Algorithmes de calcul de logarithmes discrets dans les corps finis

Emmanuel Thomé

ÉCOLE POLYTECHNIQUE

THÈSE

présentée pour obtenir le grade de

DOCTEUR DE L'ÉCOLE POLYTECHNIQUE

Spécialité :

INFORMATIQUE

par

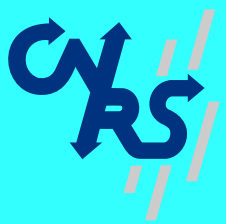
Emmanuel THOMÉ

Titre de la thèse :

ALGORITHMES DE CALCUL DE LOGARITHMES DISCRETS DANS LES CORPS FINIS

Soutenue le 12 mai 2003 devant le jury composé de :

M.	Joachim von zur GATHEN	Président
MM.	Thierry BERGER	Rapporteurs
	Don COPPERSMITH	
MM.	Xavier ROBLOT	Examinateurs
	Nicolas SENDRIER	
	Gilles VILLARD	
	François MORAIN	(Directeur)



LABORATOIRE D'INFORMATIQUE
FRE CNRS n° 2653
École polytechnique 91128 Palaiseau Cedex FRANCE

Aux jours à venir...

Version datée du 12 septembre 2003.

Remerciements

La première personne que je souhaite remercier est François Morain qui a accepté d'être mon directeur de thèse et qui m'a encadré depuis mon stage de DEA. Il s'est toujours montré d'une grande disponibilité. Il a en particulier toujours accepté de porter son regard justement critique sur les écrits plus ou moins lisibles que je lui confiais. Je lui dois les nombreux enrichissements que m'ont apportés ces années de thèse. Sa bonne humeur en a fait un apprentissage agréable.

Je tiens à remercier les membres de mon jury d'avoir accepté d'évaluer mon travail et de s'y être intéressé. Je remercie tout particulièrement Joachim von zur Gathen d'avoir assumé le rôle important de président du jury. Il a dirigé le déroulement de la soutenance avec une grande efficacité.

Je remercie grandement Thierry Berger et Don Coppersmith pour avoir accepté la lourde tâche de rapporteurs, malgré la barrière de la langue pour Don Coppersmith. Leur regard très attentif et leur analyse profonde de mon travail m'ont été très précieux. J'ai d'une certaine manière redécouvert mon travail grâce à leurs commentaires.

Je remercie Xavier Roblot qui s'est intéressé à mes travaux de thèse ainsi qu'aux autres « chantiers » sur lesquels je travaille actuellement.

Je remercie Nicolas Sendrier pour l'intérêt qu'il a porté à mon travail, et pour les discussions que nous avons eues à divers moments, en divers coins du monde.

Je remercie Gilles Villard qui a su me donner les bons conseils pour améliorer la mise en forme de mes travaux sur le calcul de générateurs linéaires matriciels.

J'ai effectué ma thèse au LIX où j'ai assidûment usé les fauteuils de la pause-café. Je remercie l'ensemble du laboratoire pour m'avoir accueilli. Je remercie d'abord Michel Weinfeld et Jean-Pierre Jouannaud, directeurs successifs du laboratoire, pour avoir mené le « navire », du petit nom donné à nos locaux depuis le déménagement de 1999. Je remercie aussi Jean-Marc Steyaert, Robert Cori, et Philippe Chassignet qui m'ont donné l'occasion d'enseigner quelques fois aux jeunes polytechniciens.

Le fonctionnement du laboratoire m'a toujours offert de bonnes conditions de travail. Ce succès est dû au travail constant d'Evelyne Rayssac qui assume les tâches administratives, ainsi qu'aux ingénieurs système Houy Kuoy et Matthieu Guionnet. Je les en remercie chaleureusement.

Je remercie les (nombreuses) personnes avec qui j'ai partagé mon bureau et aussi de nombreuses réflexions : Pierrick Gaudry, Guillaume Hanrot, Mireille Fouquet, Nicolas Gürel, Gilles Schaeffer, Philippe Baptiste, Benjamin Werner. Tous ont été à leur tour les témoins amusés de l'infinie facilité avec laquelle je peux m'auto-distraire. Ils ont eu la sagesse de m'encourager à rester concentré sur une seule chose à la fois ; je leur en suis très reconnaissant.

Je n'oublierai jamais les années de ma thèse, et la « joyeuse bande » de mes camarades thésards (et ex-thésards) du LIX. L'ambiance du labo est ainsi toujours restée à la bonne humeur, ce qui rend la vie plus facile. Pour toutes ces pauses, je remercie Pierrick Gaudry, Andreas Enge, Dominique Rossin, Mireille Fouquet, Dominique Poulalhon, Nicolas Gürel, Jérôme Waldispühl, Thomas Houtmann, Régis Dupont, Dmitri Lebedev, Simon Bliudze. Je remercie aussi tout particulièrement Gérard Guillerm pour s'être régulièrement joint à nos délires.

Le LIX a aussi été le lieu de discussions toujours intéressantes avec les visiteurs habitués comme Guillaume Hanrot ou comme nos voisins du GAGE, Éric Schost, Anne Fredet et Alexandre Sedoglavic. Je les remercie tous pour les bons moments que nous avons passé ensemble.

Je vous remercie encore tous, membres du LIX et « associés » pour votre contribution à la bonne humeur qui a toujours régné lors du traditionnel barbecue de fin d'année.

Durant ma thèse, j'ai fait souffrir beaucoup de pauvres machines innocentes. J'ai aussi causé du tracas aux administrateurs des machines concernées. Je remercie Gérard Guillerm, Bogdan Tomchuk, Joël Marchand, Teresa Gomez-Diaz pour avoir toujours été compréhensifs et coopératifs.

Je remercie John Cannon pour m'avoir invité à passer deux mois à Sydney fin 2001. J'ai pu découvrir l'Australie alors qu'il faisait froid dans l'hémisphère nord à cette époque. Je remercie particulièrement Claus Fieker, David Kohel pour m'avoir fait faire du sport, et Alan Steel pour les très nombreuses discussions que nous avons eues.

Je remercie aussi toutes les personnes que j'ai oublié de remercier de ne pas m'en tenir trop rigueur...

Je remercie enfin encore une fois Mireille Fouquet et Dominique Poulalhon qui sont devenues bien plus que des collègues de travail. Notre commune et saine horreur de la montée des marches nous a simultanément motivés pour pratiquer un covoiturage de choc. La Punto et la Clio, si les voitures ont des oreilles, ont ainsi entendu mille et mille bavardages, cancans, potins, railleries, fous rires, et parfois aussi des débats sérieux ou presque. Le trajet quotidien dans les bouchons nous servait ainsi de défouloir, et les étudiants en thèse en ont un grand besoin. Arrivant à la fin de nos thèses, nous avons aussi partagé au cours de ces bavardages divers encouragements, déceptions, colères et finalement joies. Un énorme merci à toutes les deux.

Je remercie mes parents, ma famille, mes amis qui m'ont soutenu dans cette épreuve de longue haleine qu'est la thèse. Je les remercie aussi parce qu'ils ont vite compris qu'il n'était pas de bon ton de me demander quand je « finirai mes études », et quand je « commencerai à travailler ».

Enfin, je remercie Laetitia. C'est grâce à elle que j'ai persévéré dans les moments de découragement. Elle a su au quotidien m'offrir la fraîcheur dont j'avais besoin pour ne pas m'enfermer dans ce si petit monde.

Table des matières

Table des figures

Introduction

La cryptologie est aujourd'hui au cœur du développement des nouvelles technologies de l'information et de la communication. En effet, le besoin d'assurer la confidentialité des données, au sens large, se fait sentir dans un très grand nombre d'applications : commerce électronique, communications sensibles, réseaux mobiles... Les outils proposés par la cryptologie moderne permettent de répondre à ces besoins de manière très satisfaisante, par le déploiement de cryptosystèmes à clef publique, aussi dits asymétriques. Le principe de ces cryptosystèmes est de rendre possible la publication de la clef de *chiffrement*, tout en gardant secrète la clef de *déchiffrement*. Ils permettent aussi la signature électronique de messages, ou l'établissement d'une clef secrète commune à deux acteurs via un canal de communication non sûr, sans divulgation de ce secret, par le protocole de Diffie-Hellman.

Toutes ces primitives reposent sur la notion de problème facile ou difficile à résoudre. Ces problèmes sont, pour les plus utilisés d'entre eux, issus de la théorie des nombres. Ainsi, le cryptosystème RSA [RSA78] repose sur la difficulté du problème de la factorisation d'entiers (étant donné $n = pq$, retrouver p et q), et le protocole de Diffie-Hellman repose sur celle du problème du logarithme discret dans un groupe fini : dans un groupe engendré par g, étant donné un élément a du groupe, trouver un entier x tel que $g^x = a$.

La cryptologie, lorsqu'elle s'intéresse à ces différents cryptosystèmes, traite en particulier de la difficulté des problèmes qui y sont associés. Le domaine se partage en deux branches, la cryptographie et la cryptanalyse. La première a pour objectif de bâtir des systèmes et de prouver leur sécurité : le cryptographe souhaite mener la vie dure à un espion potentiel, en ne lui laissant pas d'autre choix qu'un travail *exponentiel* pour découvrir les données secrètes. Ainsi intervient la théorie de la complexité, qui permet de donner une formalisation de la difficulté d'un problème. La cryptanalyse est au contraire l'art de montrer dans quelles situations les problèmes que l'on croit difficiles ne le sont pas véritablement, ou plus exactement ne sont pas aussi difficiles que ce que le cryptographe souhaiterait. Pour cela, le travail du cryptanalyste consiste en bonne partie à développer des outils algorithmiques nouveaux.

La cryptologie, moteur pour la théorie des nombres

Pour mettre en place des algorithmes de cryptanalyse performants, la cryptanalyse a logiquement utilisé la théorie algorithmique des nombres et l'a alimentée de nouvelles énergies. C'est ainsi que le problème de la factorisation des entiers a fait des progrès considérables, à la fois sur le plan pratique (un temps de calcul important y a été consacré) et théorique. Le crible algébrique [LL93], qui est à ce jour l'algorithme de factorisation le plus performant, a été l'aboutissement puis l'objet de nombreux travaux ayant une forte motivation cryptologique.

La théorie algébrique et algorithmique des nombres est aussi intervenue en cryptologie par le biais de la diversification des cryptosystèmes, et plus généralement des « contextes cryptographiques ». Limités à l'origine au groupe $\mathbb{Z}/n\mathbb{Z}$ pour ce qui concerne le cryptosystème RSA et le protocole de Diffie-Hellman, ces contextes sont devenus beaucoup plus variés avec la définition d'exigences de plus en plus fines de la part du « consommateur » de cryptosystèmes. D'une utilisation à une autre, celui-ci peut insister sur divers aspects : avoir une petite taille de clef, avoir un très petit temps de chiffrement ou bien de déchiffrement, avoir de petites signatures... Ce sont là autant d'exigences qui consacrent, chacune indépendamment, un

contexte cryptographique présentant les avantages adéquats, pour un usage précis, par rapport aux autres « contextes ». Pour répondre de manière précise à ces besoins variés, il est apparu nécessaire d'introduire des concepts mathématiques de plus en plus variés. Le groupe $\mathbb{Z}/n\mathbb{Z}$, à cet égard, n'est que le *début* de l'histoire.

Algorithmes sous-exponentiels

L'éventail des algorithmes déployés en cryptanalyse a fait apparaître, pour résoudre les problèmes difficiles que sont la factorisation d'entiers ou le logarithme discret, des algorithmes *sous-exponentiels*. Pour une « taille » de problème n, leur complexité s'exprime par la fonction, ou classe de fonctions :

$$L_n(\alpha, c) = O\left(\exp\left(c(1+o(1))n^{\alpha}(\log n)^{1-\alpha}\right)\right).$$

Le crible algébrique fait partie de cette classe d'algorithmes. Pour factoriser un entier N, sa complexité est de l'ordre de $L_{\log N}\left(\frac{1}{3}, c\right)$. Le caractère pratique des algorithmes sous-exponentiels, d'une manière générale, demande a être démontré par l'expérience, car la complexité sous-exponentielle est, d'une part, heuristique (dans l'immense majorité des cas), et d'autre part, peu regardante de facteurs « négligeables » au vu du comportement asymptotique global, mais qui peuvent compliquer notablement l'utilisation des algorithmes concernés. Pour le cas de la factorisation d'entiers, un effort important a été entrepris pour évaluer quelle était la taille maximale des entiers qui pouvaient être factorisés à l'aide d'une puissance de calcul donnée. C'est ainsi qu'un nombre de 512 bits, soit 155 chiffres décimaux, a été factorisé en août 1999 (une telle taille de clef est encore utilisée aujourd'hui dans un grand nombre d'applications cryptographiques).

Le calcul de logarithmes discrets est, à côté de la factorisation d'entiers, « l'autre » grand problème difficile omniprésent en cryptologie. La force de ce problème est qu'il peut être énoncé dans tout groupe possédant quelques propriétés facilement énoncées (on requiert en particulier de pouvoir calculer efficacement dans ce groupe). Il existe donc une myriade d'instances différentes du problème, et ce sont autant de cryptosystèmes, la sécurité de l'un n'étant pas nécessairement remise en question par l'existence d'une « attaque » sur le logarithme discret dans un *autre* groupe. Plus exactement, on sait démontrer qu'il n'existe pas d'attaque valide pour *tous* les groupes : on peut prouver qu'un groupe *générique* est sûr, c'est-à-dire qu'un calcul de logarithme discret est nécessairement exponentiel. Par conséquent, un algorithme de calcul de logarithmes discrets s'applique seulement à une classe de groupes bien délimitée.

Des algorithmes sous-exponentiels existent pour résoudre certaines instances du problème du logarithme discret. On peut ainsi calculer en temps sous-exponentiel des logarithmes discrets dans les groupes multiplicatifs des corps finis ou sur des courbes hyperelliptiques de genre grand. Dans la « grande famille » des algorithmes sous-exponentiels, plusieurs algorithmes partagent des traits communs. On parle là des algorithmes de crible quadratique et algébrique, ainsi que des algorithmes sous-exponentiels de calcul de logarithmes discrets. Sans rentrer dans les détails d'un algorithme spécifique ou d'un autre, on peut distinguer deux phases « principales », suivies souvent d'une troisième.

– La première phase consiste à fabriquer un système de relations linéaires. Cette phase est en général la plus coûteuse en terme de temps de calcul, mais possède l'avantage de pouvoir être *distribuée* sur un grand nombre de machines. Cet aspect a été la source des calculs à très grande échelle comme les récents records de factorisation d'entiers,

qui ont été l'aboutissement d'efforts de calculs communs menés par plusieurs groupes, partageant leurs résultats via le réseau Internet.
- La seconde phase est la *résolution* du système linéaire associé. Ce système linéaire a la propriété d'être très creux : son nombre de coefficients non-nuls par ligne est très faible. Des algorithmes spécialisés pour les systèmes creux peuvent alors être utilisés, mais en comparaison avec la possibilité de distribution quasi infinie de la première phase, ces algorithmes d'algèbre linéaire supportent mal d'être distribués.
- La dernière phase, s'il y en a une, est un peu plus facile que les deux autres phases. Suivant l'algorithme auquel on s'intéresse en particulier, elle peut prendre plusieurs formes, comme par exemple le calcul d'un facteur lorsque l'on parle d'algorithmes de factorisation, ou le calcul d'un ou de plusieurs logarithmes individuels pour les algorithmes de calcul de logarithme discret.

Une bonne partie de ce mémoire est consacrée à l'étude d'un algorithme sous-exponentiel particulier, proposé par Coppersmith en 1984 pour résoudre le problème du logarithme discret dans les groupes multiplicatifs des corps finis de caractéristique 2. De très nombreux points de l'algorithme ont été étudiés, et plusieurs améliorations ont été obtenues. En particulier, nous avons travaillé à rendre possible la distribution partielle du calcul d'algèbre linéaire.

Contenu et organisation de ce mémoire

Ce mémoire est composé de deux parties, et d'une annexe qui passe en revue les quelques prérequis de la théorie des corps finis, utiles pour la compréhension de l'ensemble du manuscrit.

La première parie décrit notre travail concernant le calcul de logarithmes discrets en général, et l'algorithme de Coppersmith en particulier. Ce dernier a été utilisé pour calculer des logarithmes discrets dans les corps finis de caractéristique 2. En examinant divers points de l'algorithme, nous avons pu le déployer à grande échelle, et mener à bien le calcul de logarithmes discrets dans $\mathbb{F}_{2^{607}}$, ce qui constitue le record mondial actuel depuis février 2002. Le chapitre 1 détaille la portée cryptographique du calcul de logarithmes discrets. Le chapitre 2 est consacré à l'exposition des différentes méthodes de cryptanalyse permettant de calculer des logarithmes discrets, en particulier la description de l'algorithme de Coppersmith. Mettre en œuvre à grande échelle des calculs de logarithmes discrets en utilisant cet algorithme a nécessité le développement de nombreuses optimisation, ainsi que l'examen de diverses caractéristiques fines du calcul. Ces études sont détaillées dans le chapitre 3. Le chapitre 4 expose notre record de calcul de logarithmes discrets dans $\mathbb{F}_{2^{607}}$, constituant l'aboutissement de cette entreprise. Nous donnons aussi dans ce dernier chapitre de la première partie un aperçu des aspects sociologiques que comporte un calcul de cette ampleur.

La deuxième partie de ce mémoire est consacrée à la résolution de systèmes linéaires creux définis sur un corps fini. Ce problème occupe une place centrale dans les calculs de logarithmes discrets que nous avons effectué sur les corps $\mathbb{F}_{2^n}$, mais plus généralement, il est de première importance pour la plupart des algorithmes sous-exponentiels. Le chapitre 5 donne un premier aperçu des méthodes disponibles. Le chapitre 6 détaille les méthodes utilisant des générateurs linéaires, comme l'algorithme de Wiedemann, et l'algorithme de Wiedemann par blocs. C'est ce dernier algorithme, inventé par Coppersmith, que nous avons utilisé. Nous avons ainsi démontré qu'il rendait possible une distribution partielle des tâches. Ceci est démontré par les développements pratiques sur l'implantation de l'algorithme de Wiedemann par blocs, détaillées au chapitre 7. Une amélioration importante que nous avons apportée à l'algorithme a été la possibilité de calculer des générateurs linéaires de suites matricielles en temps sous-

quadratique. Cette amélioration est décrite dans le chapitre 8. C'est en utilisant ces techniques que nous sommes parvenus à résoudre le système linéaire qui est intervenu au cours du calcul de logarithmes discrets dans $\mathbb{F}_{2^{607}}$. La taille de ce système est impressionnante, puisqu'elle atteint $1\,077\,513 \times 766\,150$, et que le corps de base, $\mathbb{Z}/(2^{607}-1)\mathbb{Z}$, n'est pas vraiment petit. Les détails de ce calcul, qui constitue aussi un record, sont donnés dans le chapitre 9. Là encore, on donne avec la description de ce record un témoignage des difficultés que représente la réalisation d'un tel calcul.

Première partie

Logarithmes discrets dans $\mathbb{F}_{2^n}$

Chapitre 1

Logarithme discret et cryptographie

1.1 Différentes instances du problème

1.1.1 Définitions

Si l'on veut rester très général, le problème du logarithme discret peut se formuler dans n'importe quel groupe. Soit G un groupe cyclique de cardinal n que l'on note multiplicativement. On appelle g un générateur de G. Le logarithme discret d'un élément se définit comme suit.

Définition 1.1 (Logarithme discret). *Soit $a \in \mathsf{G}$. On appelle logarithme discret en base g de a l'unique élément x de $\mathbb{Z}/n\mathbb{Z}$ tel que :*

$$g^x = a.$$

Souvent, on considérera le logarithme discret de a comme étant l'unique représentant entier de x dans $[\![0 \dots n-1]\!]$, mais il est capital de garder à l'esprit le fait que ce logarithme n'est réellement défini que modulo $n = \#\mathsf{G}$.

Le problème du *calcul* du logarithme discret (que l'on note DL en abrégé) est un problème généralement difficile (plus ou moins en fonction du groupe G). Dans de nombreuses situations, cela permet de fabriquer des cryptosystèmes, car cette asymétrie entre le problème du calcul du logarithme (difficile), et celui du calcul des puissances (facile) est propice pour la cryptographie. Diffie et Hellman [DH76] ont été les premiers à bâtir un cryptosystème à partir de cette situation.

Le protocole d'échange de clefs proposé par Diffie et Hellman est le suivant. Supposons que deux intervenants, Alice et Bob, souhaitent échanger un secret. Il n'ont comme moyen commun de communication qu'un canal de communication non sûr, où l'information circulant peut être interceptée. Ils peuvent y parvenir en choisissant communément et de façon publique un groupe G et un générateur g de G. Chacun, secrètement, choisit un entier aléatoire, k_A pour Alice et k_B pour Bob. Ils échangent au travers du canal de communication non sûr les grandeurs g^{k_A} et g^{k_B}. Ils peuvent alors chacun calculer le secret commun $g^{k_A k_B}$, sous la forme $(g^{k_B})^{k_A}$ pour Alice, et sous la forme $(g^{k_A})^{k_B}$ pour Bob. Ce protocole est schématisé par la figure 1.1.

Le problème que doit résoudre un espion potentiel pour découvrir le secret commun partagé par Alice et Bob après l'échange consiste, connaissant les paramètres publics que sont G et g, ainsi que les grandeurs g^{k_A} et g^{k_B} qu'il a pu intercepter au cours de la communication, à retrouver $g^{k_A k_B}$. Ce problème, que l'on note DH, peut être résolu si l'on sait calculer des logarithmes discrets dans G. On a donc une implication $\mathsf{DL} \Rightarrow \mathsf{DH}$. L'implication réciproque est « presque » vraie, sous certaines hypothèses [MW96, Mau94]. Ceci nous permet de nous concentrer en priorité sur le problème DL : si DL n'est pas faisable, alors l'espion est désarmé.

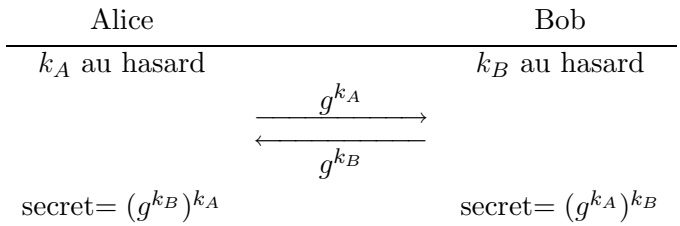


Figure 1.1 – Le protocole d'échange de clefs de Diffie-Hellman

Nous allons voir plusieurs cryptosystèmes faisant intervenir le logarithme discret. Tous reposent, de la même façon, sur l'*impossibilité*, pour un espion, de calculer des logarithmes discrets. La difficulté de cette tâche dépend du groupe G dans lequel les calculs sont menés. Nous allons donc donner un aperçu des différents groupes pouvnt être étudiés.

1.1.2 Hypothèses requises

Quelles propriétés le groupe G doit-il satisfaire ? Il faut que le logarithme discret soit difficile, et notamment beaucoup plus difficile que le calcul des puissances de g. Ces considérations nécessitent une formalisation qui fait appel à des notions de complexité. On décrit quelques-unes des qualités du groupe idéal pour l'implantation de cryptosystèmes. Comme $n = \#\mathsf{G}$, il est légitime de supposer que les éléments de G peuvent être représentés à l'aide de $O(\log n)$ bits. Si G ne vérifie pas cette hypothèse, l'utilisabilité en cryptographie est compromise. Ensuite, la « facilité » du calcul des puissances s'exprime par une complexité au plus *polynomiale* (en la taille des entrées, donc $\log n$) du calcul correspondant, à laquelle on veut opposer une complexité *exponentielle* (toujours en $\log n$) pour le calcul de logarithmes discrets.

Si G est un groupe « générique », ne satisfaisant pas d'autres hypothèses que celles que l'on vient de mentionner (à savoir, surtout, le caractère polynomial du calcul de la loi de groupe), on sait démontrer que le calcul de logarithmes discret *est* exponentiel [Sho97] si aucune autre information concernant le groupe n'est utilisée. Mais la démonstration d'une telle propriété ne peut rester valide si le groupe G est *instancié* : par définition, un groupe particulier *ne peut pas* être un groupe générique. Sans rentrer dans le détail des cryptanalyses que nous étudierons au chapitre 2, il convient bien sûr de remarquer que tous les groupes proposés ont chacun leurs spécificités et sont individuellement sujets à des attaques, plus ou moins efficaces. C'est là que l'on trouve bien entendu l'une des motivations essentielles pour préférer un groupe à un autre.

Pour ce qui est du calcul du logarithme discret, on juge la « qualité » d'un groupe à la complexité du calcul de logarithmes dans ce groupe (partant du principe que le calcul de la loi de groupe est polynomial, conformément à ce que l'on a énoncé). On peut ainsi partager les groupes proposés en trois classes.

DL dans G est exponentiel

C'est la catégorie de groupes dans laquelle on rêve de trouver un exemple. On ne peut espérer mieux, car nous verrons au chapitre 2 qu'il existe des algorithmes exponentiels pour calculer des logarithmes discrets dans des groupes n'ayant que les propriétés minimales citées plus haut. Hélas, trouver des exemples de tels groupes est encore un rêve, car on ne connaît pas de groupe où il est possible de *prouver* que DL est exponentiel. On connaît toutefois des exemples de groupes pour lesquels aucun algorithme connu ne

permet de calculer des logarithmes plus rapidement qu'en temps exponentiel.

DL dans G est polynomial

Alors, le groupe G ne mérite pas d'être considéré pour la cryptographie. En effet, un cryptosystème s'appuyant sur un tel groupe rendrait le travail de l'espion à peu près aussi facile[1] que celui des acteurs « honnêtes ».

DL est entre les deux : sous-exponentiel

Nous allons voir des exemples de groupes pour lesquels DL est de complexité sous-exponentielle. Cette complexité s'exprime, pour les cas qui nous intéressent, à l'aide de la fonction $L(\alpha, c)$, déjà définie dans l'introduction par :

$$L_{\log n}(\alpha, c) = O\left(\exp\left(c(1+o(1))(\log n)^{\alpha}(\log\log n)^{1-\alpha}\right)\right).$$

La fonction $L(\alpha, c)$ interpole ainsi entre les complexités polynomiales et exponentielles, puisque $L_{\log n}(0, c) = (\log n)^c$ et $L_{\log n}(1, c) = n^c$. L'existence d'un algorithme sous-exponentiel de calcul de logarithme discret dans un groupe n'est pas nécessairement une raison de disqualifier le groupe en question pour tout usage cryptographique. Le caractère pratique de l'algorithme en question mérite d'être analysé. Dans bien des cas toutefois, un groupe dans cette catégorie souffre d'un sérieux désavantage vis-à-vis des groupes pour lesquels seul un algorithme de calcul exponentiel est connu.

1.1.3 Groupes proposés

Il convient en premier lieu de remarquer qu'il existe bien sûr de très mauvais groupes. Par exemple, si la loi de G est en fait l'addition pour une structure d'anneau existant sur les éléments de G, alors la situation est compromise. Prenons ainsi pour G le groupe $(\mathbb{Z}/n\mathbb{Z}, +)$, que l'on note donc additivement. Ce qui tient lieu de « puissance » de g se note xg et le « logarithme » de xg est x. Retrouver x est alors de complexité seulement polynomiale, puisque l'algorithme d'Euclide répond au problème.

Groupes multiplicatifs de corps finis

Le premier groupe non trivial proposé répondant au moins en partie aux spécifications que l'on vient d'énoncer a été (dès l'article originel de Diffie et Hellman) le groupe *multiplicatif* de $\mathbb{Z}/n\mathbb{Z}$, lorsqu'il est cyclique. Nous verrons au chapitre 2 que l'algorithme de Pohlig-Hellman permet de calculer des logarithmes dans $\mathbb{Z}/n\mathbb{Z}$ à partir du calcul de logarithmes dans les groupes multiplicatifs des corps premiers $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$, pour les différents facteurs premiers p de n. Ceci nous amène à nous concentrer sur le cas où n lui-même est premier et plus généralement à considérer les groupes multiplicatifs des corps finis. Depuis 25 ans, l'état de l'art en ce qui concerne le calcul de logarithmes discrets dans les corps finis a permis de développer divers algorithmes, donnant au problème DL dans les corps finis les différentes complexités indiquées sur la figure 1.2.

Il apparaît que dans tous les cas, un algorithme sous-exponentiel pour résoudre DL existe. Le caractère pratique de ces algorithmes mérite d'être éprouvé. Une trame générale de la première partie de ce mémoire est justement le calcul de logarithmes discrets dans le groupe multiplicatif de $\mathbb{F}_{2^n}$.

[1] Si DL dans G est polynomial de complexité $O((\log n)^{2003})$, on ne peut bien sûr pas dire que « polynomial » signifie « facile ». On devrait donc plutôt faire rentrer dans cette mauvaise catégorie les groupes pour lesquels DL est polynomial d'exposant modéré.

Quoi ?	Qui ?	Quand ?	Coût ?
$\mathbb{F}_{2^n}$	Coppersmith [Cop84]	1984	$L_n\left(\frac{1}{3}, c\right)$
$\mathbb{F}_p$	Gordon [Gor93]	1993	$L_{\log p}\left(\frac{1}{3}, c\right)$
$\mathbb{F}_{p^n}$	Adleman-DeMarrais [AD93]	1993	$L_{n \log p}\left(\frac{1}{2}, c\right)$
$\mathbb{F}_{p^n}$, p petit	Adleman [Adl94, AH99]	1994	$L_n\left(\frac{1}{3}, c\right)$
	Semaev [Sem98a]	1993	

Figure 1.2 – Complexité de DL dans les corps finis

On peut remarquer qu'il est possible de travailler dans des sous-groupes de groupes multiplicatifs, si l'on dispose pour ces sous-groupes d'une représentation efficace. De cette façon, on rend l'attaque structurelle sur le corps fini moins efficace (par rapport à la taille des objets manipulés). Ce principe apparaît dans la présentation du protocole de signature de Schnorr [Sch91], ainsi que dans le « cryptosystème » XTR[2] [LV00].

Courbes elliptiques

Un autre exemple de groupe proposé est le groupe des points des courbes elliptiques définies sur les corps finis, suggéré à l'origine par Koblitz [Kob87] et Miller [Mil87]. Une courbe elliptique définie sur un corps fini K peut être vue comme l'ensemble des solutions dans K^2 de l'équation :

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6.$$

Si les coefficients a_i vérifient certaines conditions, l'ensemble des points de cette courbe, auquel on adjoint un « point à l'infini » (correspondant à la solution projective (0:1:0)) forme un groupe commutatif. Il est donc possible d'y développer les protocoles cryptographiques reposant sur le logarithme discret.

Il n'y a pas d'algorithme sous-exponentiel connu pour résoudre le problème du logarithme discret sur les courbes elliptiques en général. Elles constituent donc d'excellents candidats pour l'implantation de protocoles cryptographiques. Il convient toutefois d'éviter certaines classes de courbes, sujettes à des attaques.

- Les courbes de trace 1 (telles que le cardinal de la courbe est le cardinal de K) sont à éviter à tout prix, car le logarithme discret s'y calcule en temps polynomial [SA98, Sma99, Sem98b, Rüc99].
- Les courbes *supersingulières* ainsi que les courbes telles que le cardinal de K est d'ordre petit modulo le cardinal de la courbe sont aussi sujettes à des attaques [MOV93, FR94]. Ces attaquent montrent que DL sur la courbe peut être résolu à partir de DL sur une petite extension de K.
- Lorsque K est une extension de degré composé de son sous-corps premier, l'attaque par descente de Weil de [GHS02] peut s'appliquer (les conditions exactes sont plus subtiles). Cette attaque donne un algorithme de calcul sous-exponentiel du logarithme discret.

Éviter ces classes de courbes est aisé. Il n'en reste pas moins nécessaire d'évaluer le caractère réalisable ou non des attaques les concernant. Dans le cas des courbes supersingulières, sujettes à la seconde des attaques que l'on vient de mentionner, sous la forme par exemple de

[2] XTR n'est en réalité qu'une façon de représenter les éléments.

la réduction MOV [MOV93], on note que cela fait partie du champ d'application des calculs de logarithmes discrets sur les corps finis. En effet, la réduction MOV (ainsi que ses généralisations mentionnées) réduit le problème DL sur la courbe au problème DL dans une extension du corps de base K. Une étude sur la difficulté du calcul de logarithmes discrets dans les corps finis permet donc aussi de juger de la portée pratique de ces attaques.

Au-delà des courbes elliptiques, il est possible d'utiliser des courbes algébriques plus générales. Ainsi, les jacobiennes des courbes hyperelliptiques [Kob89, Gau00b] sont une généralisation naturelle. Elles présentent des avantages et des inconvénients spécifiques.

1.2 Cryptosystèmes utilisant le logarithme discret

Outre le protocole d'échange de clefs de Diffie-Hellman que nous avons déjà vus, le logarithme discret se prête à diverses utilisations cryptographiques [MvOV97]. Nous décrivons ici quelques-uns de ces cryptosystèmes de telle sorte que l'on puisse les mettre en œuvre dans n'importe quel groupe approprié.

1.2.1 Le système de chiffrement d'ElGamal

Malgré leurs tentatives, Diffie et Hellman ne sont pas parvenus, en 1976, à bâtir un système de *chiffrement* (à clef publique) autour du problème du logarithme discret. Après quelques essais infructueux de mise en place de tels cryptosystèmes, recensés dans [Odl85], c'est seulement en 1985 qu'un système de chiffrement praticable utilisant le logarithme discret a été proposé par ElGamal [ElG85]. Ce système repose sur le principe suivant.

Supposons que Bob souhaite envoyer un message chiffré à Alice. Pour cela, Alice et Bob s'entendent au préalable sur un groupe G dans lequel travailler, et sur un générateur g de G. Alice doit mettre en place une paire de clefs, l'une secrète (privée) et l'autre publique. Comme clef secrète, elle choisit un entier x aléatoire. Sa clef publique est alors $y = g^x$.

Pour chiffrer le message m qu'il souhaite envoyer (on suppose que m est un élément de G, pour simplifier), Bob choisit d'abord un entier k aléatoire premier avec $n = \#\mathsf{G}$. Il calcule les deux éléments suivants de G :

$$a = g^k, \quad \text{et} \quad b = my^k.$$

Le texte chiffré est alors la paire (a, b).

Pour déchiffrer le message m, Alice calcule $a^{-x}b$ qui vaut exactement m.

Ce système de chiffrement repose sur la difficulté du problème suivant : connaissant trois éléments g, g^x, et a, calculer a^x. Ce problème est équivalent au problème DH. Nous avons déjà mentionné que la difficulté du problème DH était peu ou prou équivalente à celle du problème DL [MW96, Mau94].

On peut remarquer qu'avec ce système, le message chiffré, sous la forme du couple (a, b), est de taille deux fois supérieure à l'information transmise (le message m). C'est un inconvénient du système.

1.2.2 Le système de signature d'ElGamal

Un système de signature a aussi été proposé par ElGamal [ElG85]. Nous sommes dans la situation où Alice souhaite signer un document, de telle sorte que Bob puisse vérifier la signature. Ils se mettent d'accord sur G et g (ils sont habitués), ainsi que sur une fonction

quelconque ϕ de G dans $\mathbb{Z}/n\mathbb{Z}$, où $n = \#\mathsf{G}$. Soit m le message en question que l'on prend comme étant un élément de $[\![0 \dots n-1]\!]$. Alice dispose d'une clef secrète x et publie son information publique $y = g^x$. La signature de m qu'elle produit est un couple (a, b), où $a \in \mathsf{G}$ et $b \in \mathbb{Z}$, tels que :

$$g^m = y^{\phi(a)} a^b.$$

Pour fabriquer a et b, Alice commence par choisir un entier k aléatoire, premier avec n, et calcule $a = g^k$. Elle doit ensuite trouver b tel que :

$$\begin{aligned} kb + x\phi(a) &\equiv m \mod n, \\ b &\equiv k^{-1}(m - x\phi(a)) \mod n. \end{aligned}$$

La vérification de la signature est simple. Bob doit simplement s'assurer que $g^m = y^{\phi(a)} a^b$. Comme c'est le cas pour le système de chiffrement, on peut déplorer que ce système de signature produise des signatures très longues : le couple (a, b) a une taille deux fois supérieure à celle du message transmis.

1.2.3 Le système de signature de Schnorr

Le protocole de signature de Schnorr [Sch91] est à la fois un système d'authentification et de signature. Nous décrivons ici le système de signature. On a toujours un groupe G, un générateur g, et une fonction de hachage H. L'information publique d'Alice est toujours $y = g^x$, où x est un entier aléatoire secret. La signature d'un message m est un couple (e, s) d'entiers vérifiant la propriété suivante qui devra être testée par Bob :

$$H(M \,\|\, g^s y^e) = e.$$

On a noté ici $\|$ la concaténation des informations. Pour fabriquer la signature (s, e), Alice fabrique $a = g^k$ à partir d'un entier aléatoire k. Elle calcule ensuite $e = H(M \,\|\, a)$ et $s = k - xe$. On a alors $g^s y^e = a$, donc la relation voulue est vérifiée.

Les signatures produites par ce cryptosystème peuvent être courtes, car elles reposent autant sur la difficulté de calculer des logarithmes discrets que sur la difficulté de trouver des collisions dans la fonction de hachage H.

1.2.4 Le système de signature DSA

Le système DSA a été proposé par l'organisme américain NIST (et conçu par la NSA) en 1991, approuvé en 1994, et mis à jour en 2000. Pour présenter l'algorithme de manière générique, de façon à englober ses variantes (ECDSA), supposons que nous travaillons dans un groupe G, muni d'un générateur g, et d'une fonction quelconque ϕ de G dans $\mathbb{Z}/n\mathbb{Z}$ (on englobe ainsi à la fois la présentation originelle de l'algorithme et la variante ECDSA). On se donne en outre une fonction de hachage H.

L'information publique d'Alice est toujours $y = g^x$, où x est un entier aléatoire secret. La signature d'un message m est un couple (r, s) d'entiers définis modulo n, avec s premier avec n, vérifiant la condition suivante, que Bob devra tester.

$$\phi(g^{H(m)s^{-1}} y^{rs^{-1}}) = r.$$

Pour obtenir r et s, Alice choisit au hasard un entier k premier avec n, et $a = g^k$. Elle calcule $r = \phi(a)$ et s par la formule :

$$s = k^{-1}(H(m) + x\phi(a)) \mod n.$$

Les paramètres pour l'algorithme DSA doivent être choisis avec soin. Dans la présentation originelle de l'algorithme, le groupe G est choisi comme étant un sous-groupe de cardinal q du groupe $\mathbb{F}_p^*$. On a donc $n = q$ qui doit diviser $p - 1$. Les nombres p et q sont choisis de tailles respectives 160 et 1 024 bits.

1.3 Cryptographie fondée sur l'identité

Un autre cadre d'application des logarithmes discrets est la cryptographie fondée sur l'identité. Ce concept a été proposé à l'origine par Shamir [Sha85]. Il consiste à utiliser comme clef publique d'un intervenant son identité (par exemple son adresse de courrier électronique). Dans ce contexte, un tiers de confiance (PKG, pour *private key generator*) est responsable de la certification de l'identité d'un intervenant. Si cet intervenant est Bob, c'est le tiers de confiance qui fournit à Bob sa clef secrète, à l'aide de laquelle il peut décrypter les messages qui lui sont envoyés par Alice. Pour envoyer de tels messages, Alice a simplement besoin d'utiliser l'*identité* de Bob.

Les applications d'un tel schéma sont multiples. De nombreuses propositions d'implantation concrètes ont été proposées depuis sa création [MY92, MY96], mais le seul schéma satisfaisant à ce jour est celui proposé par Boneh et Franklin [BF01]. Nous décrivons ce schéma, ainsi qu'un protocole de distribution de clef non interactive, aussi fondé sur l'identité. Tous deux reposent sur l'utilisation du couplage de Weil sur une courbe elliptique [Sil86]. Nous discutons ensuite dans quelle mesure ces schémas cryptographiques offrent une nouvelle motivation pour s'intéresser au calcul de logarithmes discrets sur les corps finis.

Nous faisons ici une description très informelle des protocoles, sans rentrer dans les détails techniques des couplages utilisés. En effet, le couplage de Weil ne correspond pas exactement à nos besoins. Comme tous les calculs sont effectués avec des points appartenant au même sous-groupe cyclique, nous devons avoir $e(P, P) \neq 1$, ce qui n'est possible qu'avec une modification du couplage. Nous n'entrons pas dans ces détails, traités dans [BF01, DE03].

1.3.1 Le système de chiffrement de Boneh et Franklin

Soit E une courbe elliptique définie sur un corps premier $\mathbb{F}_q$. Le couplage de Weil est une forme bilinéaire non dégénérée de $E \times E$ dans le groupe multiplicatif d'une extension finie $\mathbb{F}_{q^k}$ de $\mathbb{F}_q$. On le note $e(P, Q)$, où P et Q sont deux points de E. Ce couplage vérifie la relation $e(aP, bQ) = e(P, Q)^{ab}$ pour deux entiers a et b quelconques. Requérir que le couplage est non dégénéré signifie que $e(P, Q)$ n'est pas identiquement égal à 1. Bien entendu, on souhaite que le couplage de Weil soit aisément calculable.

Pour décrire le protocole proposé par Boneh et Franklin, on se donne deux fonctions de hachage, l'une notée $H_1 : \mathbb{Z} \to E$ (on considère les identités comme étant des entiers), et l'autre $H_2 : \mathbb{F}_{q^k}^* \to \mathbb{Z}/2^w\mathbb{Z}$, où w est le nombre de bits des messages transmis. Les identités d'Alice et Bob sont notées respectivement ID_A et ID_B.

Outre la courbe E et les différentes grandeurs associées, ainsi que les fonctions H_1 et H_2, les paramètres du système comprennent deux points P et Q de E. Ces deux points sont

publiquement connus. Le point Q est fabriqué sous la forme $Q = sP$, où s est un entier gardé secret : il n'est connu que de l'autorité PKG qui fournit les clefs privées.

Trois phases distinctes interviennent dans le schéma. Dans la première, Bob obtient sa clef privée de l'autorité PKG. Dans la seconde, on décrit l'algorithme de chiffrement d'un message envoyé par Alice à Bob. Enfin on décrit comment Bob déchiffre le message. Nous allons voir que les deux premières phases n'entretiennent aucun lien chronologique obligatoire.

- Pour obtenir sa clef privée, Bob en fait la requête auprès de l'autorité PKG. Cette autorité vérifie l'identité ID_B de Bob et fournit à Bob sa clef privée qui est le point $S_B = sH_1(\mathsf{ID}_B)$.
- Pour envoyer un message à Bob, Alice a juste besoin de connaître les paramètres généraux du système et l'identité ID_B de Bob. Notons m le message, formé de w bits. Alice calcule $y = e(H_1(\mathsf{ID}_B), Q)$ ainsi qu'un entier aléatoire $r \in \mathbb{Z}$ et transmet le message chiffré formé par le couple (U, V), où :

$$U = rP, \qquad V = m \oplus H_2(y^r).$$

- Pour retrouver m à partir de U et V, Bob peut utiliser sa clef secrète S_B qui lui a été fournie par l'autorité PKG. En effet, m s'obtient par :

$$V \oplus H_2(e(S_B, U)) = m.$$

  On vérifie aisément que $e(S_B, U)$ est effectivement égal à y^r.

Boneh et Franklin démontrent que la sécurité de ce cryptosystème repose sur le problème suivant : étant donné quatre points (P, sP, rP, tP), calculer $e(P, P)^{rst}$ (on rappelle que le couplage utilisé n'est pas exactement le couplage de Weil, mais une version modifiée garantissant que cette quantité est différente de 1). Ce problème est une généralisation bilinéaire du problème DH, notée BDH. Deux possibilités de résoudre ce problème apparaissent : on peut calculer des logarithmes discrets sur la courbe E, ou bien dans $\mathbb{F}^*_{q^k}$, pour retrouver individuellement r, s, et t, puis enfin leur produit.

1.3.2 Distribution de clef non-interactive

La cryptographie fondée sur l'identité ouvre la voie à une version non interactive du protocole d'échange de clefs de Diffie-Hellman, en utilisant les couplages. Nous décrivons ici le protocole proposé par Dupont et Enge [DE03]. Pour obtenir un secret commun, Alice et Bob, sans discussion préalable, calculent les quantités :

$$\begin{aligned} \text{Alice :} \quad & S = e(S_A, H_1(\mathsf{ID}_B)), \\ \text{Bob :} \quad & S = e(H_1(\mathsf{ID}_A), S_B). \end{aligned}$$

On constate aisément que ces deux quantités sont égales. On peut prouver [DE03] que comme le protocole de chiffrement fondé sur l'identité que l'on vient de présenter, ce protocole repose sur la difficulté du problème BDH.

1.3.3 Importance du logarithme discret

Pour résoudre le problème BDH, on ne connaît pas d'autre méthode que la résolution du problème DL, soit sur la courbe E, soit dans le groupe multiplicatif le corps $\mathbb{F}^*_{q^k}$. Le paramètre k reliant les deux entités dépend de la courbe E. Pour les courbes *supersingulières*, ce paramètre

est inférieur à 6. C'est ainsi que la réduction MOV réduit le calcul de logarithmes discrets dans E au calcul de logarithmes discrets dans $\mathbb{F}_{q^k}$. Pour calculer a à partir de P et aP, on calcule le logarithme de $e(aP, Q)$ en base $e(P, Q)$, pour un point Q arbitraire.

Les corps finis offrent une moins grande sécurité que les courbes elliptiques, à taille de clef semblable. Ainsi, le calcul de logarithmes discrets dans le groupe de points d'une courbe elliptique définie sur un corps premier de 160 bits est de difficulté à peu près équivalente à un calcul de logarithmes discrets dans un corps premier de 1000 bits. Si on se contraint, pour les cryptosystèmes utilisant des couplages, au cas des courbes supersingulières, on constate que l'angle d'attaque le plus facile est le corps fini $\mathbb{F}_{q^k}$, puisque k est au plus 6.

Pour cette raison, il est primordial de savoir exactement quelle taille de corps fini offre un niveau de sécurité donné. En effet, on ne souhaite pas grossir les paramètres du système sans fondement. Le déploiement de cryptosystèmes fondés sur l'identité requiert une évaluation précise des niveaux de difficulté des calculs de logarithmes discrets dans les corps finis. Bien qu'il soit possible de s'affranchir de la limite $k = 6$ en proposant des courbes non supersingulières avec un paramètre k choisi [DEM03], cette évaluation de difficulté reste de première importance.

Chapitre 2

Logarithme discret et cryptanalyse

2.1 L'algorithme de Pohlig-Hellman

Dans un groupe cyclique fini G de cardinal n, le problème du logarithme discret revient à expliciter l'isomorphisme entre G et le groupe $\mathbb{Z}/n\mathbb{Z}$. Si l'entier n se factorise sous la forme $n = \prod_{i=1}^{m} p_i^{k_i}$, on sait qu'on a l'isomorphisme de groupes additifs suivants :

$$\mathbb{Z}/n\mathbb{Z} \cong \mathbb{Z}/p_1^{k_1}\mathbb{Z} \oplus \cdots \oplus \mathbb{Z}/p_l^{k_l}\mathbb{Z}.$$

L'algorithme de Pohlig et Hellman [PH78] permet d'exploiter cette décomposition pour le calcul de logarithmes discrets. On commence par « remonter » sur G la décomposition de $\mathbb{Z}/n\mathbb{Z}$ que l'on vient de citer. Cela fait l'objet de l'énoncé suivant.

Proposition 2.1. *Soit G un groupe cyclique de cardinal $n = \prod_{i=1}^{m} p_i^{k_i}$, engendré par un élément g (noté $\mathsf{G} = \langle g \rangle$), et noté multiplicativement. Soit $j \in [\![1 \dots m]\!]$. Soit*

$$n_j = \prod_{i \neq j} p_i^{k_i} = \frac{n}{p_j^{k_j}}, \qquad n'_j = p_j^{k_j - 1} n_j = \frac{n}{p_j}.$$

Soit g_j l'élément défini par $g_j = g^{n_j}$. Le sous-groupe $\mathsf{G}_j = \langle g_j \rangle$ de G est isomorphe à $\mathbb{Z}/p_j^{k_j}\mathbb{Z}$. En outre, en posant $g'_j = g_j^{p_j^{k_j-1}} = g^{n'_j}$, le sous-groupe $\mathsf{G}'_j = \langle g'_j \rangle$ de G_j est isomorphe à $\mathbb{Z}/p_j\mathbb{Z}$.

Démonstration. L'ordre de l'élément g étant n par construction, il est clair que g_j et g'_j sont respectivement d'ordres $p_j^{k_j}$ et p_j. On peut mentionner la commutativité du diagramme suivant, où les applications sont définies par prolongement immédiat à partir des définitions de g_j et g'_j.

$$\begin{array}{ccccc} \mathsf{G} & \longrightarrow & \mathsf{G}_j & \longrightarrow & \mathsf{G}'_j \\ \downarrow{\scriptstyle \log} & & \downarrow{\scriptstyle \log} & & \downarrow{\scriptstyle \log} \\ \mathbb{Z}/n\mathbb{Z} & \longrightarrow & \mathbb{Z}/p_j^{k_j}\mathbb{Z} & \longrightarrow & \mathbb{Z}/p_j\mathbb{Z} \end{array}$$

■

Cette décomposition permet en fait de *réduire* le problème du logarithme discret dans G au problème du logarithme discret dans les groupes G'_j (une première étape facile étant la réduction aux sous-groupes G_j). On énonce donc le résultat suivant, que l'on démontre de manière constructive.

Proposition 2.2. *Notons* $\mathsf{DL}(\mathsf{G})$ *la complexité du calcul du logarithme discret dans le groupe* G*. On a :*

$$\mathsf{DL}(\mathsf{G}) \in O\left(\sum_j k_j \mathsf{DL}(\mathsf{G}'_j) + \epsilon + (\log n)^3\right),$$

où ϵ *désigne* $O(\log n)$ *opérations dans* G *(multiplications, inversions).*

Démonstration. On démontre ce résultat en deux étapes. Commençons par la relation :

$$\mathsf{DL}(\mathsf{G}) \in O\left(\sum_j \mathsf{DL}(\mathsf{G}_j) + (\log n)^3\right).$$

Soit $x \in \mathsf{G}$ un élément dont on souhaite calculer le logarithme en base g. Supposons que l'on a calculé un m-uplet $(\ell_1, \ldots, \ell_m)$ tel que $\ell_j = \log_{g_j}(x^{n_j})$. On a alors pour chaque j :

$$\begin{aligned} x^{n_j} &= g_j^{\ell_j}, \\ x^{n_j} &= g^{n_j \ell_j}, \\ g^{n_j \log_g x} &= g^{n_j \ell_j}, \\ n_j \log_g x &\equiv n_j \ell_j \bmod n, \\ \log_g x &\equiv \ell_j \bmod p_j^{k_j}. \end{aligned}$$

Par conséquent, une simple application du théorème chinois permet de recomposer la valeur de $\log_g x \bmod n$. Une grossière majoration de cette étape de théorème chinois est la cause de l'apparition du terme $(\log n)^3$.

Montrons maintenant que :

$$\mathsf{DL}(\mathsf{G}_j) \in O\left(k_j\left(\mathsf{DL}(\mathsf{G}'_j) + O((\log n)^2)\right) + \epsilon\right).$$

Nous nous intéressons donc au calcul de logarithmes discrets dans G_j. Sans restriction de généralité, on peut supposer pour se ramener à cette situation que $n = p^k$ et $\mathsf{G}' \cong \mathbb{Z}/p\mathbb{Z}$. En employant l'application définie par

$$\rho : \left\{ \begin{array}{ccc} \mathsf{G} & \longrightarrow & \mathsf{G}' \\ t & \longmapsto & t^{n'} \end{array} \right.$$

on peut calculer aisément $\lambda = \log x \bmod p$. Il s'ensuit que $xg^{-\lambda}$ est d'ordre divisant $p^{k-1} = n'$. Soit H le groupe de cardinal n' constitué des tels éléments, engendré par $h = g^p$. On a :

$$\log_g x = \lambda + p \log_h \frac{x}{g^\lambda}.$$

En d'autres termes, on a montré :

$$\mathsf{DL}(\mathsf{G}) \in O\left(\mathsf{DL}(\mathsf{G}') + \mathsf{DL}(H) + O\left((\log n)^2\right)\right) + \epsilon\right).$$

La quantité ϵ dans la formule précédente correspond au calcul de $xg^{-\lambda}$, qui nécessite $O(\log p)$ multiplications dans G.

On conclut par une récurrence sur k. Le résultat que l'on cherche à montrer est vrai pour $k = 1$, puisque dans ce cas $\mathsf{G} = \mathsf{G}'$, et se déduit donc de la formule qui précède, les termes ϵ s'additionnant pour donner effectivement au final un nombre d'opérations dans G borné par $\sum_j k_j \log p = \log n$. ∎

Algorithme **Pohlig-Hellman**
Entrée : x, élément du groupe engendré par g
Sortie : Le logarithme de x en base g

```
function PohligHellmanLog(g,x)
    // m, p_j, k_j, n_j définis comme ci-dessus
    for j in [1..m] do
        y:=x^{n_j};
        h:=g^{n_j};
        ℓ_j:=0;
        for s in [k_j..1 by -1] do
            // Invariant: ord(y)|ord(h) = p_j^s
            λ:=Log(h^{p_j^{s-1}},y^{p_j^{s-1}});
            ℓ_j+:=p_j^{k_j-s}λ;
            y:=yh^{-λ};
            h:=h^p;
        end for;
    end for;
    return ChineseRemainderTheorem([ℓ_j:j in [1..m]],[p_j^{k_j}:j in [1..m]]);
end function;
```

Programme 2.1: Algorithme de Pohlig-Hellman

Le pseudo-code 2.1 fournit une écriture en langage Magma de cet algorithme.

L'algorithme de Pohlig-Hellman s'applique au cas où l'ordre du groupe dans lequel on souhaite calculer des logarithmes discrets n'est pas premier. Toutefois, il se peut que cette réduction soit de peu d'intérêt, dans la situation où le meilleur algorithme pour résoudre le logarithme discret sur un sous-groupe n'est pas différent de celui utilisé pour la résolution du logarithme discret sur le groupe entier. Nous détaillons maintenant quelques algorithmes spécifiques dans les groupes qui nous intéressent.

2.2 Les algorithmes exponentiels

Parmi les algorithmes permettant de résoudre le problème du logarithme discret sur un groupe fini G de cardinal n, on trouve d'abord les algorithmes *exponentiels* (en $\log n$), ayant plus exactement une complexité en $O(\sqrt{n})$. Le point intéressant est que de tels algorithmes existent pour n'importe quel groupe G, pourvu qu'il satisfasse aux hypothèses minimales suivantes (ces hypothèses correspondent à la notion de *groupe générique*, au sens de [Nec94, Sho97]).

Définition 2.3 (Groupe générique). *Pour un groupe fini G de cardinal n, on fait les hypothèses minimales suivantes. On suppose qu'il existe un entier $\alpha \geqslant 0$ tel que :*

- *Les éléments de G sont représentés de façon unique sur $O((\log n)^\alpha)$ bits.*
- *Les opérations dans le groupe G (multiplication, inversion) se calculent en $O((\log n)^\alpha)$.*
- *Le cardinal du groupe G est connu.*

Cette hypothèse n'est pas tout à fait anodine. Elle est néanmoins satisfaite pour l'immense majorité des groupes rencontrés en cryptologie (à l'exception possible des groupes de tresses).

Dans bien des cas la valeur minimale possible de α est différente dans les deux conditions mentionnées ci-dessus, mais pour la présentation que l'on fait ici cela importe peu. Il convient de noter que dans certains cas, les éléments peuvent être avantageusement représentés de façon non unique, pour être réduits tardivement. Quitte à incorporer cette réduction dans la complexité des opérations de groupe, cette situation est incluse dans notre modèle.

Nous présentons ici deux méthodes en temps $O(\sqrt{n})$ pour la résolution du problème du logarithme discret, et une troisième qui présente l'avantage d'être aisément distribuable.

Baby-step / giant-step

L'algorithme dit « *baby-step / giant-step* » a été introduit par Shanks en 1971 [Sha71], à l'origine pour déterminer la structure de groupes de classes de corps quadratiques. Il s'applique très aisément au cadre du logarithme discret. L'algorithme fonctionne comme suit, pour calculer le logarithme en base g de l'élément $x \in \mathsf{G}$.

1. Soit $m = \lceil \sqrt{n} \rceil$.
2. Soit $a_i = x\,(g^{-m})^i$ pour $i \in [\![0..m]\!]$.
3. Pour $j \in [\![0..m]\!]$, si $\exists i,\ a_i = g^j$, retourner $mi + j$.

Si le groupe G vérifie les hypothèses énoncées plus haut, alors le tableau des a_i occupe une place mémoire en $O(\sqrt{n}(\log n)^\alpha)$, et le test d'appartenance peut y être effectué par hachage. La complexité en temps de l'algorithme est donc $O(\sqrt{n}(\log n)^\alpha)$.

Pollard rho

L'algorithme connu sous le nom de *Pollard rho* [Pol75, Pol78], est à la fois une méthode de factorisation (obtenant un facteur p de N en temps $\sqrt{p}$, donc adaptée pour trouver de petits facteurs), et une méthode de calcul de logarithme discret. Nous adoptons ici une description adaptée au contexte du logarithme discret.

Supposons comme précédemment que le groupe G vérifie les hypothèses données au début de ce chapitre. Notons que l'existence de l'algorithme de Pohlig-Hellman vu en 2.1 nous permet de nous concentrer sur le cas où le cardinal n de G est un nombre premier, ou tout du moins supposé l'être. Donnons-nous une fonction aléatoire f de G dans G, qui rende possible le « suivi » du logarithme discret. Plus exactement, on souhaite appliquer la fonction f à des éléments de la forme $x^a g^b$, pour obtenir une écriture de la forme

$$f(x^a g^b) = x^{a'} g^{b'},$$

où l'expression du couple (a', b') en fonction de a et b est connue. Un exemple de telle fonction sera donné plus loin.

L'algorithme utilise les propriétés classiques du graphe d'une fonction aléatoire f d'un ensemble fini dans lui-même. Ces propriétés sont obtenues à l'aide de la considération des séries génératrices exponentielles correspondant aux graphes fonctionnels. Elles sont exposées par exemple dans [FO90]. On rappelle ici très rapidement celles qui nous intéressent :

Proposition 2.4. *Soit E un ensemble fini de cardinal n et f une fonction aléatoire de E dans E. Soit Γ le graphe orienté ayant pour sommets les éléments de E et pour arêtes les $(x, f(x))$ pour $x \in E$. On a alors, en moyenne :*

– *La plus grande composante connexe de Γ est de taille $O(n)$.*

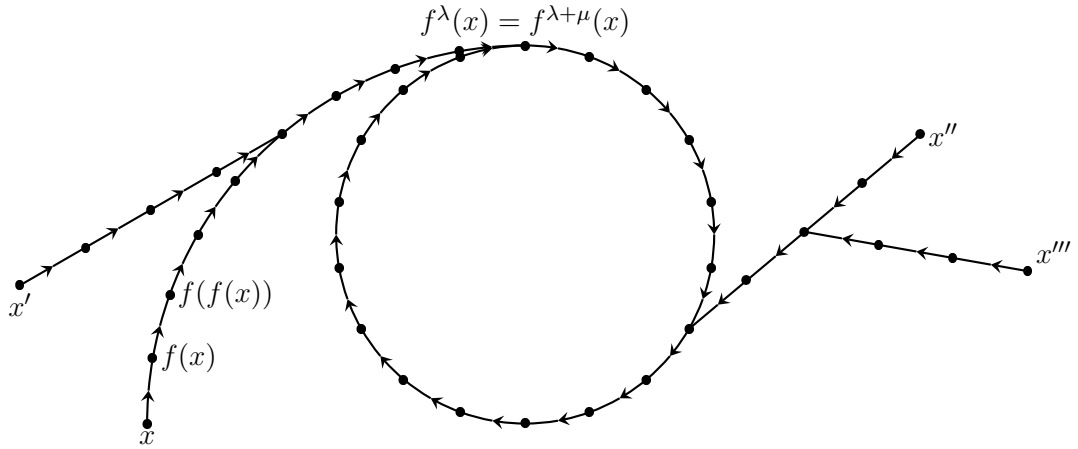


Figure 2.2 – Une composante connexe d'un graphe fonctionnel

- *Le plus grand cycle du graphe de* Γ *est de taille* $O(\sqrt{n})$.
- *La distance maximale d'un sommet à un cycle est* $O(\sqrt{n})$.

L'algorithme calcule des itérations de la suite $v_i = f(v_{i-1})$, c'est-à-dire qu'il effectue un chemin dans le graphe Γ, à partir d'un point de départ donné. La figure 2.2 donne l'aspect typique d'un tel chemin, ainsi que des possibles autres parties du graphe qui appartiennent à la même composante connexe. On constate que le chemin contient nécessairement un cycle. On obtient le logarithme discret de x si l'on sait mettre en évidence la présence d'un cycle. On utilise pour ça l'algorithme classique dû à Floyd (cf [Knu98, 3.1, exercice 6]) :

1. Soit $(u_i, a_i, b_i) = (xg^r, 1, r)$ pour $i = 1, 2$ et r aléatoire.
2. Calculer $u_1 \leftarrow f(u_1)$, et calculer le nouveau couple (a_1, b_1), .
   $u_2 \leftarrow f(f(u_2))$, et calculer le nouveau couple (a_2, b_2).
3. Si $u_1 = u_2$, retourner $\frac{b_1 - b_2}{a_2 - a_1}$. Sinon reprendre l'étape 2. Si $a_2 - a_1$ n'est pas inversible modulo n, alors cela signifie que l'on a identifié un facteur de n. On peut donc utiliser l'algorithme de Pohlig-Hellman vu en 2.1 pour reprendre le calcul modulo chacun des facteurs, avec une efficacité accrue.

Comme on sait que le plus grand cycle du graphe de f est de taille $O(\sqrt{n})$, on est assuré que le cycle de la suite des v_i est de taille au plus $O(\sqrt{n})$. Alors on sait que l'on a $u_1 = u_2$ après $O(\sqrt{n})$ étapes.

La construction originelle de la fonction f proposée par Pollard dans [Pol78] est :

$$f : \begin{cases} \mathsf{G} & \longrightarrow & \mathsf{G} \\ u & \longmapsto & \begin{cases} ux \text{ si } u \in \mathsf{G}_1, \\ x^2 \text{ si } u \in \mathsf{G}_2, \\ gx \text{ si } u \in \mathsf{G}_3, \end{cases} \end{cases}$$

où $(\mathsf{G}_1, \mathsf{G}_2, \mathsf{G}_3)$ est une partition du groupe G en trois ensembles de taille comparable. Nous voyons ainsi qu'il est facile d'exprimer (a', b') tels que $f(x^a g^b) = x^{a'} g^{b'}$.

Pour obtenir de bons résultats avec l'algorithme de *Pollard rho*, il faut que cette fonction f se rapproche le plus possible du cas moyen (parmi les fonctions de G dans G). En faisant

quelques hypothèses sur f, Sattler et Schnorr [SS85] ont montré qu'un tel choix de partitions en trois sous-ensembles ne fournissait pas un comportement très proche du cas moyen, et qu'il était préférable de partitionner G en un plus grand nombre de sous-ensembles. Expérimentalement il a été constaté dans [Tes01] qu'une partition en vingt sous-ensembles conférait à la fonction f un comportement statistiquement plus proche d'une fonction aléatoire, d'où un résultat meilleur.

Parallel collision search

Un algorithme cousin de la méthode rho qui permette une distribution efficace sur de nombreux nœuds de calcul est l'algorithme de recherche de collisions présenté dans [vOW99]. On se contente ici d'en faire une description rapide. Cet algorithme a pour fondements les mêmes propriétés du graphe de f vue comme fonction aléatoire. Puisque statistiquement il existe une composante connexe « géante », on essaie de mettre en évidence deux chemins qui « tombent » sur cette composante connexe en partant de différents points de départ. Lorsque l'on parvient à rejoindre de tels chemins, on obtient le résultat recherché.

On décrit ici l'algorithme dans une optique « maître-esclave ». Le concept nouveau est celui de *point distingué*, introduit à l'origine dans [QD90], et repris ici. On recherche uniquement les collisions entre les points distingués. La définition de ce qui constitue un point distingué est arbitraire, elle sert juste à alléger les calculs (on choisit typiquement de déterminer une fraction constante des points comme étant distingués, par exemple au vu de leur écriture binaire).

Maître

1. Soit $L = \{\}$.
2. Pour chaque *point distingué* (y, a, b) détecté par les esclaves, s'il existe un autre triplet $(y', a', b') \in L$ avec $y' = y$ et $a' \neq a$, alors retourner $\frac{b-b'}{a'-a}$. Sinon ajouter (y, a, b) à L.

Esclaves

1. Soit (a, b) aléatoire dans $(\mathbb{Z}/n\mathbb{Z})^2$, et $y = x^a g^b$.
2. $y \leftarrow f(y)$, et calculer le nouveau couple (a, b).
3. Si y est *distingué*, remonter au maître (sous la forme (y, a, b)).
4. Reprendre à l'étape 2 jusqu'à la $O(\sqrt{n})^{\text{ème}}$ itération. Ensuite recommencer en 1.

En vertu des mêmes propriétés du graphe de f que celles évoquées précédemment, on est assuré que le maître « trouve » des points distingués au bout d'un temps $O(\sqrt{n})$.

Champ d'application

Comme on va le voir par la suite, il existe dans certains cas des algorithmes ayant une complexité bien meilleure que ceux mentionnés ici. Toutefois, on ne fait pratiquement pas d'hypothèse ici sur le groupe G ; les restrictions pratiques imposées par l'énoncé 2.3 sont insignifiantes. Cela a pour conséquence que les algorithmes exposés ici s'appliquent là où aucune autre méthode n'est valide. En vérité, ces hypothèses sont les hypothèses les plus faibles que l'on peut faire pour qu'un groupe soit utilisable en cryptographie, et dans ce contexte, Shoup a montré dans [Sho97] que les algorithmes présentés ici étaient les meilleurs possibles.

Idéalement, bien sûr, un cryptosystème est conçu pour éviter les attaques avancées telles que celles qui seront décrites plus loin. Reste donc, pour effectuer la cryptanalyse de systèmes qui parviennent à éviter ces attaques, le seul choix des méthodes en $O(\sqrt{n})$ évoquées ici. Par exemple, Harley a cassé de nombreux challenges CERTICOM en utilisant l'algorithme de recherche de collisions décrit plus haut [Cer, Har].

Une conséquence du caractère générique des algorithmes décrits ici est que dans le cas du logarithme discret, la « force brute » n'est *jamais* la seule et meilleure attaque contre un cryptosystème. L'emploi d'un algorithme on $O(\sqrt{n})$ comme ici est immédiatement plus efficace que $O(n)$.

En outre, en employant l'algorithme de Pohlig-Hellman décrit en 2.1, on voit que si p est le plus grand facteur premier de n, une borne supérieure sur la complexité du logarithme discret dans un groupe de cardinal n est $O(\sqrt{p})$.

2.3 L'algorithme d'Adleman

2.3.1 Présentation des algorithmes de calcul d'index

Les algorithmes sous-exponentiels pour résoudre le problème du logarithme discret appartiennent tous à la famille des algorithmes dits d'*index-calculus*. Ces algorithmes reposent sur l'existence d'éléments *friables* dans le groupe considéré. Cette notion est particulièrement facile à mettre en évidence dans le cas des corps finis de caractéristique 2, sur lesquels nous allons nous concentrer. On peut avoir aussi des situations plus générales où une notion d'élément friable apparaît [EG02].

Par *sous-exponentiel*, on entend un algorithme dont le temps de calcul a pour expression $L_T(\alpha, c)$, ou T est la *taille* de l'entrée. Pour le cas d'un calcul de logarithmes discrets dans un groupe G de cardinal n, cette taille est $\log \#\mathsf{G}$, donc on s'intéresse à l'expression $L_{\log \#\mathsf{G}}(\alpha, c)$. La fonction L qui intervient ici a déjà été définie au chapitre précédent. Nous rappelons qu'elle correspond plus exactement à une *classe de fonctions*, qui s'écrit comme suit :

$$L_{\log \#\mathsf{G}}(\alpha, c) = O\left(\exp\left(c(1+o(1))\,(\log \#\mathsf{G})^{\alpha}\,(\log\log \#\mathsf{G})^{1-\alpha}\right)\right).$$

Afin de ne pas alourdir l'exposé avec des notations superflues, on emploiera la notation L pour noter les complexités rencontrées, en gardant à l'esprit qu'il s'agit là d'une classe de fonctions.

