations persiste, et ce quel que soit le taux d'effacements.

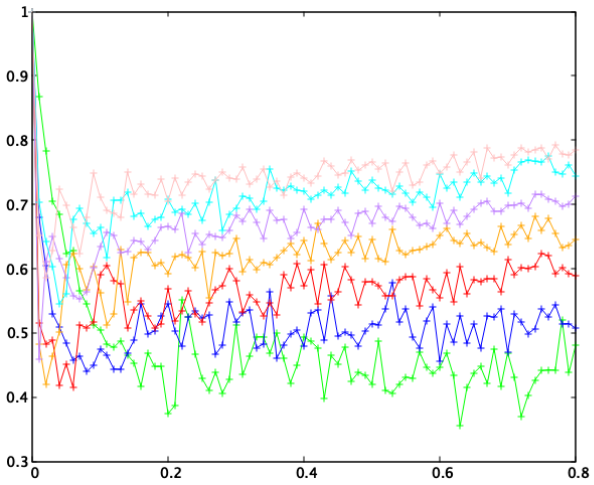


FIGURE 3.32 – Gain moyen en terme d'itérations entre le protocole TCP et notre solution en fonction du taux d'effacements P pour la transmission de $k = 16$ (vert), $k = 64$ (bleu), $k = 256$ (rouge), $k = 1024$ (orange), $k = 4096$ (violet), $k = 16384$ (cyan) et $k = 65536$ (roses) paquets.

La figure 3.32 représente les le ratio entre les courbes de la figure 3.31 (le nombre d'itérations nécessaires à notre méthode divisé par le nombre d'itérations nécessaires à TCP). On voit que le gain en terme d'itérations est substantiel, de l'ordre de 50%, et qu'il est encore meilleur à mesure qu'on a à transmettre beaucoup de paquets.

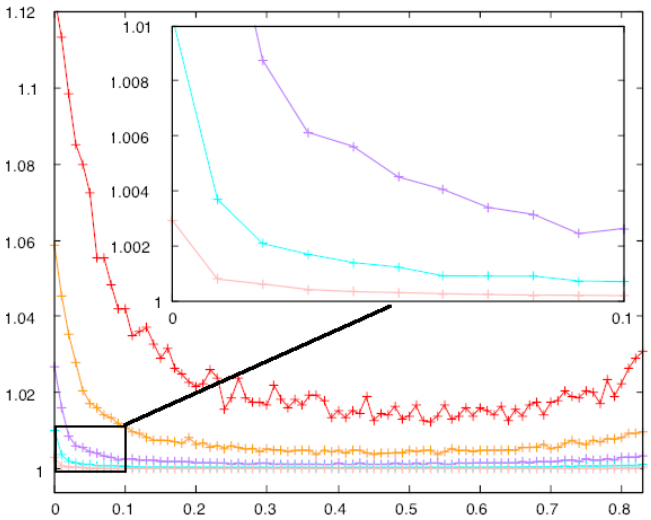


FIGURE 3.33 – Ratio moyen entre le nombre de paquets transmis avec notre méthode divisé par le nombre de paquets transmis avec le protocole TCP jusqu'à ce que la transmission termine, pour $k = 256$ (rouge), $k = 1024$ (orange), $K = 4096$ (violet), $k = 16384$ (cyan), $k = 65356$ (rose) en fonction de P

Enfin la figure 3.33 nous montre effectivement, quand il y a peu de peu d'effacements, et que le code n'est donc pas utile pour les corriger, on transmet avec notre méthode trop de paquets par rapport à ce qui serait nécessaire. Ceci est cependant moins le cas à mesure que le

taux d'effacements augmente, et que le code remplit alors son rôle. Une solution à ce problème pourrait être par exemple de ne pas transmettre les codes à la première itération, et ne les transmettre qu'à la deuxième. Néanmoins cette méthode nous ferait alors perdre l'avantage de notre méthode sur TCP, qui consiste entre autre à pouvoir se passer d'une deuxième itération dans le cas de très peu d'effacements.

Chapitre 4

Allocation de ressources temporelles et couplages

Le problème de couplage dans un graphe consiste à trouver un ensemble maximum d'arêtes deux-à-deux disjointes dans un graphe. Ce problème, très ancien, de la théorie des graphes peut être résolu en temps polynomial par le célèbre algorithme Edmonds [22]. Sur le plan théorique, le résultat d'Edmonds joue un rôle important dans l'optimisation combinatoire : il a marqué le début d'une longue liste de résultats algorithmiques résolvant ce problème par des calculs pratiques et rapides. Nous pouvons citer les travaux dans [15, 43] où la vision algébrique du problème est exploitée afin de proposer une résolution par multiplication de matrices. Dans Réf. [4], une variante de du problème où les accents sont mis sur l'équité est étudiée : une analyse a permis de résoudre en temps linéaire cette version équitable du problème de couplage. Même si ce résultat repose sur certaines hypothèses d'équité, le fait qu'un algorithme de temps linéaire existe est un très joli résultat algorithmique. De façon inattendue, malgré l'existence de plusieurs algorithmes le résolvant en temps polynomial, Couplage a su également attirer l'attention de la communauté de chercheurs en complexité paramétrée [23, 37]. Ici, l'effort de recherche a été mis dans le but de réduire le temps d'exécution de polynomial à un temps linéaire. Ceci est possible en transférant une partie de la complexité temporelle de manière à ce qu'elle s'exprime en fonction d'un autre paramètre de la donnée plutôt que la taille du problème.

D'un point de vue pratique, le problème du couplage permet de modéliser de façon simple les problèmes d'allocation de ressources. Par exemple, dans un graphe biparti où l'un des deux ensembles de sommets représente des tâches et l'autre ensemble de sommets représente des ouvriers ayant la capacité d'exécuter un sous-ensemble (potentiellement distinct) de ces tâches, la réponse au problème du couplage permet de maximiser le nombre de tâches pouvant être exécutées en parallèle par ces ouvriers. Ce problème a été étudié de manière intensive dans le cadre des données en *streaming*, où les arrivées imprévisibles des ouvriers doivent être affectées aux tâches en temps réel, cf. par exemple [41, 61]. Dans cette perspective, une solution randomisée très intelligente est décrite dans Réf. [21]. Plus généralement, dans un graphe quelconque représentant des compatibilités entre collaborateurs, le problème du couplage modélise le souci de maximiser la charge de travail globale lorsque le travail doit être effectué par des paires compatibles. Ce problème a été récemment étudié du point de vue des heuristiques [20]. En même temps, une comparaison quantitative de l'approche glouton à ce problème, sur des très larges données, est décrite dans Réf. [63].

A l'ère du tout numérique, la fouille de données issues des activités humaines montre qu'il est important de prendre en compte la dimension temporelle : les arêtes d'un graphe peuvent être datées par les instants où elles sont sauvegardées dans la base de données. Nous appelons ce type de données un flot de liens – *linkstream* – au sens de [29, 60]. L'illustration la plus naturelle de ces graphes temporels réside dans les *web logs*, où toute ligne de *log* comprend un champ sous la forme d'une date. Par exemple, considérons une application où les collaborateurs peuvent enregistrer à l'avance les intervalles de temps – discrétisés par jours, de 1 à 365 – où ils sont disponibles pour travailler en 2019. Considérons maintenant qu'un projet doit être exécuté par au moins deux collaborateurs sur un sprint d'au moins $\gamma = 14$ jours consécutifs afin que le projet soit terminé. Par limitation humaine, un collaborateur ne peut travailler que sur un nombre restreint de projets à la fois. Sous ces conditions, et en supposant qu'il y a un nombre infini de projets à faire, combien de projets peut-on terminer en 2019 ? Dans un article récent [8] nous nous posons cette question, que nous appelons un couplage temporel, avec paramètre γ. De façon surprenante, nous avons découvert que, contrairement à la version classique du problème de couplage, il est NP-difficile de répondre à ce problème. Cependant, nous avons réussi à proposer un précalcul polynomial satisfaisant dans le sens de la complexité paramétrée, en particulier sous le formalisme d'algorithme de *kernelization*[1].

Dans un premier temps, nous présentons les différents formalismes nécessaires à la compréhension du problème de couplage temporel dans un flot de lien. Puis nous présentons les résultats obtenus dans cet article, initialement publié en Anglais.

Link stream Un link stream (ou *flot de liens* en Français [60] [29]) L est une séquence de triplets de la forme (t, u, v), ou $t \in N$ représente un instant de temps, et $u \neq v$ représentent deux entités distinctes (sommets). Si t_{min} et t_{max} sont respectivement les plus petites et plus grandes valeurs de t du link stream L, on dira qu'il a une durée $t_{max} - t_{min} - 1$. Un triplet (t, u, v) dans un link stream peut par exemple signifier qu'à l'instant t, les entités u et v ont communiqué/eu une relation/ont travaillé ensemble, de la même manière que deux sommets d'un graphe sont reliés l'un à l'autre. On dira que le link stream L est de taille n quand il contient n triplets distincts de la forme (t, u, v).

Un link stream de la vie courante est par exemple la liste des logs des emails transmis au sein d'une entreprise : certains employés communiquent plus ou moins que d'autres, certains employés (du même service par exemple) communiquent souvent entre eux, d'autres jamais, et cette situation est amenée à évoluer dans le temps. Ainsi, un link stream peut être vu comme un graphe qui évolue avec le temps. Pour cet article nous utilisons notamment comme exemple de link stream la base de donnée de l'entreprise Enron [28], mise à disposition du public après des malversations financières. Nous utilisons aussi comme jeu de données l'expérience Rollernet [58] réalisée à Paris en 2006 : des personnes se déplacent en rollers dans les rue de Paris, et lorque deux personnes se retrouvent assez proches à un instant t, il existe une alors connexion entre ces deux personnes.

Pour définir un link stream, il faudra de plus définir l'unité de temps que nous considérons : au sein d'une entreprise, on choisira peut-être l'heure ou la journée comme unité de temps, mais on pourrait imaginer des cas d'utilisation différents pour lesquels on choisirait la seconde, le mois ou l'année. Nous noterons cette unité δ, et nous discutons des "bonnes" valeurs de ce

1. Nos études numériques et implémentations sont disponibles sur
`https://github.com/antoinedimitriroux/Temporal-Matching-in-Link-Streams`
`https://antoinedimitriroux.github.io`

δ dans cette publication. Ainsi, si deux employés ont échangé plusieurs emails dans la même unité de temps (ou au même instant), on dira qu'il n'y a qu'une seule arête entre eux deux à cet instant. Il y des valeurs de δ plus ou moins pertinentes suivant le link stream étudié. En effet, choisir comme unité de temps la milliseconde pour étudier les comportements humains au sein d'une entreprise n'est pas pertinent, et il faut compresser ou réunir les instants.

Maximum matching d'un graphe Dans le cas d'un graphe non-orienté G (et non d'un link stream), le problème du maximum matching (ou *couplage maximal* en Français) consiste à trouver le plus grand ensemble M de couples de sommets $M = [(v_a, v_b), (v_c, v_d), (v_e, v_f), \dots]$, tels qu'un sommet ne puisse appartenir qu'à un seul couple. Appliqué à la vie courante, ce problème revient à trouver dans une population le moyen de former le plus de couples possibles ([4]) : les sommets du graphe sont alors les personnes, et il existe une arête entre deux personnes si celles-ci sont d'accord pour former un couple. Il s'agit d'un problème simple à résoudre, faisant partie de la catégorie des problèmes P (pour lesquels il existe un algorithme de complexité polynomiale, en opposition aux problèmes difficiles, les problèmes NP). Ce problème peut notamment être rapporté à la multiplication de deux matrices [43] [15]. Dans le cas des graphes statiques, ce problème a été énormément étudié [20] [22], dans des contextes très divers, allant des algorithmes incrémentaux [21] aux algorithmes gloutons appliqués à de grandes quantités de données [63].

Temporal matching d'un link stream Le problème du temporal matching maximum d'un link stream est analogue au problème du maximum matching d'un graphe, auquel on ajouterait une dimension temporelle. Prenons un exemple concret : intéressons-nous au cas d'une entreprise ayant des employés présents ou absents de manière intermittente, et pouvant travailler parfois les uns avec les autres, et parfois non. Supposons maintenant que pour accomplir une tâche dans cette entreprise, deux employés doivent passer γ instants consécutifs ensemble, et que quand ils effectuent cette tâche, ils ne puissent pas en effectuer une autre. Le temporal matching maximum dans un link stream revient à se poser la question suivante : étant donnés ces employés, leur présence et leur possibilité de travailler à certains instants les uns avec les autres (interactions qui peuvent être représentées par un link stream), quel est le meilleur moyen pour leur faire accomplir le plus de tâches ? Cette question est une question difficile algorithmiquement : le problème est un problème NP.

Problème de décision On peut représenter le problème de temporal matching sous la forme d'un problème de décision. Un problème de décision est un problème auquel on doit répondre par "oui" ou "non". Ainsi, on peut se poser la question "dans un link stream donné, existe-t-il un couplage temporel de taille T ?" : répondre à cette question est difficile avec les ordinateurs dont nous disposons actuellement. Ci-dessous, nous présentons quelques exemples de problèmes de décision, appartenant à deux classes de complexité : les classes P et NP. Dans ces exemples, nous introduisons la notion de machine de Turing déterministe et non-déterministe, que nous précisons par la suite.

- Soit L une liste d'éléments : la liste L contient-elle l'élément X ? On peut répondre à cette question en temps polynomial sur une machine de Turing déterministe, en parcourant la liste une seule fois.
- La liste L est-elle une liste triée ? On peut répondre à cette question en temps polynomial sur une machine de Turing déterministe, en parcourant la liste une seule fois.

- L'enveloppe convexe d'un nuage de points est-elle de taille T ? On peut répondre à cette question en temps polynomial sur une machine de Turing déterministe, en calculant l'enveloppe convexe du nuage de points (ce qui peut se faire en temps polynomial), et de vérifier sa taille.
- Existe-t-il un chemin de longueur inférieure à L reliant deux sommets A et B dans un graphe G ? On peut répondre à cette question en temps polynomial sur une machine de Turing déterministe, en calculant le plus court chemin entre A et B dans le graphe G, et en comparant sa longueur à T.
- Le nombre entier X est-il un nombre premier ? Il n'existe pas à ce jour d'algorithme permettant de répondre à cette question en temps polynomial sur une machine de Turing déterministe. Cependant, on peut résoudre ce problème en temps polynomial sur une machine de Turing non-déterministe. De plus, si on nous propose un nombre entier D qui divise X, on peut vérifier en temps polynomial que c'est bien le cas.
- Existe-il un circuit hamiltonien de longueur inférieure à L dans un graphe G ? Il n'existe pas d'algorithme permettant de répondre à cette question en temps polynomial sur une machine de Turing déterministe. Cependant, on peut résoudre ce problème en temps polynomial sur une machine de Turing non-déterministe. Aussi, si on nous présente un tel circuit dans le graphe G, on peut vérifier qu'il est bien hamiltonien et de longueur inférieure à L, et répondre par "oui" à cette question en temps polynomial.

Les 4 premiers exemples appartiennent à la classe de complexité P : ils sont résolvables rapidement (en temps polynomial) sur une machine de Turing détermniste, alors que les 2 derniers exemples n'appartiennent pas à cette classe : ils appartiennent à la classe des problèmes NP, pour "No-deterministic Polynomial".

Problèmes P et NP Concrètement, les problèmes appartenant à la classe P sont des problèmes pour lesquels on connait un algorithme rapide (de complexité polynomiale) permettant de les résoudre sur un ordinateur conventionnel. Les problèmes de classe NP sont des problèmes pour lesquels on ne connaît pas encore d'algorithme rapide permettant de les résoudre, mais pour lesquels il existe un algorithme permettant de vérifier qu'une solution est valide. Si on disposait d'une infinité d'ordinateurs, ou d'un ordinateur pouvant effectuer une infinité de calculs en parallèle, on pourrait résoudre en temps polynomial les problèmes NP, en considérant toutes les solutions possibles, et en les vérifiant toutes (en temps polynomial). C'est pour cela qu'on dit que les problèmes NP sont résolubles en temps polynomial sur une machine de Turing non-déterministe : une telle machine peut s'apparenter à un ordinateur effectuant une quantité potentiellement infinie de calculs en parallèle.

Précisons que les problèmes appartenant à la classe de complexité P appartiennent aussi à la classe NP : la classe P est incluse dans la classe NP, $P \subseteq NP$. Les problèmes de la classe P sont eux-aussi résolvables en temps polynomial sur une machine de Turing non-déterministe, et appartiennent donc eux aussi à la classe NP.

Il existe d'autres classes de complexité, qui peuvent être rangées par ordre de difficulté :

$$L \subseteq NL \subseteq NC \subseteq P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME \subseteq \text{NEXPTIME} \subseteq EXPSPACE = NEXPSPACE$$

Rappelons qu'ici, ces classes sont incluses les unes dans les autres, et qu'on a bien $P \subseteq NP$.

