UNIVERSITÉ
PARIS-EST
MARNE-LA-VALLÉE

GASPARD · MONGE · LABORATOIRE D'INFORMATIQUE

Paramétrage de CADO-NFS pour le problème du logarithme discret dans les corps finis premiers

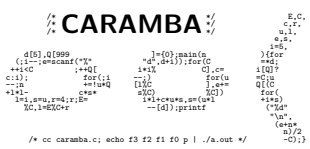
Kevin TRANCHO

Stage encadré par Pierrick GAUDRY

Master Informatique 1ère année

Année 2017 - 2018

CARAMBA



/* cc caramba.c; echo f3 f2 f1 f0 p | ./a.out */

cnrs dépasser les frontières

Inria informatics mathematics

Loria Laboratoire lorrain de recherche en informatique et ses applications

Table des matières

Remerciements

Je tiens à remercier en premier lieu Pierrick Gaudry pour l'encadrement et le suivi de mon stage, ses explications, sa disponibilité à tout moment pour répondre à mes questions et par la même occasion pour la relecture de ce document.

Je tiens de même à remercier Cyril Nicaud, qui a su répondre à ma demande de faire un stage en laboratoire de recherche sur le thème de la cryptographie en me mettant en contact avec l'équipe CARAMBA.

J'aimerais aussi remercier Aurore Guillevic pour les informations sur comment se loger à Nancy transmises après mon premier passage au Loria.

Je tiens finalement à remercier Aude, Sandra et Svyatoslav qui ont partagé mon bureau, ainsi que les membres de l'équipe CARAMBA que j'ai pu recontrer durant mon stage au Loria, c'est-à-dire Cécile, Emmanuel, Jérémie, Jianyang, Marine, Paul H., Paul Z., Pierre-Jean, Simon A., Simon M., Stéphane.

Première partie
Introduction

1 Sujet de stage

1.1 Le Loria et l'équipe CARAMBA

Pour poser le contexte de mon stage, je suis tenu dans le cadre de ma première année de master informatique d'effectuer un stage en entreprise, par curiosité pour la recherche et la cryptographie, j'ai donc souhaité faire un stage en laboratoire de recherche. J'ai fait mon stage au **Loria** (Laboratoire lorrain de recherche en informatique et ses applications) dans l'équipe **CARAMBA** (Cryptology, arithmetic : algebraic methods for better algorithms). J'ai été encadré pour ce stage par **Pierrick Gaudry**, directeur de recherche au CNRS, dont les travaux de recherche se focalisent principalement sur les courbes elliptiques et hyperelliptique, la factorisation d'entiers, le vote électronique et le logarithme discret dans les corps finis.

Le Loria se situe à Vandœuvre-lès-Nancy sur le campus scientique où se trouve l'Université de Lorraine. Il est par ailleurs affilié au **CNRS** et à l'Inria de Nancy. Le Loria c'est 190 chercheurs / enseignants chercheurs et 100 doctorants qui y travaillent, ils se divisent en 27 équipes de recherche. Pour donner une idée des principaux thèmes de recherche au Loria, je citerai ses départements de recherche qui sont les suivants :

- **Algorithmique, calcul, image et géométrie** s'intéresse principalement aux domaines de l'algorithmique, l'imagerie et la cryptologie. Les équipes en étudient les principes mathématiques pour en proposer des implémentations robustes et efficaces. Je donne une idée des thèmes de travail du département par les mots clés suivants : calcul symbolique, géométrie, traitement d'image, vision par ordinateur, réalité augmentée, combinatoire, algorithmique, cryptographie, cryptanalyse, courbes algébriques et systèmes polynomiaux.
- **Méthodes formelles** étudie principalement les preuves dans leur aspect théorique par la théorie des preuves, la logique, des méthodologie pour l'analyse et en propose des outils pour vérifier les système informatiques.
- **Réseaux, systèmes et services** regarde les réseaux dans l'idée de les faire travailler ensemble. C'est-à-dire que les équipent étudient des problèmes liés aux systèmes distribués ou au parallélisme pour la coopération entre les réseaux.
- **Traitement automatique des langues et des connaissances** s'intéresse aussi bien à l'analyse, la reconnaissance, la synthèse de la parole qu'à la représentation du discours. Ce département étudie aussi l'aspect écrit du traitement des langues par la reconnaissance de formes, la reconnaissance de symboles et la modélisation de l'écriture manuscrite. Comme autre thème de recherche, on peut lui citer ceux liés à l'intelligence artificielle : ici par la représentation des connaissances, la formalisation du raisonnement, les méthodes d'apprentissage et la classification.
- **Systèmes complexes, intelligence artificielle et robotique** s'intéresse à des domaines tels que biologie computationnelle pour améliorer la modélisation 3D de machines biomoléculaires multi-composants mais aussi aux sciences cognitives ou neurosciences, pour par exemple proposer des applications en médecine ou concevoir de meilleurs robots humanoïdes. Ce département s'intéresse de même à la robotique par la création de nouveaux algorithmes pour permettre aux robots d'être autonomes, d'intéragir avec leur environnement ou même contrôler les robots par la pensée. J'ai par exemple entendu parler d'un appartement intelligent ou d'une arène de robots au Loria.

Dans le cas présent, je suis affecté au département **Algorithmique, calcul, image et géométrie** dans l'équipe CARAMBA qui est dirigée par Emmanuel Thomé depuis Janvier 2016, anciennement CARAMEL dirigée par Pierrick Gaudry. L'équipe s'intéresse principalement à la cryptographie à clé publique, c'est-à-dire plus particulièrement aux **attaques mathématiques** utilisant la théorie des nombres sur des standards tels que le RSA avec la **factorisation d'entiers**, ou ElGamal et l'échange de clé de Diffie-Hellman avec le **logarithme discret**. Pour ceci l'équipe travaille par exemple

sur l'**implémentation robuste** d'algorithmes permettant la résolution de ces problèmes. D'autres de leurs intérêts de recherche sont aussi les réseaux euclidiens, l'algèbre linéaire ou systèmes polynomiaux par exemple. Pour rester dans le domaine de la cryptographie, l'équipe s'intéresse aussi aux courbes elliptiques et plus particulièrement aux courbes elliptiques de genre 2 et genre 3 pour proposer de même d'autres systèmes cryptographiques se basant sur le calcul dans les courbes elliptiques. De même l'équipe propose aussi des bibliothèques de calcul de nombres à virgule flottante en précision arbitraire ou de calcul efficace d'arithmétique dans des corps finis tel que GNU MPFR par exemple.

1.2 Problème du logarithme discret

Le problème du logarithme discret est un outil moderne de la cryptographie. Pour parler un peu plus de cryptograhie, c'est la science qui consiste à par exemple sécuriser des communications, des transactions ou même authentifier des organismes et signatures numériques. Longtemps nous nous sommes intéressés à des chiffrements symétriques tels que le chiffre de César, le chiffrement Vigénère, la machine Enigma dont le principe était de pouvoir partager un message secret que seuls les détenteurs de la clé et des méthodes de déchiffrement pouvaient lire. Mais dans un contexte moderne où nous souhaitons communiquer en toute sécurité sur un réseau ou s'assurer qu'un organisme qui nous vend ses services est bien celui qu'il prétend être nous devons passer par un mécanisme à clé publique.

Dans les solutions pour répondre à ce problème nous trouvons par exemple le système de cryptage RSA sur lequel repose par exemple la sécurité du système français des cartes bleues ou encore le protocole standard SSL/TLS (utilisé pour le 's' de https) sur lequel repose principalement les communications sur internet par un navigateur web. L'idée de ces deux solutions est que leur sécurité repose sur deux problèmes mathématiquement difficiles à résoudre : respectivement la factorisation d'entier pour RSA et le problème du logarithme discret pour SSL/TLS. La factorisation d'entiers a beaucoup intéressé la communauté scientifique pour son utilisation très présente en mathématiques et informatique, mais le problème du logarithme discret a aussi fait l'objet de records comme le calcul d'un logarithme discret dans un corps fini premier où p était un nombre premier 'sûr' de 768 bits [9] ou encore un nombre premier de 1024 bits piégé de manière à pouvoir calculer les logarithmes discrets [5]. Dans notre cas nous parlerons du problème du logarithme discret dans un corps fini de la forme $\mathbb{F}_p^* = (\mathbb{Z}/p\mathbb{Z})^*$ où p est un nombre premier.

Definition. *Soit p un nombre premier. Soit $g \in \mathbb{F}_p^*$ un générateur d'ordre ℓ premier tel que ℓ divise $p-1$. Soit $h \in \langle g \rangle$ un élément du sous-groupe de $\mathbb{F}_p^*$ engendré par g. Il existe $x \in \mathbb{Z}$ tel que $h = g^x$. Pour tout $k \in \mathbb{Z}$, $h = g^x = g^{x+k\ell}$, nous considérerons alors $x \in \mathbb{Z}/\ell\mathbb{Z}$. Le problème du logarithme discret consiste à retrouver x en connaissant p, g et h, ce que nous noterons :*

$$x = \log_g(h).$$

1.3 Travail du stage

Nous étudierons CADO-NFS [13], une implémentation d'un algorithme nommé le crible algébrique pour la résolution du logarithme discret. Mon sujet de stage consiste à paramétrer CADO-NFS pour les petites tailles, c'est-à-dire trouver des jeux de paramètres pertinents pour $(\mathbb{Z}/p\mathbb{Z})^*$ où les tailles des p premiers vont de 30 à 100 chiffres et ensuite comparer les performances de CADO-NFS avec d'autres implémentations du logarithme discret qui seront Sage, Magma et Pari/GP. Dans notre cas nous nous intéresserons seulement aux corps premiers $\mathbb{F}_p^*$ où p est un nombre premier de Sophie Germain, c'est-à-dire que p est tel que $p = 2\ell + 1$ avec ℓ un nombre premier, dans l'idée de construire les fichiers de paramètres dans le pire cas. Les générateurs g utilisés seront d'ordre $\ell = \frac{p-1}{2}$, car c'est le cas le plus difficile pour la résolution du problème du logarithme discret. Dans l'idée que le stage que j'ai effectué en entreprise était pour moi l'occasion de m'initier à la recherche scientifique, pour illustrer mes activités et résultats de stage j'ai choisi d'écrire mon manuscrit dans l'esprit d'un article de recherche. Pour ceci je présenterai dans un premier temps l'algorithme du crible algébrique pour comme brique de base pour la compréhension des mes travaux de recherche de paramètres pour finalement présenter les résultats que ceux-ci ont pu mettre en évidence pour les tailles étudiées.

Deuxième partie

Algorithme du crible algébrique

2 Vue d'ensemble de l'algorithme NFS

NFS (Number Field Sieve) désignera par la suite l'algorithme du 'crible algébrique'. La grande force de CADO-NFS est de pouvoir précalculer une base de données de logarithmes pour un couple (p, ℓ) donné à partir de laquelle le temps passé ensuite est réduit de manière significative pour le calcul d'un logarithme cible, ici noté h, ce qui laisse la possibilité de calculer rapidement un grand nombre de logarithmes discrets une fois les précalculs effectués. En effet, l'algorithme NFS pour le logarithme discret s'est déjà fait connaître sur des attaques d'échanges de clés, comme présenté dans l'article 'Imperfect Forward Secrecy : How Diffie-Hellman Fails in Practice' [1]. Le diagramme en figure 1 illustre les grandes étapes du déroulement de CADO-NFS. En effet l'algorithme NFS se divise en plusieurs grandes étapes que je détaille dans les sections suivantes. La complexité théorique de NFS, pour une taille de corps fini N, est donnée par une fonction sous-exponentielle :

$$L_N(\alpha, c) = exp\left(c\,(1 + o\,(1))\,(\log N)^{\alpha}\,(\log(\log N))^{1-\alpha}\right)$$

où $\alpha \in [0, 1]$. Pour comprendre un peu ce que nous dit cette formule, pour une taille de chiffres donnée pour N si $\alpha = 0$, alors $L_N(\alpha, c) = (\log N)^c$ ce qui correspondrait à une complexité polynomiale et dans le cas $\alpha = 1$, $L_N(\alpha, c) = N^c$ nous obtenons un complexité exponentielle. Plus précisément dans le cas de NFS, la complexité asymptotique est $L_N\left(\frac{1}{3}, \left(\frac{64}{9}\right)^{\frac{1}{3}}\right)$. Malheureusement, cette complexité théorique de NFS n'est pas prouvée.

En parallèle des explications, nous déroulerons un exemple pour un p nombre premier de Sophie Germain de taille 60 à l'aide de CADO-NFS. Ici :

$$p = 466706088568935459448191002584064434223453742726470240679179$$
$$\ell = 233353044284467729724095501292032217111726871363235120339589$$

Pour les explications suivantes, je me suis principalement aidé d'explications données au tableau par Pierrick, d'expérimentations sur CADO-NFS et d'informations trouvées dans des documents tels que les notes d'un cours donné aux Journées Nationales de Calcul Formel par Pierrick Gaudry [6], les thèses de Cyril Bouvier [3] et Laurent Grémy [7] par exemple.

Dans les calculs effectués par CADO-NFS, on considère que le générateur est inconnu, il nous suffit seulement de connaître son ordre ℓ dans $\mathbb{F}_p^*$, il sera malgré tout possible de retrouver le logarithme discret de h, c'est-à-dire que pour un générateur g choisi, si on a le logarithme discret en base inconnue γ, on retrouve $\log_g(h) = \log_\gamma(h) \times \log_\gamma(g)^{-1} \bmod \ell$.

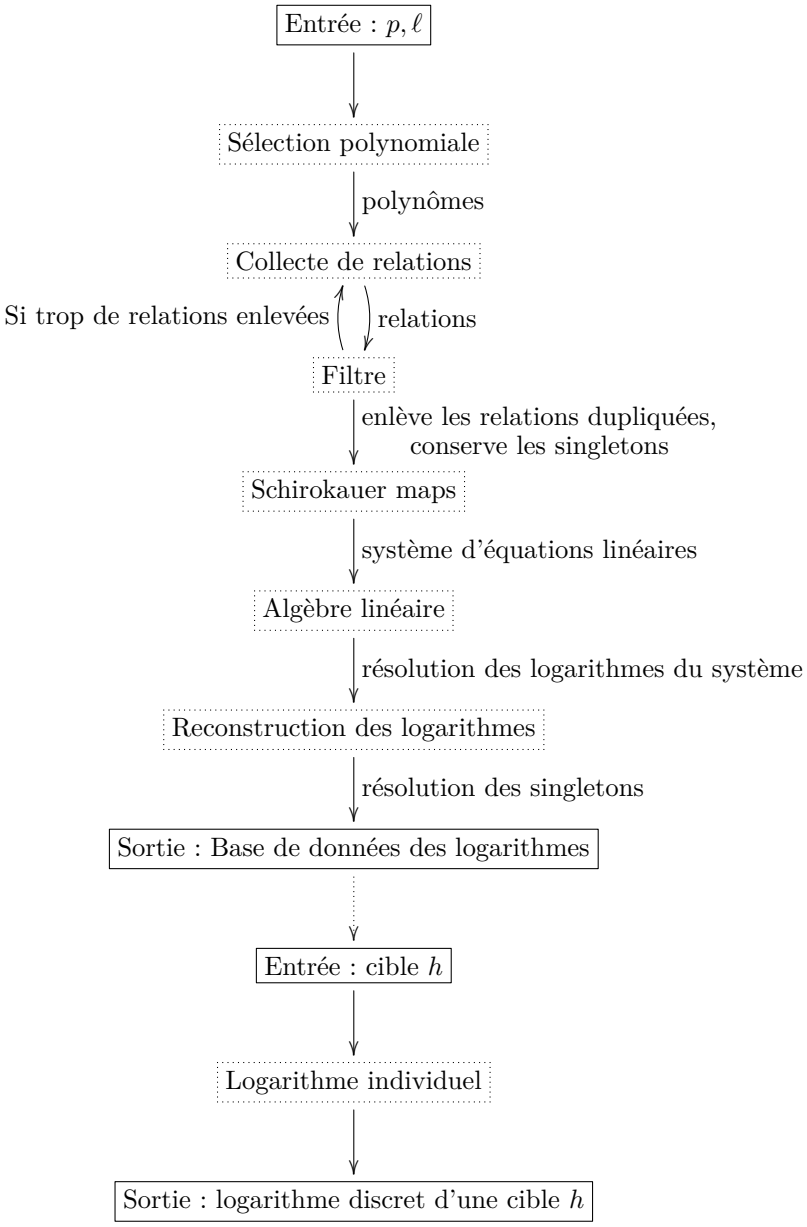


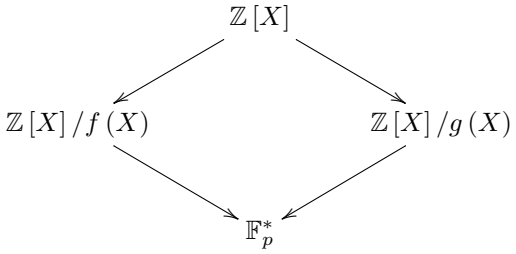
FIGURE 1 – Grandes étapes de CADO-NFS

3 Sélection polynomiale

Lors de la sélection polynomiale nous allons chercher une paire de polynômes (f, g) avec les propriétés suivantes :

— f et g sont irréductibles dans $\mathbb{Z}[X]$
— f et g ont une racine commune modulo p

Ce choix de polynômes nous donne le diagramme suivant :



La propriété d'irréductibilité nous permet de construire les anneaux intègres $\mathbb{Z}[X]/f(X)$ et $\mathbb{Z}[X]/g(X)$, de plus si un morphisme envoie un élément de $\mathbb{Z}[X]$ dans $\mathbb{Z}[X]/f(X)$ d'un côté et dans $\mathbb{Z}[X]/g(X)$ d'un autre, on retrouve une égalité en revenant depuis ces anneaux dans $\mathbb{F}_p^*$, ce qui est justifié par le choix d'avoir construit f et g tels qu'il existe $m \in \mathbb{Z}$ vérifiant :

$$f(m) \equiv g(m) \equiv 0\,[p]\,.$$

Cependant il n'existe pas une unique paire de polynômes vérifiant ces propriétés, et la 'qualité' des polynômes sélectionnés aura un impact sur le temps passé dans les étapes suivantes. Pour notre exemple, nous étudierons le cas de la sélection polynomiale de Kleinjung qui est la variante proposée dans 'Better polynomials for GNFS' [2], c'est-à-dire la même sélection polynomiale que celle utilisée pour la factorisation par CADO-NFS et le cas de la sélection polynomiale par l'algorithme de Joux-Lercier [8]. Pour illustrer la sélection polynomiale, nous donnerons ici un exemple de paire (f, g) obtenu pour la taille 60 énoncée plus haut.