Les algorithmes mentionnés dans cette section sont sous-exponentiels, avec une complexité $L_{\log \#\mathsf{G}}\left(\frac{1}{2}, c\right)$ pour l'algorithme d'Adleman, et $L_{\log \#\mathsf{G}}\left(\frac{1}{3}, c\right)$ pour l'algorithme de Coppersmith et le *function field sieve* (FFS). D'autres algorithmes sous-exponentiels existent, par exemple le crible algébrique pour la factorisation d'entiers (factorisant l'entier N en $L_{\log N}\left(\frac{1}{3}, c\right)$), ou son analogue pour le logarithme discret, de même complexité. Un algorithme de calcul d'index existe enfin dans les jacobiennes de courbes hyperelliptiques de genre grand [ADH94, Gau00a], il est sous-exponentiel de la forme $L_{\log \#\mathsf{G}}\left(\frac{1}{2}, c\right)$. Dans presque tous ces cas, on parle de complexité heuristique plutôt que prouvée puisque l'on repose sur des hypothèses comme « cette quantité se comporte comme si elle était aléatoire ». Bien que de telles hypothèses soient corroborées par l'expérience, on a parfois des difficultés à les prouver en toute rigueur.

L'exposant α qui intervient dans l'écriture $L_{\log \#\mathsf{G}}(\alpha, c)$ sert à moduler entre le polynomial ($\alpha = 0$) et l'exponentiel ($\alpha = 1$). À ce jour, aucun algorithme sous-exponentiel n'est mentionné dans la littérature avec une constante α strictement plus petite que $\frac{1}{3}$.

2.3.2 Présentation de l'algorithme d'Adleman

Nous nous concentrons désormais sur le calcul de logarithmes discrets dans $\mathbb{F}_{2^n}$. Par conséquent, nous abandonnons la notation $n = \#\mathsf{G}$.

Dans la série d'algorithmes que nous présentons pour calculer des logarithmes discrets sur $\mathbb{F}_{2^n}$, l'algorithme d'Adleman [Adl79] est le plus ancien, et aussi le plus simple. Nous nous servirons de cet algorithme pour dégager la trame générale des algorithmes d'*index-calculus*.

La première tâche consiste à mettre en évidence une notion de friabilité dans $\mathbb{F}_{2^n}$. On souhaite pouvoir dire d'un élément qu'il est friable s'il se décompose en « petits » facteurs. Cela appelle nécessairement une idée de « taille » des éléments du groupe.

La situation de $\mathbb{F}_{2^n}$ est particulièrement simple en ce qui concerne ces notions de taille et de friabilité, puisqu'une réponse immédiate nous est donnée par la factorisation du polynôme de plus petit degré parmi les représentants d'un élément : si ce polynôme a tous ses facteurs de degré inférieur à une borne b (par exemple), on dit que l'élément en question est friable.

Voici quelques exemples. Plaçons nous sur $\mathbb{F}_{2^{127}} = \mathbb{F}_2[X]/\big(X^{127} + X + 1\big)$.

- L'élément X^{400} du corps admet comme unique représentant de degré < 127 le polynôme $X^{22} + X^{21} + X^{20} + X^{19}$. Ce polynôme se factorise en $X^{19}(X+1)^3$. L'élément X^{400} est donc 1-*friable*.
- On voit facilement que le représentant minimal de $1/X$ est $X^{126} + 1 = (X^{63} + 1)^2 = (\frac{X^{2^6}+X}{X})^2$. Donc en vertu de la propriété A.12, $1/X$ est un élément 6-friable.

Comme cela apparaît dans l'exemple, le polynôme de définition du corps considéré doit être fixé une fois pour toutes au début des calculs. Supposons donc que nous avons fixé une représentation de $\mathbb{F}_{2^n}$:

$$\mathbb{F}_{2^n} = \mathbb{F}_2[X]/(f(X)).$$

L'algorithme procède en trois phases consécutives.

Phase 1

Le *degré* nous fournissant la notion voulue d'éléments « petits », il faut maintenant choisir[1] une *borne de friabilité*, notée b, qui paramètre la *base de facteurs* (*factor base*), notée $\mathcal{B}$:

$$\mathcal{B} = \{\pi \in \mathbb{F}_2[X],\ \pi \text{ irréductible},\ \deg \pi \leqslant b\}\,.$$

On peut remarquer qu'une valeur approchée du cardinal de $\mathcal{B}$ peut être obtenue sans peine. En utilisant la proposition A.15, on a : $\#\mathcal{B} \approx \frac{2^{b+1}}{b}$.

La suite des opérations consiste à engendrer des *relations* entre les π_i (plus exactement, entre les $\log \pi_i$). Dans le cadre de l'algorithme d'Adleman, on répète autant que nécessaire l'opération suivante :

- Choisir m au hasard dans $[\![0 \dots 2^n - 1]\!]$.
- Calculer $X^m \bmod f$. S'il est b-friable, conserver sa factorisation $X^m \equiv \prod_i \pi^{e_i} \bmod f$.

Nous avons supposé implicitement que X est un élément primitif, c'est-à-dire que la classe de X dans $\mathbb{F}_{2^n}$ est un élément générateur du groupe multiplicatif. Les logarithmes sont exprimés en base X.

[1] Ce choix sera détaillé lors de l'analyse. Pour l'algorithme d'Adleman, $b \sim \sqrt{n}$.

Phase 2

Si l'opération précédente est répétée un nombre suffisant de fois, on dispose d'un grand nombre de relations. Si l'on prend le logarithme de chacune de ces relations, on obtient des relations de la forme :

$$m = \sum_i e_i \log \pi_i \bmod (2^n - 1).$$

Ceci fait du vecteur $(\log \pi_i)_i$ la solution d'un système linéaire défini modulo $(2^n - 1)$. Si l'on dispose de suffisamment de relations (il nous en faut $\#\mathcal{B} \approx \frac{2^{b+1}}{b}$), ce système linéaire possède avec forte probabilité une solution uniquement déterminée. Il faut donc le résoudre.

Phase 3

La dernière phase consiste à obtenir le logarithme discret d'un élément arbitraire Q de $\mathbb{F}_{2^n}$. Pour cela, on calcule le produit QX^m pour m aléatoire, jusqu'à ce que l'on obtienne un élément friable. En remplaçant les valeurs désormais connues des $\log \pi_i$, on déduit $\log Q$.

2.3.3 Analyse de l'algorithme d'Adleman

Comme l'algorithme d'Adleman apparaîtra comme étant de complexité sous-exponentielle, on se permet de ne pas prendre en compte les opérations de complexité polynomiale en la taille des entrées. On calcule le terme principal du développement asymptotique des complexités qui nous intéressent. Nous définissons la notation $\overset{\log}{\sim}$ comme désignant l'équivalence de deux grandeurs sur une échelle logarithmique, c'est-à-dire :

$$\begin{aligned} f \overset{\log}{\sim} g &\overset{\text{déf}}{\Leftrightarrow} \log f \sim \log g, \\ &\overset{\text{déf}}{\Leftrightarrow} (\log f - \log g) \in o(\log f). \end{aligned}$$

Cette notation est adaptée aux grandeurs qui nous intéressent, puisque l'on montre que si f est *super-polynomiale* en n (i.e. $O(\log n) \subset o(\log f)$), alors $\frac{f}{g} \in \mathrm{POLY}(n)$ implique $f \overset{\log}{\sim} g$. On montre aussi que toute fonction f dans la classe de complexité $L_n(\alpha, c)$ vérifie :

$$f \overset{\log}{\sim} \exp\left(cn^{\alpha}(\log n)^{1-\alpha}\right).$$

On peut aussi définir la « fonction » L de cette façon.

Dans tous les développements que nous ferons pour analyser les algorithmes, log désigne le logarithme népérien et $\log_2$ le logarithme en base 2.

Tout d'abord, un ingrédient essentiel de l'analyse est l'évaluation de la probabilité pour un polynôme d'être b-friable. Le résultat asymptotique suivant a été obtenu par Odlyzko [Odl85].

Proposition 2.5 (Probabilité de friabilité). *La probabilité qu'un polynôme aléatoire de $\mathbb{F}_2[X]$ de degré d soit b-friable, pour b dans la plage $[d^{\frac{1}{100}} \dots d^{\frac{99}{100}}]$, est asymptotiquement :*

$$\wp(d, b) = \left(\frac{d}{b}\right)^{-(1+o(1))\frac{d}{b}}.$$

Démonstration. Ce résultat est une application de la méthode du col [FS94]. La démonstration n'est pas reprise ici. ■

Il est aisé, en appliquant la propriété précédente, de voir que la quantité de travail (tests de friabilité, factorisations) nécessaire dans la *première phase* est (puisque $X^m \bmod f$ se comporte comme un polynôme aléatoire) :

$$\begin{aligned}\mathsf{W}_1 &\overset{\log}{\sim} \frac{2^{b+1}}{b}\wp(n,b),\\ &\overset{\log}{\sim} \frac{2^{b+1}}{b}\left(\frac{n}{b}\right)^{(1+o(1))\frac{n}{b}},\\ &\overset{\log}{\sim} 2^b\left(\frac{n}{b}\right)^{\frac{n}{b}}.\end{aligned}$$

tandis que la deuxième phase de l'algorithme requiert la résolution d'un système linéaire. Nous verrons que les algorithmes d'algèbre linéaire creuse développés dans la partie II de ce mémoire permettent de résoudre un tel système linéaire en temps quadratique, en tirant parti de son caractère *creux*. Le travail de la seconde phase est donc :

$$\begin{aligned}\mathsf{W}_2 &\overset{\log}{\sim} \left(\frac{2^{b+1}}{b}\right)^2,\\ &\overset{\log}{\sim} 2^{2b}.\end{aligned}$$

Nous souhaitons obtenir un temps de calcul minimal pour $\mathsf{W}_1 + \mathsf{W}_2$. Ainsi, on souhaite minimiser le maximum[2] de $\log \mathsf{W}_1$ et $\log \mathsf{W}_2$. Cet optimum nous donnera une expression de b en fonction de n. La fonction $\log \mathsf{W}_2$ est une fonction croissante de b, pour n fixé. La fonction $\log \mathsf{W}_1$, en revanche, est d'abord décroissante puis croissante. S'il est possible d'égaler les quantités $\log \mathsf{W}_1$ et $\log \mathsf{W}_2$ dans la plage de valeurs où la fonction b est décroissante, alors on détermine ainsi le point où $\max(\mathsf{W}_1, \mathsf{W}_2)$ est minimal. L'équation est :

$$\frac{n}{b}\log\frac{n}{b} + b\log 2 \sim 2b\log 2.$$

Posons maintenant $b = cn^\alpha(\log n)^\beta$. L'égalité ci-dessus est satisfaite pour :

$$b \sim \sqrt{\frac{n\log n}{2\log 2}}.$$

On vérifie facilement que notre hypothèse sur la décroissance de $\log \mathsf{W}_1$ est satisfaite. La somme des temps de calcul des deux premières phases est alors :

$$\begin{aligned}\mathsf{W}_1 + \mathsf{W}_2 &\overset{\log}{\sim} \exp\left(\sqrt{2\log 2}\sqrt{n\log n}\right),\\ \mathsf{W}_1 + \mathsf{W}_2 &\overset{\log}{\sim} L_n\left(\frac{1}{2}, \sqrt{2\log 2}\right).\end{aligned}$$

La troisième phase de l'algorithme a une complexité bien moindre (mais néanmoins sous-exponentielle), en $L_n\left(\frac{1}{2}, \sqrt{\frac{\log 2}{2}}\right)$.

[2] On peut démontrer en effet que W_1 et W_2 tendant vers $+\infty$ avec n, si $\mathsf{W}_1 \overset{\log}{\sim} f$ et $\mathsf{W}_2 \overset{\log}{\sim} g$, alors $\mathsf{W}_1 + \mathsf{W}_2 \overset{\log}{\sim} \max(f, g)$.

2.3.4 Améliorations de l'algorithme d'Adleman

Le schéma simple de l'algorithme d'Adleman ne fournit hélas pas un algorithme très efficace. Deux améliorations intéressantes ont été apportées par Blake, Fuji-Hara, Mullin et Vanstone dans [BFHMV84, BMV85]. Nous les décrivons ici. Toutes deux visent à augmenter la probabilité de friabilité des paires considérées lors de la première phase, diminuant ainsi le nombre de tests à effectuer.

Emploi de l'algorithme d'Euclide

Le polynôme $X^m \bmod f$ que l'on souhaite factoriser dans l'algorithme d'Adleman est, de degré égal à $n-1$, ou bien à peine inférieur (le degré moyen étant $n-2$). Nommons ce polynôme $A(X)$. Une idée relativement simple pour « décomposer » A en produits d'éléments de $\mathcal{B}$ consiste à appliquer l'algorithme d'Euclide étendu à A et f pour obtenir une équation de la forme :

$$\begin{aligned} AU + fV &= W, \\ AU &\equiv W \bmod f. \end{aligned}$$

Si l'on arrête l'algorithme d'Euclide à la moitié des calculs, les polynômes U et W sont de degré $\leqslant \frac{n}{2}$.

L'intérêt de cette décomposition est que deux polynômes de degré $\frac{n}{2}$ ont une probabilité plus grande d'être simultanément friables qu'un seul polynôme de degré n ($\left(\frac{n}{2b}\right)^{\frac{n}{b}}$ contre $\left(\frac{n}{b}\right)^{\frac{n}{b}}$). Lorsque U et W sont friables et s'écrivent respectivement $\prod_i \pi_i^{e_i}$ et $\prod_i \pi_i^{f_i}$, on a alors la relation :

$$\begin{aligned} X^m \prod_i \pi_i^{e_i} &\equiv \prod_i \pi_i^{f_i} \mod f, \\ \prod_i \pi_i^{f_i - e_i} &\equiv X^m \mod f, \\ \sum_i (f_i - e_i) \log \pi_i &\equiv m \mod (2^n - 1). \end{aligned}$$

Cette décomposition ne coûte pas beaucoup, mais fait gagner beaucoup de temps dans le calcul de relations. Néanmoins, son influence sur la complexité finale est invisible, puisqu'elle est dissimulée dans la composante $o(1)$ de l'exposant. Cela montre combien les complexités sous-exponentielles sont sensibles aux variations, même invisibles, de leurs paramètres.

Équations systématiques

Les travaux de [BFHMV84, BMV85] étaient concentrés sur le cas $\mathbb{F}_{2^{127}}$. Dans ce cas précis, une méthode assez efficace pour fabriquer des relations repose sur le résultat suivant :

Proposition 2.6. *Soit A un polynôme irréductible de degré k dans $\mathbb{F}_2[X]$. Soit B un autre polynôme quelconque. Alors les facteurs irréductibles de $A(B(X))$ sont de degré multiple de k.*

DÉMONSTRATION. Soit K l'extension de degré k de $\mathbb{F}_2$ définie par A. Soit Ω une clôture algébrique de $\mathbb{F}_2$. Soit α une racine de $A(B(X))$ dans Ω. On a alors un morphisme injectif (plongement) :

$$\phi : \begin{cases} K & \longrightarrow & \mathbb{F}_2(\alpha) \\ P(X) & \longmapsto & P(B(\alpha)) \end{cases}$$

(on paramètre ici les éléments de K sous la forme $P(X)$, où P est un polynôme de $\mathbb{F}_2[X]$). Le fait que cette application est injective découle précisément de l'irréductibilité de A : si P et Q sont deux polynômes sur $\mathbb{F}_2$ tels que $P(B(\alpha))$ et $Q(B(\alpha))$ sont égaux, alors le polynôme $P - Q$ a pour racine la racine $B(\alpha)$ de A. Donc $P - Q$ est un multiple de A, donc $P(X)$ et $Q(X)$ sont égaux dans K.

Comme l'injection ϕ existe, $\mathbb{F}_2(\alpha)$ admet $\mathbb{F}_{2^k}$ comme sous-corps, donc le polynôme minimal de α est de degré multiple de k. ■

Partant de ce résultat, plaçons-nous comme dans l'exemple ci-dessus dans le cas $\mathbb{F}_{2^{127}} = \mathbb{F}_2[X]/\left(X^{127} + X + 1\right)$ (c'est-à-dire, on prend $f(X) = X^{127} + X + 1$). Pour un polynôme irréductible $A(X)$ de degré $k \leqslant b$, on a :

$$\begin{aligned} X^{128} &\equiv X^2 + X, \bmod f \\ A(X^{128}) &\equiv A(X^2 + X) \bmod f, \\ A(X)^{128} &\equiv A(X^2 + X) \bmod f. \end{aligned}$$

Dans cette situation, si le membre de droite de la dernière équation se factorise en deux polynômes de degré k, on obtient une relation entre trois éléments de la base de facteurs, tous de degré k. Ceci se produit dans la moitié des cas. Si A décrit donc l'ensemble de la base de facteurs, on obtient ainsi la moitié du nombre de relations voulues.

Hélas, ce schéma se généralise mal. Il est particulièrement bien adapté[3] au cas de $\mathbb{F}_{2^{127}}$, mais lorsque n n'est pas proche d'une puissance de 2, on n'obtient plus un aussi grand nombre de relations. Néanmoins, c'est l'introduction de considérations un peu plus structurelles sur la caractéristique 2, dans la suite des équations systématiques, qui a conduit à l'élaboration de l'algorithme de Coppersmith.

2.4 L'algorithme de Coppersmith

2.4.1 Présentation

Les deux améliorations de l'algorithme d'Adleman que nous venons d'exposer ne modifient hélas pas profondément les paramètres de l'expression $L_n(\alpha, c)$. Leur influence se cantonne à l'expression $o(1)$ dans cette expression. Coppersmith [Cop84] en revanche, est parvenu à faire baisser la complexité du calcul de manière bien plus fondamentale, amenant ainsi une avancée très substantielle pour le calcul de logarithmes discrets dans $\mathbb{F}_{2^n}$.

La borne de friabilité qui contrôle la taille de la base de facteurs est dans l'algorithme de Coppersmith de l'ordre de $O(n^{\frac{1}{3}}(\log n)^{\frac{2}{3}})$, et les polynômes à factoriser sont de degré bien moindre. La complexité globale de l'algorithme en est ainsi diminuée. L'algorithme de Coppersmith est l'algorithme le plus ancien permettant d'atteindre la complexité $L_n\left(\frac{1}{3}, c\right)$ pour calculer des logarithmes discrets.

[3] Pour les mêmes raisons que celles qui font de $\mathbb{F}_{2^{127}}$ un candidat de choix pour l'implantation...

Polynôme de définition

Tout d'abord, l'algorithme de Coppersmith requiert que le polynôme de définition $f(X)$ employé pour l'extension soit de la forme $X^n + f_1(X)$, avec $f_1(X)$ du plus petit degré possible. Heuristiquement, obtenir f_1 de degré $O(\log n)$ est toujours faisable.

Paramètres de l'algorithme

Outre le choix du polynôme de définition, l'algorithme de Coppersmith requiert le choix de plusieurs paramètres, b, d, k, et h. Nous verrons en détail lors de l'analyse les valeurs asymptotiques de ces paramètres. Nous discuterons aussi l'influence respective de chacun d'entre eux. Pour l'instant, on se contente de mentionner les contraintes auxquelles ces paramètres sont soumis. Le paramètre b est, comme précédemment, la borne de friabilité. Le paramètre d est un entier comparable à b. Le paramètre k est une puissance de 2 (le plus souvent, $k = 4$). Le paramètre h est égal à $\lceil \frac{n}{k} \rceil$.

Obtention des relations

L'idée de Coppersmith est la suivante. Pour toutes les paires (A, B) de polynômes de $\mathbb{F}_2[X]$ premiers entre eux et de degré borné par d (par la suite, on appellera souvent cet espace « *espace de crible* », pour des raisons qui apparaîtront claires en 3.3), former les polynômes :

$$C = AX^h + B,$$
$$D = C^k \bmod f = A^k X^{hk-n} f_1 + B^k.$$

La dernière identité tient bien entendu au fait que k est une puissance de 2. Si C et D sont friables et s'écrivent respectivement $\prod_i \pi_i^{e_i}$ et $\prod_i \pi_i^{f_i}$, on a alors la relation :

$$\prod_i \pi_i^{ke_i} \equiv \prod_i \pi_i^{f_i} \mod f,$$
$$\prod_i \pi_i^{f_i - ke_i} \equiv 1 \mod f,$$
$$\sum_i (f_i - ke_i) \log \pi_i \equiv 0 \mod (2^n - 1).$$

Nous obtenons ainsi une méthode pour fabriquer des relations. Cette méthode nous offre la possibilité, en jouant sur les paramètres, d'équilibrer les degrés de C et D à des valeurs peu élevées. La complexité finale en bénéficie grandement, comme on le verra lors de l'analyse de l'algorithme.

Résolution du système linéaire

La partie « algèbre linéaire » ne se différencie pas particulièrement dans l'algorithme de Coppersmith. Nous verrons dans la partie II de ce mémoire comment nous avons traité le problème.

Une caractéristique particulière des matrices qui nous intéressent peut déjà être mise en évidence : les coefficients non nuls de la matrice sont répartis de manière très inégale dans les différentes colonnes, puisque le « poids » des colonnes correspondant aux polynômes de petit degré est beaucoup plus élevé. Plus exactement, la colonne « numérotée » par un polynôme

irréductible P de degré b a pour indice environ $\frac{2^b}{b}$, et ses coefficients sont non nuls avec probabilité $\frac{2}{2^b}$, le numérateur correspondant aux *deux* polynômes C et D qui sont factorisés. À titre de comparaison, cette répartition de densité est la même que pour l'algorithme du crible quadratique (le crible quadratique n'est pas décrit dans ces pages, mais le lecteur pourra consulter [CP01]) : si x est un nombre premier, alors la colonne d'indice $\frac{x}{\log x}$ indique la présence du facteur x dans la factorisation d'un certain entier. La probabilité correspondante est donc $\frac{1}{x}$. À un facteur multiplicatif près, cette densité est identique à celle que l'on a identifiée[4].

Calcul des logarithmes individuels

Si la seconde phase de l'algorithme ne présente pas de particularité majeure dans l'algorithme de Coppersmith par rapport à ce qui a déjà été exposé, ce n'est pas le cas de la troisième et dernière phase. Il convient de détailler comment un gain substantiel de complexité peut aussi être obtenu pour le calcul de logarithmes individuels. En effet, il n'est pas du tout évident que la méthode d'obtention des relations que nous venons de décrire puisse se transporter dans cette troisième phase. C'est pourtant le cas. La description suivante est reprise de [Cop84, BMV85].

Une première étape est l'obtention des logarithmes de polynômes de taille *moyenne*. On parvient à effectuer ce calcul par un mécanisme de descentes successives. Supposons que nous souhaitions obtenir le logarithme du polynôme Q, de degré q. Pour ce faire, notre but premier est d'obtenir une expression de $\log Q$ comme combinaison linéaire de logarithmes des polynômes de degré $< \sqrt{bq}$. Ceci s'obtient similairement aux techniques déjà employées pour obtenir des relations. On choisit des paramètres d' et k', dont on précisera la valeur lors de l'analyse. L'entier k' est une puissance de 2. On pose $h' = \lceil \frac{n}{k'} \rceil$. Considérons l'ensemble des paires (A, B) de degré inférieur ou égal à d' telles que $C = AX^{h'} + B$ est divisible par Q. Cet ensemble est un espace vectoriel sur $\mathbb{F}_2$, et en obtenir une base est aisé[5]. Comme précédemment, posons :

$$C = AX^{h'} + B, \text{ et } Q \mid C$$
$$D = C^{k'} \mod f = A^{k'} X^{h'k'-n} f_1 + B^{k'}.$$

Si D et C/Q sont simultanément $\sqrt{bq}$-friables et s'écrivent respectivement $\prod_i \pi_i^{e_i}$ et $\prod_i \pi_i^{f_i}$, on a alors la relation :

$$Q^{k'} \prod_i \pi_i^{k'e_i} \equiv \prod_i \pi_i^{f_i} \mod f,$$
$$\prod_i \pi_i^{f_i - k'e_i} \equiv Q^{k'} \mod f,$$
$$\sum_i (f_i - k'e_i) \log \pi_i \equiv k' \log Q \mod (2^n - 1).$$

Cette dernière expression est suffisante pour déduire la valeur de $\log Q$. Pour obtenir lors de l'analyse une complexité performante, la « bonne » valeur à choisir pour d' est choisie entre $\frac{q}{2}$ et q.

[4] Cette analyse est à peu près valable quand on examine les colonnes une à une. Elle est totalement erronée si l'on s'intéresse aux lignes, puisque les probabilités de présence des différents facteurs ne sont pas indépendantes.

[5] Toutefois, pour des raisons d'efficacité, on préfère traiter cette ensemble comme un réseau de l'espace $(\mathbb{F}_2[X])^2$. Ce point sera détaillé en 3.6.

Une fois que nous sommes en mesure de « descendre » du degré q au degré $\sqrt{bq}$, le procédé est simplement de répéter cette opération jusqu'à arriver au degré b. On peut résumer ainsi l'algorithme pour obtenir le logarithme d'un polynôme Q de degré arbitraire :

- En utilisant l'algorithme d'Euclide étendu (voir page 27), exprimer Q comme quotient de polynômes de degré $q \leqslant \frac{n}{2}$.
- Tant que $q > b$:
- Poser $q \leftarrow \sqrt{bq}$.
- Réduire à un produit de polynômes de degré $\leqslant q$.

À la i-ème étape de cette itération, la borne q vaut $b\left(\frac{n}{2b}\right)^{2^{-i}}$. Donc au bout de $\log n$ étapes en moyenne, on atteint le degré b.

2.4.2 Analyse

Avec la proposition 2.5, on peut estimer la probabilité de friabilité d'un polynôme aléatoire. En faisant l'hypothèse que les polynômes C et D calculés dans l'algorithme de Coppersmith sont aléatoires et indépendants[6], il est possible de leur appliquer cette estimation. Partons de la supposition que le nombre de paires de polynômes (A, B), à savoir 2^{2d+1}, est exactement suffisant pour produire le nombre de relations recherché. Le temps de calcul de la première phase est donc de l'ordre de 2^{2d+1}. Comme pour l'algorithme d'Adleman, supposons que la valeur optimale de b est telle que la première phase est strictement plus coûteuse pour une valeur de b plus petite, et la seconde phase strictement plus coûteuse pour une valeur de b plus grande. Sous cette hypothèse (que nous vérifierons plus loin), nous devons équilibrer les deux premières phases. On a ainsi la relation :

$$2^{2d+1} \overset{\log}{\sim} \left(\frac{2^{b+1}}{b}\right)^2,$$
$$d \sim b.$$

Nous exprimons maintenant le fait que les 2^{2d+1} paires produisent précisément $\frac{2^{b+1}}{b}$ relations.

$$\begin{aligned}
\frac{2^{b+1}}{b} &= 2^{2d+1}\wp(\deg C, b)\wp(\deg D, b),\\
\frac{2^{b+1}}{b} &= 2^{2d+1}\wp(d+h, b)\wp(kd, b),\\
2^b &\overset{\log}{\sim} \left(\wp(d+h, b)\wp(kd, b)\right)^{-1},\\
b\log 2 &\sim \frac{d+h}{b}\log\frac{d+h}{b} + \frac{kd}{b}\log\frac{kd}{b}.
\end{aligned}$$

Le membre de gauche contrôlant la valeur de b, on souhaite minimiser le membre de droite. Étant donné que $h \sim \frac{n}{k}$, on a intérêt à prendre $k \sim \sqrt{\frac{n}{d}}$. Les polynômes C et D ont alors un degré proche de $\sqrt{nd}$.

$$b\log 2 \sim 2\frac{\sqrt{nd}}{b}\log\frac{\sqrt{nd}}{b},$$

[6] En fait, cette assertion est fausse, voire grossièrement fausse. On montre en page 35 de quelle façon D dévie nettement du comportement d'un polynôme aléatoire, et il est manifeste que C et D ne sont pas indépendants.

$$b \log 2 \sim 2\sqrt{\frac{n}{b}} \log \sqrt{\frac{n}{b}}.$$

Nous voyons d'ores et déjà que notre hypothèse sur l'équilibrage des deux phases est justifiée : pour une valeur de b plus faible, la complexité de la première phase est accrue. En posant $b = cn^{\alpha}(\log n)^{\beta}$, on arrive aux conditions de minimalité suivantes :

$$\begin{cases} \alpha = \dfrac{1-\alpha}{2} & \longrightarrow \quad \alpha = \frac{1}{3}, \\ \beta = 1 - \frac{\beta}{2} & \longrightarrow \quad \beta = \frac{2}{3}, \\ c \log 2 = \frac{2}{\sqrt{c}} \frac{1-\alpha}{2} & \longrightarrow \quad c = \left(\frac{2}{3 \log 2}\right)^{\frac{2}{3}} \approx 0.97. \end{cases}$$

Cette expression nous donne la valeur asymptotique de la borne de friabilité b :

$$b \sim \left(\frac{2}{3 \log 2}\right)^{\frac{2}{3}} n^{\frac{1}{3}} (\log n)^{\frac{2}{3}}.$$

On déduit de cette valeur les complexités des phases 1 et 2 de l'algorithme :

$$\begin{aligned} \mathsf{W}_1 \overset{\log}{\sim} \mathsf{W}_2 &\overset{\log}{\sim} \exp\left(2 \log 2 \left(\frac{2}{3 \log 2}\right)^{\frac{2}{3}} n^{\frac{1}{3}} (\log n)^{\frac{2}{3}}\right), \\ &\overset{\log}{\sim} \exp\left(\left(\frac{32 \log 2}{9}\right)^{\frac{1}{3}} n^{\frac{1}{3}} (\log n)^{\frac{2}{3}}\right), \\ &\overset{\log}{\sim} L_n\left(\frac{1}{3}, \left(\frac{32 \log 2}{9}\right)^{\frac{1}{3}}\right). \end{aligned}$$

La complexité de la dernière phase de l'algorithme s'obtient de manière tout à fait similaire. La complexité de chaque passage du degré q au degré $\sqrt{bq}$ (appelons $\mathsf{w}_3(q)$ ce coût) est donnée de deux façons : par le nombre d'essais nécessaires, dirigé par la probabilité de friabilité, et par le nombre de paires disponibles. Comme précédemment, on a intérêt à choisir k' proche de $\sqrt{\frac{n}{d'}}$. Si l'on pose $d' = zq$, le nombre d'essais à faire pour pouvoir espérer trouver une paire friable est :

$$\begin{aligned} \mathsf{w}_3(q) &\overset{\log}{\sim} \wp\left(\sqrt{nd'}, \sqrt{bq}\right)^{-2}, \\ \log_2 \mathsf{w}_3(q) &\sim \sqrt{\frac{nd'}{bq}} \log_2 \frac{nd'}{bq}, \\ &\sim \sqrt{z}\sqrt{\frac{n}{b}} \log_2 \frac{n}{b}, \\ &\sim \sqrt{z} b. \end{aligned}$$

Le nombre de paires disponibles est $2^{2d'-q-1}$. Si l'on pose $b = xq$, on doit donc s'assurer que l'on a :

$$(2z - 1)q \geqslant \sqrt{z} b,$$

$$(2z-1) \geqslant \sqrt{z}x.$$

Étant donne que $x \leqslant 1$ par construction, la plus petite valeur de z possible est toujours comprise entre $\frac{1}{2}$ et 1. Dans le cas pessimiste où $z = 1$, on peut borner ainsi la valeur de $\log_2 \mathsf{w}_3(q)$:

$$\log_2 \mathsf{w}_3(q) \sim b.$$

Nous reviendrons plus en détail sur la façon dont se comporte z en fonction de x en 3.8.

La complexité précédente doit être multipliée par le nombre de polynômes à décomposer ainsi. Le nombre d'étages de décomposition est de l'ordre de grandeur de $\log n$. De plus, on peut se permettre de majorer grossièrement le nombre de facteurs supplémentaires créés à chaque étape par $\frac{n}{b}$. Il s'ensuit que le nombre total de polynômes décomposés ainsi est :

$$\left(\frac{n}{b}\right)^{\log_2 n} = \exp\left(\log_2 n \log \frac{n}{b}\right).$$

En utilisant l'expression précédemment déterminée pour b, à savoir $b = cn^{\frac{1}{3}} (\log n)^{\frac{2}{3}}$, avec la constante c qui vaut $\left(\frac{2/3}{\log 2}\right)^{\frac{2}{3}}$, on déduit :

$$\begin{aligned}
\log \mathsf{W}_3 &\sim \mathsf{w}_3 + \log_2 n \log \frac{n}{b}, \\
\log \mathsf{W}_3 &\sim b \log 2, \\
\log \mathsf{W}_3 &\sim c \log 2 n^{\frac{1}{3}} (\log n)^{\frac{2}{3}}.
\end{aligned}$$

En conclusion, la complexité globale de l'algorithme de Coppersmith est l'addition des composantes :

$$\begin{aligned}
\mathsf{W}_1 &\overset{\log}{\sim} L_n\left(\frac{1}{3}, \left(\frac{32 \log 2}{9}\right)^{\frac{1}{3}}\right) \overset{\log}{\sim} L_n\left(\frac{1}{3}, 1.35\right), \\
\mathsf{W}_2 &\overset{\log}{\sim} L_n\left(\frac{1}{3}, \left(\frac{32 \log 2}{9}\right)^{\frac{1}{3}}\right) \overset{\log}{\sim} L_n\left(\frac{1}{3}, 1.35\right), \\
\mathsf{W}_3 &\overset{\log}{\sim} L_n\left(\frac{1}{3}, \left(\frac{4 \log 2}{9}\right)^{\frac{1}{3}}\right) \overset{\log}{\sim} L_n\left(\frac{1}{3}, 0.67\right).
\end{aligned}$$

Cette analyse est valide lorsque l'on est dans le cas optimal où k peut être pris égal à une puissance de 2. Hélas, les puissances de 2 sont rares, et il se peut que la puissance de 2 choisie soit éloignée de la valeur optimale de k. Le pire cas à cet égard est celui où n est tel que la valeur optimale de k se trouve être de la forme $2^x\sqrt{2}$. On ne va pas refaire le travail d'analyse de la complexité correspondante, mais on donne juste la conséquence d'une telle situation : les constantes dans les expressions de W_1, W_2, et W_3 sont changées. On se retrouve ainsi avec $(4 \log 2)^{\frac{1}{3}}$ en lieu et place de $\left(\frac{32 \log 2}{9}\right)^{\frac{1}{3}}$. Numériquement, la valeur de 1.35 est transformée en 1.405.

2.4.3 Choix des paramètres

Les nombreux paramètres qui apparaissent dans l'algorithme de Coppersmith ont chacun une grande importance. Bien plus que leur valeur asymptotique, la donnée essentielle pour une implantation de l'algorithme passe par une bonne compréhension des tenants et aboutissants du choix de ces paramètres. C'est dans cette optique que nous passons ici en revue les implications des modifications que l'on peut faire sur chacun d'entre eux.

La borne de friabilité b

Le paramètre b est sans doute le paramètre le plus important dans l'algorithme ; il influe sur plusieurs points. Tout d'abord, puisque l'on est intéressé par la production de relations b-friables, il est évident que leur probabilité d'apparition augmente avec b. La contrepartie est double : d'abord la phase d'algèbre linéaire est rendue plus difficile par une valeur plus grande de b. Ensuite, le nombre de relations à obtenir croît avec b (il vaut $\frac{2^{b+1}}{b}$). Ce dernier effet est susceptible de prendre le pas sur la gain que représente la probabilité de friabilité accrue, car asymptotiquement, la valeur choisie de b est telle que ces deux effets s'équilibrent. Les pénalités induites par un accroissement de b peuvent donc être importantes.

Outre le temps de calcul, la phase d'algèbre linéaire peut aussi se heurter rapidement à des difficultés en termes d'espace mémoire. Les facteurs que l'on doit prendre en compte, en résumé, pour le choix de b, sont donc :

- L'influence sur W_1 (en théorie équilibrée, mais le comportement local est incertain).
- L'influence sur W_2 (exponentiel en b).
- L'influence sur l'espace mémoire nécessaire pour la phase 2.

Le choix de d

Ce paramètre conditionne le degré maximal de A et B. Nous avons vu qu'asymptotiquement[7], $d \approx b$. La *taille* de l'espace de crible (i.e. l'ensemble des paires (A, B)) dépend donc directement de d, et il convient de s'assurer que cet espace de crible est suffisamment grand pour obtenir le nombre voulu de relations, étant donnée la probabilité de friabilité à laquelle on doit s'attendre[8]. Hélas, lorsque l'on augmente d, les degrés de C et D augmentent aussi, ce qui a pour effet de réduire leur probabilité de friabilité.

À titre de remarque, il est assez aisé de constater que les degrés de C et D sont inchangés si l'on donne un valeur un peu plus grande au degré maximum de B par rapport à celui de A – on découple ainsi d en deux paramètres distincts d_A et d_B. Un calcul aisé montre que la valeur optimale de la différence entre ces deux paramètres est :

$$d_B - d_A = \frac{hk - n + \deg f_1}{k}.$$

Le choix de k

Le paramètre k est contraint à être une puissance de 2. La valeur employée expérimentalement peut donc être relativement éloignée de la meilleure valeur asymptotique (qui est

[7] Cette approximation est valable si l'on considère que l'algèbre linéaire est de complexité quadratique. Dans [Cop84], Coppersmith montre que $\frac{d}{b} \approx \frac{\omega}{2}$, où ω est l'exposant de la complexité de l'algèbre linéaire (au plus 3 avec le pivot de Gauss).

[8] En réalité, « viser juste » en ce domaine est pratiquement impossible si l'on prend en compte les différentes techniques introduites au chapitre 3.

$\sqrt{\frac{n}{d_A}} = \left(\frac{n}{\log n}\right)^{\frac{1}{3}}$). L'influence néfaste d'une différence importante entre k et sa valeur optimale se situe bien entendu au niveau de l'*équilibre* entre les degrés de C et D. Idéalement, on a choisi dans l'analyse k de telle sorte que ces degrés soient équilibrés. S'ils ne le sont pas, c'est la probabilité de friabilité qui en souffre.

Lorsque les degrés de C et D sont non équilibrés, la probabilité de friabilité la plus faible est bien sûr celle du plus gros des polynômes. Il convient donc de tester ce polynôme en premier, pour ne tester le second polynôme que sur un faible échantillon de l'espace de crible.

Un autre aspect relatif au choix de k est son influence sur le système linéaire qui intervient dans la phase 2 de l'algorithme. Ses coefficients, de la forme $f_i - ke_i$, sont statistiquement plus gros si k augmente. Cet aspect peut se révéler encombrant, comme on le verra en 5.2.

Il est tout de même important de noter que « $k = 4$ ». En effet, pour la classe des problèmes qui nous intéressent, c'est-à-dire les problèmes non triviaux (où l'emploi d'une méthode d'*index-calculus* est pertinente) mais ou le calcul n'est pas complètement hors de portée, 4 est la valeur optimale[9]. Pour le calcul effectué dans $\mathbb{F}_{2^{607}}$, qui constitue le record du monde actuel, quelques mesures ont été effectuées avec $k = 8$, mais le résultat n'était pas satisfaisant.

Le choix du polynôme de définition

Un dernier paramètre se cache dans le choix du polynôme f_1. L'algorithme a une sensibilité évidente vis-à-vis de $\deg f_1$, puisque le degré de D vaut $kd_A + hk - n + \deg f_1$. Économiser un coefficient sur le polynôme D n'est pas négligeable, donc l'intérêt d'avoir $\deg f_1$ petit est réel. Cela laisse peu de choix possible. Néanmoins, si l'on se restreint à une poignée de petits polynômes f_1 possibles, quelle heuristique doit-on employer pour choisir f_1 ? Dans [GM93], il est démontré que la présence de facteurs *répétés* de *petit degré* dans la factorisation de f_1 a une influence importante. Nous reprenons ici la démonstration de ce résultat, en l'affinant. Cette discussion est en partie reprise de [GM93].

Le polynôme D s'écrit :

$$D(X) = A(X)^k X^{hk-n} f_1(X) + B(X)^k.$$

Pour simplifier l'écriture des formules, nous allons temporairement noter $R = X^{hk-n} f_1$ et $U(X) = \frac{B(X)}{A(X)}$. Nous nous intéressons à la présence éventuellement multiple d'un facteur irréductible w dans D. Notons tout d'abord que si la fraction rationnelle $U(X)$ a un pôle en une racine de w, alors $w \mid A$, donc $w \nmid B$, donc $w \nmid D$. On peut donc écarter cette situation. Pour que le polynôme w^e divise D (l'exposant e étant $\geqslant 1$), on doit avoir :

$$\begin{aligned} A^k X^{hk-n} f_1 + B^k &\equiv 0 \bmod w^e, \\ \left(\frac{B}{A}\right)^k &\equiv R \bmod w^e, \\ U &= \sqrt[k]{R} \bmod w^e. \end{aligned}$$

Le problème auquel on doit faire face est donc celui de l'existence de racines k-èmes modulo une puissance d'un polynôme irréductible. Si $e = 1$, ce problème n'en est pas un, car de telles

[9] à un tel point que l'auteur de ce manuscrit ne garantit pas qu'une coquille n'ait pas parfois placé 4 en lieu et place de k dans ces pages.

racines existent toujours. Dès que e grandit, la situation se complique. On aimerait trouver les racines k-èmes qui nous intéressent par un procédé de relèvement 2-adique, mais hélas la structure mathématique du problème rend un tel relèvement impossible (on est dans un contexte *ramifié*, car k est une puissance de 2). Considérons l'application :

$$\pi : \left\{ \begin{array}{ccc} \mathbb{F}_2[X]/w^e & \longrightarrow & \mathbb{F}_2[X]/w^e \\ P \bmod w^e & \longmapsto & P^k \bmod w^e \end{array} \right.$$

Il est facile de voir que si $e \leqslant k$, cette application n'est pas injective, et les antécédents d'une valeur P^k sont tous les $P + w\rho$, pour $\deg \rho < (e-1)\deg w$. Deux remarques s'ensuivent :

- Toutes les valeurs ne sont pas atteintes. Il se peut que R n'ait aucune racine k-ème modulo w^e pour $1 < e \leqslant k$. C'est le cas par exemple si n est impair et $\deg R < 2\deg w$, car on est alors certain que $R \bmod w^e$ a un coefficient de degré impair non nul (rappelons que $R = X^{hk-n} f_1$). Il s'ensuit que le polynôme D ne peut pas admettre un facteur répété de degré $\geqslant \frac{\deg R}{2}$. Cela a tendance à empêcher D d'être friable.
- S'il se trouve que R admet une racine k-ème modulo w^e (en supposant donc que w est de petit degré, en vertu de ce qui précède), alors de façon automatique on a $w^e \mid D$ dès que $w \mid D$. Cette dernière remarque se synthétise donc en la proposition qui suit.

Proposition – Heuristique 2.7. *La probabilité de friabilité de D est augmentée si le polynôme $R = X^{hk-n} f_1$ admet souvent une racine k-ème modulo une puissance d'un facteur irréductible.*

Il est important de remarquer que le critère mentionné ici est plus fin que celui cité dans [GM93] (qui mentionne seulement $w^e \mid f_1$). Ces deux critères sont bien sûr proches, mais pas équivalents. Il est aisé d'évaluer dans quelle mesure un polynôme f_1 répond à ce critère ou non. On peut donner la liste des couples (w, e) de petits polynômes w tels que R admet une racine k-ème modulo w^e, avec $e > 1$. Plus cette liste est grande, meilleur est le polynôme f_1.

Déduire exactement le gain que représente le choix d'un polynôme f_1 particulier sur la probabilité de friabilité du polynôme D n'est toutefois pas immédiat, car nous nous sommes concentrés ici sur la probabilité de divisibilité par un facteur w précis. Ces probabilités pour l'ensemble des facteurs w n'étant pas du tout indépendantes, il n'est pas évident d'adopter un point de vue transverse.

2.5 Le crible de corps de fonctions (FFS)

Nous ne détaillons pas dans ce mémoire les algorithmes permettant de calculer des logarithmes discrets dans les corps premiers $\mathbb{F}_p$. Notons toutefois que parmi ces algorithmes, on compte le crible algébrique [Gor93] (qui a été adapté de la méthode de factorisation du même nom [LL93, LLMP93]). En considérant un corps de nombres K choisi de telle sorte qu'il existe une surjection $K \to \mathbb{F}_p$, on peut en effet obtenir un algorithme de calcul de logarithmes discrets de complexité $L_{\log p}\left(\frac{1}{3}, c\right)$.

Diverses variantes ou cas particuliers de cette approche existent, comme par exemple l'algorithme de Coppersmith-Odlyzko-Schroeppel [COS86]. Il est possible de les voir sous un angle unificateur [SWD96]. Ce regard unificateur inclut aussi l'algorithme de Coppersmith pour $\mathbb{F}_{2^n}$, qui apparaît comme un cas particulier de l'algorithme du crible de corps de fonctions, ou *Function Field Sieve* (FFS), proposé en 1994 par Adleman [Adl94]. Nous allons décrire

brièvement l'algorithme FFS. On va voir que le formalisme est proche de celui du crible algébrique.

Un avantage du FFS est son caractère général. Il permet de lever de nombreuses restrictions liées à l'algorithme de Coppersmith. Notamment, il s'applique au cas de la caractéristique différente de 2. De plus, on dispose avec le FFS d'une souplesse beaucoup plus importante dans le choix des paramètres, notamment en ce qui concerne le paramètre jouant le rôle du paramètre noté k dans l'algorithme de Coppersmith. La présentation suivante du FFS est inspirée de [AH99, JL02]. Dans les notations que nous prenons, nous donnons le même nom aux variables qui intervenaient déjà dans l'algorithme de Coppersmith, puisqu'elles jouent un rôle similaire.

Supposons que nous avons choisi une représentation de $\mathbb{F}_{2^n}$ sous la forme $\mathbb{F}_2[X]/(f(X))$. Il nous faut choisir deux polynômes à deux variables $H(X,Y)$ et $G(X,Y)$ ayant une racine commune dans $\mathbb{F}_{2^n}$, le polynôme G étant de degré 1 en Y. Comme le crible algébrique, le FFS est très sensible à la taille des *coefficients* des polynômes G et H. On s'attache donc à construire de petits polynômes G et H.

Avant même de fixer le polynôme de définition f, la construction proposée par [JL02] consiste à choisir d'abord le polynôme $H(X,Y)$. Cette construction est motivée par le fait que pour la performance de l'algorithme, il est encore plus important de contrôler H que f.

Soit k un entier. Nous donnerons sa valeur asymptotique plus loin (ce paramètre joue le même rôle que dans l'algorithme de Coppersmith, mais ici il n'est pas contraint à être une puissance de 2). Soit $H(X,Y)$ un polynôme de degré k en Y, qu'on écrit :

$$H(X,Y) = h_k(X)Y^k + \cdots + h_1(X)Y + h_0(X).$$

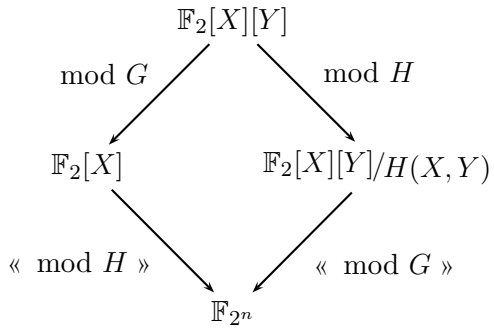
Les coefficients $h_i(X)$ sont choisis de degré $(n \bmod k)$. Soient $\mu_1(X)$ et $\mu_2(X)$ deux polynômes de degré $\lfloor \frac{n}{k} \rfloor$ en Y. On prend $G(X,Y) = \mu_2(X)Y - \mu_1(X)$. Constituons le résultant suivant :

$$\begin{aligned} Res_Y(G,H) &= \mu_2(X)^k H\left(X, \frac{\mu_1(X)}{\mu_2(X)}\right), \\ &= h_k(X)\mu_1(X)^k + \cdots + h_1(X)\mu_1(X)\mu_2(X)^{k-1} + h_0(X)\mu_2(X)^k. \end{aligned}$$

Si notre choix de μ_1 et μ_2 est bon, alors ce résultant est un polynôme de degré n irréductible. Le fait que le degré soit égal à n est très probable, grâce au choix fait pour le degré des h_i. Pour l'irréductibilité, la probabilité est de l'ordre de $\frac{1}{n}$. On répète le choix de μ_1 et μ_2 jusqu'à obtenir cette condition, et on pose alors $f = \mathrm{Res}_Y(G,H)$. Notons que pour obtenir une plus grande souplesse sur le choix des coefficients de H, on peut choisir μ_2 de degré bien inférieur à $\lfloor \frac{n}{k} \rfloor$ (voire $\mu_2 = 1$). De cette façon, la contrainte $\deg h_i = (n \bmod k)$ peut être relaxée, puisqu'en prenant simplement $\deg h_k = (n \bmod k)$ et $\deg h_i \leqslant (k-1)$, on obtient le bon degré.

La construction analogue pour l'algorithme de Coppersmith consiste à poser $G = Y - X^h$, et $H = Y^k - X^{hk-n} f_1(X)$ (le résultant est alors égal à $X^{hk-n} f(X)$, ce qui revient au même).

Une fois que les polynômes G et H ont été ainsi formés, on a alors deux polynômes ayant une racine commune modulo $f(X)$, à savoir $\frac{\mu_1(X)}{\mu_2(X)}$. On a donc le diagramme commutatif suivant :



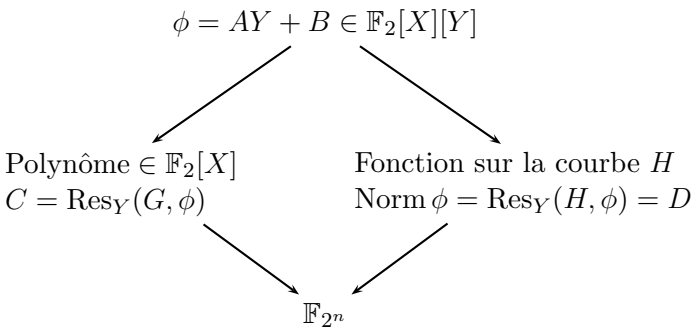
Ce diagramme mérite quelques éclaircissements, car son caractère allusif cache plusieurs points techniques. Nous donnons d'abord ces éclaircissements, avant de décrire le procédé de calcul d'index en plus grand détail.

Soit ϕ un élément de $\mathbb{F}_2[X][Y]$, de degré 1 en Y. Lorsqu'on considère ϕ « mod G » ou « mod H », on considère en fait ϕ comme un polynôme en une variable Y sur le corps des fractions rationnelles en X, noté $\mathbb{F}_2(X)$.

Du côté gauche, côté « rationnel », la quantité $\phi \bmod G$ ainsi construite est un polynôme en X. Considérer cette quantité « mod H » n'a du sens que si l'on considère en fait la réduction modulo le polynôme $H\left(X, \frac{\mu_1(X)}{\mu_2(X)}\right)$, qui est l'image du côté rationnel du polynôme H.

Du côté droit, côté « algébrique », la première réduction mod H doit encore être opérée sur le polynôme ϕ en tant qu'élément de $\mathbb{F}_2(X)[Y]$. Cette opération ne se traduit par aucun calcul, puisque ϕ est de degré 1. On obtient ainsi une *fonction* ϕ, appartenant au corps $\mathbb{F}_2(H)$ des fonctions sur la courbe d'équation $H(X,Y) = 0$ sur $\mathbb{F}_2$. À nouveau, lorsqu'on opère la seconde réduction « mod G » de cette fonction, il faut pour lui donner du sens considérer G différemment : « $\phi \bmod G$ » est plus exactement l'évaluation de ϕ sur la place de H correspondant aux zéros de G. De cette façon, le résultat est bien défini comme un élément de $\mathbb{F}_{2^n}$.

On peut redonner un diagramme, explicitant ainsi les deux chemins construits, allant de $\mathbb{F}_2[X][Y]$ à $\mathbb{F}_{2^n}$.



Le choix de la base de facteurs dans l'algorithme FFS s'obtient en considérant les deux côtés de la figure, rationnel et algébrique. On a ainsi *deux* bases de facteurs. À gauche, du coté *rationnel*, on choisit tous les polynômes irréductibles de degré inférieur à une borne b. On les note π_i. À droite, du côté algébrique, on choisit tous les idéaux premiers de degré 1 de l'ordre maximal de $K(H) = \mathbb{F}_2(X)(Y)/H(X,Y)$ dont la *norme* est de degré inférieur à la même borne b. On les note $\mathfrak{p}_i$.

On considère ensuite de nombreuses expressions de la forme $\phi = A(X)Y + B(X)$, où A et B sont soumis aux contraintes $\deg A \leqslant d_A$ et $\deg B \leqslant d_B$. On recherche la friabilité à la fois

du côté rationnel et du côté algébrique. Du côté rationnel, on souhaite donc avoir la friabilité du polynôme

$$C(X) = A(X)\mu_1(X) + B(X)\mu_2(X),$$

et du côté algébrique, on veut que la norme de la fonction $A(X)Y + B(X)$ soit friable. Cette norme s'écrit :

$$D(X) = \mathrm{Res}_Y(H(X,Y), A(X)Y + B(X)).$$

Bien que cela ne soit pas apparent, si l'on prend pour H l'expression correspondant à l'algorithme de Coppersmith, alors $C(X)$ et $D(X)$ correspondent bel et bien aux polynômes du même nom introduits dans l'algorithme de Coppersmith.

Une paire (A, B) telle que C et D sont friables correspond « presque » à une relation du type :

$$\prod \pi_i^{\gamma e_i} \equiv \Psi(\prod \mathfrak{p}_j^{f_j}) \mod f,$$

pour une certaine valeur de γ. En réalité, cette relation doit être examinée plus en profondeur. En particulier, elle n'est pas vérifiée si la courbe d'équation H possède plusieurs *places à l'infini*. Cette obstruction peut être levée en utilisant pour H une courbe C_{ab}, comme cela a été remarqué par Matsumoto [Mat99]. Deuxièmement, si l'on travaille non plus en caractéristique 2 mais en caractéristique p, alors des unités peuvent s'introduire, donc la relation n'est vérifiée que modulo $\mathbb{F}_p^*$. En dehors de ces restrictions, la relation énoncée est vérifiée, en prenant pour γ le nombre de classes de l'anneau $\mathbb{F}_2(X)[Y]/H(X,Y)$, c'est-à-dire le nombre de points $\mathbb{F}_2$-rationnels de la courbe H (si celle-ci est de genre 1). Ce nombre est petit. Une exposition minutieuse de la justification de cette construction algébrique se trouve dans [AH99].

Nous n'effectuons pas l'analyse du FFS. Elle est semblable à celle de l'algorithme de Coppersmith, et les valeurs asymptotiques des paramètres sont les mêmes. La seule différence se situe au niveau du paramètre k : celui-ci n'est pas contraint à être une puissance de 2, ce qui signifie que l'on est toujours dans le cas le meilleur pour l'algorithme de Coppersmith. Du point de vue pratique, malgré l'introduction de considérations algébriques abstraites, tous les calculs peuvent se ramener à des calculs sur les polynômes.

Nous pouvons remarquer que le critère que nous avons développé page 35 pour le choix du polynôme de définition dans l'algorithme de Coppersmith peut s'exprimer simplement si l'on regarde l'algorithme comme un cas particulier du FFS. Nous avons dit que pour le cas de l'algorithme de Coppersmith, on prenait $H = Y^k - X^{hk-n} f_1(X)$, c'est-à-dire $Y^k - R(X)$. Nous avons énoncé qu'il était souhaitable que f soit choisi de telle sorte que R admette souvent des racines k-èmes modulo des puissances de facteurs irréductibles. Ce critère est peu ou prou identique à celui que l'on utilise pour le crible algébrique, où on choisit le polynôme de définition de telle sorte que ses racines soient nombreuses modulo de petits nombres premiers [EH96, EH97].

Chapitre 3

Techniques pour l'algorithme de Coppersmith

Nous développons dans ce chapitre les différentes techniques qui peuvent être mises en œuvre pour l'implantation de l'algorithme de Coppersmith. Nous portons l'accent sur notre record de calcul de logarithmes discrets dans le corps $\mathbb{F}_{2^{607}}$, à la lumière duquel les apports des différentes techniques que nous avons employées sont illustrés.

Les techniques décrites ici sont multiples. Nous décrivons surtout des techniques ayant trait à la recherche de relations, donc la première phase de l'algorithme, et nous donnons quelques détails sur la troisième phase en 3.8. La seconde phase de l'algorithme est un calcul d'algèbre linéaire, traité dans la partie II de ce mémoire.

On peut mentionner qu'aucune des techniques développées dans ce chapitre n'a une influence fondamentale sur la complexité sous-exponentielle de l'algorithme. En revanche, certaines peuvent avoir une influence sur la composante invisible (cachée dans le $o(1)$ de l'expression de la complexité), mais néanmoins polynomiale. Souvent ce sont aussi des considérations pratiques qui apportent des arguments en faveur d'une technique.

3.1 L'emploi de *large primes*

3.1.1 *(Single) large prime variation*

Une des premières améliorations apportées aux algorithmes d'*index-calculus* et aux techniques de combinaisons de relations en général consiste à employer des *large primes*. L'idée sous-jacente est relativement ancienne et remonte aux premiers temps de la factorisation des entiers par l'algorithme CFRAC [MB75]. Elle a été réutilisée avec succès dans le crible quadratique et le crible algébrique [MB75, Pom82, LM91, DDLM94, LM94, BR96, DL95]. De même, cette méthode se transporte sans difficulté au cadre des algorithmes de calcul de logarithme discret comme l'algorithme de Coppersmith. On peut comparer à cette méthode l'emploi du crible par réseau qui est décrit en 3.6.

Par *large prime* on désigne un cofacteur apparaissant dans une relation « presque friable ». Supposons par exemple que dans l'algorithme de Coppersmith les factorisations des polynômes C et D s'écrivent sous la forme :

$$C = Q \prod_i \pi_i^{e_i},$$
$$D = \prod_i \pi_i^{f_i},$$

où Q est un facteur irréductible tel que $b < \deg Q \leqslant 2b + 1$. Le principe de la *large prime variation* est d'essayer d'utiliser ces relations « partielles » plutôt que de les considérer inutiles.

L'intérêt de la méthode présente deux facettes. D'abord, pratiquement aucun surcoût n'est imposé par la considération des relations partielles. En effet, sans rentrer dans le détail de la façon dont les paires (A, B) sont analysées (c'est l'objet du reste de ce chapitre), on peut partir de l'idée que les facteurs de degré $\leqslant b$ sont extraits un par un, ou degré par degré. Avant de pouvoir dire si une relation est friable ou non, on fait le travail qui consiste à calculer exactement le cofacteur sans facteur de degré $\leqslant b$ (ou éventuellement le degré de ce cofacteur). Un tel cofacteur, si son degré est dans $[\![b+1 \dots 2b+1]\!]$, est inévitablement irréductible. On peut donc considérer que le traitement des relations partielles n'occasionne pas de surcoût. Pour préciser cette assertion, on indiquera pour les différentes techniques de recherche de relations exposées dans ce chapitre comment détecter un possible cofacteur de degré borné par $\mathcal{L}$ (*large prime bound*), où $\mathcal{L} \leqslant 2b+1$.

Par ailleurs, les relations partielles sont rendues utiles par recombinaison. Deux relations partielles faisant intervenir le même cofacteur Q peuvent être recombinées pour obtenir une « vraie » relation[1]. L'aspect intéressant est que le nombre de recombinaisons possible croît quadratiquement en le nombre de relations partielles disponibles. C'est un résultat qui s'apparente au « paradoxe des anniversaires », que l'on reprend ici.

Proposition 3.1. *Si l'on autorise des* large primes *appartenant à un ensemble de cardinal m, l'espérance du nombre de recombinaisons possibles à partir de n relations partielles est $\frac{n^2}{2m}$ si les différents* large primes *ont une probabilité d'apparition uniforme. Si leurs probabilités respectives sont $p_1, \dots, p_m$, l'espérance vaut $n^2 \frac{S_2}{2}$, où $S_2 \stackrel{déf}{=} \sum_i p_i^2$.*

Démonstration. On reprend la démonstration de [Mor93]. Cette démonstration emploie des techniques de séries génératrices comme décrites dans [FS93, FS02] ou encore [FO90]. Notons $p_1, \dots, p_m$ les probabilités respectives d'apparition des différents *large primes* (pour fixer les idées, on peut prendre les p_i égaux, mais cela n'est pas nécessaire). La somme des p_i vaut 1. On écrit la série génératrice exponentielle en z et u, où l'exposant de z est le nombre de relations partielles, et l'exposant de u le nombre de *large primes* distincts apparaissant dans ces relations. Dans ce contexte, le coefficient du monôme en $z^n u^k$ vaut $\frac{1}{n!}$ fois la probabilité d'avoir k *large primes* distincts à partir de n relations. Cette série s'écrit :

$$\begin{aligned}\Phi(u,z) &= \prod_{i=1}^{m} \left(1 + up_i z + u\frac{p_i^2 z^2}{2!} + u\frac{p_i^3 z^3}{3!} + \dots\right), \\ &= \prod_{i=1}^{m} \left(1 + u(e^{p_i z} - 1)\right).\end{aligned}$$

L'espérance du nombre de *large primes* distincts apparaissant parmi les n premières relations partielles s'obtient donc en dérivant par rapport à u :

$$n![z^n] \left.\frac{\partial}{\partial u}\Phi(z,u)\right|_{u=1}$$

Le nombre de recombinaisons possibles M_n à partir de ces n relations partielles est alors donné

[1] Cette relation recombinée est en revanche deux fois plus « lourde » qu'une relation classique. Nous reviendrons sur l'influence de ceci sur la phase d'algèbre linéaire. Cet aspect peut être visualisé par la figure 5.1, en page 89.

par la relation :

$$\begin{aligned}
M_n &= n - n![z^n]\,\frac{\partial}{\partial u}\Phi(z,u)\bigg|_{u=1}, \\
&= n - n![z^n]\sum_{i=1}^{m}\left(\prod_{j\neq i}(1+(e^{p_j z}-1)\right)(e^{p_i z}-1), \\
&= n - n![z^n]\sum_{i=1}^{m} e^{(1-p_i)z}\,(e^{p_i z}-1), \\
&= n - n!\sum_{i=1}^{m}[z^n]\left(e^z - e^{(1-p_i)z}\right), \\
&= n - \sum_{i=1}^{m}\left(1-(1-p_i)^n\right).
\end{aligned}$$

Pour obtenir l'estimation de M_n, on développe les premiers termes de l'expression $(1-p_i)^n$. On obtient ainsi :

$$\begin{aligned}
M_n &= n - \sum_{i=1}^{m}\left(1 - 1 + np_i - \frac{1}{2}(np_i)^2\right) + \epsilon, \\
M_n &= n^2\frac{S_2}{2} + \epsilon.
\end{aligned}$$

Dans cette expression, le terme correctif ϵ reste petit tant que n est petit devant m. La vérification est pénible, mais on peut montrer que dans le cas uniforme, si $n \leqslant \frac{m^{\frac{2}{3}}}{2}$, alors l'erreur est bornée par 1. Cette plage d'approximation couvre largement le domaine d'utilisation visé. Si la distribution de probabilité est plus déséquilibrée, l'espérance M_n est supérieure, mais la précision est moins bonne (et la vérification de ce fait est fastidieuse). ■

3.1.2 *Double large prime variation*

Une extension possible de la méthode précédemment décrite est la *double large prime variation*. Celle-ci autorise l'apparition d'un plus grand nombre de cofacteurs. Par exemple, on accepte d'avoir un cofacteur dans la factorisation de C *et* un autre dans la factorisation de D. On peut aussi autoriser deux cofacteurs à être présents « du même côté », bien que cela puisse entrer en conflit avec le souci, qui doit être permanent, de ne pas payer de surcoût pour l'obtention de ces relations partielles (des cas peuvent se présenter où deux cofacteurs sont identifiables facilement). Nous décrivons ici la façon dont les relations sont recombinées. Tout d'abord, précisons la terminologie employée.

- Par relation ff (de *full-full*), on désigne les relations sans aucun *large prime*. Ces relations sont rares, et on essaye d'augmenter leur nombre grâce à des recombinaisons de relations partielles.
- Par relation pf, ou fp (de *partial-full* et *full-partial*), on désigne les relations où apparaissent *un seul large prime*. La distinction entre pf et fp tient à la place du *large prime*, suivant qu'il apparaît dans la factorisation de C ou de D. Dans le cas de l'algorithme de Coppersmith, cette distinction n'a pas cours, car on peut « mélanger » sans difficulté les *large primes* « mixtes ».

– Par relation pp (de *partial-partial*) on désigne les relations ayant deux cofacteurs (*a priori* les deux cofacteurs apparaissent chacun d'un côté, mais on accepte aussi de traiter les cas où deux cofacteurs sont présents d'un côté, et aucun de l'autre côté).

On peut voir les différentes relations partielles produites au cours de l'algorithme comme des arêtes dans un graphe. Les sommets du graphe sont tous les *large primes* possibles, plus le sommet spécial noté 1 auquel sont reliés les *large primes* intervenant dans des relations pf ou fp. La correspondance est la suivante.

$$\prod_i \pi_i^{e_i} \equiv \left(\prod_i \pi_i^{f_i}\right)^k \longrightarrow \text{pas d'arête (relation ff)},$$

$$Q\prod_i \pi_i^{e_i} \equiv \left(\prod_i \pi_i^{f_i}\right)^k \longrightarrow \text{arête 1–}Q\text{ (relation pf)},$$

$$\prod_i \pi_i^{e_i} \equiv \left(Q\prod_i \pi_i^{f_i}\right)^k \longrightarrow \text{arête 1–}Q\text{ (relation fp)},$$

$$Q_1\prod_i \pi_i^{e_i} \equiv \left(Q_2\prod_i \pi_i^{f_i}\right)^k \longrightarrow \text{arête } Q_1\text{–}Q_2\text{ (relation pp)}.$$

Il est possible de recombiner plusieurs relations pp pour produire une relation ff, il faut pour cela que les arêtes correspondantes forment un *cycle*. Toutefois, cette condition n'est pas suffisante car des multiplicités interviennent dans les relations. Considérons un cycle en toute généralité : soient k relations $R_1, \ldots, R_k$ faisant intervenir k *large primes* $Q_1, \ldots, Q_k$, où dans R_i interviennent les *large primes* Q_i et Q_{i+1} (Q_k et Q_1 pour R_k) avec les multiplicités respectives m_i et m'_i. La relation $\prod_{i=1}^k R_i^{\alpha_i}$ fait intervenir les *large primes* avec les multiplicités suivantes :

$$\begin{array}{ll} Q_1 & \text{multiplicité } \beta_1 = \alpha_1 m_1 + \alpha_k m'_k, \\ Q_2 & \text{multiplicité } \beta_2 = \alpha_2 m_2 + \alpha_1 m'_1, \\ Q_3 & \text{multiplicité } \beta_3 = \alpha_3 m_3 + \alpha_2 m'_2, \\ \ldots & \ldots \\ Q_k & \text{multiplicité } \beta_k = \alpha_k m_k + \alpha_{k-1} m'_{k-1}. \end{array}$$

On a donc une expression aisée du vecteur $(\beta_1, \ldots, \beta_k)$:

$$(\beta_1, \ldots, \beta_k) = (\alpha_1, \ldots, \alpha_k) \times \begin{pmatrix} m_1 & m'_1 & & \\ & \ddots & \ddots & \\ & & \ddots & m'_{k-1} \\ m'_k & & & m_k \end{pmatrix}$$

Si aucun des *large primes* Q_i n'est le « faux » *large prime* 1, alors tous les exposants doivent être non nuls pour que la relation soit utilisable. Cela n'est possible que si la matrice qui apparaît dans le membre de droite est singulière. Son déterminant vaut $\prod_i m_i - (-1)^k \prod_i m'_i$. Comme nous voulons que cette quantité s'annule dans $\mathbb{Z}$, on voit que la probabilité est faible compte tenu du fait que dans l'algorithme de Coppersmith, les exposants m_i et m'_i valent généralement 1 et $-k$ (sans ordre)[2]. Un tel cycle « générique » est en revanche possible à

[2]Il est toutefois possible, si la matrice n'est pas singulière, de créer à partir de cette situation une arête 1–Q_1.

gérer dans le cas des algorithmes de factorisation, où les exposants ne sont considérés que modulo 2. Le produit des relations R_i est alors une relation ff à cet égard.

Fort heureusement, l'aspect négatif de la constatation que nous venons de faire est compensé par le fait que de tels cycles « génériques » n'apparaissent jamais. En effet, le graphe considéré a un sommet spécial, le sommet 1, qui est beaucoup plus « touffu » que les autres sommets, puisque de très nombreux *large primes* y sont rattachés : toutes les relations pf ou fp correspondent à une arête du type 1–Q, et ces relations sont nombreuses[3]. Cette donnée fait que les cycles ont une probabilité extraordinairement plus importante de faire intervenir le sommet 1 que de l'omettre[4]. On voit donc qu'à l'inverse de ce qui se passait pour la *single large prime variation*, la considération d'un modèle uniforme ne suffit pas à mettre en évidence le bon comportement de ce modèle à deux *large primes*. Dans [FKP89], le cas d'un graphe aléatoire où les arêtes ont des probabilité d'apparition égales est examiné sous presque tous les aspects possibles. Parmi les résultats obtenus, on apprend que le premier cycle dans le graphe apparaît à partir de $n/3$ relations. Les données expérimentales exposées plus loin montrent que les phénomènes observés nous sont de plusieurs ordres de grandeur plus favorables que cela. Une évaluation du nombre de cycles que l'on peut attendre à partir de n relations n'a jamais été menée exactement. Il est généralement conjecturé que cette progression est au moins cubique, mais il s'agit essentiellement d'une extrapolation expérimentale.