Machine de Turing déterministe Une machine de Turing déterministe est un modèle abstrait permettant de représenter des appareils de calculs, tels que les ordinateurs actuels. Ce modèle a été décrit par Alan Turing en 1936 dans l'article [59]. Concrètement, une machine de Turing effectue successivement une série d'opérations afin d'effectuer un calcul. Pour ce faire, une telle machine va lire et écrire des informations sur un support en fonction de règles prédéfinies (représentées par un alphabet, un état initial, les états dans lesquels peut être la machine, les transitions entre ces états et les états terminaux). Les informations à écrire seront déterminées par l'état de la machine, qui va varier, et les symboles lus.

Elle dispose d'un ruban de longueur infinie, et d'une tête de lecture et écriture qui va lui permettre de modifier ce dernier en se déplaçant dessus. Les symboles lus et écrits sur le ruban appartiennent à un alphabet A fini (qui doit contenir le symbole "blanc").

Les ordinateurs actuels peuvent être vus comme des machines de Turing déterministes, disposant d'une quantité de mémoire limitée (une machine de Turing possédant une quantité de mémoire -son ruban- infinie).

Elle peut être dans différents états, choisis parmi un ensemble fini d'états E, et dispose d'un état initial E_1. Elle change d'état en fonction de son état actuel, et de la valeur du symbole lu sur le ruban se trouvant au niveau de sa tête de lecture. L'ensemble des transitions possibles d'un état à un autre en fonction d'un symbole est sa fonction de transition. A chaque état et en fonction du symbole lu sur le ruban, elle peut écrire sur le ruban et/ou se déplacer sur le ruban, puis changer d'état.

Enfin, elle dispose d'un ensemble d'états terminaux : la machine s'arrête lorsqu'elle rentre dans un de ces états. Le résultat du calcul est alors ce qui est écrit sur le ruban.

Supposons que l'on utilise l'alphabet $\{0, 1, .\}$ (. correspondant au symbole blanc), et que l'on souhaite construire une machine de Turing sur cet alphabet, qui prenne un ruban (sur lequel sont écrits des symboles de l'alphabet A), et nous retourne ce même ruban, mais pour lequel tout symbole 1 précédé d'un symbole 1 est remplacé par un symbole 0 (à la fin, pour toute séquence de plusieurs 1, on souhaite ne conserver que le premier). Concrètement, en donnant à cette machine le ruban .....00100111010011111011101..... on souhaite obtenir le ruban .....00100100010010000010001.....

La fonction de transition d'une telle machine pourra être représentée par la table de transitions suivante :

Etats	Symbole lu	Symbole écrit	Mouvement	Nouvel état
E_1	0	0	Droite	E_1
	1	1	Droite	E_2
	.	——Arrêt——	——Arrêt——	——Arrêt——
E_2	0	0	Droite	E_1
	1	0	Droite	E_2
	.	——Arrêt——	——Arrêt——	——Arrêt——

Table 4.1 – Table de transition d'états d'une machine de Turing

Supposons maintenant que l'on souhaite construire une machine de Turing sur ce même alphabet A, qui remplace chaque 0 par un 1 ou chaque 1 par un 0. La fonction de transition d'une telle machine pourra être représentée par la table de transitions suivante :

Etats	Symbole lu	Symbole écrit	Mouvement	Nouvel état
E_1	0	1	Droite	E_1
	1	0	Droite	E_1
	.	—Arrêt—	—Arrêt—	—Arrêt—-

Table 4.2 – Table de transition d'états d'une machine de Turing

Enfin, supposons que l'on souhaite construire une machine de Turing sur ce même alphabet A, qui dédouble chaque symbole 1. En donnant à cette machine le ruban .....000010110001..... on souhaite obtenir le ruban .....0000110111100011.....

La fonction de transition d'une telle machine pourra être représentée par la table de transitions suivante :

Etats	Symbole lu	Symbole écrit	Mouvement	Nouvel état
E_1	0	0	Droite	E_1
	1	.	Droite	E_2
	.	—Arrêt—	—Arrêt—	—Arrêt—-
E_2	0	1	Droite	E_3
	1	1	Droite	E_2
	.	1	Gauche	E_4
E_3	0	0	Droite	E_3
	1	0	Droite	E_2
	.	0	Gauche	E_4
E_4	0	0	Gauche	E_4
	1	1	Gauche	E_4
	.	1	Droite	E_5
E_5	1	1	Droite	E_1

Table 4.3 – Table de transition d'états d'une machine de Turing

Pour bien se figurer le fonctionnement de cette machine, nous présentons les différentes étapes des calculs qu'elle effectue sur un exemple concret dans le tableau 4.4 ci-dessous. En prenant comme ruban .0010110.... et en plaçant la tête de lecture sur le symbole le plus à gauche qui ne soit pas le symbole blanc, la machine va effectuer les opérations suivantes pour obtenir le ruban .0011011110....

On peut lire le tableau 4.4 de la manière suivante : à la 4eme itération, la machine est dans l'état E_3, et a sa tête de lecture sur le cinquième symbole qui vaut 1. Elle lit donc le symbole 1 : son état E_3 et la valeur du symbole lu 1 lui indique qu'il faut écrire à cet endroit du ruban le symbole 0 puis de déplacer sa tête de lecture vers la droite, avant de passer à l'état E_2. Le symbole sur lequel se situe la tête de lecture de la machine est souligné.

	Ruban	Etat+Lu	Ecrit	Déplacement	Nouvel etat
0	.$\underline{0}$10110.....	E_1+0	0	» Droite	E_1
1	.0$\underline{0}$10110.....	E_1+0	0	» Droite	E_1
2	.00$\underline{1}$0110.....	E_1+1	.	» Droite	E_2
3	.00.$\underline{0}$110.....	E_2+0	1	» Droite	E_3
4	.00.1$\underline{1}$10.....	E_3+1	0	» Droite	E_2
5	.00.10$\underline{1}$0.....	E_2+1	1	» Droite	E_2
6	.00.101$\underline{0}$.....	E_2+0	1	» Droite	E_3
7	.00.1011$\underline{.}$.....	E_3+.	0	» Droite	E_4
8	.00.101$\underline{1}$0.....	E_4+1	1	« Gauche	E_4
9	.00.10$\underline{1}$10.....	E_4+1	1	« Gauche	E_4
10	.00.1$\underline{0}$110.....	E_4+0	0	« Gauche	E_4
11	.00.$\underline{1}$0110.....	E_4+1	1	« Gauche	E_4
12	.00$\underline{.}$10110.....	E_4+.	1	» Droite	E_5
13	.001$\underline{1}$0110.....	E_5+1	1	» Droite	E_1
14	.0011$\underline{0}$110.....	E_1+0	0	» Droite	E_1
15	.00110$\underline{1}$10.....	E_1+1	.	» Droite	E_2
16	.00110.$\underline{1}$0.....	E_2+1	1	» Droite	E_2
17	.00110.1$\underline{0}$.....	E_2+0	1	» Droite	E_3
18	.00110.11$\underline{.}$....	E_3+0	0	» Droite	E_3
19	.00110.1$\underline{1}$0....	E_4+1	1	« Gauche	E_4
20	.00110.$\underline{1}$10....	E_4+1	1	« Gauche	E_4
21	.00110$\underline{.}$110....	E_4+.	1	» Droite	E_5
22	.001101$\underline{1}$10....	E_5+1	1	» Droite	E_1
23	.001101$\underline{1}$0....	E_1+1	.	» Droite	E_1
24	.0011011.$\underline{0}$....	E_1+0	1	» Droite	E_3
25	.0011011.1$\underline{.}$...	E_3+.	0	« Gauche	E_4
26	.0011011.$\underline{1}$0...	E_4+1	1	« Gauche	E_4
27	.0011011$\underline{.}$10...	E_4+.	1	» Droite	E_5
28	.0011011$\underline{1}$10...	E_5+.	1	» Droite	E6
29	.0011011110...	E_1+0	0	» Droite	E_1
30	.0011011110$\underline{.}$..	E_1+.	.		(Arrêt)

Table 4.4 – Déroulement des états d'une machine de Turing

On peut voir ces différentes états de la manière suivante :

— L'état E_1 consiste à trouver le premier symbole 1 que l'on va devoir dupliquer. Une fois celui-ci trouvé, on le transforme en un symbole blanc, ce qui nous permettra de retrouver sa position. On passe ensuite aux états E_2 et E_3.

— Les états E_2 et E_3 servent à "décaller" tous les symboles se situant après ce 1, d'une case vers la droite sur le ruban, jusqu'à trouver un symbole blanc (on a alors atteint la fin de notre séquence de symboles, et on peut passer à l'étape E_4).

— L'état E_4 consiste à revenir à la position du 1 qu'on a dupliqué (qui vaut maintenant le symbole blanc), et de remettre cette valeur à 1. On passe ensuite à l'étape E_5, qui va nous permettre de "sauter" le prochain symbole (on sait que ce symbole vaut nécessairement 1). On a dupliqué un symbole 1, et on se retrouve maintenant sur le symbole qui le succédait avant duplication.

— A la fin de cette succession d'états, on a dupliqué un 1 (E_1), convenablement recopié les symboles qui lui succèdent (E_2 et E_3), et donc augmenté de 1 le nombre de symboles de notre séquence. La tête de lecture se retrouve sur le symbole qui suivait le 1 dupliqué (E_4 et E_5), qui peut valoir 0, 1 ou le symbole blanc. On peut maintenant revenir à l'étape E_1, et relancer cette succession d'états.
— La machine s'arrêtera uniquement lorsqu'elle sera dans l'état E_1 et en lisant le symbole blanc.

Bien que de telles machines puissent paraître abstraites, elles peuvent réaliser des calculs utiles et complexes. Les ordinateurs actuels peuvent être vus comme la concrétisation d'une machine de Turing déterministe, à la différence que ceux-ci disposent d'une quantité finie de mémoire (alors qu'une machine de Turing dispose d'un ruban de longueur infinie). La quantité de mémoire disponible sur les ordinateurs actuels pouvant être très importante, on peut les considérer comme des machines de Turing, lorsque la quantité d'information du problème à traiter est suffisament faible.

Machine de Turing non-déterministe On a vu qu'une machine de Turing déterministe passait d'un état à l'autre en fonction de son état actuel et du symbole lu par sa tête de lecture. Cette transition d'un état à l'autre se fait de manière déterministe : la combinaison d'un état et d'un symbole lu amène à un seul et unique nouvel état.

Une machine de Turing non-déterministe effectue les mêmes types de calcul qu'une machine déterministe, mais se comporte différemment au niveau de la transition de ses états : en fonction de son état actuel et du symbole lu, elle peut entrer dans différents nouveaux états en se dupliquant. Ainsi, alors que les calculs effectués sur une machine déterministe forment une suite, ceux effectués par une machine non-déterministe forment un arbre, ou chaque chemin correspond à un calcul possible sur cette machine.

Précisons qu'une machine non-déterministe peut tout à fait se comporter comme une machine déterministe, si chaque transition mène à un seul état (les machines déterministes sont un cas particulier des machines non-déterministes) : ainsi, une machine non-déterministe est capable de résoudre les mêmes problèmes qu'une machine déterministe.

On peut voir une telle machine de la manière suivante : lorsque la machine se trouve dans un état pour lequel il existe plusieurs transitions, elle se dédouble (elle recopie son ruban, sa tête de lecture, son état, etc...), et chacun de ses doubles entre dans l'état adéquat. En se dédoublant, la machine va parcourir toute l'arborescence des calculs qu'elle a à effectuer, jusqu'à entrer dans un état terminal et s'arrêter.

La classe de complexité NP est donc constituée de tous les problèmes résolvables en temps polynomial sur ce type de machine. Les problèmes P sont les problèmes résolvables en temps polynomial sur des machines déterministes. Puisqu'ils le sont aussi sur une machine déterministe, on dit alors que les problèmes la classe des problèmes P est incluse dans NP (d'ou $P \subseteq NP$).

On sait donc que les problèmes $P \subseteq NP$, mais on ne sait actuellement pas si $NP \subseteq P$, et donc si $P = NP$. Cette question fondamentale est une des plus importantes de l'informatique. Supposons que $P = NP$: cela signifie que pour les problèmes pour lequels il est facile de vérifier une solution, il existe forcément un algorithme (qu'on ne connaît pas encore), résolvant ces problèmes en temps polynomial sur une machine déterministe. A contrario, si $P \neq NP$, il existe des problèmes qui, quand bien même il existe un algorithme permettant de vérifier une solution rapidement, ça n'est pas la peine de chercher un algorithme rapide les résolvant, puisqu'il ne peut pas exister. La seule méthode de résolution de ces problèmes est alors de

parcourir toutes les solutions possibles, et de les vérifier unes par unes.

De nombreux algorithmes importants reposent sur l'hypothèse que $P \neq NP$, comme l'algorithme de cryptage à clé publique RSA (qui repose sur le fait que factoriser un grand nombre quasi-premier est un problème NP, alors qu'on sait trouver facilement de grands nombres premiers -de manière probabiliste- et les multiplier entre eux). Si on démontrait que $P = NP$, ces algorithmes de cryptage pourraient rapidement s'avérer caduques. Si on démontrait que $P \neq NP$, cela signifierait qu'il existe des problèmes intrinsèquement difficiles, et qu'on peut continuer à utiliser cette difficulté dans de nombreux domaines.

Réduction d'un problème à un autre Soit deux problèmes A et B d'une même classe de complexité C. On dit que A peut être réduit à B si il existe une méthode (ou un algorithme) permettant de transformer le problème A en le problème B. Ainsi, si l'algorithme de réduction a une complexité inférieure au problème A, et qu'on dispose d'un moyen permettant de résoudre le problème B, il nous suffit de transformer le problème A en ce dernier, de le résoudre, puis d'utiliser la solution de B pour déterminer la solution de A.

Prenons un exemple concret : soit A le problème de calcul de l'enveloppe convexe d'un nuage de points, et B le problème du tri d'une liste. Ces deux problèmes appartiennent à la classe de complexité P. L'algorithme proposé par Graham permettant de calculer l'enveloppe convexe d'un nuage de points fonctionne de la manière suivante :

- Trouver le point X ayant la plus petite abscisse dans le nuage de points.
- Trier tous les points du nuage de points en fonction de l'angle qu'ils forment avec la droite parallèle à la droite des ordonnées passant par X, pour obtenir la liste L.
- En parcourant la liste L, et en prenant les points 3 par 3, on élimine tous les points qui ne forment pas un "tour droit".
- La liste L correspond alors à l'enveloppe convexe du nuage de points, triée dans le bon ordre.

La figure 4.1 ci-dessous représente l'algorithme de Graham calculant l'enveloppe convexe d'un nuage de 7 points.

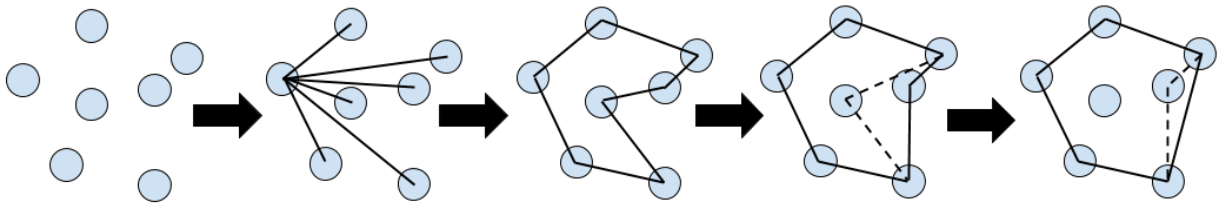


Figure 4.1 – Exemple du déroulement de l'algorithme de Graham pour calculer l'enveloppe convexe d'un nuage de points

Trouver le point de plus petite abscisse se fait rapidement (on ne parcourt qu'une seule fois le nuage de points), et une fois le tri effectué, l'élimination des "tours gauches" se fait elle aussi rapidement. On voit que dans cet algorithme, le calcul de l'enveloppe convexe consiste principalement à trier les points selon l'axe qu'ils forment avec la droite parallèle à celle des ordonnées et passant par X. On dit alors que cet algorithme est dominé par l'algorithme de tri, et sa complexité est alors en $O(n \log n)$, la complexité de l'algorithme de tri. Avec son algorithme, Graham est donc parvenu à réduire le problème de l'enveloppe convexe à un problème de tri.

On peut aussi citer comme exemple le problème de la recherche d'un élément dans une liste triée : en effectuant une recherche naïve (en parcourant la liste jusqu'à trouver l'élément recherché), on effectue un algorithme de complexité linéaire. Cependant, si on effectue une recherche dichotomique dans cette liste, on peut trouver l'élément (ou affirmer que la liste ne le contient pas) en complexité logarithmique. On a alors réduit ce problème au parcours d'un arbre binaire. La figure 4.2 représente un exemple de cet algorithme.

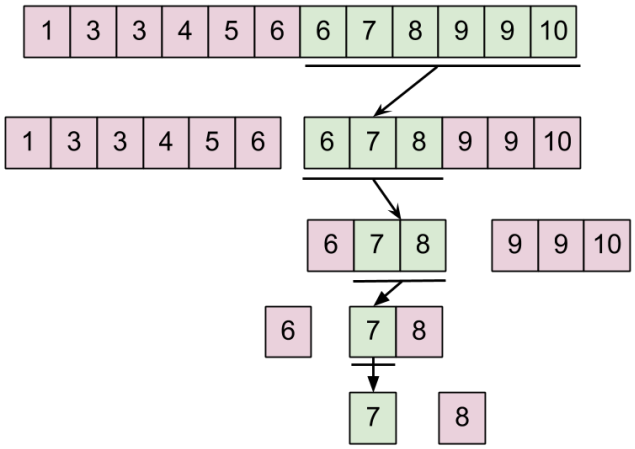


Figure 4.2 – Recherche dichotomique du nombre 7 dans une liste de 12 éléments.