Sélection polynomiale de Kleinjung

$$\begin{aligned} f(x) = \;& 360x^4 + 2125558x^3 - 16372440077x^2 - 4982864143076x \\ & +3060979830926535 \\ g(x) = \;& 113138957801x + 37368903092258 \end{aligned}$$

On vérifie que f et g ont une racine commune modulo p. En effet pour $m = 412092500565008394151192031055011685608760618492699798665040$, $f(m) \equiv g(m) \equiv 0\,[p]$.
Les principaux paramètres pour l'algorithme de sélection polynomiale de Kleinjung sont les suivants :

DEGREE :	ce paramètre est le degré de f, dans cette sélection polynomiale, le degré de g sera toujours 1
ADMAX :	est la plus grande valeur que prendra le coefficient de plus grand degré de f
INCR :	est le pas d'incrémentation du coefficient dominant de f
P :	deux facteurs premiers du coefficient dominant de g devront se trouver dans l'intervalle $[\![P; 2P]\!]$
NQ :	est le nombre maximal de spécial-q utilisés pour une évaluation d'une paire (f, g)

Les paramètres utilisés pour cet exemple sont :

DEGREE =	4
ADMAX =	1000
INCR =	60
P =	400
NQ =	250

Sélection polynomiale Joux-Lercier

$$\begin{aligned} f(x) = &\quad 4x^3 + x^2 - 4x - 4 \\ g(x) = &\quad -24975517809304245 13x^2 - 29639340115409070901x \\ &\quad -92575894709422471926 \end{aligned}$$

Pour la même vérification, on observe que pour $m = 335959898298675295428513623088321040109332984127504505193504$, on a bien $f(m) \equiv g(m) \equiv 0\,[p]$. Les principaux paramètres pour la sélection polynomiale par l'algorithme de Joux-Lercier sont les suivants :

DEGREE :	est le degré de f, le degré de g sera lui DEGREE−1
BOUND :	définit l'intervalle dans lequel on va choisir les différents coefficients de f

Dans cet exemple, les paramètres étaient :

$$\begin{aligned} \text{DEGREE} = &\quad 3 \\ \text{BOUND} = &\quad 4 \end{aligned}$$

Dans la suite nous allons générer des paires $(a, b) \in \mathbb{Z}^2$ dans le but d'obtenir des relations (Nous parlerons plus en détail des relations dans l'étape de la collecte des relations). Pour une paire (a, b) par exemple envoyée dans $\mathbb{Z}[X]/f(X)$, la norme associée à la paire (a, b) est la suivante :

$$Norm((a, b)) = b^d f(\frac{a}{b}).$$

Ensuite l'objectif est d'exploiter la propriété de friabilité de la norme associée à une paire (a, b).

Definition. *On dit qu'un entier N est B-friable pour un entier B donné si pour tout facteur premier q de N, $q < B$. Autrement dit, on a :*

$$N = \prod_{q<B} q^{e_q}.$$

Dans le but de pouvoir sélectionner un couple (f, g) prometteur pour la collecte de relations, on se base sur le critère de qualité qu'on estime par la quantité E de Murphy [11], dont le calcul donne une estimation du nombre de relations qu'on pourrait obtenir selon la probabilité de friabilité des éléments des deux côtés. On estime dans le calcul de la quantité E de Murphy la proportion d'entiers friables à l'aide de la fonction ρ de Dickman. Plus la quantité obtenue est grande, meilleure est la paire de polynômes trouvée.

Sélection polynomiale de Kleinjung

Pour le couple (f, g) donné plus haut, on obtient la quantité E de Murphy $1,85 \times 10^{-8}$.

Sélection polynomiale Joux-Lercier

Dans le cas donné pour Joux-Lercier la quantité E de Murphy est $1,43 \times 10^{-6}$.

4 Collecte de relations

Ce qui va nous intéresser ici est de regarder la friabilité de la norme associée à un élément, le but étant de décomposer cette norme en produit d'éléments dont la norme est sous une borne que nous noterons

LPB (Large Prime Bound), autrement dit qu'elle soit LPB-friable. Ceci nous donne la décomposition suivante :

$$Norm\left((a,b)\right) = \prod_{Norm(\mathfrak{q})<LPB} Norm\left(\mathfrak{q}\right)^{e_{\mathfrak{q}}},$$

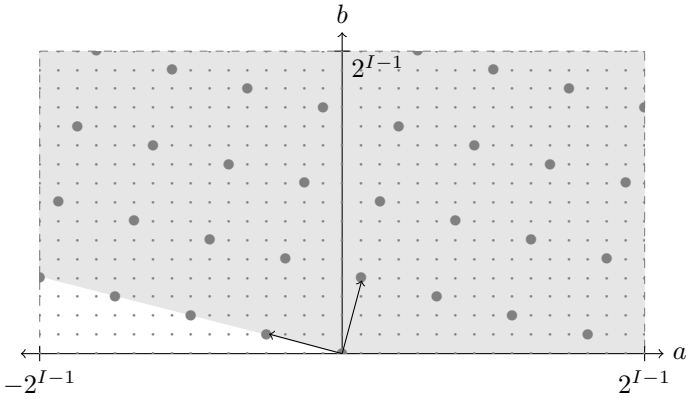
où les $Norm\left(\mathfrak{q}\right)$ sont des nombres premiers et $e_{\mathfrak{q}}$ la multiplicité associée à un élément $\mathfrak{q}$. Les $\mathfrak{q}$ sont des éléments vivant dans $\mathbb{Z}\left[X\right]/f\left(X\right)$, respectivement dans $\mathbb{Z}\left[X\right]/g\left(X\right)$ si on se place de l'autre côté. L'intérêt de décomposer les éléments dans les corps énoncés précédemment est de pouvoir les envoyer maintenant dans $\mathbb{F}_p^*$ à l'aide d'un morphisme, ce qui nous fait d'un côté une décomposition en produit des p_i et de l'autre en produit des q_j, avec p_i et q_j des facteurs premier et i, j leurs indices respectifs. Par construction on a égalité modulo p, d'où la congruence suivante :

$$\prod_i p_i^{e_i} \equiv \prod_j q_j^{d_j}\ [p].$$

Ce que nous nommerons une 'relation' est le passage au logarithme discret après obtention de cette égalité, autrement dit, une relation est de la forme :

$$\sum_i e_i \log_g(p_i) \equiv \sum_j d_j \log_g(q_j)\ [\ell]$$

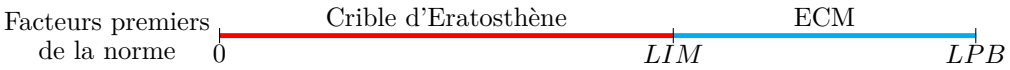
Il nous faudra avant de passer à l'étape suivante un nombre relations qu'on nommera RELS_WANTED. Pour obtenir de telles relations, nous allons utiliser le crible des spécial-$\mathfrak{q}$. Chaque 'spécial-$\mathfrak{q}$' sera ici un facteur premier qu'on impose dans une des deux normes et qui servira d'unité de calcul indépendante. Pour chaque spécial-$\mathfrak{q}$ on va explorer les paires (a, b) dans un espace à deux dimensions de taille $2^I \times 2^{I-1}$ pour I un entier fixé. On commence le crible pour un spécial-$\mathfrak{q}$ premier à partir d'un QMIN fixé, qui doit être assez petit pour obtenir un nombre de relations suffisant pour atteindre RELS_WANTED sans que QMIN ne dépasse LPB, mais il faut que QMIN soit assez grand pour éviter d'avoir trop de relations en double. On illustre par le dessin suivant les paires (a, b) marquées par le spécial-$\mathfrak{q}$ imposé dans la norme.



Pour les paires (a, b) marquées, on va lancer un crible d'Eratosthène, on criblera les normes jusqu'à une limite LIM fixée telle que $LIM < LPB$. Le crible se fait ici en deux dimensions, on se donne alors LIM_0 et LPB_0 pour le côté de g et respectivement LIM_1 et LPB_1 pour le côté de f. On peut ainsi écrire les normes sous la forme :

$$Norm\left((a,b)\right) = \left(\prod_{Norm(\mathfrak{q})<LIM} \mathfrak{q}^{e_{\mathfrak{q}}}\right) \times R,$$

où R est la partie restante composée de facteurs dont les normes sont plus grands que LIM. On se fixe alors ici MFB tel qu'on garde uniquement les paires (a, b) dont la partie restante R est plus petite que MFB, on nommera ces paires les 'survivants'. On applique sur la partie restante R des survivants l'algorithme de factorisation de Lenstra par les courbes elliptiques : ECM (Elliptic-Curve Method de Lenstra [10]) en demandant à ce que la norme des facteurs premiers de R soit inférieure à LPB. Ce qui nous donne la factorisation complète de la norme d'une paire (a, b) de chaque côté. On peut illustrer les étapes qui permettent la décomposition en facteurs premiers des normes et leur ordre de grandeur par le dessin suivant :



Sélection polynomiale de Kleinjung

Pour donner un exemple de relation, ici, le spécial-$\mathfrak{q}$ est 6217, qui est effectivement un nombre premier. Dans cet exemple, on va prendre la paire $(a, b) = (201098, 29)$. On calcule les normes de cette paire pour f et g, ce qui donne :

$$\begin{aligned} bg\left(\tfrac{a}{b}\right) &= 23835716325540980 \\ b^4 f\left(\tfrac{a}{b}\right) &= 510942669265123167970539 \end{aligned}$$

Dans notre exemple les paramètres utilisés sont les suivants :

$$\begin{array}{rl|rl} I = & 11 & & \\ LIM_0 = & 25000 & LIM_1 = & 25000 \\ LPB_0 = & 2^{17} & LPB_1 = & 2^{17} \\ MFB_0 = & 2^{34} & MFB_1 = & 2^{34} \end{array}$$

Pour 510942669265123167970539 on obtient à l'aide du crible la décomposition $3^3 \times 307 \times 1033 \times 6217 \times 11519 \times R_1$ avec $R_1 < MFB_1$, d'où $R_1 = 27893 \times 29873$ par ECM. De même pour $23835716325540980 = 2^2 \times 5 \times 7 \times 17 \times 1831 \times 72551 \times 75391$. On retrouve en effet 6217 dans la décomposition de la norme par f (le spécial-$\mathfrak{q}$ était placé côté 1). Cependant, il n'y a pas ici égalité modulo p ... En effet c'est un peu plus compliqué que la congruence donnée précédemment et l'égalité demande l'intervention de Schirokauer maps.

Sélection polynomiale Joux-Lercier

Je vais aussi donner donner un exemple de relation dans le cas Joux-Lercier où ici les paramètres du crible ne sont pas égaux des deux côtés comme je l'ai choisi pour le cas précédent. Le spécial-$\mathfrak{q} = 6011$ est ici placé du côté 0 (autrement dit côté de g). La paire (a, b) choisie est $(15893, 43196)$, ce qui nous donne les normes suivantes :

$$\begin{aligned} b^2 g\left(\tfrac{a}{b}\right) &= -19893697812756488848513888080 \\ b^3 f\left(\tfrac{a}{b}\right) &= 166644572705832 \end{aligned}$$

Le spécial-$\mathfrak{q}$ est ici placé du côté 0, c'est-à-dire celui de g car dans le cas présent c'est de ce côté qu'on obtient les plus grandes normes, c'est la situation opposée où les normes sont plus grandes du côté f dans la sélection polynomiale par l'algorithme de Kleinjung. Pour obtenir la relation, on utilisera ici les paramètres suivants :

$$\begin{array}{rl|rl} I = & 10 & & \\ LIM_0 = & 250000 & LIM_1 = & 7500 \\ LPB_0 = & 2^{18} & LPB_1 = & 2^{16} \\ MFB_0 = & 2^{18} & MFB_1 = & 2^{32} \end{array}$$

Pour 166644572705832 on travaille côté 1, on obtient par le crible jusqu'à $LIM_1 = 7500$ la décomposition $2^3 \times 3 \times 13 \times 419 \times R_1$ où $R_1 = 1274742769 < MFB_1 = 2^{32}$ et par ECM, $R_1 = 26021 \times 48989$. dans le cas de $-19893697812756488848513888080$, le crible d'Eratosthène donnera la décomposition en facteurs premiers suivante : $2^4 \times 3^3 \times 5 \times 37 \times 137 \times 157 \times 383 \times 443 \times 521 \times 1931 \times 6011 \times 11279$.

5 Schirokauer maps

Pour expliquer le problème observé dans l'exemple précédent il faut comprendre que la congruence énoncée est vraie uniquement si les $\mathbb{Z}[X]/f(X)$ et $\mathbb{Z}[X]/g(X)$ étaient des anneaux dans lesquels le théorème de factorisation unique était vérifié. Or en général ce n'est pas le cas ce qui nous oblige à utiliser

les 'logarithmes virtuels' définis à l'aide des Schirokauer maps [12]. Ce qui transforme une relation définie précédemment pour une paire (a, b) comme :

$$\sum_i e_i \log_g(p_i) - \sum_j d_j \log_g(q_j) \equiv 0 \,[\ell]$$

en une relation de la forme :

$$\sum_i e_i \mathrm{vlog}_g(\mathfrak{p}_i) - \sum_j d_j \mathrm{vlog}_g(\mathfrak{q}_j) + \sum_k SM_k\,(a,b)\,\mathrm{vlog}_g(SM_k) \equiv 0 \,[\ell]$$

Dans cette équation, les $\mathfrak{p}_i$, $\mathfrak{q}_j$ sont des idéaux et il est possible de calculer les $SM_k\,(a,b)$ avant l'étape d'algèbre linéaire. Il est de même possible de repasser du logarithme virtuel au logarithme discret, mais les notions abordées dans cette section ne sont pas à la portée de mes connaissances mathématiques actuelles pour que je puisse les définir proprement, il faut simplement comprendre ici que ça ne pose pas de problème pour être calculé. Pour connaître le nombre de Schirokauer maps, on utilise le théorème des unités de Dirichlet qui nous donne que pour un polynôme ayant $r_{\mathbb{R}}$ racines réelles et $r_{\mathbb{C}}$ racines complexes, le nombre de Schirokauer maps à utiliser est $r_{\mathbb{R}} + \frac{1}{2} r_{\mathbb{C}} - 1$. C'est ensuite ces équations qui seront envoyées à l'algèbre linéaire pour résoudre les inconnues qui sont ici les logarithmes vituels des $\mathfrak{p}_i$, $\mathfrak{q}_j$ et SM_k.

6 Filtre

On obtient une matrice où les différents logarithmes étiquettent les colonnes et les relations les lignes de cette matrice. Dans un premier temps on regarde les logarithmes des idéaux qui n'apparaissent qu'une seule fois pour les garder de côté et exprimer leur valeur en fonction des autres éléments de la ligne dont il est combinaison linéaire. Puis on enlève de la matrice les relations qui existent en plusieurs exemplaires pour obtenir une matrice de relations uniques, s'il manque trop de relations après suppression des doublons, on relance la collecte de relations à partir de cette étape. Pour se donner une estimation du nombre de relations uniques attendu, si $\pi : \mathbb{N} \longmapsto \mathbb{N}$ est la fonction qui compte les nombres premiers en dessous de la variable entière donnée en paramètre, alors on s'attend à avoir un nombre de relations uniques de l'ordre de $\pi(LPB_0) + \pi(LPB_1)$ pour LPB adapté.

Sélection polynomiale de Kleinjung

Dans notre exemple nous avions fixé les paramètres suivants :

$$\begin{aligned} \text{QMIN} &= 10000 \\ \text{RELS\_WANTED} &= 32000 \end{aligned}$$

CADO-NFS nous a obtenu 32765 relations dont 26996 étaient uniques. Pour rappel nous nous étions donné ici $(LPB_0, LPB_1) = \left(2^{17}, 2^{17}\right)$, d'où $\pi(LPB_0) + \pi(LPB_1) = 24502$. On a donc assez de relations uniques pour continuer. Obtenir un peu plus de relations uniques peut bien souvent nous faire gagner du temps dans l'algèbre linéaire, le but étant de trouver un compromis entre le temps passé dans la collecte de relations et l'algèbre linéaire pour gagner sur le temps total. Le dernier spécial-$\mathfrak{q}$ utilisé ici était 19997.