Pour détecter les cycles dans le graphe constitué par les relations partielles, on a employé l'algorithme classique *union-find* [Sed88]. Le graphe (où les cycles ne sont justement pas stockés) est vu comme une collection d'arbres (c'est-à-dire une forêt). On oriente arbitrairement les arêtes lorsqu'elles sont insérées dans le graphe, de telle sorte que chaque sommet a au plus un parent, et exactement un ancêtre (éventuellement lui-même), se caractérisant par la propriété de ne pas avoir de parent. Les différents sommets d'une même composante connexe du graphe ont ainsi le même ancêtre.

Lors de l'insertion d'une arête Q_1–Q_2, quatre cas peuvent se présenter.

- Q_1 et Q_2 sont des sommets de degré 0 dans le graphe : aucune relation ne les fait intervenir. Dans ce cas, on choisit arbitrairement Q_1 ou Q_2 comme étant le parent de l'autre. Ainsi, on ajoute au graphe deux sommets, une arête, une composante connexe (dont Q_1 est l'ancêtre, par exemple), et pas de cycle.
- Exactement l'un des sommets (Q_2 par exemple) est de degré 0. On lui attribue l'autre (Q_1) comme parent. Ce faisant, on ajoute au graphe un sommet et une arête.
- Q_1 et Q_2 sont des sommets de degré non nul, mais leurs ancêtres sont distincts. Ils appartiennent donc à deux composantes connexes distinctes, que l'on va relier. Si la composante connexe comprenant Q_1 est la plus lourde des deux (en terme de nombre de sommets), alors Q_1 devient le nouveau parent de Q_2. Cela implique de renverser le sens des arêtes menant de Q_2 à son ancêtre. Ainsi l'ancien parent Q_2 a désormais Q_1 comme parent, et caetera. La figure 3.1 illustre cette situation. Dans le cas symétrique où c'est la composante connexe comprenant Q_2 qui est la plus grosse, on effectue l'opération inverse[5]. Dans cette situation, on n'ajoute pas de sommet au graphe, on rajoute une

[3] 10% des relations pour le calcul effectué sur $\mathbb{F}_{2^{607}}$.

[4] À ce sujet, les données expérimentales du calcul sur $\mathbb{F}_{2^{607}}$ citées plus loin sont éloquentes : plus de 800 000 cycles obtenus, pas un seul ne comportant pas le sommet 1.

[5] Ce choix est dirigé par le souci de maintenir la profondeur moyenne des arbres la plus petite possible. On pourrait apporter plus de soin à l'analyse : si pour $i = 1, 2$ l'arbre où est situé Q_i a n_i sommets à une profondeur moyenne h_i, et que la profondeur de Q_i est p_i, alors la quantité à minimiser est la profondeur moyenne résultante, à savoir $\frac{1}{n_1+n_2}\left(n_1h_1 + n_2h_2 + (p_1 + p_2 + 1)n_i - p_i - h_i\right)$ si Q_i est choisi comme fils de

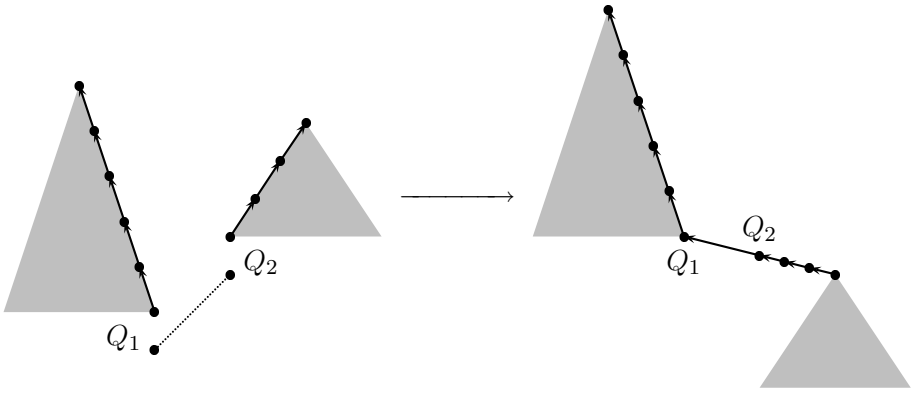


Figure 3.1 – Fusion de deux composantes

arête, et on retire une composante connexe.
- Q_1 et Q_2 ont le même ancêtre. Alors on a détecté un cycle. La stratégie employée consiste à ne rien faire et stocker à part la relation correspondant à l'arête Q_1–Q_2, dans le but de traiter ensemble toutes les relations qui engendrent des cycles.

Nous devons remarquer que dans le cas de la fusion de composantes, on diffère du *union-find* « standard », comme décrit par exemple par [Sed88]. L'opération menée est représentée par la figure 3.1. La raison de cette variation est que l'on tient à conserver une correspondance simple entre arêtes du graphe et relations. Décréter ici que l'ancêtre de Q_2 a pour parent Q_1, ou bien l'ancêtre de Q_1, nécessiterait trop de calculs. En particulier, cela nécessiterait la lecture de toutes les relations allant de Q_2 à son ancêtre, ce qui induirait une pénalité trop forte en termes d'entrées-sorties. La contrepartie du choix que nous faisons est que la profondeur moyenne des sommets dans les arbres associés aux composantes connexes est plus grande qu'avec l'algorithme « standard ». Dans la pratique, cet aspect n'a toutefois pas eu d'influence notable.

Au fur et à mesure de l'insertion des arêtes dans le graphe, on garde donc le compte exact du nombre de cycles que l'on est en mesure de créer. Lorsque l'objectif est atteint, on traite alors une par une les relations engendrant des cycles. Cela nécessite de prendre en considération toutes les relations correspondant aux arêtes des cycles concernés. En pratique, on a toujours le *large prime* 1 comme ancêtre commun des sommets considérés, donc l'expression d'une relation ff ne pose pas de problème.

Comme on l'a mentionné au fur et à mesure de la description des différentes actions possibles sur le graphe, il est aisé de maintenir une trace de plusieurs propriétés du graphe, notamment son nombre de composantes connexes, son nombre d'arêtes, de sommets, et par conséquent de cycles. Avec à peine plus de soin, on peut aussi garder la trace du poids de la composante connexe la plus grosse (car on sait dans notre cas de quelle composante connexe il s'agit), ainsi qu'une majoration de la taille de la seconde plus grosse. Diverses données de ce type sont rapportées concernant le calcul de logarithmes discrets sur $\mathbb{F}_{2^{607}}$.

l'autre. On voit que c'est n_i qui importe le plus.

3.1.3 Considérations d'implantation

Table de hachage

La mise en pratique des méthodes de *large prime variation* décrites ici nécessite l'emploi d'une structure adaptée. En effet, le nombre de relations à stocker pour obtenir un grand nombre de cycles est potentiellement grand. Pour des tailles de problèmes importantes, on peut sans hésitation estimer ce nombre à plusieurs millions, voire dizaines de millions. Pour le calcul de logarithme discret sur $\mathbb{F}_{2^{607}}$ [Tho01b, Tho02a], le nombre de relations partielles a atteint 61 058 174, impliquant 87 073 945 *large primes* distincts. Nous décrivons ici quelques détails de notre implantation avec deux *large primes*. Une contrainte de cette implantation a été de gérer la recherche de cycles en utilisant au plus 1Go de mémoire vive.

Au fur et à mesure que les relations sont collectées, l'algorithme *union-find* demande de conserver de nombreuses traces des données. De toute évidence, une trace des *large primes* rencontrés est nécessaire, ainsi que les descriptions des arêtes correspondantes. À raison d'environ 5 octets nécessaires pour identifier un *large prime*, cela reste traitable. En revanche, conserver en mémoire toutes les relations où ils interviennent est totalement exclu, car une relation nécessite bien vite un stockage de l'ordre d'une centaine d'octets. On voit donc que pour faire en sorte que cette technique passe à l'échelle, un minimum de soin est nécessaire.

Supposons que l'on dispose d'un ensemble de relations partielles (typiquement stockées dans une collection de fichiers, potentiellement en grand nombre). Pour effectuer la recherche de cycles, deux approches sont possibles : l'approche en une passe, et l'approche en plusieurs passes.

- La première de ces approches consiste à lire une fois l'ensemble de toutes les relations, et à fournir en sortie l'ensemble des relations recombinées produites à partir des cycles. L'algorithme *union-find* est employé sur la totalité des arêtes. Dans ce schéma, on autorise le programme à accéder une seconde fois aux relations qu'il a déjà rencontrées pour effectuer les recombinaisons possibles à partir des cycles.
- La seconde approche consiste à « filtrer » l'ensemble des relations partielles. Dans une première passe, on identifie quels sont les *large primes* qui interviennent plus d'une fois (et qui ont donc une chance d'être utiles). On élimine les relations contenant des *large primes* « isolés », et on répète cette première opération de filtrage. Ensuite on identifie quels sont les *large primes* qui interviennent dans des cycles, et dans une dernière passe on fabrique effectivement les recombinaisons à partir de ces cycles (plusieurs passes peuvent être rajoutées au fur et à mesure du procédé). Cette approche a été employée dans [Cav00] pour la factorisation d'un module RSA de 512 bits [CDL+00] et dans les travaux qui ont suivi [CAB00]. Cette approche a l'avantage de n'effectuer la recherche de cycles par *union-find* que sur un sous-ensemble des relations, allégeant ainsi la consommation en mémoire. La passe ultime de cette méthode fait donc *a priori* le travail minimum. Cependant, le travail de reconstruction des relations ff à partir des cycles prend un temps qui est en général contrôlé par la vitesse des périphériques de stockage (il est, comme on l'a dit, exclu de conserver toutes les relations en mémoire centrale). Sur ce point particulier, il y a peu de raisons de penser que l'approche en plusieurs passes apporte un bénéfice.

La méthode que nous avons mise en œuvre est la méthode en une passe, bien qu'il nous soit apparu au fil du temps que les avantages de la seconde approche auraient été assez appréciables. C'est bien sûr avec une table de hachage que les arêtes du graphe sont stockées.

Sommets	87 073 945
Arêtes (relations)	61 058 174
Relations ff	221 368
Relations pf/fp	6 083 209
Cycles obtenus	856 145
Taille du plus grand cycle (arêtes)	40
Taille de la composante connexe géante (arêtes)	22 226 085
Taille de la seconde plus grande composante connexe (majorant)	167

Table 3.2 – Données globales du graphe pour $\mathbb{F}_{2^{607}}$

On choisit une méthode de numérotation des *large primes* (par exemple en prenant les bits du polynôme) et une méthode de numérotation des relations (par exemple leur position dans un fichier sur le disque dur). On s'arrange pour qu'un *large prime* soit numéroté avec 5 octets au plus, et une relation avec 4 octets au plus.

Choix d'une fonction de hachage

Les sommets constituent le point d'entrée du graphe. À chaque sommet est donc associé un emplacement dans la table de hachage. Cet emplacement doit être unique. Si la table de hachage est vue comme un tableau de listes, on veut une fonction de hachage h telle que l'information concernant le *large prime* numéroté L soit dans la liste indexée par $h(\mathsf{L})$. La présence d'une telle information permet de déterminer si le sommet est de degré non nul ou pas. L'information à stocker, pour chaque sommet, est une identification de son parent (L' par exemple), ainsi que le numéro de la relation qui les relie (R). Pour les sommets qui sont des ancêtres, on peut choisir de prendre $\mathsf{L}' = \mathsf{L}$. On voit que si l'on place dans la liste indexée par $h(\mathsf{L})$ le triplet $(\mathsf{L}, \mathsf{L}', \mathsf{R})$, la consommation mémoire est de 14 octets par sommet au minimum.

Pour économiser de la place mémoire, on a fait le choix de prendre pour h un échantillonage fixe des bits de L. Pour cela, il faut s'assurer que les bits choisis présentent une distribution suffisamment aléatoire. Pour des *large primes* de degré allant de 24 à 36, on a pris une partie des bits d'indice 1 à 23 (le bit d'indice 0 valant toujours 1 puisque l'on traite des polynômes irréductibles). Nous n'avons pas constaté de conséquence négative de ce choix sur l'uniformité du remplissage de la table de hachage. Ce faisant, il est possible de stocker dans la liste indexée par $h(\mathsf{L})$ seulement les bits de L non déterminés. Nous avons ainsi économisé trois octets par entrée. En outre, stocker l'indice L' en entier n'est pas nécessaire. Il suffit de stocker $h(\mathsf{L}')$, et l'indice de l'entrée concernant L' dans la liste indexée par $h(\mathsf{L}')$. On économise encore ainsi deux octets par entrée, pour arriver à une occupation mémoire par entrée de neuf octets.

3.1.4 Mesures statistiques

Le graphe constitué lors du calcul de logarithmes discrets sur $\mathbb{F}_{2^{607}}$ est de proportions largement respectables, puisqu'il taquine les cent millions de sommets. Nous avons collecté de nombreuses données expérimentales sur ce graphe. Les tables 3.2, 3.3, 3.5 consignent les valeurs finales de ces différentes quantités. L'évolution chronologique de certaines d'entre elles est détaillée dans des graphiques 3.4 et 3.6.

Les données générales du graphe sont consignées dans la table 3.2. On y lit que le nombre de relations pf et fp s'élève à 10% du total, ce qui contribue bien sûr grandement à l'attrait du sommet spécial 1 dans le graphe, pour aider à la production de cycles. Dans la table 3.2, la

Taille	Nombre
2 sommets	154 507
3 sommets	147 122
4 sommets	129 428
5 sommets	107 122
6 sommets	85 558
7 sommets	65 687
8 sommets	48 810
9 sommets	35 956
10 sommets ou plus	81 954

Table 3.3 – Répartition de la taille des cycles

mention du nombre de relations ff a une valeur essentiellement indicative, puisqu'une relation ff n'intervient bien sûr jamais dans le graphe (il s'agit ici des relations ff hors recombinaisons des cycles).

Les cycles produits au cours de l'expérience sont allés bien au-delà du simple cas des cycles de taille 2 (qui correspond au cas des simples *large primes*). On voit dans la table 3.3 que ceux-ci représentent moins de 20% du nombre total de cycles. Le bénéfice de l'emploi des doubles *large primes* est donc très net. La figure 3.4 représente l'évolution du nombre de cycles au fur et à mesure du remplissage du graphe, en fonction du nombre de relations partielles. Il y apparaît très clairement que l'apport des cycles de taille 2 est bien en deçà de la contribution des cycles de taille plus importante. Toutefois, ces derniers apparaissent tard. Le premier cycle de taille 2 est apparu après 200 000 relations partielles, le premier cycle de taille 3 après 1 200 000 relations partielles (déjà une quantité importante). La contribution des cycles de taille $\geqslant 3$ a atteint 10% de celle des cycles de taille 2 après 4 650 000 relations partielles, pour enfin la dépasser après 27 000 000 de relations partielles (ce point peut s'observer sur la figure 3.4).

Il est difficile sur la figure 3.4 de voir que la courbe des cycles de taille 2 est de nature quadratique, car elle est largement dominée par les autres courbes. C'est pourtant le cas. Si l'on essaie de mesurer expérimentalement une interpolation polynomiale de la courbe des cycles de taille 3 et plus, on arrive à un exposant 3.

Une autre observation intéressante qui peut être faite sur l'évolution du graphe est le comportement des composantes connexes. Il existe une composante connexe « géante » (les données de la table 3.2 montrent combien le fossé est grand entre cette composante et les autres). Au fur et à mesure que le graphe se peuple, il y a de plus en plus de composantes qui viennent s'y fusionner. On peut donc s'intéresser, pour mettre en valeur ce phénomène, à l'évolution de la proportion de composantes connexes ayant une taille donnée. La table 3.5 consigne les valeurs finales de ces proportions.

On a représenté sur le graphique de la figure 3.6 l'évolution de la proportion des composantes connexes en fonction de leur taille (en nombre d'arêtes), *normalisée au maximum*. C'est-à-dire par exemple que l'on a tracé la proportion du nombre de composantes connexes de taille 4 par rapport à la valeur maximale de 0.88%. On voit sur cette figure que le point d'« effondrement » du graphe est réellement dépassé à partir de 40 millions d'arêtes environ, puisque pour toutes les tailles de composantes connexes (jusqu'à la taille de 11 *et plus*), on voit que la proportion est nettement décroissante (cette constatation inspire bien sûr un rapprochement avec les modèles de percolation).

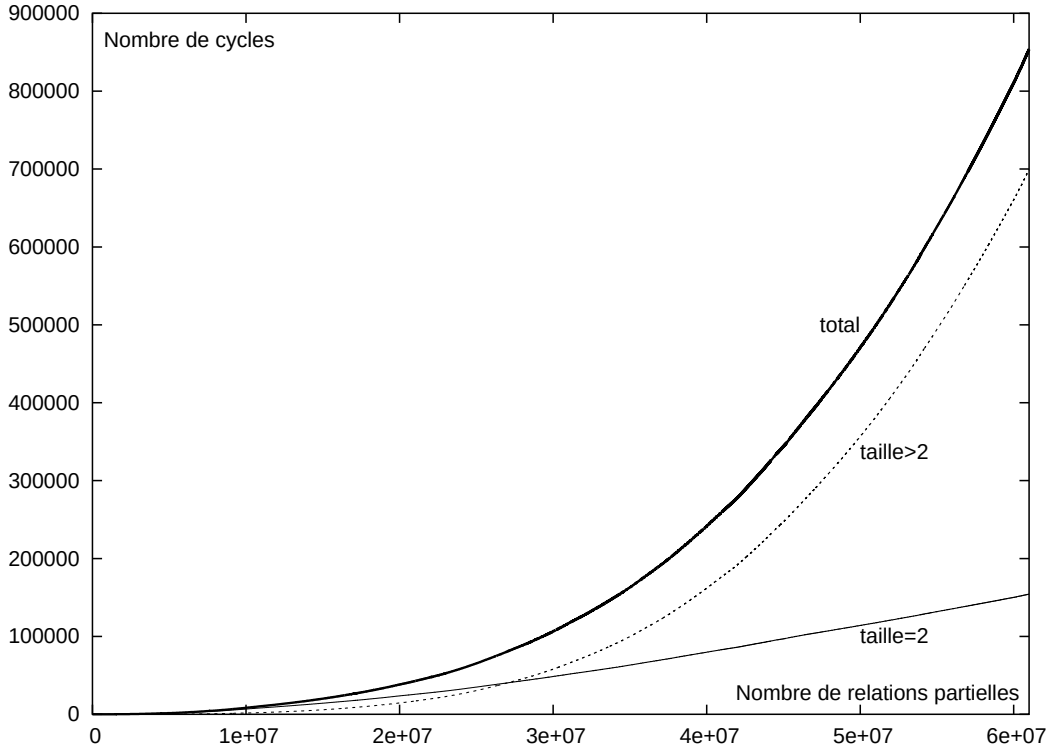


Figure 3.4 – Évolution du nombre de cycles

Taille	Nombre	Proportion	Maxi
1 arête	22 226 085	36%	93%
2 arêtes	2 755 157	4.5%	6.1%
3 arêtes	856 449	1.4%	1.9%
4 arêtes	385 286	0.63%	0.88%
5 arêtes	206 514	0.33%	0.48%
6 arêtes	124 223	0.20%	0.29%
7 arêtes	81 119	0.13%	0.19%
8 arêtes	55 325	0.091%	0.13%
9 arêtes	39 229	0.064%	0.093%
10 arêtes	29 103	0.048%	0.068%
11 arêtes ou plus	113 426	0.19%	0.26%

Table 3.5 – Répartition de la taille des composantes connexes

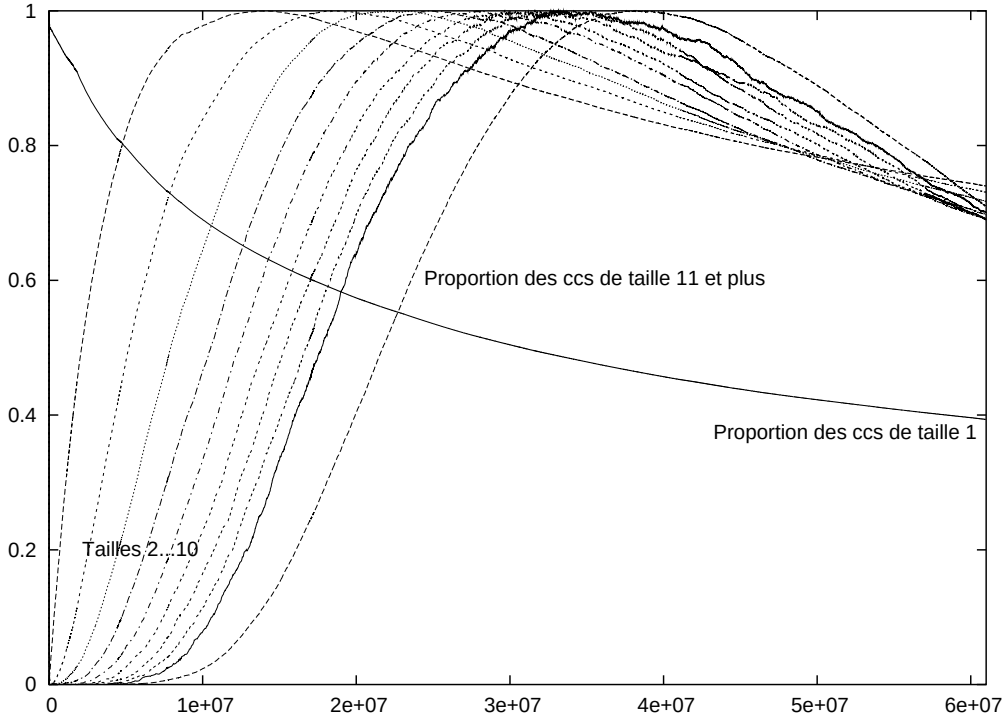


Figure 3.6 – Effondrement des composantes connexes (taille $1 \ldots 11$ et plus)

3.1.5 Alternatives

Après l'emploi d'un *large prime*, de deux *large primes*, on est bien sûr tenté de généraliser le schéma pour prendre en compte un nombre arbitraire (mais qui reste borné) de *large primes*. Cette idée a été mise en œuvre dans [DL95], et surtout dans [Cav00, Cav02], avec le principe des *k-merges* : plusieurs relations, en nombre k par exemple, ayant en commun un même *large prime* peuvent être fusionner pour former $k-1$ relations où ce *large prime* est absent. L'adaptation à notre cadre de ces techniques n'est pas immédiate : en effet, l'importance des *exposants* dans les relations est bien plus grande que pour les relations de factorisation, où ceux-ci sont seulement considérés modulo 2. Il en résulte une croissance globale des exposants des relations, qui doit être contrôlée.

3.2 Sans crible : tests de friabilité

À l'origine, la méthode proposée par Coppersmith, et qui paraît la plus naturelle au vu de la description de l'algorithme effectuée au chapitre précédent, repose sur un test de friabilité efficace. On rappelle l'énoncé de la proposition A.12 :

Proposition. *Le polynôme $X^{q^n} - X \in \mathbb{F}_q[X]$ est le produit de tous les polynômes irréductibles de $\mathbb{F}_q[X]$ de degré divisant n.*

On déduit, à partir de cette propriété, un test de friabilité facile à mettre en œuvre. Nous allons voir que ce test ne fournit pas une réponse juste à 100%, mais qu'il est largement suffisant.

Proposition 3.2. *Soit* $P \in \mathbb{F}_2[X]$. *Si* $P' \prod_{k=\lceil \frac{b}{2} \rceil}^{b} (X^{2^k} + X) \equiv 0 \bmod P$, *alors* P *est* b*-friable sauf si les facteurs de degré* $> b$ *de* P *sont de multiplicité paire.*

Démonstration. Partons d'abord du cas où P est b-friable. Le polynôme P s'écrit donc $\prod_i \pi_i^{e_i}$. Il est évident que $\prod_{i|e_i>0} \pi_i^{e_i-1}$ divise le polynôme dérivé P'. En outre, chacun des π_i est de degré $\leqslant b$, donc $\deg \pi_i$ divise un entier de l'intervalle $[\![\lceil \frac{b}{2} \rceil \dots b]\!]$. Il s'ensuit que le test est vérifié.

Soit maintenant un polynôme P vérifiant le test, soit q un facteur irréductible de degré $> b$ de ce polynôme, et k sa multiplicité dans la factorisation de P. Le polynôme q est premier avec le produit $\prod_{k=\lceil \frac{b}{2} \rceil}^{b} (X^{2^k} + X)$. Donc le test ne peut être vérifié que si q^k divise P'. Il existe un polynôme R premier avec q tel que :

$$\begin{aligned} P &= q^k R, \\ P' &= q^k R' + k q^{k-1} q' R, \\ q^k \mid P' &\Leftrightarrow q | k q', \\ q^k \mid P' &\Leftrightarrow k \equiv 0 \bmod 2. \end{aligned}$$

On a donc obtenu la propriété recherchée. ■

Les « faux témoins » pour le test précédent sont donc rares. On peut en calculer la proportion asymptotique en fonction de b en utilisant des techniques de séries génératrices. Une estimation correcte nécessite l'emploi de la méthode du point col [FS94]. Ce calcul n'est pas effectué ici.

Comme nous le verrons en 3.3, ce test n'est pas le moyen de sélection le plus employé, puisque l'on préfère largement employer des techniques de crible. Néanmoins il conserve son intérêt en bien des circonstances, car on ne crible pas à la fois pour le polynôme C et le polynôme D.

Effet de l'emploi de *large primes*

Lorsque l'on souhaite employer des *large primes*, suivant ce qui a été décrit précédemment en 3.1, il est nécessaire d'adapter le test de friabilité. Cette adaptation peut se faire à moindre coût, et c'est bienvenu car dans le cas contraire, l'emploi de *large primes* présenterait un avantage amoindri (les relations partielles ne seraient pas tout à fait « gratuites »). Si l'on note S le polynôme introduit dans la proposition ci-dessus, à savoir :

$$S = P' \prod_{k=\lceil \frac{b}{2} \rceil}^{b} (X^{2^k} + X),$$

on peut en fait préciser le résultat de la proposition en disant que, hormis dans les cas rares où P a un grand facteur répété, le degré de $\operatorname{pgcd}(S, P)$ est exactement la contribution des petits facteurs à la factorisation de P. Il s'ensuit que si $\deg P - \deg \operatorname{pgcd}(S, P) \leqslant \mathcal{L}$, alors le cofacteur dans la factorisation de P est de degré au plus $\mathcal{L}$.

3.3 Le principe du crible

Le travail à effectuer lors de la première phase de l'algorithme de Coppersmith consiste à examiner un grand nombre de paires (A, B), et de déterminer parmi ces paires lesquelles donnent lieu à des polynômes C et D *friables*. Puisque la probabilité de friabilité est très faible, nous devons tester un très grand nombre de paires, parmi lesquelles seule une infime proportion est réellement intéressante. Dans ce contexte, on peut avantageusement mettre en œuvre une technique de *crible*. Nous savons que l'évaluation de la friabilité de C permet à elle seule d'effectuer un tri drastique parmi les paires[6]. Le crible permet d'effectuer ce tri à moindres frais.

Plutôt que d'examiner les différents polynômes C possibles les uns après les autres, et d'évaluer leur friabilité, c'est-à-dire la contribution des petits facteurs dans leur factorisation, le principe du crible est de travailler en premier lieu sur les petits facteurs. Pour un petit polynôme irréductible g, on veut identifier les paires (A, B) telles que le polynôme $C = AX^h + B$ est divisible par g. Cela revient à résoudre tout simplement la congruence :

$$B \equiv AX^h \bmod g.$$

Les contraintes spécifiques à notre problème sont les suivantes. L'espace des paires (A, B) est de taille colossale (2^{51} paires pour le cas de $\mathbb{F}_{2^{607}}$). On le considère donc par tranches, où A est fixé. Alors, B est un polynôme de degré compris entre 0 et d_B. L'ensemble des polynômes B est donc de taille 2^{d_B+1}. On considère un tableau $\mathcal{S}$ de 2^{d_B+1} entiers correspondant à ces polynômes (on se permet de noter $\mathcal{S}[B]$ l'entrée correspondant à B). Les entrées de ce tableau sont initialement mises à zéro, et pour chaque polynôme irréductible g de degré $\leqslant b$, on augmente $\mathcal{S}[B]$ de la quantité $\deg g$ pour les polynômes B qui satisfont la congruence ci-dessus. Ainsi, une fois que ce travail a été effectué pour tous les polynômes de degré $\leqslant b$ (et leurs puissances), la valeur de $\mathcal{S}[B]$ correspond à la contribution des facteurs de degré $\leqslant b$ dans la factorisation de $C = AX^h + B$. Si cette contribution atteint $\deg C = h + \deg A$, alors le polynôme C est b-friable. Si cette contribution atteint seulement $h + \deg A - \mathcal{L}$, où $\mathcal{L}$ est la borne maximale autorisée pour les *large primes* (cf. 3.1), alors C est « presque friable ».

Effectuer ce travail de crible efficacement nécessite de pouvoir identifier facilement toutes les solutions de la congruence $B \equiv AX^h \bmod g$. En outre, il est important de pouvoir itérer facilement l'opération $\mathcal{S}[B]$ += $\deg g$ sur ces solutions. Comme les solutions diffèrent entre elles d'un multiple de g, il faut pouvoir parcourir l'ensemble des multiples de g rapidement. Les solutions à la congruence qui nous intéresse forment un sous-espace affine de l'espace des polynômes de degré $\leqslant d_B$, que l'on voit comme un espace vectoriel de dimension $d_B + 1$ sur $\mathbb{F}_2$. L'espace affine en question est :

$$B_0 + \langle g, Xg, \ldots, X^{d_B - \deg g} g \rangle_{\mathbb{F}_2}, \text{ où } B_0 = (AX^h \bmod g).$$

Pour parcourir efficacement cet espace affine, on veut n'avoir à effectuer que des additions polynomiales, jamais des multiplications. La notion de code de Gray permet de parcourir un espace vectoriel sur $\mathbb{F}_2$ en ne faisant qu'une addition de vecteur de base à chaque étape (on présente en général le code de Gray comme un parcours des sommets d'un hypercube, qui n'est rien d'autre qu'un espace vectoriel sur $\mathbb{F}_2$). La proposition suivante nous donne le parcours à considérer.

[6] Dans notre cas, $\deg C > \deg D$. La probabilité de friabilité de C est donc inférieure à celle de D. Dans le cas contraire, cribler sur D est plus avantageux. Ce cas sera abordé page 54.

Proposition 3.3. *Soit $V = \langle e_0, \ldots, e_{d-1}\rangle_{\mathbb{F}_2}$ un espace vectoriel de dimension d sur $\mathbb{F}_2$. Pour un entier i, notons $\ell(i)$ l'indice du premier bit non nul de i (c'est-à-dire le plus grand entier k tel que $2^k | i$). Soit $(u_n)_n$ la suite définie par :*

$$\begin{cases} u_0 = 0, \\ u_n = u_{n-1} + e_{\ell(n)}. \end{cases}$$

Alors $\{u_0, \ldots, u_{2^d-1}\}$ est l'ensemble des points de V.

Démonstration. La preuve est une récurrence. Pour $d = 0$ le résultat est trivial. Soit maintenant $d \geqslant 0$. On sait par hypothèse de récurrence que $\{u_0, \ldots, u_{2^d-1}\}$ est l'ensemble des points de $\langle e_0, \ldots, e_{d-1}\rangle_{\mathbb{F}_2}$. En outre, on a $\ell(2^d) = d$, donc $u_{2^d} = e_d + u_{2^d-1}$. Pour chaque entier i tel que $0 < i < 2^d$, on a $\ell(2^d + i) = \ell(i)$. Par conséquent, $u_{2^d+i} - u_{2^d} = u_i - u_0$. L'ensemble des vecteurs $\{u_{2^d}, \ldots, u_{2^{d+1}-1}\}$ est donc l'ensemble des points de :

$$u_{2^d} - u_0 + \langle e_0, \ldots, e_{d-1}\rangle_{\mathbb{F}_2} = e_d + \langle e_0, \ldots, e_{d-1}\rangle_{\mathbb{F}_2}.$$

Le résultat s'ensuit. ■

Un second point important est le coût que représente le calcul de la valeur initiale $AX^h \bmod g$. Pour des polynômes g de degré important, cette réduction d'un polynôme modulo g est le point délicat. En particulier, il prend le pas sur le parcours du code de Gray, dont le coût décroit exponentiellement avec $\deg g$. Ces deux aspects doivent donc être optimisés.

Un pseudo-code indiquant comment programmer un tel crible polynomial est donné en 3.7. Du point de vue de l'implantation, l'indexage du tableau $\mathcal{S}$ par un polynôme ne pose pas de problème. En effet, la représentation la plus évidente[7] des polynômes sur $\mathbb{F}_2$ utilise un bit par coefficient, et fournit donc immédiatement la bijection naturelle avec les entiers. Le pseudo-code fourni considère que c'est cette représentation qui est utilisée, et que les polynômes traités sont de degré suffisamment petit pour pouvoir être représentés par un mot machine.

Crible sur le polynôme D

Dans la description que l'on a faite du crible, rien ne prescrit la forme spécifique du polynôme $C = AX^h + B$. Il est possible aussi de cribler sur le polynôme D défini par $D = A^k X^{hk-n} f_1 + B^k$. Le choix entre ces deux possibilités est dicté par la recherche d'efficacité du crible. Comme le temps de calcul du crible est presque indépendant du nombre de paires qu'il sélectionne, on souhaite se placer dans la situation où ce nombre de paires friables sélectionnées est le plus faible. Cela nous conduit à cribler sur le polynôme parmi C et D dont le degré est le plus important, puisque c'est avant tout le degré qui contrôle la probabilité de friabilité. Cribler sur D n'est donc avantageux que si celui-ci est degré supérieur à $\deg C$, puisqu'alors le crible opère une sélection plus drastique.

Idéalement, les polynômes C et D sont de degré identique (leur valeur asymptotique commune est $\sqrt{nd_A}$). En réalité, ces degrés ne sont *pas* équilibrés : la grandeur qui « assure » cet équilibre est le paramètre k, et comme ce paramètre est contraint à être une puissance de 2, on est assez éloigné du comportement asymptotique en ce domaine[8]. Pour la plage de problèmes dans laquelle se situe la résolution de logarithmes discrets dans $\mathbb{F}_{2^{607}}$, on a $k = 4$, et

[7] Cette représentation est celle employée par exemple par Magma (en interne) et par les librairies `ZEN` et `NTL`.

[8] En cela, l'algorithme FFS, qui lève cette restriction sur le paramètre k, présente un avantage certain.

```
for(i=0;i<2^{d_B+1};i++) S[i]=0;
for(k=1;k<=b;k++) {
    for(g irréductible de degré k) {
        e[0]=g;
        for(d=1;d<=d_B − k;d++) e[d]=e[d-1]<<1;
        B=(AX^h mod g);              // Initialisation parfois coûteuse
        u=0;
        for(i=0;i<2^{d_B−k+1};i++) {     // Cette boucle est critique
            if (i) B=B ^ e[ℓ(i)];    //
            S[B]+= k;                //
        }                            //
    }
}
for(i=0;i<2^{d_B+1};i++) {
    if (S[i]>=h + deg A − L) {
        B=i;
        marquer la paire (A, B);
    }
}
```

Programme 3.7: Crible polynomial

on se rapproche du point où le choix $k = 8$ est plus avantageux. Il en résulte que le polynôme C est de degré considérablement plus important que le polynôme D (178 contre 112 dans notre cas). Par conséquent, cribler sur D n'a pas été envisagé pour le calcul de logarithmes discrets sur $\mathbb{F}_{2^{607}}$. Toutefois, nous mentionnons ici comment un tel crible peut être réalisé.

Lorsque l'on a décrit le crible sur le polynôme C, on a mis en évidence l'intérêt de la congruence $B \equiv AX^h \bmod g$. La congruence jouant le rôle analogue lorsque D prend la place de C est :

$$B^k \equiv A^k X^{hk-n} f_1 \bmod g.$$

Concentrons-nous sur le cas où g est un polynôme irréductible. En cela, on exclut le cas où g est une *puissance* d'un polynôme irréductible[9]. On peut alors simplifier ainsi l'écriture précédente :

$$B \equiv A \sqrt[k]{X^{hk-n} f_1} \bmod g.$$

La grandeur X^h qui intervient lors du crible sur C doit maintenant être remplacée par la grandeur $\sqrt[k]{X^{hk-n} f_1} \bmod g$. Cette grandeur est incontestablement plus complexe, d'autant plus qu'elle dépend de g. Toutefois, il est parfaitement envisageable de précalculer ces données : si l'on crible sur un million de polynômes irréductibles (c'est une grandeur indicative), le précalcul de ces données représente 4 mégaoctets, soit peu de choses.

En dehors de cet amendement de l'*initialisation* du crible, de l'omission « par construction » des puissances de polynômes irréductibles, et bien évidemment de la borne finale $h + \deg A - \mathcal{L}$ qui devient la quantité un peu plus complexe qui contrôle $\deg D$, l'algorithmique ne change pas : la congruence centrale est transcrite de manière très similaire.

[9] Pour les raisons qui ont déjà été exposées page 35 lors de la discussion sur le choix du polynôme, la considération de telles puissances pour le polynôme D est d'un intérêt pratiquement nul.

Il est enfin possible d'effectuer *deux* cribles, l'un pour C et l'autre pour D. Cela est intéressant si le crible s'avère considérablement plus rapide que la factorisation.

3.4 Le crible partiel : évaluation statistique des contributions des facteurs

Comme cela est indiqué en commentaire dans le pseudo-code 3.7, le schéma de crible polynomial précédemment décrit souffre de deux inconvénients :

- Lorsque le polynôme g est de degré trop important, le coût d'initialisation (qui implique une réduction modulaire) peut s'avérer important.
- Lorsque le polynôme g est de petit degré, la boucle qui met à jour les entrées du tableau est très longue. En outre, cette boucle n'accède pas à la mémoire de façon très ordonnée. Il en résulte une nette chute de performance (due au comportement de la mémoire cache).

Pour pallier ces inconvénients, on a employé une stratégie de *crible partiel*. Cette technique s'apparente à des techniques déjà employées dans les algorithmes de factorisation. Le principe est d'évaluer la probabilité de friabilité à partir des contributions des facteurs les plus « faciles » à traiter. Plus exactement, cela signifie que dans le crible, on omet les polynômes g de petit et de grand degré. Pour les deux raisons que l'on a évoquées au paragraphe précédent, les polynômes « moyens » sont en effet traités plus efficacement. Cette omission délibérée des polynômes de petit degré dans le crible s'apparente à la *small prime variation* employée dans les algorithmes de factorisation.

Bien entendu, si l'on omet certains degrés, la borne $h + \deg A - \mathcal{L}$, appelée *borne de qualification*, doit être corrigée en fonction des facteurs que l'on a omis. Cette borne doit maintenant refléter la contribution statistique des facteurs de degré intermédiaire à la factorisation des polynômes friables. Les désavantages que l'on rencontre en omettant ainsi des facteurs apparaissent lors du choix de la borne de qualification. Les deux effets suivants interviennent :

- Alors que la contribution globale de tous les polynômes irréductibles de degré 1 à b à la factorisation d'un polynôme b-friable de degré d est évidemment égale à d, la contribution des polynômes irréductibles dont le degré est dans un sous-ensemble de $[\![1 \dots b]\!]$ est plus dispersée. Pour capturer une proportion raisonnable des polynômes C friables au cours du crible, on doit donc placer la borne de qualification relativement bas (d'autant plus bas que l'écart-type de la contribution est élevé).
- Si la borne de qualification est placée trop bas, alors de très nombreux polynômes C sont « qualifiés ». Par conséquent, la proportion des polynômes friables parmi ceux-ci est de plus en plus faible (tendant à la limite vers la probabilité de friabilité, auquel cas le crible a un pouvoir de sélection nul).

Ces deux effets sont clairement antagonistes. Nous allons voir comment il est possible de les équilibrer et de bien choisir la borne de qualification, afin de maintenir le pouvoir de sélection du crible. La figure 3.8 consigne quatre exemples de résultats obtenus en évaluant les deux effets cités en fonction du choix de la borne de qualification. Dans chaque cas, on s'est intéressé à la 23-friabilité d'un polynôme aléatoire[10] de degré 178. Les mesures sont effectuées pour un crible complet d'abord, puis pour trois versions de crible « partiel », où l'on a criblé seulement pour les polynômes irréductibles g de degré compris entre 10 et 20 pour le deuxième

[10] Par conséquent, comme le polynôme C n'est pas un polynôme aléatoire, ces distributions doivent être vues comme essentiellement indicatives.

graphique, 6 et 22 pour le troisième, et 10 et 23 pour le quatrième. Les différentes valeurs possibles de la borne de qualification sont données en abscisse.

Sur chaque graphique, la partie grisée représente la proportion des polynômes friables qui passent le crible. On cherche à maximiser cette quantité. Il est clair que si le crible est complet, alors un polynôme friable a une contribution maximale, donc la zone grisée reste « calée » à 100%. Dans les autres cas, à cause de la dispersion statistique de la contribution des facteurs dans la plage sélectionnée, il est net que cette zone grisée décroît si la borne de qualification est trop « stricte ». On représente ainsi le premier des effets mentionnés ci-dessus.

La « qualité de sélection » du crible est la courbe qui se détache sur les graphiques. On souhaite aussi maximiser cette qualité. Elle indique, en fonction de la borne de qualification choisie, la proportion des polynômes friables parmi les polynômes « qualifiés ». On représente ainsi le second des effets mentionnés plus haut[11]. Le comportement de cette courbe varie peu en fonction des situations : elle doit être vue comme une référence à associer à la seconde donnée. En effet, la borne de qualification doit être choisie de telle sorte que les deux quantités étudiées soient grandes. Par exemple, dans le cas du second des graphiques de la figure 3.8, on ne peut espérer raisonnablement « capturer » la moitié des paires friables : cela imposerait une borne de friabilité au plus égale à 100, ce qui correspond à une très mauvaise qualité de sélection.

Une conclusion synthétique de la lecture de la figure 3.8 est que la qualité du crible n'est globalement pas trop affectée si l'on omet les facteurs de petit degré. En revanche, ne pas cribler pour les facteurs irréductibles de degré important est un facteur très net de dégradation de la qualité du crible. Lors du calcul de logarithmes discrets sur $\mathbb{F}_{2^{607}}$, dont les graphiques de la figure 3.8 sont extraits, le crible a été effectué pour la plage de degré $[\![10 \dots 23]\!]$, correspondant à la dernière des courbes représentées. Il est net que parmi les situations représentées, c'est celle qui offre le meilleur compromis possible.

Obtention des évaluations statistiques

Les graphiques de la figure 3.8 ont été obtenus à l'aide de séries génératrices. Si l'on note N_k le nombre de polynômes irréductibles de degré k sur $\mathbb{F}_2$, la série génératrice des polynômes unitaires (où le coefficient en z^n est le nombre de polynômes sur $\mathbb{F}_2$ de degré n, à savoir 2^n) s'écrit des trois façons suivantes :

$$\sum_{n=0}^{\infty} 2^n z^n = \frac{1}{1-2z} = \prod_{k=1}^{\infty} \frac{1}{(1-z^k)^{N_k}}.$$

Cette identité découle des résultats élémentaires de [FS93] sur les séries génératrices pour l'énumération des structures combinatoires. Si l'on souhaite exprimer la série génératrice des polynômes b-friables, il suffit de restreindre le produit dans le dernier membre de l'équation ci-dessus. On arrive à l'expression

$$f_b(z) = \prod_{k=1}^{b} \frac{1}{(1-z^k)^{N_k}}.$$

Plus généralement, on peut énoncer la proposition suivante (toujours à partir de résultats élémentaires tels que présentés dans [FS93]) :

[11] On remarque que la courbe en question « plafonne » à partir de la borne de qualification 155. En effet, $155 = 178 - 23$: un polynôme de degré 178 dont on a identifié une partie 23-friable de degré $\geqslant 155$ est nécessairement 23-friable.

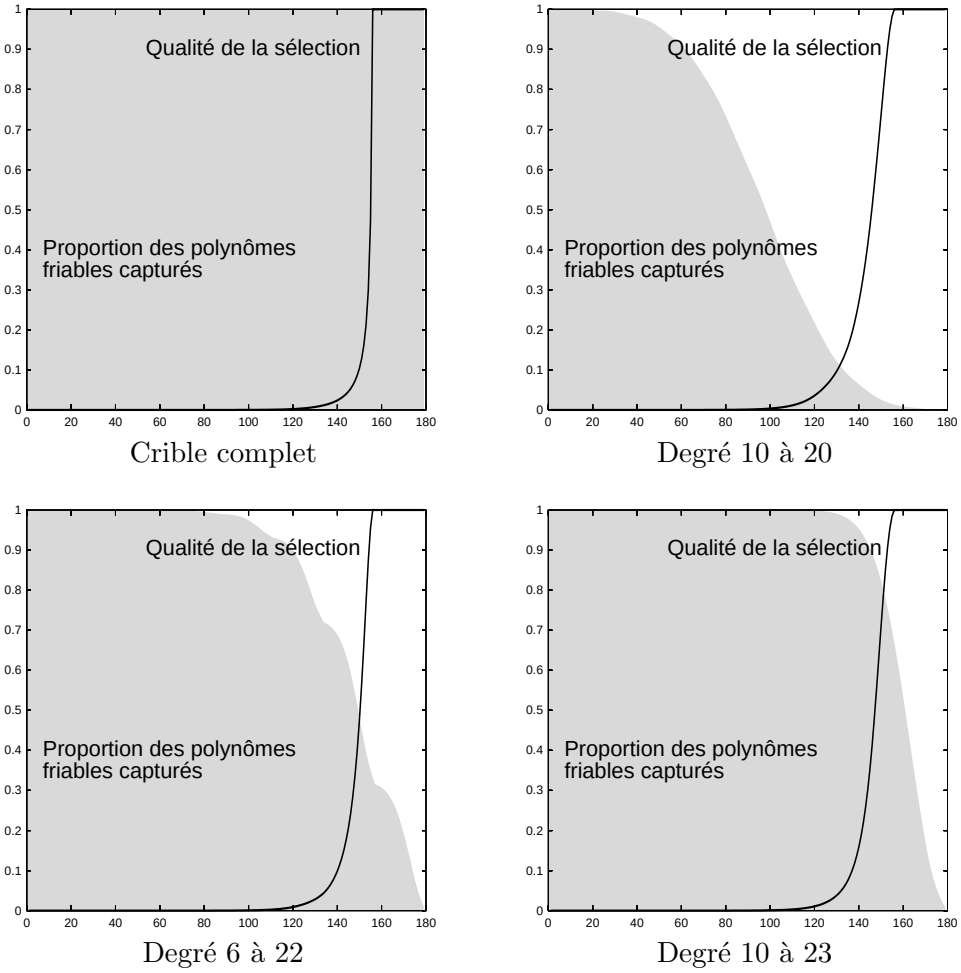


Figure 3.8 – Évolution de la qualité du crible partiel

Proposition 3.4. *Soit $R \subset [\![1 \dots b]\!]$.*

– *Soit la série $f_{R,b}(u,z)$ définie par :*

$$f_{R,b}(u,z) = \prod_{\substack{1 \leqslant k \leqslant b, \\ k \notin R}} \frac{1}{(1-z^k)^{N_k}} \prod_{k \in R} \frac{1}{(1-(uz)^k)^{N_k}}.$$

Alors le coefficient du monôme en $u^m z^n$ de $f_{R,b}(u,z)$ (noté $[u^m z^n] f_{R,b}(u,z)$) est le nombre de polynômes b-friables de degré n dont la contribution des facteurs de degré appartenant à R dans la factorisation est égale à m.

– *Soit en outre la série $g_R(u,z)$ définie par :*

$$g_R(u,z) = f_{R,\infty}(u,z) = \frac{1}{1-2z} \prod_{k \in R} \left(\frac{1-z^k}{1-(uz)^k} \right)^{N_k}.$$

Alors $[u^m z^n] g_R(u,z)$ est le nombre de polynômes de degré n (b-friables ou non) dont la contribution des facteurs de degré appartenant à R dans la factorisation est égale à m.

Il s'ensuit que pour une borne de qualification égale à q, le nombre de polynômes b-friables de degré d qui seraient qualifiés s'ils apparaissaient dans un crible[12] est égal à : $\sum_{k\geqslant q}[z^d u^k] f_{R,b}(u,z)$. On peut simplifier un peu cette écriture, si l'on note $h(u)$ la série $[z^d] f_{R,b}(u,z)$ (en fait $h(u)$ est un polynôme en u). Les coefficients de $h(u)$ sont notés $h_k = [u^k]h(u)$. On a :

$$\begin{aligned}
\sum_{k\geqslant q}[z^d u^k] f_{R,b}(u,z) &= \sum_{k\geqslant q} h_k = [u^q] \sum_{k\geqslant 0} \left(\sum_{l\geqslant k} h_l \right) u^k, \\
&= [u^q] \sum_{k\geqslant 0} \left(h(1) - \sum_{l\leqslant k-1} h_l \right) u^k, \\
&= [u^q] \left(\frac{h(1)}{1-u} - \sum_{k\geqslant 0} \left(\sum_{l\leqslant k-1} h_l \right) u^k \right), \\
&= [u^q] \left(\frac{h(1) - uh(u)}{1-u} \right), \\
&= [u^q z^d] \left(\frac{f_{R,b}(1,z) - u f_{R,b}(u,z)}{1-u} \right).
\end{aligned}$$

Parallèlement, on montre que le nombre total de polynômes qualifiés pour une borne de qualification égale à q, que ces polynômes soient b-friables ou pas, est donné par :

$$\sum_{k\geqslant q}[z^d u^k] g_R(u,z) = [u^q z^d] \left(\frac{g_R(1,z) - u g_R(u,z)}{1-u} \right).$$

De ces formules dérivent les représentations de la figure 3.8. Les grandeurs considérées font qu'aucun soin particulier n'est nécessaire pour l'évaluation des coefficients des séries formelles mentionnées, il est donc possible de mener des calculs exacts[13] (malgré le manque d'intérêt de la vingtième décimale). L'emploi de développements asymptotiques est alors probablement de peu d'intérêt : leur obtention n'apparaît pas comme évidente, et pour les petits degrés de polynômes qui nous intéressent, il n'est pas impossible que le développement asymptotique doive être poussé assez loin pour obtenir une approximation correcte des grandeurs qui nous intéressent.

Généralisations de la méthode d'évaluation

Bien que la description qui vient d'être faite ne tienne pas compte des *large primes*, ceux-ci peuvent s'insérer aisément dans le schéma. Les séries formelles manipulées deviennent un peu plus compliquées. Un autre facteur de complication des écritures est l'omission du crible sur les *puissances* des polynômes irréductibles. En effet, le bénéfice est là aussi maigre comparé au coût. Similairement, on peut faire entrer cette considération en ligne de compte dans l'écriture des séries formelles.

[12] Les polynômes de la forme $C = AX^h + B$ ne forment bien sûr qu'un très maigre sous-ensemble de cette classe de polynômes.

[13] Le programme Magma, qui ne dispose pas d'algorithmes très avancés pour la manipulation de séries formelles à ce jour, a traité les graphiques de la figure 3.8 en moins d'une minute.

3.5 Le groupement de cribles

3.5.1 L'espace de crible nécessaire

Un des aspects de la recherche de relations que nous avons durement constaté par l'expérience est que les probabilités de friabilité sont très instables. Bien entendu, des paires (A, B) de petit degré correspondent à des paires (C, D) de degré modéré, et donc à des probabilités de friabilité de ces polynômes plus importantes. Le *rendement* en relations de chaque « paquet » de paires (A, B) décroît avec l'évolution du degré de A et B. Une évaluation précise de l'espace de crible nécessaire importe donc des tests sur les différentes plages de degré.

Hélas, pour rendre la situation encore plus complexe, d'autres facteurs déjà évoqués interviennent. L'emploi de cribles partiels a pour effet d'augmenter le rendement local pour des paquets de paires de même degré, mais cette approche a le défaut de « consommer » l'espace de crible plus vite. Pour cette raison, les graphiques de la figure 3.8 attirent l'attention sur la proportions des polynômes friables qui sont capturés par le crible. En effet, il ne sert à rien de cribler très vite et peu soigneusement une zone où, avec du soin, on peut s'attendre à obtenir une grande quantité de relations.

Un dernier facteur est bien entendu l'emploi de *large primes*. Il est très difficile de prévoir à quel moment aura lieu l'explosion combinatoire du nombre de cycles du graphe constitué par les relations partielles (correspondant à l'« effondrement » de ce graphe). Par conséquent, prévoir le nombre de relations partielles sur lequel on doit compter pour pouvoir obtenir le nombre recherché de relations recombinées est extrêmement ardu.

Pour toutes ces raisons, on peut difficilement espérer « viser juste » pour l'espace de crible. Et dans le calcul de logarithmes discrets sur $\mathbb{F}_{2^{607}}$ qui a été effectué au cours de cette thèse, il est clair que l'espérance initiale était bien optimiste par rapport à la réalité. Nous avons donc mis en place un schéma permettant de concentrer les calculs d'abord sur les zones où l'on espère obtenir un plus grand nombre de relations. Le schéma consistant à découper l'espace de crible en tranches de la forme $(A, -)$ où A est fixé et B varie sur toute la plage de degré possible nous est apparu trop restrictif.

Les développements que nous présentons ici sont donc essentiellement motivés par des considérations pratiques de gestion du calcul. De telles méthodes ont aussi été utilisées pour les grands efforts de factorisation, avec le crible algébrique par exemple.

3.5.2 Distribution du crible en paquets

Un premier découpage, dont la raison d'être est essentiellement une raison de gestion du calcul pour la distribution, a défini une grande quantité de sous-tâches, que l'on a nommées *paquets* (i.e. paquets de paires à cribler). Ces paquets sont définis par deux paramètres A_f et B_f représentant les parties fixes des polynômes A et B concernés :

$$\begin{aligned}\text{paquet}(A_f, B_f) =\{&(A, B) = (A_f X^{\delta_A+1} + A_v, B_f X^{\delta_B+1} + B_v),\\ &\deg A_v \leqslant \delta_A, \deg B_v \leqslant \delta_B\}.\end{aligned}$$

Les parties fixes A_f et B_f sont donc des polynômes de degrés inférieurs ou égaux à $d_A - \delta_A - 1$ et $d_B - \delta_B - 1$, respectivement. À titre indicatif, on donne les paramètres choisis pour le calcul sur $\mathbb{F}_{2^{607}}$: $\delta_A = 6$, $\delta_B = 24$.

Un ordinateur qui prend part au calcul et qui reçoit la responsabilité de cribler un paquet peut effectuer ce crible de la façon qui l'arrange le mieux. L'approche simple consiste à traiter

séparément les paires correspondant à des valeurs différentes de A_v. Avec les valeurs issues du calcul sur $\mathbb{F}_{2^{607}}$, cela correspond à $2^7 = 128$ cribles, traitant chacun un espace de 2^{25} polynômes B_v, ce qui nécessite 32Mo de mémoire vive. Pour chacun de ces 128 cribles, la congruence à résoudre se modifie par rapport à celle qui a été présentée plus haut, à cause du terme B_f. Cette congruence devient :

$$B_v \equiv AX^h + B_f X^{\delta_B+1} \bmod g, \qquad \text{où } A = A_f X^{\delta_A+1} + A_v.$$

En faisant ainsi, on ne modifie pas profondément le fonctionnement du crible.

3.5.3 Division de la table de crible

On peut trouver, et cela a été le cas parfois, qu'imposer 32Mo pour la table de crible pour toutes les machines prenant part au calcul est un peu difficile, surtout si l'on souhaite utiliser ces machines en concurrence avec les programmes des autres utilisateurs[14]. À cet effet, une variation simple permet de résoudre la consommation en mémoire : diviser encore la table de crible, par exemple par 2^γ, pour γ un petit entier, en fixant γ bits supplémentaires dans B_v (ce qui amène le nombre de sous-cribles à traiter pour un paquet à $2^{\delta_A+1+\gamma}$). Ainsi, la taille de la table de crible est abaissée à $2^{\delta_B+1-\gamma}$, soit dans notre cas $2^{-\gamma} \times 32$Mo. Le problème de cette approche est bien sûr que lorsque l'on diminue autant la table de crible, le coût d'initialisation payé pour chacun des polynômes irréductibles considérés devient dominant. L'efficacité en souffre durement. En employant des précalculs, on parvient à amoindrir cet effet néfaste, mais le résultat reste décevant.

3.5.4 Amortissement du coût d'initialisation

Nous avons souhaité modifier le schéma précédent pour permettre de réduire le coût d'initialisation. La constatation de départ est que pour deux valeurs de A qui ne diffèrent que de quelques bits, la valeur d'initalisation du crible (notée B_0 lors de la description du crible polynomial) change peu. Nous allons donc calculer les valeurs d'initialisation du crible pour plusieurs A possibles : choisissons une certaine quantité, ϵ, des bits de A_v que l'on va laisser varier au sein de chaque crible (étant donné que l'on est contraint à rester dans le même « paquet », on doit avoir $\epsilon \leq \delta_A + 1$). Ainsi, aux entrées de la table de crible ne correspondent plus seulement des valeurs de B_v, mais des couples (α, B_v), où α est un polynôme de degré $\epsilon - 1$. La paire associée au couple (α, B_v) est la paire $(A + \alpha, B_f X^{\delta_B+1} + B_v)$, où A est $A_f X^{\delta_A+1} + A_v$, et A_v est divisible par X^ϵ. Lorsque, dans le crible, on s'intéresse au facteur irréductible g, la congruence à résoudre est alors :

$$B_v + \alpha X^h \equiv AX^h + B_f X^{\delta_B+1} \bmod g.$$

Les solutions de cette congruence forment un sous-espace affine $\mathbb{S}$ de l'espace vectoriel sur $\mathbb{F}_2$ noté $\mathbb{V} = F \oplus G$, où F et G sont définis par :

$$F = \langle 1, X, X^2, \ldots X^{\delta_B} \rangle,$$
$$G = \langle X^h, \ldots X^{h+\epsilon-1} \rangle.$$

[14] L'inévitable disparité de l'ensemble des machines utilisées pour un calcul de cette envergure entraîne nécessairement des problèmes de ce type.

	$\epsilon = 0$	$\epsilon = 1$	$\epsilon = 2$	$\epsilon = 3$	
$\gamma = 0$	33.28 (0%) 105.68 (0%) 144.24 (0%)	22.76 (-31%) 109.96 (+4%) 138.20 (-4%)	12.26 (-63%) 107.75 (+1%) 125.35 (-13%)	7.01 (-78%) 110.62 (+4%) 122.98 (-14%)	
$\gamma = 1$	38.84 (+16%) 109.48 (+3%) 153.80 (+6%)	24.46 (-26%) 108.94 (+3%) 138.88 (-3%)	12.94 (-61%) 110.84 (+4%) 129.29 (-10%)	6.89 (-79%) 107.73 (+1%) 119.95 (-16%)	256MB
$\gamma = 2$	46.16 (+38%) 107.12 (+1%) 158.60 (+9%)	28.66 (-13%) 109.38 (+3%) 143.50 (0%)	14.87 (-55%) 105.92 (0%) 126.12 (-12%)	7.99 (-75%) 106.29 (0%) 119.63 (-17%)	128MB
$\gamma = 3$	63.04 (+89%) 108.92 (+3%) 179.08 (+24%)	35.22 (+5%) 109.98 (+4%) 152.66 (+5%)	19.37 (-41%) 109.46 (+3%) 134.31 (-6%)	10.18 (-69%) 105.58 (0%) 121.11 (-16%)	64MB
$\gamma = 4$	96.56 (+190%) 108.68 (+2%) 210.72 (+46%)	54.56 (+63%) 111.26 (+5%) 171.28 (+18%)	28.27 (-15%) 110.42 (+4%) 144.14 (0%)	14.69 (-55%) 106.84 (+1%) 126.87 (-12%)	32MB
	2MB	4MB	8MB		16MB

Figure 3.9 – Influence de γ et ϵ sur le temps de crible

On s'attend à ce que la dimension de l'espace directeur de $\mathbb{S}$ soit $\delta_B + 1 + \epsilon - \deg g$. Nous allons voir que calculer quels sont les points de $\mathbb{S}$ peut se faire aisément, même lorsque ϵ croît. Les opérations qui nous sont utiles sont des décalages arithmétiques et des opérations logiques sur les bits.

Faisons la supposition que $\deg g \leqslant \delta_B + 1$. Un point de base de $\mathbb{S}$ est le point s_0 donné par :

$$s_0 = (AX^h + B_f X^{\delta_B + 1}) \bmod g.$$

On identifie aisément $\delta_B + 1 - \deg g$ vecteurs de l'espace directeur $\mathbb{S}$' de $\mathbb{S}$, qui sont les $X^i g$ pour $0 \leqslant i \leqslant \delta_B - \deg g$. On obtient aisément ϵ autres vecteurs, une fois que l'on a calculé la valeur de $X^h \bmod g$ (et cette valeur est une valeur intermédiaire du calcul de s_0) : ce sont les $X^{h+i} + (X^{h+i} \bmod g)$, pour $0 \leqslant i < \epsilon$. Ces vecteurs sont faciles à calculer car le passage de i à $i + 1$ fait simplement intervenir un décalage arithmétique, et éventuellement une opération XOR (ou-exclusif).

Le coût de l'initialisation du crible en fonction de ϵ est donc presque invariant avec ϵ. Il est de l'ordre d'une réduction modulaire, et quelques opérations comparablement triviales. Ce coût est à comparer aux 2^ϵ réductions modulaires qui doivent être effectuées si l'on fait 2^ϵ cribles distincts.

On peut similairement calculer aisément les données d'initialisation du crible lorsque $\deg g > \delta_B + 1$, mais ce cas est de peu d'intérêt, car le polynôme g n'atteint pas de telles valeurs. La description du procédé général est faite dans [Tho01b].

3.5.5 Influence combinée des deux effets

Nous avons montré que l'on pouvait « jouer » avec la table de crible de deux façons. Il est intéressant de voir que la combinaison des deux effets permet des gains de temps. En effet, l'amortissement du coût d'initialisation (en augmentant ϵ) combinée à la réduction de la table

de crible (en augmentant γ) réduit le temps final de calcul. Dans la figure 3.9, l'influence de ces effets a été consignée, pour les paramètres déjà mentionnés qui ont été utilisés pour le calcul sur $\mathbb{F}_{2^{607}}$. Ces mesures ont été effectuées sur un Pentium II à 450MHz. Pour chaque couple de valeurs (γ, ϵ), on a mentionné le temps d'initialisation du crible et le temps de remplissage de la table stockée en mémoire. La dernière donnée est le temps total (donc *a priori* la somme des deux, aux imprécisions de mesure près). Bien entendu, la taille mémoire de la table de crible varie en fonction de γ et ϵ (d'où les flèches diagonales). Pour pouvoir comparer des grandeurs similaires, on a donc ramené les temps aux temps cumulés nécessaires pour traiter une taille mémoire (fictive) de 128Mo. Pour les cas où $\gamma \neq 0$, on a pris en compte le temps nécessaire pour relire les données précalculées.

3.6 Le crible par réseau

Une autre technique de recherche de relations consiste à forcer un facteur irréductible Q dans la factorisation de C (par exemple – on peut aussi prendre D), et cribler sur l'espace des polynômes (A, B) tels que ce facteur est présent. Pour que cette méthode présente un intérêt, il convient de prendre Q en dehors de la base de facteurs. On peut par exemple prendre un polynôme irréductible de degré $b + 1$ quelconque. Avant d'aborder les points techniques liés à cette méthode, mentionnons ses avantages les plus apparents. Tout d'abord, le fait de forcer un facteur dans la factorisation d'un des polynômes a un effet immédiat : c'est seulement à la friabilité du cofacteur restant que l'on s'attache. Celui-ci est de degré moindre, sa probabilité d'être friable est donc plus importante. Par ailleurs, un second point intéressant de cette méthode réside dans le fait que deux choix distincts pour Q amènent deux ensembles de relations totalement disjoints[15], et où les probabilités de friabilité sont les mêmes. Il en résulte, puisque l'on a *a priori* une très grande liberté pour choisir[16] le polynôme Q, que l'emploi de cette méthode amène des probabilités de friabilité beaucoup plus stables.

Cette méthode a pris le nom, à l'origine de *special-Q sieving* [Odl85, DH84]. Sa difficulté principale réside dans l'expression efficace du nouvel espace de polynômes (A, B). À cet effet, l'introduction de réseaux par Pollard [Pol93] a rendu la méthode très compétitive. Depuis lors, cette méthode est plus habituellement référencée en tant que *lattice sieving*. Cette méthode est très utilisée avec le crible algébrique. Elle a été employée par [JL02] avec l'algorithme FFS, et peut aussi s'employer avec l'algorithme de Coppersmith.

Nous souhaitons donc, étant donné un polynôme Q, exprimer simplement l'ensemble des polynômes (A, B) tels que :

$$AX^h + B \equiv 0 \bmod Q.$$

Cette congruence est la même que celle que nous avons eu à résoudre pour l'introduction du crible « simple » un peu plus haut, à la différence notable près que le polynôme Q n'est plus désormais un petit polynôme mais un polynôme de taille moyenne, et que l'on ne projette pas d'employer cette congruence de la même façon.

L'espace des paires (A, B) que nous étudions est bien entendu un espace vectoriel sur $\mathbb{F}_2$. Mais mieux que cela, si l'on lève la condition de degré sur A et B, on voit que cet espace est

[15] Cela reste vrai tant que l'on n'autorise pas de *large primes* appartenant à l'ensemble de polynômes dans lequel Q est choisi.

[16] Si l'on se restreint aux polynômes de degré $b + 1$ par exemple, ce qui n'a rien de nécessaire, on a déjà à peu près autant de choix possibles pour Q que d'éléments dans la base de facteurs.

aussi un réseau de $\mathbb{F}_2[X]^2$ (c'est-à-dire un $\mathbb{F}_2[X]$-module de rang maximal dans $\mathbb{F}_2[X]^2$). À ce titre, ce réseau a une base naturelle donnée par :

$$u = (1, X^h), \qquad v = (0, Q).$$

On veut transposer la terminologie connue pour les réseaux réels à notre situation (où $\mathbb{F}_2[X]$ joue le rôle de $\mathbb{Z}$). Pour cela, il faut définir une norme sur les polynômes ainsi que sur les vecteurs de $\mathbb{F}_2[X]^2$. La norme choisie est naturellement la norme de Dedekind :

$$\|P\| = \#\left(\mathbb{F}_2[X]/P\right) = 2^{\deg P}.$$

On munit ensuite $\mathbb{F}_2[X]^2$ de la norme L^2. Dans ce contexte, les normes des vecteurs u et v ainsi que le volume du parallélogramme formé par u et v sont donnés par :

$$\begin{aligned} \|u\| &\approx 2^h, \\ \|v\| &= 2^{\deg Q}, \\ \|\det(u, v)\| &= 2^{\deg Q}. \end{aligned}$$

Il s'ensuit comme dans le cas réel que nous avons bien une base du réseau. Bien entendu, cette base n'est pas convenable pour effectuer des calculs, car les normes des vecteurs u et v sont trop importantes. Des combinaisons $\mathbb{F}_2[X]$-linéaires petites de u et v donnent en particulier des polynômes B de grande taille[17], ce qui n'est pas souhaitable. Il est important de trouver une base du réseau avec des vecteurs *plus courts*. Idéalement, on souhaiterait pouvoir trouver des vecteurs « presque orthogonaux », c'est-à-dire tels que $\|\det(u, v)\|$ soit aussi proche que possible du produit $\|u\|\,\|v\|$. Deux vecteurs de norme avoisinant $2^{\frac{1}{2}\deg Q}$ constitueraient donc une base bien meilleure. L'algorithme classique de réduction de réseaux de Gauss en dimension deux permet de résoudre ce problème. Son adaptation au cas polynomial est aisée. On la trouve par exemple dans [JL02]. Le pseudo-code 3.10 illustre le procédé, et donne l'idée d'une implantation possible en langage Magma.

En produisant une base courte avec l'algorithme du programme 3.10, on obtient *en moyenne* deux vecteurs dont les entrées sont chacune de degré $\frac{1}{2}\deg Q \pm \epsilon$, où ϵ est de l'ordre de 1. Il en résulte que les vecteurs de $\mathbb{F}_2[X]^2$ (c'est-à-dire les paires (A, B)) de la forme $\alpha u + \beta v$, pour α et β deux polynômes de degré $\leqslant d - \frac{1}{2}\deg Q$, correspondent à des vecteurs A et B de de degré $\leqslant d + \epsilon$. Les différentes paires (A, B) de degré d sont obtenues à partir d'un espace de polynômes (α, β) de cardinal $2^{2d+2-\deg Q}$, ce qui est optimal (à ϵ près).