De nombreux problèmes algorithmiques de différentes classes de complexité peuvent être réduits de cette manière à d'autres problèmes. Ceci est intéressant car, si on parvient à réduire tous les problèmes d'une classe de complexité C à un problème A, alors trouver une méthode résolvant rapidement A revient à trouver une méthode résolvant rapidement tous les problèmes de cette classe.

Problèmes C-difficiles et C-complets On a vu qu'il était parfois possible de réduire un problème A à un autre problème B.

Soit C une classe de complexité, et soit A un problème, pas nécessairement de la classe C. Si tous les problèmes de la classe C peuvent être réduits au problème A en temps polynomial, on dit alors que A est C-difficile. Autrement dit, si une méthode résolvant le problème A permet de résoudre tous les problèmes de classe C, alors A est C-difficile.

Lorsque le problème A appartient à la classe C (et qu'il est C-difficile), on dit alors que ce problème est C-complet.

Cette notion est importante, car comme on l'a dit précédemment, trouver une méthode résolvant rapidement un problème C-difficile revient à trouver une méthode permettant de résoudre rapidement tous les problèmes de classe C.

Problèmes NP-difficiles et NP-complets On vient de définir ce qu'était un problème C-difficile et C-complet, pour une classe de complexité C quelconque. On a aussi vu que certains problèmes algorithmiques appartenaient à la classe NP, résolvables en temps polynomial sur une machine de Turing non-déterministe, mais pas sur une machine déterministe.

Soit A un problème : si on parvient à démontrer que tout problème NP peut être réduit au problème A, on dit alors que ce problème est NP-difficile. De plus, si le problème A est lui-même un problème NP, on dit alors que c'est un problème NP-complet.

Précisons que pour démontrer qu'un problème A est NP-difficile, il n'est pas nécessaire de démontrer que tous les problèmes NP peuvent être réduits à ce problème : on peut en effet prendre un problème B dont on sait qu'il est NP-difficile (comme le problème $SAT3$), et démontrer que le problème B peut être réduit au problème A. Ainsi, tous les problèmes NP pouvant être réduits au problème B, il peuvent être réduits au problème A. De nombreux problèmes ont été démontrés NP-complets en réduisant le problème $SAT3$ (un problème NP-complet) à ces problèmes. C'est notamment de cette manière que nous démontrons la NP-complétude du couplage temporel dans cet article.

La figure ci-dessous 4.3 représente graphiquement la notion de NP-complétude : les problèmes NP-complets sont à l'intersection des problèmes NP-difficiles et des problèmes NP.

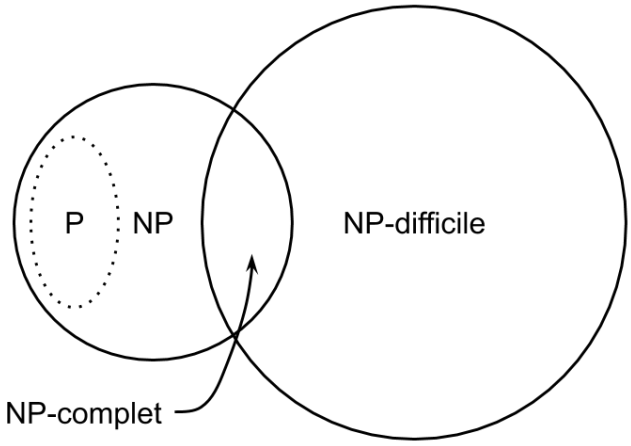


Figure 4.3 – Un problème à la fois NP-difficile et NP est un problème NP-complet.

Heuristique Trouver un couplage maximal dans un link stream est donc un problème difficile appartenant à la catégorie des problèmes NP-difficiles et NP-complets. Pour espérer obtenir une bonne solution en temps raisonnable à ces problèmes, on peut avoir recours à une heuristique [20] : il s'agit d'une règle qu'on applique lors de l'exécution d'un algorithme, et qui a tendance à nous approcher rapidement d'une solution optimale. Un exemple d'heuristique est de prendre le chemin le plus en ligne droite lorsqu'on cherche le plus court chemin entre deux sommets d'un graphe géométrique : c'est souvent la bonne manière de faire, bien que parfois, il faille s'éloigner de l'objectif (au sens de la distance Euclidienne) pour y aboutir au plus vite.

Dans cet article, nous proposons deux algorithmes permettant de nous approcher de la solution optimale pour le problème du couplage temporel : le premier est une 2-approximation, le deuxième, utilisant les résultats du premier, est le calcul d'un Kernel ("noyau" en Français) quadratique en la taille de la 2-approximation (et donc de la solution optimale). Ces deux algorithmes ont tous deux une complexité polynomiale.

2-approximation Supposons que la meilleure solution au problème du temporal matching de taille maximum pour un link stream L soit k (mais qu'il soit bien évidemment impossible de

le savoir, étant donné la complexité du problème). Une 2-approximation est une solution sous-optimale au problème, dont la taille k_{approx} est telle que $k/2 \leq k_{approx}$. Une 2-approximation peut donc être vue comme une réponse "au moins à moitié bonne" à la question posée.

Dans cet article, nous proposons un algorithme glouton ("greedy algorithm" en Anglais) calculant une 2-approximation du problème de temporal matching dont la complexité est quasi-linéaire.

Il consiste à prendre la première γ-arête disponible dans la liste des γ-arêtes du link stream L, puis de retirer toutes les γ-arêtes chevauchant cette dernière, et de recommencer jusqu'à ce que la liste soit vide. Pour que l'algorithme retourne une 2-approximation, il faut néanmoins choisir les arêtes dans l'ordre chronologique (ou dans l'ordre inverse). Si on ne tient pas compte de l'ordre chronologique (et que par exemple on peut choisir comme première arête une arête se situant au "milieu" du link-stream) l'algorithme retourne une 4-approximation.

On peut aussi ajouter que pour un link stream donné, on peut trouver des 2-approximations de différentes tailles, suivant l'ordre dans lequel on choisit les γ-arêtes. Ainsi, si on trouve deux 2-approximations, dont l'une contient n γ-arêtes et l'autre $2n$, on sait que la deuxième est un couplage maximal, car la première ne serait pas une 2-approximation s'il existait un couplage maximal de taille supérieure ou égale à $2n + 1$.

La figure 4.4 ci-dessous représente des 2-approximations et une 4-approximation pour deux link-streams. Dans notre article, nous ne parlons pas de la 4-approximation, la 2-approximation étant bien meilleure.

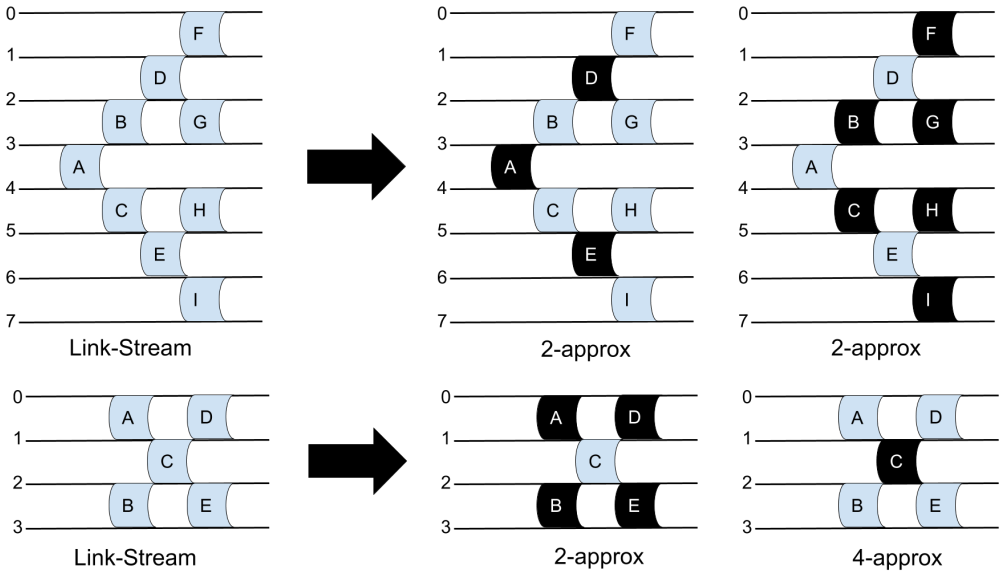


Figure 4.4 – Pour le link-stream du haut, on a trouvé une 2-approximation de taille 3 (γ-arêtes A, D et E trouvables dans le sens chronologique) et une de taille 6 (γ-arêtes B, C, F, G, H et I trouvables dans le sens inverse) : cela signifie que le couplage maximal est nécessairement de taille 6, puisque le γ-matching de taille 3 est une 2-approximation.
Pour le link-stream du bas, on a trouvé une 2-approximation de taille 4 (γ-arêtes A, B, D et E trouvables dans les deux sens), et une 4-approximation de taille 1 (on a pris l'arête C, qui nous interdit les 4 autres arêtes). On sait que le couplage maximal est de taille au plus 4, et on a une solution de taille 4.

On notera que l'ordre dans lequel se trouvent les γ-arêtes peut influer sur la taille de la 2-approximation. On pourrait chercher à itérer un certain nombre de fois notre algorithme, en triant la liste aléatoirement, et en cherchant la meilleure 2-approximation.

La figure ci-dessous 4.5 représente un exemple de ce cas de figure : même en respectant l'ordre chronologique, on peut obtenir de plus ou moins bonnes 2-approximations.

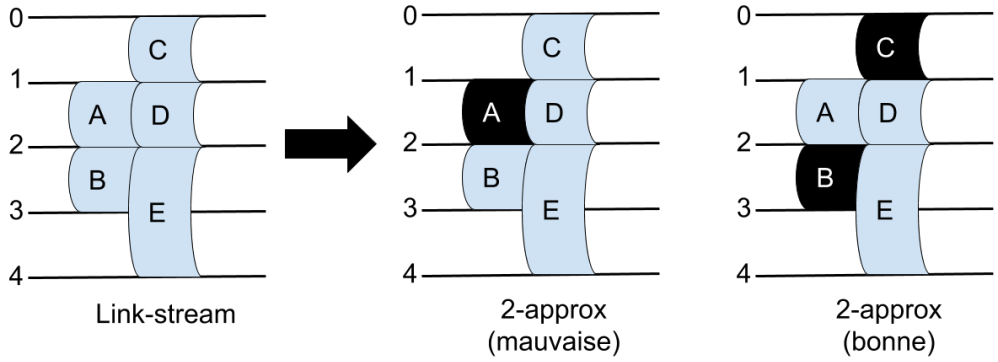


Figure 4.5 – Suivant la première γ-arête qu'on a choisie, on a trouvé une 2-approximation de taille 1 et une de taille 2. On peut donc trouver de plus ou moins bonnes 2-approximations suivant l'ordre dans lequel sont rangées les γ-arêtes. Là-encore, on sait que le couplage maximal est de taille 2, car on a trouvé deux 2-approximations dont l'une est deux fois plus petite que l'autre.

Ajoutons aussi qu'il est possible d'adapter cet algorithme à un flux : on parle alors d'algorithme "online" (ou algorithme incrémental en Français [41] [61]), qui permet de calculer une 2-approximation en recevant les arêtes de notre link-stream au fur et à mesure, et non en disposant de celles-ci de manière statique. Précisons néanmoins que pour que la solution retournée soit une 2-approximation, il faut que les arêtes soient reçues dans le "bon ordre" : si on a reçu une arête pour l'instant t, on ne recevra plus jamais d'arête pour un instant ultérieur à t.

Kernel Soit un γ-link stream L de taille t. Nous savons qu'il existe un couplage temporel maximal, dont nous ne connaissons pas la taille. Un kernel de ce problème revient à trouver un "sous" link stream L_{Kernel}, inclus dans L ($L_{Kernel} \subseteq L$), et donc de taille $t_{Kernel} \leq t$, qui contient la solution optimale. Ainsi, résoudre le problème dans L_{Kernel} est plus rapide que de le résoudre dans L, mais nous permet d'obtenir un résultat néanmoins optimal. L'algorithme de kernelisation que nous proposons est de complexité quasi-linéaire. Cet algorithme est aussi un algorithme paramétrique [23] [37] : il prend en entrée le link stream dont on cherche le temporal matching maximal, ainsi qu'une valeur k correspondant au nombre d'arêtes retournées par l'algorithme de 2-approximation.

Cet article présente donc dans un premier temps un algorithme permettant de calculer une (ou plusieurs) 2-approximations en temps quasi-linéaire : il s'agit d'un algorithme glouton ("greedy algorithm"), et démonstration est faite que la solution de cet algorithme est bien une 2-approximation de la solution optimale.

Supposons que l'algorithme de 2-approximation nous retourne une taille de couplage k_{approx} pour un link stream L. Le deuxième algorithme présenté dans cet article prend en entrée le

link stream L et le résultat du premier algorithme k_{approx}, et nous retourne un sous-link stream L_{Kernel} (qui peut être égal à L) de taille au maximum égale à $2(k_{approx}-1)*(2k_{approx}-1)*\gamma^2$, donc proportionnelle au carré de la 2-approximation. Notons d'ailleurs que chercher une petite 2-approximation peut donc aussi être intéressant pour trouver des petits kernels.

Nous proposons une implémentation de ces algorithmes, ainsi que des simulations sur différents jeux de tests, tirés de la vie réelle, ou bien que nous avons générés nous-mêmes. Ces simulations montrent qu'il est possible dans certains cas de réduire considérablement la taille du problème (donc du link stream) par l'algorithme de Kernelisation.

Cas d'utilisation Dans la figure 4.6, nous présentons un exemple de link-stream, et un cas d'application du problème de couplage maximal. Attention cependant, cet exemple n'est pas tout à fait exact mais assez parlant : deux épées (sommets dans le link-stream) peuvent être capables de viser deux cibles différentes au même instant, ce qui peut conduire à avoir plusieurs γ-arêtes entre deux sommets au même instant. Pour rendre cet exemple exact, nous pourrions par exemple ajouter que si deux épées peuvent viser plusieurs vaisseaux, elles ne peuvent en réalité viser que le vaisseau le plus proche. Un autre exemple présenté plus bas, déjà évoqué précédemment, est un exemple exact et correspond parfaitement à un link-stream et au problème du couplage temporel.

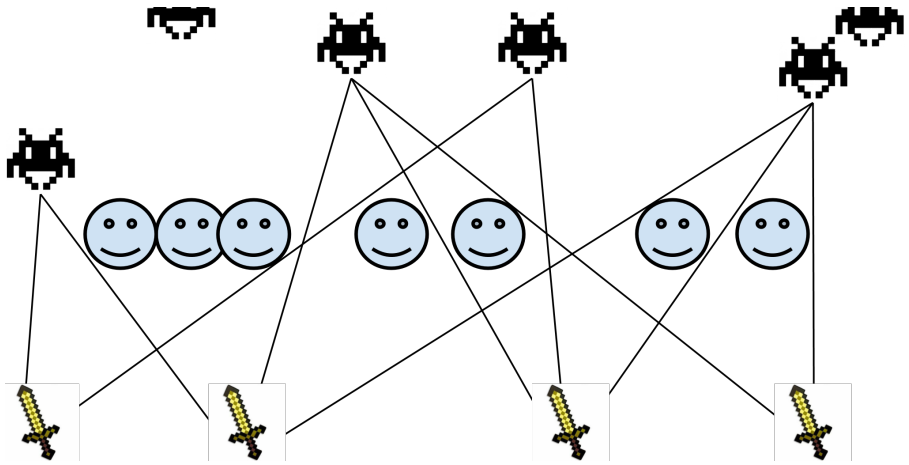


Figure 4.6 – Un cas d'utilisation du couplage maximal dans les link-streams (pas tout à fait exact mais parlant) : les épées en bas représentent des soldats défendant leur base (ceux sont les sommets du link-stream), les smileys représentent des alliés à éviter, et les vaisseaux en haut des cibles à abattre. Les smileys et les cibles se déplacent, les cibles sont donc parfois atteignables et parfois non. Pour abattre une cible, deux épées doivent la viser avec un laser pendant γ secondes consécutives, elles ne peuvent se concentrer que sur une cible à la fois, et il ne sert à rien de viser une cible avec plus de deux lasers. La question est alors : combien peut-on abattre de cibles au maximum ?

Un exemple exact concerne l'exécution de tâches en binôme au sein d'une entreprise. Deux employés peuvent effectuer une tâche si ils s'y concentrent tous les deux pendant γ jours consécutifs, et ne peuvent travailler que sur une seule tâche à la fois. La figure 4.7 représente ce cas d'utilisation pour une entreprise de 6 employés. Etant présents au sein de l'entreprise par

intermittence, il faut trouver le meilleur couplage pour leur faire effectuer le plus de tâches possibles.