On profite du fait que la matrice obtenue est assez creuse pour éliminer des coefficients sur les premières colonnes par un pivot de Gauss jusqu'à un certain seuil. De plus les coefficients de la matrice sont les exposants de la décomposition en facteurs premiers des éléments des relations obtenues donc bien souvent de petits éléments. Il est aussi possible de réarranger les colonnes de la matrice si un pivot est plus intéressant ou est sur une colonne avant un plus petit nombre de coefficients. Le but du filtre est d'obtenir une matrice de la forme :

$$\begin{pmatrix} A & B \\ 0 & C \end{pmatrix}$$

où A est une matrice échelonnée. On effectue cette étape de filtre souvent jusqu'à une densité cible, car l'élimination gaussienne fabrique une matrice C de plus petite taille mais plus dense au cours des itérations. C'est cette matrice C dont on va calculer le noyau par une méthode itérative par la suite.

7 Algèbre linéaire

La méthode itérative utilisée ici est celle de Block-Wiedemann [4]. Dans un premier temps nous parlerons dans le cas de la méthode de Wiedemann simple. Soit M la matrice dont on veut calculer le noyau. L'objectif ici est de reconstruire le polynôme minimal χ_M de la matrice M dont les coefficients sont les m_i pour $i \in [\![1, n]\!]$:

$$\chi_M(x) = \sum_{i=1}^{n} m_i x^i,$$

où l'on pose n le rang de la matrice M. Pour des vecteurs x et y choisis aléatoirements, on calcule les scalaires :

$$\lambda_i = x^T M^i y \quad pour \quad i \in [\![1, 2n]\!].$$

Pour reconstruire les coefficients m_i, on utilise l'algorithme de Berlekamp–Massey ou l'algorithme d'Euclide arrêté au milieu.

On détermine $k \in \mathbb{N}$ la multiplicité du facteur X dans $\chi_M(X)$, ce qui nous donne P un polynôme tel que :

$$\chi_M(X) = X^k P(X).$$

On fabrique un vecteur v aléatoire tel que $w = P(M)v$ soit non nul. On calcule les $M^i w$ jusqu'à ce que $M^i w = 0$, ce qui nous donne $M^{i-1}w$ un vecteur de $Ker(M)$. La grande modification de la méthode de Wiedemann en la méthode du Block-Wiedemann est que au lieu de choisir x, y des vecteurs aléatoires, on prend X, Y des blocs de vecteurs, ce qui permet une parallélisation du calcul.

7.1 Reconstruction des logarithmes

La reconstruction des logarithmes consiste à reprendre les singletons mis de côté lors de la première étape de l'algèbre linéaire et calculer leur valeur, ce qui nous donne des colonnes de la forme suivante :

$$\begin{pmatrix} & & & & 0 & & & & \\ & & & & \vdots & & & & \\ & & & & 0 & & & & \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & \mathbf{1} & \cdots & SM_{n,1} & \cdots & SM_{n,S} \\ & & & & 0 & & & & \\ & & & & \vdots & & & & \\ & & & & 0 & & & & \end{pmatrix}$$

Il suffit ensuite de lire la combinaison linéaire sur la ligne et de retrouver le logarithme de l'élement associé à la colonne en fonction des logarithmes connus et déjà calculés par les étapes précédentes de l'algèbre linéaire ainsi que des valeurs pour les Schirokauer maps.

8 Logarithme individuel

La reconstruction des logarithmes donne un pourcentage de logarithmes connus pour les bornes de friabilités LPB fixées mais tous les logarithmes ne sont pas connus, nous les calculerons alors si besoin dans l'étape du logarithme individuel. Nous noterons h l'élement de $\langle g \rangle$ dont on veut $\log_g(h)$.

8.1 Amorce

On prend un élément $e \in \mathbb{F}_\ell$ aléatoire. On calcule $z \equiv h^e \,[p]$. Par passage au logarithme discret, on a que $\log_g(z) \equiv e \log_g(h) \,[l]$, donc en connaissant $\log_g(z)$ on pourra facilement retrouver $\log_g(h)$ par le calcul suivant : $\log_g(h) \equiv e^{-1} \log_g(z) \,[l]$.

Puis, on déroule les étapes de l'algorithme d'Euclide étendu comme si nous souhaitions calculer le PGCD de z et p. La différence est que la condition d'arrêt ici n'est pas d'être arrivé au PGCD, mais d'avoir u, v, w tels que pour $vz + wp = u$, on aurait u et v assez proches de $\sqrt{p}$. L'idée étant que :

$vz = u - wp \Rightarrow vz \equiv u\,[p]$. Ce qui nous donne que : $z \equiv uv^{-1}\,[p]$. Et par passage au logarithme discret dans $\mathbb{F}_\ell$:

$$\log_g(z) \equiv \log_g(u) - \log_g(v)\,[l]\,.$$

On aimerait si possible que u et v soient LPB-friables, dans l'idée que u et v soient produits de facteurs premiers pas trop grands, on gardera le meilleur triplet (z, u, v) pour l'envoyer dans l'étape suivante.

8.2 Descente

On décompose u est v en produit de facteurs premiers :

$$u = \prod_{q \in \mathbb{P}} q^{v_u(q)},$$

où $v_u(q)$ est la valuation de q dans u. Dans le cas où le logarithme d'un des facteurs q n'est pas connu, on va procéder à la construction d'un arbre de descente. On utilise le même procédé que lors de la collecte de relations avec des paramètres donnés par une «table de hint» pour produire une relation qui implique q et d'autres éléments dont la friabilité est donnée par la ligne de la table en question. Dans cette table, on trouve des lignes donnant les paramètres à utiliser pour un nombre de bits donné (la taille de q) et le côté duquel est issu l'élement q. On obtient la relation :

$$\log_g(q) \equiv \sum_i \log_g(e_i)\,[l]\,.$$

Pour chaque e_i, dans le cas où $\log_g(e_i)$ n'est pas connu, on recommence récursivement sur le e_i courant. Le but étant à la fin d'evaluer récursivement les logarithmes inconnus de notre arbre à jusqu'à ce que les éléments soient assez petits pour exister dans la base de données calculée lors de l'étape de précalculs.

Sélection polynomiale de Kleinjung

Pour illustrer par un exemple la construction de l'arbre donné précédemment, on se donne par exemple la cible suivante :
h = 219175094069146219651006701222016123245631811655897378952869
La relation de départ est ici $2^3 \times 19 \times 23 \times 331 \times 449 \times 23819 \times 23857 \times 25097 \times 56101 \times 70823 \times 3950159 \equiv 41 \times 47 \times 53 \times 59 \times 89 \times 251 \times 277 \times 5479 \times 26706991 \times 35210641\,[p]$. Dans cette relation, les logarithmes inconnus sont ceux de 3950159 (22 bits), 26706991 (25 bits) et 35210641 (26 bits), il faudra donc rechercher récursivement des relations permettant de résoudre ces logarithmes. Par exemple pour le cas de 3950159, nous allons obtenir 22 relations impliquant 3950159, nous garderons la relation où il y a le moins de logarithmes récursifs à faire et où ceux nécessitant un appel récursif soient de taille la plus petite possible, la relation gardée est $3 \times 5^3 \times 929 \times 155119 \times 3950159 \equiv 2^4 \times 7 \times 13 \times 719^2 \times 1249 \times 2207 \times 11483 \times 55889 \times 66701\,[p]$ qui implique en effet 3950159 et ayant pour autre logarithme inconnu 155119 (18 bits). Récursivement on obtient 5 relations impliquant 155119, on conserve la relation : $7 \times 701 \times 21001 \times 28111 \times 155119 \equiv 2^4 \times 3^2 \times 5 \times 11 \times 19 \times 23^2 \times 29 \times 179 \times 1051 \times 1571 \times 3967 \times 64891 \times 123377\,[p]$ qui nous permet de calculer $\log(155119) \equiv \log(7) + \log(701) + \ldots + \log(28111) + 4\log(2) + \ldots + \log(123377)\,[p]$. En appliquant le même procédé sur 26706991 et 35210641, on obtient l'arbre en figure 2.

Les logarithmes ainsi obtenus sont les logarithmes virtuels des éléments, on en déduit alors leur valeur à l'aide des Schirokauer maps. Ce qui nous donne finalement $\log_\gamma(h)$:

$$174067338462612790164161047638630225900475287582008197949712.$$

On peut vérifier le résultat obtenu par exemple ici avec $\log_\gamma(2)$ par la formule $\log_2(h) = \log_\gamma(h) \times \log_\gamma(2)^{-1} \bmod \ell$ ce qui nous donne :

$$204278518695979584507817934398576469694715999872508058179109.$$

On vérifie que $2^{204278518695979584507817934398576469694715999872508058179109} \equiv 219175094069146219651006701222016123245631811655897378952869\,[p]$.

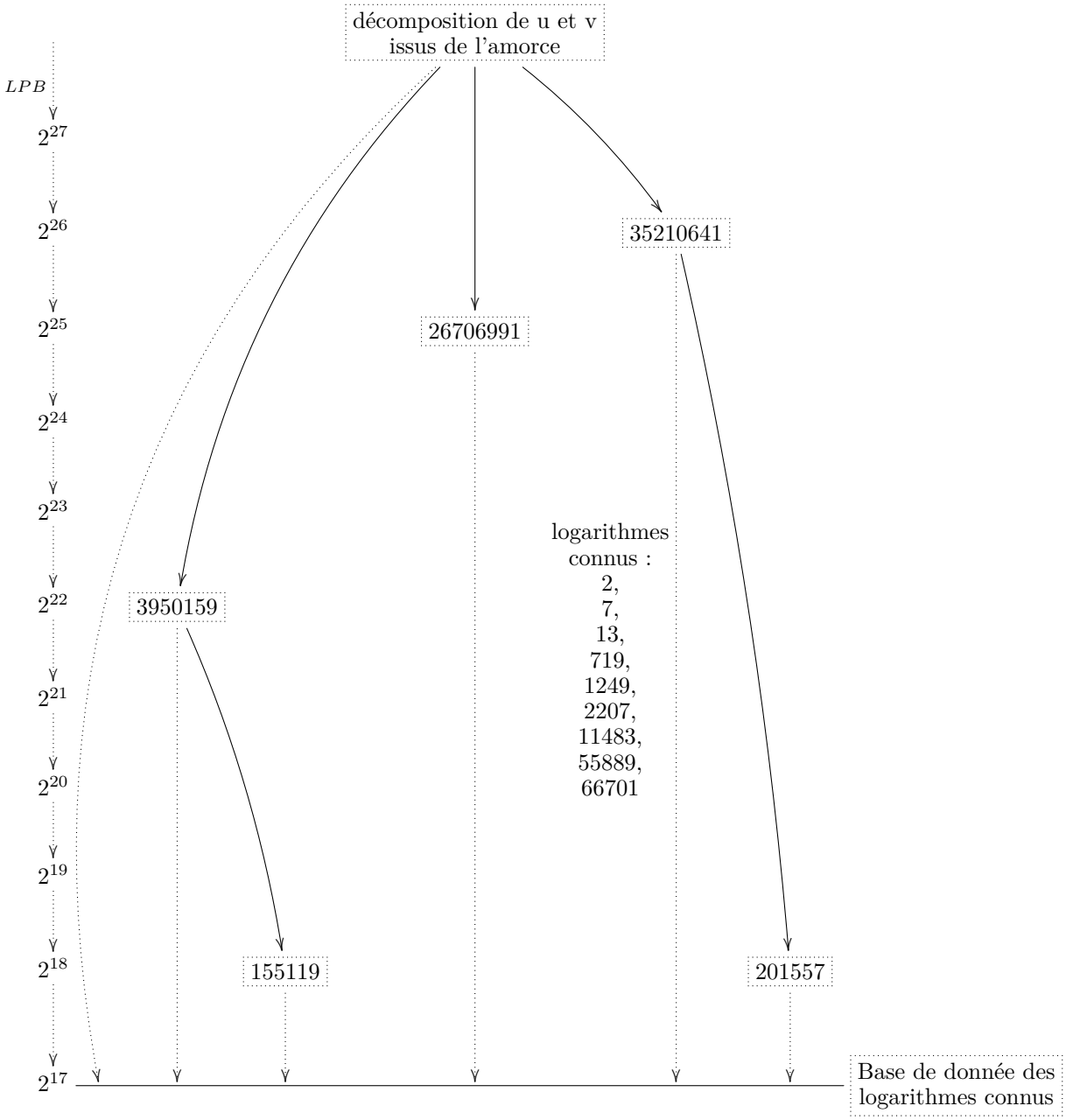


FIGURE 2 – Exemple d'arbre de descente

Maintenant que les précalculs sont effectués, on peut maintenant obtenir les logarithmes discrets dans $\mathbb{F}_p^*$ en temps négligeable devant le temps des précalculs. Par exemple, si maintenant on se donne de calculer le logarithme en base $g = 42$ (d'ordre ℓ) de $h = 14947290668168544889528743687824309887219484884164004857477 3$, CADO-NFS nous donne en base inconnue $\log_\gamma(h) = 3153136370026252446542132586977885825531469447096887833187 3$. Puis on peut calculer $\log_\gamma(42) = 5762007178577009224638514341977592408760297849239981943933 9$. Ce qui nous donne finalement $\log_{42}(h) = \log_\gamma(h) \times \log_\gamma(42)^{-1} \bmod \ell = 1337$.

Troisième partie

Recherche de paramètres pour CADO-NFS par l'expérimentation

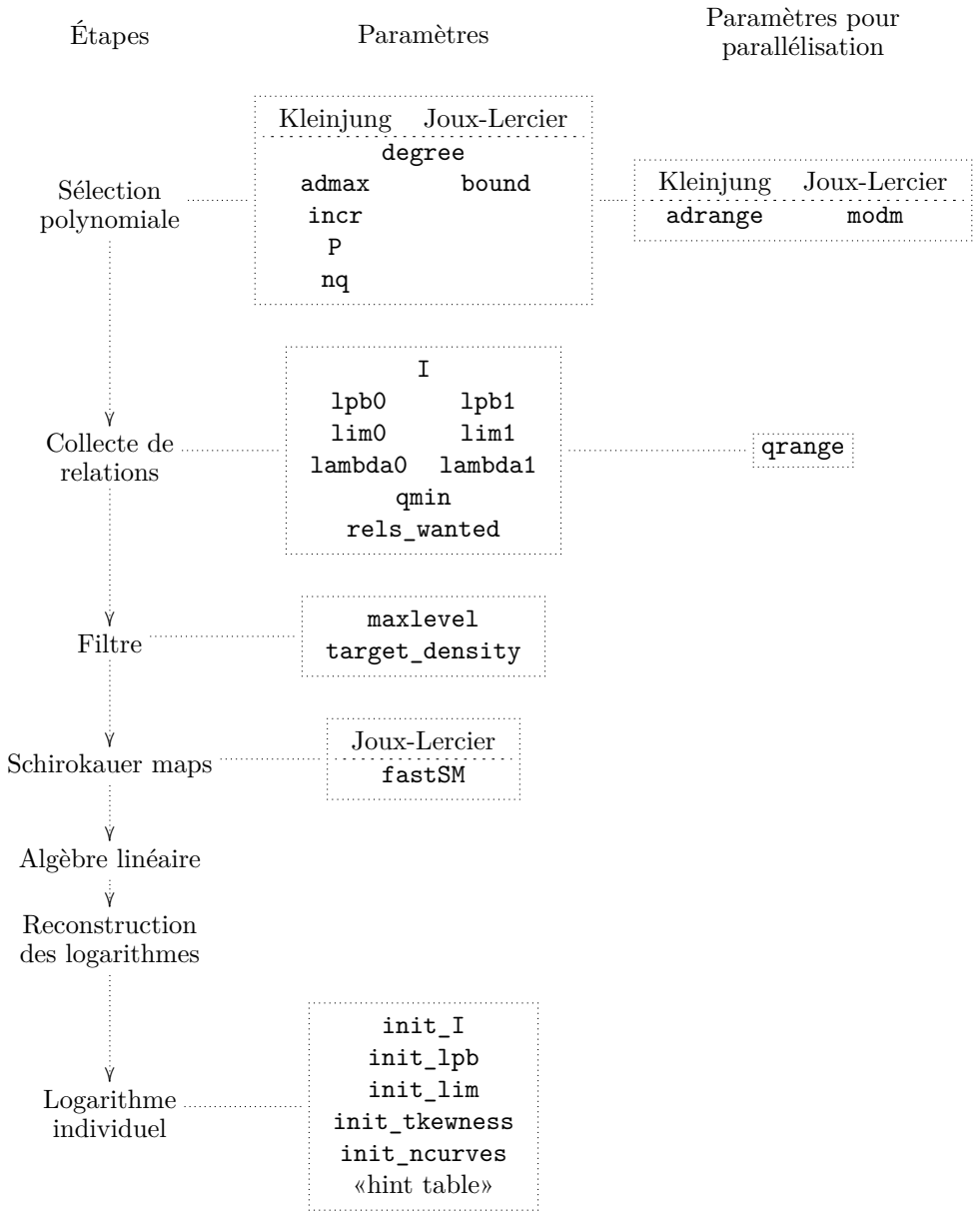


FIGURE 3 – Principaux paramètres des étapes de CADO-NFS

Pour le calcul du logarithme discret dans un corps fini $\mathbb{F}_p^*$, il est possible de paramétrer CADO-NFS pour une taille de p donnée avec un jeu de paramètres adapté. Pour donner une idée des possiblités de paramétrage des étapes énoncées précédemment, nous donnerons en figure 3 les principaux paramètres utilisés pour créer nos fichiers de paramètres. Nous remarquons qu'il est principalement possible de paramétrer les premières étapes du crible algébrique et le choix de ces paramètres aura un impact sur les temps passés dans les étapes suivantes comme par exemple l'algèbre linéaire.