3.7 Stratégies de factorisation des relations

3.7.1 Particularités du problème posé

La première étape consiste à sélectionner les relations qui ont de bonnes chances d'être friables, en employant par exemple les techniques de crible décrites plus haut. Une fois ce premier tri effectué, il reste à *factoriser* les polynômes ainsi sélectionnés. Pour cette étape, il convient de savoir quelle est la quantité d'information disponible lorsque l'on doit ainsi factoriser une paire de polynômes (C, D). Dans le cadre des calculs qui ont été effectués, les considérations suivantes doivent être prises en compte.

[17] Bien sûr, on peut commencer par réduire u en remplaçant X^h par $X^h \bmod Q$, mais cela n'est pas suffisant.

Algorithme **GaussReducePoly**
Entrée : Deux vecteurs u et v de $\mathbb{F}_2[X]^2$
Sortie : Une base courte du même réseau

```
if ‖u‖ > ‖v‖ then return GaussReducePoly(v,u); end if;

d1:=Max(Degree(v[1])-Degree(u[1]),0);
d2:=Max(Degree(v[2])-Degree(u[2]),0);
w^(1):=<v[1]-X^d1 * u[1],v[2]-X^d1 * u[2]>;
w^(2):=<v[1]-X^d2 * u[1],v[2]-X^d2 * u[2]>;
if ‖w^(1)‖ > ‖w^(2)‖ then w:=w^(2); else w:=w^(1); end if;

if ‖w‖ < ‖v‖ then return GaussReducePoly(w,u); end if;

return u,v;
```

Programme 3.10: Réduction de réseaux de $\mathbb{F}_2[X]^2$ en dimension 2

- Le polynôme C a été identifié comme ayant une probabilité de friabilité importante[18]. En revanche, le polynôme D n'a pas de probabilité particulièrement plus élevée que la moyenne d'être friable.
- Quand bien même un crible a pu être effectué pour sélectionner la paire (C, D) en fonction de la probabilité de friabilité de C, il est exclu de conserver la liste des facteurs identifiés du polynôme C. Cette donnée serait trop encombrante, et ralentirait considérablement le crible.
- Les polynômes que l'on cherche à factoriser sont de degré plutôt modeste (ce degré n'a jamais dépassé 200 dans les calculs menés), et les facteurs que l'on veut mettre en évidence dans leur factorisation sont petits (quelques dizaines au plus dans le cas où des *large primes* sont autorisés).

Afin de minimiser le travail, il est préférable dans cette étape de factorisation de commencer par les calculs qui ont le plus de chances de ne pas aboutir, afin de n'effectuer que le minimum de travail inutile. En particulier, comme le polynôme D n'a pas été sélectionné pour être friable, on peut s'attendre à ce qu'une partie très importante des paires à traiter soit rejetée au motif que D n'est pas friable. Par conséquent, la première des choses à faire est un test de friabilité pour D. Ce test, tel qu'on l'a décrit en 3.2, est beaucoup plus rapide qu'une factorisation complète du polynôme (il n'y a notamment pas de pgcd à calculer). Une fois ce test réussi, on peut lancer une batterie d'algorithmes de factorisation.

Il est important de noter ici la chose suivante. Comparé à l'ensemble colossal de paires traitées par un crible (y compris même pour une « tranche » de crible), seul un petit nombre *a priori* passe le crible avec succès. Parmi celles-ci, les paires ayant un polynôme D friable sont encore moins nombreuses, de telle sorte que ce n'est que très rarement que l'on a recours à la factorisation totale. Il s'ensuit que la vitesse de l'algorithme de factorisation employé n'est pas une composante capitale de l'efficacité du programme. Ou, plus exactement, c'est une composante dont l'importance décroît au fur et à mesure de l'évolution de la complexité des problèmes (qui fait que les paires friables sont de plus en plus rares). Nous avons considéré

[18] Dans le cas où le crible a été effectué sur le polynôme D, comme décrit en page 54, la situation est bien entendu inversée.

les algorithmes décrits dans les paragraphes qui suivent. Le choix entre ces algorithmes s'est avéré subtil pour des petites tailles de problèmes où, comparativement, un temps important était passé dans la factorisation. Lorsque l'on s'est concentré sur $\mathbb{F}_{2^{607}}$, il est apparu, pour les raisons que l'on vient d'évoquer, que cela n'avait plus une très grande influence.

3.7.2 L'algorithme de factorisation de Niederreiter

Pour des polynômes de taille moyenne définis sur $\mathbb{F}_2$ comme ceux qui nous occupent, l'algorithme de Niederreiter [Nie93b, Göt94] offre un niveau de performances intéressant. Il est assez semblable en structure à l'algorithme « classique » de Berlekamp décrit par exemple dans [Knu98, 4.6.2]. L'algorithme de Niederreiter s'appuie sur le résultat suivant :

Proposition 3.5. *Soit $P \in \mathbb{F}_2[X]$. Considérons l'équation :*

$$(HP)' = H^2.$$

Les polynômes $H \in \mathbb{F}_2[X]$ qui sont solutions de cette équation sont exactement les polynômes de la forme $\frac{P}{V}V'$, où V décrit l'ensemble des diviseurs de P sans facteurs carrés. En outre, deux tels polynômes H sont toujours distincts.

Démonstration. La preuve de ce résultat est donnée dans [Nie93a]. Soit H une solution quelconque de l'équation. Notons $U = \operatorname{pgcd}(P, H)$, et V et W les deux polynômes tels que $P = UV$ et $H = UW$. On a alors :

$$\begin{aligned} (UVUW)' &= (UW)^2, \\ VW' + V'W &= W^2, \\ W &\mid VW', \\ \text{Or, comme } \operatorname{pgcd}(V, W) = 1 : \quad W &\mid W', \\ \text{d'où il découle } W' = 0, \text{ et donc :} \quad V' &= W. \end{aligned}$$

Il s'ensuit que $U = \frac{P}{V}$, et donc H est de la forme annoncée. Comme les polynômes V et W, c'est-à-dire V et V', sont premiers entre eux, le polynôme V est sans facteurs carrés. Réciproquement, il est aisé de vérifier qu'un polynôme H ayant la forme annoncée satisfait l'équation.

Le fait que les polynômes H ainsi construits sont tous distincts tient au fait que, pour $H = \frac{P}{V}V'$, on a :

$$\operatorname{pgcd}(P, H) = \frac{P}{V} \operatorname{pgcd}(V, V') = \frac{P}{V},$$

d'où on conclut que la correspondance $V \leftrightarrow H$ est biunivoque. ■

Construction d'un système linéaire

On déduit de la proposition précédente le nombre de solutions de l'équation différentielle considérée. Si P se factorise sous la forme

$$P = F_1^{e_1} \dots F_m^{e_m},$$

alors le nombre de polynômes H distincts possibles est 2^m. Comme en outre l'espace des solutions de l'équation $(HP)' = H^2$ constitue un sous-espace vectoriel de l'espace des polynômes sur $\mathbb{F}_2$ de degré $\leqslant \deg P - 1$, on conclut que la dimension de cet espace est m. L'algorithme de Niederreiter procède en écrivant l'équation $(HP)' = H^2$ sous forme d'un système linéaire homogène défini sur $\mathbb{F}_2$ et en calculant une base de son espace de solutions.

Le système linéaire en question se récrit de la manière suivante, si l'on note $H = \sum_{i=0}^{n-1} h_i X^i$ et $P = \sum_{i=0}^{n} p_i X^i$:

$$\begin{cases} p_1 h_0 + p_0 h_1 & = h_0, \\ p_3 h_0 + p_2 h_1 + p_1 h_2 + p_0 h_3 & = h_1, \\ \qquad\qquad \dots & = \vdots \\ p_n h_{n-1} & = h_{n-1}. \end{cases}$$

Ceci revient à dire que le vecteur $h = (h_0 \dots h_{n-1})$ est solution du système $Mh = h$, où M est la matrice binaire suivante :

$$M = \begin{pmatrix} p_1 & p_0 & & & & \\ p_3 & p_2 & p_1 & p_0 & & 0 \\ & & & & & \\ & & & & & \\ 0 & & & & p_n & p_{n-1} \; p_{n-2} \\ & & & & & p_n \end{pmatrix}$$

Du point de vue de la programmation, la mise en place de la matrice M est très facile, et le calcul d'une base du noyau de M se fait rapidement (on tire parti de la représentation binaire, de telle sorte que les opérations de lignes se ramènent essentiellement à des opérations logiques sur les mots machines).

Séparation des facteurs

Une fois qu'une base de l'espace des polynômes H_i solutions de $(HP)' = H^2$ est créée, il faut en déduire la factorisation de P. Pour cela, on doit calculer un certain nombre de pgcd. Comme c'est une opération qui, en certains cas, peut s'avérer assez coûteuse, on essaie de restreindre autant que faire se peut le nombre de pgcd à calculer (ce nombre s'élève à m^2 si l'on applique l'algorithme tel que décrit dans [Nie93b] ou [Göt94]). Un procédé simple et néanmoins efficace peut passer par les étapes suivantes :

1. Calculer les $\dfrac{P}{\operatorname{pgcd}(P, H_i)}$. Ce sont les polynômes V_i. Soit la liste $L = \{V_i\}$, et $I = \{\}$.
2. Pour chaque *petit* polynôme V irréductible dans L, remplacer L et I par :
$$L' = \left\{ \frac{W}{V} \,\middle|\, W \in L,\ W \neq V,\ V \mid W \right\} \cup \left\{ W \,\middle|\, W \in L,\ V \nmid W \right\},$$
$$I' = I \cup \{V\}.$$
3. Soit $V \in L$ de degré minimal. Remplacer L par :
$$L' = \{V\} \cup \left\{ \operatorname{pgcd}(V, W) \,\middle|\, W \in L - \{V\} \right\} \cup \left\{ \frac{W}{\operatorname{pgcd}(V, W)} \,\middle|\, W \in L - \{V\} \right\}.$$

Si la liste L est alors inchangée, V est nécessairement irréductible. Transférer V de L vers I.

4. Reprendre à l'étape 2.

Il est aisé de montrer que ce processus termine, en considérant par exemple le nombre total de facteurs irréductibles des éléments de la liste L. Par construction, ce nombre est une quantité strictement décroissante. En outre, comme ce nombre reste à chaque étape borné par m, le procédé décrit prend un temps $O(m^2)$ (les opérations de gestion de liste étant négligeables[19]).

La composante particulière de la méthode exposée plus haut pour trouver les facteurs irréductibles par l'algorithme de Niederreiter est que l'on tire parti de notre capacité à détecter les *petits* facteurs irréductibles. En effet, maintenir une table des polynômes irréductibles de la base de facteurs est facile, et de toute façon nécessaire en divers endroits de l'algorithme de Coppersmith. Et comme les polynômes que l'on a à factoriser sont par construction riches en petits facteurs, nous accélérons ainsi le processus de factorisation. Expérimentalement, le nombre de pgcd effectués en employant cette méthode de retrait des facteurs est très faible (nettement inférieur à m^2).

Comparaison avec l'algorithme de Berlekamp

Par rapport à l'algorithme similaire de Berlekamp, l'algorithme de Niederreiter présente quelques avantages, mentionnés dans [Nie93a] :

- L'algorithme ne nécessite pas que le polynôme P soit sans facteurs carrés.
- Le coût de mise en place de la matrice M est très faible.
- Si le polynôme P est creux, la matrice M aussi.

Les deux premiers de ces avantages sont pertinents pour le cadre qui nous concerne. Toutefois, il existe un inconvénient très net à l'algorithme de Niederreiter : les contorsions par lesquelles on doit passer pour séparer les facteurs. Mais on vient de voir que cette étape peut être dans notre cas grandement facilitée, puisqu'un test d'irréductibilité pour les polynômes de petit degré peut être effectué très facilement.

3.7.3 La méthode SFF/DDF/EDF

Lorsque l'on parle de factorisation de polynômes, on ne peut éviter de mentionner toute une famille de méthodes passant successivement par les trois étapes suivantes.

SFF *Square-free factorization* Le polynôme est factorisé en produit de puissances de polynômes sans facteurs carrés.

DDF *Distinct-degree factorization* Chaque polynôme sans facteurs carrés est factorisé en un produit de polynômes dont tous les facteurs irréductibles sont de même degré.

EDF *Equal-degree factorization* Chaque polynôme dont tous les facteurs irréductibles sont de même degré est factorisé en produit de polynômes irréductibles.

Ce schéma général est partagé par les algorithmes les plus performants connus actuellement pour la factorisation de polynômes. Il est apparu avec l'algorithme de Cantor-Zassenhaus [CZ81]. L'étape coûteuse est en général la DDF. Dans la plupart des algorithmes avancés, c'est sur cette étape que se concentrent les efforts [vzGP01, vzGG99]. Ainsi, on peut atteindre

[19] Une implantation réelle n'emploie que des tableaux.

une complexité quadratique ou même inférieure en le degré du polynôme [vzGS92, Sho90, Sho95, KS98, vzGG02]. Ces algorithmes sont indiscutablement supérieurs aux algorithmes de Berlekamp et Niederreiter pour des polynômes de grand degré.

Dans notre cas, le degré est relativement faible. Dans les premières expériences que nous avons menées, il dépassait à peine la centaine et pour le calcul de logarithmes discrets sur $\mathbb{F}_{2^{607}}$ que nous avons effectué, ce degré est inférieur à 200. À titre de comparaison, [vzGG02] s'intéresse à la factorisation de polynômes de degré supérieur à 16 384.

Pour les degrés que nous avons rencontrés, il est possible que l'algorithme de Shoup par exemple [Sho90] commence à se montrer compétitif. Toutefois à ce stade, le coût de la factorisation de polynômes a cessé d'être critique, et nous n'avons pas ressenti la nécessité d'améliorer encore cette phase, jugeant l'emploi d'un procédé par DDF peu avantageux. On peut donner plusieurs explications.

Tout d'abord, les ingrédients essentiels des algorithmes asymptotiquement rapides de factorisation sont la multiplication rapide de matrices d'une part, et de polynômes d'autre part. Il est clair que les tailles que nous avons eu à traiter ne sont pas telles que les algorithmes les plus rapides asymptotiquement se montrent compétitifs en pratique.

Dans cette même optique, nous nous sommes davantage intéressés à la minimisation du nombre exact (et non asymptotique) de pgcd à effectuer, car ce nombre contrôle une bonne partie du coût de la factorisation. Pour cette raison, l'algorithme de Niederreiter a présenté dans notre cas le double avantage de ne pas nécessiter d'étape SFF préalable, et de permettre une réduction du nombre de pgcd par le procédé de séparation des facteurs décrit plus haut.

On peut remarquer néanmoins un intérêt spécifique de l'utilisation de la DDF dans notre cas. Pour un polynôme P, les quantités suivantes sont calculées au cours d'une DDF :

$$\begin{aligned} P_1 &= \mathrm{pgcd}(P, X^2 + X), \\ P_2 &= \mathrm{pgcd}(\frac{P}{P_1}, X^{2^2} + X), \\ &\dots \\ P_d &= \mathrm{pgcd}(\frac{P}{P_1 P_2 \dots P_{d-1}}, X^{2^d} + X), \\ &\dots \end{aligned}$$

La similitude de ces calculs avec ceux effectués lors du test de friabilité évoqué en 3.2 est bien sûr frappante. Si une méthode par DDF était pertinente pour notre problème, et si la tâche de factorisation correspondante était d'importance critique (mais on a vu que ce n'était pas réellement le cas), le recyclage partiel d'informations s'avèrerait vraisemblablement intéressant entre le test de friabilité et la DDF.

3.8 Détermination de logarithmes individuels

Nous décrivons ici les aspects algorithmiques du calcul des logarithmes individuels qui peut être mené une fois que les logarithmes des éléments de la base de facteurs sont connus. Nous avons rapidement décrit page 30 la méthode employée, proposée par Coppersmith [Cop84]. Nous entrons ici dans les détails plus précis, notamment en vue de l'utilisation pratique.

3.8.1 Complexité réelle et pratique

Nous avons vu page 33 que pour des calculs dans $\mathbb{F}_{2^n}$, une fois la base de facteurs obtenue, la complexité du calcul était sous-exponentielle, de la forme :

$$\mathsf{W}_3 = L_n\left(\frac{1}{3}, \left(\frac{4\log 2}{9}\right)^{\frac{1}{3}}\right) = L_n\left(\frac{1}{3}, 0.67\right).$$

Cette complexité est nettement inférieure à la complexité des autres étapes de l'algorithme. Par conséquent, les calculs qu'elle nécessite sont de bien moindre ampleur que ceux des deux premières phases, ce qui explique qu'on les considère comme négligeables lors de l'établissement de records [Cop84, BMV85, GM93, Tho01b]. Pour le calcul de logarithmes discrets dans $\mathbb{F}_{2^{607}}$, la détermination de logarithmes individuels peut se faire en une heure de calcul sur une machine du type d'un PC de bureau.

Lorsque l'on cherche à calculer des logarithmes discrets dans des corps de taille « moyenne » ($\mathbb{F}_{2^{127}}$ ou $\mathbb{F}_{2^{163}}$ par exemple), le problème est différent. La question n'est plus celle du caractère réalisable ou non de la cryptanalyse, mais celle de l'utilisabilité de ce genre de primitive, par exemple dans un système de calcul formel comme Magma. Pour l'utilisation en calcul formel, « faisable » peut dans certains cas signifier un calcul de moins d'une minute. Notre objectif est de détailler, dans la perspective de l'usage pour des corps de taille moyenne, comment ce calcul peut être optimisé (bien que l'on n'échappe pas, bien entendu, à la nature sous-exponentielle du calcul).

Les différents développements que nous donnons ici ont fait (pour certains) et feront l'objet (pour les autres) d'une incorporation dans le logiciel de calcul formel Magma, destinée à rendre plus efficace cet aspect. Il ne s'agit toutefois que d'un ensemble de pistes.

3.8.2 Nature du problème

Le problème auquel nous sommes confrontés peut se résumer de la manière suivante. Nous devons, étant donné un élément de $\mathbb{F}_{2^n}$ représenté par un polynôme P, trouver le logarithme discret de P en base X (si X est primitif). Plus exactement, on souhaite exprimer le logarithme discret de P sous la forme d'une combinaison linéaire d'un ensemble de logarithmes connus (les π_i sont les éléments de la base de facteurs, à savoir les polynômes de degré $\leqslant b$) :

$$\log P \equiv \sum c_i \log \pi_i \mod (2^n - 1).$$

Présenter notre objectif sous cette forme a l'avantage de permettre de travailler simplement avec la notion de logarithme connu ou inconnu, en faisant abstraction dans un premier temps des valeurs des $\log \pi_i$: le travail ne commence pas le jour où le système linéaire provenant de la phase de recherche de relations est résolu. On peut commencer à chercher une « décomposition » de P avant.

Déterminer une telle expression ne se fait pas en une seule étape. Chercher « brutalement » à décomposer PX^m, pour m aléatoire, sous forme d'un produit des π_i est l'approche de l'algorithme d'Adleman, et on sait que cet algorithme est en $L_n\left(\frac{1}{2}, c\right)$: on aurait alors la désagréable surprise de constater que cette troisième phase de l'algorithme devient le facteur limitant (la borne b étant même inférieure ici à sa valeur dans l'algorithme d'Adleman, on aurait en fait une complexité en $L_n\left(\frac{2}{3}, c\right)$).

L'approche est plus fine. Elle consiste à obtenir des décompositions *partielles*, sous forme de combinaisons linéaires de logarithmes de polynômes de degrés décroissants. En présentant

les choses autrement, on peut aussi dire que l'on cherche à exprimer P comme un produit d'éléments de plus en plus petits (mais de plus en plus nombreux). La structure de donnée principale ainsi manipulée est donc une liste de polynômes dont le produit (comme éléments de $\mathbb{F}_{2^n}$) est égal à P. Cette liste est munie d'une hauteur : le degré maximal des polynômes qui la composent, et elle a bien entendu sa longueur. Si l'on sait estimer la difficulté d'expression du logarithme d'un polynôme dans cette liste, on peut aussi donner un *poids* à la liste, en prenant la somme de ces estimations.

Au fur et à mesure du calcul, on veut baisser progressivement la hauteur de la liste. Cela implique de remplacer chacun des éléments de la liste par un produit d'éléments plus petits. Deux facteurs rendent ce calcul, dans sa globalité, difficile.

- Si l'on veut aller *trop vite* vers les petites tailles, alors la probabilité de friabilité est le facteur limitant. C'est ce qui se passe avec l'algorithme d'Adleman.
- Si au contraire on descend *trop lentement*, par exemple si la hauteur descend de un en un, alors le calcul devient exponentiel, puisque la taille de la liste croît exponentiellement (chaque membre de la liste étant remplacé par plusieurs membres de taille à peine inférieure).

Notre objectif est d'équilibrer ces deux aspects. Une partie du travail fait ici est une précision du travail d'analyse de la complexité réalisé page 32.

3.8.3 Première décomposition : l'algorithme d'Euclide

La première des transformations à appliquer à notre liste est celle consistant à employer l'algorithme d'Euclide. Plus exactement, prenons m quelconque et utilisons l'algorithme d'Euclide étendu (on le décrira en détail en 8.2.2, avec une version sous-quadratique et un exemple de programme) pour décomposer PX^m sous la forme :

$$PX^m \equiv \frac{P_1}{P_2} \quad \text{dans } \mathbb{F}_{2^n}.$$

On fait plusieurs essais de décompositions de cette forme, jusqu'à en trouver une qui nous satisfasse. On cherche à obtenir de cette façon des polynômes P_1 et P_2 *friables*, b_1-friables pour être précis, pour une certaine borne b_1. Cette borne conditionne la hauteur de la liste qui résulte.

Comment doit-on choisir b_1 ? Si l'on pose $b_1 = \frac{n}{2s}$, les probabilités de friabilité sont telles qu'il faut s'attendre à essayer s^{2s} décompositions avant d'en trouver une « bonne ». Nous avons vu que la complexité asymptotique de la troisième phase de l'algorithme était en $L_n\left(\frac{1}{3}, c\log 2\right)$. Nous allons donc voir comment choisir b_1 pour que le temps passé dans cette première décomposition représente une fraction non nulle du temps total de la troisième phase. Ainsi, on mesure le niveau d'exigence à avoir sur la hauteur du résultat de cette décomposition.

Pour obtenir une complexité de la première décomposition en $L_n\left(\frac{1}{3}, c\log 2\right)$, on prend pour b_1 une fraction de la moyenne géométrique de b et n, notée sous la forme $\zeta\sqrt{nb}$. En effet, si l'on reprend les notations de l'analyse faite en 2.4.2, le facteur de travail w_3^0 pour obtenir une décomposition b_1-friable est alors donné par :

$$\begin{aligned}\log \mathsf{w}_3^0 &\sim \frac{n}{b_1}\log\frac{n}{2b_1},\\ &\sim \frac{1}{\zeta}\sqrt{\frac{n}{b}}\log\sqrt{\frac{n}{b}},\end{aligned}$$

$$\sim \frac{1}{\zeta}\frac{\log 2}{2}b \qquad \text{(cf page 141)},$$
$$\mathsf{w}_3^0 \overset{\log}{\sim} L_n\left(\frac{1}{3}, \frac{\log 2}{2\zeta}\right).$$

On peut par conséquent se permettre de prendre ζ de l'ordre de $\frac{1}{2c} \approx 0.52$. La première borne de friabilité à prendre, si l'on veut placer une bonne partie du travail sur cette première décomposition, est :

$$b_1 \sim \frac{1}{2c}\sqrt{nb}.$$

Il se trouve que dans la pratique, on ne choisit pas nécessairement $\zeta = \frac{1}{2c}$, car l'emploi des techniques de crible et de *special-Q descent* décrites ci-après peut donner des performances meilleures. Cela est dû à la composante polynomiale non écrite des complexités, ainsi bien entendu qu'aux facteurs d'implantation.

3.8.4 Seconde décomposition : la descente par *special-Q*

Nous reprenons ici de façon détaillée le processus de *refriabilisation* esquissé pages 30–31. Ce processus nous fournit un moyen de décomposer un polynôme Q de degré q en un produit de polynômes de degré plus petit. Ainsi, toujours en ayant en tête notre type de données général sous forme d'une liste de polynômes, on peut faire décroître la hauteur de la liste.

Nous avons présenté des méthodes permettant d'obtenir des relations faisant intervenir Q : elles sont similaires aux méthodes employées lors de la première phase de l'algorithme pour produire des relations. Plus spécifiquement, les méthodes de crible par réseau vues en 3.6 sont parfaitement adaptées, à la différence près qu'il faut choisir les paramètres d', h', k' en fonction de q. Le paramètre d' donne le degré maximal des paires (A, B) considérées. Si l'on veut raisonner en termes de coefficients dans la base du réseau associé, ces coefficients ont un degré maximal de $d' - \frac{q}{2}$, puisque les vecteurs de base du réseau sont de degré $\frac{q}{2}$ en moyenne. Nous posons $d' = zq$. Le choix de k' et h' s'en déduit, puisque l'on prend pour k' une puissance de 2 proche de $\sqrt{\frac{n}{d'}}$, et $h' = \left\lceil \frac{n}{k'} \right\rceil$. Dans le cas le meilleur, on obtient ainsi une paire (C, D) équilibrée.

Nous cherchons à obtenir une relation $\sqrt{bq}$-friable. Nous avons vu que les contraintes pour qu'une telle relation existe dans le réseau considéré nous imposaient, pour $b = xq$, l'équation :

$$2d' - q \geqslant \sqrt{\frac{nd'}{bq}}\log_2\frac{nd'}{bq},$$
$$(2z - 1) \geqslant \sqrt{z}x.$$

La plus petite valeur de z satisfaisant cette relation s'obtient en résolvant l'équation associée, ce qui donne :

$$z = \frac{1}{2}\left[\frac{x}{2\sqrt{2}} + \sqrt{1 + \frac{x^2}{8}}\right]^2,$$
$$\overset{\text{déf}}{=} \phi(x).$$

Par rapport au choix « simple » $z = \frac{1+x}{2}$, qui donne $d' = \frac{1}{2}(q + b)$ (soit le choix fait par Coppersmith), l'avantage ici est que z est parfois plus petit. Comme z conditionne la complexité de cette étape de réduction (qui est en $L_n\left(\frac{1}{3}, \sqrt{z}c\log 2\right)$), on obtient un gain dans

l'exposant. Ce gain est toutefois faible, puisqu'il est de l'ordre de 3% (dans l'exposant) pour le cas optimal où $x = \frac{b}{q} = 0.43$. Le gain n'est pas visible lorsque q est très grand devant b (ce qui n'arrive pas dans la pratique), ni non plus quand q est très proche de b. Par conséquent, les décompositions les plus difficiles sont celles qui consistent à passer du degré $b+1$ au degré b, avec une complexité en $L_n\left(\frac{1}{3}, c\log 2\right)$.

On peut se demander si le choix de $\sqrt{bq}$ peut être amélioré. En vérité, ce n'est pas possible, puisque si l'on remplace $\sqrt{bq}$ par $\alpha\sqrt{bq}$ avec $\alpha \leqslant 1$, alors le terme $z = \phi(x)$ dans l'expression de la fonction L devient $\frac{\phi(x/\alpha)}{\alpha^2}$, qui est supérieur à $\phi(x)$ puisque ϕ est croissante. Si l'on recherche une optimisation du même type pour $\alpha \geqslant 1$, alors la contrainte sur α est $\alpha \leqslant \frac{1}{\sqrt{x}}$ (car sinon la hauteur de la liste de polynômes ne baisse pas), ce qui implique qu'aucun gain ne peut être obtenu au point limite où $x = 1$. Par conséquent, la complication associée n'est sans doute pas nécessaire.

La taille de l'espace de crible à considérer pour chaque pas de refriabilisation est de l'ordre de 2^b, soit à peu près le cardinal de la base de facteurs. La gestion du calcul ne pose donc pas de contraintes trop fortes.

Chapitre 4

Record de logarithmes discrets : $\mathbb{F}_{2^{607}}$

Nous exposons rapidement dans ce chapitre de quelle façon nous sommes parvenus à calculer des logarithmes discrets dans $\mathbb{F}_{2^{607}}$.

4.1 Travaux antérieurs

Le travail faisant état de référence en la matière lorsque l'on a commencé à travailler sur $\mathbb{F}_{2^{607}}$ était celui mené par Gordon et McCurley [GM93], il y a dix ans. En utilisant l'algorithme de Coppersmith, ils sont parvenus à calculer des logarithmes discrets dans $\mathbb{F}_{2^{401}}$, et ont construit le système linéaire correspondant à un calcul dans $\mathbb{F}_{2^{503}}$, sans toutefois pouvoir le résoudre[1].

À la toute fin du calcul sur $\mathbb{F}_{2^{607}}$, Joux et Lercier ont annoncé avoir calculé des logarithmes discrets dans $\mathbb{F}_{2^{521}}$ [JL01], en utilisant la méthode du *Function field sieve (FFS)*, que nous avons décrite en 2.5. Cette approche est prometteuse, et nous revenons sur les perspectives qu'elle ouvre en 4.7

4.2 Paramètres

La première des décisions à prendre pour le déploiement d'un calcul d'une telle envergure est le choix des paramètres. Nous avons détaillé la logique qui dirigeait ces choix en 2.4.3, page 34. En tout premier lieu, on doit choisir le polynôme de définition, sous la forme $X^{607} + f_1$. Page 35, nous avons expliqué comment le choix de f_1 influait sur la performance de l'algorithme. Après examen des différentes valeurs possibles pour f_1, on a choisi de prendre :

$$\begin{aligned} f_1 &= X^9 + X^7 + X^6 + X^3 + X + 1, \\ &= (X+1)^2(X^2+X+1)^2(X^3+X+1). \end{aligned}$$

Ce choix s'explique par le fait que ce polynôme présente le double avantage d'être le candidat de plus petit degré, et de répondre particulièrement bien au critère développé pages 35–36.

Ensuite, nous avons choisi la borne de friabilité, que nous avons fixée à $b = 23$. La base de facteurs $\mathcal{B}$, par conséquent, est constituée de $\#\mathcal{B} = 766\,150$ polynômes irréductibles. Outre la bonne adéquation de ce choix avec les évaluations asymptotiques [Cop84, GM93] (si l'on pousse le développement asymptotique de la valeur optimale de b, on obtient 22.5), un point méritant une attention particulière ici est la limite de faisabilité en ce qui concerne l'algèbre linéaire. En effet, on n'aurait pas pu supporter une valeur supérieure de b, car cela aurait

[1] Ce système linéaire, qui est maintenant bien sûr à la portée des méthodes que nous exposons dans la partie II de ce mémoire, n'a jamais été résolu par la suite (D. Gordon, communication privée, octobre 2000).

impliqué la gestion d'un système linéaire deux fois plus gros, ce qui apparaissait hors de nos possibilités.

Les paramètres d_A et d_B ont été choisis en conséquence. Initialement, les premières mesures que nous avons menées nous ont laissé imaginer que $d_A = 20$ et $d_B = 24$ suffiraient. Mais en fait, nous avons dû réviser ce choix, et prendre plutôt $d_A = 21$ et $d_B = 28$.

Le paramètre k, dont la valeur optimale est $\sqrt{\frac{n}{d_A}}$, a été choisi égal à 4. Ce paramètre est contraint à être une puissance de 2, et pour cette raison la valeur 4 est assez éloignée de $\sqrt{\frac{n}{d_A}}$: le quotient de ces deux valeurs est 1.37 (le cas le « pire » étant un quotient valant $\sqrt{2}$).

La valeur de h, qui découle de celle de k, est 152.

4.3 Techniques de crible

Le choix des paramètres que l'on vient d'évoquer conditionne en particulier les degrés maximaux des polynômes C et D lors du crible, valant ici 173 et 112 (lorsque A et B sont de degré maximal). Pour obtenir le nombre voulu de relations, à savoir au moins 766 150, nous avons utilisé un crible polynomial tel qu'il a été décrit en 3.3. Pour accélérer ce crible, la technique de crible partiel décrite en 3.4 a été employée. En accord avec les calculs menés en 3.4, et en particulier les graphiques de la figure 3.8 page 58, nous avons ainsi choisi de ne considérer pour le crible que les facteurs irréductibles de degré compris dans l'intervalle $[\![10 \dots 23]\!]$. L'espace de crible à examiner, en fonction des degrés maximaux d_A et d_B qui ont été choisis, compte 2^{51} paires (A, B) (ou encore 2^{50} paires telles que $\text{pgcd}(A, B) = 1$).

Nous avons aussi utilisé la *double large prime variation*, telle qu'on l'a décrite en 3.1.2. Nous avons donc collecté trois types de relations.

- Les relations « complètes », ou ff (*full-full*), ne comportant pas de *large prime*.
- Les relations pf (*partial-full*), où la factorisation de l'un des polynômes C et D contient un *large prime*.
- Les relations pp (*partial-partial*), où deux *large primes* interviennent, soit dans l'une des deux factorisations de C ou de D, soit un dans chacune.

Les calculs de recherche de relations ont été menés en grande partie sur les machines du laboratoire LIX, de la direction des études de l'École polytechnique, et de l'UMS Medicis. Cet ensemble compe approximativement une centaine de machines, dont la machine « type » est un PC de type pentium II à 450MHz. Ces machines ont été mises à contribution pendant leur « temps libre », c'est-à-dire lorsqu'aucun autre processus ne tournait sur ces machines. À l'issue d'un temps de calcul de 19 000 années MIPS environ[2], soit un temps réel d'environ une année sur une centaine de machines, le nombre de relations collectées s'élevait à 61 279 542 en incluant les différents types de relations. Plus en détail, on avait exactement à l'issue de ce calcul 221 368 relations ff, 6 083 209 relations pf, et 61 058 174 relations pp.

Les relations pf et pp ont ensuite été recombinées pour produire 856 145 cycles, par le procédé que l'on a expliqué en 3.1.2, En additionnant ces relations recombinées aux relations ff qui étaient déjà disponibles, on a ainsi atteint le total de 1 077 513 relations entre les éléments

[2]Pour donner une estimation formulée en des termes plus proches de la réalité, 370 000 sous-tâches ont été effectuées, occupant chacune une heure en moyenne sur un PC à 450MHz. On a donc effectué comme calcul :

$$\frac{370\,000 * 3\,600 * 450}{86\,400 * 365} \approx 19\,000.$$

de la base de facteurs. Les données plus fines sur les différents cycles produits par la *double large prime variation* sont consignées dans les tables 3.2 page 48 et 3.3 page 49. On peut noter que le plus gros des cycles ainsi produits impliquait pas moins de 40 relations partielles, donnant une relation avec un nombre de termes dépassant 500. En excluant les cas extrêmes comme celui-ci, la densité moyenne des 766 150 relations les moins « lourdes » s'élevait à 67.7

4.4 Algèbre linéaire

La deuxième phase de l'algorithme a consisté à effectuer un calcul d'algèbre linéaire. Les algorithmes sous-jacents, ainsi que les détails de ce calcul, sont développés dans la partie II de ce mémoire.

4.5 Logarithmes individuels

Une fois que l'on a obtenu la solution de notre système linéaire, le 18 février 2002, on a pu entamer la *fin* du calcul.

D'abord, les différents logarithmes ainsi trouvés par la résolution du système linéaire ont été substitués dans toutes les relations dont nous disposions, pour déterminer ainsi tous les logarithmes qui pouvaient l'être[3]. De cette façon, 766 009 logarithmes ont été obtenus, ce qui signifie que notre imprécision est très faible : seulement 141 logarithmes ont été ainsi « oubliés » (parce qu'aucune relation ne les faisait intervenir). On a aussi utilisé le même mécanisme de substitution dans les relations partielles dont nous disposions, afin de déterminer autant de logarithmes que possibles parmi les polynômes de degré 24 et 25. On a pu en déterminer 80%.

Pour démontrer que nous pouvions calculer, avec les informations dont on disposait, n'importe quel logarithme dans $\mathbb{F}_{2^{607}}$, on a illustré la méthode sur un élément « aléatoire » de $\mathbb{F}_{2^{607}}$. On a choisi de prendre l'élément représenté par le polynôme P, représenté en binaire par les octets suivants (le tout premier bit correspondant au terme constant) :

```
0000000: 54 65 73 20 79 65 75 78 20 73 6f 6e 74 20 73 69
0000010: 20 70 72 6f 66 6f 6e 64 73 20 71 75 27 65 6e 20
0000020: 6d 27 79 20 70 65 6e 63 68 61 6e 74 20 70 6f 75
0000030: 72 20 62 6f 69 72 65 0a 4a 27 61 69 20 76 75 20
0000040: 74 6f 75 73 20 6c 65 73 20 73 6f 6c 65 69 6c 73
0000050: 20 79 20 76 65 6e 69 72 20 73 65 20 6d 69 72 65
0000060: 72 0a
```

Ces octets correspondent au deux premiers vers du poème de L. Aragon, *Les yeux d'Elsa* :

Tes yeux sont si profonds qu'en m'y penchant pour boire
J'ai vu tous les soleils y venir se mirer

En utilisant les techniques que l'on a décrites en 3.8, on a pu écrire le logarithme de P comme combinaison linéaire de 1010 éléments dont le logarithme était connu (des éléments de la base de facteurs, ainsi que des éléments de degré 24 et 25 dont on avait pu déterminer le logarithme). Ce calcul a pris quelques heures sur une machine de type alpha ev67, cadencée à 667MHz. Il nous a permis de déterminer le logarithme de P, qui vaut :

[3]Cette substitution est pertinente car au cours de la phase d'algèbre linéaire, certaines relations n'ont délibérément pas été considérées (par exemple parce qu'on les a considérées comme trop lourdes).

$$\begin{aligned}\log P = \quad & 478\,911\,461\,661\,946\,696\,753 \\ & 672\,487\,974\,955\,175\,947\,078\,100\,949 \\ & 897\,401\,737\,706\,214\,043\,974\,054\,397 \\ & 090\,373\,933\,613\,593\,697\,064\,947\,460 \\ & 160\,895\,949\,314\,765\,939\,949\,543\,387 \\ & 334\,053\,322\,259\,124\,498\,269\,177\,310 \\ & 650\,885\,248\,209\,789\,392\,038\,650\,635 \bmod 2^{607}-1.\end{aligned}$$

4.6 Comparaison avec les calculs précédents

Il n'est pas aisé de mener des comparaisons précises des efforts de calcul lors des différents records de calcul de logarithmes discrets dans $\mathbb{F}_{2^n}$. Nous donnons toutefois un ordre de grandeur de la puissance de calcul déployé par Gordon et McCurley en 1993. Leurs calculs ont permis d'obtenir suffisamment de relations pour résoudre des logarithmes discrets dans $\mathbb{F}_{2^{503}}$ (bien que le système linéaire associé n'ait jamais été résolu ensuite). Ce calcul a nécessité 88 jours sur les 1 024 processeurs d'une machine de type nCube-2. Chacun de ces processeurs a une vitesse d'environ 15 MIPS, si on en croit les documentations. On arrive ainsi à un effort de calcul d'environ 3 700 années MIPS. C'est donc beaucoup moins que les 19 000 années MIPS qui nous ont été nécessaires pour mener à bien le calcul dans $\mathbb{F}_{2^{607}}$. Toutefois, l'écart entre les deux efforts est largement compensé par l'écart de difficulté des deux problèmes. Calculer des logarithmes discrets dans $\mathbb{F}_{2^{607}}$ est au moins vingt fois plus dur que dans $\mathbb{F}_{2^{503}}$. Ces chiffres montrent que l'importance des améliorations théoriques et pratiques que nous avons apportées au processus de crible sont bien réelles.

4.7 Tailles pouvant être atteintes

Le calcul que nous avons mené est un calcul de grande envergure. En terme de temps de calcul, l'effort est deux fois plus important que celui qui a été nécessaire à la factorisation de RSA-155. On peut considérer que ces travaux représentent la limite des tailles pouvant aujourd'hui être atteintes par une approche *semblable* à la notre. Nous devons toutefois faire état de deux points précis.

Tout d'abord, nos travaux ne permettent que partiellement de répondre à la question de la capacité que pourrait avoir une institution gouvernementale pour s'attaquer à ce genre de problèmes. En novembre 2002, le record du plus gros supercalculateur mondial a été battu par le constructeur japonais NEC, avec la mise en service du *Earth Simulator*. La puissance de calcul d'un tel « monstre » est formidable : 40 téraflops, soit 40 000 milliards d'instruction par seconde. Utilisons-la comme ordre de grandeur. En considérant que chacun des 5 120 processeurs de ce supercalculateur, cadencés à 500MHz, a la puissance d'un PC à 500MHz[4], il faudrait seulement *trois jours* à ce supercalculateur pour effectuer les calculs que nous avons

[4]La puissance annoncée de chacun de ces 5 120 processeurs est de 8 gigaflops. L'équivalent-gigaflop d'un PC n'étant pas une donnée très claire, notre estimation est peut-être pessimiste d'un certain facteur, mais là n'est pas la question.

menés (en un an). Et le temps pour effectuer l'algèbre linéaire se compterait vraisemblablement en jours aussi[5]. Notre calcul peut donc être dans le domaine de la « routine », pour qui manipule une telle puissance de calcul. Si, munie d'une telle puissance, une institution veut calculer des logarithmes discrets, et y passer un peu de temps, elle peut sans doute atteindre des tailles de l'ordre de $\mathbb{F}_{2^{700}}$.

Le résultat obtenu par Joux et Lercier [JL01], en calculant des logarithmes discrets dans $\mathbb{F}_{2^{521}}$, est particulièrement prometteur. En effet, il démontre la portée pratique de l'algorithme FFS, qui est apparemment substantiellement plus performant que l'algorithme de Coppersmith. Des calculs dans $\mathbb{F}_{2^{607}}$ avec cet algorithme pourraient être effectués avec moins de calculs que ce que l'on a entrepris. Pour des calculs plus avancés, c'est certainement l'algorithme FFS qui devra être préféré à l'avenir.

4.8 De la gestion d'un calcul distribué : aspects techniques et sociologiques

4.8.1 Structure de la distribution des tâches

Parler d'un calcul distribué reste un discours vague tant que l'on n'a pas précisé les caractéristiques exactes des calculs menés. La première phase de l'algorithme de Coppersmith, phase de recherche de relations, a été effectuée de manière distribuée. Plus exactement, l'espace de crible (gigantesque, puisque 2^{51} paires ont été examinées) a été divisé en de nombreux paquets, comme cela a déjà été évoqué en 3.5.2. Le nombre de paquets ainsi considérés s'est élevé à 2^{19}, soit 562 144. L'estimation d'origine était que les quelques 16 384 premiers paquets suffiraient. Cette estimation était manifestement très largement erronée, et pour cette raison on a dû étendre l'espace de crible plusieurs fois : un coefficient de plus pour B, puis pour A, etc...

Le schéma global utilisé est un schéma maître-esclave. Un serveur central observe et distribue l'évolution du calcul sur les différents esclaves. Un esclave prenant part au calcul reçoit, en guise d'identification du travail qu'il doit accomplir, un numéro correspondant au paquet dont il a la charge. Cette information est donc relativement légère. Lorsque l'esclave a fini de cribler le paquet en question (l'ordre de grandeur du calcul pour un paquet est d'environ une heure sur un PC à 450MHz), il doit faire savoir au maître qu'il a fini le calcul, et recevoir en échange un nouveau numéro de paquet à cribler. Pour que l'esclave ne perde pas son temps avant la réception de ce nouveau numéro, le schéma prévoit qu'un esclave soit en charge de plusieurs paquets simultanément de sorte qu'il puisse toujours déterminer quel paquet cribler.

Courrier électronique

La première approche envisagée a été l'usage du courrier électronique comme moyen de communication entre le maître et les esclaves. Cette approche a déjà été employée, au moins en partie, par Lenstra et Manasse [LM90] pour factoriser des nombres de 100 chiffres avec le crible quadratique. Nous avons donc commencé par envisager un schéma où le serveur de courrier électronique du laboratoire LIX sert de maître pour le calcul, et reçoit les courriers électro-

[5] Nous verrons dans la partie II que l'algèbre linéaire ne se distribue pas autant que la recherche de relations. Aussi, sans avoir expérimenté la *crossbar* d'un site de calcul, il est difficile de donner une estimation de temps de calcul *a priori*.

niques en provenance des esclaves (nous avons utilisé pour cela le programme `procmail`). En réponse aux courriers des esclaves, le maître doit leur renvoyer des informations.

Un problème inhérent à cette méthode est que l'envoi d'information *vers* les esclaves a peu de chance de pouvoir être réalisé par courrier électronique, puisque chaque machine n'est pas un serveur de courrier électronique. Pour résoudre cet inconvénient, on a supposé que pour chaque machine prenant part au calcul, il était possible de trouver un serveur de courrier électronique quelque part qui partagerait une partition de disque dur avec l'esclave. Sur ce « serveur secondaire » est alors aussi installé un programme `procmail` qui met à jour un fichier lisible par l'esclave.

Cette approche s'est avérée complètement impraticable à l'échelle qui nous concernait, pour plusieurs raisons.

- La vétusté de l'installation informatique du serveur de courrier électronique principal, sur lequel des problèmes permanents de *lock* ont été apparents. Il a été par conséquent impossible d'utiliser l'installation de courrier électronique de façon fiable, puisque régulièrement le calcul reprenait au paquet numéro 1.
- Communiquer de cette façon avec des machines trop distantes d'un serveur de courrier électronique s'est avéré difficile, à nouveau à cause de problèmes de *lock*.
- Les très nombreux courriers électroniques générés se perdent parfois en route, mais atterrissent toujours quelque part, éventuellement dans une file d'attente sur une machine plus ou moins aléatoire. On a vu arriver ainsi des « trains » de plusieurs milliers de courriers électroniques perdus avec des mois de retard.
- Une boîte de courrier électronique est un instrument de travail sensible, et faire rentrer ainsi dans le circuit de gestion du courrier des *scripts* infernaux tendait à rendre ce circuit un peu trop fragile (perdre des courriers électroniques n'est pas agréable).

Cette expérience nous a apporté divers enseignements.

- La première erreur que l'on a souhaité corriger a été le recours aux partitions de disque dur partagées par réseau. Le système de fichiers `NFS` associé a indéniablement de nombreux avantages, mais aussi des défauts[6] qui deviennent des cauchemars dans un environnement ou les clients et les serveurs constituent un parc de machines complètement hétérogène. Par conséquent, le remplacement par une solution utilisant uniquement l'arborescence de fichiers locale `/tmp` a été recherché.
- L'emploi d'intermédiaires pour la communication avec les esclaves multiplie par trop le nombre de machines du bon fonctionnement desquelles le résultat dépend.
- Envoyer un courrier électronique n'est pas une opération anodine. Outre l'aspect déjà mentionné, à savoir que des courriers peuvent rester bloqués dans une file d'attente de manière inattendue, la manipulation de programmes fonctionnant par envoi automatique de courriers électroniques est vue comme l'une des plus noires faces du piratage par certains sites informatiques. Envoyer dix mille courriers de façon automatique, même à un rythme relativement peu élevé, n'est pas une bonne façon de se faire des amis.

Utilisation d'un serveur autonome

Pour toutes les raisons que l'on vient d'évoquer, qui ont constitué autant d'embûches pour nos premiers essais de calcul distribué de logarithmes discrets, on a choisi pour le calcul

[6] « *There are many infelicities in the protocol underlying NFS* », peut-on lire dans plusieurs pages de manuel dont `open(2)`, sur les distributions Linux. C'est très vrai !

sur $\mathbb{F}_{2^{607}}$ de partir sur une base plus saine, constituée d'un serveur autonome. On a ainsi programmé en `perl` un *démon* simple (tout de même 500 lignes de programme), écoutant les connexions réseau sur un port bien déterminé. Ce démon assume la tâche de maintenance de la liste des paquets à distribuer. Contrairement à la plupart des démons, il ne répond pas aux requêtes de manière *asynchrone* mais *synchrone*. Cela a l'avantage d'éconduire la plupart des problèmes de *lock* concernant le serveur. C'est par ailleurs la bonne approche à adopter étant donnée la trivialité des opérations à effectuer (lire ou écrire une ligne dans un fichier).

Le programme gère aussi l'historique du calcul, en maintenant d'une part une liste des paquets dont le résultat a été calculé jusqu'au bout (parce que l'esclave a réussi à informer le maître de l'achèvement de sa tâche), et surtout d'autre part une liste des paquets « sortis », distribués à des esclaves. De cette façon, il est possible de ne pas perdre trop de paquets « dans la nature » : le serveur considère qu'un paquet qui n'a pas été achevé depuis plusieurs jours peut être redistribué.

En dernier lieu, les structures employées par le serveur sont très simples (uniquement des fichiers texte), de telle sorte que des sauvegardes temporaires et des ajustements manuels sont toujours possibles.

Cette approche a permis à l'entreprise de calcul distribué de logarithmes discrets de passer à l'échelle. Néanmoins, quelques difficultés sont apparues. Tout d'abord, le modèle de communication entre le serveur et les esclaves (par connexion TCP) s'est heurté aux politiques des « pare-feu » (*firewall*) utilisées par certains sites. Effectuer une connexion TCP devient presque, de nos jours, une acte de piratage caractérisé. Pour y parvenir malgré tout, des relais (*socket bouncer*) ont été installés pour servir de *proxys* aux points stratégiques. Fort heureusement, cela s'est toujours avéré possible.

Une fonctionnalité que notre mini-serveur en `perl` n'offre pas est la vérification active du fonctionnement des esclaves. Se connecter manuellement ou même de façon semi-automatique à 120 machines pour vérifier ce qu'elles font n'est pas facile. On aurait souhaité faire assumer cette tâche par le serveur, ou du moins avoir un regard plus global sur l'avancement général du calcul. En termes d'efficacité, ce léger flou s'est traduit par d'assez nombreux paquets (10%) perdus malgré toutes les précautions prises, en partie à cause des raisons spécifiques à la gestions des esclaves, que nous détaillons en 4.8.2.

Outils génériques de calcul distribué

Aujourd'hui, la pierre d'angle de bon nombre de calculs distribués est la bibliothèque MPI, destinée aux communications entre processus [MPI]. Cette bibliothèque a pris le relais de [PVM]. De nombreux outils d'optimisation et de diagnostics existent pour les applications utilisant ces bibliothèques. Pour les calculs distribués que nous avons menés, nous avons néanmoins préféré notre approche « maison » à l'emploi de ces bibliothèques. La raison principale de ce choix est l'importance de la tolérance aux déconnexions des différentes machines prenant part au calcul, ainsi que la résistance à l'existence de machines « pare-feu » (*firewall*) séparant les différents réseaux utilisés. Par ailleurs, on a estimé que le gain éventuel représenté par l'usage de MPI, comparé à la place peu critique tenue par les communications dans ce calcul, ne justifiait pas pleinement l'emploi d'une bibliothèque extérieure.

4.8.2 Gestion des nœuds de calcul

Dans un calcul distribué, il faut savoir gérer les nœuds de calcul, ici appelés esclaves. C'est facile quand le nombre d'esclaves est de l'ordre de la dizaine. Quand il est de l'ordre de la centaine, la tâche est plus ardue. Pour s'enquérir de la bonne santé d'une liste de machines, on peut toujours utiliser une boucle `sh` :

```
for i in `cat liste_esclaves` ; do echo ; echo "$i" ; ssh $i ps xu ; done
```

Hélas, on ne résout pas tous les problèmes de cette façon, pour plusieurs raisons :
- Le cable réseau d'un esclave particulier peut être débranché ponctuellement. Cela bloque la boucle.
- Lancer une telle boucle sur une centaine de machines prend beaucoup de temps.
- Voir si un processus tourne n'est pas tout. On peut avoir, pêle-mêle, à relancer le programme sur une machine esclave, à le stopper, à transférer des fichiers (résultats de cribles) depuis l'esclave vers une autre machine, enfin à vérifier quels sont les paquets confiés par le maître à l'esclave, et la cohérence de cette liste de paquets avec les listes gérées par le maître.

Le produit le plus précieux du calcul est le résultat des opérations de crible, prenant la forme de nombreux fichiers (un par paquet), produits par les esclaves. Ces fichiers ne sont pas gérés de façon automatique, pour éviter la « perte en ligne ». Ils sont simplement stockés par les esclaves sur l'espace disque local. Pour alléger la charge sur le réseau, on a choisi de placer ces fichiers dans l'arborescence locale `/tmp` de chaque esclave. L'inconvénient de cette approche est que le répertoire `/tmp`, sur certaines machines, est périodiquement « nettoyé », ce qui nous a parfois obligé à avancer artificiellement la date de modification des fichiers concernés. La situation la plus désagréable que l'on a rencontrée à cet égard est celle de quelques stations de travail qui s'obstinent à effacer l'arborescence de `/tmp` à chaque réinitialisation.

Notre objectif a été, depuis l'origine, d'utiliser le temps machine *disponible*, en faisant en sorte que nos esclaves laissent la priorité à tout autre calcul pouvant avoir lieu sur la machine concernée. Nos programmes ont ainsi toujours été lancés avec le niveau de priorité Unix `19`, signifiant la priorité la plus basse. Dans certains cas, cette approche a très bien fonctionné, puisque sur certaines machines[7], nos programmes n'ont pas eu l'occasion de cribler un seul paquet entier en plusieurs semaines (à un tel point que les paquets en question avaient été jugés perdus par le serveur central).

Hélas, laisser tourner un processus en « tâche de fond » de cette façon implique une gestion sociologique parfois difficile. Les machines sur lesquelles on a voulu utiliser ce système ont été essentiellement les machines du laboratoire LIX et du *cluster* Medicis, et des machines de salles de travaux dirigés destinées aux élèves, principalement à l'École polytechnique, mais aussi (bien que très marginalement) à l'École normale supérieure de Paris et à l'*University of Illinois at Chicago*.

D'un côté, sur le *cluster* Medicis, le système a très bien fonctionné, dans un certain sens, puisque tous les calculs des autres utilisateurs « passaient devant » nos calculs de logarithmes discret. Le seul point négatif est que la raison d'être de la basse priorité que nous avons donnée à nos programmes est qu'il s'agit de calculs de longue haleine. Mais souvent, les calculs devant lesquels nos programmes se sont poliment effacés étaient aussi des calculs de longue haleine (on parle ici de milliers d'heures de calcul). Cet aspect peut paraître décourageant. Il n'a hélas

[7] Il s'agit ici des machines `leon` du *cluster* Medicis.

pas été possible, d'une manière générale, d'apprendre à tous les gros utilisateurs de temps de calcul à avoir la même politesse que nous.

Mais de très loin, la plus grosse difficulté dans la gestion d'un calcul distribué est la gestion des utilisateurs « naïfs » (pour ne pas dire plus). Plus exactement, le programme `xload` est un frein majeur au déploiement d'un calcul distribué. Car utiliser un ordinateur, aujourd'hui, ne consiste pas majoritairement à exécuter des programmes sur cet ordinateur. Et lorsque le système de fenêtrage standard sur un site donné informe l'utilisateur en temps réel de l'évolution de la charge du microprocesseur de la machine, le passage du blanc au noir de l'icône associé est une source d'affolement. C'est ainsi que l'on a pu subir les foudres d'utilisateurs persuadés que la présence d'un processus parasite sur « leur » machine était la cause de leurs difficultés à charger une page *web*.

C'est aussi d'une façon similaire, dans un registre plus grave, que l'on a pu constater un comportement intéressant de la part de certains utilisateurs qui, pour « nettoyer » l'ordinateur sur lequel ils souhaitent travailler, commencent par en arracher la prise de courant. Ils semblent ignorer hélas deux points. Ce n'est d'abord pas le mode d'emploi normal d'un ordinateur (mais plutôt d'un aspirateur, qui fonctionne différemment), et surtout ce genre de pratique se repère assez facilement à partir des traces générées par le système.

Devant ce genre de comportement hostile, il est clair que la notion de priorité n'est pas suffisante. Pour utiliser une machine sans craindre le débranchement sauvage, une approche plus raisonnable aurait pu consister à stopper agressivement les calculs dès l'arrivée du moindre utilisateur. Hélas une telle approche implique une chute considérable du rendement des esclaves, car la présence d'un utilisateur et l'« utilisation », en terme de calcul, de l'ordinateur en question, ne sont pas des événements corrélés.

4.8.3 Rassemblement des données

L'ensemble des fichiers de données rassemblés à partir du travail des esclaves se compte par centaines de milliers, pour un encombrement total d'environ 10Go (gigaoctets). Au moment où ce calcul a été mené, aucun des disques durs auxquels nous avions accès ne permettait de stocker une telle quantité de données. Il a donc fallu les répartir sur plusieurs disques, et faire un travail de vases communicants entre ces disques. Ce point aurait pu être évité avec l'achat de quelques disques durs de 40Go.

Deuxième partie

Résolution de systèmes linéaires creux

Chapitre 5

Présentation du problème

5.1 Algèbre linéaire rapide et algèbre linéaire creuse

Nous avons achevé dans la partie I de ce mémoire la présentation de l'algorithme de Coppersmith ainsi que de plusieurs algorithmes de calcul d'index. Tous passent par une phase d'algèbre linéaire. Le problème, plus explicitement, est le suivant : on dispose d'une matrice singulière à N colonnes, et possiblement un plus grand nombre de lignes. Appelons cette matrice B. On recherche un élément du noyau de B, c'est-à-dire un vecteur w qui soit solution de l'équation

$$Bw = 0.$$

Notre problème est donc la résolution de systèmes linéaires homogènes. Il s'agit bien sûr d'un sujet sur lequel la littérature est très vaste. Il est donc nécessaire de cerner un peu plus précisément le problème pour déterminer le champ des algorithmes applicables. La caractéristique principale des systèmes linéaires que l'on doit résoudre est qu'ils sont *creux*. On entend par là que ces systèmes ont peu de coefficients non nuls par ligne (de l'ordre de grandeur de $\log N$).

5.1.1 Nécessité d'employer l'algorithmique « creuse »

On peut se demander si cette caractéristique impose ou non l'emploi d'une certaine catégorie d'algorithme. En effet, des algorithmes efficaces permettent de résoudre des systèmes linéaires sans tenir compte de ce caractère creux, notamment en utilisant la méthode de factorisation récursive et l'algorithme de multiplication rapide de Strassen [Str69], de complexité $O(N^{2.81})$. La complexité théorique que l'on peut atteindre pour la résolution de systèmes linéaires par ce type de procédé est reliée à la complexité de la multiplication de matrices. Si cette dernière est $O(N^w)$, on peut résoudre des systèmes linéaires par factorisation récursive en temps $O(N^w \log N)$, la meilleure valeur de w connue à ce jour étant 2.376 pour l'algorithme de Coppersmith-Winograd.

On s'interroge généralement sur la pertinence *pratique* des algorithmes asymptotiquement rapides de multiplication de matrices, car la constante du $O()$ est importante. Toutefois, il est faux de croire que le plus simple de ces algorithmes, l'algorithme de Strassen, est de portée exclusivement théorique [Knu98, sec. 4.6.4]. Dans le programme Magma, il se révèle meilleur que l'algorithme classique pour des matrices de taille 32×32 à coefficients entiers.

Face à ces méthodes « denses », il existe des algorithmes qui tirent parti du caractère creux des matrices traitées. En terme de complexité, les algorithmes que l'on verra dans ce chapitre sont de complexité $O(\gamma N^2)$, où γ est le nombre moyen de coefficients non nuls dans les lignes de la matrice M. Il s'agit donc d'une complexité bien meilleure que celles des algorithmes « denses » si $\gamma \sim \log N$. En outre, cette complexité n'inclut pas de constantes importantes cachées dans l'exposant. Enfin et surtout, les algorithmes « creux » *conservent* le

caractère creux de la matrice d'entrée. Bien souvent, c'est ce facteur qui est déterminant, car la taille des matrices traitées interdit tout stockage en mémoire sous forme dense. Ces trois arguments rendent l'algorithmique « creuse » absolument incontournable pour la résolution des problèmes qui nous intéressent.

5.1.2 Différents algorithmes existants

Pour commencer, une littérature très abondante et déjà relativement ancienne existe dans le cadre des systèmes linéaires « numériques », c'est-à-dire définis sur $\mathbb{R}$ ou $\mathbb{C}$, où les matrices qui interviennent sont souvent très creuses [GL81, GGL93]. Hélas, ces travaux reposent sur des propriétés structurelles des systèmes considérés, comme la concentration des coefficients non nuls autour de la diagonale. De tels algorithmes ne peuvent s'appliquer à notre cas.

Les algorithmes applicables à notre cas sont décrits dans ce chapitre. Tous tiennent compte du caractère creux. En revanche, tous ne sont pas des outils indifféremment interchangeables, ayant uniquement vocation à fournir en sortie le résultat final. Par exemple, l'élimination structurée que nous exposons en 5.2 doit être vue comme une étape *préalable* de réduction de la matrice. Après cette étape, on doit utiliser un algorithme qui effectue la résolution à proprement parler, comme par exemple les autres algorithmes exposés ici.

Pour représenter en mémoire une matrice creuse de façon économique, le choix universellement fait est celui d'une liste des coefficients non nuls. Pour chacun des algorithmes que nous allons exposer, on peut avoir recours à des présentations légèrement différentes, mais le principe est le même. Bien entendu, lorsque l'on considère des matrices sur $\mathbb{F}_2$, il suffit de stocker en mémoire les positions des coefficients non nuls, puisqu'il n'est nécessaire de stocker aucune information concernant la *valeur* de ces coefficients.

5.2 Préconditionnement : l'élimination structurée (SGE)

Nous exposons d'abord l'algorithme d'élimination gaussienne structurée, ou *structured gaussian elimination* (SGE). Cet algorithme est exposé par exemple dans [LO90, PS92], et étudié avec un peu plus de recul dans [BC99]. Son mode de fonctionnement n'est autre que celui de l'élimination gaussienne classique, mais avec le souci supplémentaire de conserver le caractère creux de la matrice. La description d'un tel algorithme est délicate, car le procédé est très sensible à des variations mineures de la présentation.

5.2.1 Utilisation des propriétés de structure

Le point de départ de la SGE est la structure particulière des matrices rencontrées dans les problèmes de factorisation et de logarithme discret. Ces matrices ont en effet leurs coefficients non nuls concentrés « à gauche » dans la matrice. La raison de ce phénomène a été donnée page 29. Un exemple de matrice produite par l'algorithme de Coppersmith est donné par la figure 5.1. L'intensité du niveau de gris correspond à la densité des coefficients non nuls, l'échantillonage étant fait sur des sous-matrices de taille $2\,000 \times 2\,000$. La présence de deux motifs distincts, mettant en évidence des lignes plus lourdes que les autres, est due à la *large prime variation*, décrite en 3.1.

L'algorithme d'élimination structurée utilise le très faible remplissage de la partie droite de la matrice, en contraste avec la forte densité de la partie gauche. Au cours du processus, on conserve en permanence une information sur le *poids* des lignes et des colonnes. Le poids

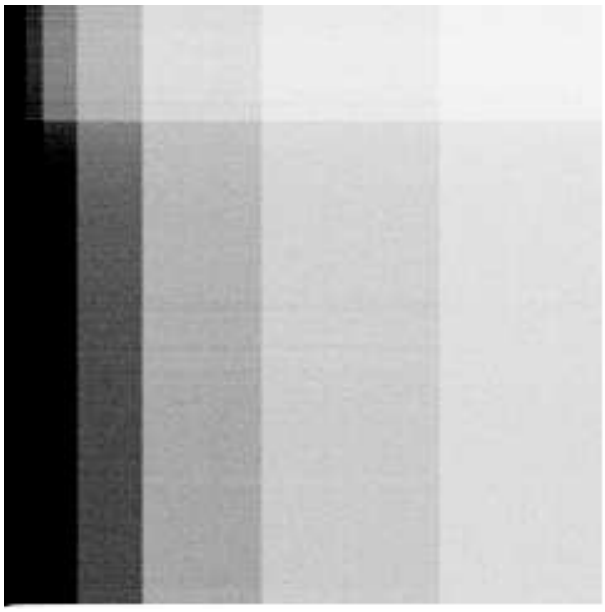


Figure 5.1 – Un exemple de matrice de logarithme discret

d'une ligne n'est toutefois jugé qu'au regard d'une partie de la matrice : on ne compte pas les coefficients qui sont dans la partie gauche de la matrice, que l'on considère comme dense de toute façon. On distingue ainsi les colonnes *actives* et les colonnes *inactives*.

Ces préliminaires passés, nous pouvons maintenant aborder la description du fonctionnement de l'algorithme d'élimination structurée.

5.2.2 Étapes de l'algorithme

Le processus d'élimination structurée passe consécutivement par les étapes énumérées ci-dessous. Il est important d'insister sur le fait que le poids d'une ligne est le poids de la partie active.

Étape 0. – Déclarer les 1% de colonnes les plus lourdes comme inactives.

Étape 1. – Ôter toutes les colonnes de poids 0 : elles n'ont pas d'intérêt pour la résolution de notre système, puisqu'elles correspondent à des vecteurs parasites du noyau de B.

Étape 2. – Retirer toutes les colonnes de poids 1, en gardant de côté les lignes correspondantes. Les coordonnées correspondantes d'un vecteur du noyau pourront être déterminées à l'aide de ces lignes. Répéter les étapes 1 et 2 tant que c'est possible.

Étape 3. – Retirer arbitrairement une partie des lignes les plus lourdes pour conserver un certain ratio entre le nombre de lignes et de colonnes. Reprendre à l'étape 1 si des colonnes de poids 0 ou 1 sont apparues.

Étape 4. – Choisir les lignes les plus avantageuses comme pivots, pour éliminer par exemple les colonnes de poids 2 qui intersectent des lignes de poids 1. Répéter les étapes 1 à 4 tant que c'est possible.

Étape 5. – Déclarer de nouvelles colonnes comme inactives, et reprendre à l'étape 1.

Dans les différentes étapes que l'on vient de mentionner, plusieurs points méritent d'être précisés. Tout d'abord, l'étape 3 souligne l'intérêt d'avoir comme point de départ une matrice avec des lignes surnuméraires. Cela permet éventuellement de se débarrasser au cours du processus des lignes qui deviennent trop encombrantes. Bien que l'on souhaite ultimement avoir à gérer une matrice carrée, il n'est pas judicieux de forcer la matrice à être carrée tout au long du processus. On se doit de garder une marge de manœuvre, quitte à faire converger lentement la forme de la matrice vers une forme carrée.

5.2.3 Comparaison de l'intérêt des opérations

Comment détermine-t-on les « bonnes » lignes à choisir pour pivoter dans l'étape 4 ? Si l'on choisit une ligne de poids r pour éliminer une colonne de poids c, on montre facilement que le nombre de coefficients que l'on rajoute ainsi dans la matrice est au plus

$$(c-1)(r-1) - (c-1) - (r-1) - 1 = (c-2)(r-2) - 2.$$

C'est donc par l'évaluation de cette quantité[1] que l'on détermine quelles sont les meilleures colonnes possibles pour l'élimination. Tant qu'elle est négative, on est assuré que l'on réduit strictement la difficulté de résolution de notre système linéaire.

Un raffinement de la règle qui précède consiste à maintenir dans une table la densité des différentes colonnes. Si la densité de la colonne j est $D(j)$, alors le choix comme pivot d'une ligne possédant un coefficient non nul dans cette colonne y entraîne l'apparition de $(1 - D(j))$ coefficients en moyenne par réplication de la ligne. Si $D(j) = 1$, la colonne est dense, et cela explique que l'on ne considère que le poids *actif* des lignes. Si $D(j)$ est proche de 0, alors l'approximation $(c-1)$ faite plus haut est fidèle. Toutefois, dans un souci d'accroître la précision, il n'est pas coûteux de conserver une évaluation de $D(j)$.

Lors de l'élimination des lignes (étape 3), il est pertinent de retirer prioritairement les lignes qui intersectent de nombreuses colonnes de poids 3, car cela rend possible des opérations d'élimination dans l'étape 4 par la suite.

5.2.4 Lien avec l'étape suivante

Lors des différentes opérations effectuées, le nombre de colonnes de la matrice décroît, tandis que le nombre total de coefficients (ou plutôt le nombre de coefficients dans les N lignes les plus légères, N étant le nombre de colonnes) est d'abord décroissant, puis croissant. Au delà de cet extremum, jusqu'à quel point poursuit-on le processus d'élimination structurée ?

Le coût de l'algorithme utilisé *après* l'élimination structurée a une expression connue en fonction de N et du nombre de coefficients. Pour conserver les notations que l'on a fixées et que l'on continuera à utiliser, il est équivalent de dire que l'on dispose d'une telle expression en fonction de N et γ (le nombre de coefficients étant γN). Si l'on prend l'exemple de la fonction γN^2, qui indique le coût des algorithmes creux en général, ce qui nous intéresse est en fait un minimum de γN^2 au cours de l'évolution du processus d'élimination structurée. Une telle approche a été employée dans [WD98], et nous l'avons mise en pratique aussi. Pour être exact, le minimum qui nous intéresse est un minimum global, tandis que ce que l'on est en mesure d'évaluer facilement est la présence d'un minimum local. Dans les expériences qui ont été menées, ce point précis ne semble pas avoir introduit de différence flagrante.

[1] Pomerance et Smith [PS92] y font référence en tant que « règle de Markowitz ».