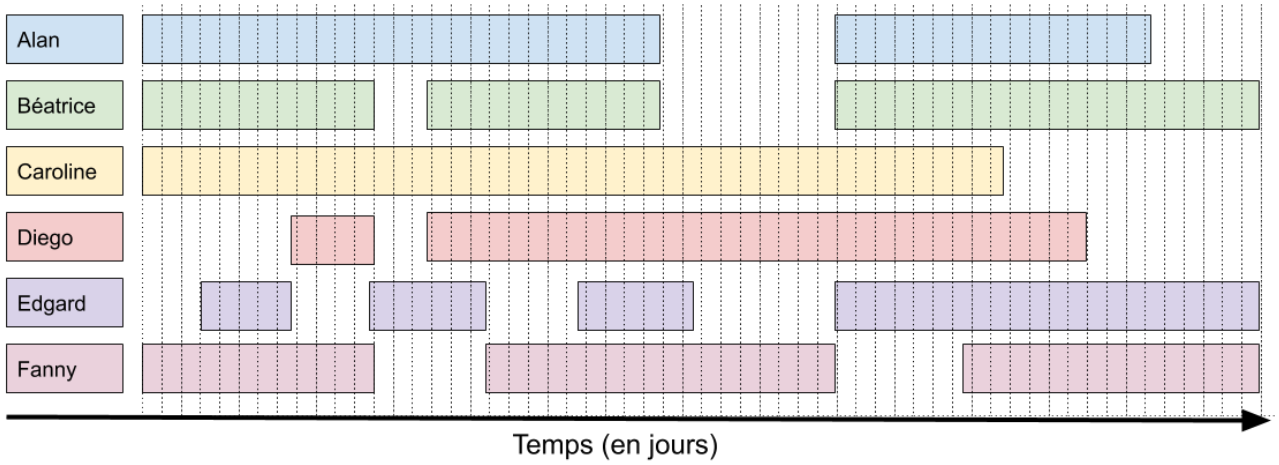


FIGURE 4.7 – Cas d'utilisation du couplage maximal dans les link-stream (exact) : Comment faire effectuer le plus de tâches possibles à Alan, Béatrice, Caroline, Diego, Edgard et Fanny (les sommets de notre link-stream), sachant qu'il leur faut γ jours consécutifs pour mener une tâche à bien, et qu'ils ne peuvent effectuer qu'une tâche à la fois ?

4.1 Couplage temporel

Sauf indication contraire, les graphes dans ce chapitre sont simples, non orientés et sans boucle. On note $\mathbb{N}$ l'ensemble des entiers positifs. Etant donné deux entiers p et q, on note p, q l'ensemble $\{r \in \mathbb{N} \mid p \leq r \leq q\}$. Un *flot de liens* L est un triple (T, V, E) tel que $T \subseteq \mathbb{N}$ est un intervalle, V est un ensemble, et $E \subseteq T \times \binom{V}{2}$. Les flots de liens peuvent être vus comme une extension des graphes. En effet, un graphe est un flot de liens où $|T| = 1$. Les éléments de V sont appelés *sommets* et les éléments de E sont appelés *arêtes (temporisés)*. Un *sommet temporel* de L est une paire (t, u) telle que $t \in T$ et $u \in V$.

Soit un entier γ, une γ-arête entre deux sommets u et v à l'instant t, notée $\Gamma_\gamma(t, u, v)$, est l'ensemble $\{(t', \{u, v\}) \mid t' \in t, t + \gamma - 1\}$. Nous disons qu'une γ-arête Γ contient un sommet temporel (t, u) s'il existe un sommet $v \in V$ tel que $(t, \{u, v\}) \in \Gamma$. Nous disons que deux γ-arêtes sont indépendantes si elles ne contiennent aucun sommet en commun. Un γ-couplage $\mathcal{M}$ d'un flot de liens L est un ensemble de γ-arêtes deux à deux indépendantes. Nous disons qu'une γ-arête Γ est adjacente à un sommet $u \in V$ s'il existe un sommet $v \in V$ et un entier $t \in T$ tel que $\Gamma = \Gamma_\gamma(t, u, v)$. Nous disons qu'une arête $e \in E$ appartient à un γ-couplage $\mathcal{M}$ s'il existe $\Gamma \in \mathcal{M}$ tel que $e \in \Gamma$.

Nous nous concentrons sur le problème suivant.

γ-COUPLAGE Un flot de liens L et un entier k. Un γ-couplage de L de taille k ou une réponse correcte qu'un tel ensemble n'existe pas.

Theorem 3. γ-COUPLAGE *est NP-dur pour* $\gamma > 1$.

Démonstration. Nous prouvons la NP-complétude de la version de décision de γ-COUPLAGE avec une réduction à partir de 3-SAT. Soit φ une formule avec n variables $x_1, \ldots, x_n$ et m

clauses $C_0, \ldots, C_{m-1}$ telles que chaque clause a une taille d'au plus 3. Sans perte de généralité, nous supposons qu'une clause ne contient pas deux fois la même variable. Nous appelons X l'ensemble contenant les n variables et $\mathcal{C}$ l'ensemble contenant les m clauses.

Nous définissons le flot de liens $L = (T, V, E)$ de la manière suivante :

— $T = 0, (m+1)\gamma - 1$.

— $V = \{x^-, x^=, x^+ \mid x \in X\} \cup \{x_t^{++}, x_t^{--} \mid x \in X, t \in 0, m-1\} \cup \{c\}$

— $E = E_{var} \cup E_{cla}$ où :

$$\begin{aligned}
E_{var} &= \{(t, \{x^=, x^+\}), (t, \{x^=, x^-\}) \mid t \in 0, (m+1)\gamma - 1, x \in X\} \\
&\cup \{(t, \{x^+, x_i^{++}\}) \mid t \in i\gamma + 1, (i+1)\gamma, i \in 0, m-1, x \in X\} \\
&\cup \{(t, \{x^-, x_i^{--}\}) \mid t \in i\gamma + 1, (i+1)\gamma, i \in 0, m-1, x \in X\} \\
E_{cla} &= \{(t, \{c, x_i^{++}\}) \mid t \in i\gamma + 1, (i+1)\gamma, i \in 0, m-1, x \in X, \\
&\qquad x \text{ est une variable positive dans } C_i\} \\
&\cup \{(t, \{c, x_i^{--}\}) \mid t \in i\gamma + 1, (i+1)\gamma, i \in 0, m-1, x \in X, \\
&\qquad x \text{ est une variable positive dans } C_i\}.
\end{aligned}$$

Nous décrivons dans la Figure 4.8 le flot de liens créé pour $\gamma = 3$ et $\varphi = (w \vee \overline{x} \vee y) \wedge (w \vee x \vee \overline{z})$.

Nous montrons qu'il existe une affectation des variables qui satisfait φ si et seulement si L contient un γ-couplage de taille $(2m+1)n + m$.

De manière intuitive, l'arête entre $(0, \{x^=, x^+\})$ et $(0, \{x^=, x^-\})$ qui appartient au γ-couplage retourné en réponse au problème γ-COUPLAGE détermine si la variable x aura pour valeur vrai ou faux. De plus, la taille demandée pour ce γ-couplage garantit le fait suivant. Si l'arête $(0, \{x^=, x^+\})$ (resp. $(0, \{x^=, x^-\})$) appartient au γ-couplage, alors toute arête parmi $(t, \{x^=, x^+\})$ (resp. $(t, \{x^=, x^-\})$), $t \in 0, (m+1)\gamma - 1$ et $(t, \{x^-, x_i^{--}\})$ (resp. $(t, \{x^+, x_i^{++}\})$), $t \in 1, m\gamma\}$, $i = \left\lfloor \frac{t-1}{\gamma} \right\rfloor$, appartient également au γ-couplage. Enfin, pendant l'intervalle de temps $i\gamma + 1, (i+1)\gamma$, nous allons montrer que la clause C_i est vérifiée.

Supposons d'abord que φ est vérifiée. Soit ψ une affectation positive de φ et soit $\chi : \mathcal{C} \to V$ une fonction qui, pour chaque clause C_i, $i \in 0, m-1$, choisit arbitrairement une variable $x \in X$, de manière à ce que l'affectation de x par ψ vérifie C_i, et retourne x_i^{++} (resp. x_i^{--}) si $\psi(x) = \texttt{true}$ (resp. $\psi(x) = \texttt{false}$). Soit

$$\begin{aligned}
\mathcal{M} &= \{\Gamma_\gamma(i \cdot \gamma, x^=, x^+) \mid x \in X, \psi(x) = \texttt{true}, i \in 0, m\} \\
&\cup \{\Gamma_\gamma(i \cdot \gamma, x^=, x^-) \mid x \in X, \psi(x) = \texttt{false}, i \in 0, m\} \\
&\cup \{\Gamma_\gamma(i \cdot \gamma + 1, x^-, x_i^{--}) \mid x \in X, \psi(x) = \texttt{true}, i \in 1, m\} \\
&\cup \{\Gamma_\gamma(i \cdot \gamma + 1, x^+, x_i^{++}) \mid x \in X, \psi(x) = \texttt{false}, i \in 1, m\} \\
&\cup \{\Gamma_\gamma(i \cdot \gamma + 1, c, \chi(C_t)) \mid i \in 1, m\}.
\end{aligned}$$

On peut vérifier que $\mathcal{M}$ est un γ-couplage de L de taille $(2m+1)n + m$.

Supposons maintenant que L contient un γ-couplage $\mathcal{M}$ de taille $(2m+1)n+m$. Nous allons prouver les faits suivants afin de construire une affectation positive de φ. Pour chaque $x \in X$, $\mathcal{M}$ contient au plus $m+1$ γ-arêtes adjacentes à x^+ (resp. x^-).

Démonstration. Ce résultat découle du fait que $T = 0, (m+1)\gamma - 1$ est de taille $(m+1)\gamma$ et ne peut donc pas être divisé en $m+2$ ensembles deux à deux disjoints de taille γ. □

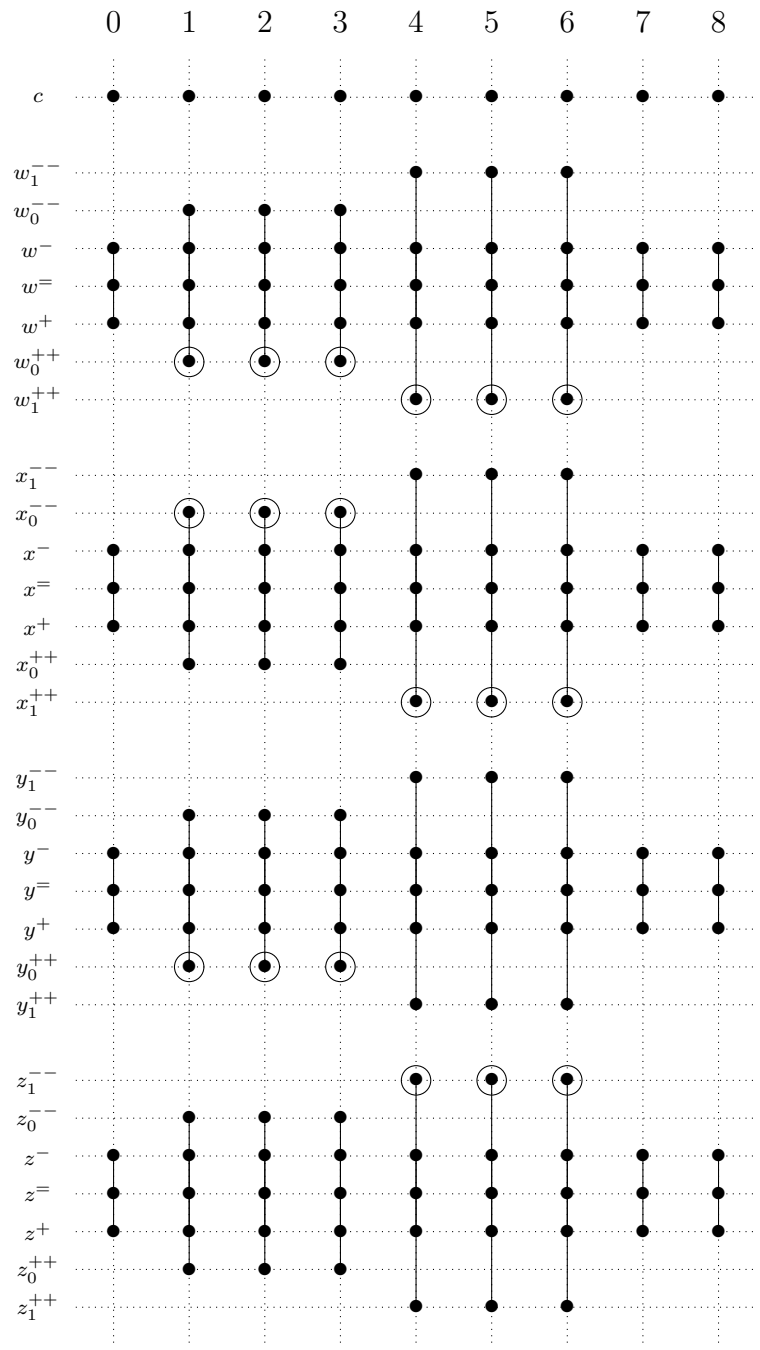


FIGURE 4.8 – Le flot de liens L construit à partir de $\varphi = (w \vee \overline{x} \vee y) \wedge (w \vee x \vee \overline{z})$ et $\gamma = 3$. Ici $T = 0, 8$ et une arête $(t, \{u, v\})$ de L est représentée par une arête verticale suivant la colonne correspondant au temps t partant de la ligne horizontale correspondant à u à la ligne horizontale correspondant à v. Pour des raisons de lisibilité, les arêtes adjacentes avec c ne sont pas dessinées. Au lieu de cela, nous avons encerclé les sommets qui sont adjacents à c pour tout instant t.

Soit $x \in X$, si $\Gamma_\gamma(0, x^=, x^+) \notin \mathcal{M}$ (resp. $\Gamma_\gamma(0, x^=, x^-) \notin \mathcal{M}$), alors $\mathcal{M}$ contient au plus m γ-arêtes adjacentes à x^+ (resp. x^-).

Démonstration. $\Gamma_\gamma(0, x^=, x^+)$ est la seule γ-arête de L contenant l'arête $e_x^0 = (0, \{x^=, x^+\})$. Donc l'arête e_x^0 n'appartient à aucune γ-arête de $\mathcal{M}$. Ainsi, les γ-arêtes de $\mathcal{M}$ qui sont adjacentes avec x^+ doivent exister dans l'intervalle de temps $I = 1, (m+1)\gamma - 1$, dont la taille est $(m+1)\gamma - 1$. Donc, I ne peut pas être divisé en $m+1$ ensembles deux à deux disjoints de taille γ. □

$\mathcal{M}$ contient exactement m γ-arêtes adjacentes à c et contient exactement $2m+1$ γ-arêtes adjacentes à x^+ ou x^-, pour tout $x \in X$.

Démonstration. Puisque $\mathcal{M}$ est un γ-couplage, nous savons pour tout $x \in X$ que $\Gamma_\gamma(0, x^=, x^+) \notin \mathcal{M}$ ou $\Gamma_\gamma(0, x^=, x^-) \notin \mathcal{M}$. Donc, Claim 4.1 et Claim 4.1 impliquent que, pour tout $x \in X$, $\mathcal{M}$ contient au plus $2m+1$ γ-arêtes adjacentes à x^+ ou x^-. De plus, par construction $\mathcal{M}$ contient au plus m γ-arêtes adjacentes à c et L ne contient aucune arête de la forme $(t, \{x^+, y^-\})$, $(t, \{x^+, y^+\})$, $(t, \{x^-, y^-\})$, $(t, \{x^+, c\})$, ou $(t, \{x^-, c\})$, pour tout $x, y \in X$. □

Nous remarquons par construction le fait suivant. Si $\mathcal{M}$ contient une γ-arête adjacente à x_i^{++} pour certain $x \in X$ et $i \in 0, m-1$, alors cette γ-arête doit être soit $\Gamma_\gamma(i\gamma+1, c, x_i^{++})$, soit $\Gamma_\gamma(i\gamma+1, x^+, x_i^{++})$. De plus, Claim 4.1 ci-dessous nous indique plus de précision dans le cas où $\Gamma_\gamma(i\gamma+1, x^+, x_i^{++}) \in \mathcal{M}$. Soit $x \in X$. Si $\mathcal{M}$ contient une γ-arête $\Gamma_\gamma(i\gamma+1, x^+, x_i^{++})$ (resp. $\Gamma_\gamma(i\gamma+1, x^-, x_i^{--})$) pour certain $i \in 0, m-1$, alors $\mathcal{M}$ contient au plus m γ-arêtes adjacentes à x^+ (resp. x^-).