Ce qui va nous intéresser ici est de trouver une combinaison de paramètres permettant de minimiser le temps CPU total passé d'une part dans la phase de précalculs puis d'autre part pour le calcul individuel des logarithmes discrets. La difficulté pour créer un fichier de paramètres pour les petites tailles de p est qu'il n'existe pas de conjecture ou formule donnant le meilleur jeu de paramètres, de même nous pourrons observer que certains phénomènes négligeables pour les grandes tailles ont un impact plus important dans le cas des petites tailles. Pour chercher un jeu de paramètres intéressant dans notre cas l'idée est donc de les trouver par l'expérimentation.

Une approche naïve serait de chercher le jeu de paramètres donnant le meilleur temps en testant un grand nombre de combinaisons, le problème posé par cette approche est que le nombre d'essais à réaliser serait bien trop important et que chaque essai demande un temps non négligeable à partir d'une certaine taille de p. Par exemple pour p de taille 60 on peut attendre une minute, et pour p de taille 100 attendre de l'ordre de 2 heures par calcul.

Une autre approche serait d'utiliser un outil automatique pour explorer un espace à autant de dimensions qu'on a de paramètres et suivre ce qui semble être la meilleure direction par un calcul de gradient comme le fait OPAL. L'idée est intéressante mais le problème ici est que l'espace à explorer est bien trop grand, il faudrait donc essayer de trouver un moyen de réduire l'espace des paramètres à explorer et le temps d'un essai pour utiliser ce type d'outil.

L'approche et les idées que je proposerai dans la suite demandent d'isoler les différentes étapes de NFS pour s'intéresser aux résultats de ces étapes qui nous donnerons une idée de la pertinence des paramètres choisis pour la suite et demande une compréhension un peu plus fine de NFS que les approches énoncées plus haut.

Pour illustrer la démarche que je propose, nous utiliserons les méthodes suivantes pour fabriquer un fichier de paramètres pour le cas de la sélection polynomiale de Kleinjung et celle de Joux-Lercier pour des tailles de 70 chiffres. Ce qui va nous intéresser ici est de fabriquer un fichier de paramètres qui minimise le temps total pour des p premiers de Sophie Germain aléatoires de taille en nombre de chiffres dans l'intervalle $[\![67, 72]\!]$ dans l'idée ici de créer des fichiers de paramètres pour des tailles allant de 5 en 5. La difficulté sera donc de faire en sorte que le fichier ne soit pas local à un seul p et qu'il supporte la limite supérieure des p qu'il est possible de prendre pour ce jeu de paramètres.

Nous noterons par la suite $\mathbb{S}_i$ l'ensemble des nombres premiers de Sophie Germain de taille i chiffres et $\mathbb{S}_{[\![i,j]\!]}$ l'ensemble des nombres de Sophie Germain de taille de chiffre dans $[\![i, j]\!]$. En annexe, je propose les fichiers pour les tailles de 70 chiffres obtenus pour illustrer les fichiers de paramètres manipulés dans les explications suivantes.

9 Sélection polynomiale

Dans cette phase de sélection polynomiale, nous avons vu précédemment qu'un critère de qualité pour sélectionner les polynômes f et g était la quantité E de Murphy, c'est principalement sur cette quantité que nous allons baser la recherche de nos paramètres. Il est possible de trouver dans un fichier '.poly' la quantité E de Murphy dans le répertoire de travail utilisé par CADO-NFS une fois la sélection polynomiale effectuée.

On se donne l'exemple suivant :

$$\begin{aligned} \mathrm{p} &= 57473963025992479514446886943836367361516017867286106636972973169733 03 \\ \ell &= 28736981512996239757223443471918183680758008933643053318486486 58486651 \end{aligned}$$

On lance CADO-NFS avec la commande suivante :

```
./cado-nfs.py -dlp -ell 28736981512996239757223443471918183680758008933643053318486486
58486651 57473963025992479514446886943836367361516017867286106636972973169733 03
```

Pour connaître le répertoire de travail il faudra lire au début de la sortie d'erreur la ligne suivante :

```
Info:root: Created temporary directory /tmp/cado.8sl54kjp
```

Ensuite, nous pourrons trouver les informations qui nous intéressent ici sur la sélection polynomiale dans le fichier `/tmp/cado.8sl54kjp/p70.poly` :

```
# MurphyE (Bf=5.000e+04,Bg=5.000e+04,area=1.049e+11) = 1.04e-08
# f(x) = 1320*x^4-52502866*x^3+566046845749*x^2+287154422549494*x-336534450152661457
# g(x) = 3601816737557*x-81721979237119830
```

Il est tout à fait possible de regarder ce fichier une fois la sélection polynomiale terminée sans avoir à attendre la fin de l'execution de `cado-nfs.py`. C'est d'ailleurs ce que nous ferons pour paramétrer la sélection polynomiale.

En premier lieu il faut choisir un degré pour notre polynôme f (et g dans le cas Joux-Lercier). Cependant pour les petites tailles la quantité E de Murphy n'est pas le seul critère de sélection, car par exemple dans le cas du passage du degré 3 au degré 4, le temps passé dans la sélection polynomiale occupe une place importante dans le temps total ce qui fait que ce temps est aussi à prendre en compte et on peut à ce moment opter pour un degré plus petit même s'il sera plus difficile de régler les étapes suivantes pour gagner sur le temps total des précalculs. Cette difficulté était posée lors du paramétrage des tailles de 40 et 45 chiffres dans le cas Kleinjung ce qui me demandait dans ce cas de plutôt s'intéresser au temps total qu'à la quantité de Murphy. Pour le cas Joux-Lercier garder le degré 3 pour f (implicitement ici 2 pour g) me semblait le plus pertinent pour les tailles de 30 à 100 chiffres.

Sélection polynomiale de Kleinjung

Dans cette section nous allons chercher à paramétrer les valeurs de `nq`, `P` (plus précisément : `tasks.polyselect.P`) et `admax`. Je ne pense pas que ma méthode soit optimale car ici très approximative, mais l'idée est la suivante :

- Trouver `nq` petit qui est le nombre maximum de spécial-`q` à utiliser pour la sélection polynomiale tel que la quantité E de Murphy semble maximal et ne plus progresser.
- Trouver `P` qui impose à g deux facteurs premiers dans $[\![P, 2P]\!]$, ici nous allons chercher l'intervalle qui permet de maximiser la quantité E de Murphy.
- Ensuite nous chercherons à réduire `admax` de sorte à réduire le temps passé dans la sélection polynomiale mais en gardant une bonne quantité E de Murphy, c'est ici que se font les compromis entre le temps et la quantité de Murphy.

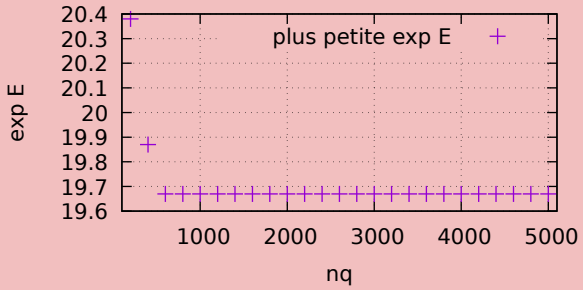
Ici, je propose de lancer CADO-NFS pour qu'il nous fabrique un répertoire de travail, puis de prendre la première ligne du fichier `p70.wucmd` dans mon cas par exemple :

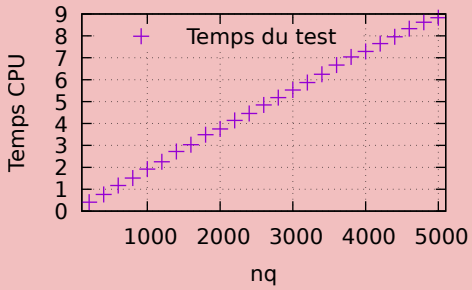
```
/users/ktrancho/Documents/cado_norm/cado-nfs/build/all/polyselect/polyselect -P 1000 -N
 57473963025992479514446886943836367361516017867286106636972973169733O3 -degree 4 -t 2
 -admin 0 -admax 2000 -incr 60 -nq 1000
```

Maintenant nous allons modifier les valeurs pour `P`, `nq` et `admax`. Les résultats qui vont m'intéresser ici sont ceux obtenus à la ligne suivante (dans les dernières lignes de l'output) :

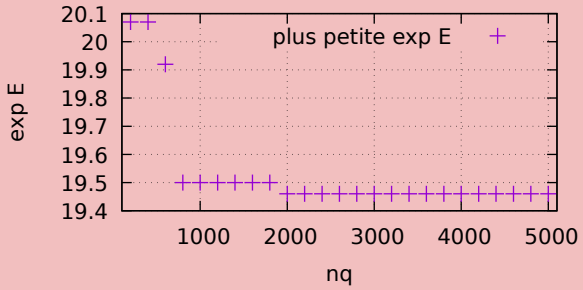
```
# Stat: best exp_E after size optimization: 19.50 19.57 19.64 19.64 19.97 19.99 20.03
20.11 20.14 20.14
```

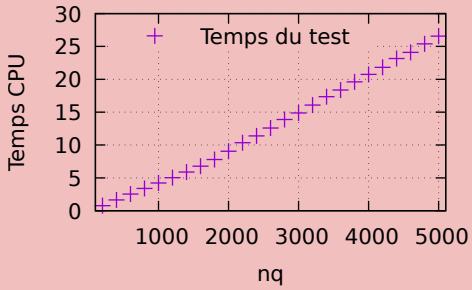
Le but du jeu est de faire en sorte que la première valeur soit la plus petite possible. Une autre contrainte est d'éviter que le temps passé pour ce résultat soit trop grand, je propose donc aussi de regarder à côté le temps demandé par l'unité de calcul. Ce que je propose ici est de se fixer `admax` pas trop petit et de rechercher pour différents `P` le plus petit `nq` tel que les résultats ne semblent plus progresser. Pour illustrer mon hypothèse qui dit qu'il existe un seuil pour lequel augmenter `nq` ne donne pas de meilleurs résultats sur la qualité du polynôme obtenu, je propose le graphique suivant pour lequel j'ai fixé `P` = 2000 et `admax` = 2000 :



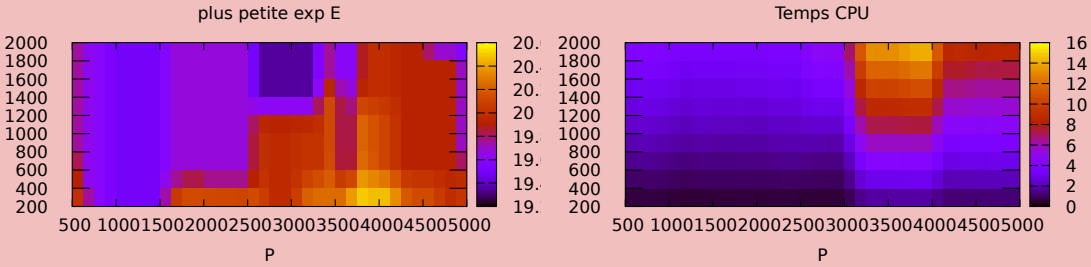


On prend ici le plus petit `nq` dans l'idée de passer le moins de temps pour obtenir ce résultat. A noter que ici le `nq` est local à `P` = 2000 et `admax` = 2000, ce qui ne fait pas pour autant de `nq` = 600 le choix idéal pour la suite, en effet par exemple pour `P` = 5000 et `admax` = 2000, j'obtiens les graphiques suivants :

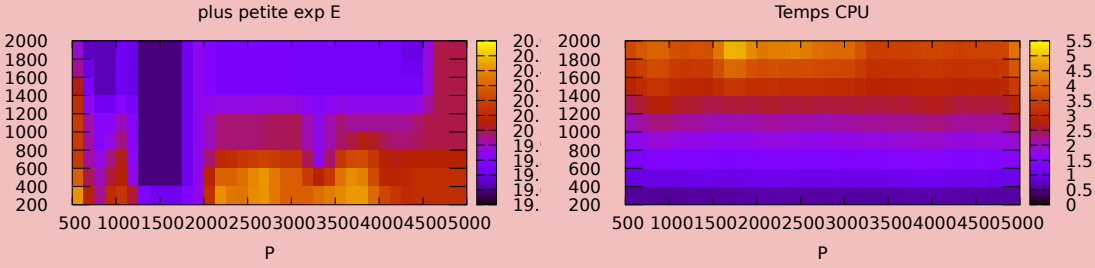




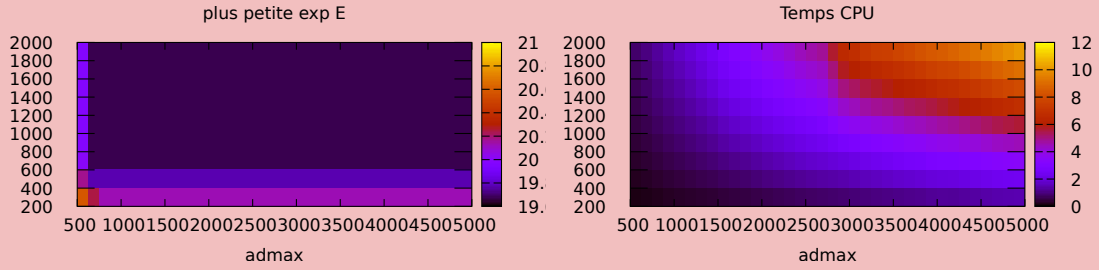
Ce que je propose ici est de figer `admax` = 2000 car sa principale influence sera sur le temps passé dans la sélection polynomiale. Dans les graphiques suivants je prends `P` de 500 à 5000 et `nq` de 200 à 2000 :



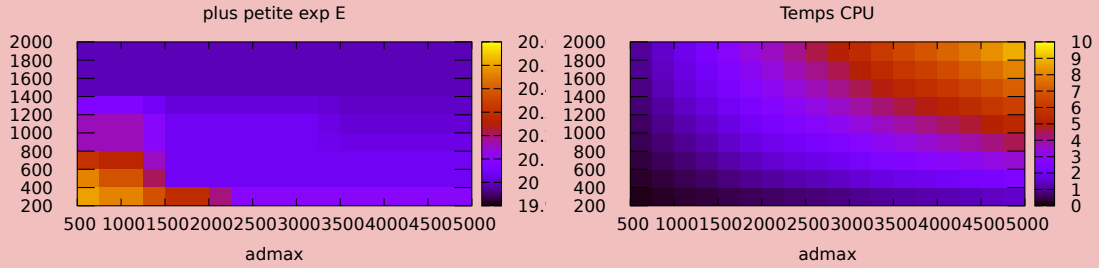
A la vue de ces résultats nous pourrions vouloir prendre `P` = 2750 par exemple. Mais je propose de changer de p en prennant par exemple 6360075255930878030260499861562198834880471095298813076436732203276087, ce qui nous donne les résultats suivants où `P` = 1500 semble plus intéressant :



Je n'ai pas d'idée précise sur les critères ou la manière de choisir `P`, j'essaie donc de le prendre de sorte qu'il permette malgré tout de trouver des polynômes acceptables pour la suite. Ici j'ai choisi $\texttt{P} = 2000$. Mon idée est ensuite de chercher `admax` et `nq` tels qu'ils produisent un couple de polynômes intéressants en un temps raisonnable pour un grand nombre de cas possibles. En regardant ici les résultats obtenus en faisant évoluer `admax` et `nq`, on observe qu'il peut falloir attendre beaucoup plus de temps pour obtenir un résultat un peu meilleur :



Si nous fixions nos paramètres tels qu'ils soient minimaux pour un exemple particulier, ils seraient par exemple moins pertinents pour le cas suivant où $p = 68860763716308226660669287228037657383394599762302737918244494010843 27$:

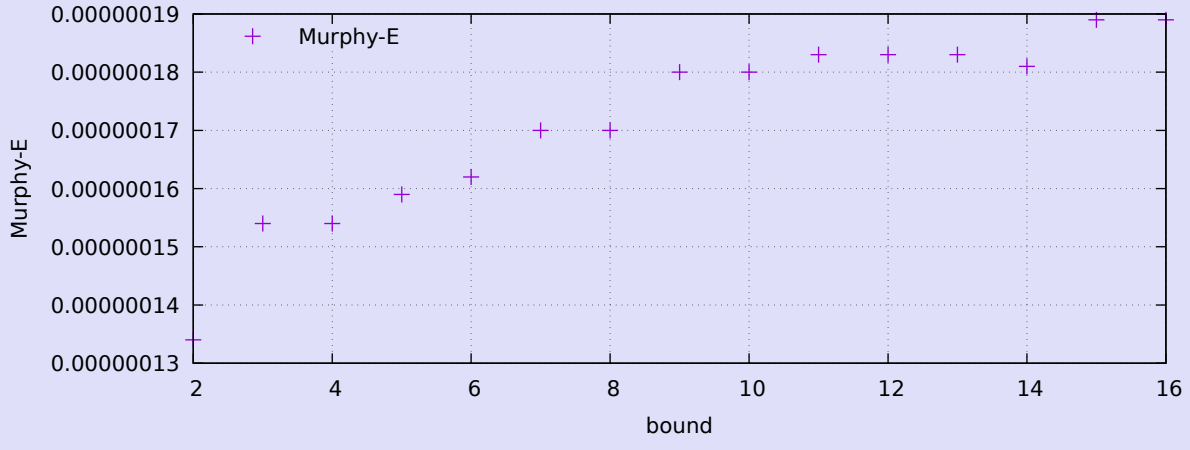


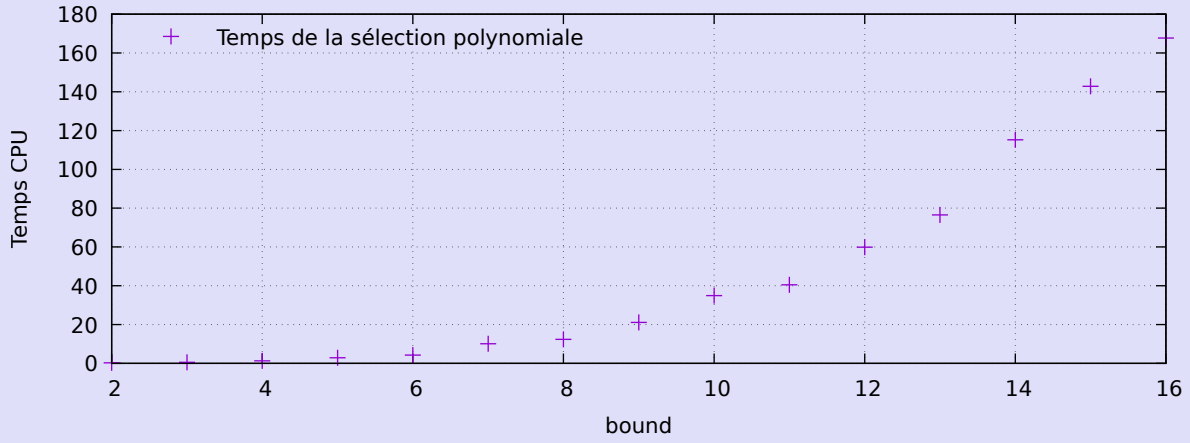
Etant donné qu'ici nous travaillons sur des petites tailles, le temps supplémentaire passé à rechercher des meilleurs polynômes compte aussi dans le temps total. Ce qui fait que même si on demande à passer beaucoup plus de temps à rechercher des polynômes il se peut que cela améliore peu le temps passé dans la collecte des relations par rapport au temps perdu à le trouver. Dans l'idée de garder `admax` et `nq` assez grands pour d'autres p de taille 70, j'ai choisi de conserver $\texttt{admax} = 2000$ et $\texttt{nq} = 1000$.