Il est à noter que l'on peut faire le choix de traiter la matrice résultant de l'élimination structurée par un algorithme « dense » dont le temps de calcul ne dépend pas de γ. Dans ce cas, le minimum est obtenu pour N minimal, ce qui revient à dire que l'on effectue une élimination gaussienne simple. Le facteur limitant devient alors celui-là même qui rend les algorithmes denses impraticables pour notre problème : l'espace mémoire requis pour stocker la matrice. En effet, en utilisant l'élimination structurée en tant qu'étape préalable à l'utilisation d'un algorithme dense, les considérations de minimisation du coût que l'on vient d'évoquer font que l'occupation mémoire reste raisonnable. Si l'on enlève ces contraintes, elle peut exploser.

5.2.5 Nature des coefficients

Un point important à remarquer dans l'étude de l'élimination structurée est l'importance de la nature des coefficients. Que le système soit défini sur un petit corps comme $\mathbb{F}_2$ ou au contraire sur un corps de grande taille (comme les systèmes provenant d'algorithmes de logarithme discret), les coefficients de la matrice B en entrée sont en général *petits* : ils correspondent typiquement à des exposants dans une factorisation, donc de l'ordre de grandeur de $\log N$. On peut considérer qu'en entrée ces coefficients peuvent sans problème être stockés dans un mot machine. Si le corps de base est $\mathbb{F}_p$, et qu'on laisse l'élimination gaussienne se dérouler sans prêter attention à la croissance des coefficients, ceux-ci peuvent largement dépasser la taille d'un mot machine à la faveur des différentes additions et multiplications de lignes de la matrice.

Ceci ouvre deux possibilités distinctes : laisser ou pas les coefficients croître au-delà de la taille d'un mot machine. Il s'avère que le surcoût, à la fois en mémoire et en temps, d'un traitement en multiprécision est totalement prohibitif (aussi bien pour l'élimination structurée que pour l'algorithme utilisé par la suite). Pour cette raison, on a fait le choix logique de contraindre les coefficients à rester de la taille d'un mot machine. Cela signifie qu'au cours de l'algorithme, une ligne peut devenir « trop lourde » au regard de la taille de ses coefficients et devenir ainsi inutilisable en tant que pivot pour toute élimination. Ce phénomène, pour le cas qui nous concerne (l'algorithme de Coppersmith pour le calcul de logarithmes discrets), s'est montré très visible puisqu'une bonne proportion (presque la moitié) des coefficients valent $-k$, donc -4 avec les paramètres choisis. Chaque élimination implique ainsi un plus grand nombre de multiplications de lignes que si tous les coefficients valaient ± 1.

Notons en outre une seconde conséquence de cette considération : dans l'élimination structurée, aucune inversion de coefficient n'est effectuée. En effet, une inversion aurait la fâcheuse conséquence de faire exploser la taille des autres coefficients de la ligne correspondante. En résumé, on constate donc que le modèle de calcul dans lequel on se place pour l'élimination structurée n'est pas le modèle algébrique.

5.2.6 Implantation

L'algorithme d'élimination structurée peut se montrer assez coûteux en mémoire (mais pas en temps, comparé aux calculs dans lesquels cette méthode intervient). Plusieurs moyens de réagir pour réduire la consommation mémoire de l'algorithme peuvent être évoqués.

– Tout d'abord, on peut constater que l'élimination structurée éjecte agressivement des lignes et des colonnes de la matrice. Certaines d'entre elles conservant un intérêt potentiel, elles peuvent éventuellement être stockées sur le disque dur, mais en aucun cas être conservées en mémoire.

– Par ailleurs, la partie dense, ou « inactive » de la matrice contient la plupart des coefficients. Il est parfois recommandé de ne pas effectuer les opérations de ligne sur cette partie, pour ne les effectuer que plus tard. Nous n'avons pas eu besoin de recourir à ce procédé.
– Si la matrice est représentée sous forme d'une liste de coefficients non nuls pour chaque ligne, on ne peut éviter un accès « par colonne », qui permette de localiser les lignes ayant un coefficient non nul dans une colonne donnée, pour effectuer les éliminations voulues. Des listes de liens croisés entre les lignes et les colonnes doivent ainsi être maintenues. Il est évident que l'on doit éviter de stocker cette information pour les colonnes de la partie inactive de la matrice.

Le maintien des diverses données quantitatives sur la matrice (poids des lignes et des colonnes, liste des pivots potentiels, taille des coefficients d'une ligne) représente un surcoût non significatif.

Dans l'implantation que nous avons effectuée, les étapes 3 et 4 de l'algorithme ont été globalisées. En considérant l'ensemble des actions possibles au cours de ces étapes (éliminations pures et simples de lignes, ou éliminations par pivot), on constitue une liste où on associe un « gain » heuristique à chaque opération. Une des composantes de gain heuristique est l'influence de l'action programmée sur le coût de l'algorithme utilisé après la SGE. Mais par ailleurs, nous avons aussi incorporé une heuristique pour favoriser les « catastrophes » au sens de [PS92]. Ainsi, l'action consistant à retirer une ligne de la matrice a été considérée comme provoquant d'une part le retrait du poids de ses coefficients au poids total de la matrice (rien d'étonnant), et d'autre part l'abaissement du nombre de colonnes pour chacun des coefficients de la ligne intersectant une colonne de poids 3.

Cette liste d'actions et de leurs gains respectifs une fois constituée (on ne s'intéresse qu'aux actions avantageuses), on trie la liste, et on effectue les actions les meilleures, à concurrence, d'une part, d'un total de 1 000 actions distinctes, et d'autre part, du retrait d'un nombre donné de lignes. Ce nombre a été fixé à 33% de la différence entre le nombre de lignes et de colonnes lorsque le quotient de ces deux derniers dépasse 1.5, et 5% sinon. On n'a toutefois jamais contraint ainsi le nombre de retraits de lignes à être inférieur à 100, pour ne pas ralentir artificiellement la convergence de l'algorithme. On n'a bien sûr pas non plus laissé le nombre de lignes devenir inférieur à N.

5.3 Algorithmes pour terminer la résolution

Après l'étape de l'élimination structurée, on s'intéresse aux algorithmes qui permettent réellement de « terminer » le calcul. Nous nous intéressons à deux algorithmes : l'algorithme de Lanczos et l'algorithme de Wiedemann. Leurs versions « par blocs » seront aussi étudiées.

Les derniers paragraphes de ce présent chapitre sont consacrés d'abord à l'introduction de l'algèbre linéaire *black-box*, contexte général dans lequel on peut raisonnablement bien inclure les différents algorithmes étudiés, ainsi qu'à la description de ce qu'on entend lorsqu'on évoque des versions « par blocs » d'algorithmes d'algèbre linéaire. Nous introduisons aussi brièvement l'algorithme de Lanczos. L'algorithme de Wiedemann, qui occupe une place plus centrale dans ce mémoire, est étudié au chapitre 6.

5.3.1 La multiplication matrice × vecteur : algèbre linéaire *black-box*

En notant toujours γ le nombre de coefficients non nuls par ligne de la matrice B considérée, on peut tirer parti de la faible valeur de γ en n'autorisant qu'un seul emploi de la matrice B : la multiplication par un vecteur. On exclut toute manipulation à l'intérieur de la matrice B (à l'inverse de ce que l'on a pratiqué dans la SGE précédemment exposée). On dit alors que B agit comme une boîte noire, comme le représente la figure 5.2.

Figure 5.2 – Une boîte noire

La théorie de l'algèbre linéaire *black-box* consiste à reposer exclusivement sur cette opération en ignorant tout de la matrice B. Ceci n'impose pas le contexte de l'algèbre linéaire creuse : pour qu'une méthode utilisant ainsi une boîte noire de manière exclusive soit efficace, il suffit que le procédé d'évaluation soit efficace, sans considération du caractère creux de sa représentation matricielle dont on ignore tout si l'on refuse de voir la matrice autrement que comme une boîte noire.

Dans une certaine mesure, les travaux que nous exposons dans cette partie de ce mémoire peuvent être vus comme des méthodes *black-box*. Toutefois, nous tenons à remarquer que notre intérêt n'est pas là (nous nous contentons de mentionner la terminologie). Tout en n'utilisant effectivement la matrice B que pour effectuer des produits matrice × vecteur, on se permet des considérations sur sa représentation mémoire, et sur le fait qu'il s'agit d'une matrice creuse. Ces considérations, si elles ne rendent pas notre approche nécessairement divergente d'une approche *black-box*, sont guidées par les réalités d'un modèle qui n'est pas un modèle tout à fait générique. Notre souci d'efficacité calculatoire est ainsi complètement ancré dans la réalité des matrices qui nous concernent.

L'exemple parfait d'algorithme s'inscrivant dans le modèle de boîte noire est l'algorithme de Wiedemann, que nous décrirons au chapitre 6. L'algorithme de Lanczos, par lequel nous allons commencer, nécessite l'extension suivante du modèle : autoriser le calcul du produit $v \to B^T v$. Aussi bien dans la théorie que dans la pratique, cette opération est effectivement intimement liée au produit $v \to Bv$: si l'on voit la boîte noire comme un circuit, la multiplication par la transposée s'obtient en « renversant » le circuit. Dans le cas qui nous concerne, celui des matrices creuses, ce renversement est réellement aisé, et la même représentation en mémoire de la matrice B suffit aux deux opérations.

5.3.2 Introduction de blocs

Les algorithmes sur les matrices creuses qui s'inscrivent dans le modèle de boîte noire peuvent bénéficier d'une façon générique de l'introduction de blocs. Par extension d'un vecteur, que l'on voit comme une matrice à N lignes et une colonne, on appelle *bloc de vecteurs* une matrice de taille $N \times n$, où n est un petit entier. Une généralisation par blocs d'un algorithme d'algèbre linéaire consiste, lorsqu'il s'agit d'algorithmes reposant uniquement sur le produit matrice × vecteur, à considérer maintenant comme primitive la multiplication par un bloc de vecteurs.

Pour être rentable, une telle généralisation doit permettre une réduction du nombre d'utilisations de la boîte noire au cours de l'algorithme, et le calcul du produit matrice × bloc de vecteurs doit s'effectuer de manière efficace. Idéalement, on voudrait que ce produit s'effectue en le même temps qu'un simple produit matrice × vecteur. C'est le cas dans les deux configurations suivantes.

- Si le corps de base est $\mathbb{F}_2$, pour effectuer le produit w d'une matrice creuse B par un vecteur v, il est difficile de s'écarter du schéma suivant :
  - $w \leftarrow 0$,
  - Pour chaque coefficient non nul B_{ij} de B, $w_i \leftarrow w_i \oplus v_j$,

  où $\oplus$ désigne l'addition modulo 2 (le « ou exclusif »). Le caractère inévitable de ce schéma tient au fait que la matrice est creuse. Sur une machine ayant des mots machines de 32 bits, on peut en le même temps effectuer le produit de B par un *bloc* v de 32 vecteurs : Si chaque v_j est un entier de 32 bits, et qu'il en est de même pour chaque w_i, la même écriture de la méthode produit le résultat voulu, en interprétant $\oplus$ comme l'opération logique « ou exclusif » de deux mots machines.
- Si le corps de base est $\mathbb{F}_q$, un procédé identique à ce que l'on vient de décrire peut difficilement être mis en œuvre, à moins de disposer d'une architecture spécifique. En revanche, le produit matrice × bloc de vecteurs peut être facilement réparti sur plusieurs processeurs ou plusieurs machines. Si le vecteur v est constitué des *colonnes* $v_1, \ldots, v_n$ (on rompt avec les notations prises précédemment), alors les colonnes $w_1, \ldots, w_n$ peuvent être calculées indépendamment puisque $w_k = Bv_k$. Ainsi, n machines distinctes peuvent se partager la tâche, chacune d'entre elles prenant en charge l'une des colonnes. Un tel schéma est particulièrement rentable si les produits de blocs de vecteurs peuvent être enchaînés sans globalisation intermédiaire des données : ainsi, on n'a pas de surcoût de communication trop important. Nous verrons que ce résultat est atteint par l'algorithme de Wiedemann par blocs, mais pas par l'algorithme de Lanczos par blocs.

5.3.3 L'algorithme de Lanczos

Nous entamons maintenant la description de quelques algorithmes qui permettent de produire un vecteur du noyau de la matrice B. Le premier est l'algorithme de Lanczos, qui est en fait l'adaptation du procédé d'orthogonalisation de Gram-Schmidt. Cet algorithme provient des méthodes numériques.

L'orthogonalisation de Gram-Schmidt

Le procédé d'orthogonalisation de Gram-Schmidt est une méthode constructive pour obtenir une base de vecteurs orthogonaux relativement à une forme bilinéaire symétrique non dégénérée donnée. Si la forme bilinéaire considérée *est* dégénérée, alors des vecteurs auto-orthogonaux sont produits. Le procédé est adapté au cas où le corps de base K est $\mathbb{R}$ ou $\mathbb{C}$. Nous allons l'appliquer au cadre des corps finis, en soulignant à quel moment des problèmes apparaitront. Nous supposons dans un premier temps que K est $\mathbb{R}$ ou $\mathbb{C}$.

Soit A une matrice symétrique de $K^{N\times N}$. On considère la forme bilinéaire symétrique associée à A sur K^N, définie par $(x, y) \to (x|Ay) = x^T Ay$. Soit $(e_0, \ldots, e_{N-1})$ une base quelconque de K^N. Soient $w_0, \ldots, w_{N-1}$ les vecteurs définis par :

$$w_0 = e_0,$$

$$w_1 = e_1 - \frac{(Ae_1|Aw_0)}{(w_0|Aw_0)}w_0,$$
$$\dots$$
$$w_k = e_k - \sum_{i=0}^{k-1} \frac{(e_k|Aw_i)}{(w_i|Aw_i)}w_i, \ \dots$$

Il est alors aisé de montrer que les vecteurs ainsi construits sont orthogonaux, et que $w_0, \dots, w_k$ engendrent le même sous-espace vectoriel que $e_0, \dots, e_k$. La seule obstruction possible au procédé est la nullité éventuelle d'une des formes quadratiques $(w_i|Aw_i)$, qui correspond à l'existence d'un vecteur auto-orthogonal pour la forme bilinéaire représentée par A.

Introduction d'espaces de Krylov

Cherchons maintenant à obtenir un algorithme à partir de la méthode précédemment exposée. Pour cela, la matrice B définissant notre système linéaire n'étant pas symétrique, on prend $A = B^T B$. On se place, non pas dans K^N, mais dans le sous-espace de Krylov [Kry31] associé à un vecteur de départ arbitraire v. Il s'agit du sous-espace vectoriel engendré par les itérés de v par la matrice A. La base choisie pour le sous-espace est donc naturellement $(v, Av, \dots, A^k v, \dots)$. La suite des vecteurs w_k est alors :

$$w_0 = v,$$
$$w_1 = Aw_0 - \frac{(Aw_0|Aw_0)}{(w_0|Aw_0)}w_0,$$
$$\dots$$
$$w_k = Aw_{k-1} - \sum_{i=0}^{k-1} \frac{(Aw_i|Aw_{k-1})}{(w_i|Aw_i)}w_i.$$

Cette dernière expression peut se simplifier :

$$\begin{aligned} w_k &= Aw_{k-1} - \sum_{i=0}^{k-1} \frac{(Aw_i|Aw_{k-1})}{(w_i|Aw_i)}w_i, \\ &= Aw_{k-1} - \frac{(Aw_{k-1}|Aw_{k-1})}{(w_{k-1}|Aw_{k-1})}w_{k-1} - \frac{(Aw_{k-1}|Aw_{k-2})}{(w_{k-2}|Aw_{k-2})}w_{k-2}, \\ &\dots \end{aligned}$$

On a pu simplifier l'expression récurrente de w_k grâce aux relations d'orthogonalité qu'ils satisfont, ainsi que l'égalité des sous-espaces vectoriels engendrés :

$$\langle w_0, \dots, w_k \rangle = \langle v, Av, \dots, A^k v \rangle.$$

On peut bien sûr normaliser les vecteurs w_k à chaque étape, pour ne pas avoir à traiter des produits scalaires inutiles.

Justification

Quels peuvent être les problèmes rencontrés par la méthode ? On a vu que le procédé échouait lorsqu'il trouvait un vecteur auto-orthogonal. Si x est un tel vecteur, cela signifie $(x|Ax) = (Bx)^T(Bx) = 0$. Comme le corps de base sur lequel nous travaillons est un corps fini et non pas $\mathbb{R}$ ou $\mathbb{C}$, cela n'implique pas que le vecteur Bx est nul. On peut néanmoins considérer que c'est le cas si K est un corps de grande caractéristique. On a alors un vecteur x du noyau de B.

Il se peut maintenant que le vecteur x obtenu soit nul. C'est impossible tant que i n'excède pas la dimension du sous-espace de Krylov dans lequel on travaille, mais par la suite, si A n'est pas singulière sur ce sous-espace, le seul vecteur du noyau qui peut être obtenu de cette façon est le vecteur nul.

Ces deux possibilités d'échec rendent l'algorithme tel qu'il est décrit ici impraticable pour les petits corps finis, en particulier $\mathbb{F}_2$. Deux approches peuvent être employées pour contrer cette difficulté. L'approche proposée par [LO90] consiste à travailler sur une extension $\mathbb{F}_{2^k}$, en prenant pour la matrice A non pas la matrice B^TB, mais plutôt B^TD^2B, où D est une matrice diagonale à coefficients aléatoires dans $\mathbb{F}_{2^k}$. La version par blocs présentée en 5.3.4 en est une autre, plus efficace. Par ailleurs, comme mentionné dans le paragraphe précédent, il faut s'assurer que l'espace de Krylov choisi est tel que B reste singulière dans ce sous-espace. Les deux approches que l'on vient de citer résolvent aussi cette difficulté.

Complexité

Le calcul de la complexité de l'algorithme de Lanczos est plutôt aisé. On peut s'attendre à ce que la dimension de l'espace de Krylov considéré s'approche de N. On compte donc N itérations de l'algorithme. Le coût de chacune des ces opérations est de deux multiplications de la matrice B par un vecteur. Une telle multiplication nécessite γN multiplications scalaires. À cela doivent s'ajouter les coûts des produits scalaires effectués, les coûts éventuels de normalisation, et éventuellement les surcoûts impliqués par les précautions indiquées au paragraphe précédent. Si la valeur de γ est de l'ordre de quelques dizaines, alors les multiplications matrice-vecteur représentent la composante dominante de la complexité. On arrive au nombre de multiplications scalaires suivant :

$$(2\gamma + \epsilon)N^2,$$

où ϵ est introduit pour tenir compte des différentes opérations *a priori* peu coûteuses mentionnées ci-dessus.

5.3.4 L'algorithme de Lanczos par blocs

La mise en place d'une version par blocs de l'algorithme de Lanczos a été effectuée simultanément par Coppersmith [Cop93] et Montgomery [Mon95, EH96]. Il mêle à la fois l'introduction de blocs, et une généralisation de la méthode de *lookahead* présentée dans [PTL85, Lam96], combinant donc des bénéfices théoriques et pratiques. Nous décrivons ici la méthode proposée par Montgomery. L'algorithme de Lanczos par blocs est hélas intrinsèquement complexe, et présenté de manière généralement peu éclairante. La présentation qui suit n'échappe pas à cet aspect « technique ».

Le principe de l'algorithme s'énonce relativement aisément : plutôt que de tenter d'orthogonaliser une suite de vecteurs, on fabrique une suite de sous-espaces vectoriels. Dans

la mesure du possible nous conservons dans notre présentation les notations utilisées dans [Mon95, EH96], sans modifier toutefois les notations en vigueur dans ce mémoire. On se donne une matrice symétrique A, de taille $N \times N$, définie sur un corps K. Un entier n est introduit, conditionnant la taille des blocs. L'étude que l'on mène est toujours centrée sur la forme bilinéaire symétrique définie par A. Toutefois, plutôt qu'une base orthogonale pour cette forme, on construit maintenant une suite de sous-espaces vectoriels $\mathcal{W}_i$, de dimensions respectives n_i, avec $n_i \leqslant n$. Chacun de ces sous-espaces est représenté par une base, constituée des colonnes d'une matrice $\mathbf{W}_i$ de taille $N \times n_i$. Pour rendre la présentation aussi compréhensible que possible, on commence par exposer quelques principes généraux de fonctionnement de l'algorithme, avant de décrire exactement les équations qui le régissent.

Les matrices $\mathbf{W}_i$ sont construites par *extraction* à partir de matrices $\mathbf{V}_i$. Les matrices $\mathbf{V}_i$ jouent le rôle des vecteurs w_i dans l'algorithme de Lanczos simple. On construit les matrices $\mathbf{W}_i$ et donc les sous-espaces $\mathcal{W}_i$ de telle sorte que les conditions suivantes soient vérifiées :

i) La forme bilinéaire représentée par A est non dégénérée sur $\mathcal{W}_i$, c'est-à-dire :

$$\mathbf{W}_i^T A \mathbf{W}_i \in \mathrm{GL}_{n_i}(K).$$

ii) Les $\mathcal{W}_j$ sont mutuellement A-orthogonaux :

$$\forall i, j \quad i \neq j \Rightarrow \mathbf{W}_j^T A \mathbf{W}_i = 0.$$

iii) Les $\mathcal{W}_j$ « ressemblent » à une décomposition d'un sous-espace de Krylov, puisque l'on requiert la propriété :

$$A\mathcal{W}_i \subset \bigoplus_{j \leqslant i+2} \mathcal{W}_j.$$

Nous exprimons le fait que la matrice $\mathbf{W}_i$ est extraite de la matrice $\mathbf{V}_i$ par l'existence d'une matrice $\mathbf{S}_i \in K^{n \times n_i}$ telle que $\mathbf{W}_i = \mathbf{V}_i \mathbf{S}_i$. La matrice $\mathbf{S}_i$ est une simple matrice d'extraction de colonnes : elle a exactement une entrée non nulle, égale à 1, dans chaque colonne, et pas plus d'une entrée non nulle par ligne. Ainsi, la ligne k est non nulle si et seulement si la k-ème colonne de $\mathbf{V}_i$ est « choisie » dans $\mathbf{W}_i$. Ce choix de la matrice $\mathbf{S}_i$ garantit l'identité $\mathbf{S}_i^T \mathbf{S}_i = I_{n_i}$. La matrice $\mathbf{S}_i \mathbf{S}_i^T$, par ailleurs, est une matrice diagonale avec exactement n_i entrées non nulles égales à 1.

Le point de départ de l'algorithme est une matrice arbitraire $\mathbf{V}_0$. L'entier n_0 est le rang de la matrice $\mathbf{V}_0^T A \mathbf{V}_0$ (voir lemme ci-dessous), et la matrice $\mathbf{S}_0$ est choisie de façon à sélectionner des indices correspondant à des colonnes linéairement indépendantes de $\mathbf{V}_0$. Ce procédé, plus généralement, sera employé à chaque étape de l'algorithme pour choisir $\mathbf{W}_i$ en fonction de $\mathbf{V}_i$. Il est justifié par le lemme suivant :

Lemme 5.1. *Soit Q une matrice symétrique $n \times n$ de rang r. Soient $k_1, \ldots, k_r$ les indices de r colonnes linéairement indépendantes de la matrice Q. Alors le mineur correspondant à ces indices est non nul.*

Démonstration. Dans cet énoncé, le mineur que l'on considère est une des entrées de la *diagonale* de la puissance extérieure r-ème $\Lambda^{(r)}Q$ de Q (la matrice $\Lambda^{(r)}Q$ est carrée de taille $\binom{n}{r}^2$). C'est le déterminant d'une matrice de la forme $\mathbf{S}^T Q \mathbf{S}$, où $\mathbf{S}$ est une matrice de taille $n \times n_i$ semblables aux matrices $\mathbf{S}_i$ que l'on a introduites (d'où le lien de ce lemme avec notre problème), les lignes non nulles de $\mathbf{S}$ étant les lignes d'indice $k_1, \ldots, k_r$.

On peut supposer sans perte de généralité que les r premières colonnes de la matrice Q sont linéairement indépendantes : en effet, une permutation des indices mettra « en tête » le mineur qui nous intéresse (devenant ainsi le mineur principal d'ordre r). On peut alors écrire la matrice Q sous la forme $\left(\begin{array}{c|c} Q_{11} & Q_{12} \\ \hline Q_{21} & Q_{22} \end{array}\right)$. On a $Q_{21} = Q_{12}^T$. Comme la matrice est de rang r, il existe una matrice $\Sigma \in K^{r\times(n-r)}$ telle que $Q_{12} = Q_{11}\Sigma$ et $Q_{22} = Q_{21}\Sigma$. Cela implique $Q_{21} = Q_{12}^T = (Q_{11}\Sigma)^T = \Sigma^T Q_{11}$. On peut donc écrire :

$$
\begin{aligned}
Q &= \left(\begin{array}{c|c} Q_{11} & Q_{12} \\ \hline Q_{21} & Q_{22} \end{array}\right), \\
Q &= \left(\begin{array}{c|c} Q_{11} & 0 \\ \hline Q_{21} & 0 \end{array}\right)\left(\begin{array}{c|c} I_r & \Sigma \\ \hline 0 & I_{n-r} \end{array}\right), \\
Q &= \left(\begin{array}{c|c} I_r & 0 \\ \hline \Sigma^T & I_{n-r} \end{array}\right)\left(\begin{array}{c|c} Q_{11} & 0 \\ \hline 0 & 0 \end{array}\right)\left(\begin{array}{c|c} I_r & \Sigma \\ \hline 0 & I_{n-r} \end{array}\right).
\end{aligned}
$$

La matrice Q_{11} est donc inversible, comme on l'a requis. ■

Nous appliquons ce lemme pour construire $\mathbf{W}_i$ à partir de $\mathbf{V}_i$. La structure itérative de l'algorithme de Lanczos par blocs apparaît par l'usage d'une récurrence. Pour construire la matrice $\mathbf{V}_{i+1}$, on a deux points de départ : d'une part la matrice $A\mathbf{W}_i$, et d'autre part la matrice $\mathbf{V}_i$ elle-même. Ainsi, les vecteurs de $\mathbf{V}_i$ qui n'ont pas été sélectionnés dans $\mathbf{W}_i$ sont pris en compte à l'itération suivante. On a donc une formule du type :

$$\mathbf{V}_{i+1} = A\mathbf{W}_i + \mathbf{V}_i - \text{termes correctifs.}$$

Ajouter ainsi la contribution de $\mathbf{V}_i$ permet de conserver le rang de la suite des matrices $\mathbf{V}_i$, qui descendrait à zéro sinon. Il n'est pas nécessaire de procéder ainsi dans le Lanczos « standard » que nous avons décrit plus haut, mais c'est en revanche très important ici. Les termes correctifs qui interviennent permettent d'assurer la condition d'orthogonalité suivante, qui précise la condition *ii* plus haut :

iv) $\forall j < i+1,\ \mathbf{W}_j^T A\mathbf{V}_{i+1} = 0.$

Les termes correctifs sont choisis sous la forme d'une combinaison linéaire des matrices $\mathbf{W}_j$. L'expression de $\mathbf{V}_{i+1}$ se précise alors sous la forme suivante :

$$\mathbf{V}_{i+1} = A\mathbf{W}_i + \mathbf{V}_i - \sum_{j=0}^{i} \mathbf{W}_j \mathbf{C}_{i+1,j}.$$

En raisonnant par conditions nécessaires, on peut dériver à partir de la condition *iv* et de la condition *ii* l'expression de $\mathbf{C}_{i+1,j}$:

$$
\begin{aligned}
\mathbf{W}_j^T A\mathbf{V}_{i+1} &= 0, \\
\mathbf{W}_j^T A^2\mathbf{W}_i + \mathbf{W}_j^T A\mathbf{V}_i &= \sum_{k=0}^{i} \mathbf{W}_j^T A\mathbf{W}_k \mathbf{C}_{i+1,k}, \\
\mathbf{W}_j^T A^2\mathbf{W}_i + \mathbf{W}_j^T A\mathbf{V}_i &= \left(\mathbf{W}_j^T A\mathbf{W}_j\right)\mathbf{C}_{i+1,j}, \qquad \text{(condition } ii\text{)}
\end{aligned}
$$

$$\mathbf{C}_{i+1,j} = \left(\mathbf{W}_j^T A \mathbf{W}_j\right)^{-1} \mathbf{W}_j^T A \left(A\mathbf{W}_i + \mathbf{V}_i\right). \qquad \text{(condition } i\text{)}$$

Essayons de simplifier cette expression de $\mathbf{C}_{i+1,j}$. Pour les indices j tels que $j < i$, on a $\mathbf{W}_j^T A \mathbf{V}_i = 0$, par la condition *iv*. On souhaite annuler le terme $\mathbf{W}_j^T A^2 \mathbf{W}_i$, ce qui requiert un peu de travail. Ce terme peut s'écrire alternativement de la façon suivante, en utilisant l'écriture $\mathbf{W}_j = \mathbf{V}_j \mathbf{S}_j$. On se place dans le cas $j < i$.

$$\begin{aligned}
\mathbf{W}_j^T A^2 \mathbf{W}_i &= \mathbf{S}_j^T \mathbf{W}_j^T A^2 \mathbf{W}_i, \\
&= \mathbf{S}_j^T (A\mathbf{W}_j)^T A \mathbf{W}_i, \\
&= \mathbf{S}_j^T \left(\mathbf{V}_{j+1} - \mathbf{V}_j + \sum_{k=0}^{j} \mathbf{W}_k \mathbf{C}_{j+1,k} \right)^T A\mathbf{W}_i \qquad (j < i), \\
&= \mathbf{S}_j^T \left(\mathbf{V}_{j+1} - \mathbf{V}_j\right)^T A \mathbf{W}_i, \\
&= \mathbf{S}_j^T \mathbf{V}_{j+1}^T A \mathbf{W}_i,
\end{aligned}$$

Il est hélas impossible de simplifier encore ce terme en toute généralité. Si la matrice $\mathbf{S}_{j+1}$ est égale à I_n, alors $\mathbf{V}_{j+1} = \mathbf{W}_{j+1}$, et le terme ci-dessus est nul pour $j < i - 1$. C'est ce qui se passe dans l'algorithme de Lanczos simple, où l'extraction $\mathbf{S}_{j+1}$ est toujours complète.

Ici, pour obtenir l'annulation du terme $\mathbf{V}_{j+1}^T A \mathbf{W}_i$, on requiert une propriété plus fine de l'extraction : les colonnes de $\mathbf{V}_{j+1}$ n'apparaissant pas dans $\mathbf{W}_{j+1} = \mathbf{V}_{j+1}\mathbf{S}_{j+1}$ doivent « apparaître » dans $\mathbf{W}_{j+2}$. Ceci s'exprime par la condition suivante :

v) $\forall i,\ \operatorname{Im} \mathbf{V}_i \subset \operatorname{Im} \mathbf{W}_i + \operatorname{Im} \mathbf{W}_{i+1}$.

Grâce à cette condition, on peut garantir que $\mathbf{V}_{j+1}^T A \mathbf{W}_i = 0$ dès que $j < i - 2$ (puisque l'on peut écrire par exemple $\mathbf{V}_{j+1} = \mathbf{W}_{j+1} T + \mathbf{W}_{j+2} Z$). L'expression de récurrence peut alors se simplifier sous la forme :

$$\mathbf{V}_{i+1} = A\mathbf{W}_i + \mathbf{V}_i - \mathbf{W}_i \mathbf{C}_{i+1,i} - \mathbf{W}_{i-1} \mathbf{C}_{i+1,i-1} - \mathbf{W}_{i-2} \mathbf{C}_{i+1,i-2}.$$

La profondeur de la récurrence est donc bornée. Nous pouvons vérifier, après notre raisonnement par conditions nécessaires que les équations que nous donnons permettent de garantir les différentes conditions qui ont été énoncées, en précisant toutefois deux points.

- Le processus se poursuit tant que $\mathbf{V}_i^T A \mathbf{V}_i$ ne s'annule pas. Comme dans l'algorithme de Lanczos simple, cette annulation intervient seulement à la fin du calcul, et nous permet d'obtenir un vecteur du noyau de B, cette fois avec une certitude plus grande, puisqu'à partir de l'identité $(B\mathbf{V}_i)^T (B\mathbf{V}_i) = 0$, on a de bonnes chances de pouvoir déduire des vecteurs de $\operatorname{Ker} B$.
- La condition v ne peut pas être garantie automatiquement. À chaque étape, il faut s'assurer qu'il est possible d'inclure dans $\mathbf{W}_{i+1}$ une base du sous-espace engendré par les colonnes de $\mathbf{V}_i$ omises dans $\mathbf{W}_i$. Ce n'est pas nécessairement possible : si les colonnes correspondantes de $\mathbf{V}_i^T A \mathbf{V}_i$ sont dépendantes, l'algorithme échoue. Dans la pratique, le nombre de ces colonnes est très petit devant leur nombre de coordonnées, et dans la pratique elles sont toujours linéairement indépendantes.

Dans le processus que nous venons de décrire, l'opération la plus coûteuse qui est effectuée à chaque étape est le calcul du produit $A\mathbf{W}_i$. Les calculs des matrices $\mathbf{C}_{i+1,j}$ requièrent quelques produits scalaires, et des opérations sur des matrices $n \times n$. Il en est de même pour

le choix des matrices d'extraction $\mathbf{S}_i$. La composante majeure du coût de l'algorithme est donc l'application répétée de la matrice A à un bloc de vecteurs. Le nombre d'étapes nécessaires est $\frac{N}{n-\epsilon}$, où $n-\epsilon$ est l'espérance de n_i. Ici, ϵ est une constante. Montgomery a déterminé la valeur $\epsilon = 0.76$.

L'introduction de blocs dans l'algorithme de Lanczos est avantageuse sur $\mathbb{F}_2$. En effet, comme on l'a dit plus haut, cela nous permet d'effectuer 32 opérations en une. En outre, si l'on souhaite utiliser l'algorithme sur $\mathbb{F}_p$, dans la perspective d'une exécution parallèle ou distribuée par exemple, il n'est *pas* possible ici d'« enchaîner » les récurrences sans communication entre les nœuds prenant part au calcul. Nous allons voir que cela rend l'algorithme de Lanczos par blocs peu attractif pour une utilisation distribuée, par comparaison avec l'algorithme de Wiedemann.

5.3.5 Unification des approches « Lanczos » et « Wiedemann »

Nous anticipons ici sur la description que nous allons faire de l'algorithme de Wiedemann et de sa version par blocs, qui occuperont la partie restante de l'exposé. Il est possible de mettre en évidence une analogie profonde entre les algorithmes de Lanczos et de Wiedemann. Cette analogie a été explicitée par Lambert [Lam96] en prenant comme point de départ l'algorithme de Lanczos, et en démontrant que celui-ci pouvait produire comme sous-produit le même diviseur du polynôme minimal que celui calculé dans l'algorithme de Wiedemann.

Dans cette analogie, l'apparition de vecteurs auto-orthogonaux dans l'algorithme de Lanczos est mise en relation directe avec une chute locale de degré dans la suite des restes partiels de l'algorithme d'Euclide, utilisé dans l'algorithme de Wiedemann. Pour qui utilise dans l'algorithme de Berlekamp-Massey, cette « chute de degré » correspond à un écartement ponctuel des degrés des candidats générateurs au cours de l'algorithme. Ainsi, rendre cette analogie complètement explicite permet d'obtenir une description de l'algorithme de Lanczos avec *lookahead*, en transposant aussi l'effet de ces chutes de degré.

Nous remarquons, de notre point de vue, que cette analogie est « évidente » dans l'*autre sens* : l'algorithme de Wiedemann n'est rien d'autre qu'un algorithme de Lanczos « déplié ». En effet, l'algorithme de Wiedemann se découpe en trois phases, comme nous allons le voir extensivement dans le chapitre suivant :

- Calcul de $A(X) = \sum_{i=0}^{2N} a_i X^i$, où $a_i = x^T B^i y = x^T B^{i+1} z$ (x et z sont deux vecteurs, et $y = Bz$).
- Calcul de $F(X)$ tel que $A(X)F(X) = G(X) + O(X^{2N})$, avec F et G de degré $\leqslant N$.
- Calcul de $w = \widehat{F}(B)z$.

La première et la troisième de ces phases sont traitées de manière itérative, en un temps identique pour chaque itération (qui se résume à une multiplication d'un certain vecteur par la matrice B). La phase intermédiaire est résolue de manière itérative aussi si on utilise par exemple l'algorithme de Berlekamp-Massey, mais sa complexité globale est quadratique, car les différentes itérations prennent un temps croissant.

Que se passe-t-il si on souhaite « enchaîner » ces différentes phases ? La connaissance du résultat des premières itérations de la première phase est suffisante pour entamer la seconde, et par la suite entamer aussi la troisième phase. On obtient ainsi une sorte de « *pipeline* » pour l'algorithme de Wiedemann. Il est particulièrement intéressant de remarquer, alors, que dans l'algorithme de Berlekamp-Massey, le coût croissant des itérations disparaît, car on a à disposition l'ensemble des *vecteurs* requis (qui contiennent une plus grande information que les scalaires a_i).

Il est donc possible de fusionner les trois étapes de l'algorithme de Wiedemann en un procédé itératif. Expliciter la bijection serait relativement complexe, et nous ne le faisons pas ici. En particulier, on retrouverait la propriété de « *lookahead* », obligeant à conserver la mémoire d'un nombre accru de vecteurs pour pallier aux éventuelles annulations (prenant la forme d'écart des degrés des candidats générateurs dans l'algorithme de Berlekamp-Massey).

Une telle bijection peut aussi être construite pour les versions par blocs des algorithmes de Lanczos et Wiedemann, mais là encore, expliciter une telle correspondance est un travail d'écriture peu aisé.

Chapitre 6

Méthodes utilisant des générateurs linéaires

6.1 Générateurs linéaires

Les algorithmes décrits dans ce chapitre font appel à diverses notions de suites linéairement engendrées. Nous énonçons ici une définition très générale de ce que l'on entendra par suite linéairement engendrée, et par générateur linéaire. Cette définition se spécialise dans la plupart des cas en la définition « intuitive » de ces concepts, à un renversement près des coefficients du générateur. Plus exactement, nous proposons ici une distinction entre générateur linéaire et polynôme minimal, lorsque ce dernier est défini.

6.1.1 Formalisme

Soit K un corps et R un K-espace vectoriel. On va considérer le K-espace vectoriel des suites à coefficients dans R, noté $R^{\mathbb{N}}$. Une suite dans $R^{\mathbb{N}}$ est notée $u = (u_n)_{n\in\mathbb{N}}$. On associe à la suite u une série formelle dans le $K[[X]]$-module $R \otimes_K K[[X]]$, en associant chacun des u_n à une puissance de X. On note cette série $U(X)$. Elle s'écrit :

$$U(X) = \sum_{n\geqslant 0} u_n X^n.$$

Soient maintenant R_0 et R_1 deux K-espaces vectoriels et une application bilinéaire de $R_0 \times R$ dans R_1. On note cette application bilinéaire comme le produit (que l'on ne suppose pas nécessairement commutatif). On définit la notion de suite linéairement engendrée (sur R_0).

Définition 6.1 (Générateur linéaire). *Soit $u \in R^{\mathbb{N}}$. On dit que u est linéairement engendrée (sur R_0) par le polynôme $P \in R_0 \otimes_K K[X]$ si et seulement si P est non nul et :*

$$P(X)U(X) \in R_1 \otimes_K K[X].$$

Dans un contexte non commutatif, il convient de distinguer entre générateur linéaire à gauche (comme énoncé ci-dessus) et générateur linéaire à droite. Pour un générateur à droite, l'application produit va de $R \times R_0$ dans R_1.

D'autres définitions peuvent amener à considérer le polynôme réciproque de P comme générateur. Les diverses formulations sont équivalentes. Celle employée dans ce mémoire est la plus pratique pour le développement des chapitres concernés. Cette définition appelle la présentation de quelques exemples.

6.1.2 Exemples

- Quand on est dans le cas simple où $R = R_0 = R_1 = K$ et que l'on considère la suite définie par $u_n = \alpha^n$ pour $\alpha \in K$, alors u est linéairement engendrée par $P = 1 - \alpha X$, puisque $P(X)U(X) = 1$. P étant d'ailleurs inversible dans $K[[X]]$, on peut écrire $U(X) = \frac{1}{P(X)}$.
- Dans le cas un peu similaire où R est une algèbre de type fini non nécessairement commutative sur K (une algèbre de matrices par exemple, ou bien simplement une extension de corps), mais $R_0 = K$ (et $R_1 = R$), le résultat se généralise ainsi :

  Proposition 6.2. *Soit R une K-algèbre de type fini. Soit $\alpha \in R$, et μ le polynôme minimal de α sur K. Soit u la suite des puissances de α, $u_n = \alpha^n$. La suite u est linéairement engendrée par les polynômes P qui sont multiples du polynôme réciproque $\widehat{\mu}$ de μ.*

  Démonstration. Pour vérifier cette assertion, il suffit d'écrire le candidat générateur sous la forme $P(X) = \sum_{k=0}^{\deg P} p_k X^k$. Le coefficient de degré n du produit $P(X)U(X)$ s'écrit, pour $n \geqslant \deg P$:

$$\begin{aligned}[X^n]P(X)U(X) &= \sum_{i=n-\deg P}^{n} \alpha^i p_{n-i}, \\ &= \alpha^{n-\deg P}\widehat{P}(\alpha).\end{aligned}$$

  Cette expression s'annule à partir du rang n si et seulement si $X^{n-\deg P}\widehat{P}(X)$ est multiple de μ, ce qui équivaut à P multiple de $\widehat{\mu}$ et $n \geqslant \deg \mu$. ■

- Un dernier exemple est celui où R est un espace de matrices rectangulaires $R = K^{m\times n}$. On va énoncer l'existence d'un générateur linéaire à droite, puisque c'est cette formulation qui sera retenue ensuite. Soit donc $R_0 = K^{n\times n}$, $R_1 = R$, le produit étant bien entendu le produit de matrices. Une suite de matrices $a \in R^{\mathbb{N}}$ est linéairement engendrée par $f(X) \in R_0 \otimes_K K[X] = K[X]^{n\times n}$ si et seulement si le produit $A(X)f(X)$ est dans $R_1 \otimes_K K[X] = K[X]^{m\times n}$ (on a pris la notation $A(X) = \sum_{n\geqslant 0} a_n X^n$). Un tel $f(X)$ est appelé *générateur linéaire matriciel* (*matrix generating polynomial*). Si l'on reprend le même énoncé avec $R_0 = K^n$ et $R_1 = K^m$, on obtient ce que l'on appelle un *générateur linéaire vectoriel* (*vector generating polynomial*).

6.1.3 Degré

Pour l'étude des générateurs linéaires et des polynômes minimaux dans les cas « classiques » qui correspondent au deuxième exemple ci-dessus, la notion de *degré* est une notion importante. Nous en introduisons ici une généralisation aux $K[X]$-modules que l'on considère, à savoir les modules $R_0 \otimes_K K[X]$ et $R_1 \otimes_K K[X]$. Nous allons voir aussi que cette notion doit être assortie d'une grandeur auxiliaire notée δ, afin de conserver tout le pouvoir d'expression auquel on est habitué.

Définition 6.3. *Soit K un corps et M un $K[X]$-module de type fini. On appelle* degré *d'un élément $f \in M$, noté $\deg f$, la valeur maximale du degré des coefficients de l'écriture de f dans une base quelconque de M sur $K[X]$.*

Cette définition conserve toutes les propriétés du degré. On a pris soin de préciser que l'on considérait des $K[X]$-modules *de type fini*, mais cette précision devra être entendue comme implicite dans ce qui suit.

On introduit la notation suivante :

Notation 6.4. *Soient f et g deux éléments de $K[X]$-modules (pas nécessairement le même). On note $\delta(f,g)$ la quantité* $\max(\deg f, 1+\deg g)$.

Cette notation est particulièrement destinée au cas où f est un générateur linéaire de la suite a représentée par la série formelle $A(X)$, de telle sorte que $f(X)A(X)$ soit de degré fini. On s'intéresse alors à la quantité $\delta(f, fA)$ (abrégée $\delta(f)$ lorsque le contexte est clair). Cette quantité permet de compléter l'information cachée par notre définition là où interviennent des polynômes réciproques. En effet, le générateur linéaire de la suite des puissances d'un élément de polynôme minimal P est le polynôme réciproque $\widehat{P}$. Mais le passage de P à $\widehat{P}$ n'est pas univoque (en effet $\widehat{P} = \widehat{XP}$) : on manque parfois d'information si l'on considère seulement $\widehat{P}$. En revanche, il est assuré que $\delta(\widehat{P}) = \deg P$. Cela découle de la démonstration de la proposition 6.2 vue plus haut. Si $P = X^k Q$, avec $Q(0) \neq 0$, alors $\widehat{P} = \widehat{Q}$ et $\widehat{\widehat{Q}} = Q$. Le coefficient de degré n de $\widehat{P}(X)A(X)$ est nul si et seulement si $X^{n-\deg\widehat{P}}\widehat{\widehat{P}}$ est multiple de $P = X^k\widehat{\widehat{P}}$, c'est-à-dire si et seulement si $n \geqslant \deg P$. Donc $\delta(\widehat{P}, \widehat{P}(X)A(X)) = \deg P$.

6.1.4 Minimalité

Dans le cadre des deux premiers exemples mentionnés plus haut, on a $R_0 = K$. On peut alors parler *du* générateur linéaire, à savoir le polynôme minimal. En effet, il est aisé de montrer que les générateurs forment un idéal de $K[X]$, qui est un anneau principal. On peut donc en extraire *le* générateur minimal. Cela nous permet d'énoncer le résultat suivant.

Proposition 6.5. *Soit $A \in K[[X]]$. Soit $I = (f_0)$ l'idéal des générateurs linéaires de A. Tout générateur linéaire f de A s'écrit sous la forme $f = kf_0$ (avec $k \in K[X]$) et vérifie donc :*

$$f_0 \mid f \qquad \delta(f_0) \leqslant \delta(f),$$
$$\delta(f) - \deg f = \delta(f_0) - \deg f_0.$$

Dans le cas général, l'ensemble des générateurs forme un sous-$K[X]$-module de $R_0 \otimes_K K[X]$. Ce dernier est un module sur un anneau principal, donc ses sous-modules ont une base. Néanmoins, il n'y a pas de raison de privilégier un des éléments de cette base par rapport aux autres. On ne peut donc pas nécessairement dans ce cas parler *du* générateur. Toutefois, dans le troisième exemple mentionné plus haut, une telle formulation est justifiée. En effet, pour $R_0 = K^{n\times n}$, chaque colonne d'un générateur linéaire peut servir de générateur linéaire sur K^n. On peut donc dire qu'un générateur linéaire sur $K^{n\times n}$ est minimal si et seulement si ses colonnes engendrent l'ensemble des générateurs linéaires sur K^n. Un tel générateur linéaire matriciel minimal est unique, à l'action près de $\mathrm{GL}(K[X]^{n\times n})$ à gauche. Il est même possible de construire une forme normale relativement à cette action.

6.1.5 Descriptions en fractions rationnelles

Dans la suite de ce qui a été exposé précédemment, si R_0 est une K-algèbre, il se peut que le générateur linéaire $P(X)$ soit inversible dans $R_0 \otimes_K K[[X]]$ (il suffit pour cela que

$P(0)$ soit inversible). Un tel générateur $P(X)$ est dit *unimodulaire*. Dans un tel cas, si l'on note $V(X) = P(X)U(X)$, on peut écrire $U(X) = P(X)^{-1}V(X)$. On appelle une telle écriture une *description en fraction rationnelle* (ici à gauche). On peut similairement définir une telle description à droite.

6.1.6 Générateur linéaire et polynôme minimal

Nous avons évoqué à plusieurs reprises le cas où $R_0 = K$. Dans cette situation, il est parfois plus commode d'employer la terminologie équivalente de polynôme minimal. Pour une suite $u = (u_n)_{n\in\mathbb{N}} \in R^{\mathbb{N}}$, on note l'opérateur de décalage σ défini par $\sigma(u) = (u_{n+1})_{n\in\mathbb{N}}$. Sur la série formelle associée, $\sigma(U)$ est la partie entière de la série formelle $\frac{U}{X}$. On définit le polynôme minimal comme suit :

Définition 6.6 (Polynôme minimal d'une suite). *Le polynôme minimal de la suite $u = (u_n)_{n\in\mathbb{N}} \in R^{\mathbb{N}}$ est le générateur de l'idéal des polynômes $P \in K[X]$ vérifiant*

$$P(\sigma)(u) = 0.$$

Le polynôme minimal est la réciproque du générateur linéaire. C'est l'objet de l'énoncé suivant.

Proposition 6.7. *Soit $u = (u_n)_{n\in\mathbb{N}} \in R^{\mathbb{N}}$ une suite, soit f son générateur linéaire minimal et μ son polynôme minimal. Alors on a :*

$$\mu = X^{\delta(f)} f\left(\frac{1}{X}\right).$$

Démonstration. La démonstration de la proposition 6.2 s'adapte très exactement à notre situation et permet d'obtenir le résultat. ■

6.2 L'algorithme de Wiedemann

6.2.1 Présentation et principe

L'algorithme de Wiedemann a été introduit en 1986 [Wie86]. C'est le premier algorithme spécifiquement conçu pour être appliqué dans le cadre des systèmes linéaires définis sur les corps finis, à la différence de l'algorithme de Lanczos qui a été adapté d'une méthode numérique ancienne.

L'algorithme de Wiedemann est un algorithme probabiliste de type Monte-Carlo : avec une faible probabilité, il peut ne pas produire de résultat correct. Le principe de fonctionnement de l'algorithme permet de relier cette probabilité aux propriétés de réduction de la matrice d'entrée considérée.

Nous nous intéressons toujours à la résolution du système linéaire homogène $Bw = 0$, où B est singulière de taille $N \times N$ définie sur un corps fini K, possédant γ coefficients non-nuls par ligne.

Comme la matrice B est singulière, X divise son polynôme minimal μ. Il existe donc un polynôme μ' et un entier k tels que $\mu = X^{k+1}\mu'$, $k \geqslant 0$ et $\mu'(0) \neq 0$. Dans ce cas, on a la proposition suivante.

Proposition 6.8. *Soit $k+1$ la valuation en X du polynôme minimal μ, et $\mu = X^{k+1}\mu'$. Avec probabilité $1 - 1/q^{m_0}$, où m_0 est la multiplicité de la valeur propre 0 de B, un vecteur x de K^N est tel que $\mu'(B)(x) \neq 0$. On peut obtenir à partir d'un tel vecteur un vecteur de* $\operatorname{Ker} B$.

Démonstration. Notons $E = \operatorname{Ker} B^{k+1}$ le sous-espace caractéristique associé à la valeur propre 0, et F un supplémentaire de E dans K^N stable par B. Comme $\mu'(0) \neq 0$, l'endomorphisme $\mu'(B)$ est nul sur F, et inversible sur E. Il s'ensuit que $\mu'(B)x = 0$ si et seulement si $x \in F$. La dimension de E étant m_0, on a la probabilité annoncée.

Soit maintenant un x tel que $y = \mu'(B)x \neq 0$. On sait que $B^{k+1}y = 0$. Donc il existe un entier $j \in [\![0 \dots k]\!]$ tel que $B^j y \neq 0$, et $B^{j+1}y = 0$. Le vecteur $B^j y$ est donc un vecteur non nul de $\operatorname{Ker} B$. ■

L'algorithme de Wiedemann utilise cette dernière propriété, d'abord en calculant μ (ou plus exactement un diviseur de μ, souvent égal à μ lui-même), puis en déduisant un vecteur de $\operatorname{Ker} B$.

On s'intéresse au sous-espace vectoriel des itérés par B d'un vecteur aléatoire y (appelé sous-espace de Krylov [Kry31]). Ce sous-espace est un sous-espace stable par B. Il est donc possible de définir le polynôme minimal de B sur ce sous-espace. On le note μ_y. Ce polynôme est aussi, en accord avec la définition 6.6, le polynôme minimal de la suite des $B^i y$. C'est *a priori* un diviseur de μ. La proposition 6.8 nous indique qu'avec probabilité $1 - 1/q^{m_0}$, on a $X \mid \mu_y$.

Cela nous laisse entrevoir la possibilité de calculer μ_y par le biais d'un calcul de générateur linéaire. Mais pour rendre ce calcul faisable efficacement, on est amené à considérer non pas une suite de vecteurs, mais la suite constituée des *scalaires* :

$$a_k = x^T B^k y,$$

où x est un second vecteur aléatoire. Ces coefficients a_k peuvent être aisément calculés par itération des opérations $v \leftarrow Bv$, $a_k \leftarrow x^T v$. Cette suite admet un polynôme minimal (au sens de la définition 6.6) que nous noterons $\mu_{x,y}$.

Nous verrons au chapitre 8 que pour la suite des a_k, à coefficients dans un corps fini, représentée par la série $A(X) \in K[[X]]$, des algorithmes comme l'algorithme de Berlekamp-Massey permettent de calculer le générateur linéaire F minimal à partir de $2\delta(F)$ termes de la suite et en temps $O(\delta(F)^2)$ (on a noté $\delta(F) = \delta(F, AF)$, où δ est conforme à la définition 6.4). En vertu de la proposition 6.7, ce générateur linéaire minimal est relié au polynôme $\mu_{x,y}$: si l'on note $\ell = \delta(F) - \deg F$, le polynôme $X^\ell \widehat{F}$ est égal à $\mu_{x,y}$ et divise donc μ_y, et μ. En particulier, on a $\ell \leqslant k+1$ et $\widehat{F} \mid \mu'$.

Si on a la chance d'avoir $X^\ell \widehat{F} = \mu$, alors on choisit un vecteur aléatoire v. Le vecteur $\widehat{F}(B)v$ permet, selon la proposition 6.8, de dériver un vecteur non nul du noyau avec forte probabilité. Le calcul de $\widehat{F}(B)v$ s'effectue encore une fois par répétition du produit matrice-vecteur (on peut évaluer l'expression polynomiale avec un schéma du type Hörner).

6.2.2 Récupération des échecs et implantation

Nous disposons maintenant des outils essentiels pour décrire l'implantation de l'algorithme de Wiedemann. Toutefois, pour prévoir les cas où l'on n'obtient pas $X^\ell \widehat{F} = \mu$, mais plutôt une divisibilité stricte, il faut s'intéresser à la possibilité de réutiliser l'information obtenue par le calcul de $X^\ell \widehat{F}$. Il se trouve que l'algorithme s'en accommode très bien.

Algorithme **Plain-Wiedemann**
Entrée : Une matrice $B \in K^{N\times N}$ singulière.
Un couple (λ,ν) tel que $X \nmid \nu$ et $X^\lambda\nu \mid \mu$.
Sortie : Un élément de $\operatorname{Ker} B$.

```
{
    x, z=vecteurs aléatoires dans K^N;
    z=ν(B)z;  y = B^λ z;  v = y;
    for(k=0;k<2(N − deg ν);k++) { a[k]=x^T v;  v=Bv; }
    F=(générateur linéaire des a_i);                    /* tel que X ∤ F */
    ℓ=δ(F) − deg F;
    v=0;
    for(k=0;k<=deg F;k++) {                    /* On calcule v = F̂(B)z */
        v = Bv;
        v = v + F_k z;
    }
    if (v!=0) for(k=0;k<λ + ℓ;k++) {
        u = Bv;
        if (u==0) return v;
        v = u;
    }
    return PlainWiedemann(B,λ + ℓ,νF̂);                  /* Improbable */
}
```

Programme 6.1: Algorithme de Wiedemann

En effet, supposons qu'à l'entrée de l'algorithme on dispose, outre la matrice B, d'une information partielle sur son polynôme minimal. On présente cette information sous la forme d'un couple (λ,ν) tel que $X \nmid \nu$ et $X^\lambda\nu \mid \mu$. Alors, le vecteur y à partir duquel est construite la suite des a_k n'est pas construit au hasard : on le choisit sous la forme $B^\lambda z$, où $z = \nu(B)u$ et u est un vecteur aléatoire. Il est alors clair que le polynôme minimal $P = X^\ell\widehat{F}$ de la suite des a_k est un diviseur (peut-être strict) du quotient $\frac{\mu}{X^\lambda\nu}$. Par conséquent, on a :

$$\nu\widehat{F} \mid \mu' \quad \text{et} \quad \lambda+\ell \leqslant k+1.$$

Puisque $\deg\widehat{F} \leqslant (N-\deg\nu)$, le générateur linéaire F peut être calculé à partir de seulement $2(N-\deg\nu)$ coefficients de la suite des a_k. Par ailleurs, on peut déduire de l'identité précédente que le vecteur $v = \widehat{F}(B)z$ est un bon candidat pour utiliser la proposition 6.8. Si d'aventure il n'était pas possible de déduire un vecteur non nul du noyau à partir de v, on peut recommencer la procédure avec l'information augmentée $(\lambda+\ell,\nu\widehat{F})$.

De cette façon, on a transformé la description « Monte-Carlo » de l'algorithme de Wiedemann, effectuée plus haut, en une version « Las-Vegas ». Il n'y a à cela rien d'extraordinaire, si ce n'est que l'on recycle une quantité importante d'information lors des répétitions de la procédure. Initialement, on commence l'algorithme avec $(\lambda,\nu) = (1,1)$.

La description ci-dessus permet d'établir le pseudo-code 6.1. Il apparaît clairement dans ce programme que B n'intervient pas autrement que sous forme de boîte noire.

6.2.3 Justification

Il convient d'expliquer dans quelle mesure le commentaire apparaissant dans le programme 6.1 sur la faible probabilité des appels récursifs est justifié.

Deux situations distinctes peuvent amener à un appel récursif. La première est que le vecteur v soit non nul, mais que le vecteur $B^{\lambda+\ell}v$ calculé par la dernière boucle du programme ne soit pas non plus le vecteur nul. Ainsi, on n'est pas capable d'obtenir un élément du noyau. Cela n'est possible que si le polynôme $X^\ell \widehat{F}$ calculé, aussi noté $\mu_{x,y}$, n'est qu'un diviseur strict de μ_y. Dans ce cas, même si l'on a $x^T B^\ell \widehat{F}(B)y = 0$, on n'a pas $B^\ell \widehat{F}(B)y = B^{\lambda+\ell}(\widehat{F}\nu)(B)u = 0$. Nous détaillons plus loin la probabilité d'avoir $\mu_{x,y} \neq \mu_y$.

La seconde possibilité d'obtenir une récursion de l'algorithme est la situation où le vecteur $v = \widehat{F}(B)z$ calculé est nul. Clairement, cette situation exclut la précédente. On montre le résultat suivant.

Proposition 6.9. *Supposons $B^{\lambda+\ell}v = 0$. Alors $v = 0 \Leftrightarrow u \in \operatorname{Ker} \mu'(B)$.*

Démonstration. Reprenons la décomposition $K^N = E \oplus F$ introduite dans la démonstration de la proposition 6.8, où E est le sous-espace caractéritique de B associé à la valeur propre 0 (donc $E = \operatorname{Ker} B^{k+1}$), et F un supplémentaire stable. Comme $(\widehat{F}\nu)(B)$ est inversible sur E, si $v = (\widehat{F}\nu)(B)u = 0$ alors u appartient à $F = \operatorname{Ker} \mu'(B)$.

Réciproquement, si $u \in F$, alors $\mu_u \mid \mu'$. Comme $B^{\lambda+\ell}(\widehat{F}\nu)(B)u = 0$, on a aussi $\mu_u \mid (X^{\lambda+\ell}\widehat{F}\nu)$. La combinaison de ces deux propriétés implique $\mu_u \mid (\widehat{F}\nu)$, donc $v = (\widehat{F}\nu)(B)u = 0$. ∎

En appliquant cette propriété conjointement avec la proposition 6.8, il ressort que la probabilité de récursion due à $v = 0$ est $1 - \frac{1}{q^{m_0}}$.

Quelle est la probabilité d'avoir $\mu_{x,y} \neq \mu_y$? Elle peut être calculée exactement. Pour chaque facteur irréductible ϕ du polynôme caractéristique de la matrice B, on appelle sous-espace caractéristique généralisé de B le sous-espace vectoriel $\operatorname{Ker} \phi(B)^\infty$ (il suffit de prendre comme puissance la multiplicité de ϕ dans la factorisation du polynôme caractéristique de B). On note $c_\phi \deg \phi$ la dimension de ce sous-espace. On a alors :

Théorème 6.10. *La probabilité pour que deux vecteurs x et y de K^N soient tels qu'on ait l'égalité $\mu_y = \mu_{x,y}$ est :*

$$\prod_\phi \left(1 - \frac{1}{q^{\deg \phi}} \left(1 - \frac{1}{q^{c_\phi \deg \phi}}\right)\right).$$

Démonstration. Pour démontrer ce théorème, on a recours aux propriétés de réduction de la matrice B. Commençons par justifier cette assertion, en montrant que l'on peut se ramener au cas où le polynôme caractéristique de B est une puissance d'un polynôme irréductible. En effet, si l'on écrit K^N comme somme directe des sous-espaces caractéristiques généralisés C_i, on a, indépendamment de x, y et k :

$$x^T B^k y = \sum_i x_i^T \left. B^k \right|_{C_i} y_i,$$

où les vecteurs x et y ont été décomposés conformément à la somme directe $\bigoplus C_i$. On associe à chacune des suites $x_i^T \left. B^k \right|_{C_i} y_i$ un polynôme minimal respectif $\mu_{x,y}^{(i)}$. On sait par construction

que ces polynômes sont premiers entre eux, car les C_i sont les sous-espaces caractéristiques généralisés. Il s'ensuit que le polynôme minimal $\mu_{x,y}$ est le ppcm des $\mu^{(i)}_{x,y}$ et donc que $\mu_{x,y} = \mu_y$ si et seulement si l'égalité est vérifiée localement pour tout i. Il s'ensuit que si la propriété de probabilité que l'on cherche à montrer est vraie localement, elle est vraie globalement. La réciproque est évidente.

Montrons maintenant que l'on peut se contenter d'étudier le cas où $\deg \phi = 1$. Supposons que $\deg \phi > 1$. Soit L le corps de rupture de ϕ, λ une racine de ϕ dans L et σ l'endomorphisme de Frobenius dans L, générateur du groupe de Galois de L sur $\mathbb{F}_q$. Le sous-espace caractéristique $C \otimes_{\mathbb{F}_q} L$ associé à ϕ dans L se décompose sous la forme d'une somme directe de $\deg \phi$ sous-espaces caractéristiques $\Gamma, \Gamma^\sigma, \ldots, \Gamma^{\sigma^{\deg\phi-1}}$, où l'action de B restreinte à Γ est λid. Il existe un $\mathbb{F}_q$-isomoprhisme de C dans Γ, noté γ, tel qu'un vecteur y de C s'écrive sous la forme

$$y = \gamma(y) + \gamma(y)^\sigma + \cdots + \gamma(y)^{\sigma^{\deg\phi-1}}.$$

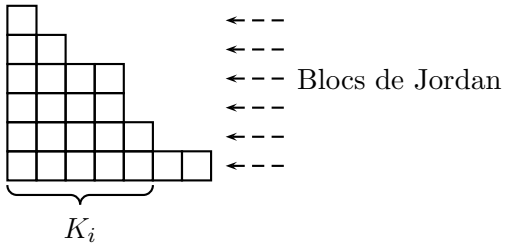
De façon semblable au calcul déjà fait, on a alors :

$$x^T B^k y = \sum_i \left(\gamma(x)^{\sigma^i}\right)^T \left(\lambda^{\sigma^i}\right)^k \gamma(y)^{\sigma^i}.$$

Comme les $(X - \lambda^{\sigma^i})$ sont premiers entre eux, le polynôme minimal de cette somme est le ppcm des polynômes minimaux. Cela implique que $\mu_{x,y} = \mu_y$ si et seulement si x et y sont tels que dans l'espace Γ, on a l'égalité (car alors elle se transmet aux conjugués). Ce lemme implique qu'il est suffisant de s'intéresser au cas où $\deg \phi = 1$ (on rappelle que γ est un *iso*morphisme).

Pour finir de spécifier notre cas d'étude, il est clair qu'étudier la valeur propre $\lambda = 0$ suffit à traiter le cas général, quitte à changer la base de $K[X]$ en considérant les puissances de $X - \lambda$.

Soit donc B une matrice nilpotente, dans un espace de dimension N. La suite des noyaux itérés $K_i = \operatorname{Ker} B^i$ est croissante, concave, et stationnaire. Si l'on note $d_i = \dim K_i - \dim K_{i-1}$, on peut écrire la décomposition suivante de K^N, où K_i est somme directe de i sous-espaces de tailles respectives $d_1, \ldots, d_i$. Dans cette écriture, on peut lire les blocs de Jordan généralisés (sous-espaces stables maximaux) sur les lignes, puisque B agit comme le décalage vers la gauche sur cette figure.



Nous cherchons à savoir avec quelle probabilité on a $\mu_{x,y} = \mu_y$. Les choix possibles pour les coordonnées des vecteurs x et y sont au nombre total de q^{2N} (où q, au besoin, est remplacé par le cardinal d'une extension). Parmi ces choix, ceux où y est nul sont vite tranchés puisqu'alors, on a $\mu_{x,y} = \mu_y = 1$. Ces choix sont au nombre de q^N. Pour les $q^N - 1$ autres valeurs possibles de y, combien de valeurs de x sont telles qu'on a l'égalité ? Il est facile de répondre à cette question. Écrivons $X^k = \mu_y$. Le vecteur $B^{k-1}y$ est alors un vecteur non nul, avec un certain nombre de

coordonnées non nulles *indépendantes* (elles sont extraites des coordonnées de y). Le produit scalaire $x^T B^{k-1} y$ s'écrit comme une *forme linéaire* en les coordonnées correspondantes du vecteur x. Cette forme linéaire prend uniformément toutes les valeurs possibles dans $\mathbb{F}_q$ et s'annule donc pour une proportion $\frac{1}{q}$ des valeurs de x. Il s'ensuit que la probabilité d'égalité est :

$$\begin{aligned}
\wp &= q^{-2N}\left(q^N + (q^N-1)q^N\left(1-\frac{1}{q}\right)\right),\\
\wp &= \frac{1}{q^N} + \left(1-\frac{1}{q}\right)\left(1-\frac{1}{q^N}\right),\\
\wp &= 1 - \frac{1}{q}\left(1-\frac{1}{q^N}\right).
\end{aligned}$$

Ce résultat correspond exactement à la propriété que l'on veut montrer, en tenant compte en outre des cas où $\mathbb{F}_q$ est remplacé par une extension finie. ■

En combinant les résultats de la proposition 6.9 et du théorème 6.10, on peut donner exactement la probabilité de succès de l'algorithme de Wiedemann, sans tenir compte des récursions. Le cas de la valeur propre 0 doit être traité à part. On a $m_0 = c_X$ par définition et en vertu de la proposition 6.9, la probabilité $\wp$ que l'on vient de calculer est remplacée dans le produit par l'expression $\left(1-\frac{1}{q}\right)\left(1-\frac{1}{q^{c_X}}\right)$. La probabilité de succès « du premier coup » est alors :

$$\left(1-\frac{1}{q}\right)\left(1-\frac{1}{q^{c_X}}\right)\prod_{\phi\neq X}\left(1-\frac{1}{q^{\deg\phi}}\left(1-\frac{1}{q^{c_\phi \deg\phi}}\right)\right).$$

Il est même possible d'améliorer cette probabilité d'un facteur $\left(1-\frac{1}{q}\right)$: en effet, si l'on a $\mu_{x,y} \neq \mu_y$ sur le sous-espace caractéristique associé à la valeur propre 0, ce n'est pas très grave : il est possible de continuer à calculer certaines itérations de B pour obtenir le vecteur nul. Le nombre d'itérations que l'on doit ainsi « rajouter » est borné par $N - \delta(F\nu)$.

Ce résultat donne exactement la probabilité de succès de la version « Monte-Carlo » de l'algorithme. Pour statuer sur le nombre de récursions nécessaires à l'algorithme « Las-Vegas », ou encore sur le nombre de récursions nécessaires pour calculer μ_B, il faut des résultats plus précis. Une amorce d'une telle précision des résultats peut se trouver dans [BGL03].

6.3 L'algorithme de Wiedemann par blocs

Nous avons présenté l'algorithme de Wiedemann pour la résolution de systèmes linéaires sur les corps finis. Cet algorithme présente l'inconvénient d'être trop séquentiel, dans le sens où les coefficients $a_i = x^T B^i y$ ne peuvent, par construction, être calculés que un à un au cours de l'algorithme. On aimerait mettre en place une version par blocs de cet algorithme, pour bénéficier des avantages énoncés en 5.3.2. Cette version par blocs présente l'avantage que la probabilité de succès est augmentée, à l'instar de ce qui se passe avec l'algorithme de Lanczos.

6.3.1 Introduction de blocs de vecteurs

Pour obtenir une version par blocs de l'algorithme de Wiedemann, on remplace les vecteurs x et z par deux *blocs* de vecteurs. Ces blocs sont choisis de tailles respectives m et n, où m et

n sont deux entiers arbitraires. Bien sûr, plusieurs changements se déduisent. En particulier, la suite des a_k, tout en gardant la même formulation :

$$a_k = x^T B^k y,$$

est désormais une suite de *matrices* (on a toujours $y = Bz$). La série qui la représente est $A(X) = \sum_k a_k X^k$. Elle appartient désormais à $K[[X]]^{m\times n}$.