Démonstration. Soit $x \in X$. Soit i la première valeur telle que $\Gamma_\gamma(i\gamma+1, x^+, x_i^{++}) \in \mathcal{M}$. Comme γ n'est pas un diviseur de $i\gamma+1$, pendant l'intervalle $0, i\gamma$, au moins une arête $e_x^t = (t, \{x^=, x^+\})$, $t \in 0, i\gamma$ n'appartient pas à $\mathcal{M}$. Donc, la γ-arête de $\mathcal{M}$ qui est adjacente à x^+ doit exister dans l'intervalle de temps $I = T \setminus t$, dont la taille est $(m+1)\gamma - 1$. Donc, I ne peut pas être divisé en $m+1$ ensembles deux à deux disjoints de taille γ. □

Soit $x \in X$. D'après Claim 4.1, nous savons que $\mathcal{M}$ contient exactement $2m+1$ γ-arêtes adjacentes à x^+ ou x^-. Donc, pour x^+, resp. x^-, nous savons que $\mathcal{M}$ contient exactement $m+1$ γ-arêtes adjacentes à x^+, resp. x^- (principe de Dirichlet). D'après Claim 4.1, nous savons que $\{\Gamma_\gamma(i\gamma, x^=, x^+) \mid i \in 0, m\} \subseteq \mathcal{M}$. Comme $\mathcal{M}$ est un γ-couplage contenant m γ-arêtes qui sont toutes adjacentes à x^-, nous savons également que $\{\Gamma_\gamma(i\gamma+1, x^-, x_i^{--}) \mid i \in 0, m-1\} \subseteq \mathcal{M}$.

Pour chaque variable $x \in X$, nous affectons x à `vrai` (resp. `faux`) si $\Gamma_\gamma(0, x^=, x^+) \in \mathcal{M}$ (resp. $\Gamma_\gamma(0, x^=, x^-) \in \mathcal{M}$). Soit φ la fonction d'affectation ainsi définie. Soit $i \in 0, m-1$. Soit $x \in X$ tel que nous avons soit $\Gamma_\gamma(i\gamma+1, c, x_i^{++}) \in \mathcal{M}$, soit $\Gamma_\gamma(i\gamma+1, c, x_i^{--}) \in \mathcal{M}$. Sans perte de généralité supposons que $\Gamma_\gamma(i\gamma+1, c, x_i^{++}) \in \mathcal{M}$, ie. x est une variable positive dans C_i. Nous avons donc $\Gamma_\gamma(i\gamma+1, x^+, x^{++}) \notin \mathcal{M}$, qui implique également $\{\Gamma_\gamma(i'\gamma, x^=, x^+) \mid i' \in 0, m\} \subseteq \mathcal{M}$. Nous déduisons que x est affectée à `vrai` par φ et C_i est satisfaite. □

4.2 Algorithme d'approximation

Dans la théorie classique des graphes, nous savons que tout couplage maximal est également une 2-approximation d'un couplage maximum, voir par exemple [14, Exercice 35.4]. Heureusement, il en est de même pour les flots de liens. Précisément, dans cette section, nous

adoptons l'approche glouton –trouver un γ-couplage maximal– afin de fournir un algorithme de 2-approximation pour γ-COUPLAGE.

Soit $L = (T, V, E)$ un flot de liens. Soit $\mathcal{P}$ l'ensemble de toutes les γ-arêtes de L. Notez que ces γ-arêtes ne sont pas indépendantes les uns des autres, au contraire, ils se chevauchent fortement. Soit $\preceq$ un ordre total sur les éléments de $\mathcal{P}$ tels que pour toute paire (Γ_1, Γ_2) d'éléments de $\mathcal{P}$, si $\Gamma_1 = \Gamma_\gamma(t_1, u_1, v_1)$, $\Gamma_2 = \Gamma_\gamma(t_2, u_2, v_2)$ et $t_1 < t_2$, alors nous avons $\Gamma_1 \preceq \Gamma_2$.

On note $\mathcal{A}$ l'algorithme glouton suivant. L'algorithme commence par $\mathcal{M} = \emptyset$, $\mathcal{Q} = \mathcal{P}$ et une fonction $\rho : V \times T \to \{0, 1\}$ tels que pour chaque $(t, v) \in T \times V$, $\rho(t, v) = 0$. Le but de ρ est de garder une trace des sommets temporels contenus dans un γ-arête de $\mathcal{M}$. Tant que $\mathcal{Q}$ n'est pas vide, l'algorithme sélectionne Γ, la γ-arête de $\mathcal{Q}$ minimum pour $\preceq$, et la supprime de $\mathcal{Q}$. Soit K l'ensemble des 2γ sommets temporels contenus dans Γ. Si, pour chaque $(t, v) \in K$, $\rho(t, v) = 0$, alors l'algorithme ajoute Γ à $\mathcal{M}$, sinon il ne fait rien pendant cette itération. Pour chaque $(t, v) \in K$, l'algorithme définit $\rho(t, v) = 1$ et réitère. Si $\mathcal{Q} = \emptyset$, il retourne $\mathcal{M}$.

En supposant que $\mathcal{P}$ est trié sur la dimension temporelle (ce qui correspond à la plupart des cas d'utilisation réel), cet algorithme s'exécute en $\mathcal{O}(n\tau + m)$, où $\tau = |T|$, $n = |V|$, $m = |E|$, et γ est une constante cachée dans le $\mathcal{O}$.

Pour tout γ-couplage $\mathcal{M}$, nous définissons les *sommets temporels les plus tardifs* de $\mathcal{M}$, notés $\texttt{bot}(\mathcal{M})$, l'ensemble $\{(t + \gamma - 1, u), (t + \gamma - 1, v) \mid \Gamma_\gamma(t, u, v) \in \mathcal{M}\}$. Nous montrons ci-après dans Lemma 4 le rôle crucial de ces sommets tardifs définis par le γ-couplage calculé par $\mathcal{A}$.

Lemma 4. *Soit γ un entier positif, L un flot de liens, et $\mathcal{M}$ un γ-couplage calculé par l'algorithme $\mathcal{A}$ en prenant L en entrée. Si $\mathcal{M}'$ est un γ-couplage de L, alors chaque γ-arête de $\mathcal{M}'$ contient au moins un sommet temporel de $\texttt{bot}(\mathcal{M})$.*

Démonstration. Nous notons que toute γ-arête de $\mathcal{M}$ contient deux sommets temporels de $\texttt{bot}(\mathcal{M})$, et, par conséquent, au moins un. Soit Γ' une γ-arête de $\mathcal{M}'$ qui n'est pas dans $\mathcal{M}$. Soit $\mathcal{M}^* \subseteq \mathcal{M}$ l'ensemble de toute γ-arête Γ^* de $\mathcal{M}$ telle qu'il existe un sommet temporel (t, u) qui appartient à la fois à Γ' et à Γ^*. Supposons que $\Gamma' = \Gamma_\gamma(t, u, v)$. S'il existe $\Gamma^* \in \mathcal{M}$ tel que $\Gamma^* = \Gamma_\gamma(t', u, v')$ et t' *leqt*, alors nous savons que $(t' + \gamma - 1, u) \in \texttt{bot}(\mathcal{M})$ appartient à Γ'. Sinon, nous aurait pour chaque $\Gamma^* \in \mathcal{M}^*$ que $\Gamma^* = \Gamma_\gamma(t', u, v')$, $t' > t$. Cependant, ce fait est contradictoire avec la construction de $\mathcal{A}$. Ceci conclut la preuve. □

Le Lemma 4 joue un rôle fondamental dans la preuve du Théorème 5 de la section suivante. Nous obtenons également le résultat suivant. $\mathcal{A}$ est un algorithme de 2-approximation du problème γ-COUPLAGE.

Démonstration. Soit L le flot de lien donné en argument à l'algorithme $\mathcal{A}$ et $\mathcal{M}$ la solution retournée par cet algorithme. Soit $\mathcal{M}'$ un γ-couplage de L. Comme $|\texttt{bot}(\mathcal{M})| = 2|\mathcal{M}|$, deux γ-arêtes de $\mathcal{M}'$ ne peuvent pas contenir le même sommet temporel, et, par Lemma 4, chaque γ-arête de $\mathcal{M}'$ contient au moins un élément de $\texttt{bot}(\mathcal{M})$, nous déduisons que $|\mathcal{M}'| \leq 2|\mathcal{M}|$. □

4.3 Algorithme de kernelisation

Le problème γ-COUPLAGE a un *noyau* s'il existe une fonction calculable $f : \mathbb{N} \to \mathbb{N}$ et un algorithme en temps polynomial qui prend en entrée une instance (L, k) de γ-COUPLAGE et produit une instance (L', k') telle que : $k' \leq k$; $|L'| \leq f(k)$; et (L', k') donne une réponse

positive à γ-COUPLAGE si et seulement si (L, k) donne une réponse positive à γ-COUPLAGE. Dans ce cas, l'algorithme s'appelle un *algorithme de kernelisation* pour γ-COUPLAGE [19].

Nous montrons maintenant un algorithme de kernelisation pour γ-COUPLAGE par un processus d'élagage directement basé sur Lemma 4. Par commodité, supposons qu'une γ-arête Γ soit incident avec un sommet temporel (t, u) quand il existe un sommet $v \neq u$ tel que $(t, \{u, v\}) \in \Gamma$. L'idée principale est la suivante. Tout d'abord, nous calculons l'ensemble S de tous les sommets temporels les plus tardifs d'un γ-couplage produit par l'algorithme d'approximation $\mathcal{A}$ défini dans la section précédente. Ensuite, nous filtrons le flot de liens initial en ne conservant que les arêtes appartenant à un γ-arête adjacente à un sommet temporel de S. Plus précisément, nous prouvons le résultat suivant.

Theorem 5. *Il existe un algorithme en temps polynomial qui, pour chaque instance (L, k), renvoie une instance positive qui correspond correctement au fait que L contient un γ-couplage de taille k, ou renvoie une instance équivalence (L', k) telle que le nombre d'arêtes de L' soit $2(k-1)(2k-1)\gamma^2$.*

Démonstration. Soit $L = (T, V, E)$ un flot de liens et k un entier. Nous appliquons d'abord l'algorithme $\mathcal{A}$ sur L. Soit $\mathcal{M}$ le γ-couplage produit par l'algorithme et soit $\ell = |\mathcal{M}|$. Si $\ell \geq k$, nous avons déjà une solution, donc, nous retournons une instance vraie. Si $\ell < \frac{k}{2}$, alors, par Corollaire 4.2, nous savons que l'instance ne contient pas de γ-couplage de taille k, donc, nous retournons une instance fausse. Nous supposons maintenant que $\frac{k}{2} \leq \ell < k$.

Le Lemme 4 justifie que nous pouvons nous concentrer sur les sommets temporels de $\texttt{bot}(\mathcal{M})$ afin de trouver le noyau demandé. Nous allons construire un ensemble $\mathcal{P}$ de γ-arêtes et nous allons montrer que toute arête e, qui ne se trouve pas dans une γ-arête de $\mathcal{P}$, est inutile dans le calcul d'un γ-couplage de taille k. Pour chaque $(t, u) \in \texttt{bot}(\mathcal{M})$, et pour chaque t' tel que $\max(0, t-\gamma+1) \leq t' \leq t$, on considère l'ensemble $\mathcal{S}(t', u)$ de chaque γ-arêtes, existant dans L, de la forme $\Gamma_\gamma(t', u, v)$ avec $v \in V$. Si l'ensemble $\mathcal{S}(t', u)$ a une taille d'au plus $2k-1$, nous ajoutons tous les éléments de $\mathcal{S}(t', u)$ à $\mathcal{P}$. Sinon, nous sélectionnons $2k-1$ éléments de $\mathcal{S}(t', u)$ que nous ajoutons à $\mathcal{P}$. Dans les deux cas, on note $\mathcal{S}'(t', u)$ l'ensemble des éléments de $\mathcal{S}(t', u)$ que nous avons ajoutés à $\mathcal{P}$. Ceci termine la construction de $\mathcal{P}$. Comme $|\texttt{bot}(\mathcal{M})| = 2\ell$ et pour chaque élément de $\texttt{bot}(\mathcal{M})$, nous avons ajouté au plus $(2k-1)\gamma$ γ-arêtes à $\mathcal{P}$, nous avons que $|\mathcal{P}| \leq 2\ell(2k-1)\gamma \leq 2(k-1)(2k-1)\gamma$.

Nous prouvons maintenant que si L contient un γ-couplage $\mathcal{M}'$ de taille k, il contient également un γ-couplage $\mathcal{M}''$ de taille k tels que $\mathcal{M}'' \subseteq \mathcal{P}$. Soit $\mathcal{M}'$ un γ-couplage de L de taille k telle que $p = |\mathcal{M}' \setminus \mathcal{P}|$ soit minimum. Nous devons prouver que $p = 0$. Supposons que $p \geq 1$. Soit Γ une γ-arête dans $\mathcal{M}' \setminus \mathcal{P}$. Soit (t, u) un sommet temporel de $\texttt{bot}(\mathcal{M})$ contenu dans Γ. On sait par le Lemme 4 que ce sommet temporel existe. Supposons que $\Gamma = \Gamma_\gamma(t', u, v)$ pour certains $v \in V$ et certains t' tels que $\max(0, t-\gamma+1) \leq t' \leq t$. Comme $\Gamma \notin \mathcal{P}$, nous avons que $\Gamma \in \mathcal{S}(t', u) \setminus \mathcal{S}'(t', u)$, et ainsi $|\mathcal{S}'(t', u)| = 2k-1$. Soit $N_{\mathcal{S}'}(t', u)$ l'ensemble des sommets w de $V \setminus \{u\}$ tels qu'une γ-arête de $\mathcal{S}'(t', u)$ est adjacente à w. Comme $\mathcal{M}' \setminus \{\Gamma\}$ est de taille $k-1$, les γ-arêtes qu'il contient peuvent être adjacentes à au plus $2k-2$ sommets. Cela signifie qu'il existe $w \in N_{\mathcal{S}'}(t', u)$ tel qu'aucune γ-arête de $\mathcal{M}' \setminus \{\Gamma\}$ est adjacente à w. Ainsi, $(\mathcal{M}' \setminus \{\Gamma\}) \cup \{\Gamma_\gamma(t', u, w)\}$ est un γ-couplage de taille k . Comme $\Gamma \notin \mathcal{P}$ et $\Gamma_\gamma(t', u, v) \in \mathcal{P}$, cela contredit le fait que p est minimum.

Nous pouvons maintenant définir le flot de liens $L' = (T, V, E')$ tel que $E' = \{e \in E \mid \exists \Gamma \in \mathcal{P} : e \in \Gamma\}$. Comme $|\mathcal{P}| \leq 2(k-1)(2k-1)\gamma$ et comme tout élément de $\mathcal{P}$ est une γ-arête, nous déduisons que $|E'| \leq 2(k-1)(2k-1)\gamma^2$. □

4.4 Etude numérique

Pour plus de facilité dans la diffusion, nos algorithmes de 2-approximation et de kernelisation sont tous deux implémentés en Java et JavaScript [2]. Les expériences sont exécutées sur un ordinateur portable standard cadencé à 3,1 Ghz avec une mémoire DDR3 16Go.

4.4.1 Jeux de données

Nous avons effectué nos expériences sur deux types principaux de jeux de données : ceux qui sont générés aléatoirement [3] ; et ceux qui sont collectés à partir d'activités humaines.

Flot de liens créés artificiellement et test de performance :
Afin de générer des ensembles aléatoires d'instances de flot de liens, nous adoptons le point de vue plus réaliste des graphes géométriques aléatoires, plutôt que le modèle classique de Erdös-Rényi, comme suit. Soit un espace euclidien de dimension 2. Nous définissons une particule comme un point dans l'espace . Chaque particule est donnée avec un rayon représentant la distance de communication maximale qu'elle peut avoir avec une autre particule. Ainsi, la particule et son rayon définissent un disque dans l'espace . Soit un ensemble de particules, donné avec la même valeur pour leur rayon. Nous partitionnons l'ensemble en n parties, $= P_1 \cup P_2 \cup \cdots \cup P_n$, de taille à peu près égale. Nous allons construire un flot de liens $L = (T, V, E)$ comme suit. Soit $V = \{P_1, P_2, \ldots, P_n\}$. Au temps zéro, soit $E_0 = \{(0, \{P_i, P_j\}) \mid \exists a \in P_i \wedge b \in P_j,$ *la distance entre a et b est inférieure à leur rayon*$\}$. En d'autres termes, s'il y a au moins deux particules à portée de communication $a \in P_i, b \in P_j$ dans deux groupes différents $P_i \neq P_j$ (à l'heure zéro), nous considérons qu'il y a une arête au temps zéro entre P_i et P_j. Chaque particule a une vitesse qui est définie comme suit. Premièrement, la vitesse d'une particule au temps t est une fraction de la vitesse au temps $t-1$ de cette particule (frottement). Deuxièmement, tous les vecteurs de vitesse sont soumis à un petit facteur supplémentaire aléatoire (facteur de marche aléatoire modélisant les conditions de vent). Enfin, nous tronquons chaque vecteur de vitesse afin de nous assurer que la norme du vecteur est inférieure à une vitesse maximale de particule donnée (limites physiques). Nous avons ensuite laissé le système évoluer pendant un laps de temps donné, que nous appelons également T. A chaque instant $t \in T$, nous définissons, de la même manière que précédemment, les arêtes au temps t $E_t = \{(t, \{P_i, P_j\}) \mid \exists a \in P_i \wedge b \in P_j,$ *la distance entre a et b est inférieure à leur rayon*$\}$. Enfin, nous définissons notre flot de liens comme $E = \bigcup_{t \in T} E_t$.