Sélection polynomiale Joux-Lercier

Pour la sélection polynomiale Joux-Lercier, nous allons devoir paramétrer `bound` et `modm`. La contrainte sur `modm` ici est que `modm` et $2 \cdot \texttt{bound}(2 \cdot \texttt{bound} + 1)(\texttt{bound} + 1)$ sont premiers entre eux. Ce qui rend son paramétrage beaucoup plus simple.

`modm` est principalement un paramètre de parallélisation qui définit le nombre de processus à lancer pour la sélection polynomiale Joux-Lercier. Etant donné que nous travaillons ici sur des petites tailles, j'ai choisi de le prendre petit. La valeur à laquelle je m'intéresse est donc `bound`, mon idée est de trouver `bound` assez petit pour ne pas passer trop de temps à rechercher les polynômes mais assez grand pour donner l'impression que la quantité-E de Murphy ne semble plus ou très peu progresser.

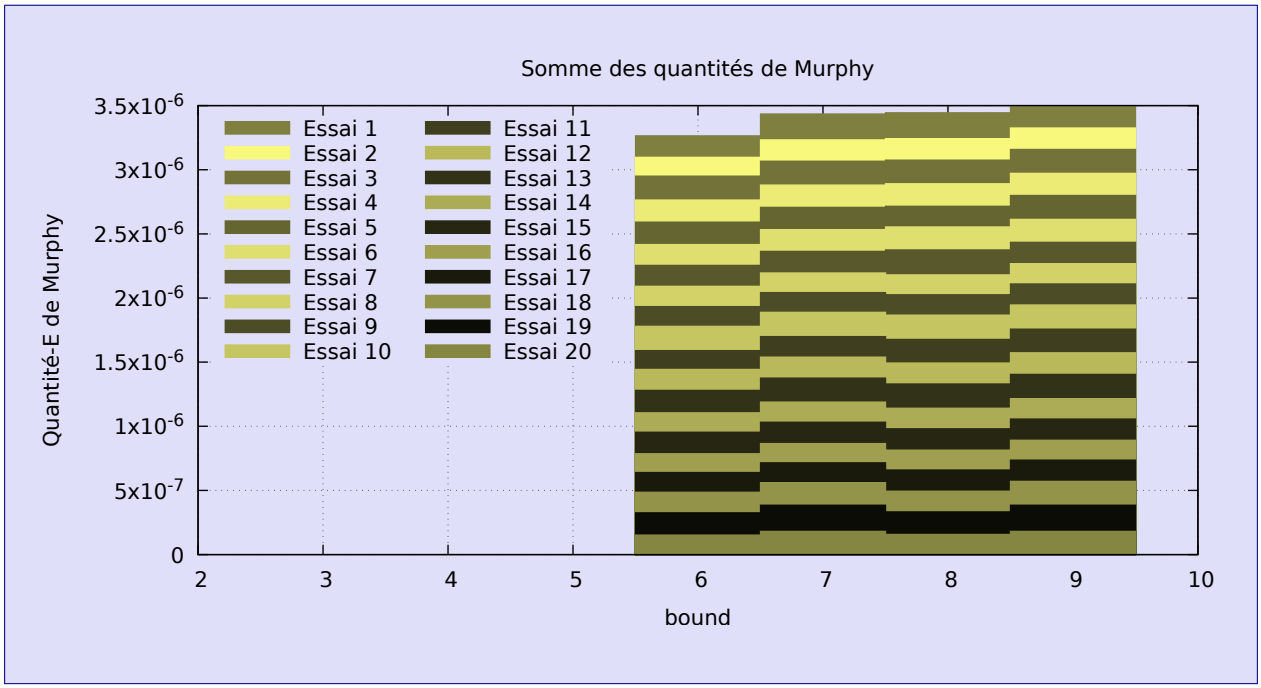




À la vue de ces résultats, une valeur intéressante pour `bound` me semble être 7. Pour confirmer mon hypothèse, je vais regarder le temps CPU total pour les valeurs de `bound` suivantes :

`bound`	6	7	9
`modm`	5	11	7
Temps CPU (en secondes)	197.07	178.44	178.22

Il semble en effet que la qualité des polynômes pour `bound` $= 6$ par exemple ne soit pas pertinente ici. Bien que `bound` $= 9$ donne une quantité-E de Murphy bien supérieure à `bound` $= 7$, le fait de passer 2 fois plus de temps dans la sélection polynomiale ne fait pas gagner beaucoup plus sur le temps CPU total. Mais les temps CPU étant équivalents, il n'est pas possible de trancher ici sur le `bound` à conserver. L'idée est donc de faire plus de tests sur des exemples différents pour déterminer quelle valeur est la plus pertinente et si en effet on retrouve des quantités de Murphy bien plus intéressantes pour `bound` $= 9$. Pour `bound` entre 6 et 9, je regarde les résultats de la sélection polynomiale pour 20 tests sur $p_i \in \mathbb{S}_{70}$. Finalement je garderai `bound` $= 7$ et `modm` $= 11$ car comme le graphique suivant le représente : la différence des quantités de Murphy entre `bound` $= 7$ et 9 n'est pas aussi significative qu'elle l'était pour l'exemple précédent.



10 Collecte de relations

C'est principalement le paramétrage de la collecte de relations qui va influencer par la même occasion le temps passé dans l'algèbre linéaire et ainsi majoritairement le temps CPU total. Dans un premier temps nous allons chercher à optimiser la partie de la collecte de relations qui consiste à trouver un bon nombre de relations en un temps raisonnable. Pour cela nous allons chercher à régler `lpb`, `lim` et `mfb`. Ensuite nous allons chercher à régler la collecte des relations dans le but de minimiser le temps passé dans celle-ci puis dans l'algèbre linéaire, pour ceci nous jouerons principalement sur les valeurs de `qmin` et `rels_wanted`.

10.1 Optimisation du crible

Pour optimiser la collecte de relations nous allons effectuer des essais sur `las` dans le but de paramétrer `lpb`, `lim` et `mfb`. Dans cette étape de paramétrage on va se donner un p aléatoire dans $\mathbb{S}_{70}$, il sera tout à fait possible de refaire cette étape avec différents $p \in \mathbb{S}_{70}$ pour vérifier la pertinence des paramètres obtenus. En effet le but est que le jeu de paramètres ne soit pas optimisé que pour un seul p arbitraire. Pour obtenir une unité de travail pour `las`, nous allons lancer CADO-NFS sur un exemple, attendre la fin de la sélection polynomiale seulement et interrompre l'execution de CADO-NFS une fois la collecte de relations lancée (Une fois arrivé à l'execution dans la phase de `Lattice Sieving`). Ensuite nous allons chercher nos unités de travail pour `las` dans le fichier `.wucmd`. Par exemple :

```
/users/ktrancho/Documents/cado_norm/cado-nfs/build/all/sieve/las -I 11 -poly /tmp/cado
.8sl54kjp/p70.poly -q0 20000 -q1 25000 -lim0 50000 -lim1 50000 -lpb0 18 -lpb1 18 -mfb0
 36 -mfb1 36 -lambda0 2.2 -lambda1 2.2 -fb /tmp/cado.8sl54kjp/p70.roots.gz -out /tmp/c
ado.8sl54kjp/p70.20000-25000.gz -t 2 -stats-stderr
```

Le resultat qui va nous intéresser se trouve à la dernière ligne de l'execution, par exemple :

```
# Total 9054 reports [0.00128s/r, 18.9r/sq] in 13.6 elapsed s [85.2% CPU]
```

Pour un test on peut faire varier `lpb`, `lim` et `mfb`, sous les constraintes que `mfb` est un multiple de `lpb` et qu'on donne `lambda` comme étant un peu plus grand que le facteur entre `mfb` et `lpb`, ici $\texttt{mfb} = 2 \times \texttt{lpb}$ et $\texttt{lambda} = 2.2$. Je propose à cette étape de calculer un 'score' pour chaque test réalisé, donné par la

formule suivante :

$$\frac{relations\ par\ spécial - \mathtt{q}}{temps\ écoulé \times pourcentage\ de\ temps\ CPU}$$

Ce qui nous donnerait ici : $\frac{18.9}{13.6\times 0.852} = 1.6311101905550953$. Le but étant d'avoir une idée du nombre de relations ainsi obtenues et le temps CPU passé pour les avoir pour un jeu de paramètres donné, en effet pour donner une unité au score, ce serait le nombre de relations obtenues par spécial-`q` par seconde.

Le problème de ce score est qu'il permet de comparer les résultats pour un `lpb` donné, pour comparer les scores de `lpb` différents, je propose d'utiliser π la fonction qui compte les nombres premiers sous une borne pour définir un 'score général' par la formule :

$$\frac{score}{\pi\left(\mathtt{lpb0}\right) + \pi\left(\mathtt{lpb1}\right)}.$$

Le but de ce score et de donner une estimation de la qualité de la collecte de relations pour un nombre de relations attendu. Pour déterminer `lpb`, je sélectionne des `lpb` potentiels. Pour essayer d'avoir les plus pertinents possibles, on regarde s'ils existent les `lpb` des jeux de paramètres proches de celui qu'on souhaite créer pour ne pas viser trop loin des valeurs intéressantes. Ensuite pour chaque couple `lpb` potentiel, je cherche grossièrement le couple `lim` donnant le meilleur score, ensuite je trie les couples `lpb` en fonction de leur score général.

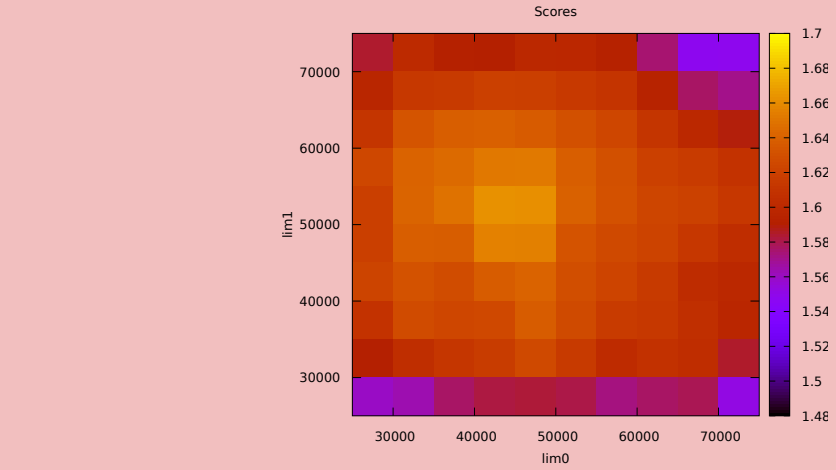
Sélection polynomiale de Kleinjung

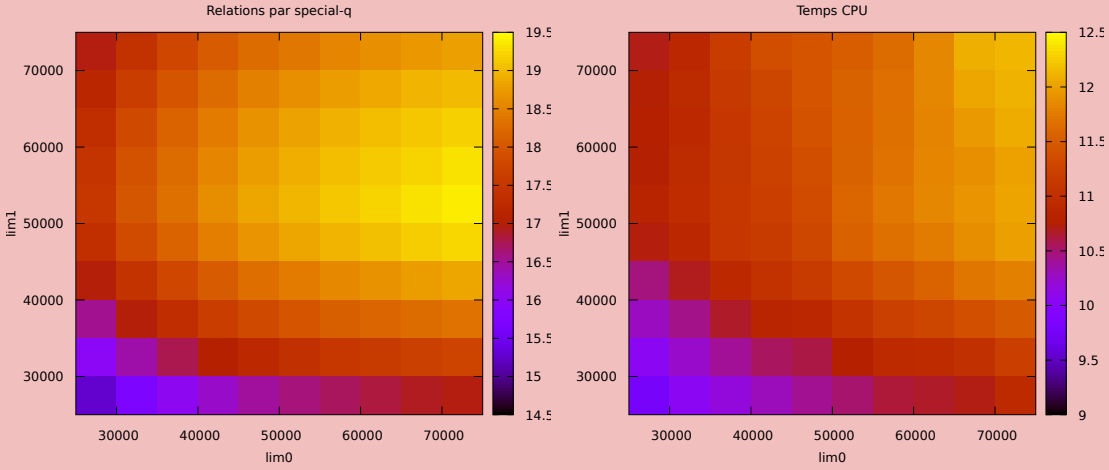
Pour notre exemple taille 70, pour rester large j'ai choisi pour ensemble de `lpb` potentiels $[\![16, 20]\!] \times [\![16, 20]\!]$. Dans le tableau suivant je donne un exemple des résultats que j'ai obtenu :

Rang	`lpb`	`lim`	rels/spécial-`q`	score	score général
1	18, 18	52428, 52428	19	1.66138	3.61169×10^{-5}
2	19, 19	52428, 52428	39.8	2.95939	3.41023×10^{-5}
3	18, 19	52428, 52428	27.9	2.24091	3.37537×10^{-5}
⋮	⋮	⋮	⋮	⋮	⋮
9	17, 17	26214, 39321	6	0.65337	2.66659×10^{-5}
⋮	⋮	⋮	⋮	⋮	⋮
15	20, 20	314572, 104857	72.5	3.46641	2.11302×10^{-5}
⋮	⋮	⋮	⋮	⋮	⋮
23	16, 16	26214, 32768	1.6	0.19216	1.46863×10^{-5}

J'ai remarqué que bien souvent dans le cas de la sélection polynomiale de Kleinjung, avoir `lpb0` = `lpb1` semble donner de meilleurs résultats. De plus, j'ai aussi constaté que de même il semble pertinent de prendre `mfb` $= 2 \times$ `lpb`. Les tests réalisés dans ce tableau sont obtenus avec ce paramétrage de MFB. De ce que nous observons $(\mathtt{lpb0}, \mathtt{lpb1}) = (18, 18)$ semble le plus prometteur, nous ne garderons pas $(19, 19)$ car en plus de prendre plus de temps dans la collecte de relations, nous prendrons aussi plus de temps dans l'algèbre linéaire car ceci produirait une plus grande matrice. Il pourrait être intéressant de tester aussi $(17, 17)$ mais dans le cas présent il ne semble pas vraiment pertinent. Bien souvent lorsqu'on prend un `lpb` trop petit, nous aurons plus tard le problème que le spécial-`q` dépasse `lpb` et ceci car le filtre demande de collecter bien plus de relations que ce que nous avions estimé pour `rels_wanted`. Pour la suite nous garderons donc `lpb` $= (18, 18)$.