En formulant ainsi la généralisation de l'algorithme, on voit que la formulation adoptée plus haut pour l'algorithme de Wiedemann ne peut pas tout à fait se transposer. Pour reprendre les notations employées dans la section 6.1 et particulièrement en 6.1.6, considérer $R_0 = K$ afin de pouvoir utiliser la proposition 6.7 nous force à nous placer dans un contexte où les bénéfices de la version par blocs n'apparaissent pas : on ne voit pas comment faire moins de produits matrice $\times$ bloc de vecteurs que l'on fait de produits matrice $\times$ vecteur dans l'algorithme de Wiedemann.

Nous allons voir pourtant, en détaillant quels sont les outils dont nous aurons besoin pour mener le calcul, que ce nombre de produits peut être réduit par l'usage de blocs. En effet, seul le calcul des L premiers coefficients de A nous est utile, L valant $\frac{N}{m} + \frac{N}{n} + O(1)$. Ainsi, nous pouvons bénéficier de tous les avantages évoqués en 5.3.2. Le temps de calcul peut être réduit, en effectuant 32 produits *simultanément* lorsque le corps de base est $\mathbb{F}_2$, ou bien en distribuant le calcul lorsque le corps de base est $\mathbb{F}_p$.

6.3.2 La notion de générateur linéaire à utiliser

Afin de mettre en évidence que seulement L coefficients de $A(X)$ sont utiles, nous devons nous placer dans le contexte des générateurs linéaires vectoriels matriciels, tel qu'on les a définis page 104. La quantité que nous souhaitons obtenir est un vecteur à coefficients polynomiaux, noté $F(X) \in K[X]^{n\times 1}$, vérifiant :

$$A(X)F(X) = G(X) \in K[X]^{m\times 1}.$$

En accord avec le formalisme qui a été mis en place en 6.1, ce vecteur $F(X)$ est un générateur linéaire vectoriel. Il est trivial de constater que de cette façon, prendre $m = n = 1$ nous fait retomber sur l'algorithme de Wiedemann tel que décrit précédemment.

Hypothèses à vérifier

La façon dont un tel générateur linéaire vectoriel peut être calculé est l'objet du chapitre 8 de ce mémoire. Pour l'instant, nous n'entrons pas dans le détail de ce calcul, que l'on considère comme un outil. La proposition 8.7 et le théorème 8.6 fixent les hypothèses que nous devons remplir. On peut résumer celles-ci en l'énoncé suivant, qui sera démontré de manière constructive au chapitre 8.

Proposition. *Soit $A \in K[[X]]^{m\times n}$. On suppose qu'il existe une description en fraction rationnelle à gauche $D^{-1}(X)N(X)$ (cf page 105), avec $\delta(D) = d$. Soit s le plus petit entier tel que les colonnes de $a_0, \ldots, a_{s-1}$ engendrent K^m.*
Alors, en connaissant les $L = s + \left\lceil \frac{m+n}{n} d \right\rceil$ premiers coefficients de $A(X)$ (i.e. $A \bmod X^L$), on peut calculer de manière déterministe un générateur linéaire vectoriel F vérifiant $\delta(F) \leqslant s + \left\lceil \frac{m}{n} d \right\rceil$.

Dans la proposition suivante, seule l'existence de la description en fraction rationnelle est supposée. Les matrices N et D ne sont pas supposées connues.

Nous devons déterminer, dans le cas de l'algorithme de Wiedemann par blocs, quelles sont les valeurs des paramètres s et d que l'on peut s'attendre à rencontrer. Ces paramètres conditionnent la valeur de L, qu'on a évoquée comme étant égale à $\frac{N}{n} + \frac{N}{m} + O(1)$. Les discussions que nous mènerons concernant ces paramètres introduisent implicitement autant d'hypothèses, sur lesquelles nous reviendrons en 6.3.5.

Valeur de s

Le paramètre s est de l'ordre de $\lceil \frac{m}{n} \rceil$. En effet, l'ensemble des vecteurs colonnes des s premiers coefficients de A est de cardinal sn. Donc s doit être au moins égal à $\lceil \frac{m}{n} \rceil$ pour que le rang de cette famille de vecteurs atteigne m. Hormis dans les cas dégénérés, on a égalité. Le paramètre s est de toute façon aisément calculable. Il est à noter que si aucune valeur de s ne convient, alors le rang de l'ensemble des colonnes de $A(X)$ est strictement inférieur à m. Dans ce cas, un autre choix de x convient probablement. On rappelle que x est le bloc de vecteurs choisi aléatoirement pour formes les coefficients $a_k = x^T B^k y$.

Valeur de d

Nous montrons que l'on peut prendre pour d le plus petit entier tel que l'espace engendré par les colonnes des matrices $x, \ldots, (B^T)^{d-1}x$ est maximal. En effet, si tel est le cas, chacune des colonnes de la matrice $(B^T)^d x$ peut s'exprimer comme combinaison linéaire des $(B^T)^i x_j$ pour $0 \leqslant i < d$, $1 \leqslant j \leqslant m$. On a donc des coefficients $\lambda_{i,j,k}$ tels que

$$(B^T)^d x_k = \sum_{j=1}^{m} \sum_{i=0}^{d-1} \lambda_{i,j,k} (B^T)^i x_j.$$

Appelons $\Omega(X)$ la matrice de taille $m \times m$ dont le terme en position (j,k) vaut :

$$\Omega_{i,j} = \sum_{i=1}^{d} \lambda_{d-i,j,k} X^i.$$

Cette matrice nous permet d'écrire de façon synthétique l'identité précédente :

$$\begin{aligned}
(B^T)^d x_k &= [X^d] \sum_{j=1}^{m} \sum_{i=0}^{d-1} (B^T)^i x_j X^i \lambda_{i,j,k} X^{d-i}, \\
(B^T)^d x &= [X^d] \left(\left(\sum_{i \geqslant 0} X^i (B^T)^i x \right) \Omega(X) \right), \\
0 &= [X^d] \left(\left(\sum_{i \geqslant 0} X^i (B^T)^i x \right) (\Omega(X) - I_m) \right).
\end{aligned}$$

On peut refaire ces calculs en multipliant par n'importe quelle puissance de B^T à gauche. Cela implique que tous les termes de degré $\geqslant d$ du produit ci-dessus sont nuls. Par multiplication par y à gauche, puis par transposition, on déduit que $(I_m - \Omega^T)A$ est dans $K[X]^{m \times n}$, et que

son degré est strictement inférieur à d. Comme $D = I_m - \Omega^T$ est unimodulaire (car $\Omega(0) = 0$), on a bien une description en fraction rationnelle à gauche, vérifiant $\delta(D) \leqslant d$.

La valeur que l'on a prise pour d fait que l'on peut s'attendre à avoir $d \approx \lceil \frac{N}{m} \rceil$.

Si l'on applique ces valeurs « typiques » de s et d, Le nombre de termes de A à considérer est :

$$\begin{aligned} L &= s + \left\lceil \frac{m+n}{n} d \right\rceil, \\ &\approx \frac{m+n}{mn} N, \\ &\approx \frac{N}{m} + \frac{N}{n}. \end{aligned}$$

Quant à la valeur $\delta(F)$ obtenue, elle vaut $s + \lceil \frac{m}{n} d \rceil$, soit à peu près $\frac{N}{n}$.

6.3.3 Obtention d'un vecteur du noyau

Dans le cadre de l'algorithme de Wiedemann, nous avons calculé le générateur linéaire de la suite des a_k et formé ensuite le vecteur $\widehat{F}(B)z$, en espérant en déduire un vecteur du noyau de B. Avec l'introduction de générateurs linéaires vectoriels, la nature de $\widehat{F}$ rend désormais impossible la prise en compte de cette quantité, mais l'esprit reste le même. On calcule le vecteur v défini par :

$$v = \sum_{i=0}^{\deg F} B^{\deg F - i} z[X^i]F.$$

Ce vecteur est bel et bien une généralisation de la quantité $\widehat{F}(B)z$ dans l'algorithme de Wiedemann sans blocs. À l'instar de ce qui se passait précédemment, v nous permet d'obtenir un élément du noyau de F (pourvu que l'on ait $v \neq 0$). Pour cela, nous requérons une propriété supplémentaire du bloc de vecteurs x. Le coefficient de degré k du produit $A(X)F(X)$, pour $k \geqslant \deg F$, vaut :

$$\begin{aligned} [X^k](AF) &= x^T \sum_{i=0}^{\deg F} B^{k-i} y[X^i]F, \\ &= x^T B^{k-\deg F+\lambda} \sum_{i=0}^{\deg F} B^{\deg F - i} z[X^i]F, \\ &= x^T B^{k-\deg F+\lambda} v. \end{aligned}$$

Par construction, ce coefficient est nul pour $k \geqslant \delta(F)$. Si l'on note donc comme on l'a fait pour l'algorithme de Wiedemann $\ell = \delta(F) - \deg F$, le vecteur $B^{\ell+\lambda}v$ est orthogonal à tous les vecteurs de la forme $(B^T)^i x_j$. Si ces vecteurs engendrent K^N, cela implique $B^{\ell+\lambda}v = 0$.

6.3.4 Structure de l'implantation

Le programme 6.2 donne un exemple d'implantation de l'algorithme de Wiedemann par blocs. Un tel programme ressemble assez à la façon dont une implantation pourrait être réalisée sur $\mathbb{F}_2$. Si le corps de base est $\mathbb{F}_q$, on a dit que notre intérêt était la parallélisation ou la distribution, donc le programme 6.2 qui ne rentre pas dans ces considérations doit être regardé uniquement en tant que trame générale.

Algorithme **Block-Wiedemann**
Entrée : Une matrice $B \in K^{N\times N}$ singulière.
Deux paramètres de blocs m et n.
Sortie : Un élément de $\operatorname{Ker} B$.

```
{
    x=matrice aléatoire dans K^{N×m}; z=matrice aléatoire dans K^{N×n};
    y = Bz; v = y;
    for(k=0;k<N/m + N/n + O(1);k++) { a[k]=x^T v; v=Bv; }
    F=(générateur linéaire des a_i); ℓ=δ(F) − deg F;
    v=0;
    for(i=0;i<=deg F;k++) {
        v = Bv;
        v = v + z[X^i]F;
    }
    if (v!=0) for(k=0;k<1 + ℓ;k++) {
        u = Bv;
        if (u==0) return v;
        v = u;
    }
    return FAILED;                                   /* Improbable */
}
```

Programme 6.2: Algorithme de Wiedemann par blocs

6.3.5 Correction de BW

La présentation des valeurs attendues des paramètres s et d, ainsi que la justification de l'existence d'un vecteur du noyau laisse quelques zones d'incertitude. Nous devons les lever pour garantir que l'algorithme tel qu'on le présente dans le programme 6.2 produit un résultat non trivial.

Pour discuter de ces différents points, nous introduisons quelques notations.

Notation 6.11.
- $\mathcal{K}_y(B)^{(r)} = \left\langle \left\{ B^k y_j,\ j \in [\![1 \dots n]\!], k \in [\![0 \dots r-1]\!] \right\} \right\rangle$, *et* $\mathcal{K}_y(B) = \mathcal{K}_y(B)^{(\infty)}$.
- $\mathcal{K}_x(B^T)^{(r)} = \left\langle \left\{ (B^T)^k x_i,\ i \in [\![1 \dots m]\!], k \in [\![0 \dots r-1]\!] \right\} \right\rangle$, *et* $\mathcal{K}_x(B^T) = \mathcal{K}_x(B^T)^{(\infty)}$.

Nous avons fait les hypothèses suivantes :
- $\dim \left\langle \left\{ x^T v,\ v \in \mathcal{K}_y(B)^{(s)} \right\} \right\rangle = \dim \left\langle \left\{ x^T v,\ v \in \mathcal{K}_y(B) \right\} \right\rangle = m$.
- $\dim \mathcal{K}_x(B^T) = \dim \mathcal{K}_x(B^T)^{(d)}$ (définition de d), et $d \approx \frac{N}{m}$.
- $\left\{ w \in \mathcal{K}_z(B) \,\middle|\, \forall u \in \mathcal{K}_x(B^T),\ u^T v = 0 \right\} = \{0\}$ (pour $B^{l+\lambda} v = 0$).

Par ailleurs, nous n'avons pas donné de condition permettant de garantir que le vecteur v produit est non nul.

Remarquons que la dernière des conditions que l'on vient d'énumérer est plus subtile que la simple assertion $\dim \mathcal{K}_x(B^T) = N$. En effet, nous pouvons nous satisfaire de cette formulation plus précise. Pour faire l'analogie avec l'algorithme de Wiedemann simple, la supposition était bel et bien $\mu_{x,y} = \mu_y$ et non pas $\langle x, B^T x, \dots \rangle = K^N$, qui est une assertion plus forte. Dans [BGL03], ce dernier cas est traité, mais il ne correspond pas *précisément* à notre situation.

Les différentes hypothèses que nous avons faites sont garanties par les analyses réalisées par Kaltofen [Kal95] et Villard [Vil97]. Ces résultats montrent que l'on peut effectivement attendre un résultat non trivial de l'algorithme de Wiedemann par blocs dans les cas suivants :

- Si la caractéristique du corps est grande (par rapport à N).
- Si la matrice B n'est pas trop particulière. On demande par là que la matrice B n'ait pas un nombre anormalement élevé de valeurs propres avec de fortes multiplicités.

Par ailleurs, il ressort des preuves de ces résultats que l'introduction de blocs permet, en définitive, d'augmenter la probabilité de succès de l'algorithme.

6.3.6 Complexité de BW

Nous donnons maintenant une première évaluation de la complexité de l'algorithme de Wiedemann par blocs. Une étude plus approfondie de cette complexité sera menée en 8.7, une fois que nous aurons développé les algorithmes permettant le calcul du générateur linéaire vectoriel F.

On distingue trois étapes dans l'algorithme de Wiedemann par blocs, que l'on note BW1, BW2, BW3.

BW1 est le calcul de la matrice $A = \sum (x^T B^i y) X^i$.

BW2 est le calcul du générateur linéaire matriciel F.

BW3 est le calcul du vecteur v à partir duquel on compte obtenir un vecteur du noyau, v étant donné par la formule $v = \sum_{i=0}^{\deg F} B^{\deg F - i} z[X^i]F$.

Nous souhaitons donner une expression de la complexité faisant ressortir les motivations de l'introduction de blocs dans l'algorithme. Celles-ci ont été introduites en 5.3.2. On s'intéresse donc au nombre d'application de la boîte noire « matrice × *bloc* de vecteurs », en considérant le coût de celle-ci égal à celui du produit matrice × vecteur. Implicitement, on évalue donc la complexité parallèle de l'algorithme lorsque le corps de base est $\mathbb{F}_p$. Notons que la complexité en *communication* de l'algorithme est nulle (en omettant l'initialisation).

Nous évaluons la complexité des étapes BW1 et BW3 lorsque le corps de base est $\mathbb{F}_p$ (pour le cas de $\mathbb{F}_2$, cette approche doit être modifiée, car on se concentre uniquement ici sur les multiplications). Pour cela, nous introduisons deux constantes, M_0 et M_1, définies comme suit.

- M_0 désigne le temps pour multiplier un élément de $\mathbb{F}_p$ par un coefficient de la matrice (qui n'est pas nécessairement un élément *aléatoire* de $\mathbb{F}_p$).
- M_1 désigne le temps pour multiplier deux éléments de $\mathbb{F}_p$.

Souvent, les coefficients de la matrice sont petits, comme on l'a vu en 5.2.1 et 5.2.5. On doit donc considérer par exemple qu'un coefficient de la matrice occupe un seul mot machine, c'est-à-dire qu'il peut être représenté par un entier dans l'intervalle $[\![-2^{31}, 2^{31}-1]\!]$. Dans ce cas, on a typiquement $\mathsf{M}_0 \ll \mathsf{M}_1$.

Proposition 6.12. *Sur le corps de base $\mathbb{F}_p$, en utilisant n machines, la complexité parallèle des étapes* BW1 *et* BW3 *est donnée par :*

BW1 $(\gamma \mathsf{M}_0 + m\mathsf{M}_1)\frac{m+n}{mn}N^2$ *(cf remarque 6.13).*

BW3 $\gamma \mathsf{M}_0 \frac{1}{n} N^2$.

Démonstration. Le nombre de termes de A qui doivent être calculés est $L = \frac{N}{m} + \frac{N}{n} + O(1)$. Pour chacun de ces termes, le coût est celui d'une application de la boîte noire (γN multiplications par des coefficients de la matrice), ainsi que le calcul d'un produit scalaire, soit mN

multiplications d'éléments de $\mathbb{F}_p$ (cf remarque 6.13). En additionnant ces deux composantes, on obtient le coût parallèle annoncé.

Par ailleurs, l'étape BW3 requiert $\delta(F)$ applications de la boîte noire. La valeur de $\delta(F)$ est de l'ordre de $\frac{N}{n}$, comme mentionné en 6.3.2. Par conséquent, on obtient le coût parallèle annoncé pour l'étape BW3. ■

Remarque 6.13. Dans la pratique, le temps requis par l'étape BW1 est seulement $\gamma \mathsf{M}_0 \frac{m+n}{mn} N^2$ si l'on choisit pour les colonnes de x des vecteurs de la base canonique. Ainsi le coût du calcul des produits scalaires devient négligeable. Toutefois, une telle manipulation a pour effet que l'on ne peut plus reposer sur les théorèmes assurant la validité de l'algorithme, puisqu'alors le bloc de vecteurs x n'est plus un vecteur aléatoire.

Le calcul du générateur linéaire F a un coût borné par $O(N^2)$, et nous allons même voir au chapitre 8 une façon d'effectuer ce calcul en temps sous-quadratique. Par conséquent la complexité globale de l'algorithme de Wiedemann par blocs est $O(N^2)$.

Chapitre 7

Schémas d'implantation pour un passage à l'échelle

L'algorithme de Wiedemann par blocs nous a permis de résoudre un système linéaire de très grande taille, à savoir $480\,108 \times 480\,108$, défini sur le corps $\mathbb{F}_{2^{607}-1}$, qui n'est pas un petit corps. Ce calcul a pris trois semaines sur un cluster de six machines de type alpha 4×ev67, cadencées à 667 MHz. En 1997, Kaltofen et Lobo étaient parvenus, avec cet algorithme, à résoudre un système défini sur $\mathbb{F}_2$, de taille $570\,000 \times 570\,000$, en utilisant deux machines de type sparc 20 cadencées à 107 MHz.

De tels résultats nécessitent d'aller au-delà de la présentation académique de l'algorithme. Il est nécessaire de dépenser de l'énergie pour obtenir un bon niveau de parallélisation et pour mener à bout le calcul de grande envergure que cela représente.

Nous décrivons ici l'ossature du programme que nous avons développé pour effectuer des calculs avec l'algorithme de Wiedemann par blocs, sans parler pour l'instant de la phase de calcul de générateurs linéaires qui fait l'objet du chapitre 8. Nous nous concentrons sur le cas des corps premiers, car c'est un système défini sur un corps premier que nous avons résolu. Ce sont donc les spécificités du cadre de travail associé aux corps premiers qui ont guidé nos choix.

7.1 À grande échelle : distribution du calcul

7.1.1 Étape BW1

L'algorithme de Wiedemann par blocs. se prête bien à la distribution sur un réseau de machines. En effet, comme on l'a dit lors de la première évocation des algorithmes par blocs en 5.3.2, le produit matrice × bloc de vecteurs peut être effectué sur n machines distinctes (si n est la taille du bloc de vecteurs y). Cette opération joue un rôle central pour le calcul de la suite de matrices a_k, représentée par la série :

$$A(X) = \sum_{k=0}^{L} a_k X^k \quad \text{où } a_k = x^T B^k y.$$

L'aspect le plus important pour la phase BW1 est qu'entre deux opérations $v \leftarrow Bv$ successives, aucune communication entre les n machines n'est nécessaire. Aussi n machines disposant chacune d'une copie locale de B et de x, ainsi que de la j-ème colonne de y notée y_j peuvent effectuer *concurentiellement* les opérations suivantes :

- $v \leftarrow y_j$,
- Pour k allant de 0 à L :
- $(a_k)_j \leftarrow x^T v$,
- $v \leftarrow Bv$,

– Retourner $A(X)_j$ (i.e. la j-ème colonne de $A(X)$).

Cette capacité à distribuer un calcul d'algèbre linéaire est extrêmement difficile à atteindre, et l'algorithme de Wiedemann par blocs est le seul algorithme à y parvenir aussi bien (l'algorithme de Lanczos par blocs nécessite des communications intermédiaires à l'intérieur de la boucle). La distribution est toutefois incomplète puisque le calcul de générateur linéaire reste de nature complètement séquentielle. La dernière phase consistant à calculer un vecteur du noyau se distribue de manière identique à ce que l'on vient d'évoquer pour la première phase.

Pour énoncer de façon sommaire la tâche que doit accomplir chacune des machines prenant part au calcul, on choisit la formulation suivante :

Entrée : Un vecteur v et un indice k_0.
Sortie : Dans un fichier, tous les $x^T B^{k-k_0} v$, pour $k = k_0, \ldots, L$.

Le principe est de faire en sorte que toute l'information utile soit disponible sous forme de fichiers. Nous verrons que la tolérance aux pannes en est facilitée.

Une fois tous ces fichiers constitués, ils sont rassemblés pour calculer le générateur linéaire. Hormis les transferts de fichiers pour l'initialisation et pour la récupération du résultat du calcul, aucune communication n'a lieu entre les machines. Ceci entraîne deux bénéfices très importants :

– Le calcul sur des machines distinctes, non nécessairement connectées par un réseau à haut débit, est rendu possible.
– Le travail peut se faire de manière désynchronisée. Cela permet de mettre à profit des machines de puissances diverses. En effet, si par exemple deux machines de puissance différente ont ensemble deux colonnes à traiter, le temps de calcul optimal est obtenu en échangeant les tâches à mi-parcours. Cela est possible avec l'algorithme de Wiedemann par blocs. D'un point de vue pratique, la description sommaire faite ci-dessus correspond bien à ce schéma.

Lorsque l'on s'intéresse à la résolution de systèmes linéaires sur $\mathbb{F}_2$, la distribution du calcul n'est pas notre objectif prioritaire puisque le bénéfice est surtout la capacité à effectuer 32 opérations en une. On peut toutefois envisager de choisir $n = 64$ et de distribuer le calcul sur deux machines, comme cela a été fait dans [KL99].

7.1.2 Étape BW2

Nous n'avons pas encore détaillé les algorithmes utilisés pour calculer le générateur linéaire vectoriel de la suite représentée par $A(X)$. Nous rappelons que ce générateur vérifie la relation :

$$A(X)F(X) \in K[X]^{m\times 1}.$$

Les algorithmes pour calculer F seront détaillés au chapitre 8. Nous verrons qu'ils ne sont pas distribuables. Aussi, l'étape BW2 impose de rassembler au préalable les différentes colonnes de $A(X)$ sur une seule machine. Une fois le générateur linéaire vectoriel calculé, on répartit à nouveau les *colonnes* de F sur plusieurs machines, pour entamer la phase BW3.

7.1.3 Étape BW3

Le vecteur qui doit être calculé au cours de la phase BW3 s'écrit sous la forme

$$v = \sum_{i=0}^{\deg F} B^{\deg F - i} z[X^i]F.$$

Au cours de l'étape BW1, on calcule des vecteurs de la forme $B^k y$. Ces vecteurs s'écrivent aussi $B^{k+1}z$. On ne peut les stocker sur le disque dur, car cela représenterait un stockage trop important. Toutefois, il est regrettable de constater que ces mêmes vecteurs sont alors *recalculés* pendant la phase BW3.

Afin de rendre possible une distribution accrue du calcul lors de la phase BW3, il est possible de sauvegarder sur le disque dur *quelques-uns* de ces vecteurs. Ainsi, si l'on sauvegarde les colonnes des blocs $y, B^{L/8}y, \ldots, B^{3L/8}y$, il est possible de distribuer la phase BW3 sur $4n$ machines au lieu de n.

7.2 À petite échelle : parallélisation

Bien que nous n'ayons pas encore abordé la question du calcul de générateur linéaire, on imagine bien que c'est là que se situe le contrecoup du gain représenté par m et n sur la première et la troisième phase de l'algorithme. La valeur optimale des paramètres de bloc m et n sera discutée en 8.7. On verra alors que la valeur optimale de n n'est pas appelée à être très grande. Pour fixer les idées, disons que la valeur optimale de n ne dépasse pas 10.

Pour la résolution de système linéaire qui a motivé notre travail sur l'algorithme de Wiedemann par blocs, les n machines les plus puissantes auxquelles nous avons eu accès étaient toutes des machines multiprocesseurs (architecture de type SMP). Pour réduire autant que possible le temps de calcul de l'algorithme de Wiedemann par blocs, nous avons souhaité paralléliser le calcul sur les différents processeurs de chaque nœud. Il convient de noter que le nombre de processeurs va ici de 2 à 8 et que la parallélisation sur 8 processeurs est quelque chose de très aisé comparé à la parallélisation sur 256 processeurs ou plus faite dans [GM93] par exemple. L'approche que nous proposons pour la parallélisation est simplissime, mais son champ d'application ne va pas au-delà des machines SMP à une dizaine de processeurs au maximum.

7.2.1 Produit matrice × vecteur : répartition sur plusieurs processeurs

Nous nous intéressons au travail à effectuer sur chacune des machines prenant part au calcul, c'est-à-dire au produit matrice × vecteur. Supposons que le nombre de processeurs disponibles est donné par l'entier T. La i-ème coordonnée w_i du produit Bv s'écrit :

$$w_i = \sum_{j, B_{i,j} \neq 0} B_{i,j} v_j.$$

Un produit matrice × vecteur parcourt donc toute la matrice B. Pour répartir ce travail sur T processeurs, on peut choisir de répartir des tranches de la matrice B aux différents processeurs, soit par lignes, soit par colonnes. Ces choix sont distincts :

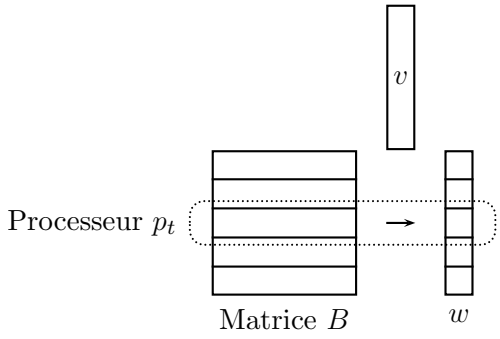


Figure 7.1 – Segmentation de la matrice B pour la parallélisation

- Si chaque processeur dispose d'un paquet de lignes, alors les processeurs lisent concurentiellement les valeurs des v_j, au gré des coefficients non nuls, et chacun écrit sur une partie des coordonnées w_i qui lui est propre : seul le processeur en charge de la ligne i modifie la valeur de w_i.
- Si chaque processeur dispose d'un paquet de colonnes, c'est l'inverse qui se passe : la lecture des coefficients v_j est complètement séparée sur les différents processeurs : seul le processeur en charge de la colonne j accède à la valeur de v_j. En revanche, tous les processeurs contribuent à chacune des coordonnées w_i.

À nos yeux, la seconde de ces possibilités entraîne trop d'inconvénients. Le premier d'entre eux est le fait que la contribution aux w_i, qui est une écriture, est globale. Bien sûr, il est possible que chaque processeur calcule sa contribution au vecteur w et que ces contributions soient ajoutées ensuite, mais cela induit une complication du schéma, notamment en ce qui concerne la synchronisation. Par ailleurs, nous allons voir que l'équilibrage des tâches est plus ardu dans ce contexte.

Nous avons donc choisi le premier schéma. Chaque processeur se voit distribuer un paquet de lignes. Plus exactement, pour t allant de 0 à $T-1$, le processeur p_t est en charge des lignes d'indices $\lfloor t\frac{N}{T} \rfloor$ à $\lfloor (t+1)\frac{N}{T} \rfloor - 1$. On réalise ainsi une segmentation de la matrice qui est représentée par la figure 7.1.

Les opérations effectuées par le processeur p_t sont les suivantes. Notons que la réduction des coefficients modulo le nombre premier p est faite tardivement.

`set_zero` : Pour i allant de $\lfloor t\frac{N}{T} \rfloor$ à $\lfloor (t+1)\frac{N}{T} \rfloor - 1$:

$$w_i \leftarrow 0.$$

`multiply` : Pour i allant de $\lfloor t\frac{N}{T} \rfloor$ à $\lfloor (t+1)\frac{N}{T} \rfloor - 1$, et pour chaque $B_{i,j}$ non nul :

$$w_i \leftarrow w_i + B_{i,j} v_j.$$

`reduce_modp` : Pour i allant de $\lfloor t\frac{N}{T} \rfloor$ à $\lfloor (t+1)\frac{N}{T} \rfloor - 1$:

$$w_i \leftarrow w_i \bmod p.$$

7.2.2 Synchronisation des processeurs

L'implantation de ce calcul multiprocesseurs a été effectuée en langage `C`, en utilisant les *threads* POSIX [But97]. Cette approche de « bas niveau » a été préférée à l'emploi d'une

Algorithme **Multithread-ApplyBlackBox**
Entrée : Une matrice B, un vecteur de départ v et un indice k_0, tels que $v = B^{k_0}y$.
Sortie : Dans un fichier, les $x^T B^k y$, pour $k = k_0, \ldots, L$

```
for(k=k0;k<L;)
{
    /* À ce point on a: v = B^k y, pour la tranche locale de v */
    src_vec=v; dst_vec=w;
    dst_area=pointeur vers la bonne tranche du vecteur w;

    set_zero(dst_area);
    multiply(B,dst_area,src_vec);
    reduce_modp(dst_area);                    // réduit modulo p les entrées du vecteur
    barrier_wait();

    k++;
    save_dot_products(i,dst_vec);
    if (k==L) break;

    /* À ce point on a: w = B^k y, pour la tranche locale de w */
    src_vec=w; dst_vec=v;
    dst_area=pointeur vers la bonne tranche du vecteur v;

    set_zero(dst_area);
    multiply(B,dst_area,src_vec);
    reduce_modp(dst_area);                    // réduit modulo p les entrées du vecteur
    barrier_wait();

    k++;
    save_dot_products(i,dst_vec);
}
```

Programme 7.2: Implantation *multithread* du produit matrice × vecteur

bibliothèque comme MPI [MPI], dans le but de pouvoir traiter avec le maximum de finesse le cœur du problème, à savoir la gestion des points de synchronisation.

Le modèle d'exécution est un modèle SIMD, où chaque processus, ou *thread*, est une instance distincte du programme, possédant une pile propre, mais partageant les variables globales. Le calcul impose des points de *synchronisation* entre les processeurs. Pour obtenir le meilleur niveau de performances possibles, ce nombre de points de synchronisation doit être minimal. Le programme 7.2 reproduit, de manière un peu simplifiée, l'implantation qui a été réalisée. La figure 7.3 représente de manière schématique l'organisation du calcul pour l'exemple de deux processeurs. Quelques précisions s'imposent.

- Les appels à la fonction `barrier_wait` sont les points de synchronisation des *threads*. Un exemple de programmation d'une telle fonction est donné dans [But97], en partant des primitives de base de la libraire de *threads* POSIX, les *mutexes* (pour *mutually exclusive*).
- Comme on l'a mentionné, la pile et donc toutes les variables locales du programme, sont spécifiques à chaque *thread*. En particulier, il est absolument capital que l'indice de boucle `i` soit une variable locale.
- La fonction `save_dot_products` calcule les produits scalaires avec les colonnes du bloc de vecteurs x. À cet effet, on peut supposer que l'ensemble du vecteur calculé est nécessaire. Toutefois, pour rendre le calcul plus efficace, on a choisi de prendre pour x un bloc de vecteurs appartenant à la base canonique. Ainsi, le calcul des produits scalaires n'est que la lecture d'une coordonnée. Si l'on est dans le cas où le nombre m de colonnes de x est un multiple du nombre k de processeurs utilisés, on peut même faire en sorte que les produits scalaires puissent être calculés avec uniquement la tranche propre à chaque *thread* du programme.
- Ce n'est pas dans un but cosmétique que la boucle du programme 7.2 a été déroulée pour contenir deux itérations. Cela permet d'éviter les échanges de vecteurs et économise un point de synchronisation. C'est donc un point fondamental pour obtenir de bonnes performances, puisque l'on utilise *un seul* point de synchronisation, soit deux fois moins que dans l'implantation qui avait été réalisée par Kaltofen et Lobo [KL99].

7.2.3 Mise au point de la synchronisation

Cet aspect « parallèle » du calcul a nécessité un travail important, inhérent à la mise au point d'algorithmes fonctionnant en parallèle sur plusieurs processeurs. Comme on l'a souligné, il n'était pas question d'avoir de trop nombreux points de synchronisation dans la boucle du programme. On estime avoir rempli correctement cet objectif, puisque le nombre de points de synchronisation que l'on utilise est minimal (un seul). On pourrait donner quelques exemples d'implantations de **Multithread-ApplyBlackBox** qui ne marchent pas, pour comparer avec le programme 7.2, ou bien donner tous les détails de l'implantation réelle, qui reprend l'itération courante si elle échoue, mais une telle exposition présenterait certainement quelques longueurs, et serait sans doute peu éclairante. Nous nous contentons d'insister sur quelques exemples de difficultés.

En examinant le programme 7.2, on peut ainis remarquer que le positionnement de la réduction modulaire (la fonction `reduce_modp`) des coefficients ne s'effectue pas du tout au hasard. L'appel à cette fonction peut avoir lieu à aucun autre moment de l'itération. En effet, si celui-ci est effectué simultanément avec le calcul des produits scalaires (`save_dot_products`),

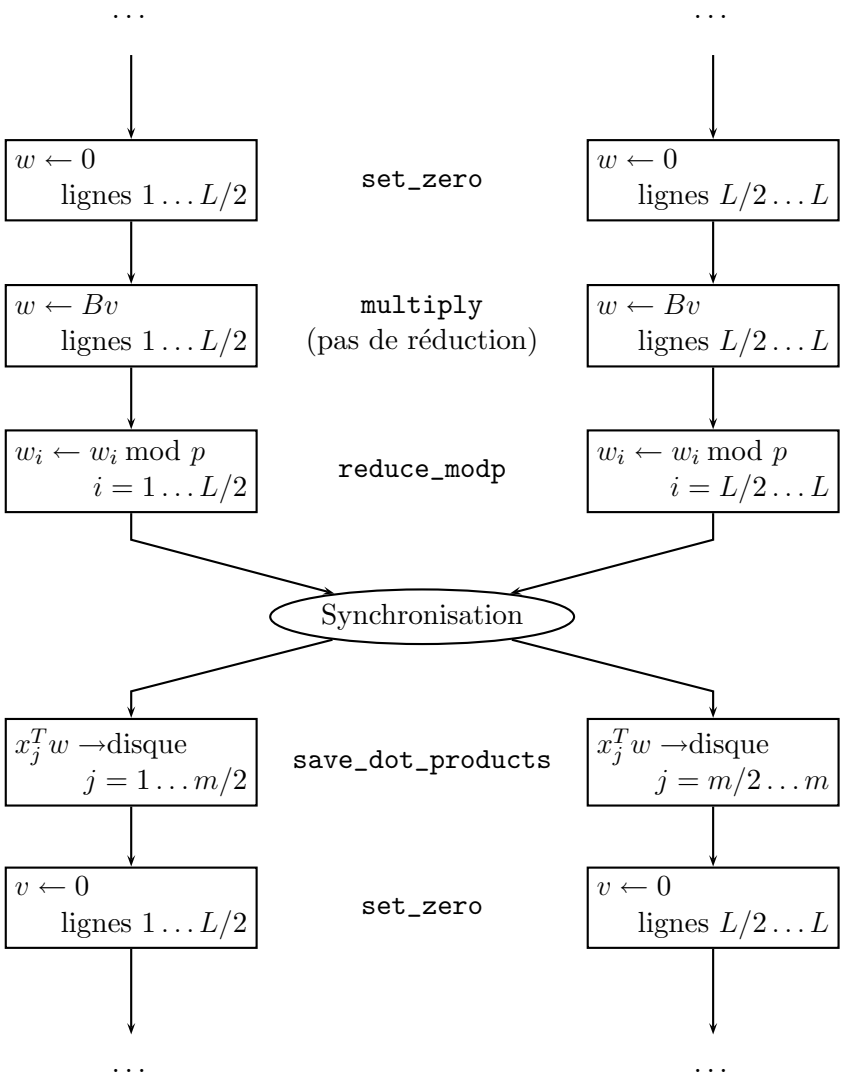


Figure 7.3 – Organisation du calcul pour Multithread-ApplyBlackBox

on court au désastre (ce n'est pourtant pas évident) : on est face à une *race condition*[1]. En effet, si un *thread* est plus rapide que les autres et passe à l'étape suivante, `multiply`, pendant qu'un autre est en train de mettre à jour le vecteur w (destination de l'itération courante, et donc source de l'itération suivante), le comportement du programme est différent de ce à quoi on s'attend.

Ainsi, toutes les opérations dans le programme 7.2 doivent nécessairement être soigneusement ordonnées, car sinon on court le risque d'avoir un comportement non déterministe du programme. Et ce risque est bien réel !

7.2.4 Équilibrage

Le programme 7.2 contient des points de synchronisation. Les *threads* ont donc tendance à attendre le *thread* le plus lent. Il faut donc éviter toute raison structurelle pouvant faire qu'un *thread* soit plus lent que les autres. Or le temps d'exécution d'une itération par un *thread* est proportionnel au nombre de coefficients non nuls dans la tranche concernée de la matrice B. Nous devons donc équilibrer ces tranches.

Si l'on observe la figure 5.1 page 89, on voit que les matrices que nous avons eu à traiter n'ont pas leurs coefficients répartis de façon uniforme. En particulier, ces coefficients ne sont pas du tout répartis uniformément sur les colonnes, ce qui aurait rendu un bon équilibrage particulièrement difficile si l'on avait choisi de découper selon les colonnes. L'observation de la figure 5.1 montre toutefois que les lignes de la matrice ne sont pas non plus particulièrement équilibrées. Nous avons donc écrit un petit programme qui permute les lignes de telle sorte que le résultat soit équilibré, que l'on souhaite paralléliser le travail sur 2, 4, 6, 8 ou 16 processeurs. Le principe en est très simple. Si κ est le ppcm des nombres de processeurs envisagés pour la répartition, on maintient κ listes de lignes. Ensuite, on insère une à une les lignes de la matrice, en prenant toujours la ligne de poids maximal dans la matrice, pour l'ajouter dans celle des κ listes dont le poids global est le plus faible. On concatène enfin nos κ listes pour constituer la permutation des lignes à appliquer.

Une méthode plus simple d'équilibrage, que l'on peut appeler « équilibrage du pauvre », est l'équilibrage par saturation. Sur une machine avec 4 processeurs, si on lance 16 *threads*, on peut espérer ne pas avoir trop d'écart entre les temps de calculs des différents processeurs. Cette façon d'équilibrer les calculs est mauvaise car elle impose une pression nettement plus importante sur le système[2] et ses résultats sont très médiocres par rapport à la méthode précédemment citée.

7.2.5 Portabilité, performance

Les *threads* POSIX constituent une interface raisonnablement stable pour une interaction avec le système d'exploitation qui est inévitablement complexe. Le programme réalisé fonctionne sans modification sur les systèmes Linux et Tru64 Unix. La documentation précise des différents comportements à attendre s'est avérée très utile. Toutefois, arriver au programme 7.2, qui marche correctement, en utilisant seulement un point de synchronisation par itération, n'a pas été facile ([KL99] utilise deux points de synchronisation par itération, par exemple).

[1] La traduction française n'est pas évocatrice...

[2] En surchargeant ainsi une machine de type PC à 8 processeurs, on a obtenu de manière déterministe un plantage du système.

Dans tous les tests que nous avons menés, sur des machines très diverses, avec un nombre de processeurs allant jusqu'à 8, le facteur d'accélération obtenu par l'utilisation de *threads* s'est avéré être presque exactement le nombre de processeurs utilisés. Tout au plus a-t-on déploré une perte de l'ordre de 5%. Par ce bon comportement, on doit comprendre que notre chance a été le fait de ne devoir paralléliser que sur un petit nombre de processeurs. Tous les essais de parallélisation à plus grande échelle, sur des machines massivement parallèles, montrent que les choses sont beaucoup plus difficiles lorsque l'on cherche à utiliser concurentiellement un nombre de processeurs de l'ordre de 256 ou plus.

7.3 Tolérance aux pannes, récupération

Notre mode de gestion des tâches à effectuer sur les différentes machines (qui sont parfois assaillies par d'autres utilisateurs) n'est pas sophistiqué. Par exemple, aucune considération de migration automatique de travail n'a été envisagée. Le système de gestion des tâches employé pourrait s'appeler « `kill -9` ». La raison en est simple : comme nous le verrons au chapitre 9, la disparition des processus est une chose à laquelle il faut s'attendre lors de calculs très gourmands en ressources système. Aussi, il est nécessaire d'incorporer dans le programme un mécanisme garantissant la sauvegarde des états intermédiaires du calcul.

L'approche adoptée est simple. Toutes les 100 itérations, par exemple, on sauve le vecteur v sur le disque dur, en remplaçant éventuellement l'ancienne sauvegarde pour économiser de la place (on peut aussi choisir de garder certains des vecteurs v pour pouvoir distribuer plus amplement la troisième phase). Ainsi, chaque nouveau processus lancé teste l'existence d'un fichier de sauvegarde du vecteur v et a donc la possibilité de reprendre les calculs à partir de ce point.

Un exercice plus délicat est la gestion des erreurs de calcul. Si jamais un bit du calcul de $B^{3457}y$ est faux, il est clair que peu d'itérations plus tard, tous les bits sont faux, et le calcul perdu. Il est nécessaire d'éviter cela et surtout de pouvoir le détecter.

Pour cela, nous avons déterminé au début du calcul deux vecteurs α et β, reliés par $\beta = B^T\alpha$, et ayant chacun des coefficients *petits* et *non-nuls*. Cela nous a été possible grâce à la petite taille des coefficients contenus dans les matrices traitées. À chaque itération, pour vérifier $w = Bv$, on a vérifié $\alpha^T w = \beta^T v$. Nous avons pu ainsi détecter les erreurs avec une bonne probabilité.

L'introduction de cette vérification dans la boucle du programme 7.2 est délicate. Il nous faut, avant le `i++` et après le point de synchronisation, revenir si besoin à l'étape `src_vec=`v `; dst_vec=`w `;`. On souhaite éviter l'introduction d'un point de synchronisation supplémentaire. Pour cela, les modifications suivantes doivent être apportées. On n'inclut pas ici le code de la fonction `barrier_wait()`, disponible avec l'ouvrage [But97].

- Avant le point de synchronisation, chaque *thread* calcule sa propre contribution aux produits $\beta^T v$ et $\alpha^T w$. Ce calcul peut être effectué concurentiellement par les différents *threads*.
- À l'entrée dans la fonction `barrier_wait()` (qui doit donc être modifiée en conséquence), chaque *thread* profite de l'instant où il dispose d'un *lock* exclusif (c'est-à-dire qu'il est « tranquille ») pour ajouter ces contributions à un compteur.
- Le dernier[3] des *threads* à rejoindre ce point de synchronisation vérifie l'égalité des deux

[3] À l'entrée du point de synchronisation, pour qu'un *thread* teste de façon sûre qu'il est le dernier à entrer, il faut réfléchir !

compteurs en question pendant qu'il dispose du *lock*. Une fois cette égalité testée, il met à jour un indicateur binaire selon qu'elle est vraie ou fausse.

– Tous les *threads*, en sortie du point de synchronisation, testent cet indicateur binaire, et recalculent le cas échéant l'itération.

Chapitre 8

Calcul sous-quadratique de générateurs linéaires pour des séquences de matrices

8.1 Présentation du problème

Soit une matrice $m \times n$ à coefficients dans les séries formelles sur un corps fini K. On note $A(X) \in K[[X]]^{m \times n}$. Comme on l'a vu en 6.3.2, c'est un objet de ce type qui est produit dans la première phase de l'algorithme de Wiedemann par blocs. Cet objet est indifféremment vu comme une matrice de séries formelles, comme une série formelle de matrices, ou enfin comme une suite de matrices. Nous avons déjà mis en évidence que la grandeur associée qui nous intéresse est un générateur linéaire de A sur $K[X]^n$, tel que défini par l'énoncé 6.1, et précisé par l'exemple en page 104 (au sens de la terminologie mise en place lors de cette discussion, on s'intéresse plus exactement à un générateur linéaire vectoriel). L'équation que doit satisfaire un tel générateur $F(X) \in K[X]^n$ est tout simplement :

$$AF \in K[X]^m.$$

Cette équation, lorsque les entiers m et n sont égaux à 1, peut être résolue simplement avec efficacité. Si le générateur linéaire recherché F est de degré $\leqslant N$, on va montrer que l'algorithme de Berlekamp-Massey [Ber68, Mas69] permet de calculer F à partir d'au plus $2N$ coefficients de A (vu comme une suite – on a donc besoin de $2N$ matrices $m \times n$, soit $2N$ scalaires puisque l'on est dans le cas $m = n = 1$). Nous verrons aussi que l'algorithme d'Euclide étendu résout ce problème. En fait, les deux algorithmes peuvent, dans le cas scalaire, être vus sous un angle unificateur (il suffit de prendre la réciproque des polynômes concernés [Dor87]).

La complexité de l'algorithme de Berlekamp-Massey est quadratique en N, tout comme celle de l'algorithme d'Euclide si on l'applique « naïvement ». Il existe des améliorations sous-quadratiques de l'algorithme d'Euclide ([GY79] notamment, qui reprend la présentation de l'algorithme HGCD de [AHU74]). On sait donc traiter le cas scalaire du calcul de générateur linéaire de manière tout à fait satisfaisante et efficace. Étant donnée l'ubiquité de ce problème, c'est un constat rassurant.

Pour traiter le cas matriciel, plusieurs algorithmes existent, dont certains sont de complexité sous-quadratique. Il est possible de formuler de manière assez générale le problème que nous souhaitons résoudre. Il s'agit du calcul de générateurs linéaires matriciels de taille $n \times r$ (avec bien sûr $r < n$) pour une suite de matrices $m \times n$. Nous ne prétendons traiter efficacement que le cas $r = 1$.

Un panel d'algorithmes traitant ce problème sont recensés dans [Kal95, Vil97]. Le premier travail mentionné à ce sujet est [Ris72], qui résout le problème de la réalisation minimale partielle, important en théorie du contrôle. Un autre algorithme d'origine « numérique » est dû à

[BA80, Mor80]. Coppersmith [Cop94] a proposé une généralisation matricielle de l'algorithme de Berlekamp-Massey, permettant de traiter le cas particulier de $r = 1$, qui est le cas que l'on souhaite résoudre ici. L'algorithme qu'il obtient a une complexité $O((m+n)N^2)$, et utilise $L = \frac{N}{m} + \frac{N}{n} + O(1)$ coefficients de A. Nous décrivons cet algorithme en 8.4.

Beckermann et Labahn ont présenté dans [BL94] un algorithme très générique « *power Hermite-Padé solver* » traitant le cas de r quelconque, sans aucune supposition de régularité de la matrice d'entrée $A(X)$, et de complexité sous-quadratique $O((m+n)^2 mk \log^2 k)$ pour calculer un générateur de degré k.

Pour des calculs à grande échelle [Lob95, KL99, Pen98], seuls les algorithmes de Coppersmith et de Beckermann et Labahn semblent avoir été considérés. L'algorithme de [BA80, Mor80], nécessitant une randomisation de l'entrée, n'a apparemment pas été employé. Il est fort regrettable de constater que l'algorithme de Beckermann et Labahn n'a été implanté que dans sa version quadratique par [Pen98]. Dans tous les cas, la performance de l'algorithme de Coppersmith a été jugée satisfaisante voire supérieure aux alternatives. Une des raisons pour cela peut être l'absence de coefficient multiplicatif important caché par le $O()$: nous verrons en 8.4.6 que le nombre exact de multiplications requises est $\frac{1}{2}(m+n)(nk)^2$ pour le calcul d'un générateur linéaire de degré k.

Pour les problèmes que nous avons eu à résoudre, à savoir de très gros systèmes linéaires sur $\mathbb{F}_p$, l'algorithme de Coppersmith [Cop94] ne nous a pas paru satisfaisant. Nous avons, pour pallier cet inconvénient, fourni une version sous-quadratique de l'algorithme de Coppersmith, en s'inspirant à la fois de l'amélioration sous-quadratique de l'algorithme d'Euclide et de la dualité entre l'algorithme d'Euclide et l'algorithme de Berlekamp-Massey. La difficulté majeure du passage au cadre matriciel est la gestion de la non-maximalité ponctuelle du rang de certaines quantités. Écrire un algorithme d'Euclide dans ce contexte est un exercice bien délicat, et c'est pour cette raison que nous avons préféré faire ce « détour » par l'algorithme de Berlekamp-Massey, qui a rendu la présentation plus claire à notre sens. Ce travail, détaillé ici, a fait l'objet des publications [Tho01a] et [Tho02b].

L'algorithme que nous proposons conserve les avantages de l'algorithme de Coppersmith : pas de randomisation et peu de constantes cachées, puisque nous avons pu mener le compte du nombre de multiplications requises. Si on s'aventure au jeu de la comparaison des complexités, l'algorithme que nous proposons a une complexité de $O(\frac{(m+n)^3}{m} k \log^2 k + (m+n)^3 k \log k)$, ce qui est meilleur que la complexité de l'algorithme de Beckermann et Labahn, $O((m+n)^2 mk \log^2 k)$. Pour ces raisons, on peut supposer que l'algorithme que nous proposons est plus rapide. Toutefois, l'élément primordial de cette différenciation des complexités tient à la modification profonde introduite par l'utilisation de la transformée de Fourier dans notre algorithme : pour multiplier des matrices de polynômes, on parvient à *découpler* la partie coûteuse de la multiplication d'une part, et la complexité du produit de matrices d'autre part, puisqu'on fait $(m+n)^2$ calculs de transformées de Fourier discrètes, et $(m+n)^3$ convolutions. Il est envisageable d'obtenir un apport du même type par un examen de l'algorithme de Beckermann et Labahn. Par ailleurs, il est très vraisemblable que ces deux algorithmes puissent être compris dans une théorie unifiée[1], au prix de quelques suppositions sur le caractère générique de l'entrée que nous énonçons en 8.3.

L'algorithme obtenu a été implanté, et son efficacité en pratique a été démontrée par le calcul d'un générateur linéaire de taille 4×4 et de degré 121 152, le corps de base étant le

[1] Dans le cadre du projet LINBOX, une implantation de l'algorithme de Beckermann et Labahn est en cours, donc une comparaison est possible à moyen terme (G. Villard, communication privée, Janvier 2003).

corps premier $\mathbb{F}_{2^{607}-1}$. Dans une seconde expérience, on a aussi calculé un générateur linéaire 8×8 de degré 60 014 sur le même corps.

8.2 Algorithmes classiques

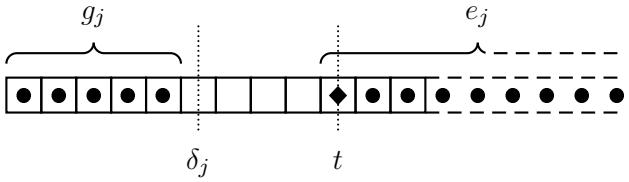
8.2.1 L'algorithme classique de Berlekamp-Massey dans le cas scalaire

Pour pouvoir plus aisément comprendre le fonctionnement des algorithmes matriciels exposés dans ce chapitre, nous allons commencer par reprendre l'exposition de l'algorithme de Berlekamp-Massey en adoptant délibérément un point de vue qui se généralise sans peine au cas matriciel que nous traiterons par la suite.

L'algorithme de Berlekamp-Massey calcule le générateur linéaire $F(X)$ de $A(X)$ par *approximations successives*. Deux « candidats générateurs » $f_1(X)$ et $f_2(X)$ sont maintenus à chaque étape du calcul. À chacun de ces candidats générateurs f_j est associé un second membre g_j, et un terme d'erreur e_j, de telle sorte que l'équation suivante soit vérifiée à l'itération t de l'algorithme :

$$Af_j = g_j + X^t e_j.$$

Pour chacun des candidats générateurs, on maintient aussi la quantité δ_j qui est un majorant de $\max(\deg f_j, 1 + \deg g_j)$ (cette dernière quantité a aussi été notée $\delta(f_j, g_j)$ en 6.4). Pour chaque i, les coefficients non nuls du produit Af_j s'organisent comme sur le schéma suivant :



Initialisation

Pour l'initialisation de l'algorithme de Berlekamp-Massey, on fait intervenir la valuation de la série A, que l'on note $s - 1$. Les valeurs initiales sont alors :

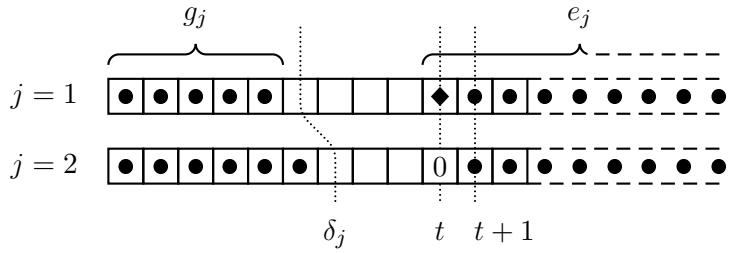
$$\begin{aligned} t = s, \qquad f_1 &= 1, \qquad \delta_1 = s, \\ f_2 &= X, \qquad \delta_2 = 1. \end{aligned}$$

Pour passer de l'étape t à l'étape $t + 1$ dans l'algorithme, il faut parvenir à éliminer les coefficients de degré 0 dans les e_j. Étant donné le choix que nous avons fait en fonction de la valuation de A, on peut constater que le coefficient $e_2(0)$ est non nul, donc pour l'étape de départ, les $e_j(0)$ ne sont pas tous deux nuls. Nous allons voir que ces coefficients vont conserver cette propriété.

Évolution

On souhaite passer de t à $t + 1$ en n'augmentant $\sum \delta_j$ que de 1. Sans perte de généralité, on peut supposer que $\delta_1 \leqslant \delta_2$. Deux cas distincts peuvent se présenter. Si $e_1(0) \neq 0$, alors on remplace f_2 par la quantité $f_2 - \lambda f_1$, où $\lambda = \frac{e_2(0)}{e_1(0)}$. Ce nouveau polynôme f_2 est tel que $e_2(0)$ (i.e. le coefficient de degré t de Af_2) est désormais nul. On ne fait rien dans l'autre cas. Sans modifier les δ_j, on a donc annulé l'un des coefficients, et l'on sait par hypothèse que l'autre

coefficient est non nul. Les coefficients des produits Af_j s'organisent désormais sous la forme (on a choisi d'illustrer l'un des cas, sans perte de généralité) :



Supposons que l'on est dans le cas représenté ci-dessus, c'est-à-dire $e_2(0) = 0$ et $e_1(0) \neq 0$, on remplace maintenant f_1 par Xf_1, et δ_1 par $\delta_1 + 1$. Les nouveaux polynômes f_1 et f_2 sont des approximations convenables pour l'itération $t+1$. De plus, la nouvelle valeur de $e_1(0)$ est la même que l'ancienne (une fois que l'on a incrémenté t), donc il reste vrai que les $e_j(0)$ ne sont pas tous deux nuls : on reste bien dans le cas générique.

Terminaison

Au fur et à mesure des étapes de ce procédé, si l'on note $\overline{\delta}$ la moyenne des δ_j, la différence $t - \overline{\delta}$ croît (elle croît d'$\frac{1}{2}$ à chaque étape). Supposons que l'on connaît l'*existence* d'un générateur linéaire de degré $\leqslant N$ (en fait ce n'est pas exactement le degré qui nous intéresse). Alors, lorsque $t - \overline{\delta} > N$, il existe un j tel que $t - \delta_j > N$, et la proposition suivante nous montre que le polynôme f_j correspondant est un générateur linéaire.

Théorème 8.1. *Soit $A \in K[[X]]$, admettant un générateur linéaire Φ (inconnu) tel que :*

$$\delta(\Phi, \Phi A) \leqslant N.$$

Soit $F \in K[X]$, $G \in K[X]$, $E \in K[[X]]$, et t un entier tels que :

$$AF = G + X^t E.$$

Si $t - \delta(F, G) \geqslant N$, alors $E = 0$.

Démonstration. Il suffit de multiplier l'expression AF par Φ. On obtient :

$$\Phi AF = (\Phi A)F = \Phi G + X^t \Phi E.$$

Étant donnée l'information disponible sur les degrés, on a :

$$\begin{array}{rrl} \deg \Phi A < N, & \deg F \leqslant \delta(F,G) \leqslant t - N & \Rightarrow \deg (\Phi A)F < t, \\ \deg \Phi \leqslant N, & \deg G < t - N & \Rightarrow \deg \Phi G < t. \end{array}$$

Il s'ensuit que $\Phi E = 0$, donc $E = 0$. ∎

Par conséquent, le pseudo-code 8.1 fournit un exemple d'implantation de l'algorithme de Berlekamp-Massey, par exemple en langage Magma. Notons que dans cette implantation, on ne considère que les coefficients constants des séries e_1 et e_2, puisque ce sont les grandeurs intéressantes.

Algorithme **BerlekampMasseyScalar**
Entrée : Une série $A \in K[[X]]$, et N une borne sur $\delta(\Phi, \Phi A)$ pour un générateur.
Sortie : Un générateur linéaire F de A, et une borne sur $\delta(F, AF)$

```
f_1:=1; f_2:=X; t:=Valuation(A)+1; δ_1=t; δ_2:=1;
ε_1:=Coefficient en X^t de Af_1;   // correspond à e_1(0)
ε_2:=Coefficient en X^t de Af_2;   // correspond à e_2(0)

while t − (δ_1+δ_2)/2 < N do
    if δ_1 ⩽ δ_2 then
        if ε_1 ≠ 0 then
            f_1:=X * f_1;
            f_2:=f_2 − (ε_2/ε_1) f_1;
            ε_2:=Coefficient en X^{t+1} de Af_2;
            δ_1:=δ_1+1;          // ε_1 ne change pas
        else                     // cas exceptionnel, mais permanent si f_1 est générateur
            f_2:=X * f_2
            ε_1:=Coefficient en X^{t+1} de Af_1;
            δ_2:=δ_2+1;          // f_1, ε_2 ne changent pas
        end if;
    else
        ... Opérations inverses...
    end if;
    t:=t + 1;
end while;

if t − δ_1 ⩾ N then return f_1,δ_1; else return f_2,δ_2; end if;
```

Programme 8.1: Algorithme de Berlekamp-Massey

Correction du résultat

Il reste à prouver que l'algorithme de Berlekamp-Massey fournit une solution non triviale : en effet, si le générateur linéaire F rendu est nul, il est de peu d'utilité. Nous donnons une preuve qui est en fait une spécialisation de celle que l'on utilisera plus tard dans le cas matriciel. Considérons la matrice $h(X) \in K[X]^{2\times 2}$ suivante :

$$h = \begin{pmatrix} f_1 & f_2 \\ e_1 & e_2 \end{pmatrix}.$$

À l'initialisation de l'algorithme, on a par construction $f_1 = 1$, $f_2 = X$, et $e_2(0) \neq 0$. Par conséquent, $(\det h)(0) = e_2(0) \neq 0$. Par ailleurs, le passage de l'étape t à l'étape $t+1$ se transcrit, pour la matrice h, par la formule suivante (à permutation près des lignes et des colonnes pour τ et D) :

$$h^{(t+1)} = \begin{pmatrix} 1 & 0 \\ 0 & \frac{1}{X} \end{pmatrix} h^{(t)}\, \tau\, D, \quad \text{où } \tau = \begin{pmatrix} 1 & -\lambda \\ 0 & 1 \end{pmatrix} \text{ et } D = \begin{pmatrix} X & 0 \\ 0 & 1 \end{pmatrix}.$$

Donc la valeur de $\det h$ est *inchangée* au cours de l'algorithme. Ceci démontre que l'algorithme produit nécessairement un résultat non trivial, car si $f_1 = 0$ à l'étape t, cela impliquerait qu'il existe une colonne nulle dans $h^{(t)}$, ce qui est exclu.

8.2.2 L'algorithme d'Euclide (étendu)

Lien avec le problème, et notations

Le calcul d'un générateur linéaire peut aussi se faire en employant l'algorithme d'Euclide étendu aux polynômes $U = X^{2N}$ et $V = A \bmod X^{2N}$. Cet algorithme bien connu procède en écrivant une suite d'équations :

$$U(X)P_i(X) + V(X)Q_i(X) = R_i(X),$$

où l'on part de $(P_0, Q_0, R_0) = (1, 0, U)$ et $(P_1, Q_1, R_1) = (0, 1, V)$. Le passage à l'étape $i+1$ se fait par division euclidienne de R_{i-1} par R_i (on note Γ_i le quotient). On obtient ainsi :

$$\begin{aligned} R_{i+1} &= R_{i-1} - \Gamma_i R_i, \\ P_{i+1} &= P_{i-1} - \Gamma_i P_i, \\ Q_{i+1} &= Q_{i-1} - \Gamma_i Q_i. \end{aligned}$$

De telle sorte que : $UP_{i+1} + VQ_{i+1} = R_{i+1}$.

En examinant les degrés des polynômes concernés, on arrive facilement aux identités :

$$\begin{aligned} \deg \Gamma_i &= \deg R_{i-1} - \deg R_i, \\ \deg P_{i+1} &= \deg P_i + \deg \Gamma_i, \qquad \text{(sauf pour } i = 1) \\ \deg Q_{i+1} &= \deg Q_i + \deg \Gamma_i. \end{aligned}$$

On en déduit les deux invariants suivants :

$$\forall i \geqslant 1 \quad \deg P_{i+1} + \deg R_i = \deg V,$$

$$\forall i \geqslant 0 \quad \deg Q_{i+1} + \deg R_i = \deg U,$$

Comme $\deg Q_i$ est une grandeur strictement croissante, il existe un entier i tel que

$$\deg Q_i \leqslant \frac{\deg U}{2} < \deg Q_{i+1}.$$

Pour cette valeur, on déduit en utilisant l'invariant que $\deg R_i < \frac{\deg U}{2}$. Si l'on spécialise ce résultat aux valeurs qui nous intéressent, arrêter ainsi le processus de l'algorithme d'Euclide au milieu de son chemin permet de mettre en évidence deux polynômes Q et R tels que :

$$\begin{aligned} AQ &= R \bmod X^{2N}, \\ \deg Q &\leqslant N, \\ \deg R &< N. \end{aligned}$$

Cette propriété, si l'on sait qu'il existe un générateur Φ (inconnu) tel que $\delta(\Phi, \Phi A) \leqslant N$, nous permet de conclure en vertu du théorème 8.1 que $AQ = R$ (sans modulo), et donc que le polynôme Q est un générateur linéaire pour A, vérifiant en outre $\delta(Q, AQ) \leqslant N$. On a donc démontré :

Proposition 8.2. *Soit $A \in K[[X]]$, possédant un générateur Φ (inconnu) tel que $\delta(\Phi, \Phi A) \leqslant N$. L'algorithme d'Euclide étendu appliqué à X^{2N} et A produit, si on l'arrête au milieu, un générateur linéaire Q de A, vérifiant $\delta(Q, AQ) \leqslant N$.*

Une accélération sous-quadratique de l'algorithme d'Euclide

Arrêter l'algorithme d'Euclide « au milieu » comme nous le faisons ici est en fait le cas le mieux adapté à une amélioration de la complexité. L'algorithme HGCD proposé dans [AHU74] et reformulé dans [GY79] permet de réduire cette complexité de $O(N^2)$ à $O(N \log^2 N)$.

Pour la mise en place d'un algorithme sous-quadratique, on a besoin d'introduire encore quelques notations qui n'allègent bien sûr pas l'exposition, mais sont nécessaires. En effet, les degrés des polynômes R_i au cours des étapes de l'algorithme sont variables. Le degré de R_i est noté r_i. Il peut descendre « plus vite que prévu ». Pour mesurer cette descente, on introduit la notation suivante :

Notation 8.3. *Soient $U(X)$ et $V(X)$ deux polynômes, avec $\deg V < \deg U$. Soit $s \in \mathbb{R}^+$. Avec les notations précédemment exposées, on note $\ell(s)$ l'unique indice tel que :*

$$r_{\ell(s)} \geqslant s > r_{\ell(s)+1}.$$

On a ainsi par construction $\ell(\deg U) = 0$. En dehors de cette donnée, tout ce que l'on peut dire de $\ell(s)$ est que c'est une fonction décroissante, et aussi que $\ell(\lceil s \rceil) = \ell(s)$ (cette relation se vérifie facilement). Pour tout le raisonnement qui suit, on va fixer l'hypothèse $\deg V < \deg U$.

Comme on l'a vu précédemment, c'est la quantité Γ_i qui permet de calculer P_{i+1}, Q_{i+1}, et R_{i+1}. On peut écrire matriciellement :

$$\begin{pmatrix} P_i & P_{i+1} \\ Q_i & Q_{i+1} \\ R_i & R_{i+1} \end{pmatrix} = \begin{pmatrix} P_{i-1} & P_i \\ Q_{i-1} & Q_i \\ R_{i-1} & R_i \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & -\Gamma_i \end{pmatrix},$$

$$M_i(X) = M_{i-1}(X)T_i(X),$$

où l'on a pris les notations convenables pour $M_i(X)$ et $T_i(X)$. Il est important de remarquer que la matrice M_0 n'est autre que :

$$M_0 = \left(\begin{array}{c} I_2 \\ \hline U \quad V \end{array}\right).$$

Nous allons baptiser $\Pi_{i,j}(X)$ la matrice $T_{i+1}(X)\dots T_j(X)$ (en prenant la convention $\Pi_{i,i} = I_2$). Cette matrice vérifie la propriété $M_j = M_i\Pi_{i,j}$. En outre, en vertu de la forme connue de M_0, on sait que $\Pi_{0,i}$ n'est autre que :

$$\Pi_{0,i} = \begin{pmatrix} P_i & P_{i+1} \\ Q_i & Q_{i+1} \end{pmatrix}.$$

Utilisation d'une information incomplète

Le point-clef de l'algorithme est la possibilité d'utiliser une information incomplète sur R_{i-1} et R_i pour calculer Γ_i. En effet, seuls les $r_{i-1} - r_i + 1$ coefficients de tête de ces deux polynômes sont nécessaires. En fonction du nombre de coefficients connus de U et V, on peut déterminer combien de matrices $T_i(X)$ peuvent être calculées.

Proposition 8.4. *Si U et V sont connus à partir du coefficient de degré m (inclus) avec $m > 0$, alors $\Pi_{0,i}$ est uniquement déterminé si et seulement si $i \leqslant \ell\left(\frac{n+m}{2}\right)$, où l'on a noté $n = \deg U$.*

Ce résultat est un résultat pauvre dans le cas où $m = 0$ (la partie « seulement si » est bien sûr incorrecte). Nous démontrons qu'il est optimal pour $m > 0$.

Démonstration. Pour commencer, nous déterminons combien de coefficients sont connus dans l'expression de R_i. Comme on sait que $\deg \Gamma_i = r_{i-1} - r_i$, l'expression connue de R_2 en fonction de $R_1 = V$ et $R_0 = U$ nous donne que R_2 est connu à partir du coefficient de degré $m + r_0 - r_1 = m + n - r_1$. Une récurrence triviale permet d'obtenir, pour $i \geqslant 1$:

$$R_i \text{ est connu à partir du coefficient de degré } m + n - r_{i-1} \text{ inclus.}$$

Par ailleurs, nous souhaitons calculer des matrices Π, c'est-à-dire avoir la possibilité de déterminer les quotients Γ_i. On a vu que seuls les $r_{i-1} - r_i + 1$ coefficients de tête importaient. Le coefficient de plus bas degré dans R_i intervenant dans la détermination de Γ_i est donc le coefficient de degré $r_i - (r_{i-1} - r_i) = 2r_i - r_{i-1}$. Il convient, avec les contraintes de précision qui ont été énoncées, que ce coefficient soit connu pour que l'on puisse déterminer Γ_i. On en déduit donc :

$$\begin{gathered} \Gamma_i \text{ est uniquement déterminé ssi : } r_i - (r_{i-1} - r_i) \geqslant m + n - r_{i-1}, \\ \text{i.e. } 2r_i \geqslant m + n. \end{gathered}$$

Notons $s = \frac{n+m}{2}$. Nous devons démontrer d'abord que $\Pi_{0,\ell(s)}$ est uniquement déterminé, et ensuite que $\Gamma_{\ell(s)+1}$ ne peut pas l'être. On peut statuer sur la possibilité de déterminer $\Pi_{0,\ell(s)}$ en considérant seulement $\Gamma_{\ell(s)}$, puisque r_i est une grandeur décroissante. On a, pour $i = \ell(s)$:

$$\begin{aligned} r_i &\geqslant s = \frac{n+m}{2}, \\ 2r_i &\geqslant n + m. \end{aligned}$$

Pour l'impossibilité de déterminer $\Gamma_{\ell(s)+1}$, on fait un raisonnement similaire, en utilisant la définition de $\ell(s)$ (on a toujours $i = \ell(s)$) :

$$\begin{aligned} r_{i+1} &< s, \\ 2r_{i+1} &< m + n. \end{aligned}$$

Il s'ensuit que la condition nécessaire pour la déterminabilité de Γ_i énoncée plus haut n'est pas remplie. ■

Il est aisé de déduire le corollaire suivant, avec lequel nous nous rapprochons de la mise en place de l'algorithme sous-quadratique recherché.