Grosso modo, augmenter l'un des trois paramètres définis par le rayon de la particule, la cardinalité de et la vitesse maximale de la particule produit le même effet sur le flux de liens généré, c'est-à-dire qu'il génère un flux de liens dense. Les données générées nous permettent de tester notre code, et en particulier de vérifier que l'approximation et le temps d'exécution de la kernelisation sont raisonnables sur les données volumineuses. Par exemple, avec des données contenant des centaines de milliers d'arêtes temporelles, notre temps d'exécution est inférieur à dix secondes, voir Fig. 4.9.

Il est intéressant de noter dans le graphique de gauche de la Fig. 4.9, que nous avons, pour certaines données avec un grand nombre d'arêtes temporelles mais sans γ-arêtes, un temps

2. Le code source est disponible sur
`https://github.com/antoinedimitriroux/Temporal-Matching-in-Link-Streams`
3. Lien direct vers l'interface graphique du générateur :
`https://antoinedimitriroux.github.io`

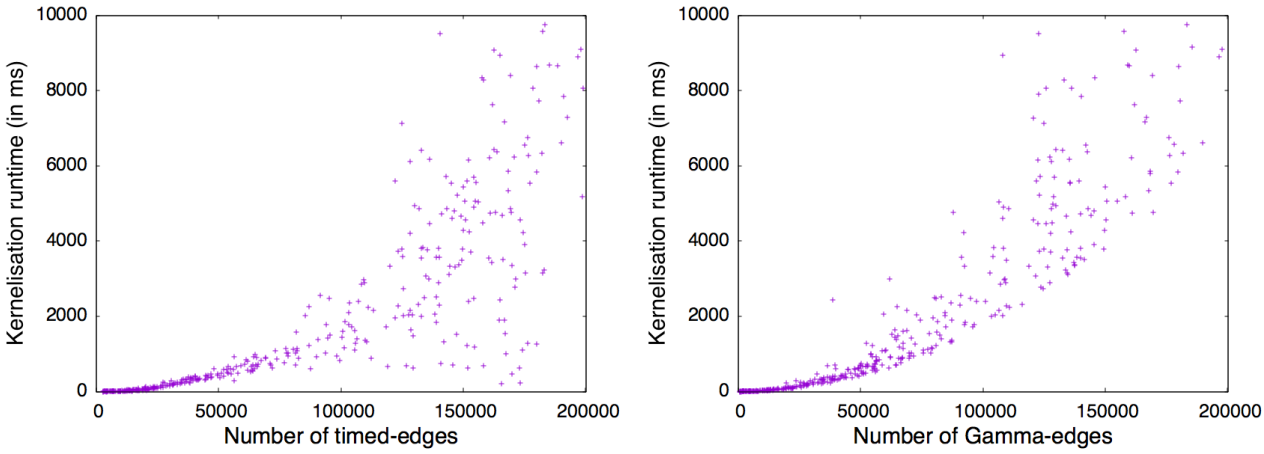


FIGURE 4.9 – Test de performance sur les données générées, avec $\gamma = 5$. Bien que le nombre de γ-arêtes puisse être extrêmement élevé comparé aux arêtes temporelles (à cause des cas de chevauchement), nous avons paramétré le générateur pour qu'ils aient la même apparence dans le graphique de gauche et dans le graphique de droite. Ce n'est en aucun cas une propriété générale.

d'exécution très rapide. Nous discutons également de ce phénomène sur des ensembles de données du monde réel, en Fig. 4.10 et 4.11. Pour cette raison, il est beaucoup plus intéressant d'examiner le temps d'exécution en fonction du nombre de γ-arêtes du flot de liens pris en entrée, cf. le graphique de droite dans la Fig. 4.9. Pour faciliter la comparaison visuelle, nous avons essayé de paramétrer le générateur de manière à ce que la plupart des instances générées aient à peu près le même nombre d'arêtes temporelles et de γ-arêtes. Par exemple, la plupart des instances avec moins de 100000 arêtes temporelles ont également à peu près le même nombre de γ-arêtes. Cependant, la situation est plus aléatoire pour les instances ayant un nombre d'arêtes temporelles compris entre 100000 et 200000.

Jeux de données du monde réel, avec méthodologie de nettoyage de la donnée :
Nous confrontons également les implémentations de nos algorithmes à deux flots de liens particuliers collectés à partir de données réelles. Dans un ensemble de données, le flot de liens est construit à partir d'informations par courrier électronique collectées auprès de la société Enron [28]. L'autre ensemble de données a été construit en analysant un enregistrement de tournée de Roller à Paris pendant deux périodes de 80 minutes chacune [58]. En raison de la longue durée de ces deux expériences, le nombre de sommets (moins de deux cents personnes dans les deux cas) est négligeable par rapport au nombre de sommets temporels (près d'un million pour Rollernet). Pour un flot de liens $L = (T, V, E)$, nous comparons généralement $|T|$ à $|E|$ pour avoir un aperçu de la densité des liens. Pour une vision complète de $|T|$, $|V|$ et $|E|$ dans les jeux de données, nous renvoyons le lecteur à la Figure 4.14. De plus, nous avons remarqué avec nos jeux de données brutes que les instants où une arête temporelle apparaît de façon consécutive peuvent être très épars, ne laissant aucune chance à un γ-arête d'exister dès que $\gamma > 1$. Par exemple, dans l'expérience Enron (Fig. 4.10), nous pouvons voir qu'il n'y aucun couple d'employés qui continuent à s'écrire 1 courriel par heure pendant 24 heures consécutives. Ceci est certainement du à une inactivité naturelle pendant la nuit. Pour cette raison, nous allons compresser dans le temps nos ensembles de données brutes selon le processus suivant.

δ	$\|T\|$	$\|E\|$	γ	$\|\gamma_E\|$
$3600s = 1h$	27300	21959	24	0
$7200s = 2h$	13650	20962	12	0
$10800s = 3h$	9100	20284	8	0
$14400s = 4h$	6825	19732	6	16
$21600s = 6h$	4550	19071	4	69
$28800s = 8h$	3413	18402	3	335
$43200s = 12h$	2275	17610	2	2667

FIGURE 4.10 – Jeu de données Enron : nombre d'arêtes temporelles et γ-arêtes après compression temporelle. Les valeurs sont prises telles que $\delta * \gamma = 24heures$. En particulier, nous observons que les employés d'Enron ne s'écrirons pas en permanence 1 courriel par heure pendant 24 heures, parce que la société est fermée la nuit. Comparé au nombre $|T|$ d'instants, le nombre $|V|$ de sommets est très petit (moins de deux cents) et n'est pas présenté ici.

Definition 23 (Compression temporelle de la donnée)**.** Pour tout flot de liens $L = (T, V, E)$ et pour tout $1 < \delta < |T|$, nous définissons la *δ-compression* $L_\delta = (T_\delta, V_\delta, E_\delta)$ comme $V_\delta = V$, $T_\delta = \frac{\min T}{\delta}, \frac{\max T}{\delta}$, et

$$E_\delta = \{(t, \{u, v\}) \mid \exists t' \in T : \delta t \leq t' < \delta(t+1) \wedge (t', \{u, v\}) \in E\}.$$

Fig. 4.10 et 4.11 montrent que les paramètres δ et γ ont une influence importante sur le nombre de γ-arêtes. Les Figures 4.10 et 4.11 indiquent également que le processus de compression, en général, diminue le nombre $|E|$ d'arêtes temporelles dans le jeu de données. Cependant, nous soulignons que ce n'est pas nécessairement le cas : Fig. 4.12 illustre deux compressions temporelles différentes du même flot de liens initial, respectivement avec $\delta = 3$ et $\delta = 4$, où il est possible d'obtenir plus d'arêtes même si nous avons un plus grand δ. Néanmoins, l'effet usuel de la compression temporelle est de réduire considérablement le nombre d'arêtes temporelles. Sur les jeux de données Enron et Rollernet, nous devons nous assurer que la compression temporelle ne vide pas complètement le flot de liens de ses arêtes. Heureusement, pour les valeurs réalistes de δ, par ex. $\frac{1}{2}$ semaine pour Enron ou 15 minutes pour Rollernet, il reste encore un grand nombre d'arêtes temporelles après δ-compression, cf. Fig. 4.13. Nous montrons en suite en Fig. 4.14 et 4.15 le temps d'exécution de notre algorithme sur des parties aléatoires des deux jeux de données Enron et Rollernet, où nous observons que notre temps d'exécution est très rapide.

4.4.2 Hypothèse de test

Nous voulons tester trois hypothèses. Pour chaque hypothèse, nous testons les jeux de données décrits ci-dessus et exposons nos résultats en Section 4.4.3 ci-dessous. Nous discutons et concluons notre analyse numérique en Section 4.4.4.

Hypothèse 1. Consistance du formalisme :
Nous voudrions vérifier que γ-COUPLAGE est non-trivial sur les valeurs réalistes de δ et γ. Par exemple, nous supposons, lors de l'extraction de courriels, que la collaboration pendant un mois à un rythme d'au moins deux courriels par semaine (aller-retour) constitue des valeurs réalistes. Dans une étude de la distance de parcours des 2×80 minutes de balade en Roller,

δ	$\|T\|$	$\|E\|$	γ	$\|\gamma_E\|$
$1s$	9977	403834	7200	0
$5s$	1996	127401	1240	0
15	666	77989	480	0
$30s$	333	60919	240	0
$60s = 1m$	167	45469	120	0
$300s = 5m$	34	22484	24	51
$600s = 10m$	17	15808	12	357
$1200s = 20m$	9	10735	6	1893
$1800s = 30m$	6	8324	4	2745
$3600s = 1h$	3	5000	2	3094

FIGURE 4.11 – Jeu de données Rollernet : nombre d'arêtes temporelles et γ-arêtes après compression temporelle. Les valeurs sont prises telles que $\delta * \gamma = 7200s = 2heures$. En particulier, nous observons que chaque personne participant à l'expérience Rollernet s'est éloignée d'une autre personne pendant au moins 1 minute pendant 2 heures. Comparé au nombre $|T|$ d'instants, le nombre $|V|$ de sommets est très petit (inférieur à cent) et n'est pas présenté ici.

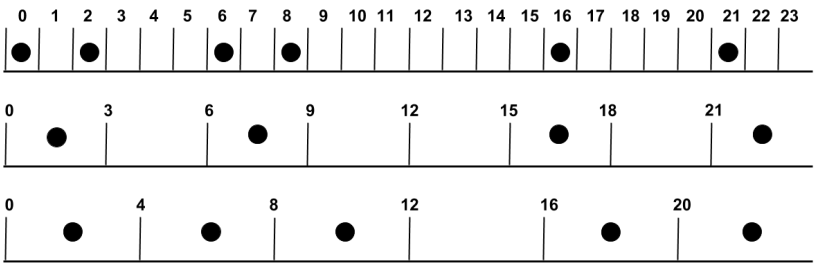


FIGURE 4.12 – Compression temporelle avec $\delta = 3$ et $\delta = 4$, donnant respectivement 4 et 5 arêtes temporelles.

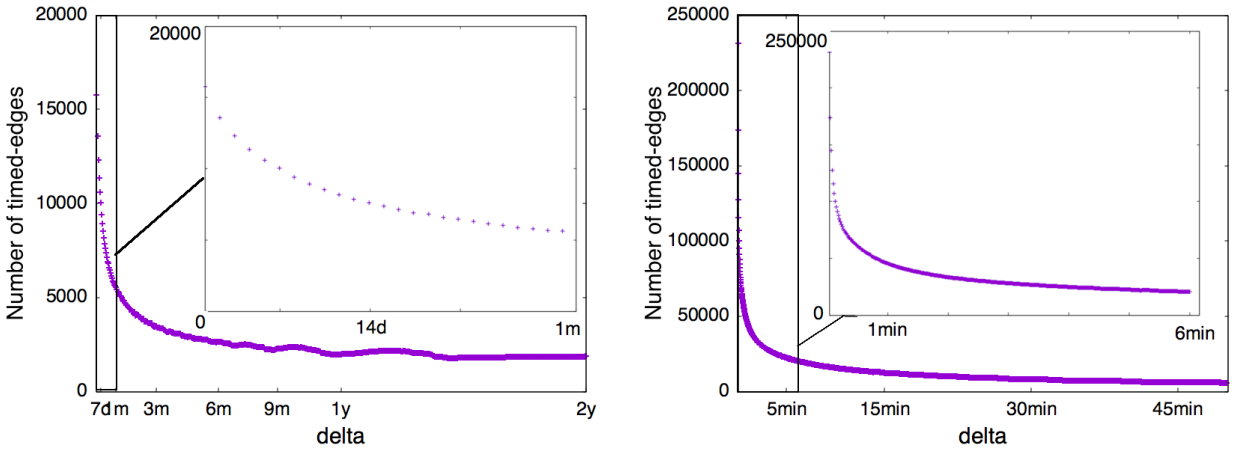


FIGURE 4.13 – Jeu de données Enron (à gauche) et Rollernet (à droite) : arêtes temporelles restants après δ-compression. Par exemple, avec $\delta = \frac{1}{2}$ semaine, le nombre d'arêtes temporelles après δ-compression sur la base Enron dépasse dix mille. Avec $\delta = 15$ minutes, le nombre d'arêtes temporelles après δ-compression sur la base Rollernet dépasse également dix mille. Nous concluons que pour les valeurs réalistes de δ, notre processus de compression temporelle ne réduit pas les données en une instance vide.

nous considérons que collaborer pendant 80 minutes avec une visite de proximité tous les quarts d'heure (approvisionnement en eau / grignotage) constitue des valeurs réalistes.

Hypothèse 2. Qualité de la kernelisation :
Nous reconnaissons que, en plus des constantes cachées sous la notation de Landau, une réduction d'espace de $O(n)$ à $O(k^2)$, lorsque $k \approx \sqrt{n}$ n'a pas tellement de sens. Malheureusement, quand nous considérons que le paramètre de la kernelisation est la taille de la solution, comme le cas de notre lernelisation, on aboutit très souvent au cas où k est numériquement de l'ordre de $\sqrt{n}$. Ceci est particulièrement vrai pour notre étude de couplage temporel, sur les jeux de données Enron et Rollernet. Pour cette raison, notre deuxième hypothèse est que, en plus d'une garantie théorique de réduction de la complexité en espace de $O(n)$ à $O(k^2)$, la résolution de γ-COUPLAGE peut être aidé, numériquement, par l'algorithme de kernelisation que nous avons décrit dans le Théorème 5.

Hypothèse 3. Qualité de l'approximation :
Nous soulignons que COUPLAGE dans un graphe classique est polynomial. Malheureusement, γ-COUPLAGE dans un flot de liens est NP-difficile. Cependant, dans la pratique, de nombreux problèmes NP-complets ne sont pas difficiles pour les données récoltées à partir d'activités humaines. De plus, certains problèmes de ce type peuvent être résolus de manière quasi optimale à l'aide d'algorithmes simples, tels qu'une approche aléatoire ou glouton, ou une combinaison des deux, même sur des données arbitraires. Un exemple populaire est COLORATION [42]. Par conséquent, notre dernière hypothèse est que, dans la pratique, trouver un γ-couplage optimal n'est pas nécessairement difficile. De plus, nous émettons l'hypothèse que l'algorithme de 2-approximation glouton décrit dans le Lemme 4 peut produire un γ-couplage quasi-optimal sur un jeu de données du monde réel, ainsi que sur des jeux de données artificiels qui imitent des jeux de données monde réel.

Flot de liens$_\delta$	$\|V\|$	$\|T\|$	$\|E\|$	$\|\gamma_E\|$	appr(s)	kern(s)	total(s)
$Enron_{1heure}$	150	27300	21959	1991	0.010	0.397	0.408
$Enron_{3heures}$	150	9100	20284	2695	0.018	0.694	0.696
$Enron_{1jour}$	150	1138	16224	4416	0.007	1.76	1.774
$Enron_{3jours}$	150	380	12868	4644	0.007	1.932	1.939
$Enron_{7jours}$	150	163	10028	4812	0.005	1.173	1.179
$Enron_{30jours}$	150	38	5573	2917	0.001	0.147	0.149
$Enron_{90jours}$	150	13	3480	1650	6.6E-4	0.029	0.030
$Rollernet_{1min}$	61	167	45469	24009	0.098	1.696	1.794
$Rollernet_{2mins}$	61	84	33304	17089	0.047	0.561	0.609
$Rollernet_{5mins}$	61	34	22484	13346	0.018	0.140	0.158
$Rollernet_{15mins}$	61	12	12410	8544	0.005	0.044	0.050
$Rollernet_{30mins}$	61	6	8324	5979	0.005	0.032	0.038
$Rollernet_{1heure}$	61	3	5000	3094	0.001	0.007	0.008
Generated	10	50	684	83	4.4E-4	0.001	0.001
Generated	10	100	1384	136	4.8E-4	7.4E-4	0.001
Generated	10	200	2906	322	4.1E-4	0.002	0.002
Generated	20	50	2599	705	0.001	0.004	0.006
Generated	20	100	5326	1508	0.003	0.016	0.020
Generated	20	200	10667	2998	0.004	0.030	0.035
Generated	50	50	15842	6534	0.005	0.034	0.040
Generated	50	100	31113	12876	0.018	0.125	0.144
Generated	50	200	63032	26495	0.054	0.426	0.480
Generated	100	50	53665	32235	0.093	0.280	0.374
Generated	100	100	107524	65145	0.342	1.054	1.396
Generated	100	200	214728	130371	1.437	4.277	5.713

FIGURE 4.14 – Le temps d'exécution requis par notre algorithme de -2–approximation, notre algorithme de kernelisation et le temps d'exécution du processus total. Les valeurs sont prises pour $\gamma = 2$. Le paramètre k de l'algorithme de kernelisation est exactement la taille de la solution trouvée par l'algorithme de 2-approximation. Le temps d'exécution est très rapide, et ne peut être pris que qualitativement à cause des (probables) phénomènes de bruits dans la mise en place du processus et ses variables d'environnement.