Je propose ensuite de chercher `lim` dans $[\![25000, 75000]\!] \times [\![25000, 75000]\!]$ en prenant des pas de 5000 par exemple. Cette méthode peut être remplacée par l'utilisation d'un outil semi-automatique, mais dans le cas présent, je trouve intéressant de proposer les graphiques suivants illustrant les résultats obtenus dans notre grille de recherche :





Ce qu'il est possible d'observer ici et que nous remarquerons bien souvent est qu'il semble exister un couple `lim` optimal pour ma formule de 'score'. Une idée pourrait aussi être de s'intéresser aux nombre de relations par spécial-`q` mais dans notre cas nous gagnerions du temps à considérer le score car pour un même nombre de relations la collecte de relations aurait été plus rapide. Ce qui nous donne à cette étape les paramètres suivants :

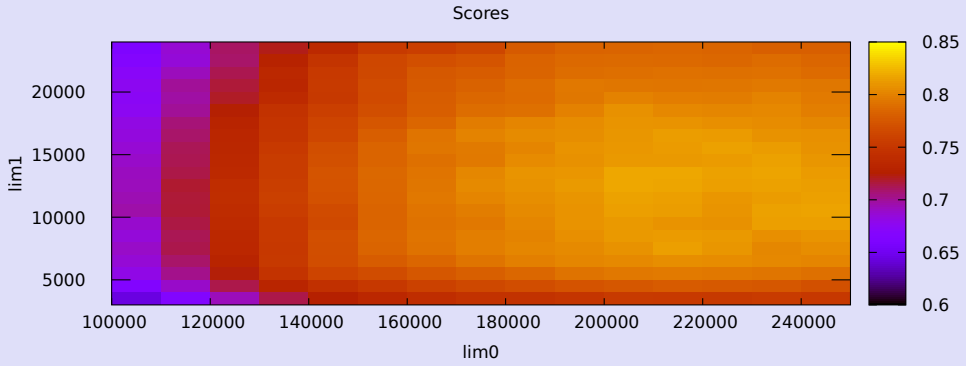
`lpb0` $= 18$	`lpb1` $= 18$
`lim0` $= 50000$	`lim1` $= 50000$
`mfb0` $= 36$	`mfb1` $= 36$
`lambda0` $= 2.2$	`lambda1` $= 2.2$

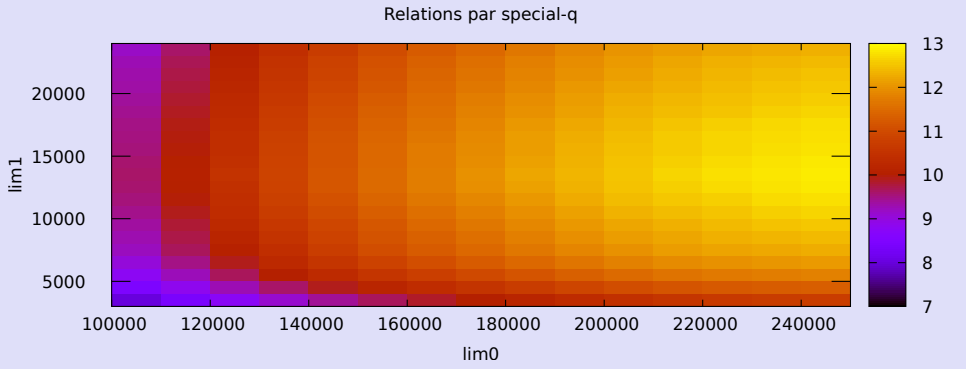
Sélection polynomiale Joux-Lercier

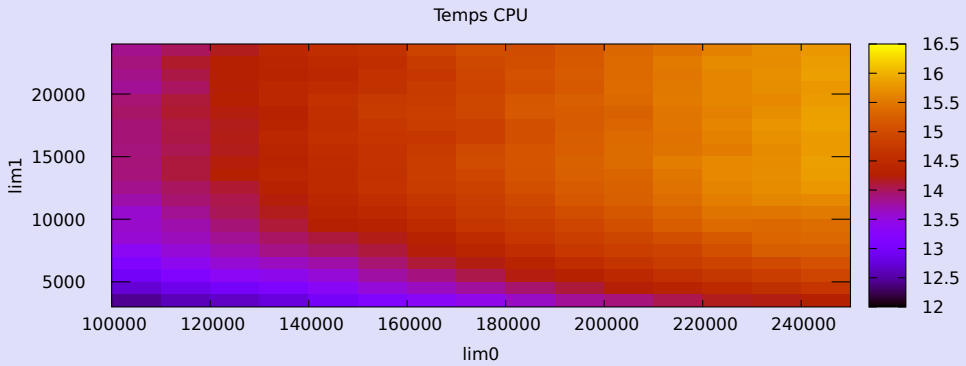
Il est possible de paramétrer le cas Joux-Lercier en se basant sur la même idée. Alors qu'il semble possible de choisir `lpb0` $=$ `lpb1` et `lim0` $=$ `lim1` dans le cas précédent, ceci ne me semble pas aussi pertinent dans le cas Joux-Lercier car les tailles des normes sont très différentes et le spécial-`q` utilisé ne premettra pas de diminuer de beaucoup la différence des tailles. Comme fait précédemment, je cherche un couple de (`lpb0`, `lpb1`) intéressant pour la suite, ici dans $[\![17, 19]\!] \times [\![17, 19]\!]$, ce qui me donne le classement suivant :

Rang	`lpb`	`lim`	rels/spécial-`q`	score	score général
1	18, 17	235929, 13107	12.8	0.82047	2.32752×10^{-5}
2	18, 18	183500, 26214	15.5	1.03003	2.23921×10^{-5}
3	18, 19	183500, 52428	17.0	1.16967	1.76181×10^{-5}
⋮	⋮	⋮	⋮	⋮	⋮

Ce que nous remarquons ici est que `lpb` = (18, 17) semble le plus prometteur. En effet il semble intéressant de prendre `lpb0` plus grand que `lpb1`. Je vais donc ici choisir de regarder `lim0` de 100000 à 250000 et `lim1` de 3000 à 24000.







Alors que dans le cas précédent, on peut facilement trouver un bon compromis entre le nombre de relations et le temps passé à les chercher dans le cas Joux-Lercier il est possible d'avoir plusieurs solutions. Le choix que je fais est dans ce cas de prendre le couple qui maximise le nombre de relations

obtenues. Je garde donc ici le couple (`lim0`, `lim1`) = (250000, 12500). Dans le cas Joux-Lercier je pose `mfb0` = `lpb0`. Bien que je garde `mfb1` = $2 \cdot$ `lpb1`, en effet pour le côté 0 (le côté de g) la norme est bien plus grande et demande des paramètres plus grands pour le crible. Dans le cas présent, demander au crible d'Eratosthène de trouver les facteurs premiers de la norme côté 0 donne de meilleurs résultats bien qu'il demande plus de temps. De même on placera le spécial-`q` dans le cas Joux-Lercier du côté 0 avec `tasks.sieve.sqside = 0` dans le but d'aider la factorisation de la norme côté 0 en la réduisant d'un premier facteur.

10.2 Paramétrage pour l'algèbre linéaire

Maintenant que nous avons paramétré chaque unité de calcul indépendante nous allons nous intéresser au temps total passé dans la collecte des relations et le temps passé ensuite dans l'algèbre linéaire. Ici, nous n'allons pas prendre $p \in \mathbb{S}_{70}$, mais nous allons prendre $p \in \mathbb{S}_{72}$ de manière à choisir nos paramètres pour supporter le passage de la taille 70 à 75. Ce qui va nous intéresser ici est de paramétrer `qmin`, `rels_wanted` et `I` dans l'idée suivante :

- `qmin` doit être assez grand pour éviter d'avoir trop de doublons mais assez petit pour produire assez de relations sans dépasser `lpb`.
- On veut que `qmin` et `rels_wanted` minimise la somme des temps passés dans la collecte de relations et dans l'algèbre linéaire.
- `I` doit être assez petit pour gagner du temps dans la collecte de relations sans être trop petit et risquer de manquer de relations.

Dans cette étape le temps à attendre risque d'être très important selon la taille que l'on souhaite paramétrer. Dans le but d'essayer de réduire le temps passé à tester nos paramètres, nous allons dans un premier temps lancer une collecte de relations complète jusqu'à l'étape du filtre pour un I fixé. Pour ceci on va se donner `qmin` = `qrange` et `rels_wanted` assez grand, je propose de l'ordre d'un facteur au moins 10 par rapport à $\pi(LPB_0) + \pi(LPB_1)$, le but étant simplement de faire que `qmin` dépasse `lpb` et ne pas avoir à recalculer de relations pour les tests suivants (un critère de rejet pour le test sera que CADO-NFS relance la collecte de relations). Pour jouer ensuite sur la finesse de `qmin`, on fixera `qrange` comme le pas sur lequel on incrémentera `qmin`. Une fois les relations obtenues, nous pourrons les trouver dans le répertoire `p70.upload` du répertoire de travail. À cette étape nous listerons les relations sous la forme `p70.xxx-yyy.XXXXXXXX.gz` dans un fichier, par exemple `p70.rels` pour les importer dans la suite. Les `qmin` possibles dans les étapes suivantes seront ceux dont on trouve une relation où il est possible de poser `qmin` = `yyy` et ignorer les relations plus petites lors de l'import. Si `/tmp/cado.0igfal_b` est notre répertoire de travail, il sera possible de relancer CADO-NFS en utilisant les relations déjà calculées en ajoutant dans la commande pour `cado-nfs.py` les arguments suivants :

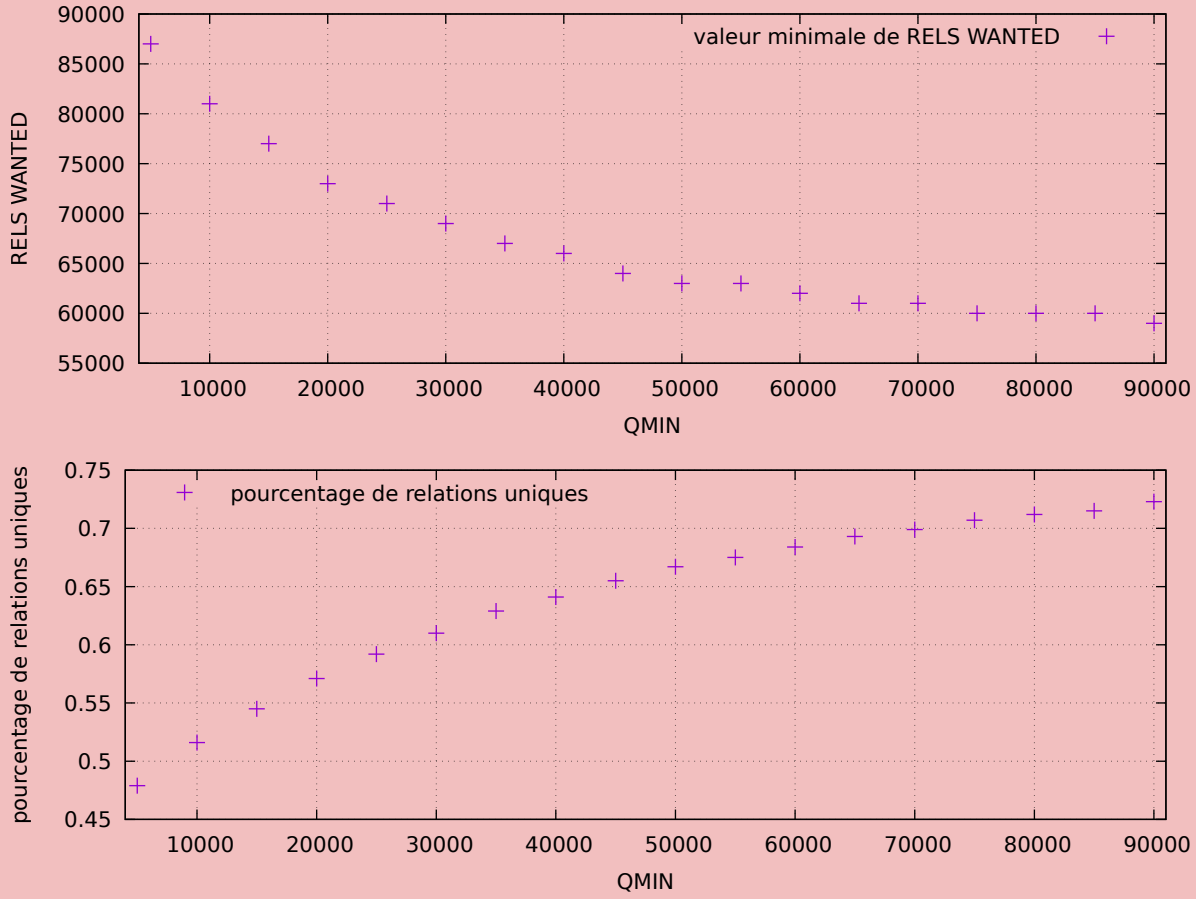
```
tasks.polyselect.import=/tmp/cado.0igfal_b/p70.poly tasks.sieve.import=@/tmp/cado.0igf
al_b/p70.rels
```

Disant de reprendre les polynômes f et g déjà calculés et de recharger les relations dont les noms sont énumérés dans `/tmp/cado.0igfal_b/p70.rels`. Attention si le nombre de relations présentes dans le répertoire est plus grand que `rels_wanted`, CADO-NFS rechargera toutes les relations listées, il faudra donc chercher à n'énumérer que les relations qui nous permettent de rester sous cette borne sans avoir à les recalculer, par exemple : on peut générer cette liste à l'aide d'un script. De même j'ai automatisé la méthode que je propose en `python3`.

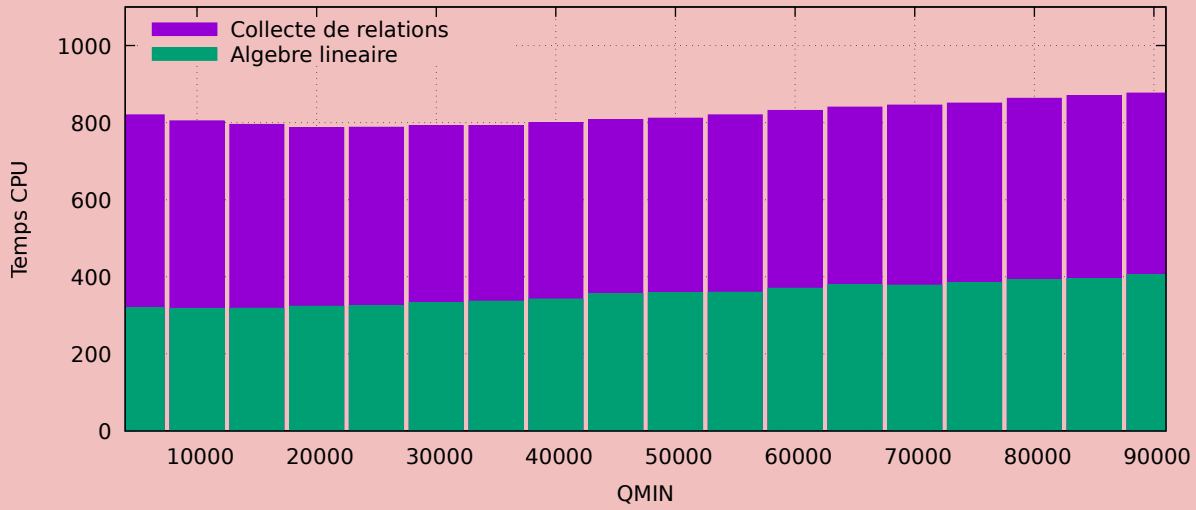
Ensuite nous pouvons chercher pour un `qmin` donné, le nombre de relations à atteindre pour ne pas avoir besoin de relancer le filtre et passer à l'algèbre linéaire. Nous voulons éviter que le filtre relance la collecte de relations car en plus de prendre plus de temps, il est possible que `qmin` dépasse `lpb` ce qui entraînera l'arrêt des calculs. Plus un `qmin` est grand plus le taux de relations uniques sera élevé. Donc nous permet de prendre `rels_wanted` plus petit ce qui réduit le temps passé dans la collecte des relations. Dans cette étape là nous cherchons à obtenir un taux de relations uniques minimal pour choisir nos `qmin`. Dans le cas où le filtre nous bloque en relançant la collecte de relations de manière à faire que `qmin` dépasse `lpb`, ce sera causé bien souvent par le fait que le `I` choisi est trop petit ou `qmin` trop grand. Le critère de qualité ici sera de minimiser la somme des temps passés dans la collecte de relations puis dans l'algèbre linéaire.

Sélection polynomiale de Kleinjung

Nous nous placerons dans le cas où `I` = 11 pour illustrer des exemples de couples (`qmin`, `rels_wanted`) possibles. Nous donnerons sur le premier graphique les `rels_wanted` minimaux obtenus en fonction de `qmin` et dans le second le pourcentage de relations uniques obtenues, les tests ont été réalisés pour des `qmin` allant de 5000 à 90000 par pas de 5000, le nombre de relations précalculées n'étant plus suffisant pour `qmin` = 95000 :

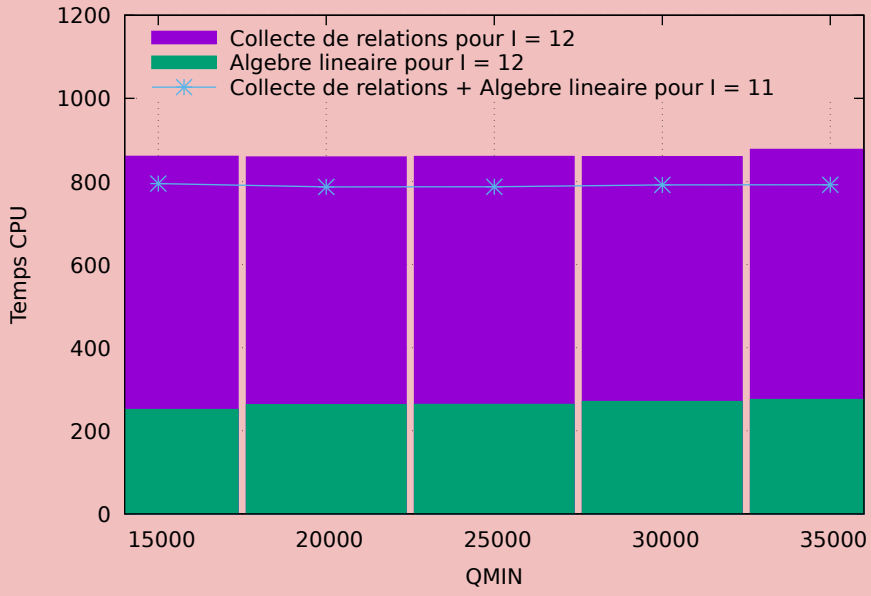


Maintenant nous allons chercher le couple qui minimise la somme des temps passés dans l'algèbre linéaire et dans la collecte des relations. Pour l'algèbre linéaire, nous devrons calculer ces temps, mais pour la collecte des relations, il est possible de calculer le temps CPU passé dans la collecte des relations sans avoir à la relancer, il suffit de faire la somme des temps CPU obtenus à la dernière ligne des fichiers de relations importés. Dans le graphique suivant nous donnerons les temps CPU obtenus pour les différents couples (`qmin`, `rels_wanted`) dans la collecte de relations, l'algèbre linéaire pour observer le temps passé dans la somme des deux.