Corollaire 8.5. *Soient U et V deux polynômes (toujours avec $\deg U > \deg V$). Soient deux entiers m et s dans $[\![0 \dots \deg U]\!]$, et $s \geqslant 0$. Si l'inégalité $s \geqslant \frac{n+m}{2}$ est vérifiée, alors :*

$$\Pi_{0,\ell(s)}(U, V) = \Pi_{0,\ell(s-m)}(U \operatorname{div} X^m, V \operatorname{div} X^m).$$

À partir de cette proposition, on déduit un algorithme récursif pour réaliser le calcul qui nous intéresse. La conception d'un algorithme exact dans ce cas n'est pas une mince affaire : [AHU74] donne un algorithme incorrect, et l'algorithme présenté par [GY79] est d'une élégance discutable. Nous présentons ici un algorithme qui a l'avantage de mieux se comporter vis-à-vis de l'équilibrage des récursions. Cela est obtenu au prix de quelques contorsions.

Nous supposons, en entrée de notre algorithme, que nous disposons d'un couple de polynômes (U, V) qui ont été obtenus sous la forme d'un couple $R_{\ell(z)}, R_{\ell(z)+1}$. En d'autres termes, on sait que $\deg U \geqslant z > \deg V$. Le cas « typique » que l'on utilise en pratique est bien sûr celui où $z = \deg U$, mais nous allons voir que d'autres valeurs de z sont aussi utilisées au cours des récursions. En sortie, l'algorithme renvoie la matrice $\Pi_{0,\ell(s)}(U, V)$, où s est un entier tel que $s \geqslant \frac{n}{2}$, toujours en notant $n = \deg U$ (on rappelle que par construction de $\ell(s)$, on a toujours $\ell(\lceil s \rceil) = \ell(s)$). Il se décompose en plusieurs étapes comme suit :

- Si $s > \deg V$, retourner I_2.
  Si $s = \deg V$, retourner la matrice $T_1 = \begin{pmatrix} 0 & 1 \\ 1 & -\Gamma_1 \end{pmatrix}$.
  Sinon poursuivre.
- Calculer récursivement $\Pi_{0,\ell(\frac{z+s}{2})}(U, V)$. Cette matrice s'obtient en utilisant le corollaire 8.5 vu précédemment. Pour l'employer, on prend la plus grande valeur de m possible, c'est-à-dire $s - (n - z)$. L'identité que l'on utilise est donc :

$$\Pi_{0,\ell(\frac{z+s}{2})}(U, V) = \Pi_{0,\ell(\frac{z+s}{2}-m)}(U \operatorname{div} X^m, V \operatorname{div} X^m).$$

- En déduire $U' = R_{\ell(\frac{z+s}{2})}$ et $V' = R_{\ell(\frac{z+s}{2})+1}$, par le calcul du produit matriciel :

$$M_{\ell(\frac{z+s}{2})} = M_0 \Pi_{0,\ell(\frac{z+s}{2})}(U, V).$$

  On sait par construction que $\ell(\frac{z+s}{2})$ est tel que $r_{\ell(\frac{z+s}{2})} \geqslant \lceil \frac{z+s}{2} \rceil > r_{\ell(\frac{z+s}{2})+1}$.
- Calculer récursivement $\Pi_{\ell(\frac{z+s}{2}),\ell(s)}(U, V) = \Pi_{0,\ell(s)}(U', V')$. Comme précédemment, on utilise le corollaire 8.5 pour réduire le degré des polynômes considérés. La plus grande valeur possible de m est $2s - \deg U'$. On utilise l'identité :

$$\Pi_{0,\ell(s)}(U', V') = \Pi_{0,\ell(s-m)}(U' \operatorname{div} X^m, V' \operatorname{div} X^m).$$

Algorithme **PartialGCD**
Entrée : Deux polynômes U, V dans $K[X]$,
deux entiers z et s tels que $\deg U \geqslant z > \deg V$ et $s \geqslant \frac{\deg U}{2}$
Sortie : La matrice $\Pi_{0,\ell(s)}(U, V)$

```
function PartialGCD(U,V,z,s)
    KP:=Parent(U); X:=KP.1; n:=Degree(U);
    assert (z le n) and (z gt Degree(V)) and (s ge n/2);

    if s gt Degree(V) then
        return IdentityMatrix(KP,2);
    elif s eq Degree(V) then
        return Matrix(2,2,[KP|0,1,1,-(U div V)]);
    end if;

    m:=s-(n-z);
    pi_left:=PartialGCD(U div X^m, V div X^m, z-m, Ceiling((z+s)/2)-m);

    next:=Vector([U,V])*pi_left; nU:=next[1]; nV:=next[2];

    m:=2*s-Degree(nU);
    pi_right:=PartialGCD(nU div X^m, nV div X^m, Ceiling((z+s)/2)-m,s-m);

    return pi_left * pi_right;
end function;
```

Programme 8.2: Algorithme *partial-gcd* sous-quadratique

– Retourner $\Pi_{0,\ell(s)}(U, V)$ qui est le produit des deux matrices calculées lors des appels récursifs.

Le programme 8.2 est une implantation fidèle de cet algorithme en langage Magma. Cet algorithme fournit le résultat correct, y compris lorsque $z \neq n$. On n'a pas effectué de mesures quant au comportement asymptotique effectif de cette implantation. Ce n'est pas notre objet (nous ne présentons ici qu'un modèle).

Généralisation du champ d'application

Il convient de noter que l'algorithme que l'on vient de décrire se généralise aisément pour calculer n'importe quelle matrice $\Pi_{0,\ell(s)}$ sans la contrainte $s \geq \frac{n}{2}$. Il suffit d'appliquer le même procédé que dans [GY79] : on calcule $\Pi_{0,\ell(n/2)}$, puis $\Pi_{0,\ell(n/4)}$, $\Pi_{0,\ell(n/8)}$, et ainsi de suite. Toutefois, nous ne détaillons pas cette opération car elle sort de notre cadre d'intérêt.

Complexité du calcul de générateur linéaire

Nous voulons exprimer la complexité du calcul du générateur linéaire pour une suite, représentée par une série $A(X)$. Avec les notations précédemment établies, ce générateur, pourvu que l'on sache qu'un générateur Φ existe avec $\delta(\Phi) \leqslant N$, est obtenu en calculant

la matrice $\Pi_{0,\ell(N)}(X^{2N}, A \bmod X^{2N})$. La complexité de ce calcul s'évalue donc de la même manière que la complexité du calcul de $\Pi_{0,\ell(n/2)}(U, V)$ d'une manière générale (où $n = \deg U$). Ce coût est solution de l'équation :

$$\mathsf{C}(n) = 2\mathsf{C}(n/2) + c\mathsf{M}(n).$$

Dans cette équation, c est une constante, et $\mathsf{M}(n)$ est le coût de la multiplication de deux polynômes de degré n. On obtient immédiatement $\mathsf{C}(n) \in O(\mathsf{M}(n) \log n)$. Si l'on prend pour $\mathsf{M}(n)$ la complexité obtenue par l'usage de la FFT, on a le résultat asymptotique :

$$\mathsf{C}(n) \in O(n \log^2 n).$$

Notons l'importance prise dans cette analyse par l'usage d'un algorithme rapide de multiplication comme la FFT. Il est essentiel pour la qualité du résultat et ce n'est pas un hasard. Le travail que l'on a fourni a consisté explicitement à « tailler » le problème de telle sorte que l'on puisse utiliser des algorithmes de multiplication rapide sur des gros polynômes, plutôt que de faire beaucoup de multiplications très déséquilibrées comme c'est le cas dans la présentation élémentaire de l'algorithme.

8.3 Cas matriciel : hypothèses de généricité

Nous nous concentrons maintenant plus explicitement sur le cas du calcul de générateurs linéaires matriciels. Notre approche consiste à présenter la généralisation de l'algorithme de Berlekamp-Massey mise au point par Coppersmith, et à lui apporter une accélération sous-quadratique comme on a vu que cela pouvait se faire pour l'algorithme d'Euclide. Un sous-produit du résultat est une amélioration sous-quadratique de l'algorithme de Berlekamp-Massey scalaire, sans passer par l'algorithme d'Euclide.

Nous allons nous intéresser au calcul de générateurs linéaires, au sens de la définition 6.1 et de l'exemple en page 104. Pour une suite de matrices de taille $m \times n$, c'est-à-dire en prenant la notation sous forme de séries, pour $A(X) \in K[[X]]^{m\times n}$, on veut calculer des générateurs linéaires vectoriels à droite, en d'autres termes des éléments de $K[X]^{n\times 1}$. On fera quelques brèves excursions pour l'examen du calcul de générateurs linéaires matriciels (donc des éléments de $K[X]^{n\times n}$), ce qui nous amène à formuler plus généralement notre intérêt pour le problème suivant :

Problème : Soient m, n, r entiers et $A \in K[[X]]^{m\times n}$. Calculer $F \in K[X]^{n\times r}$ tel que :

$$A(X)F(X) = G(X) \in K[X]^{m\times r}.$$

Comme on l'a vu pour l'algorithme de Berlekamp-Massey scalaire, on a besoin d'un résultat pour prouver que la sortie de l'algorithme est bien un générateur linéaire. C'est l'objet du théorème 8.1 dans le cas scalaire. On lui donne ici une extension matricielle qui nous montre que l'introduction de la non-commutativité amène une dimension frappante : l'hypothèse naturelle de ce théorème n'est plus une hypothèse d'existence d'un générateur linéaire comparable à celui que l'on cherche, mais l'existence d'un générateur linéaire *de l'autre côté*[2].

[2]Ces quantités sont reliées dans les situations non dégénérées. On va voir d'ailleurs que les algorithmes présentés ici en fournissent une preuve constructive.

Théorème 8.6. *Soit $A \in K[[X]]^{m\times n}$, et r un entier. On suppose que A dispose d'une description en fraction rationnelle à gauche, c'est-à-dire qu'il existe deux matrices polynomiales $N(X)$ et $D(X)$, de tailles respectives $m \times n$ et $n \times n$, avec D unimodulaire, telles que $A = D^{-1}N$. Soient maintenant trois matrices polynomiales $F(X) \in K[X]^{n\times r}$, $G(X) \in K[X]^{m\times r}$, $E(X) \in K[[X]]^{m\times r}$, et un entier t vérifiant :*

$$AF = G + X^t E,$$

On a alors :

$$t - \delta(F, G) \geqslant \delta(D, N) \Rightarrow E = 0.$$

Démonstration. La preuve est une généralisation facile de la preuve du théorème 8.1. Il suffit de multiplier le produit AF par la matrice D à gauche. On obtient alors :

$$DAF = NF = DG + X^t DE.$$

En raisonnant sur les degrés comme dans le cas scalaire, on obtient :

$$\deg N < d, \deg F \leqslant t - d \Rightarrow \deg NF < t,$$
$$\deg D \leqslant d, \deg G < t - d \Rightarrow \deg DG < t,$$

Il s'ensuit que le produit DE est inévitablement nul. Comme la matrice D est inversible, cela implique la nullité de E. ■

Pour utiliser le théorème 8.6 dans notre exposition, nous supposons que la matrice A peut être écrite sous la forme $D^{-1}N$. Comme cela est visible dans la preuve du théorème, la quantité $\delta(D, N)$ est importante, nous allons donc la baptiser d. Il est important de noter que le calcul des matrices polynomiales N et D n'a pas besoin d'être mené. Leur seule *existence* est suffisante.

On fait une seconde supposition sur les colonnes de la matrice A. Nous appelons s le plus petit indice tel que les colonnes des matrices (scalaires) $[X^0]A, \ldots [X^{s-1}]A$ engendrent l'espace $K^{m\times 1}$ entier, en supposant donc implicitement qu'un tel indice s existe[3]. Si ce n'est pas le cas, alors l'ensemble des colonnes des coefficients de A forme un sous-espace vectoriel strict. On peut donc combiner certaines lignes de A pour faire apparaître une ligne nulle dans A. En ôtant cette ligne et en diminuant m, on peut donc se ramener au cas générique.

En fonction des deux grandeurs s et d que l'on vient d'introduire, les deux algorithmes que nous présentons maintenant (l'algorithme de Coppersmith et notre amélioration sous-quadratique) fournissent, en suivant le même schéma de calcul, une preuve de l'assertion qui suit.

Proposition 8.7. *Un générateur linéaire à droite $F \in K[X]^{n\times 1}$ pour A peut être calculé de manière déterministe en utilisant uniquement les L premiers coefficients de A (c'est-à-dire $A \bmod X^L$), où :*

$$L = s + \left\lceil \frac{m+n}{n} d \right\rceil.$$

Le générateur linéaire F calculé vérifie :

$$\delta(F, AF) \leqslant s + \left\lceil \frac{m}{n} d \right\rceil.$$

[3]Cet entier s joue le même rôle que l'entier s introduit dans le cas scalaire qui valait $1 + v$, v étant la valuation de A.

8.4 L'algorithme proposé par Coppersmith

Nous commençons par exposer la généralisation de l'algorithme de Berlekamp-Massey au cas matriciel proposée par Coppersmith [Cop94] lorsqu'il a décrit l'algorithme de Wiedemann par blocs. Cet algorithme, si on l'applique dans le cas $m = n = 1$, est très exactement l'algorithme de Berlekamp-Massey tel qu'on l'a décrit en 8.2.1. On suit donc le fil de la description qui en a été faite.

8.4.1 Schéma

Comme dans le cas de l'algorithme de Berlekamp-Massey *scalaire*, on travaille par approximations successives. La différence est ici que le nombre de « candidats générateurs » passe de 2 à $m + n$. À chacun de ces candidats générateurs f_j, on associe un second membre g_j, un terme d'erreur e_j, et un entier δ_j. Les quantités traitées sont donc :

$$\begin{aligned} f_1(X), \ldots, f_{m+n}(X) &\in K[X]^n, \\ g_1(X), \ldots, g_{m+n}(X) &\in K[X]^m, \\ e_1(X), \ldots, e_{m+n}(X) &\in K[X]^m, \\ \delta_1, \ldots, \delta_{m+n} &\in \mathbb{N} \cup \{-\infty\}. \end{aligned}$$

Ces grandeurs sont regroupées en des matrices dont elles forment les $m + n$ colonnes. On fabrique ainsi :

$$\begin{aligned} f &\in K[X]^{n\times(m+n)}, \\ g &\in K[X]^{m\times(m+n)}, \\ e &\in K[X]^{m\times(m+n)}, \\ \Delta &\in (\mathbb{N} \cup \{-\infty\})^{m+n}. \end{aligned}$$

Pour chacun des f_j, c'est-à-dire pour chacune des colonnes des matrices précédentes (l'indice j est employé au cours de cette description pour désigner une colonne), l'équation suivante est vérifiée. Elle joue le même rôle central que dans le cas scalaire :

$$\begin{aligned} \forall j, \qquad Af_j &= g_j + X^t e_j, \\ \delta(f_j, g_j) &\leqslant \delta_j. \end{aligned} \tag{C1}$$

Une autre condition va nous donner le lien entre les différentes étapes du processus.

$$\operatorname{rg}(e(0)) = m. \tag{C2}$$

Comme dans le cas scalaire, et dans le but d'utiliser de façon similaire le théorème 8.6, on va chercher à « éloigner » l'erreur autant que possible au fur et à mesure du processus. Pour cela, on avance de l'étape t à $t + 1$ de telle sorte que la moyenne des coefficients δ_j avance de strictement moins que 1 (on va voir que l'incrément est $\frac{m}{m+n}$).

Les quantités que nous traitons au cours de l'algorithme évoluent d'étape en étape. Aussi, pour distinguer la valeur de f_j entre les différentes étapes, on utilise la notation $f_j^{(t)}$, et plus généralement l'exposant $^{(t)}$, pour distinguer les valeurs associées à l'étape t de l'algorithme. On omettra souvent cette précision lorsque le contexte le permet.

8.4.2 Initialisation

Pour commencer, on détaille la façon dont on initialise le processus. Cette initialisation doit être effectuée avec soin car elle conditionne la correction de l'algorithme. L'initialisation que nous proposons est une généralisation de celle proposée en 8.2.1.

La donnée de départ est l'entier s, défini tel que les colonnes des s matrices $[X^0]A$ jusqu'à $[X^{s-1}]A$ engendrent l'espace $K^{m\times 1}$. De ces colonnes on peut extraire une base de $K^{m\times 1}$. On le fait en choisissant m vecteurs $r_1, \ldots, r_m$ appartenant à la base canonique de $K^{n\times 1}$, ainsi que m entiers $i_1, \ldots, i_m$ appartenant à $[\![0, s-1]\!]$, tels que les vecteurs $v_k = \big([X^{i_k}]A\big)\, r_k$ pour $k \in [\![1, m]\!]$ constituent une base (un couple (i_k, r_k) identifie ainsi dans quel coefficient et dans quelle colonne on choisit un vecteur).

L'indice de départ est s (comme c'était déjà le cas pour Berlekamp-Massey scalaire[4]). La matrice $f^{(s)}$ est initialisée comme suit. Pour les n premières colonnes, on choisit la matrice identité I_n. Les m colonnes restantes sont choisies comme étant les $X^{s-i_k} r_k$, pour $k \in [\![1, m]\!]$:

$$f^{(s)} = \left(\begin{array}{c|ccc} I_n & X^{s-i_1} r_1 & \cdots & X^{s-i_m} r_m \end{array}\right).$$

On prend s comme valeur initiale commune des δ_j. De cette façon, la condition (C1) est valide pour chaque colonne (puisque pour tout j, on a $\delta_j = s$, il n'y a pas de contrainte sur la détermination de g_j et e_j).

La condition (C2) sur le rang de la matrice $e(0)$ est une conséquence facile du choix des i_k et des r_k : si l'on note $\beta(X)$ la matrice $m \times m$ formée par les m dernières colonnes de $e^{(s)}(X) = A(X) f^{(s)}(X) \operatorname{div} X^s$, on voit que le choix des i_k et des r_k fait que les colonnes de $\beta(0)$ sont les vecteurs v_k, qui forment une base de $K^{m\times 1}$. La matrice $\beta(0)$ est donc de rang maximal m. Il en est alors de même pour la matrice $e^{(s)}(0)$.

Pour permettre par la suite de prouver que l'algorithme fournit un résultat non trivial, on généralise la preuve faite dans le cas scalaire en 8.2.1. On considère la matrice $h(X)$ de taille $(m+n) \times (m+n)$ obtenue en concaténant verticalement les matrices f et e. Initialement, la matrice $h^{(s)}$ est :

$$h^{(s)} = \left(\begin{array}{c|ccc} I_n & X^{s-i_1} r_1 & \cdots & X^{s-i_m} r_m \\ \hline \vdots & & \beta(X) & \end{array}\right).$$

Comme $i_k < s$ pour tout k, il est facile de voir que la partie en haut à droite de $h^{(s)}(0)$ est nulle. On peut donc facilement calculer le déterminant de la matrice $h^{(s)}$, qui est :

$$\det h^{(s)}(0) = \det \beta(0) \neq 0.$$

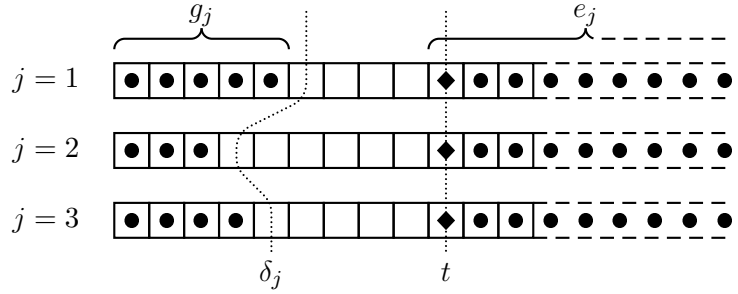
La matrice $h^{(s)}(X)$ est donc *unimodulaire*, c'est-à-dire inversible dans $K[[X]]^{(m+n)\times(m+n)}$, puisque son déterminant est une unité dans $K[[X]]$. Cette information nous sera utile ultérieurement.

8.4.3 Description de l'itération

On souhaite passer de l'étape t à l'étape $t+1$. Examinons la répartition des coefficients non nuls des produits $A(X) f_j(X)$. Ces produits sont au nombre de $m+n$, ils constituent les

[4] Le choix de s comme nom de variable correspond à *start*.

colonnes de la matrice Af. En considérant de telles colonnes comme des polynômes ayant des vecteurs comme coefficients, on reprend des schémas semblables à ceux qui avaient été faits dans le cas scalaire. Sur chaque ligne, on représente les coefficients non nuls de $A(X)f_j(X)$, en marquant une case d'un point. La k-ème case d'une ligne est vide (ou marquée 0) si l'on sait par construction que le coefficient correspondant $[X^k]Af_j$ est le vecteur nul. Pour prendre un exemple fictif, on a le schéma suivant, à l'étape t :



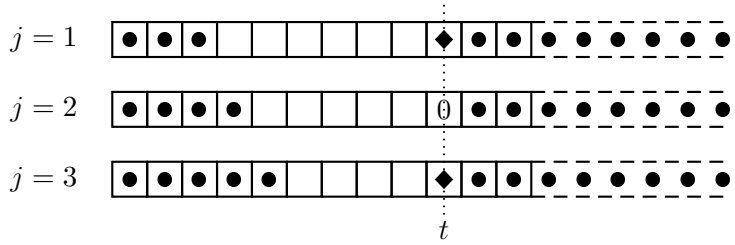
On a marqué certaines cases par ◆ pour le cas spécifique du coefficient de degré t. En vertu de la condition (C2), on sait que la matrice $e(0)$ est de rang m.

On souhaite annuler les coefficients de degré t. Cela doit être fait sans modifier profondément les valeurs des δ_j. Le procédé employé est en fait une sorte d'élimination gaussienne. La matrice dont on veut annuler les coefficients est $[X^t]Af$, c'est-à-dire $e(0)$. On sait que cette matrice, de taille $m \times (m+n)$, est de rang m. On s'autorise les opérations suivantes, définies invariablement sur les colonnes de $e(0)$, $e(X)$, où $f(X)$. Comme ce sont des opérations sur les colonnes, elles reviennent toujours à la multiplication *à droite* par une certaine matrice carrée.

– Échanger deux colonnes.
– Ajouter un multiple de la colonne j_1 à la colonne j_2 pourvu que l'on ait $\delta_{j_1} \leqslant \delta_{j_2}$.
– Multiplier une colonne par X (cette opération doit être vue du point de vue polynomial).

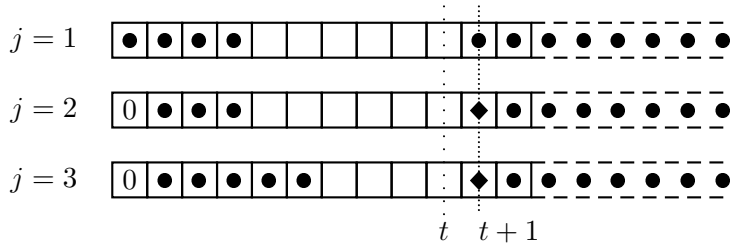
Ces opérations correspondent respectivement à la multiplication à droite par une matrice de permutation, par une matrice de transvection, et dans le dernier cas par une matrice diagonale (avec des 1 partout et un X pour l'un des coefficients). Dans les deux premiers cas, il s'agit de matrices unimodulaires (le déterminant vaut 1), et dans le dernier cas le déterminant vaut X.

Ces opérations sont suffisantes. On commence par réordonner les colonnes par ordre des δ_j croissants. On fait ensuite une élimination gaussienne sur les colonnes de la matrice, en additionnant uniquement des colonnes à des colonnes d'indice plus élevé. Cela a pour effet de conserver seulement m colonnes non nulles dans la matrice $e(0)$, puisque le rang de cette matrice est m. Pour les indices correspondant aux colonnes que l'on a ainsi annulées, l'équation (C1) est maintenant vérifiée sans modification pour l'étape $t+1$. Les indices δ_j correspondant à ces colonnes sont inchangés. Sur notre représentation graphique des coefficients non nuls, le processus d'élimination gaussienne a produit des 0 pour certains coefficients, sans autre modification (hormis la permutation des f_j qui correspond au tri des δ_j).



Pour garder les choses ordonnées, on se permet une nouvelle permutation des indices des colonnes, de manière à avoir des colonnes nulles pour les n premiers indices, de 1 à n, et les m autres colonnes ensuite. L'ensemble des opérations effectuées jusqu'à présent (le tri et l'élimination gaussienne) rentrent dans le cadre des deux premières opérations que nous nous sommes autorisées. Elles peuvent être synthétisées, ensemble, par la multiplication à droite par une matrice que l'on nomme $\tau^{(t)}$. Cette matrice est un produit de matrices de permutation et de matrices de transvection. C'est donc une matrice scalaire de déterminant 1.

L'étape qui suit consiste à se débarrasser des colonnes restantes dans la matrice $e(0)$. Ces colonnes sont représentées par les losanges sur la figure qui précède. On sait que la sous-matrice de $e(0)$ formée par ces m colonnes est de rang maximal m, par construction. On fait comme on a déjà fait pour Berlekamp-Massey scalaire : on « pousse » ces colonnes en multipliant les f_j correspondants par X, c'est-à-dire la troisième des opérations mentionnées plus haut. Cela a pour effet de décaler d'un coefficient les représentations graphiques qui nous servent de support. Les coefficients δ_j correspondants sont alors incrémentés de 1. Cette étape correspond à la multiplication à droite par la matrice $D = \mathrm{diag}(1, \ldots, 1, X, \ldots, X)$ où les occurrences de X sont au nombre de m. La situation est maintenant :



Nous avons maintenant une situation qui correspond parfaitement à ce que l'on doit avoir à l'étape $t+1$ pour satisfaire la condition (C1). Par ailleurs, les losanges ont été conservés sur le dessin. Ils représentent des colonnes qui, mises ensemble, ont un rang maximal. Il s'ensuit que la condition (C2) est elle aussi vérifiée, la nouvelle matrice $e^{(t+1)}$ s'écrivant :

$$e^{(t+1)} = e^{(t)} \tau^{(t)} D \frac{1}{X}.$$

Cette formulation de la nouvelle valeur de e en fonction de l'ancienne ainsi que des matrices $\tau^{(t)}$ et D n'est pas spécifique, et s'obtient de manière identique pour les autres grandeurs que sont f_j, g_j, et δ_j. Si l'on adopte la notation $P^{(t)} = \tau^{(t)} D$, on vient en fait de fournir une preuve constructive du résultat suivant. La procédure correspondante est consignée dans le programme 8.3, en langage Magma, de telle sorte que l'on espère le résultat compréhensible et relativement complet.

Théorème 8.8. *Si les conditions (C1) et (C2) sont vérifiées à l'étape t, il existe un algorithme qui, en fonction de $e^{(t)}(0)$ et $\Delta^{(t)}$, produit une matrice $\tau^{(t)}$ telle que (en notant $P^{(t)} = \tau^{(t)} D$) :*

$$f^{(t+1)} = f^{(t)} P^{(t)}, \qquad g^{(t+1)} = g^{(t)} P^{(t)}, \qquad e^{(t+1)} = e^{(t)} P^{(t)} \frac{1}{X},$$
$$\Delta^{(t+1)} = \Delta^{(t)} P^{(t)}, \qquad h^{(t+1)} = D^{-1} h^{(t)} \tau^{(t)} D,$$

où l'on entend le produit $\Delta^{(t)} P^{(t)}$ au sens des lois d'anneau $(\max, +)$ sur $\mathbb{N} \cup \{-\infty\}$. Ces grandeurs satisfont alors les conditions (C1) et (C2) à l'étape $t+1$. De plus, on a $\sum_j \delta_j^{(t+1)} - \sum_j \delta_j^{(t)} = m$.

Ce théorème implique en particulier que le déterminant de la matrice $h^{(t)}$ est constant et donc en particulier que cette matrice reste unimodulaire.

Démonstration. On a déjà démontré tout le cheminement inclus dans cet énoncé. Il convient de commenter brièvement la notation $\Delta^{(t)} P^{(t)}$. On munit l'ensemble $\mathbb{N} \cup \{-\infty\}$ des lois d'anneau $(\max, +)$. Ainsi, l'opérateur deg est un morphisme d'anneaux de $K[X]$ vers $\mathbb{N} \cup \{-\infty\}$. C'est en ce sens que doit être entendu le produit de matrices en question. Il est facile de vérifier que cette expression de $\Delta^{(t+1)}$ est correcte.

L'expression de $h^{(t+1)}$ se déduit de celle de $f^{(t+1)}$ et de $e^{(t+1)}$. ■

8.4.4 Terminaison

Décrire l'algorithme de Coppersmith une fois que l'on a obtenu le théorème 8.8 se fait sans difficulté : on itère l'algorithme ComputePMatrix1 du programme 8.3 jusqu'à obtention d'un générateur. Il est aisé de constater que l'on peut se contenter, pour les données de l'algorithme, de ne maintenir que les matrices $e(0)$ et f (ainsi bien sûr que les coefficients δ_j). On discute maintenant à quel moment et pourquoi un générateur est obtenu.

La valeur moyenne $\overline{\delta}$ des coefficients δ_j, comme on l'avait annoncé, augmente d'exactement $\frac{m}{m+n}$ lorsque t augmente de 1. C'est une conséquence du théorème 8.8. Nous pouvons donc exprimer la moyenne de la différence $t - \overline{\delta}$.

$$t - \overline{\delta} = t - (s + (t-s)\frac{m}{m+n}) = (t-s)\frac{n}{m+n}.$$

Pour $t = s + \left\lceil \frac{m+n}{n} d \right\rceil$, on applique l'égalité qui précède, et on trouve :

$$t - \overline{\delta} \geqslant d.$$

Par conséquent, pour au moins un indice j, on a $t - \delta_j \geqslant d$. Cette condition, en vertu du théorème 8.6, implique que e_j est nul, donc que f_j est un générateur linéaire vectoriel, c'est-à-dire ce que l'on cherche. On prouve aisément qu'il ne s'agit pas d'un générateur trivial, car cela provoquerait la nullité d'une colonne de la matrice $h(X)$. Or on sait que cette matrice est constamment unimodulaire, donc elle ne peut avoir une colonne nulle.

Une autre façon plus terre-à-terre de vérifier si l'on a un générateur est la suivante. Si f_j est un générateur à partir de l'étape t, alors, dans toutes les étapes suivantes, la colonne j de la matrice $e(0)$ est nulle en *entrée* de l'algorithme ComputePMatrix1. Si une telle situation se répète plusieurs fois, on peut parier que l'on a un générateur.

Algorithme **ComputePMatrix1** « 1 » car on avance d'une étape

Entrée : Un couple E==(e, Δ), avec:
e une matrice de taille $m \times (m+n)$ (la matrice $e(0)$).
Δ un vecteur de $m+n$ entiers.

Sortie : P telle que définie par 8.8.

```
function ComputePMatrix1(E)
    e:=E[1]; delta:=E[2]; m:=Nrows(e); n:=Ncols(e)-m; assert #delta eq m+n;
    KP:=CoefficientRing(e); X:=KP.1;

    // Tri.
    sorted:=[c[2] : c in Sort([<delta[i],i> : i in [1..m+n]])];
    P:=Matrix(m+n,m+n,[<sorted[i],i,KP!1> : i in [1..m+n]]);
    e*:=P;

    // Élimination gaussienne
    busy:=[false : i in [1..m+n]];
    for i in [1..m] do
        pivots:=[j : j in [1..m+n] | e[i][j] ne 0 and not busy[j]];
        if IsEmpty(pivots) then continue; end if;

        j:=pivots[1]; Remove(~pivots,1); busy[j]:=true;
        for k in pivots do
            c:= - e[i][k] / e[i][j]; AddColumn(~e,c,j,k); AddColumn(~P,c,j,k);
        end for;
    end for;

    // Ici, on ne retrie pas.
    for j in [j : j in [1..m+n] | busy[j]] do MultiplyColumn(~P,X,j); end for;
    return P;
end function;
```

Programme 8.3: Calcul de $P^{(t)}$

Il reste à conclure la preuve de la proposition 8.7 page 140. On doit montrer que $\delta(f_j, g_j)$ est inférieur ou égal à la borne annoncée, à savoir $s + \left\lceil \frac{m}{n} d \right\rceil$. Il faut donc examiner $\overline{\delta}$, et tenter d'obtenir la bonne borne supérieure.

$$\begin{aligned}
\overline{\delta} &= s + \frac{m}{m+n} \left\lceil \frac{m+n}{n} d \right\rceil, \\
&= s + \frac{m}{m+n} d + \frac{m}{m+n} \left\lceil \frac{m}{n} d \right\rceil, \\
&= s + \left\lceil \frac{m}{n} d \right\rceil - \frac{n}{m+n} \left(\left\lceil \frac{m}{n} d \right\rceil - \frac{m}{n} d \right), \\
&\leqslant s + \left\lceil \frac{m}{n} d \right\rceil.
\end{aligned}$$

8.4.5 Obtention d'une description en fractions rationnelles

Tel que nous avons décrit l'algorithme, le résultat obtenu est manifestement un générateur linéaire vectoriel. On peut attendre un peu mieux que ce simple générateur dans le cas

« générique », où l'on choisit d'exclure les cas particuliers.

Si, au cours de l'évolution de l'algorithme, tout se passe « comme prévu », les entiers δ_j augmentent de concert, et sont tous concentrés autour de leur valeur moyenne. En effet, si $\delta_j < \overline{\delta}$ après l'itération t, il y a de fortes chances pour que la colonne j serve de pivot dans l'élimination gaussienne à l'étape $t+1$, ce qui provoque une incrémentation de δ_j.

Cette situation se maintient tant qu'aucun des δ_j ne devient inférieur à la borne $t-d$. Lorsque c'est le cas, la colonne j se stabilise, car elle a convergé vers un générateur linéaire, en vertu du théorème 8.8 : la colonne correspondante de la matrice $e(X)$ devient nulle, et donc la colonne j ne sert plus à aucun moment de pivot et ne reçoit plus aucun ajout d'une autre colonne. Si l'évolution des autres coefficients δ_j continue conformément au comportement moyen, on démontre sans peine qu'une itération plus tard, il y a exactement n colonnes pour lesquelles $\delta_j \leqslant t-d$. Les f_j correspondants sont tous des générateurs linéaires vectoriels.

Nous montrons que dans cette situation, les générateurs vectoriels forment ensemble un générateur linéaire matriciel unimodulaire qui nous donne donc une description en fraction rationnelle à droite pour la série A. Pour obtenir cette propriété, examinons la matrice $h^{(t)}$. En permutant les colonnes de façon appropriée, on peut nommer $f^+(X)$ la sous-matrice de taille $n \times n$ constituée des n générateurs linéaires vectoriels (les f_j). On place cette matrice en haut à gauche. Le « reste » de la matrice f est noté $f^-(X)$. La matrice h s'écrit donc :

$$h^{(t)} = \left(\begin{array}{c|c} f^+(X) & f^-(X) \\ \hline e^+(X) & e^-(X) \end{array}\right).$$

Comme les colonnes de $f^+(X)$ sont des générateurs linéaires vectoriels, on déduit que $e^+(X) = 0$. Ensuite, on sait que la matrice $e^-(X)$ a pour terme constant une matrice de rang m, en vertu de la condition (C2). La matrice $e^-(X)$ est donc unimodulaire. C'est aussi le cas de $h^{(t)}(X)$, donc $f^+(X)$ possède aussi cette propriété. La matrice $f^+(X)$ est donc effectivement un générateur linéaire matriciel unimodulaire.

Si l'on peut montrer que « si tout se passe bien », suffisamment de colonnes vérifient $\delta_j \leqslant t-d$ à partir d'une certaine itération, la composante réellement intéressante est le moment où cela intervient. Dans tous les cas, si on laisse t augmenter suffisamment, il est certain que n colonnes vérifient cette identité. Mais on est hélas incapable d'obtenir une borne correcte sur le moment où une telle situation est atteinte. La meilleure borne que l'on peut montrer est $s+(1+m)d$. Et dans des cas très particuliers (très éloignés du cas générique), cette borne peut être atteinte. C'est bien au-delà de $s+\left\lceil \frac{m+n}{n} d \right\rceil$ qui est l'itération à partir de laquelle on peut garantir que l'on dispose d'un générateur linéaire vectoriel.

8.4.6 Complexité

Évaluons la complexité de l'algorithme. À chaque étape t, les opérations suivantes sont effectuées.

- Calcul de la matrice $e^{(t)}(0) = [X^t](Af^{(t)})$.
- Application de l'algorithme ComputePMatrix1.
- Multiplication de $f^{(t)}$ par la matrice P ainsi calculée.

On peut remarquer que les coefficients des m colonnes de la matrice $e(0)$ qui ont servi de pivot pour l'élimination gaussienne réapparaissent automatiquement à l'étape $t+1$. Seules n colonnes doivent donc être calculées. Le degré moyen des colonnes de f est donné par $\overline{\delta}$ qui vaut $t\frac{m}{m+n}$ (asymptotiquement). Le nombre de multiplications dans le corps de base

nécessaires pour calculer la matrice $e^{(t)}(0)$, en tenant compte de l'information qui peut être recyclée depuis les étapes précédentes, est :

$$t\frac{m}{m+n}mn^2 = t\frac{m^2n^2}{m+n}.$$

L'algorithme ComputeMatrix1 prend un temps qui ne varie pas, du moins tel qu'on l'a décrit. Toutefois, pour calculer $f^{(t)}P^{(t)}$ efficacement, on le calcule au cours de la construction de la matrice P (qui ne devient plus qu'un sous-produit de l'algorithme). Le nombre de multiplications à effectuer est gouverné par le nombre de multiplications par des matrices de transvection (les additions de multiples de colonnes). Cela amène, compte tenu du degré moyen de f, $t\frac{m}{m+n}mn$ multiplications dans le corps de base.

La valeur maximale de t étant $\frac{m+n}{n}d$, on déduit que le nombre de multiplications requises par l'algorithme est :

$$\begin{aligned}\frac{1}{2}\left(\frac{m+n}{n}d\right)^2\frac{m^2(n^2+n)}{m+n} &= \frac{(m+n)m^2(n^2+n)}{2n^2}d^2,\\ &= \frac{1}{2}(m+n)(md)^2\left(1+\frac{1}{n}\right).\end{aligned}$$

8.5 Une version sous-quadratique

Nous décrivons maintenant comment les différents éléments présentés peuvent être rassemblés pour former une généralisation matricielle de l'algorithme de Berlekamp-Massey qui ait une complexité sous-quadratique. La complexité de l'algorithme que nous présentons est en $O(N\log^2 N)$. Une évaluation exacte du nombre d'opérations requises est plus délicate, mais néanmoins réalisée (en 8.5.3).

8.5.1 Structure récursive

On peut dégager un trait essentiel de la version sous-quadratique de l'algorithme d'Euclide étendu qui a été présentée en 8.2.2. En effet, une structure récursive a été adoptée, de telle sorte que des données de taille réduite soient gérées dans les pas récursifs, et qu'elles soient composées ultérieurement. Une telle approche permet de tirer parti des algorithmes rapides de multiplication de polynômes (sans lesquels l'effort serait vain). Par ailleurs, la clef de voûte de cet algorithme est la possibilité d'utiliser une information réduite pour calculer une partie des données : c'est ce qui rend les pas récursifs possibles.

Nous adoptons ici une stratégie identique. Nous montrons dans quelle mesure une information partielle nous permet d'« avancer », et nous mettons ensuite en place une structure récursive. La présentation que nous avons faite de l'algorithme de Berlekamp-Massey matriciel proposé par Coppersmith nous fournit déjà toute l'ossature d'une telle démarche. Les notations et données sont les mêmes dans l'algorithme que nous présentons. Le principe d'« approximations successives » reste valide, bien que la notion d'« étape » de l'algorithme doive surtout être comprise comme une référence à l'algorithme itératif. Nous commençons par mettre en place le résultat sur l'utilité des informations partielles. La donnée qui nous importe est la suivante :

Définition 8.9. *On appelle* k-contexte *une paire* $E = (e(X), \Delta)$ *correspondant à une étape de l'algorithme, comme expliqué en 8.4.1, et où seulement* k *coefficients de la matrice* $e(X)$ *sont connus (*e *est une classe modulo* X^k*).*

Cette notion nous permet de généraliser ainsi le théorème 8.8.

Théorème 8.10. *Un* k*-contexte* E *correspondant à l'étape* t *de l'algorithme permet de déterminer de manière unique les matrices* $P^{(t)}, \ldots, P^{(t+k-1)}$*.*

Démonstration. On démontre le résultat par récurrence sur k. Pour le cas où $k = 1$, l'énoncé coïncide avec le théorème 8.8. Dans le cas où $k > 1$, on peut utiliser le théorème 8.8 qui nous donne à tout le moins la matrice $P^{(t)}$. On connaît aussi les expressions suivantes :

$$e^{(t+1)} = e^{(t)} P^{(t)} \frac{1}{X},$$
$$\Delta^{(t+1)} = \Delta^{(t)} P^{(t)}.$$

De ces relations, on conclut aisément que $(e^{(t+1)}, \Delta^{(t+1)})$ forme un $(k-1)$-contexte. Par application de l'hypothèse de récurrence, le résultat voulu se déduit. ■

Ce résultat occupe une place centrale dans l'algorithme que l'on décrit. Il permet en particulier de justifier la notation suivante.

Définition 8.11. *Soit* E *un* k*-contexte (*k *est un entier positif). Supposons que* E *corresponde à l'étape* t *de l'algorithme. Soient* a *et* b *deux entiers tels que* $0 \leqslant a \leqslant b \leqslant k$*. On note* $\pi_E^{(a,b)}$ *la matrice polynomiale de taille* $(m+n) \times (m+n)$ *définie par :*

$$\pi_E^{(a,b)} = P^{(t+a)} \ldots P^{(t+b-1)}.$$

Dans le cas où $a = b$*, la matrice* $\pi_E^{(b,b)}$ *est l'identité.*

Cette définition nous permet d'énoncer la proposition suivante qui est un corollaire immédiat du théorème 8.10.

Corollaire 8.12. *Soit* t *une étape de l'algorithme. Soient* $f^{(t)}$*,* $g^{(t)}$*,* $e^{(t)}$*,* $\Delta^{(t)}$*, et* $h^{(t)}$ *les données correspondantes, et* $E^{(t)} = (e^{(t)}, \Delta^{(t)})$ *un* k*-contexte associé (pour un entier* $k \geqslant 0$*). On a :*

$$f^{(t+k)} = f^{(t)} \pi_{E^{(t)}}^{(0,k)}, \qquad g^{(t+k)} = g^{(t)} \pi_{E^{(t)}}^{(0,k)}, \qquad e^{(t+k)} = e^{(t)} \pi_{E^{(t)}}^{(0,k)} \frac{1}{X^k}$$
$$\Delta^{(t+k)} = \Delta^{(t)} \pi_{E^{(t)}}^{(0,k)}, \qquad h^{(t+k)} = D^{-k} h^{(t)} \pi_{E^{(t)}}^{(0,k)}.$$

Notre examen du fonctionnement de l'algorithme quadratique présenté en 8.4 montre que cet algorithme est centré sur le calcul d'une quantité principale : la matrice $\pi^{(0,L-s)}\left(E^{(s)}\right)$. Cette matrice est bien définie car on peut faire aisément de $E^{(s)}$ un $(L-s)$-contexte : il suffit de calculer. Nous montrons pourquoi c'est cette quantité qui est importante. Par définition de la matrice π, et par application de l'énoncé qui précède, on a en particulier :

$$f^{(L)} = f^{(s)} \pi_{E^{(s)}}^{(0,L-s)},$$
$$e^{(L)} = e^{(s)} \pi_{E^{(s)}}^{(0,L-s)}.$$

On a choisi L pour qu'un générateur linéaire vectoriel se trouve parmi les colonnes de $f^{(L)}$ (et même possiblement plusieurs générateurs, donnant une description en fraction rationnelle). Ce point a été démontré en 8.4.4. Il nous est possible d'identifier quels sont ces générateurs par la nullité des colonnes correspondantes de la matrice $e^{(L)}$. Donc, une fois les données d'initialisation calculées ($f^{(s)}$, $e^{(s)}$, et $\Delta^{(s)}$, les deux derniers formant $E^{(s)}$), la matrice $\pi_{E^{(s)}}^{(0,L-s)}$ est une donnée suffisante pour finir le calcul.

Notre algorithme calcule cette matrice récursivement, en utilisant seulement $E^{(s)}$ comme donnée. Nous pouvons donner le mode de fonctionnement de la procédure récursive qui, étant donné un b-contexte $E = (e, \Delta)$, calcule la matrice $\pi_E^{(0,b)}$.

- Si b vaut 1, on calcule $\pi_E^{(0,1)} = P$ par l'algorithme ComputePMatrix1.
- Dans le cas général, on forme pour commencer le $\lfloor \frac{b}{2} \rfloor$-contexte E_L obtenu par restriction :

  $$E_L = (e \bmod X^{\lfloor \frac{b}{2} \rfloor}, \Delta).$$

  Nous pouvons calculer la matrice $\pi_{E_L}^{(0,\lfloor \frac{b}{2} \rfloor)}$ récursivement. Cette matrice, par définition, est égale à la matrice $\pi_L = \pi_E^{(0,\lfloor \frac{b}{2} \rfloor)}$.
- En utilisant E et $\pi_E^{(0,\lfloor \frac{b}{2} \rfloor)}$, on calcule $E_M = E^{(t+\lfloor \frac{b}{2} \rfloor)}$ (si $E = E^{(t)}$), c'est-à-dire le contexte formé par :

  $$E_M = (e\pi_E^{(0,\lfloor \frac{b}{2} \rfloor)} \operatorname{div} X^{\lfloor \frac{b}{2} \rfloor}, \Delta\pi_E^{(0,\lfloor \frac{b}{2} \rfloor)}).$$

  Par application du théorème 8.10 et de son corollaire, on a que E_M forme ainsi un $\lceil \frac{b}{2} \rceil$-contexte.
- On utilise ce $\lceil \frac{b}{2} \rceil$-contexte pour calculer récursivement la matrice $\pi_{E_M}^{(0,\lceil \frac{b}{2} \rceil)}$. Par définition, cette matrice est égale à la matrice $\pi_R = \pi_E^{(\lfloor \frac{b}{2} \rfloor,b)}$.
- On calcule $\pi_E^{(0,b)}$ en effectuant le produit $\pi_L\pi_R$, soit :

  $$\pi_E^{(0,b)} = \pi_E^{(0,\lfloor \frac{b}{2} \rfloor)}\pi_E^{(\lfloor \frac{b}{2} \rfloor,b)}.$$

Un exemple d'implantation en Magma de cet algorithme est donné par le programme 8.4. Cet exemple ne tient pas compte de la possibilité d'utiliser la transformée de Fourier.

8.5.2 Usage de la transformée de Fourier

Deux opérations dans l'algorithme précédent sont coûteuses. Il s'agit du calcul des produits $e\pi_L$ et $\pi_L\pi_R$. Ces matrices ont toutes des degrés assez grands (proportionnels à b). Les produits peuvent donc être calculés avantageusement en utilisant la transformée de Fourier rapide (FFT).

On rappelle brièvement le mode d'opération de la FFT. Un exposé plus complet se trouve par exemple dans [vzGG99, chap. 8]. On s'intéresse à la multiplication de deux polynômes de degré N. Pour pouvoir utiliser la FFT dans sa version la plus simple, on a besoin de supposer que le corps de base K contienne des racines 2^d-èmes de l'unité, pour $2^d > 2N$. Il est possible de se placer dans une extension K' (au prix de l'augmentation du coût des opérations, notamment des multiplications). L'opération de multiplication s'effectue alors en trois étapes. Appelons P et Q les deux polynômes que l'on cherche à multiplier, et w une racine 2^d-ème de l'unité, pour $d = 2 + \lfloor \log N \rfloor$. Par log, on note le logarithme en base 2. On parle de FFT *d'ordre d*.

Algorithme **ComputePMatrixRec**
Entrée : Un b-contexte $E = (e, \Delta)$.
Sortie : La matrice $\pi_E^{(0,b)}$.

```
{
    if b eq 0 then return I_{m+n}; end if;
    if b eq 1 then return ComputePMatrix1(<e,Δ>); end if;
    c:=b div 2;

    E_L:=<e mod X^c,Δ>;                          /* Un ⌊b/2⌋-contexte */
    π_L:=ComputePMatrixRec(E_L,c);

    E_R:=<e * π_L mod X^b div X^c,Δ * π_L>;      /* Un ⌈b/2⌉-contexte */
    π_R:=ComputePMatrixRec(E_R,b-c);

    π:=π_L * π_R;
    return π;
}
```

Programme 8.4: Algorithme récursif pour calculer les matrices π.

– On commence par le calcul des transformées de Fourier discrètes (DFT) de chacune des entrées. Chacune de ces opérations nécessite $d2^{d-1} \approx \frac{1}{2}N \log N$ multiplications. La DFT du polynôme P s'écrit comme le 2^d-uplet :

$$\widehat{P} = (P(1), P(w), \ldots, P(w^{2^d-1})).$$

– La convolution, c'est-à-dire le produit terme à terme des deux DFTs $\widehat{P}$ et $\widehat{Q}$, est formée au prix de $2^d \approx N$ multiplications. Ce produit terme à terme est le 2^d-uplet :

$$\widehat{P}\widehat{Q} = (P(1)Q(1), P(w)Q(w), \ldots, P(w^{2^d-1})Q(w^{2^d-1})) = \widehat{PQ}.$$

– On calcule PQ à partir de $\widehat{PQ}$ par une opération semblable au calcul de la DFT : c'est une DFT inverse, ou IDFT, qui nécessite $d2^{d-1} \approx \frac{1}{2}N \log N$ multiplications. L'opération IDFT donne une réponse définie modulo $X^{2^d} - 1$ (cela tient bien sûr au choix de w), ce qui explique la contrainte $2^d > 2N$

Remarques

Ce schéma amène plusieurs remarques. Toutes sont pertinentes dans notre situation.

Connaissance partielle du résultat

Tout d'abord l'entier d, qui paramètre le nombre de points utilisés pour le calcul de la DFT, est essentiellement conditionné par le nombre de coefficients inconnus du résultat. En effet, d'une manière générale, s'il existe deux fractions rationnelles Φ et Ψ telles que le résultat R s'écrive $R = \Psi(X, \Phi(X, P, Q))$ où la complexité de l'évaluation des fractions rationnelles est indépendante de d, c'est le degré de $\Phi(X, P, Q)$ qui importe. On illustre cette idée par un exemple. Si l'on sait que le produit des polynômes P et Q vaut $1 + X^a S(X)$, on pose :

$$\Phi(X, P, Q) = \frac{PQ - 1}{X^a}, \qquad \Psi(X, S) = 1 + X^a S.$$

Il est aisé de remplacer l'étape de convolution par le calcul du 2^d-uplet suivant :

$$\Big(\Phi(1, P(1), Q(1)), \Phi(w, P(w), Q(w)), \ldots, \Phi(w^{2^d-1}, P(w^{2^d-1}), Q(w^{2^d-1}))\Big) = \widehat{S}.$$

Chacune des évaluations coûte un nombre constant de multiplications car les puissances de w sont précalculées (on peut considérer que la DFT de X^a est « à portée de la main »). Donc l'étape de convolution garde la même complexité linéaire en N. Il est ensuite suffisant de s'assurer que $2^d > \deg S$ pour avoir une détermination unique de S par la DFT inverse. Cette formalisation pourrait être poussée plus loin, mais l'exemple que nous donnons correspond à nos besoins.

Une autre manière de traiter le cas particulier que nous avons pris en exemple est d'utiliser les algorithmes de *middle-product* comme dans [HQZ03].

Adaptation au produit de matrices

Un point primordial est la façon dont ce procédé « évaluation-convolution-interpolation » s'adapte au cas du produit de matrices de polynômes. En effet, pour calculer le produit de deux matrices de taille $p \times p$ et de degré N par exemple, on doit :

- calculer toutes les DFTs : $p^2 N \log N$ multiplications.
- calculer les convolutions : $p^3 N$ multiplications.
- calculer les IDFTs : $\frac{1}{2} p^2 N \log N$ multiplications.

Ainsi, on découple les deux composantes importantes. Si l'on tente d'exprimer la complexité algébrique du produit de matrices, on arrive à $p^3 \mathsf{M}(N)$, et ici on fait mieux que spécifier $\mathsf{M}(N) = N \log N$. Le produit de matrices polynomiales est donc un exemple de contexte où l'emploi de la transformée de Fourier permet de modifier l'algorithme en profondeur.

Extensions du corps de base

On a parlé de description « simple » de la FFT, en requérant l'existence de racines 2^d-èmes de l'unité dans le corps de base. Si ce n'est pas le cas, on peut avoir à se placer sur une extension algébrique pour obtenir ces racines, ce qui est praticable lorsque le degré de cette extension est modéré. Dans ce cas, on peut remarquer que plusieurs DFTs peuvent être calculées en une. On illustre ce propos en montrant comment, dans le cas d'une extension $K' = K(\sqrt{-1})$ de degré 2, deux DFTs peuvent être regroupées. D'une manière générale, pour une extension de degré k, où k est une puissance de 2, on peut regrouper k DFTs en une. Pour notre exemple, notons $i = \sqrt{-1}$. Pour calculer simultanément $\widehat{P}$ et $\widehat{Q}$, on calcule la DFT de $P + iQ$. Comme la conjugaison complexe permute les racines de l'unité, on connaît pour chaque puissance w^j de w les valeurs de $(P + iQ)(w^j)$ et $(P + iQ)(\overline{w}^j)$. Il est trivial d'en déduire $P(w^j)$ et les autres valeurs qui nous intéressent.

Usage de la FFT entière

Par le procédé que l'on vient de décrire, on amoindrit la pénalité que représente le passage dans une extension algébrique. Toutefois, un gain nettement plus intéressant peut être fait en se plaçant tout simplement sur les entiers. L'algorithme de multiplication rapide de Schönhage-Strassen [SS71], décrit dans [vzGG99] et implanté par exemple par [Zim98], fonctionne selon le même schéma que celui que l'on vient d'exposer. Pour multiplier deux entiers de N bits, sa complexité est $O(N \log N \log\log N)$. La composante $\log\log N$ ne doit pas être vue comme un inconvénient de cette méthode, puisque log log est une fonction très faiblement croissante.

Bien au contraire, elle est en fait la marque de ce qui fait l'avantage de l'algorithme. Sans rentrer dans les détails de son fonctionnement, remarquons que dans la description faite au paragraphe précédent, l'arithmétique sur les coefficients du polynôme se fait à une taille figée. Et il se peut que ce ne soit ni une taille vraiment facile, ni une taille où une algorithmique rapide peut fonctionner avantageusement. Sur les entiers, l'algorithme de Schönhage-Strassen procède autrement. Il commence par découper un entier de N bits en $\sqrt{N}$ blocs de $\sqrt{N}$ bits. Les multiplications de ces blocs (multiplications entières) sont effectuées selon le même procédé (c'est-à-dire qu'au second niveau de la récursion on a des blocs de $\sqrt[4]{N}$ bits). Ce découpage « variable » est en fait bien plus avantageux (et est la cause du $\log\log N$).

Bien entendu, il n'est pas parfaitement clair que multiplier des polynômes sur des corps finis puisse se faire en ayant recours à la multiplication d'entiers. C'est une application de la technique dite *pack and pad*, attribuée à Kronecker. Pour simplifier, supposons que K est le corps premier $\mathbb{F}_\ell$(cette technique se généralise). On a un polynôme $P = \sum_{i=0}^{N} p_i X^i \in K[X]$, où l'on prend pour p_i l'unique représentant dans $[\![0 \dots \ell - 1]\!]$. Soit f l'entier $\lceil \log((N+1)\ell^2) \rceil$. Définissons un entier $\underline{P}$ de la manière suivante.

$$\underline{P} = \sum_{i=0}^{N} p_i 2^{fi}.$$

Si l'on compose de même $\underline{Q}$, on a :

$$\underline{P}\,\underline{Q} = \sum_{k=0}^{2N} \left(\sum_{i+j=k} p_i q_j \right) 2^{fk}.$$

Notre choix de f fait que les coefficients de cette somme restent bornés par 2^f. Par réduction modulo ℓ, on retrouve donc les termes $\sum_{i+j=k} p_i q_j$ qui sont les coefficients du produit des polynômes P et Q. Ainsi, la multiplication entière par FFT permet de multiplier des objets dans des structures variées[5]. Cette méthode de *pack and pad* est très efficace car la constitution de $\underline{P}$ peut se faire très rapidement. Bien sûr, les données sont grossies, mais seulement d'un facteur $2 + \frac{\log N}{\log \ell}$. Si ℓ est grand, ce n'est donc pas plus coûteux que le fait de prendre une extension algébrique lorsque c'est nécessaire (et dans les faits, c'est plus efficace). Si ℓ est très petit, d'autres techniques peuvent être utilisées [vzGG96].

Il est important de noter que l'usage de la FFT entière n'empêche pas de bénéficier de la remarque faite plus haut sur l'adaptation du schéma évaluation-convolution-interpolation au produit de matrices[6].

Adaptation à l'algorithme

Nous voulons examiner, dans l'algorithme 8.4, comment les produits $e\pi_L$ et $\pi_L\pi_R$ peuvent être effectués en utilisant la FFT. Nous resterons dans le cadre de la FFT polynomiale, une analyse similaire pouvant être menée si l'on utilise la FFT entière. Les dimensions des objets concernés sont données par la table 8.5 et exprimées en utilisant l'abréviation $\phi = \frac{m}{m+n}$. On néglige les termes d'influence minimale (les parties entières disparaissent, par exemple).

[5] Voire très variées. Cf [GG01] pour une multiplication de polynômes sur des séries sur une extension non ramifiée d'un corps p-adique, par FFT entière.

[6] Il est néanmoins regrettable qu'aucune librairie multiprécision ne fournisse un accès spécifique à ces trois fonctions distinctes. Il se peut toutefois que ce soit le cas de la librairie GMP à moyen terme (P. Zimmermann, communication privée, janvier 2003).

Matrice	Taille	degré
e	$m \times (m+n)$	b
π_L	$(m+n) \times (m+n)$	$\phi\frac{b}{2}$
π_R	$(m+n) \times (m+n)$	$\phi\frac{b}{2}$
$e\pi_L$	$m \times (m+n)$	$b + \phi\frac{b}{2}$, et $\equiv 0 \bmod X^{\lfloor \frac{b}{2} \rfloor}$
$\pi_L\pi_R$	$(m+n) \times (m+n)$	ϕb

Table 8.5 – Données du calcul récursif des matrices π

Données	Opération	Ordre	Complexité
$e \to \widehat{e}$	DFT	$\log((1+\phi)b)$	$\frac{1}{2}m(m+n)(1+\phi)b\log((1+\phi)b)$
$\pi_L \to \widehat{\pi_L}$	DFT	$\log((1+\phi)b)$	$\frac{1}{2}(m+n)^2(1+\phi)b\log((1+\phi)b)$
$\widehat{e_R} \to e_R$	IDFT	$\log((1+\phi)b)$	$\frac{1}{2}m(m+n)(1+\phi)b\log((1+\phi)b)$
$\pi_R \to \widehat{\pi_R}$	DFT	$\log(2\phi b)$	$\frac{1}{2}(m+n)^2 2\phi b\log(2\phi b)$
$\widehat{\pi} \to \pi$	IDFT	$\log(2\phi b)$	$\frac{1}{2}(m+n)^2 2\phi b\log(2\phi b)$

Table 8.6 – Ordres maximaux des DFTs pour le calcul récursif des matrices π

On commence par le cas le plus simple, celui du produit $\pi_L\pi_R$. Étant donné le degré du produit, on a besoin d'une FFT d'ordre d, ou $2^d > \phi b$, donc $d \geqslant \log(\phi b)$. Pour calculer la matrice $e_R = e\pi_L \operatorname{div} X^{\lfloor \frac{b}{2} \rfloor}$, il faut une FFT d'ordre d, où 2^d est strictement supérieur au nombre de coefficients inconnus du produit. Cela impose :

$$\begin{aligned} 2^d &> (b + \phi\frac{b}{2} - \frac{b}{2}) \\ &> (1+\phi)\frac{b}{2} \\ d &\geqslant \log\left((1+\phi)\frac{b}{2}\right). \end{aligned}$$

Pour obtenir des bornes *supérieures* sur les ordres des DFTs, on est obligé de multiplier par deux les bornes inférieures qui viennent d'être choisies. On peut résumer les ordres nécessaires pour les opérations de DFTs et IDFTs, ainsi que les nombres de multiplications qui en découlent. Les informations correspondantes sont consignées dans la table 8.6.

8.5.3 Complexité

Nous disposons maintenant de l'information nécessaire pour évaluer la complexité de l'algorithme récursif proposé pour le calcul de générateurs linéaires vectoriels. Cette complexité passe d'abord par le calcul du coût de chaque récursion. Notons $\mathsf{C}(b)$ le nombre de multi-

plications nécessaires pour calculer $\pi_E^{(0,b)}$ (ainsi, cette analyse est essentiellement pertinente lorsque le corps de base est grand). On démontre le résultat suivant.

Théorème 8.13. *Le nombre de multiplications requises par l'algorithme* ComputePMatrixRec *pour calculer la matrice* $\pi_E^{(0,b)}$ *est :*

$$\mathsf{C}(b) \leqslant 2\mathsf{C}\left(\frac{b}{2}\right) + \left(c(\phi)(m+n)^2 b \log b\right) + \left((3+\phi)m(m+n)^2 b\right) + O\left((m+n)^2 b\right),$$

où l'on définit $c(\phi) = \phi^2 + 3.5\phi + 0.5$.

Démonstration. Le coût en dehors des récursions est l'addition des complexités mentionnées dans les cinq lignes de la table 8.6, ainsi que du coût des convolutions. On néglige tous les termes dont la contribution rentre dans la composante $O((m+n)^2 b)$. Tout d'abord, le coût des DFTs et IDFTs est :

$$\begin{aligned}\text{DFTs} + \text{IDFTs} &= \frac{1}{2}\left((m+n)^2(2\phi+1)(1+\phi)b\log b\right) + \frac{1}{2}\left((m+n)^2 4\phi b \log b\right), \\ &= (m+n)^2 c(\phi) b \log b, \text{ comme annoncé.}\end{aligned}$$

Quant au coût des convolutions, il s'obtient de façon similaire, en lisant les tables 8.5 et 8.6.

$$\begin{aligned}\text{CONV} &= m(m+n)^2(1+\phi)b + (m+n)^3 2\phi b, \\ &= m(m+n)^2(3+\phi)b.\end{aligned}$$

L'équation de coût annoncé est donc correcte. ■

En utilisant ce théorème, on déduit la complexité du calcul d'un générateur linéaire vectoriel en fonction du paramètre d :

$$\mathsf{C}(L-s) = \mathsf{C}\left(\frac{m+n}{n}d\right) = c(\phi)\frac{(m+n)^3}{n} d \log^2 d + O\left(\frac{m}{n}(m+n)^3 d \log d\right).$$

Si l'on souhaite exprimer la complexité en fonction du degré du générateur linéaire calculé (appelons k ce degré), la proposition 8.7 nous donne $k \sim \frac{m}{n}d$, donc une complexité :

$$c(\phi)\frac{(m+n)^3}{m} k \log^2 k + O\left((m+n)^3 k \log d\right).$$

8.6 Performance de l'algorithme récursif

8.6.1 Implantation

Comme c'est souvent le cas avec les algorithmes ayant une structure récursive, il est préférable de ne pas descendre récursivement jusqu'aux sous-problèmes de taille minimale, car l'approche récursive engendre pour ces tailles un surcoût relativement important. C'est le cas de l'algorithme que nous avons présenté. En-dessous d'une certaine valeur du paramètre b qui mesure directement la taille de l'entrée, la performance est meilleure si l'on utilise l'algorithme ComputePMatrix1 de façon itérative, comme dans l'algorithme quadratique décrit en 8.4. Un tel algorithme, baptisé opportunément ComputePMatrix_k pour avancer de k étapes à partir d'un k-contexte, est résumé par les opérations suivantes (le k-contexte en entrée est noté $E = (e, \Delta)$).

K	L	m	n	Coppersmith	algorithme récursif	Threshold
$\mathbb{F}_{2^{127}-1}$	$1,000$	4	4	35s	36s	958
	$10,000$			1h01mn	14mn	
	$100,000$			$\approx$ 4d	6h10mn	
$\mathbb{F}_{2^{607}-1}$	$1,000$	4	4	112s	118s	923
	$10,000$			3h03mn	45mn	
	$100,000$			$\approx$ 12d	19h34mn	
	$242,304$			$\approx$ 75d	47h48mn	
$\mathbb{F}_{2^{607}-1}$	$10,000$	10	20	$\approx$ 5d	1h57mn	880
$\mathbb{F}_{2^{1279}-1}$	$1,000$	4	4	267s	292s	916
	$10,000$			7h15mn	1h50mn	
	$100,000$			$\approx$ 30d	47h38mn	

Table 8.7 – Temps de calcul de générateurs linéaires

– Poser $\pi = \mathrm{id}_{m+n}$.
– Pour i allant de 0 à $k-1$:
– Calculer P=ComputePMatrix1($[X^i]e\pi, \Delta$).
– Poser $\pi = \pi P$, $\Delta = \Delta P$.
– Retourner π.

Il est clair que les deux premières lignes du programme 8.4 peuvent alors être remplacées par la ligne :

```
if b le T then return ComputePMatrix_k(<e,Δ>,b) ; end if ;
```

La borne `T` (*threshold*) doit ici être choisie de manière optimale. Les mesures expérimentales qui suivent donnent la valeur de cette borne.

8.6.2 Mesures expérimentales

Notre intérêt pour le calcul de générateurs linéaires provient à l'origine de l'algorithme de Wiedemann par blocs. Nous avons mesuré les performances de notre calcul de générateurs linéaires sur diverses tailles de données, provenant toutes de notre implantation de cet algorithme. On a toujours choisi $\mathbb{F}_p$ pour le corps de base, en prenant pour p un nombre premier de Mersenne. Cela a l'avantage de rendre l'utilisation de la FFT possible dans une extension de degré 2. Toutefois, les quelques mesures qui ont été menées semblent montrer qu'une implantation reposant sur la FFT entière, décrite page 152, permettrait d'obtenir des performances meilleures. La table 8.7 consigne les différents temps obtenus par l'algorithme, ainsi que les valeurs des *thresholds* qui ont été mesurées (ces valeurs dépendent grandement de l'implantation, elles sont donc purement indicatives).

8.7 Influence sur l'algorithme de Wiedemann par blocs

8.7.1 Paramètres optimaux

Nous avons vu en 6.3.6 des éléments de départ pour l'évaluation du coût de l'algorithme de Wiedemann par blocs. Le chaînon manquant dans cette analyse était le coût du calcul

du générateur linéaire vectoriel, que nous avons justement déterminé dans ce chapitre, pour divers algorithmes. Nous rassemblons maintenant les différentes composantes.

Pour l'algorithme de Wiedemann par blocs, la matrice B sur laquelle nous nous concentrons est de taille $N \times N$, et compte en moyenne γ coefficients non nuls dans chaque ligne. Comme cela a été fait en 6.3.6, on va noter M_1 le coût d'une multiplication de deux éléments de $\mathbb{F}_p$ et détailler uniquement dans le cas de $\mathbb{F}_p$ quelles sont les complexités finales obtenues en fonction de m et n. On en déduit les valeurs optimales de ces deux paramètres. Le calcul de générateur linéaire intervient au cœur de l'algorithme de Wiedemann par blocs, comme la phase que l'on a baptisée BW2. Le calcul exact que l'on doit mener est celui d'un générateur linéaire vectoriel pour une suite de matrices $m \times n$, où les paramètres s et d sont donnés par les arguments qui ont été développés en 6.3.2. Ces valeurs sont respectivement $s = \lceil \frac{m}{n} \rceil$, et $d = \lceil \frac{N}{m} \rceil$.

Proposition 8.14. *Dans l'algorithme de Wiedemann par blocs, le coût de l'étape* BW2 *est donné par l'une des formules suivantes (on continue à noter $\phi = \frac{m}{m+n}$) :*

- $\mathsf{M}_1 \frac{m+n}{2} N^2 + O(N)$ *en utilisant l'algorithme de Coppersmith (cf 8.4.6).*
- $c(\phi)\mathsf{M}_1 \frac{(m+n)^3}{mn} N \log^2 N + O(N \log N)$ *en utilisant notre algorithme (cf 8.5), sous l'hypothèse que m et n sont en $O(\log N)$.*

Ces expressions découlent des présentations qui ont été faites des algorithmes en 8.4 et 8.5. Notons qu'à l'inverse des complexités qui ont été données pour les étapes BW1 et BW3 en 6.3.6, nous ne parlons pas ici de complexité parallèle, puisqu'il n'a pas été question de paralléliser aucun des algorithmes mentionnés ici.