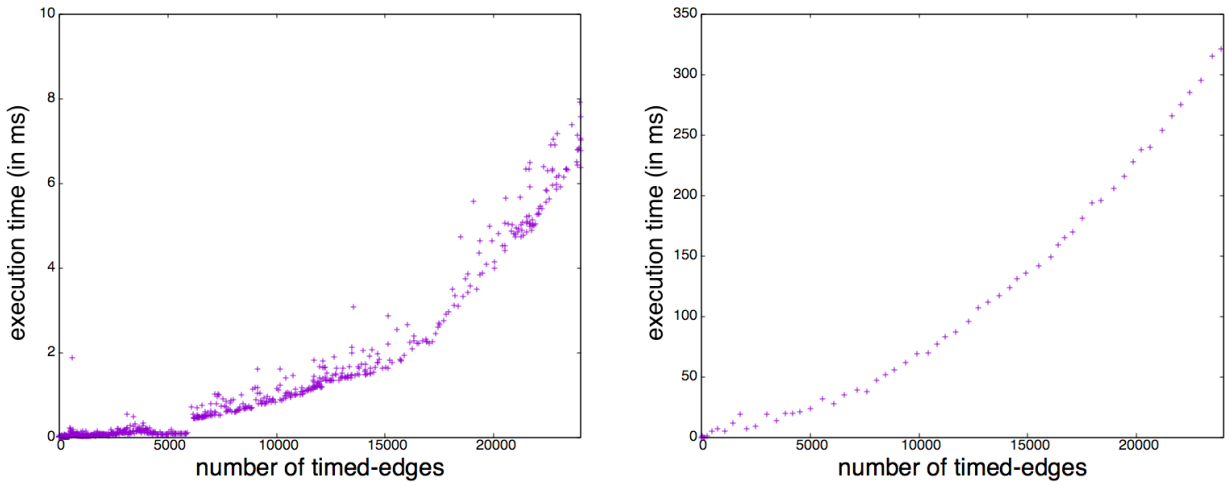


FIGURE 4.15 – Jeu de données Enron (à gauche) et Rollernet (à droite) : exécution de nos algorithmes en fonction du nombre d'arêtes temporelles, avec $\gamma = 2$. Le paramètre k pour la kernelisation est la taille du résultat de la 2-approximation. Le temps d'exécution (très rapide) est probablement sujet à de nombreux bruits. Nous nous référons plutôt à la Fig. 4.9 pour évaluer les performances. Chaque point de la Fig. 4.15 ici est obtenu en tronquant d'abord la donnée brute avec une valeur maximum des instants de temps, puis en appliquant une δ-compression sur le flot de liens ainsi obtenu, avec $\delta = 100$. La seule observation que nous faisons avec cette figure est que, sur un jeu de données du monde réel, notre temps d'exécution total des algorithmes de 2-approximation et de la kernelization est très rapide.

4.4.3 Résultats

Nos algorithmes de 2-approximation et de kernelisation sont tous deux mis en œuvre et comparés aux jeux de données décrits dans les sections précédents.

Observations relatives à Hypothèse 1. Consistance du formalisme :
Les résultats sont donnés dans la Fig. 4.16. On observe sur le jeu de données Enron avec $\delta \approx \frac{1}{2}$ semaine qu'après la δ-compression , le nombre de γ-arêtes pour γ variant de 2 à 10 est : plus de 500 pour $\gamma = 10$; plus de 1000 pour $\gamma = 6$; et plus de 4500 pour $\gamma = 2$. De plus, nous verrons dans le paragraphe suivant que notre analyse numérique ne peut trouver que des γ-couplage dont la taille est environ un cinquième des chiffres précédents. Nous avons également essayé d'autres techniques pour améliorer la taille des γ-couplage mais nous n'avons pas réussi à trouver une différence substantielle. Avec le jeu de données Enron, nous pensons que γ-COUPLAGE pour $\gamma \approx 10$ est une question non-triviale lorsque le taux de compression temporelle est $\delta \approx \frac{1}{2}$ semaine. Ces valeurs traduisent le fait que toute paire de collaborateurs dans le γ-couplage s'échangent continuellement des courriels pendant un mois, à raison d'au moins un courriel par semaine et en moyenne d'au moins deux courriels par semaine.

Observations relatives à Hypothèse 2. Qualité de la kernelisation :
Les résultats sont donnés dans la Fig. 4.17. Le paramètre k de l'algorithme de kernelisation est la taille de la solution trouvée par l'algorithme de 2-approximation. Nous observons que sur des intervalles bien choisis de δ et γ, la kernelisation réduit la taille de la données à moins de 20%. Ceci est particulièrement vrai pour $\gamma \approx 20, 30$ avec $\delta \leq 2$ mois sur Enron ; et $\gamma \leq 10$ avec $\delta \leq 20$ minutes pour Rollernet. Nous observons sur ces valeurs que l'algorithme de kernelization

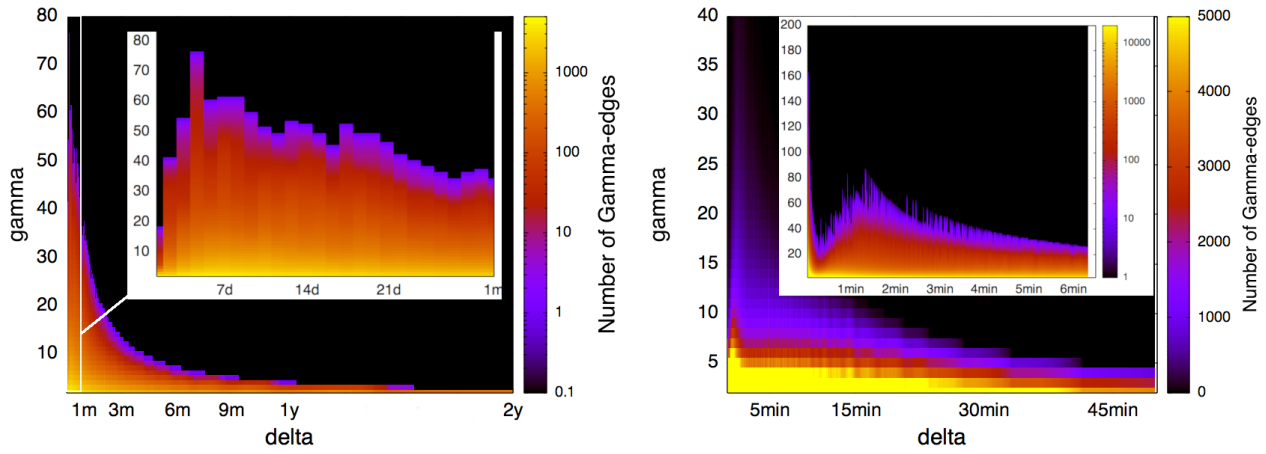


FIGURE 4.16 – Jeu de données Enron (à gauche) et Rollernet (à droite) : nombre de γ-arêtes après δ-compression, pour différentes valeurs de δ et γ.

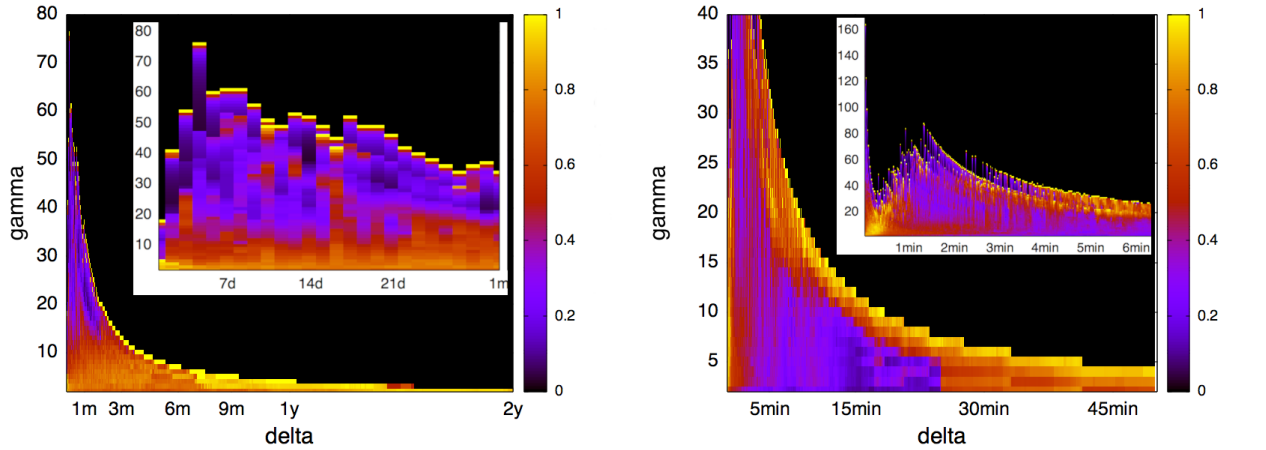


FIGURE 4.17 – Jeu de données Enron (à gauche) et Rollernet (à droite - γ-arêtes) : rapport obtenu en divisant le nombre de γ-arêtes dans le résultat retourné par la kernelisation par le nombre de γ-arêtes de l'instance donnée en entrée à la kernelisation (qui elle est obtenue après δ-compression). Ici, plus la couleur est sombre, mieux est le rapport. Le paramètre k de l'algorithme de kernelisation est la taille de la solution trouvée par l'algorithme de 2-approximation.

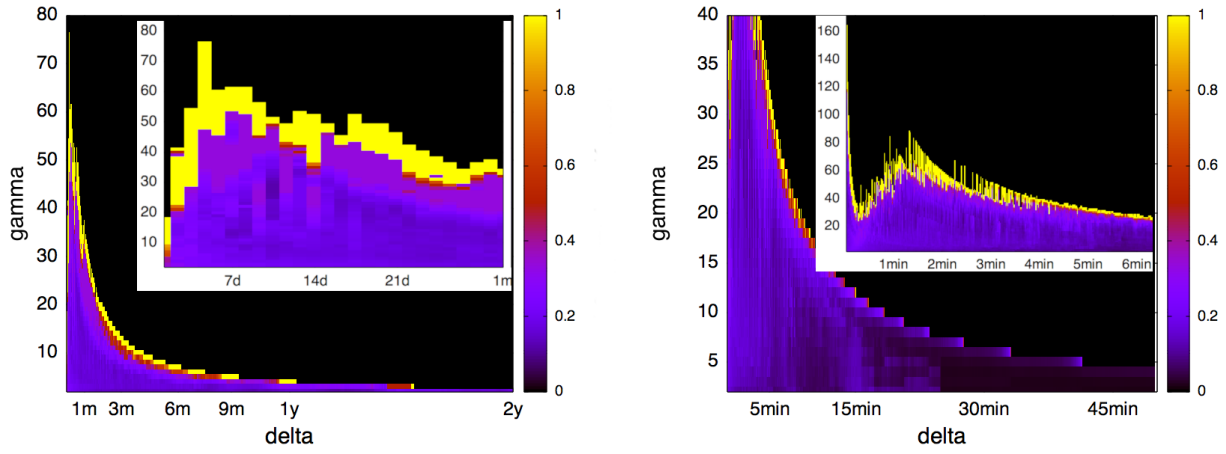


FIGURE 4.18 – Jeu de données Enron (à gauche) et Rollernet (à droite) : rapport obtenu en divisant le résultat de l'algorithme de 2-approximation par le nombre de γ-arêtes dans le résultat retourné par la kernelisation. Ici, plus la couleur est claire, mieux est le rapport : l'approximation résout de manière optimale γ-COUPLAGE dans les régions où le rapport est 100%, ce qui est indiqué par la couleur jaune. Le paramètre k de l'algorithme de kernelisation est la taille de la solution trouvée par l'algorithme de 2-approximation.

réduit le flot de liens initial à une instance équivalente de taille inférieure à 20% la taille de la donnée initiale et parfois inférieure à 10%. Nous concluons que l'hypothèse 2 est vérifiée.

Observations relatives à Hypothèses 3. Qualité de l'approximation :
Les résultats sont donnés à la Fig. 4.18. Le paramètre k de l'algorithme de kernelisation est la taille de la solution trouvée par l'algorithme de 2-approximation. Nous remarquons d'après la définition que l'algorithme de 2-approximation produit une borne inférieure pour γ-COUPLAGE, ce qui correspond à au moins la moitié de la valeur optimale. De plus, l'algorithme de kernelisation donne une borne supérieure triviale pour γ-COUPLAGE en comptant simplement le nombre de γ-arête présentes dans le noyau. Nous remarquons certains régions frontalières dans la Figure 4.18 où ces deux bornes supérieure et inférieure se touchent. Cela signifie que la 2-approximation produit une solution optimale pour γ-COUPLAGE sur ces zones. Cependant, nous observons que pour la plupart des parties de notre jeu de données, l'hypothèse 3 n'est pas confirmée. À ce stade, nos expériences avec l'hypothèse 3 ne permettent pas de conclure. D'autres expériences doivent être effectuées afin de clarifier cette question. Nous pensons cependant que l'hypothèse 3 est généralement fausse. Nous supposons que le facteur 2 de l'algorithme d'approximation est loin d'être optimal.

4.4.4 Discussion

Les résultats de notre expérience sont optimistes quant à l'utilité numérique de la kernelisation pour trouver un couplage temporel. En même temps, ils soulèvent plusieurs questions. Alors que nos expériences nous permettent d'observer que :

— le traitement d'une instance de γ-COUPLAGE par un processus glouton, puis par une kernelisation comme décrit dans Théorème 5, semble être bénéfique à la résolution de ce problème ;
— ce traitement est très rapide lors de l'exécution sur les données du monde réel ;

— ce traitement reste rapide sur les test de performance sur des grandes quantités de données artificielles entrées, utilisant un ordinateur portable courant : moins de dix secondes pour traiter une instance contenant quelques centaines de milliers d'arêtes temporelles ;

ils témoignent également qu'il reste encore des travaux à faire pour approfondir les recherches sur γ-COUPLAGE, notamment que :

— le problème d'optimisation est NP-difficile ;
— les preuves numériques d'optimalité du résultat de la 2-approximation ne sont disponibles que sur des données très marginales dans les jeux de données Enron et Rollernet (cf. Fig 4.18) ;
— bien qu'il soit vrai que l'algorithme de kernelisation aide à réduire la données à $10-20\%$ pour des paramètres intéressants sur les jeux de données Enron et Rollernet, nous ne savons toujours pas comment trouver ensuite un γ-couplage dans le noyau autre que le résultat de la 2-approximation ;
— en particulier, nous ne savons pas si le facteur 2 de l'algorithme d'approximation peut être amélioré.

4.5 Conclusion et perspectives

Nous introduisons la notion de couplage temporel dans un flot de liens. Malheureusement, le problème de calculer un couplage temporel de taille maximum, appelé γ-COUPLAGE, s'avère être NP-difficile. Nous montrons ensuite un algorithme de kernelisation pour γ-COUPLAGE paramétré par la taille de la solution. Notre processus produit des noyaux quadratiques. Pour obtenir l'algorithme de kernelisation, nous fournissons également un algorithme de 2-approximation pour γ-COUPLAGE. Nous pensons que les mêmes techniques s'appliquent à une large classe de problèmes d'ensemble de type *hitting sets* dans les flot de liens.

Conclusion

Cette thêse concernant un code correcteur pour les transmissions unidirectionnelles par paquets, développé chez Thales durant mon année d'apprentissage en Master 2, s'est située à l'intersection entre la théorie et la pratique. Cette thêse se veut donc comme une sorte de manuel d'utilisation de ce code correcteur, Fauxtraut, ainsi que d'un ensemble de recherches plus générales concernant notamment la théorie des graphes.

Les problèmes posés par nos différents cas d'utilisation nous ont conduits à développer un code correcteur performant, dont la complexité est linéaire en la taille des informations à encoder, permettant de l'utiliser sur du matériel disposant de peu de puissance de calcul ou de mémoire. Il s'agit d'un code correcteur linéaire pour le canal à effacements de paquets, particulièrement efficace lorsque les effacements ont tendance à apparaître de manière groupée.