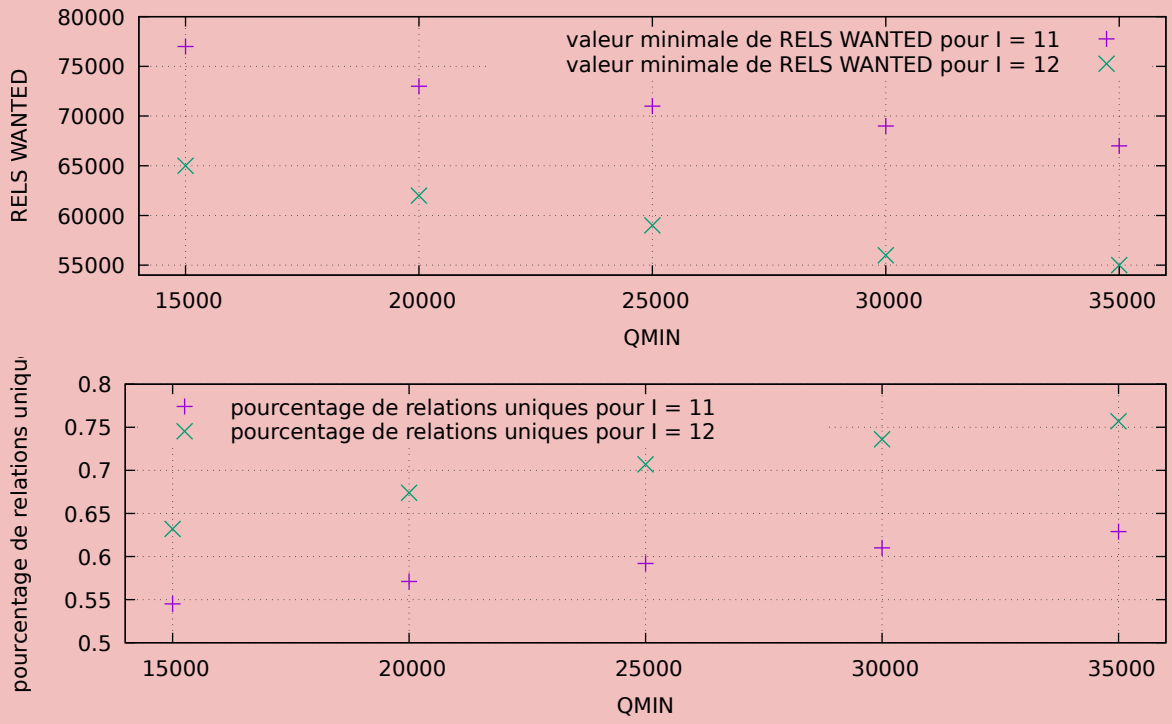


Nous observons ici qu'un `qmin` pertinent pour $\mathbb{S}_{72}$ pourrait être dans $[\![15000, 35000]\!]$, c'est donc avec cet intervalle que nous continuerons à étudier les couples (`qmin`, `rels_wanted`). Nous avions fixé `I` à 11, avec pour hypothèse que plus `I` est petit plus le temps total est petit, nous chercherons `I` le plus petit possible permettant de trouver nos couples (`qmin`, `rels_wanted`). Nous gardons alors le $p \in \mathbb{S}_{72}$ et relançons la collecte de relations pour `I` $= 10$ (Nous ne pouvons pas garder les relations précédemment obtenues ici). Sur des tests pour `qmin` $\in \{5000, 25000, 50000\}$ j'observe que le nombre de relations n'est pas suffisant pour passer à l'algèbre linéaire, donc `I` est au minimum 11.

Nous pourrions nous demander s'il ne serait pas intéressant de prendre `I` $= 12$ et s'il faut vraiment prendre I le plus petit possible. Pour vérifier si cette piste est intéressante, il est possible de comparer les temps obtenus. Pour comparer les résultats obtenus, nous pouvons par exemple regarder le même graphique que pour précédemment mais pour `I` $= 12$ et `qmin` dans $[\![15000, 35000]\!]$:



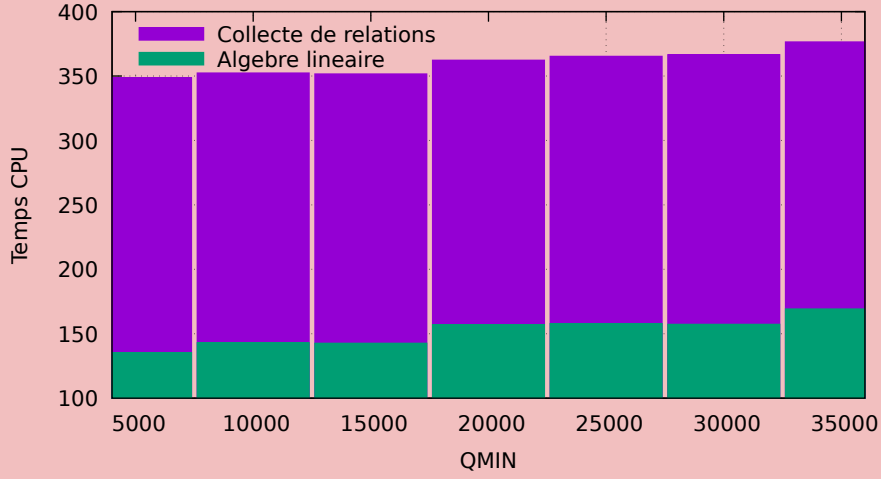
Le choix de `I` plus grand va réduire le temps passé dans l'algèbre linéaire car en effet on aura ici besoin de moins de relations et un meilleur ratio de relations uniques, mais en contrepartie va demander plus d'efforts à la collecte de relations. J'illustre la différence du nombre de relations minimal et le ratio de relations uniques par les graphiques suivants :



Pour la suite nous garderons `I` $= 11$ et utiliserons les `qmin` inférieurs à 35000 ainsi obtenus avec les `rels_wanted` associés. Dans la suite nous allons chercher à trouver le meilleur couple (`qmin`, `rels_wanted`) pour $\mathbb{S}_{70}$. C'est-à-dire que nous chercherons à optimiser ces paramètres dans le cas $\mathbb{S}_{70}$ dans l'idée que les paramètres ne soient pas adaptés seulement à $\mathbb{S}_{72}$ tout en restant pertinents pour cette taille et qu'ils puissent aussi donner des temps intéressant pour un grand nombre de $p \in \mathbb{S}_{[\![67,72]\!]}$. Pour cela nous nous donnons par exemple :

```
p = 128515791447715057015098691012962151472681017049708065555446698725688 3
ell = 642578957238575285075493455064810757363405085248540327777233493628441
```

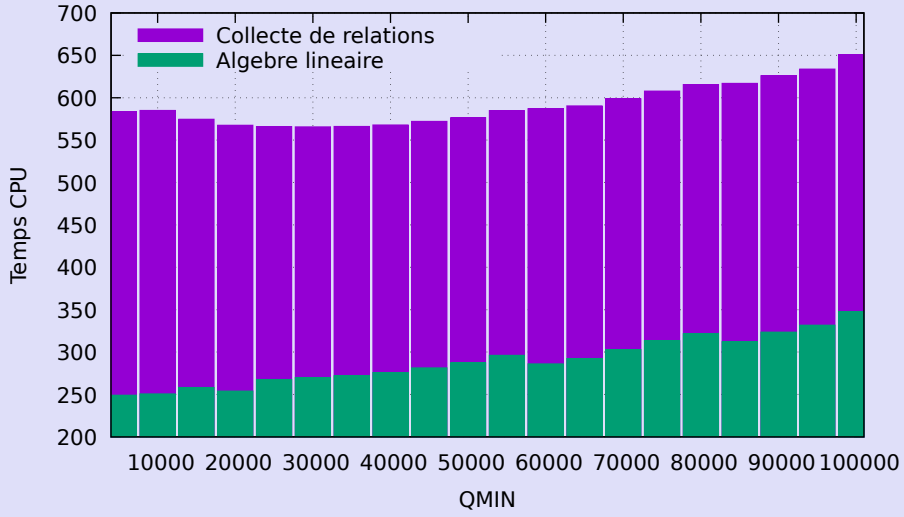
Ce qui va nous intéresser ici est de regarder de la même manière la somme des temps CPU dans $\mathbb{S}_{70}$ de la collecte de relations et de l'algèbre linéaire pour déterminer `qmin` mais en gardant `rels_wanted` adapté à $\mathbb{S}_{72}$.



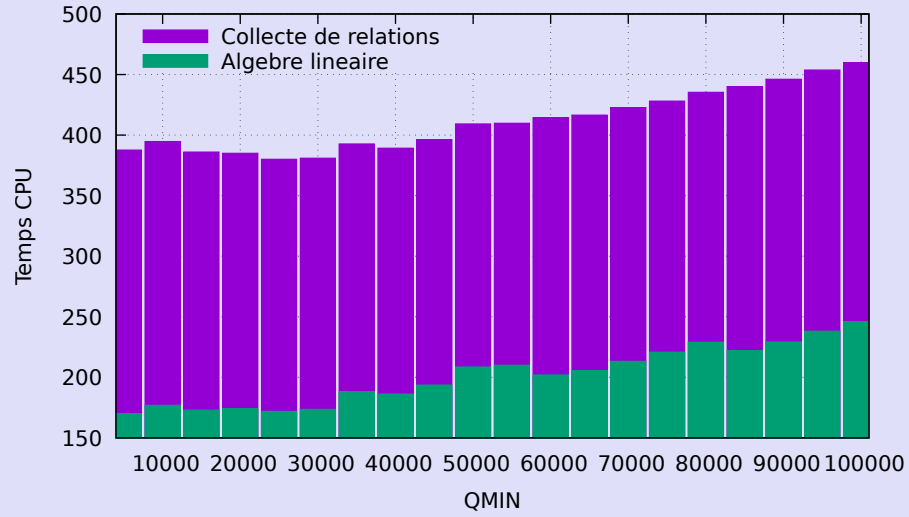
Les résultats de ce graphique nous donneraient envie de fixer `qmin` assez bas, mais le problème est que ceci rendrait le jeu de paramètres beaucoup moins intéressant pour $\mathbb{S}_{72}$. Un optimal pour $\mathbb{S}_{72}$ semble être proche de 20000 et 15000 et être un résultat intéressant pour $\mathbb{S}_{70}$ qui ne pénalise pas trop $\mathbb{S}_{72}$. Ensuite il faut vérifier que le couple obtenu n'est pas local à un p particulier que nous avions choisi pour la construction des nos paramètres. Pour valider notre résultat je propose de choisir un nombre arbitraire de $p_i \in \mathbb{S}_{72}$ et vérifier que le filtre ne demande pas de relancer la collecte de relations ce qui demande d'augmenter `rels_wanted` ou que `qmin` dépasse `lpb` ce qui demanderait de diminuer `qmin`. Après vérification sur au moins 20 tests où $p_i \in \mathbb{S}_{72}$ je trouve seulement 693500626677481597860790004795069410104213209890628545673823528009499079 qui demande de relancer la collecte de relations, ce qui fait que nous allons devoir augmenter `rels_wanted`. Dans le cas où on observe beaucoup d'erreurs, le mieux est de refaire l'expérience avec un $p \in \mathbb{S}_{72}$ demandant plus de relations que le p utilisé en regardant au niveau du filtre les excès pour déterminer celui avec lequel retravailler nos paramètres, mais ici je pense que pour un seul relancement de la collecte, c'est simplement que `rels_wanted` a été choisi trop juste car minimal pour le p utilisé. Il se trouve qu'à `rels_wanted` = 78000 je ne trouve plus d'erreur. Nous garderons alors le couple (`qmin`, `rels_wanted`) = (15000, 78000). Il est possible ici d'affiner `qmin` à l'aide de d'autres $(p_1, p_2) \in \mathbb{S}_{70} \times \mathbb{S}_{72}$ pour avoir des résultats plus pertinents sur un plus grand nombre de corps finis tout en faisant attention à garder `rels_wanted` assez large dans $\mathbb{S}_{72}$ comme précédemment ce qui est aussi un avantage dans $\mathbb{S}_{70}$ car on gagne en général un peu de temps dans l'algèbre linéaire sans trop pénaliser le temps passé dans la collecte de relations. Ce que je remarque ici sur mes différents fichiers de paramètres obtenus est que en général on trouve des `qmin` intéressants assez proche de $\frac{\texttt{lim}}{2}$.

Sélection polynomiale Joux-Lercier

La méthodes que j'ai utilisée pour le cas de la sélection polynomiale de Kleinjung pour trouver `qmin` et `rels_wanted` est de mon point de vue aussi pertinente pour le cas Joux-Lercier, je fais donc de même dans cette configuration. Ce qui me donne en premier lieu, le graphique suivant pour p de taille 72 chiffres :



Un `qmin` intéressant ici semble être proche de 30000. Pour se donner une idée des résultats sur la taille qui nous intéresse, autrement dit 70 chiffres, on regarde aussi le graphique suivant :



Dans le cas Joux-Lercier un `qmin` possible pourrait être 25000, dans le cas présent ceci demanderait au moins `rels_wanted` = 57000. Pour au moins 20 tests `rels_wanted` = 60000 est encore trop petit, je propose donc `rels_wanted` = 62000 qui lui semble ne pas poser problème pour des p de taille 72 chiffres.

11 Filtre pour algèbre linéaire

Dans cette étape il est possible d'améliorer légèrement le temps passé dans l'algèbre linéaire principalement pour les plus petites tailles. La matrice obtenue après la collecte de relations est une matrice assez creuse, mais on peut ici paramétrer l'élimination gaussienne. Pour ceci on utilise les paramètres `target_density` pour demander au filtre de réduire la matrice jusqu'à une densité particulière et `maxlevel` qui le nombre maximal de coefficients présents sur la colonne pour qu'elle puisse être utilisée pour l'élimination gaussienne. Ici, comme pour régler `qmin` et `rels_wanted` on lance une première fois la collecte de relations pour les importer plus tard dans le but de n'avoir que l'algèbre linéaire à calculer. Ensuite ce que je propose est de faire varier `target_density` jusqu'à obtenir un temps pour l'algèbre linéaire qui semble minimal, ici je fais simplement en sorte de garder `maxlevel` assez grand pour me permettre d'atteindre la densité souhaitée.

12 Logarithmes individuels

Il existe dans CADO-NFS un script pour fabriquer les fichiers de hint : `hintfile-helper.py`. J'ai voulu de moi-même tester plusieurs hypothèses sur les fichiers de hint. Pour ceci je me suis intéressé aux résultats que je pouvais obtenir en lançant moi-même `las` sur des exemples des différentes tailles en nombre de bits possibles et essayer de paramétrer chaque ligne du fichier de hint selon les idées suivantes :

— Est-ce qu'il est intéressant de calculer un score qui prend en compte le temps passé et le nombre de relations de relations obtenues ?
— Faut-il chercher à faire en sorte que la relation donne les plus petits facteurs possibles en sacrifiant le nombre de relations pour construire un arbre moins haut ?
— Faut-il maximiser le nombre de relations ce qui peut potentiellement nous en donner des meilleures ?

En fabriquant des fichiers de hint sur ces 3 idées, j'ai ensuite comparé les temps total des logarithmes individuels. Ce que j'ai trouvé est que la dernière solution qui vise à obtenir un maximum de relations gagnait sur le temps des logarithmes individuels d'au moins un facteur 3 sur les autres solutions. De plus les fichiers que j'obtiens en m'aidant de `hintfile-helper.py` donnent les mêmes résultats que ma dernière solution, son utilisation me semble donc pertinente pour la construction des fichiers de hint.