Nous déduisons maintenant des formules pour les valeurs optimales de m et n. Nous supposons pour cela que n machines sont toujours disponibles, de telle sorte que les étapes BW1 et BW3 peuvent être effectuées en parallèle. Rappelons les expressions qui ont été données en 6.3.6 pour les complexités parallèles des étapes BW1 et BW3 :

$$\mathsf{BW1} : \gamma \mathsf{M}_0 \frac{m+n}{mn} N^2. \qquad \mathsf{BW3} : \gamma \mathsf{M}_0 \frac{1}{n} N^2.$$

Théorème 8.15. *Si l'algorithme de Coppersmith (cf 8.4) est employé pour l'étape* BW2*, alors le temps total pour l'exécution de l'algorithme de Wiedemann par blocs est minimal pour*

$$n_{\text{opt}} = 2\sqrt{\frac{\gamma \mathsf{M}_0}{\mathsf{M}_1}}, \quad \textit{et} \quad m_{\text{opt}} = 0.7 n_{\text{opt}}.$$

Dans ce cas, le temps total est

$$\mathsf{W}_{\text{opt}} = \mathsf{BW1} + \mathsf{BW2} + \mathsf{BW3} = 3.4\sqrt{\gamma \mathsf{M}_0 \mathsf{M}_1} N^2.$$

Démonstration. Nous avons tous les éléments pour calculer l'expression $\mathsf{W} = \mathsf{BW1} + \mathsf{BW2} + \mathsf{BW3}$. On obtient :

$$\mathsf{W} = \gamma \mathsf{M}_0 \frac{m+n}{mn} N^2 + \mathsf{M}_1 \frac{m+n}{2} N^2 + \gamma \mathsf{M}_0 \frac{1}{n} N^2,$$
$$\mathsf{W} = \left(\gamma \mathsf{M}_0 (1 + \frac{1}{\phi}) \frac{1}{n} + \mathsf{M}_1 \frac{n}{2(1-\phi)} \right) N^2.$$

K	N	m, n	BW1	Coppersmith	algo. récursif	BW3	Threshold
$\mathbb{F}_{32479}$	$10,000$	2	4h01mn	1h12mn		1h57mn	
	($\gamma \sim 35$)	4	2h02mn	2h02mn		1h04mn	
		8	1h05mn	4h06mn		34mn	
	$20,000$	2	29h05mn	4h38mn		14h30mn	
	($\gamma \sim 65$)	4	14h44mn	8h15mn		7h17mn	
		8	8h07mn	16h29mn		3h48mn	
$\mathbb{F}_{65537}$	$10,000$	2	1h16mn	52mn	3mn50s	38mn	147
	($\gamma \sim 35$)	4	38mn	1h27mn	7mn47s	19mn	132
		8	19mn	2h20mn	18mn56s	10mn	74
	$20,000$	2	8h58mn	3h07mn	8mn45s	4h31mn	161
	($\gamma \sim 65$)	4	4h41mn	5h10mn	18mn32s	2h22mn	132
		8	2h19mn	9h12mn	52mn01s	1h10mn	80

Table 8.8 – Comparaison avec les résultats de [Lob95]

Si l'on cherche à minimiser W pour une valeur donnée de ϕ, les valeur optimales W_{opt} et n_{opt} sont :

$$n_{\text{opt}} = \sqrt{\frac{\gamma \mathsf{M}_0 2(\phi+1)(1-\phi)}{\mathsf{M}_1 \phi}},$$
$$\mathsf{W}_{\text{opt}} = 2N^2 \sqrt{\gamma \mathsf{M}_0 \mathsf{M}_1 \frac{\phi+1}{2\phi(1-\phi)}}.$$

Nous devons donc minimiser $\frac{\phi+1}{\phi(1-\phi)}$, ce qui impose $\phi = \sqrt{2}-1$. En incorporant cette valeur de ϕ, on obtient les valeurs annoncées. ■

Théorème 8.16. *Si l'algorithme décrit en 8.5 est employé pour l'étape* BW2*, alors le temps total pour l'exécution de l'algorithme de Wiedemann par blocs est minimal pour*

$$n_{\text{opt}} = 0.6\sqrt{\frac{\gamma \mathsf{M}_0 N}{\mathsf{M}_1 \log^2 N}}, \quad \textit{et} \quad m_{\text{opt}} = 0.5 n_{\text{opt}}.$$

Dans ce cas, le temps total est

$$W_{\text{opt}} = 13.8\sqrt{\gamma \mathsf{M}_0 \mathsf{M}_1} N \sqrt{N} \log N.$$

Démonstration. Le raisonnement est identique. On écrit W :

$$\mathsf{W} = \gamma \mathsf{M}_0 (\phi + \frac{1}{\phi}) \frac{1}{n} N^2 + c(\phi) \mathsf{M}_1 \frac{1}{\phi(1-\phi)^2} n N \log^2 N.$$

Ensuite, on obtient le minimum pour $\phi \approx 0.3$, ce qui correspond aux valeurs annoncées. ■

8.7.2 Comparaison avec d'autres implantations

On trouve peu d'expériences menées avec l'algorithme de Wiedemann par blocs dans la littérature. Les seuls résultats d'expériences sur des corps autres que $\mathbb{F}_2$ dont nous ayons la connaissance sont ceux de la thèse de Lobo [Lob95]. Nous avons tenté de reproduire les conditions d'expérience pour voir comment les implémentations se comparaient. Les microprocesseurs utilisés par Lobo étaient de type sparc à 107MHz. Nous avons déniché quelques microprocesseurs sparc à 143MHz, à partir desquels on a mené la comparaison. Dans le tableau 8.8, les temps de calcul sur le corps de base $\mathbb{F}_{32479}$ sont ceux de [Lob95], et nos temps de calculs sont ceux obtenus sur $\mathbb{F}_{65537}$.

À la lecture du tableau 8.8, on constate que le rapport des temps de calculs correspond à celui des fréquences des microprocesseurs pour le cas de l'étape BW2, mais qu'il nous est en revanche très favorable pour les étapes BW1 et BW3. On peut avancer plusieurs explications pour cette disparité, qui est assez surprenante. Bien sûr, les implantations sont totalement différentes, et cette différence peut induire un écart important. En ce qui concerne les phases BW1 et BW3, on peut noter aussi que l'influence de la mémoire cache est très importante, et nous ne disposons pas de caractéristiques de ce niveau de précision concernant le matériel utilisé dans [Lob95]. En dernier lieu, il n'est pas exclu qu'une erreur se soit glissée dans la mention de la densité γ concernant les expériences de [Lob95].

Chapitre 9

Algèbre linéaire « extrême »

Nous décrivons dans ce chapitre les données de la résolution de système linéaire qui est intervenue dans notre calcul de logarithmes discrets dans $\mathbb{F}_{2^{607}}$. Ce système est de grande taille : 1 077 513 équations, 766 150 inconnues, sur le corps de base $\mathbb{Z}/(2^{607}-1)\mathbb{Z}$. Nous avons utilisé pour le résoudre l'algorithme de Wiedemann par blocs.

9.1 Élimination structurée

Le premier traitement que nous avons appliqué à notre système linéaire est une première passe d'élimination structurée. Comme cela a été décrit en 5.2, nous avons tiré parti à cet effet des lignes surnuméraires dont nous disposions, pour pouvoir *éjecter* au fur et à mesure de l'algorithme les lignes devenant trop lourdes. Comme l'excès de lignes sur le système initial s'élevait à 40%, notre marge de manœuvre était confortable.

Le nombre initial de coefficients non nuls parmi les 766 150 lignes les moins lourdes (c'est cette grandeur qu'il faut retenir) s'élevait à 50 millions environ, soit un nombre moyen de coefficients non nuls égal à 67.7. Nous avons suivi les étapes du processus d'élimination structurée, telles qu'elles ont été décrites en 5.2.2. Le point d'arrêt de l'élimination structurée a été chois de façon à minimiser les formules développées en 8.7 pour le coût de l'algorithme de Wiedemann par blocs.

À l'issue de l'élimination structurée, qui n'a nécessité que quelques heures de calcul sur une machine de type alpha ev67 à 667MHz, nous avons obtenu un système plus petit que le système d'origine, de taille $480\,108 \times 480\,108$. L'élimination structurée a bien fait son travail, puisqu'elle a permis cette réduction sans augmentation du nombre de coefficients : ce nombre est resté à 50 millions, correspondant à une densité moyenne des lignes de 104.8 coefficients.

9.2 Calcul de la suite $A(X)$

Il a fallu ensuite se lancer dans l'algorithme de Wiedemann par blocs, le premier choix à faire étant bien entendu celui des paramètres m et n. Nous avons choisi $m = n = 8$. On peut remarquer que ce choix n'est pas en accord avec les formules données en 8.7 pour les valeurs optimales. Nous avons choisi $n = 8$ car nous pensions avoir accès à 8 machines pour nos calculs. Il ne s'agissait là que d'une estimation, car les machines auxquelles nous avons eu accès étaient de nature très hétérogène, et leur accessibilité n'était pas une donnée certaine (les machines du *cluster* MEDICIS, en particulier, sont généralement très chargées). La valeur de $m = 8$ a ensuite été choisie en conséquence, car nos mesures semblaient indiquer qu'en toute état de cause, l'étape BW2 ne serait pas limitante (ce qui a été confirmé).

En fonction de ces choix, nous avons ensuite dû « préparer » le calcul. Cela a impliqué deux manipulations. Premièrement, on a remplacé la matrice B par une matrice ΣB, Σ étant une matrice de permutation, dans le but d'équilibrer les lignes de la matrice, en vue d'un

usage sur des machines multiprocesseurs. Le procédé employé pour construire ΣB a été décrit brièvement en 7.2.4. Par ailleurs, nous avons mentionné en 7.3 que pour être en mesure de détecter les erreurs pouvant intervenir lors du produit matrice $\times$ vecteur, nous devions construire deux vecteurs α et β reliés par $\beta = B^T\alpha$. C'est à ce stade du calcul que α et β sont construits. Notre matrice ayant des coefficients très petits, il nous a été facile d'obtenir de petits vecteurs α et β.

Le nombre de produits matrice $\times$ vecteur à effectuer dans la phase BW1 s'élève à 120 032 pour chacune des colonnes de $A(X)$. Des machines très diverses ont été utilisées pour ce calcul. Ces machines ont été mises à disposition par les institutions suivantes.

- LIX, École polytechnique.
- MEDICIS, École polytechnique.
- Compaq Computer Corporation (Testdrive program), Boston, USA.

La machine « moyenne » que nous avons utilisée pour la phase BW1 est celle qui a le plus contribué aux calculs (il s'agit d'une machine du LIX), de type Compaq DS20E, à 2 microprocesseurs alpha ev67 à 667MHz. Sur cette machine, le temps d'un produit matrice $\times$ vecteur est de 15 secondes. Si l'on exprime le temps de la phase BW1 en fonction de cette donnée, sur un parc fictif de 8 machines identiques, on arrive donc à un temps de calcul de 20 jours. Le temps de calcul réel de la phase BW1 a été un peu supérieur à un mois, en raison des charges importantes des *autres* machines qui ont pris part au calcul.

9.3 Obtention du générateur linéaire

Le calcul de générateur linéaire a été effectué sur la machine alpha DS20E du LIX, dont on vient de parler. Ce calcul a demandé six jours de travail, en n'utilisant qu'un seul des deux processeurs[1], en utilisant l'algorithme sous-quadratique que nous avons développé et présenté dans le chapitre 8. Le degré du générateur linéaire que l'on a ainsi obtenu est de 60 013.

La structure récursive de l'algorithme sous-quadratique du chapitre 8 a fonctionné à plein régime. En effet, le *threshold* entre l'algorithme quadratique et l'algorithme récursif se situe au degré 600. Le degré de $A(X)$ étant 120 032, on comprend que le gain représenté par l'utilisation de l'algorithme récursif est énorme. Neuf niveaux de récursion ont été ainsi utilisés. Le niveau inférieur représentant tous les calculs de matrices $\pi_E^{(0,b)}$ (cf 8.5) pour $b = \frac{120\,032}{2^8} = 468$ (par la méthode quadratique). Additionnés, ces calcul « en-bas » de l'arbre de récursion représentent un quart du temps de calcul final. Les huit niveaux récursifs « au-dessus » se partagent presque à parts égales le temps restant. Pour être exact, le niveau récursif le plus bas représente 11% du temps de calcul restant, contre 14% pour le niveau récursif au sommet de l'arbre.

La consommation en mémoire de l'algorithme récursif s'est avérée importante, car les transformées de Fourier ont dû être stockées à l'avance (l'implantation exacte du programme 8.4 détruit la matrice e). Cela n'a toutefois pas posé de problème, car il a suffi d'ajouter à la machine la quantité adéquate de mémoire virtuelle (20Go tout de même), gérée sans aucune difficulté par le système (sur une machine ayant 4Go de mémoire physique réelle) : jamais l'échange de données avec le disque dur n'a limité les performances du programme.

[1] On aurait aussi pu tirer parti des fonctionnalités multiprocesseur pour accélérer le calcul, mais cela n'est pas apparu nécessaire.

9.4 Obtention d'un vecteur du noyau

La dernière étape, BW3, est très semblable à la première. Elle a été décrite en 6.3.3. Pour cette étape, on a utilisé une modification du programme utilisé pour l'étape BW1, ce qui nous a permis de bénéficier à nouveau des fonctionnalités multiprocesseur des machines. Pour accélérer encore ce calcul, nous avons conservé quelques-uns des vecteurs $B^i y$ calculés lors de la phase BW1 : ces vecteurs ont pu être réutilisés pour calculer des portions de l'expression donnée en 6.3.3 pour le vecteur du noyau :

$$v = \sum_{i=0}^{\deg F} B^{\deg F - i} z F_i.$$

Pour finir les calculs, nous avons eu accès aux machines du centre de calcul IDRIS, à Orsay. Plus exactement, l'installation que nous avons utilisée est un *cluster* de six machines de type Compaq ES40, équipées de 4 processeurs ev67 à 833MHz. En utilisant ces machines, nous avons pu achever le calcul en six jours.

9.5 Obstacles rencontrés : technique et sociologie

9.5.1 Mise en place d'un calcul d'algèbre linéaire, parallèle et distribué

Pour la résolution de systèmes linéaires par l'algorithme de Wiedemann par blocs, on a utilisé une approche partiellement distribuée. En comparaison avec la distribution « à grande échelle » que nous avons dû effectuer pour la recherche de relations, l'approche ici était plus simple par deux aspects. Tout d'abord, le nombre d'esclaves à gérer est beaucoup plus faible (au maximum, on a utilisé huit machines simultanément), et surtout ces machines n'entretiennent aucune communication une fois passé le stade d'initialisation.

Nous avons présenté au chapitre 7 notre implantation *multithread* de la multiplication matrice × vecteur. Cela a représenté un travail de développement important, car la mise au point du programme 7.2 page 123 n'est pas une entreprise facile. Nous avons rencontré, au cours de ce travail, diverses difficultés, les premières d'entre elles étant toutes les difficultés inhérentes à la programmation *multithread*.

Comme cela a été mentionné en 9.2, le nombre d'itérations effectuées dans la phase BW1 de l'algorithme de Wiedemann par blocs a dépassé la centaine de milliers. La matrice B occupant 400Mo de mémoire vive, cela implique qu'une grande quantité de donnée transite par les microprocesseurs prenant part au calcul, dans un contexte où les machines utilisées le sont « à plein régime ». Une conséquence de cette lourde charge imposée sur les machines a été l'apparition d'erreurs de calcul, qui ont justifié le processus de vérification décrit en 7.3. Cette méthode a servi de diagnostic de qualité de barettes mémoire pour certaines machines du *cluster* Medicis.

Par ailleurs, la forte sollicitation du matériel que représente un calcul *multithread* comme celui que nous avons mené a été la source de nombreux « plantages », tantôt par manque de fiabilité du matériel, tantôt aussi par effroi de la personne en charge de l'adiminstration des machines en question. Pour ces raisons, on a du prendre comme hypothèse de départ que les processus lancés pouvaient être stoppés à tout moment, et méritaient une certaine surveillance. En cela, on a constaté que la distribution de tâches lourdes comme celles-ci était plus dure à mener que la distribution de tâches « légères » comme les programmes de recherche de relations pour le calcul de logarithmes discrets, évoqués au chapitre 4.

Enfin, le faible coût en communications impliqué par l'emploi de l'algorithme de Wiedemann par blocs s'est montré indispensable. En effet, même maigres, les communications ont représenté un point d'achoppement : le calcul d'algèbre linéaire a été mené sur différents sites, sans connexion réseau privilégiée. Nous avons donc dû faire transiter des fichiers par le réseau, provoquant une charge importante sur cette infrastructure. C'est ainsi que l'on a saturé la liaison extérieure de l'École polytechnique. Une quantité de communication plus grande aurait à coup sûr rendu les problèmes de ce type encore plus aigus.

9.5.2 Mode d'emploi ou de non-emploi d'un centre de calcul

Au cours des calculs que nous avons menés, plusieurs types de ressources informatiques ont été mises à contribution. Le cas dont la gestion est de loin la plus simple est celui des machines du laboratoire LIX (une machine alpha DS20E ev67 à 667MHz, et une grappe de calcul de 13 PCs Pentium II à 450MHz). En effet, avoir le contrôle total de ces machines, ainsi que la quasi-exclusivité de leur accès, nous a permis d'en tirer un très bon niveau de performance. Ainsi, les 13 « petits » PCs de la grappe de calcul du laboratoire ont contribué très largement à l'effort de crible, car ils y ont travaillé « à 100% » pendant l'essentiel de la durée du calcul. Du strict point de vue de la puissance brute, aussi bien que du nombre de machines, cette ressource n'est pourtant pas la plus puissante de celles auxquelles nous avons eu accès. Mais comme nous en avons fait mention, la « gestion » des utilisateurs a parfois rendu le calcul difficile le cas échéant.

Des ressources plus « organisées » ont aussi été utilisées, comme le centre de calcul Medicis à l'École polytechnique, le programme *TestDrive* proposé par Compaq (désormais HP), ainsi que le centre de calcul IDRIS du CNRS, à Orsay. L'emploi de ces centres de calcul a signifié quelques contraintes. Plusieurs contorsions ont été nécessaires pour se défaire de ces contraintes (et ainsi pouvoir travailler).

Nous illustrons ces contorsions par un exemple détaillé, relatant comment nous avons pu nous affranchir d'une restriction rencontrée sur le nombre de processeurs simultanément utilisés par une tâche. Comment une telle restriction peut-elle être mise en place ? Cela paraît peu évident. En effet, les systèmes Unix offrent la possibilité de limiter l'usage de certaines ressources par un programme au travers de l'interface `setrlimit`/`getrlimit`. Ces ressources sont par exemple le temps maximal d'exécution d'un programme, ou son utilisation mémoire maximale. Il n'est fait aucune mention dans les systèmes Unix actuels d'une limitation possible du nombre de processeurs simultanément utilisés. En outre, une telle limitation nécessiterait une définition très précise : veut-on limiter le nombre de programmes travaillant de manière distribuée, veut-on limiter le nombre de *threads* utilisés, ou les deux ?

Connaissant ces difficultés qui empêchent l'établissement radical d'une telle limite on a supposé (et la supposition était juste) qu'elle était mise en place de manière bien plus « artisanale ». Les programmes étant lancés via un programme de soumission (c'est-à-dire en *batch*), c'est ce programme de soumission qui tout simplement compte le nombre de ses fils, au sens de la hiérarchie des processus Unix. Cette approche est totalement inadaptée à l'emploi de *threads* : limiter par exemple à quatre le nombre de processeurs « simultanément utilisés » revient à interdire l'usage de plus de deux *threads*. En effet, l'implantation sous Linux des *threads* POSIX crée un processus par *thread*[2], et deux autres processus viennent s'ajouter

[2]On traite ici de l'implantation LinuxThreads dûe à X. Leroy. Cette implantation a été la première à être mise en place pour les systèmes Linux. Elle dévie marginalement de la spécification POSIX précisément en ce point, puisque plusieurs processus sont créés pour les différents *threads*. Cette implantation semble être en

pour la gestion des communications entre les threads. Cette approche impose donc une limite bien trop forte. Elle est en outre assez proche de l'inutile puisqu'on peut éliminer la restriction aisément. Il suffit de « faire croire » au programme de soumission que la tâche finit immédiatement, en ayant entre temps essaimé quelques processus fils qui se seront *détachés* de leur processus père normal, ceci en utilisant l'appel système `setsid` : on crée ainsi des processus « fils de `init` ». Le petit programme 9.1, écrit en `perl`, sert ce but.

```
#!/usr/bin/perl -w
use strict;
use POSIX 'setsid';

# Usage: ./daemon.pl <running_dir> <output_file> <program> [<arguments>...]

$ENV{PATH} = '/bin:/usr/bin';

my $dir=shift(@ARGV);
my $output=shift(@ARGV);
chdir "$dir"                or die "Can't chdir to $dir: $!";

defined(my $pid = fork)     or die "Can't fork: $!";
exit if $pid;

open STDOUT, ">$output"     or die "Can't write to $output: $!";
setsid()                    or die "Can't detach: $!";
open STDERR, ">&STDOUT"     or die "Can't dup stdout: $!";
open STDIN, "</dev/null"    or die "Can't dup stdin: $!";

exec @ARGV;
```

Programme 9.1: Le programme `daemon.pl`

Nous avons rencontré d'autres restrictions, comme par exemple des restrictions sur le temps de calcul maximal. C'est là qu'on apprend qu'un programme n'a « pas le droit » de travailler plus de dix heures[3]. Notre programme résistant aisément à ce genre d'interruption, il a fallu, pour une semaine de calcul, mettre en séquence une quinzaine d'invocations de notre programme, à l'intérieur d'un *script* shell.

De telles contorsions nous ont toujours semblé contre-productives. Elles atteignent rarement le but souhaité, mais elles rendent l'utilisation des ressources de calcul ardue, et c'est regrettable. Dans cet esprit, nous ne dirons jamais assez combien la puissance du *cluster* `vedia` du centre de calcul IDRIS nous paraît séduisante : six machines quadri-processeur alpha EV67 cadencés à 833MHz. Une telle installation est bien adaptée à l'exécution d'un programme comme le nôtre. Hélas, pour obtenir le résultat final, nous avons du faire face à des restrictions comme celles que nous venons d'évoquer, si bel et si bien que notre programme

cours de remplacement au sein de la librairie standard C `glibc` par l'implantation NPTL (*Native Posix Threads Library*), dûe à U. Drepper. L'implantation NPTL semble obéir strictement à la norme, et les développements que nous mentionnons ici ne s'y appliquent pas nécessairement.

[3]Et surtout, un programme tournant aussi longtemps n'a le droit d'« utiliser », au sens vu plus haut, qu'un seul processeur ! Grâce à notre lanceur en `perl`, on satisfait heureusement cette contrainte.

a constitué l'*unique* utilisation de ce cluster sur la période s'étalant de janvier à avril 2002 (compte-rendu du comité des utilisateurs de l'IDRIS du 24 avril 2002).

Annexes

Annexe A

Rappels sur les corps finis

Nous rappelons dans ce chapitre les éléments essentiels de la théorie des corps finis, dont on a fait implicitement usage dans une bonne partie de ce mémoire (notamment la première partie). Ces rappels sont allusifs. Pour une présentation plus complète, on pourra consulter les ouvrages de référence sur le sujet tels que [LN83] ou [Ber68].

A.1 Caractéristique, cardinal

Les premiers pas de l'établissement de la théorie concernent la détermination des cardinaux possibles des corps finis.

Définition A.1 (Caractéristique). *On appelle* caractéristique *d'un anneau R l'entier c tel que $c\mathbb{Z}$ soit le noyau du morphisme de groupes additifs suivant :*

$$\kappa : \left\{ \begin{array}{ccc} \mathbb{Z} & \longrightarrow & R \\ n & \longmapsto & \underbrace{1_R + \cdots + 1_R}_{n \text{ fois}} \end{array} \right.$$

Si l'anneau R est fini, c est nécessairement non-nul. On montre très aisément que si R est intègre, c est nécessairement un nombre premier. Ces deux remarques s'appliquent bien entendu aux corps finis. On obtient donc la propriété suivante :

Proposition A.2. *Un corps fini K a pour caractéristique un nombre premier p. Il contient en conséquence un sous-corps isomorphe à $\mathbb{F}_p \overset{déf}{=} \mathbb{Z}/p\mathbb{Z}$. Le corps K peut être vu comme un espace vectoriel sur $\mathbb{F}_p$. Son cardinal est une puissance de p.*

Démonstration. La première assertion découle de la factorisation du morphisme κ, et de la définition d'un sous-corps. Le sous-corps $\mathbb{F}_p$ ainsi mis en évidence est appelé *sous-corps premier* de K. Le corps K est d'ailleurs appelé *premier* s'il est égal à son sous-corps premier. Ensuite, la propriété du cardinal est une conséquence. ■

On peut aussi étudier les cardinaux des extensions de corps finis.

Notation A.3. *Soit L une extension finie d'un corps K. On note $[L : K]$ la dimension de L en tant que K-espace vectoriel.*

Proposition A.4. *Soit K–L–M une tour d'extensions finies. On a :*

$$[M : K] = [M : L][L : K].$$

Démonstration. On construit une base *ad hoc* de M en tant que K-espace vectoriel, pour arriver à l'égalité. ■

Cette propriété implique que pour deux corps finis K et L tels que $K \subset L$, si le cardinal de L s'écrit p^n, alors celui de K s'écrit p^d, avec d divisant n.

Dans ce qui précède, on n'a pas fait d'hypothèse sur la commutativité de K. La structure d'espace vectoriel sur $\mathbb{F}_p$ peut donc être non commutative. Néanmoins, cela n'est que théorique, puisqu'un théorème dû à Wedderburn montre que seul le cas commutatif mérite d'être traité.

Théorème A.5 (Wedderburn). *Tout corps fini est commutatif.*

La preuve de ce théorème n'est pas reproduite ici.

On a désormais un ensemble de conditions nécessaires concernant les corps finis. On va montrer maintenant que pour chaque puissance d'un nombre premier de la forme $q = p^n$, on peut construire un corps de cardinal q.

A.2 Construction des corps finis

Un corps de cardinal premier étant égal à son sous-corps premier, il s'ensuit que tous les corps de cardinal premier p sont isomorphes à $\mathbb{F}_p$. Pour construire des corps *non* premiers, on a recours au procédé suivant.

Proposition A.6. *Soit p un nombre premier. Soit P un polynôme irréductible de $\mathbb{F}_p[X]$. Alors l'anneau quotient $K = \mathbb{F}_p[X]/(P)$ peut être muni d'une structure de corps commutatif, de cardinal $p^{\deg P}$.*

Démonstration. Nous devons vérifier qu'il est possible d'inverser les éléments non nuls de K. Soit donc A un polynôme non multiple de P. Le polynôme P étant irréductible, alors A et P sont premiers entre eux. On peut donc écrire une relation de Bézout entre A et P, qui donne deux polynômes U et V tels que $AU + PV = 1$. Alors, la classe de U modulo P est un inverse de A.

Pour obtenir le cardinal de K, on remarque qu'une base de K sur $\mathbb{F}_p$ en tant qu'espace vectoriel est $(1, X, X^2, \ldots, X^{\deg P - 1})$. ■

On montrera plus loin que tous les corps finis se construisent ainsi à isomorphisme près, et que la structure d'un corps dépend en fait uniquement de son cardinal. On se permettra alors de faire référence à $\mathbb{F}_q$ comme étant l'unique structure de corps fini de cardinal q, à isomorphisme près.

A.3 Le groupe multiplicatif

Nous arrivons à la propriété essentielle de structure des groupes multiplicatifs des corps finis. Cette propriété est vraie en toute généralité, même en dehors du cadre des corps finis ou commutatifs.

Proposition A.7. *Tout sous-groupe fini du groupe multiplicatif d'un corps est cyclique.*

Démonstration. Soit K le corps considéré, soit G un sous-groupe fini de K^*, et n son cardinal. Pour tout $x \in G$, $x^n = 1$. Donc un élément de G a nécessairement un ordre qui

divise n. On va classer les éléments de G selon leur ordre. Soit $f(d)$ le nombre d'éléments de G ayant pour ordre exactement d, pour d un diviseur quelconque de n.

L'étude du groupe fini $\mathbb{Z}/m\mathbb{Z}$ amène la propriété suivante, faisant intervenir l'indicatrice d'Euler :

$$\sum_{d|n} \varphi(d) = n.$$

De plus, par construction de $f(d)$, on peut écrire :

$$\sum_{d|n} f(d) = n,$$

$$\text{d'où :} \qquad \sum_{d|n} \varphi(d) - f(d) = 0.$$

Soit d un diviseur de n tel que $f(d) \neq 0$. Il existe donc un élément α de K^* d'ordre exactement d. Par conséquent, les éléments $1, \alpha, \ldots, \alpha^{d-1}$ sont distinctes et constituent les d racines du polynôme $X^d - 1$ dans K (ce sont nécessairement les seules car K est un corps). Les éléments d'ordre d dans K sont donc les puissances de α d'ordre d, c'est-à-dire les α^k avec $(k, d) = 1$. On a donc $f(d) = \varphi(d)$ dès que $f(d) \neq 0$.

Comme les termes de la somme $\sum_{d|n} \varphi(d) - f(d)$ sont tous positifs, on en déduit que $\forall d \mid n,\ f(d) = \phi(d)$. Comme $\varphi(n)$ est non nul, il existe un élément g, d'ordre n dans G. Cela revient à dire que G est cyclique, engendré par g. ■

Pour un corps fini, une autre façon d'énoncer le résultat qui précède est de dire que K^* est isomorphe à $\mathbb{Z}/(q-1)\mathbb{Z}$. Rendre explicite cet isomorphisme revient, étant donné le choix d'un générateur g, à savoir donner pour un élément $x \in K^*$, un entier k tel que $g^k = x$. L'entier k est le logarithme de x en base g. Calculer de tels logarithmes est le problème, souvent difficile, sur lequel porte la première partie de ce mémoire.

A.4 Propriétés des corps finis

Une propriété incontournable des corps finis est l'existence de l'automorphisme de Frobenius :

Proposition A.8.

1. *Soit K un corps fini de cardinal $q = p^d$. Soit $x \in K$. Alors $x^q = x$.*
2. *Soit L une extension de degré n de K. L'application $\sigma : \left\{ \begin{array}{ccc} L & \longrightarrow & L \\ x & \longmapsto & x^q \end{array} \right.$ est un automorphisme de corps fixant K. De plus, $\sigma^n = \mathrm{id}_L$.*

Démonstration. La première assertion est triviale pour $x = 0$. Pour un x non nul, donc élément du groupe K^* des éléments inversibles de K, on a $x^{\#K^*} = 1$. Comme $\#K^* = q - 1$, on déduit le résultat en multipliant par x.

Pour la seconde assertion, comme $p = 0$ dans K, on utilise le développement par la formule du binôme de l'expression $\sigma(a + b) = (a + b)^q$. Comme p divise $\binom{q}{i}$ pour $0 < i < q$, on a l'additivité de σ. De plus $\sigma(ab) = \sigma(a)\sigma(b)$, et au vu de la première assertion, $k \in K$ implique $\sigma(k) = k$. Comme σ est injectif (c'est un morphisme de corps non nul), c'est un automorphisme. Enfin, en appliquant toujours la première assertion à L, on a $\sigma^n = \mathrm{id}_L$. ■

Cette propriété nous suffit pour montrer que deux corps finis de même cardinal sont nécessairement isomorphes.

Lemme A.9. *Soit K une extension de degré n d'un corps F de cardinal q. Alors il existe un polynôme P tel que K est isomorphe à $F[X]/(P)$.*

Ce lemme montre que la construction des corps finis par extension polynomiale est générique.

Démonstration. Soit g un générateur du groupe multiplicatif K^*. Le corps K est alors égal à $F(g)$ qui est isomorphe à $F[X]/(P)$, où P est le polynôme minimal de g sur F. ■

Théorème A.10. *Soient K et L deux extensions finies de degré n d'un même corps F de cardinal q. Alors K et L sont isomorphes.*

Démonstration. Appliquons le lemme. Soient P_K et P_L deux polynômes tels que K et L soient respectivement isomorphes aux quotients $F[X]/(P_K)$ et $F[X]/(P_L)$. Tous les éléments de K et L vérifient $x^{q^n} = x$. Les polynômes P_K et P_L sont donc des diviseurs de $X^{q^n} - X$ (on applique cette identité à la classe de X dans l'anneau quotient). En outre, comme les éléments de K sont en nombre q^n, le polynôme $X^{q^n} - X$ est scindé dans K (il a q^n racines distinctes). En particulier, le polynôme P_L qui en est un diviseur a une racine dans K. Soit h une telle racine. Le polynôme P_L étant irréductible, P_L est le polynôme minimal de h. Nommons g la classe de X dans le quotient $F[X]/(P_L)$. Comme g et h ont le même polynôme minimal, l'application suivante est un isomorphisme :

$$\tau : \left\{ \begin{array}{rcl} L & \longrightarrow & K \\ \sum a_i g^i & \longmapsto & \sum a_i h^i \end{array} \right.$$

Il s'ensuit que L et K sont isomorphes. ■

Il est maintenant légitime de parler *du* corps fini à q éléments.

Notation A.11. *Pour q une puissance d'un nombre premier, on note $\mathbb{F}_q$ le corps fini à q éléments, unique à isomorphisme près.*

La propriété suivante est importante lorsque l'on s'intéresse à la factorisation de polynômes sur $\mathbb{F}_q$:

Proposition A.12. *Le polynôme $X^{q^n} - X \in \mathbb{F}_q[X]$ est le produit de tous les polynômes irréductibles de $\mathbb{F}_q[X]$ de degré divisant n.*

Démonstration. Soit P un diviseur irréductible de $X^{q^n} - X$, de degré k. Comme $X^{q^n} - X$ est scindé dans $L = \mathbb{F}_{q^n}$, P a une racine α dans L. Il existe donc un sous-corps $E = \mathbb{F}_q(\alpha)$ dans L. On a donc $[L : \mathbb{F}_q] = [L : E][E : \mathbb{F}_q]$, donc $k = [E : \mathbb{F}_q]$ divise $n = [L : \mathbb{F}_q]$.

La réciproque reprend une idée déjà utilisée plus haut. Soit P un polynôme irréductible de $\mathbb{F}_q$ de degré k divisant n. Soit E l'extension de $\mathbb{F}_q$ définie par P, et α une racine de P dans E. Comme E est de cardinal q^k, on a $\alpha^{q^k} = \alpha$. Mais comme k divise n, cela implique que $\alpha^{q^n} = \alpha$. Le polynôme $X^{q^n} - X$ est donc un multiple de P. ■

A.5 Nombre de polynômes irréductibles sur $\mathbb{F}_q$

On peut s'intéresser au nombre de polynômes irréductibles ayant un degré donné sur $\mathbb{F}_q$. On a besoin pour cela d'introduire la fonction de Möbius, et la formule d'inversion du même nom.

Définition A.13 (Fonction de Möbius). *On note μ l'application telle que $\mu(n) = (-1)^r$ si n peut s'écrire comme produit de r nombres premiers* distincts, *et $\mu(n) = 0$ sinon. La fonction μ est appelée* fonction de Möbius.

Cette fonction vérifie la propriété suivante.

Proposition A.14.

1. *$\forall m, n \in \mathbb{N}^*, m$ et n premiers entre eux $\Rightarrow \mu(mn) = \mu(m)\mu(n)$.*
2. *Si f et g sont deux fonctions de $\mathbb{N}^*$ dans $\mathbb{C}$ telles que :*

$$\forall n \in \mathbb{N}^*,\ f(n) = \sum_{d|n} g(d),$$

alors g s'obtient à partir de f par la formule suivante :

$$\forall n \in \mathbb{N}^*,\ g(n) = \sum_{d|n} f(d)\mu\left(\frac{n}{d}\right).$$

La vérification de cette proposition n'est pas effectuée ici. La formule donnée pour $g(n)$ s'appelle formule d'inversion de Möbius. On peut l'utiliser pour obtenir l'expression du nombre de polynômes irréductibles de degré n sur $\mathbb{F}_q$.

Proposition A.15. *Soit $I(q, n)$ le nombre de polynômes irréductibles de degré n sur $\mathbb{F}_q$, pour $n \in \mathbb{N}^*$. On a :*

1. $\displaystyle\sum_{d|n} dI(q, d) = q^n.$
2. $\displaystyle I(q, n) = \frac{1}{n}\sum_{d|n} q^d \mu(\frac{n}{d}).$

Démonstration. On obtient la première assertion à partir de la propriété A.12. Il suffit de compter le degré du polynôme $X^{q^n} - X$ de deux façons distinctes. Pour obtenir la deuxième propriété, on applique la formule d'inversion de Möbius. ■

Index

Bibliographie

[Adl79] L. M. Adleman. A subexponential algorithm for the discrete logarithm problem with applications to cryptography. Dans *20th Annual Symposium on Foundations of Computer Science (FOCS '79)*, pages 55–60. IEEE Computer Society Press, 1979. San Juan, Puerto Rico, October 29–31, 1979.

[Adl94] L. M. Adleman. The function field sieve. Dans L. M. Adleman et M.-D. Huang, éditeurs, *ANTS-I. Lecture Notes in Comput. Sci.*, volume 877, pages 108–121. Springer–Verlag, 1994. 1st Algorithmic Number Theory Symposium, Cornell University, May 6–9, 1994.

[AD93] L. M. Adleman et J. DeMarrais. A subexponential algorithm for discrete logarithms over all finite fields. *Math. Comp.*, 61(203):1–15, Jul. 1993.

[ADH94] L. M. Adleman, J. DeMarrais, et M.-D. Huang. A subexponential algorithm for discrete logarithms over the rational subgroup of the jacobians of large genus hyperelliptic curves over finite fields. Dans L. M. Adleman et M.-D. Huang, éditeurs, *ANTS-I. Lecture Notes in Comput. Sci.*, volume 877, pages 28–40. Springer–Verlag, 1994. 1st Algorithmic Number Theory Symposium, Cornell University, May 6–9, 1994.

[AH99] L. M. Adleman et M.-D. Huang. Function field sieve methods for discrete logarithms over finite fields. *Inform. and Comput.*, 151(1):5–16, 1999.

[AHU74] A. V. Aho, J. E. Hopcroft, et J. D. Ullman. *The design and analysis of computer algorithms*. Addison–Wesley, Reading, MA, 1974.

[BL94] B. Beckerman et G. Labahn. A uniform approach for the fast computation of matrix-type Padé approximants. *SIAM J. Matrix Anal. Appl.*, 15(3):804–823, Jul. 1994.

[BC99] E. A. Bender et E. R. Canfield. An approximate probabilistic model for structured Gaussian elimination. *J. Algorithms*, 31(2):271–290, 1999.

[Ber68] E. R. Berlekamp. *Algebraic coding theory*. McGraw-Hill, 1968.

[BA80] R. R. Bitmead et B. D. O. Anderson. Asymptotically fast solution of Toeplitz and related systems of linear equations. *Linear Algebra Appl.*, 34:103–116, 1980.

[BFHMV84] I. F. Blake, R. Fuji-Hara, R. C. Mullin, et S. A. Vanstone. Computing logarithms in finite fields of characteristic two. *SIAM J. Alg. Disc. Meth.*, 5(2):276–285, Jun. 1984.

[BMV85] I. F. Blake, R. C. Mullin, et S. A. Vanstone. Computing logarithms in $\mathbb{F}_{2^n}$. Dans G. R. Blakley et D. Chaum, éditeurs, *Advances in Cryptology – CRYPTO '84. Lecture Notes in Comput. Sci.*, volume 196, pages 73–82. Springer–Verlag, 1985. Proc. Cryptology Workshop, Santa Barbara, CA, USA, August 19–22, 1984.

[BR96] H. Boender et H. J. J. te Riele. Factoring integers with large-prime variations of the quadratic sieve. *Experiment. Math.*, 5(4):257–273, 1996.

[BF01] D. Boneh et M. Franklin. Identity-based encryption from the Weil pairing. Dans J. Kilian, éditeur, *Advances in Cryptology – CRYPTO 2001. Lecture Notes in Comput. Sci.*, volume 2139, pages 213–229. Springer–Verlag, 2001. Proc. 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001.

[BGL03] R. P. Brent, S. Gao, et A. G. B. Lauder. Random Krylov spaces over finite fields. *SIAM J. Discrete Math.*, 16(2):276–287, 2003.

[But97] D. R. Butenhof. *Programming with POSIX threads*. Professional computing series. Addison–Wesley, 1997.

[CAB00] CABAL. 233-digit SNFS factorization. Disponible à l'adresse `ftp://ftp.cwi.nl/pub/herman/SNFSrecords/SNFS-233`, Oct. 2000.

[CZ81] D. G. Cantor et H. Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Math. Comp.*, 36(154):587–592, Apr. 1981.

[Cav00] S. Cavallar. Strategies in filtering in the number field sieve. Dans W. Bosma, éditeur, *ANTS-IV. Lecture Notes in Comput. Sci.*, volume 1838, pages 209–231. Springer–Verlag, 2000. 4th Algorithmic Number Theory Symposium, Leiden, The Netherlands, July 2–7, 2000.

[Cav02] S. Cavallar. *On the number field sieve factorization algorithm.* Doctor's thesis, University of Leiden, 2002.

[CDL+00] S. Cavallar, B. Dodson, A. K. Lenstra, W. Lioen, P. L. Montgomery, B. Murphy, H. J. J. te Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. Leyland, J. Marchand, F. Morain, A. Muffett, C. Putnam, C. Putnam, et P. Zimmermann. Factorization of a 512-bit RSA modulus. Dans B. Preneel, éditeur, *Advances in Cryptology – EUROCRYPT 2000. Lecture Notes in Comput. Sci.*, volume 1807, pages 1–18. Springer–Verlag, 2000. Proc. International Conference on the Theory and Application of Cryptographic Techniques, Brugge, Belgium, May 2000.

[Cer] Certicom corp. The Certicom ECC challenge. Description at `http://www.certicom.com/resources/ecc_chall/challenge.html`.

[Cop84] D. Coppersmith. Fast evaluation of logarithms in fields of characteristic two. *IEEE Trans. Inform. Theory*, IT–30(4):587–594, Jul. 1984.

[Cop93] D. Coppersmith. Solving linear equations over GF(2): Block Lanczos algorithm. *Linear Algebra Appl.*, 192:33–60, Jan. 1993.

[Cop94] D. Coppersmith. Solving linear equations over GF(2) via block Wiedemann algorithm. *Math. Comp.*, 62(205):333–350, Jan. 1994.

[COS86] D. Coppersmith, A. M. Odlyzko, et R. Schroeppel. Discrete logarithms in GF(p). *Algorithmica*, 1:1–15, 1986.

[CP01] R. Crandall et C. Pomerance. *Prime numbers – A Computational Perspective.* Springer–Verlag, 2001.

[DH84] J. A. Davis et D. B. Holridge. Factorization using the quadratic sieve algorithm. Dans D. Chaum, éditeur, *Advances in Cryptology – CRYPTO '83*, pages 103–113, New York and London, 1984. Plenum Press. Proc. Cryptology Workshop, Santa Barbara, CA, August 22–24, 1983.

[DDLM94] T. Denny, B. Dodson, A. K. Lenstra, et M. S. Manasse. On the factorization of RSA-120. Dans D. R. Stinson, éditeur, *Advances in Cryptology – CRYPTO '93. Lecture Notes in Comput. Sci.*, volume 773, pages 166–186. Springer–Verlag, 1994. Proc. 13th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 1993.

[DH76] W. Diffie et M. E. Hellman. New directions in cryptography. *IEEE Trans. Inform. Theory*, IT–22(6):644–654, Oct. 1976.

[DL95] B. Dodson et A. K. Lenstra. NFS with four large primes: an explosive experiment. Dans D. Coppersmith, éditeur, *Advances in Cryptology – CRYPTO '95. Lecture Notes in Comput. Sci.*, volume 963, pages 372–385. Springer–Verlag, 1995. Proc. 15th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 27–31, 1995.

[Dor87] J.-L. Dornstetter. On the equivalence between Berlekamp's and Euclid's algorithm. *IEEE Trans. Inform. Theory*, IT–33(3):428–431, May 1987.

[DE03] R. Dupont et A. Enge. Practical non-interactive key distribution based on pairings. À paraître dans Proceedings WCC '03, 2003.

[DEM03] R. Dupont, A. Enge, et F. Morain. Building curves with arbitrary mov defree over finite prime fields. Manuscrit en préparation, 2003.

[ElG85] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inform. Theory*, IT–31(4):469–472, Jul. 1985.

[EH96] R. M. Elkenbracht-Huizing. An implementation of the number field sieve. *Experiment. Math.*, 5(3):231–253, 1996.

[EH97] R.-M. Elkenbracht-Huizing. *Factoring integers with the number field sieve*. Doctor's thesis, University of Leiden, 1997.

[EG02] A. Enge et P. Gaudry. A general framework for subexponential discrete logarithm algorithms. *Acta Arith.*, 102(1):83–103, 2002.

[FKP89] P. Flajolet, D. E. Knuth, et B. Pittel. The first cycles in an evolving graph. *Discrete Math.*, 75:167–215, 1989.

[FO90] P. Flajolet et A. M. Odlyzko. Random mapping statistics. Dans J.-J. Quisquater et J. Vandewalle, éditeurs, *Advances in Cryptology – EUROCRYPT '89. Lecture Notes in Comput. Sci.*, volume 434, pages 329–354. Springer–Verlag, 1990. Proc. Eurocrypt '89, Houthalen, April 10–13, 1989.

[FS93] P. Flajolet et R. Sedgewick. The average case analysis of algorithms: counting and generating functions. Rapport de recherche RR-1888, INRIA, Apr. 1993. chapitres 1–3 de *Analytic combinatorics*, à paraître.

[FS94] P. Flajolet et R. Sedgewick. The average case analysis of algorithms: saddle point asymptotics. Rapport de recherche RR-2376, INRIA, 1994. chapitre 6 de *Analytic combinatorics*, à paraître.

[FS02] P. Flajolet et R. Sedgewick. Analytic combinatorics – symbolic combinatorics. Disponible à l'adresse `http://algo.inria.fr/flajolet/Publications/books.html`, May 2002. chapitres 1–3 de *Analytic combinatorics*, à paraître. Version revue et augmentée de [FS93].

[FR94] G. Frey et H.-G. Rück. A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves. *Math. Comp.*, 62(206):865–874, Apr. 1994.

[vzGG96] J. von zur Gathen et J. Gerhard. Arithmetic and factorization of polynomials over $\mathbb{F}_2$ (extended abstract). Tech. Report tr-rsfb-96-018, University of Paderborn, Germany, 1996.

[vzGG99] J. von zur Gathen et J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, Cambridge, England, 1999.

[vzGG02] J. von zur Gathen et J. Gerhard. Polynomial factorization over $\mathbb{F}_2$. *Math. Comp.*, 71(240):1677–1698, Oct. 2002.

[vzGP01] J. von zur Gathen et D. Panario. Factoring polynomials over finite fields: A survey. *J. Symbolic Comput.*, 31(1):3–17, 2001.

[vzGS92] J. von zur Gathen et V. Shoup. Computing Frobenius maps and factoring polynomials. *Comput. Complexity*, 2:187–224, 1992.

[Gau00a] P. Gaudry. An algorithm for solving the discrete log problem on hyperelliptic curves. Dans B. Preneel, éditeur, *Advances in Cryptology – EUROCRYPT 2000. Lecture Notes in Comput. Sci.*, volume 1807, pages 19–34. Springer–Verlag, 2000. Proc. International Conference on the Theory and Application of Cryptographic Techniques, Brugge, Belgium, May 2000.

[Gau00b] P. Gaudry. *Algorithmique des courbes hyperelliptiques et applications à la cryptologie.* Thèse, École Polytechnique, Oct. 2000.

[GG01] P. Gaudry et N. Gürel. An extension of Kedlaya's algorithm to superelliptic curves. Dans C. Boyd et E. Dawson, éditeurs, *Advances in Cryptology – ASIACRYPT 2001. Lecture Notes in Comput. Sci.*, volume 2248, pages 480–494. Springer–Verlag, 2001. Proc. 7th International Conference on the Theory and Applications of Cryptology and Information Security, Dec. 9–13, 2001, Gold Coast, Queensland, Australia.

[GHS02] P. Gaudry, F. Hess, et N. Smart. Constructive and destructive facets of Weil descent on elliptic curves. *J. of Cryptology*, 15:19–46, 2002.

[GGL93] A. George, J. Gilbert, et J. W.-H. Liu, éditeurs. *Graph theory and sparse matrix computation. IMA Vol. Math. Appl.*, volume 56. Springer–Verlag, 1993.

[GL81] A. George et J. W.-H. Liu. *Computer Solutions of Large Sparse Positive Definite Systems*. Prentice-Hall Series in Computational Mathematics. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Gor93] D. M. Gordon. Discrete logarithms in GF(p) using the number field sieve. *SIAM J. Discrete Math.*, 6(1):124–138, Feb. 1993.

[GM93] D. M. Gordon et K. S. McCurley. Massively parallel computation of discrete logarithms. Dans E. F. Brickell, éditeur, *Advances in Cryptology – CRYPTO '92. Lecture Notes in Comput. Sci.*, volume 740, pages 312–323. Springer–Verlag, 1993. Proc. 12th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16–20, 1992.

[GMP] T. Granlund. `GMP`, the GNU multiple precision arithmetic library, 1996–. Homepage at `http://www.swox.com/gmp`.

[GY79] F. G. Gustavson et D. Y. Y. Yun. Fast algorithms for rational Hermite approximation and solution of Toeplitz systems. *IEEE Trans. Circuits Systems*, CAS–26(9):750–755, Sept. 1979.

[Göt94] R. Göttfert. An acceleration of the Niederreiter factorization algorithm in characteristic 2. *Math. Comp.*, 62(206):831–839, Apr. 1994.

[HQZ03] G. Hanrot, M. Quercia, et P. Zimmerman. The middle product algorithm, I. Speeding up the division and square root of power series. À paraître, 2003.

[Har] R. Harley. The ECDL project. `http://pauillac.inria.fr/~harley/ecdl_top/`.

[JL01] A. Joux et R. Lercier. Discrete logarithms in GF(2^n) (521 bits). E-mail sur la liste `NMBRTHRY`. Disponible à l'adresse `http://listserv.nodak.edu/archives/nmbrthry.html`, Sept. 2001.

[JL02] A. Joux et R. Lercier. The function field sieve is quite special. Dans C. Fieker et D. R. Kohel, éditeurs, *ANTS-V. Lecture Notes in Comput. Sci.*, volume 2369, pages 431–445. Springer–Verlag, 2002. 5th Algorithmic Number Theory Symposium, Sydney, Australia, July 2002.

[Kal95] E. Kaltofen. Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Math. Comp.*, 64(210):777–806, Apr. 1995.

[KL99] E. Kaltofen et A. Lobo. Distributed matrix-free solution of large sparse linear systems over finite fields. *Algorithmica*, 24(4):331–348, 1999.

[KS98] E. Kaltofen et V. Shoup. Subquadratic-time factoring of polynomials over finite fields. *Math. Comp.*, 67(223):1179–1197, Jul. 1998.

[Knu98] D. E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison–Wesley, 3$^{\text{e}}$ édition, 1998.

[Kob87] N. Koblitz. Elliptic curve cryptosystems. *Math. Comp.*, 48(177):203–209, Jan. 1987.

[Kob89] N. Koblitz. Hyperelliptic cryptosystems. *J. of Cryptology*, 1:139–150, 1989.

[Kry31] A. N. Krylov. On the numerical solutions of the equation by which the frequency of small oscillations is determined in technical problems (in russian). *Izv. Akad. Nauk SSSR Ser. Fiz.-Mat.*, 4:491–539, 1931.

[LO90] B. A. LaMacchia et A. M. Odlyzko. Solving large sparse linear systems over finite fields. Dans A. J. Menezes et S. A. Vanstone, éditeurs, *Advances in Cryptology – CRYPTO '90. Lecture Notes in Comput. Sci.*, volume 537, pages 109–133. Springer–Verlag, 1990. Proc. 10th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 11–15, 1990.

[Lam96] R. Lambert. *Computational aspects of discrete logarithms.* Phd thesis, University of Waterloo, 1996.

[LL93] A. K. Lenstra et H. W. Lenstra, Jr., éditeurs. *The development of the number field sieve. Lecture Notes in Math.*, volume 1554. Springer–Verlag, 1993.

[LLMP93] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, et J. M. Pollard. The number field sieve. Dans A. K. Lenstra et H. W. Lenstra, Jr., éditeurs, *The development of the number field sieve. Lecture Notes in Math.*, volume 1554, pages 11–42. Springer–Verlag, 1993.

[LM90] A. K. Lenstra et M. S. Manasse. Factoring by electronic mail. Dans J.-J. Quisquater et J. Vandewalle, éditeurs, *Advances in Cryptology – EUROCRYPT '89. Lecture Notes in Comput. Sci.*, volume 434, pages 355–371. Springer–Verlag, 1990. Proc. Eurocrypt '89, Houthalen, April 10–13, 1989.

[LM91] A. K. Lenstra et M. S. Manasse. Factoring with two large primes. Dans I. B. Damgård, éditeur, *Advances in Cryptology – EUROCRYPT '90. Lecture Notes in Comput. Sci.*, volume 473, pages 72–82. Springer–Verlag, 1991. Proc. Workshop on the Theory and Application of Cryptographic Techniques, Aarhus, Denmark, May 21–24, 1990.

[LM94] A. K. Lenstra et M. S. Manasse. Factoring with two large primes. *Math. Comp.*, 63(208):785–798, Oct. 1994.

[LV00] A. K. Lenstra et E. R. Verheul. The XTR public key system. Dans M. Bellare, éditeur, *Advances in Cryptology – CRYPTO 2000. Lecture Notes in Comput. Sci.*, volume 1880, pages 1–19. Springer–Verlag, 2000. Proc. 20th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20–24, 2000.

[LN83] R. Lidl et H. Niederreiter. *Finite fields. Encyclopedia of mathematics and its applications*, volume 20. Addison–Wesley, Reading, MA, 1983.

[Lob95] A. Lobo. *Matrix-free linear system solving and applications to symbolic computations.* Phd thesis, Rensselaer Polytechnic Institute, 1995.

[Mas69] J. L. Massey. Shift-register synthesis and BCH decoding. *IEEE Trans. Inform. Theory*, IT–15(1):122–127, Jan. 1969.

[Mat99] R. Matsumoto. Using C_{ab} curves in the function field sieve. *IEICE Trans. Fundamentals*, E82-A(3), Mar. 1999.

[MY92] U. Maurer et Y. Yacobi. Non-interactive public-key cryptography. Dans D. Davies, éditeur, *Advances in Cryptology – EUROCRYPT '91. Lecture Notes in Comput. Sci.*, volume 547, pages 498–507. Springer–Verlag, 1992. Proc. Workshop on the Theory and Application of Cryptographic Techniques, Brighton, United Kingdom, April 8–11, 1991.

[MY96] U. Maurer et Y. Yacobi. A non-interactive public-key distribution system. *Des. Codes Cryptogr.*, 9(3):305–316, 1996.

[Mau94] U. M. Maurer. Towards the equivalence of breaking the Diffie-Hellman protocol and computing discrete logarithms. Dans Y. G. Desmedt, éditeur, *Advances in Cryptology – CRYPTO '94. Lecture Notes in Comput. Sci.*, volume 839, pages 271–281. Springer–Verlag, 1994. Proc. 14th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 21–25, 1994.

[MW96] U. M. Maurer et S. Wolf. Diffie-Hellman oracles. Dans N. Koblitz, éditeur, *Advances in Cryptology – CRYPTO '96. Lecture Notes in Comput. Sci.*, volume 1109, pages 268–282. Springer–Verlag, 1996. Proc. 16th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 1996.

[MOV93] A. Menezes, T. Okamoto, et S. A. Vanstone. Reducing elliptic curves logarithms to logarithms in a finite field. *IEEE Trans. Inform. Theory*, IT–39(5):1639–1646, Sept. 1993.

[MvOV97] A. Menezes, P. C. van Oorschot, et S. A. Vanstone. *Handbook of applied cryptography*. CRC Press, 1997.

[Mil87] V. Miller. Use of elliptic curves in cryptography. Dans A. M. Odlyzko, éditeur, *Advances in Cryptology – CRYPTO '86. Lecture Notes in Comput. Sci.*, volume 263, pages 417–426. Springer–Verlag, 1987. Proc. 7th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 1986.

[Mon95] P. L. Montgomery. A block Lanczos algorithm for finding dependencies over GF(2). Dans L. C. Guillou et J.-J. Quisquater, éditeurs, *Advances in Cryptology – EUROCRYPT '95. Lecture Notes in Comput. Sci.*, volume 921, pages 106–120, 1995. Proc. International Conference on the Theory and Application of Cryptographic Techniques, Saint-Malo, France, May 1995.

[Mor93] F. Morain. Analyzing pmpqs. Disponible à l'adresse `ftp://lix.polytechnique.fr/pub/submissions/morain/Preprints/pmpqs.ps.Z`, 1993. Note informelle.

[Mor80] M. Morf. Doubling algorithms for Toeplitz and related equations. Dans *Proc. IEEE Internat. Conference Acoustics, Speech and Signal Processing*, pages 954–959, New York, NY, 1980. IEEE.

[MB75] M. A. Morrison et J. Brillhart. A method of factoring and the factorization of F_7. *Math. Comp.*, 29(129):183–205, Jan. 1975.

[MPI] `MPI`, message passing interface, 1994–. Documentation, and homepage of the MPIch implementation at `http://www-unix.mcs.anl.gov/mpi/`.

[Nec94] V. I. Nechaev. Complexity of a determinate algorithm for the discrete logarithm. *Mathematical Notes*, 55(2):165–172, 1994.

[Nie93a] H. Niederreiter. Factorization of polynomials and some linear-algebra problems over finite fields. *Linear Algebra Appl.*, 192:301–328, 1993.

[Nie93b] H. Niederreiter. A new efficient factorization algorithm for polynomials over small finite fields. *Appl. Algebra Engrg. Comm. Comput.*, 4:81–87, 1993.

[Odl85] A. M. Odlyzko. Discrete logarithms in finite fields and their cryptographic significance. Dans T. Beth, N. Cot, et I. Ingemarsson, éditeurs, *Advances in Cryptology – EUROCRYPT '84. Lecture Notes in Comput. Sci.*, volume 209, pages 224–314. Springer–Verlag, 1985. Proc. Eurocrypt '84, Paris (France), April 9–11, 1984.

[vOW99] P. C. van Oorschot et M. J. Wiener. Parallel collision search with cryptanalytic applications. *J. of Cryptology*, 12:1–28, 1999.

[PTL85] B. N. Parlett, D. R. Taylor, et Z. A. Liu. A look-ahead Lanczos algorithm for unsymmetric matrices. *Math. Comp.*, 44(169):105–124, Jan. 1985.

[Pen98] O. Penninga. Finding column dependencies in sparse matrices over $\mathbb{F}_2$ by block Wiedemann. Report MAS-R9819, Centruum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1998. Available from `http://www.cwi.nl/`.

[PH78] S. Pohlig et M. E. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Trans. Inform. Theory*, IT–24:106–110, 1978.

[Pol75] J. M. Pollard. A Monte-Carlo method for factorization. *BIT*, 15:331–334, 1975.

[Pol78] J. M. Pollard. Monte Carlo methods for index computation (mod p). *Math. Comp.*, 32(143):918–924, Jul. 1978.

[Pol93] J. M. Pollard. The lattice sieve. Dans A. K. Lenstra et H. W. Lenstra, Jr., éditeurs, *The development of the number field sieve. Lecture Notes in Math.*, volume 1554, pages 43–49. Springer–Verlag, 1993.

[Pom82] C. Pomerance. Analysis and comparison of some integer factoring algorithms. Dans H. W. Lenstra, Jr. et R. Tijdeman, éditeurs, *Computational methods in number theory*, pages 89–140. Mathematisch Centrum, Amsterdam, 1982. Mathematical Center Tracts 154/155.

[PS92] C. Pomerance et J. W. Smith. Reduction of huge, sparse matrices over finite fields via created catastrophes. *Experiment. Math.*, 1(2):89–94, 1992.

[PVM] `PVM`, parallel virtual machine, 1992–. Documentation at `http://www.csm.ornl.gov/pvm/pvm_home.html`.

[QD90] J.-J. Quisquater et J.-P. Delescaille. How easy is collision search? Application to DES. Dans J.-J. Quisquater et J. Vandewalle, éditeurs, *Advances in Cryptology – EUROCRYPT '89. Lecture Notes in Comput. Sci.*, volume 434, pages 429–434. Springer–Verlag, 1990. Proc. Eurocrypt '89, Houthalen, April 10–13, 1989.

[Ris72] J. Rissanen. Realizations of matrix sequences. Tech. Report RJ-1032, IBM Research, T. J. Watson Research Center, Yortown Heights, New York, NY, 1972.

[RSA78] R. L. Rivest, A. Shamir, et L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM*, 21(2):120–126, 1978.

[Rüc99] H. G. Rück. On the discrete logarithm in the divisor class group of curves. *Math. Comp.*, 68(226):805–806, Apr. 1999.

[SA98] T. Satoh et K. Araki. Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves. *Comment. Math. Helv.*, 47(1):81–92, 1998.

[SS85] J. Sattler et C. P. Schnorr. Generating random walks in groups. *Ann. Univ. Sci. Budapest. Sect. Comput.*, 6:65–79, 1985.

[SWD96] O. Schirokauer, D. Weber, et T. F. Denny. Discrete logarithms: The effectiveness of the index calculus method. Dans H. Cohen, éditeur, *ANTS-II. Lecture Notes in Comput. Sci.*, volume 1122, pages 337–361. Springer–Verlag, 1996. 2nd Algorithmic Number Theory Symposium, Talence, France, May 18–23, 1996.

[Sch91] C. P. Schnorr. Efficient signature generation by smart cards. *J. of Cryptology*, 4(3):161–174, 1991.

[SS71] A. Schönhage et V. Strassen. Schnelle Multiplikation größer Zahlen. *Computing*, 7:281–292, 1971.

[Sed88] R. Sedgewick. *Algorithms*. Addison–Wesley, 2^{e} édition, 1988.

[Sem98a] I. A. Semaev. An algorithm for evaluation of discrete logarithms in some nonprime finite fields. *Math. Comp.*, 67(224):1679–1689, Oct. 1998.

[Sem98b] I. A. Semaev. Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curves in characteristic p. *Math. Comp.*, 67(221):353–356, Jan. 1998.

[Sha85] A. Shamir. Identity-based cryptosystems and signature schemes. Dans G. R. Blakley et D. Chaum, éditeurs, *Advances in Cryptology – CRYPTO '84. Lecture Notes in Comput. Sci.*, volume 196, pages 47–53. Springer–Verlag, 1985. Proc. Cryptology Workshop, Santa Barbara, CA, USA, August 19–22, 1984.

[Sha71] D. Shanks. Class number, a theory of factorization, and genera. Dans D. J. Lewis, éditeur, *1969 Number theory institute. Proc. Sympos. Pure Math.*, volume 20, pages 415–440. Amer. Math. Soc., 1971.

[Sho90] V. Shoup. On the deterministic complexity of factoring polynomials over finite fields. *Inform. Process. Lett.*, 33:261–267, 1990.

[Sho95] V. Shoup. A new polynomial factorization algorithm and its implementation. *J. Symbolic Comput.*, 20(4):363–397, 1995.

[Sho97] V. Shoup. Lower bounds for discrete logarithms and related problems. Dans W. Fumy, éditeur, *Advances in Cryptology – EUROCRYPT '97. Lecture Notes in Comput. Sci.*, volume 1233, pages 256–266. Springer–Verlag, 1997. Proc. International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 1997.

[Sil86] J. H. Silverman. *The arithmetic of elliptic curves. Grad. Texts in Math.*, volume 106. Springer–Verlag, 1986.

[Sma99] N. Smart. The discrete logarithm problem on elliptic curves of trace one. *J. of Cryptology*, 12(3):193–196, 1999.

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[Tes01] E. Teske. On random walks for Pollard's rho method. *Math. Comp.*, 70(234):809–825, Apr. 2001.

[Tho01a] E. Thomé. Fast computation of linear generators for matrix sequences and application to the block Wiedemann algorithm. Dans B. Mourrain, éditeur, *ISSAC 2001*, pages 323–331. ACM Press, 2001. Proc. International Symposium on Symbolic and Algebraic Computation, July 22–25, 2001, London, Ontario, Canada.

[Tho01b] E. Thomé. Computation of discrete logarithms in $\mathbb{F}_{2^{607}}$. Dans C. Boyd et E. Dawson, éditeurs, *Advances in Cryptology – ASIACRYPT 2001. Lecture Notes in Comput. Sci.*, volume 2248, pages 107–124. Springer–Verlag, 2001. Proc. 7th International Conference on the Theory and Applications of Cryptology and Information Security, Dec. 9–13, 2001, Gold Coast, Queensland, Australia.

[Tho02a] E. Thomé. Discrete logarithms in GF(2^{607}). E-mail sur la liste `NMBRTHRY`. Disponible à l'adresse `http://listserv.nodak.edu/archives/nmbrthry.html`, Feb. 2002.

[Tho02b] E. Thomé. Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm. *J. Symbolic Comput.*, 33(5):757–775, Jul. 2002.

[Vil97] G. Villard. A study of Coppersmith's block Wiedemann algorithm using matrix polynomials. Rapport de recherche 975, LMC-IMAG, Grenoble, France, Apr. 1997.

[WD98] D. Weber et T. Denny. The solution of McCurley's discrete log challenge. Dans H. Krawczyk, éditeur, *Advances in Cryptology – CRYPTO '98. Lecture Notes in Comput. Sci.*, volume 1462, pages 458–471. Springer–Verlag, 1998. Proc. 18th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 23–27, 1998.

[Wie86] D. H. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Trans. Inform. Theory*, IT–32(1):54–62, Jan. 1986.

[Zim98] P. Zimmermann. An implementation in GMP of Schönhage's fast multiplication algorithm modulo $2^N + 1$, 1998. Programme `mul_fft.c` dans la distribution de GMP, versions 4.x [GMP].

Résumé

Le calcul de logarithmes discrets est un problème central en cryptologie. Lorsqu'un algorithme sous-exponentiel pour résoudre ce problème existe, le cryptosystème concerné n'est pas nécessairement considéré comme disqualifié, et il convient d'actualiser avec soin l'état de l'art de la cryptanalyse. Les travaux de ce mémoire s'inscrivent dans cette optique. Nous décrivons en particulier comment nous avons atteint un record de calculs de logarithmes discrets: $\mathbb{F}_{2^{607}}$.

Dans une première partie, nous exposons les différentes améliorations que nous avons apportées à l'algorithme de Coppersmith pour le calcul de logarithmes discrets en caractéristique 2. Ces améliorations ont rendu possible le record que nous avons atteint. La portée de ce calcul dépasse le simple cadre des corps finis, à cause de l'existence de la réduction MOV d'une part, et de la récente introduction des cryptosystèmes fondés sur l'identité.

On s'intéresse plus en détail, dans une seconde partie du mémoire, au problème classique de la résolution d'un système linéaire creux défini sur un corps fini, porté aux limites de ce que la technologie (théorique et pratique) permet. Nous montrons comment une amélioration substantielle de l'algorithme de Wiedemann par blocs a rendu celui-ci compétitif pour la résolution d'un grand système linéaire creux sur $\mathbb{F}_p$.

Une partie de ce mémoire est consacrée au point de vue de l'expérimentateur, grand utilisateur de moyens de calcul, de la surcharge de travail humain que cela impose, et des constatations que cette position amène.

Abstract

Computing discrete logarithms is a fundamental task for public key cryptanalysis. The mere existence of a subexponential algorithm for this purpose is not sufficient to definitely rule on the security level provided by some cryptosystem. Assessing state-of-the-art cryptanalysis calls for a thorough evaluation process. This dissertation contributes to such an evaluation. In particular, a record computation for discrete logarithms over $\mathbb{F}_{2^{607}}$ is described.

The first part of this thesis focuses on our study and use of Coppersmith's algorithm for computing discrete logarithms in finite fields of characteristic two. We brought several improvements to this algorithm, which made the record computation feasible. The relevance of such a computation extends beyond the realm of finite fields, because of the existence of the MOV reduction on the one hand, and the recently introduced identity-based cryptography on the other hand.

The second part of this work addresses the classical problem of solving large sparse linear systems over finite fields, using the full power of existing algorithms and hardware in order to solve the largest possible linear systems. Specifically, we show how the block Wiedemann algorithm can be substantially improved in order to become very competitive for solving large sparse linear systems over $\mathbb{F}_p$.

Practical considerations on the achievement of the computations implied by this work are also discussed. These computations involved large resources, and required an important management work on the human side. Driving such tasks also yields some observations.

Laboratoire d'informatique École polytechnique
91128 Palaiseau Cedex – FRANCE

Tél: +33 (0)1 69 33 40 73 – Fax: +33 (0)1 69 33 30 14
http://www.lix.polytechnique.fr/