Il apparaît que ce code correcteur est bien adapté à notre cas d'utilisation. Il le serait cependant moins dans d'autres cas que nous avons présentés : dans le cas d'un grand taux de perte au niveau binaire, la transmission par paquets n'est plus pertinente, et il nous faudrait alors utiliser des codes MDS tels que les codes de Reed-Solomon. Dans le cas d'un canal de transmission dont nous ne connaîtrions pas le taux d'effacement, nous devrions utiliser des codes sans rendement, tels que les codes LT par exemple. Nous avons formulé un certain nombre de mesures qui pourraient être mises en oeuvre pour permettre l'utilisation de notre code pour d'autres types de canaux de transmission. Nous proposons par exemple une version légèrement modifiée de ce code, qui permet de l'utiliser sur une transmission par paquets, dont les paquets ne disposeraient pas de checksum : le code se comporte alors comme un code pour le canal à erreurs. Le récepteur doit d'abord déterminer quels paquets sont erronés avant de pouvoir les réparer, ce qui est rendu possible avec cette version du code, qui repose sur une utilisation élégante du théorème des restes Chinois. Nous espérons que cette thêse permettra de fiabiliser encore plus la transmission via la diode réseau Elips-SD que propose Thales.

Nous avons présenté les pseudo-codes de l'algorithme d'encodage et de décodage, un grand nombre de simulations permettant de se figurer comment fonctionne ce code suivant différents cas de figure et différents métriques. Nous avons aussi analysé les différents aspects de ce code, tels que sa consommation mémoire, sa complexité, ou sa capacité de correction. L'étude des différentes métriques concernant sa complexité (temps, mémoire) s'est avérée relativement simple, le code étant un code correcteur linéaire comparable à des codes LDPC par exemple. Les pseudo-codes nous indiquent directement la consommation mémoire maximale, et les calculs à réaliser lors des algorithmes d'encodage et décodage.

A contrario, l'étude de la capacité de correction du code s'est avérée plus complexe, posant notamment des questions encore non résolues de la théorie des graphes. Elle correspond à la probabilité que le sous-graphe d'un graphe biparti soit une forêt, ou au contraire qu'il contienne au moins un cycle : dans le premier cas, le code permet de décoder convenablement un bloc, alors que dans le second le décodage échoue. La méthode proposée par Rebecca Stones présentée

dans cette thêse nous permet de calculer exactement cette probabilité pour des graphes bipartis et des sous-graphes de petites tailles, en dénombrant l'ensemble des forêts de taille fixée dans un graphe de Rook. En interpolant ces différentes valeurs, nous sommes parvenus à proposer un certain nombre de formules closes valides pour des graphes de très grandes tailles (mais des sous-graphes restant relativement petits). Ceci nous permet de calculer la probabilité exacte que le décodage réussisse pour un paramétrage de notre code et pour un nombre de paquets à décoder fixé. Cependant, cette méthode ne fonctionne que dans le cas ou le paramètre P de notre code ne contient que deux éléments, paramétrage que nous n'utilisons pas en pratique. Pour des ensembles P comportant deux éléments ou plus, nous proposons des bornes inférieure et supérieure ainsi qu'une formule close tendant asymptotiquement vers la probabilité de décoder convenablement un bloc.

Nous avons aussi présenté une méthode permettant d'obtenir automatiquement de bons paramétrages pour notre code, via l'utilisation de la fonction de Landau. La fonction de Landau nous permet de connaître l'ensemble de nombres dont le plus petit commun multiple est maximal pour une somme fixée. Ainsi, cette fonction nous permet de calculer l'ensemble P dont la somme $\Sigma(P)+1$ correspond au nombre de paquets de redondance que nous ajoutons à un bloc, et dont $PPCM(P)$ correspond au nombre maximum de paquets sources que l'on peut encoder. Afin de calculer cette fonction, nous utilisons l'algorithme proposé par Marx Deléglise, Jean-Louis Nicolas et Paul Zimmerman, qui nous retourne des valeurs pouvant atteindre des milliards. Il apparaît que ces ensembles sont de très bons candidats en tant que paramètres de notre code. Cependant, la fonction de Landau pose problème pour certaines valeurs de $\Sigma(P)+1$, et retourne des ensembles inutilisables en pratique. De plus, il serait intéressant pour le cas d'utilisation de pouvoir fixer le cardinal de P, ce qui n'est pas le cas pour la fonction de Landau. Là-encore, un certain nombre de questions restent ouvertes : pour deux entiers s et c, quel est l'ensemble P dont la somme vaut s, le cardinal vaut c, et le $PPCM$ est maximal ? Répondre à cette question nous permettrait d'obtenir les meilleurs paramétrages envisageables pour notre code.

Enfin, en collaboration avec Binh-Minh Bui-Xuan et Julien Baste nous avons étudié un problème NP-complet de la théorie des graphes temporels, appelé couplage temporel. Les graphes temporels sont une classe de graphe évoluant dans le temps, pour lesquels les arêtes apparaissent ou disparaissent à certains instants. Ce sont des graphes particulièrement adaptés pour représenter les relations humaines, tels qu'au sein d'une entreprise par exemple. Ces recherches ont notamment abouti à une publication dans le journal Theorical Computer Science. En l'occurrence, le problème étudié est le problème de couplage maximal : supposons qu'au sein d'une entreprise, les employés puissent parfois travailler les uns avec les autres, et parfois non (auquel cas il existe ou non une arête à cet instant), et qu'une tâche à réaliser nécessite que deux employés travaillent ensemble pendant γ instants consécutifs. Trouver le γ-couplage maximal revient alors à déterminer le nombre de tâches maximal pouvant être accomplies. Ce problème étant NP-complet, nous proposons deux algorithmes distincts. Le premier est un algorithme de 2-approximation, qui retourne un couplage dont la taille est au pire 2 fois inférieure au couplage maximal du graphe. C'est un algorithme glouton dont la complexité est linéaire en la taille du graphe. Le deuxième algorithme est un algorithme de Kernelisation, qui retourne une instance de graphe plus petite que le graphe original (ou de même taille dans le pire des cas) qui contient à coup sûr un couplage maximal du graphe original. Cet algorithme a lui aussi une complexité linéaire. Nous proposons les implémentations de ces algorithmes ainsi qu'un certain nombre de simulations permettant de juger leur efficacité suivant les différents graphes temporels qui

leur sont fournis. Nous avons notamment appliqué ces algorithmes sur deux graphes temporels rencontrés souvent dans ce domaine de recherche. Le premier est le graphe "Rollernet", qui correspond à une balade en roller réalisée à Paris durant laquelle deux participants sont reliés par une arête si ils sont proches, sinon non. Le deuxième est le graphe "Enron", les échanges d'emails au sein de l'entreprise Enron qui a fait faillite en 2001 suite à des malversations. Nous avons de plus proposé un générateur de graphes temporels, qui mime le graphe Rollernet : des particules (qui peuvent être rangées en groupes) se déplacent sur un tor, et lorsque deux particules de deux groupes différents u et v entrent en contact à l'instant t, on ajoute alors au graphe une arête entre les deux groupes u et v à l'instant t. Le nombre de groupes, de particules par groupe, le rayon, la vitesse maximale, l'accélération maximale des particules sont paramétrables.

Bibliographie

[1] Ieee standards for local area networks : carrier sense multiple access with collision detection (csma/cd) access method and physical layer specifications. *IEEE Communications Magazine*, 1985.

[2] Ip datacast over dvb-h : Content delivery protocols (cdp). *ETSI*, 2006.

[3] 3gpp. multimedia broadcast/multicast service (mbms) ; protocols and codecs. 2014.

[4] D. Abraham, R. W. Irving, T. Kavitha, and K. Mehlhorn. Popular matchings. *SIAM Journal on Computing*, 37(4) :1030–1045, 2007.

[5] Swarup Acharya, Michael Franklin, and Stanley Zdonik. Dissemination-based data delivery using broadcast disks. *IEEE Personal Communications*, 2 :50–60, 1995.

[6] ANSSI. Classification method and key measures. 014.

[7] ANSSI. Profil de protection d'une diode industrielle et de ses guichets. 015.

[8] Julien Baste, Antoine Roux, and Binh-Minh Bui-Xuan. Temporal matching. *Theoretical Computer Science*, 2019. doi : `10.1016/j.tcs.2019.03.026`.

[9] R.C. Bose and D.K. Ray Chaudhuri. On a class of error correcting binary group codes. *Information and Control*, 3(1) :68–79, 1960.

[10] K. Bouchireb and P. Duhamel. Transmission schemes for scalable video streaming in a multicast environment. *Wireless Communications and Mobile Computing Conference*, pages 419–424, 2008.

[11] S. Burleigh, A. Hooke, L. Torgerson, K. Fall, V. Cerf, B. Durst, K. Scott, and H. Weiss. Delay-tolerant networking : an approach to interplanetary internet. *IEEE Communications Magazine*, 41(6) :128–136, 2003.

[12] John W Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A digital fountain approach to reliable distribution of bulk data. *IGCOMM Comput. Commun. Rev.*, 28 (4) :56–67, 1998.

[13] M. Centenaro. *Analysis of HARQ protocols for LTE cellular system.* PhD thesis, Università degli Studi di Padova, 2014.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* The MIT Press, 2009.

[15] M. Cygan, H. N. Gabow, and P. Sankowski. Algorithmic applications of Baur-Strassen's theorem : Shortest cycles, diameter, and matchings. *Journal of the ACM*, 62(4) :28 :1–28 :30, 2015.

[16] George I. Davida and Sudhakar M. Reddy. Forward-error correction with decision feedback. *Information and Control*, 21(2) :117–133, 1972.

[17] Marc Deléglise, Jean-Louis Nicolas, and Paul Zimmermann. Landau's function for one million billions. *Journal de Théorie des Nombres de Bordeaux*, 20(3) :625–671, 2008. URL `https://hal.archives-ouvertes.fr/hal-00264057`.

[18] C. Di, D. Proietti, T. Richardson, Telatar E., and R. Urbanke. Finite length analysis of low-density parity-check codes on the binary erasure channel. *IEEE Transactions on Information Theory*, 48 :1570–1579, 2002.

[19] R.G. Downey and M. R. Fellows. *Parameterized Complexity.* Springer, 1999.

[20] F. Dufossé, K. Kaya, I. Panagiotas, and B. Uçar. Approximation algorithms for maximum matchings in undirected graphs. In *SIAM Workshop on Combinatorial Scientific Computing*, 2018. `https://hal.archives-ouvertes.fr/hal-01740403`.

[21] C. Dürr, C. Konrad, and M. P. Renault. On the Power of Advice and Randomization for Online Bipartite Matching. In *24th Annual European Symposium on Algorithms*, volume 57 of *LIPIcs*, pages 37 :1–37 :16, 2016.

[22] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17 :449–467, 1965.

[23] F. V. Fomin, D. Lokshtanov, S. Saurabh, M. Pilipczuk, and M. Wrochna. Fully polynomial-time parameterized computations for graphs and matrices of low treewidth. *ACM Transactions on Algorithms*, 14(3) :34 :1–34 :45, 2018.

[24] Marc P. C. Fossorier. Quasi-cyclic low-density parity-check codes from circulant permutation matrices. *IEEE Transactions on Information Theory*, 50(8) :1788–1793, 2004.

[25] R. G. Gallager. *Low Density Parity Check Codes.* PhD thesis, MIT, Cambridge, Mass., 1960.

[26] R. G. Gallager. *Low density parity check codes.* The MIT Press, 1963.

[27] I. B. Kalugin. The number of components of a random bipartite graph. *Discrete Mathematics and Applications*, 1(3) :289–299, 1991.

[28] B. Klimt and Y. Yang. Introducing the Enron Corpus. In *CEAS*, 2004.

[29] M. Latapy, T. Viard, and C. Magnien. Stream graphs and link streams for the modeling of interactions over time. 2017. `https://arxiv.org/abs/1710.04073`.

[30] S. Lin. Performance analysis of a hybrid ARQ error control scheme for near earth satellite communications. Technical Report NASA-CR-181120, NAS 1.26 :181120, TR-1, Department of Electrical Engineering, Hawaii University, 1987.

[31] M. Luby. Lt codes. *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, page 271–280, 2002.

[32] M. Luby, A. Shokrollahi, M. Watson, and T. Stockhammer. Raptor forward error correction scheme for object delivery. RFC 5053, RFC Editor, 2007.

[33] Michael Luby, Michael Mitzenmacher, Amin Shokrollahi, Daniel Spielman, and Volker Stemann. *Practical loss-resilient codes*. 1997.

[34] Michael Luby, Michael Mitzenmacher, M. Amin Shokrollahi, and Daniel A. . Spielman. Analysis of low density codes and improved designs using irregular graphs. *In Proc. of ACM STOC*, pages 249–258, 1998.

[35] David J.C. MacKay. Information theory, inference and learning algorithms. *Cambridge University Press*, 2003.

[36] David J.C. MacKay and Radford M. Neal. Near shannon limit performance of low density parity check codes. *Electronics Letters*, 32 :1645–1646, 1996.

[37] G. B. Mertzios, A. Nichterlein, and R. Niedermeier. The power of linear-time data reduction for maximum matching. In *42nd International Symposium on Mathematical Foundations of Computer Science*, volume 83 of *LIPIcs*, pages 46 :1–46 :14, 2017.

[38] Nicholas Metropolis and S. Ulam. The monte carlo method. *Journal of the American Statistical Association*, 44(247) :335–341, 1949.

[39] Pierre Meyer, Antoine Roux, and Binh-Minh Bui-Xuan. Lightweight FEC : rectangular codes with minimum feedback information. *CoRR*, abs/1904.02076, 2019. URL `http://arxiv.org/abs/1904.02076`.

[40] G. Miller and D. Burshtein. Bounds on the maximum likelihood decoding error probability of low density parity check codes. *Information Theory*, pages 290–, 2000.

[41] S. Miyazaki. On the advice complexity of online bipartite matching and online stable marriage. *Information Processing Letters*, 114(12) :714–717, 2014.

[42] M. Molloy and B. Reed. *Graph Colouring and the Probabilistic Method*. Springer, 2002.

[43] M. Mucha and P. Sankowski. Maximum matchings via Gaussian elimination. In *45th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '04, pages 248–255, 2004.

[44] Christoph Neumann, Vincent Roca, Aurélien Francillon, and David Furodet. Impacts of packet scheduling and packet loss distribution on fec performances : observations and recommendations. *CoNEXT 2005 : Proceedings of the 2005 ACM conference on Emerging network experiment and technology*, pages 166–176, 2005.

[45] Elias P. Coding for two noisy channels. *Information Theory*, 3(6) :61–76, 1955.

[46] H. Pishro-Nik and F. Fekri. On decoding of low-density parity-check codes over the binary erasure channel. *Information Theory*, 50(3) :439–454, 2004.

[47] J. Postel. User datagram protocol. RFC 768, RFC Editor, 1980.

[48] J. Postel. Transmission control protocol. RFC 793, RFC Editor, 1981.

[49] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2) :300–304, 1960.

[50] Ronald L. Rivest. The md5 message-digest algorithm. RFC 1321, RFC Editor, 1992.

[51] V Roca, C Neumann, and D Furodet. Low density parity check (ldpc) forward error correction. RFC 5170, RFC Editor, 2008.

[52] A. I. Saltykov. The number of components of a random bipartite graph. *Discrete Mathematics and Applications*, 5(6) :515–523, 1995.

[53] Claude E. Shannon. *A Mathematical Theory of Communication.* CSLI Publications, 1948.

[54] Amin Shokrollahi. Raptor codes. *IEEE/ACM Transactions on Networking*, (14) :2551–2567, 2006.

[55] R. Singleton. Maximum distance q-nary codes. *Information Theory*, 10(2) :116–118, 1964.

[56] R. J. Stones. Computing the number of h-edge spanning forests in complete bipartite graphs. *Discrete Mathematics & Theoretical Computer Science*, 16(1) :313–326, 1994.

[57] Michael R. Tanner. A recursive approach to low complexity codes. *IEEE Transactions on Information Theory*, 27(5) :533–547, 1981.

[58] P.-U. Tournoux, J. Leguay, F. Benbadis, V. Conan, M. D. De Amorim, and J. Whitbeck. The Accordion Phenomenon : Analysis, Characterization, and Impact on DTN routing. In *28th IEEE Conference on Computer Communications*, 2009.

[59] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 42(2) :230–265, 1937.

[60] T. Viard, M. Latapy, and C. Magnien. Computing maximal cliques in link streams. *Theoretical Computer Science*, 609 :245–252, 2016.

[61] Y. Wang and S. C.-W. Wong. Two-sided Online Bipartite Matching and Vertex Cover : Beating the Greedy Algorithm. In *42nd International Colloquium on Automata, Languages, and Programming*, volume 9134 of *LNCS*, pages 1070–1081, 2015.

[62] Hakim Weatherspoon and D. Kubiatowicz John. Erasure coding vs. replication : A quantitative comparison. *Computer Science*, (2429) :328–338, 2002.

[63] S. Wøhlk and G. Laporte. Computational comparison of several greedy algorithms for the minimum cost perfect matching problem on large graphs. *Computers and Operations Research*, 87(C) :107–113, 2017.

[64] E. Zedini, A. Chelli, and M.-S. Alouini. On the performance analysis of hybrid ARQ with incremental redundancy and with code combining over free-space optical channels with pointing errors. *IEEE Photonics Journal*, 6(4) :1–18, 2014.