Ce que je remarque est quand dans la construction des fichiers de hint, à partir d'un certain seuil pour `lpb`, les lignes commencent à se répéter, par exemple pour un extrait du fichier `p60.hint` :

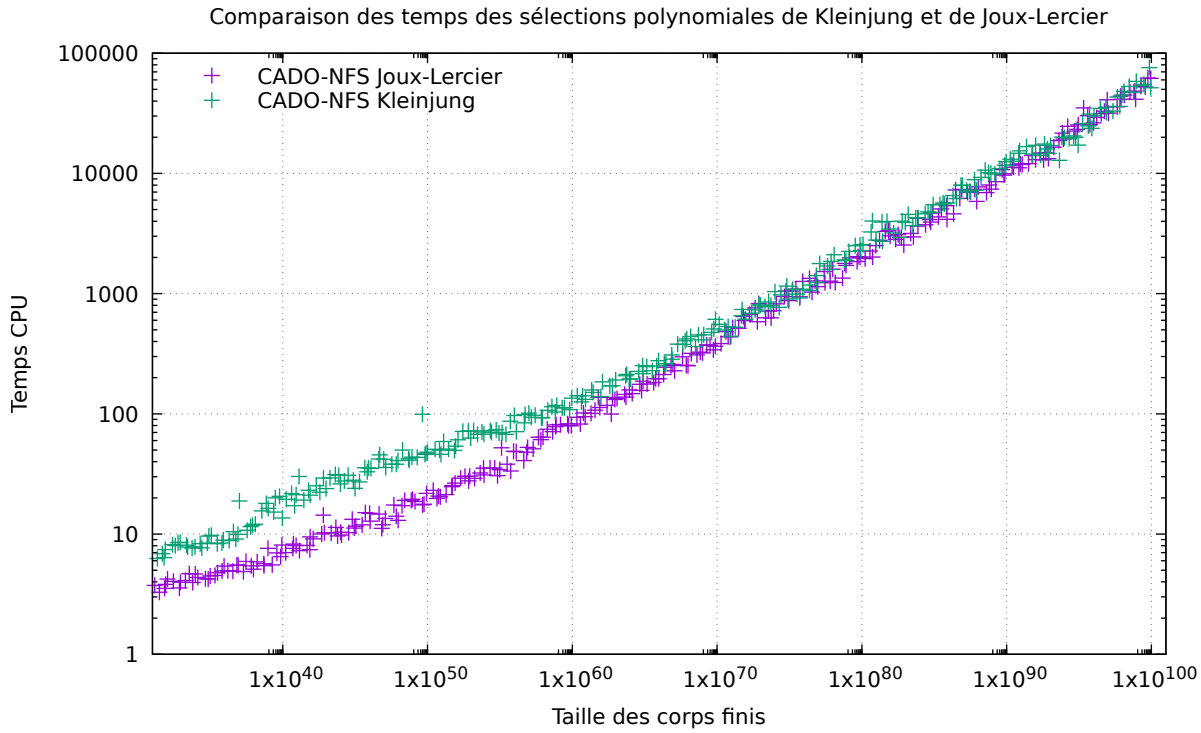
```
24@0 1 1 I=11 25000,23,46 25000,24,48
24@1 1 1 I=11 25000,23,46 25000,23,46
25@0 1 1 I=11 25000,24,48 25000,25,50
25@1 1 1 I=11 25000,24,48 25000,24,48
26@0 1 1 I=11 25000,25,50 25000,26,52
26@1 1 1 I=11 25000,25,50 25000,25,50
27@0 1 1 I=11 25000,26,52 25000,27,54
27@1 1 1 I=11 25000,25,50 25000,25,50
28@0 1 1 I=11 25000,26,52 25000,27,54
28@1 1 1 I=11 25000,25,50 25000,25,50
29@0 1 1 I=11 25000,26,52 25000,27,54
```

Ce que j'ai pu constaté expérimentalement est que ici le `init_lpb` donnant le meilleur temps est celui au niveau de ce seuil de répétition, autrement dit par exemple ici `init_lpb` = 27. Pour le choix des autres paramètres je prends les mêmes que ceux de la collecte de relations. Il y a en plus le paramètre `init_tkewness` qui joue ici le même rôle que `qmin` dans la collecte de relations. Ici je fixe sa valeur à `lim` principalement dans l'idée de nous économiser un nœud de l'arbre de descente.

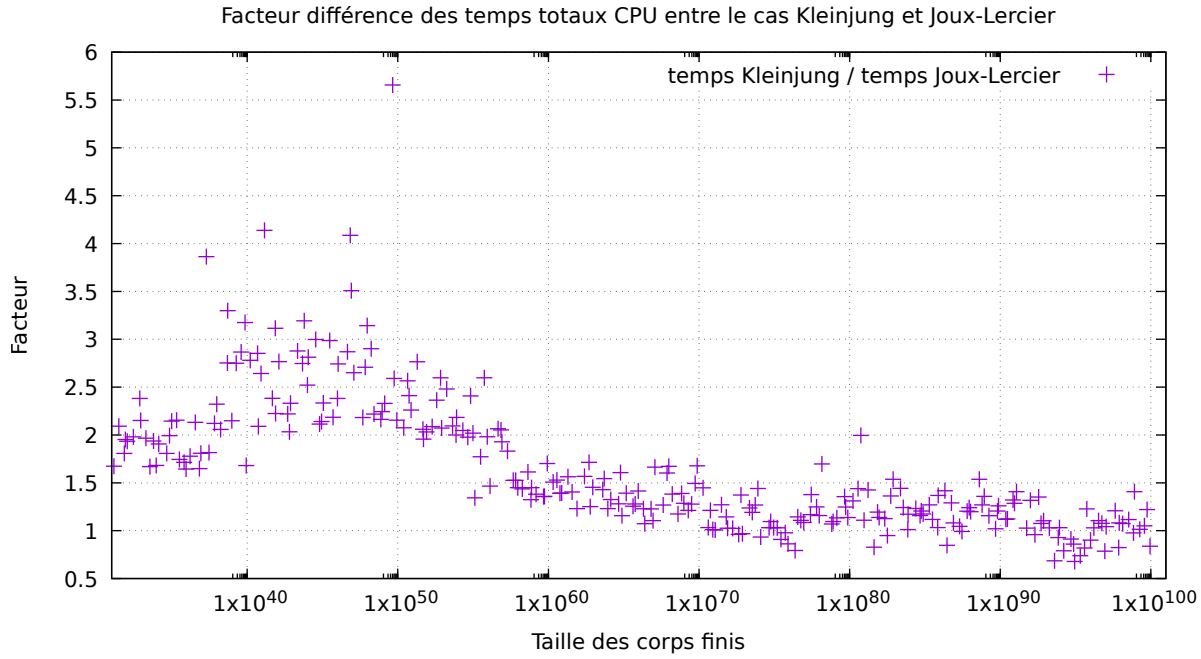
Quatrième partie
Résultats expérimentaux

13 Comparaison sélection polynomiale de Kleinjung et Joux-Lercier

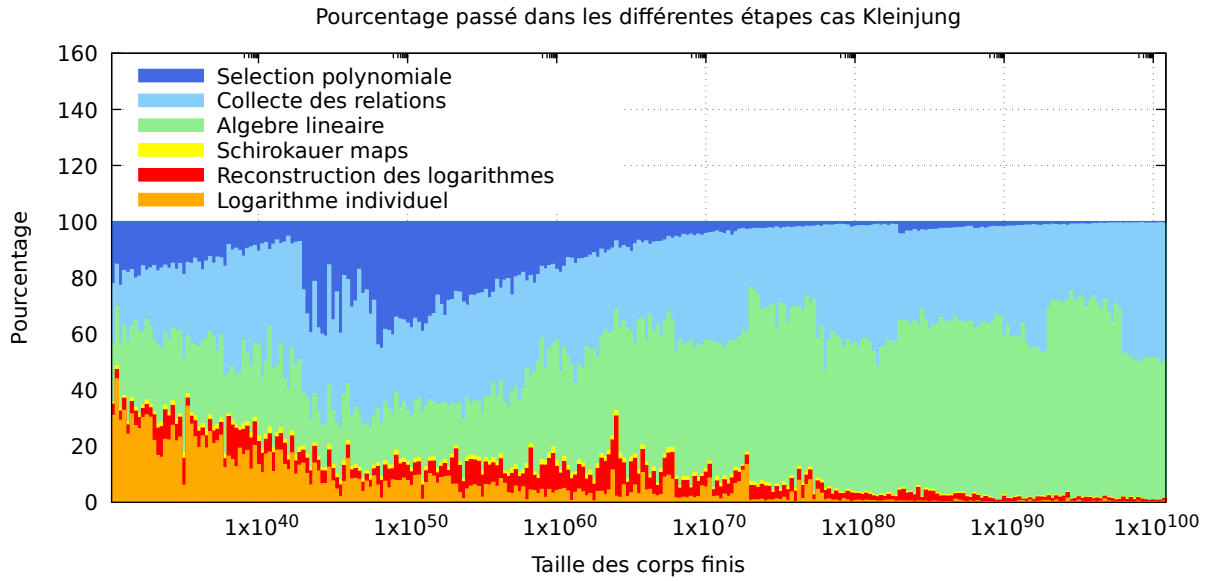
Dans les sections précédentes j'ai parlé du cas de deux types de sélections polynomiales, celle de Kleinjung et celle de Joux-Lercier. La sélection polynomiale de Kleinjung étant utilisée pour la factorisation par CADO-NFS, elle a donc été utilisée pour le logarithme discret. La sélection polynomiale de Joux-Lercier a elle été développée pour le logarithme discret uniquement. Nous comparons dans le graphique suivant les temps obtenus des p premiers de Sophie Germain pour les différentes sélections polynomiales. À noter que ces p ont été frabriqués tels que le polynôme f dans le cas de Kleinjung se factorise en un produit de racines modulo ℓ et Joux-Lercier est utilisé avec l'option `fastSM` ceci dans le but de pouvoir comparer les performances des deux sélections polynomiales pour le logarithme discret sans le problème des Schirokauer maps sur les petites tailles. La section suivante parlera du problème observé pour comprendre ce choix de construction.



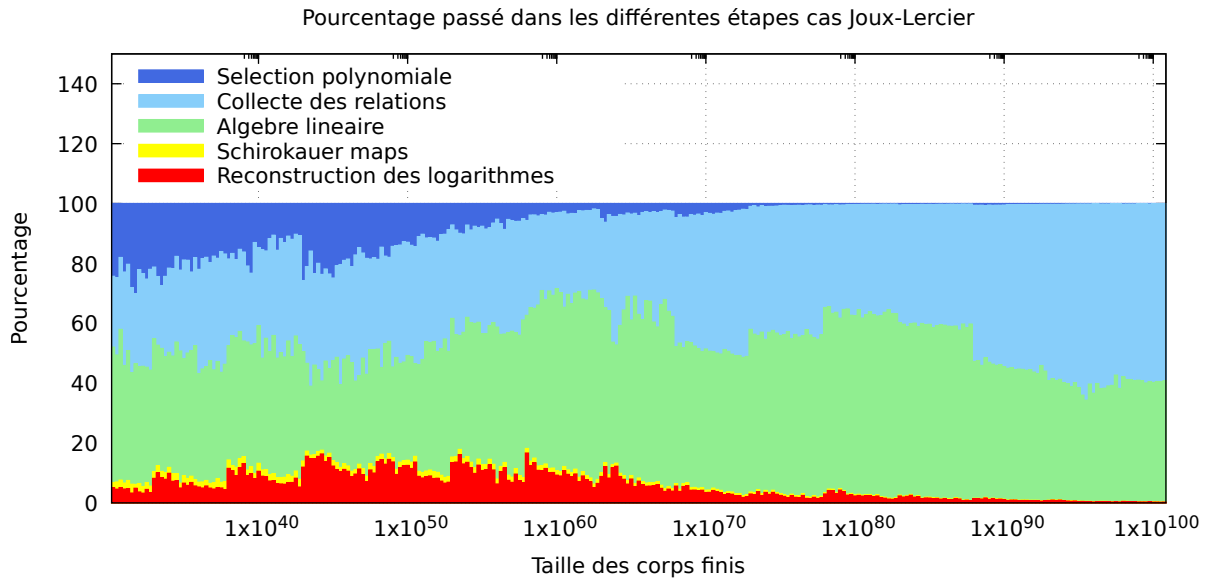
Ce qu'on observe ici est que la méthode Joux-Lercier gagne sur les petites tailles et qu'ensuite les deux méthodes semblent équivalentes sur les grandes tailles. On peut sûrement expliquer cette différence par le fait que pour le cas de Kleinjung nous passons d'un polynôme f de degré 3 pour les tailles en dessous de 42 chiffres à un polynôme de degré 4 pour les tailles supérieures à 42. Au niveau de ce palier ni le degré 3, ni le degré 4 n'est optimal et malheureusement il n'est pas possible d'utiliser un polynôme de degré 3,5. Ce qui peut expliquer ici cette différence de performances car dans le cas Joux-Lercier on garde f et g de degrés respectifs 3 et 2 sur la plage de valeurs observée.



On observe en effet que le gain principal par la méthode Joux-Lercier sur la méthode de Kleinjung est d'au moins un facteur 2 lors du changement de degré par Kleinjung. On observe malgré tout que dans la majorité des cas pour les petites tailles Joux-Lercier semble meilleur au moins jusqu'aux tailles de 90 chiffres.



Ce qu'il est possible d'observer ici est que pour les grandes tailles c'est les temps passés dans la collecte de relations et l'algèbre linéaire qui occupent la majorité du temps total. Par contre pour les petites tailles, il est possible d'observer que le temps des logarithmes individuels compte beaucoup plus bien qu'il soit en dessous de 4 secondes CPU sur ma machine pour les tailles de 50 chiffres ou moins. Un autre temps intéressant à remarquer ici est celui de la sélection polynomiale car en effet jusqu'aux tailles à 42 chiffres on recherche un polynôme de degré 3, mais ensuite le fait de rechercher un polynôme de degré 4 fait que le temps passé à rechercher f et g est aussi important que le temps de la collecte de relations ou l'algèbre linéaire par exemple pour les tailles de 50 chiffres.



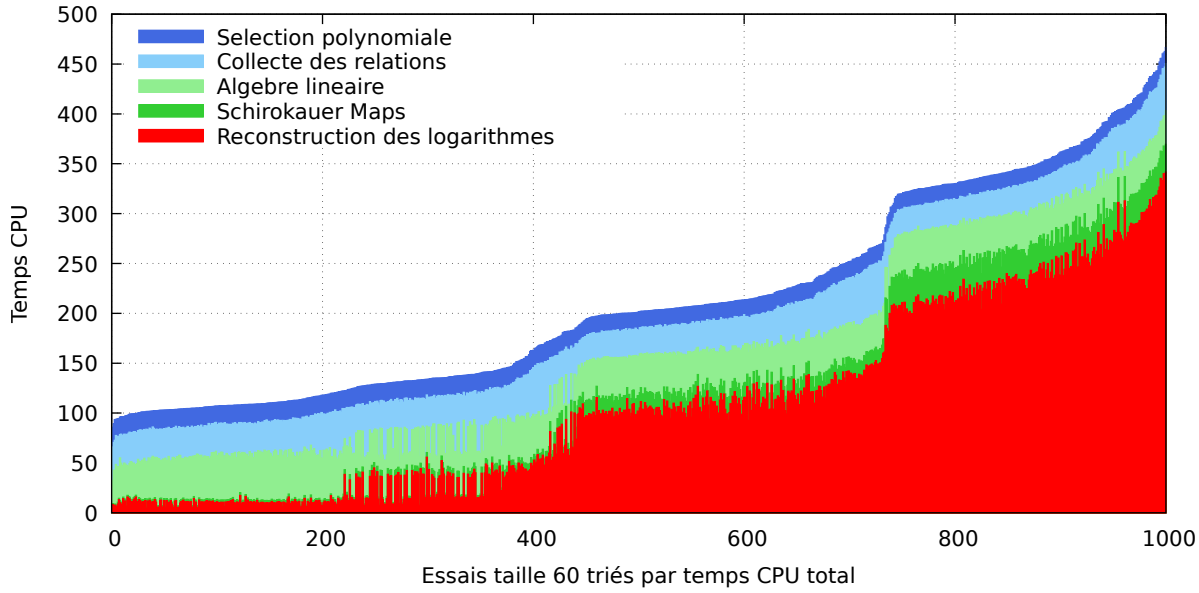
Dans le cas Joux-Lercier les logarithmes individuels étant encore très expérimentaux au moment de la rédaction de mon rapport, je ne les ai pas paramétrés ce qui explique leur absence sur ce graphique.

La principale observation pour le cas Joux-Lercier est que comme dans le cas de Kleinjung c'est principalement le temps passé dans la collecte de relations et l'algèbre linéaire qui détermine le temps total et que pour les grandes tailles les autres étapes deviennent négligeables même si pour le cas des petites tailles les temps de la sélection polynomiale et de la reconstruction des logarithmes peuvent aussi avoir une importance.

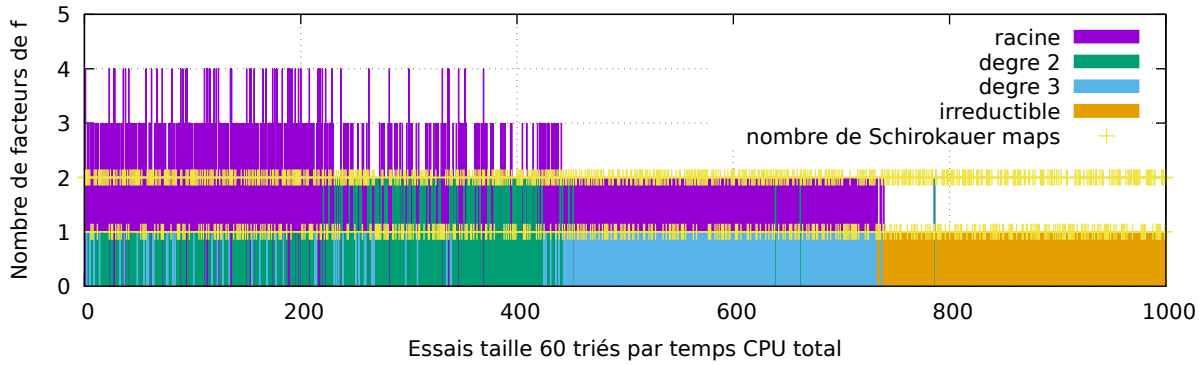
14 Schirokauer maps

14.1 Problème observé pour les petites tailles

Lorsqu'on se donne une taille de chiffres T, ce que l'on cherche est de créer un fichier de paramètres qui minimise le temps CPU passé pour les précalculs pour des $p \in \mathbb{S}_T$ que pourraient choisir l'utilisateur. Pour cela je regarde les temps obtenus pour des $p \in \mathbb{S}_T$ aléatoires. J'ai pu observer dans un premier temps pour quelques exemples des écarts de temps importants sur le temps CPU total. Le temps passé dans la reconstruction des logarithmes en était la cause. Dans le graphique suivant, on a tiré 1000 $p_i \in \mathbb{S}_{60}$, ici classés par temps CPU total :



On observe ici une différence d'au moins un facteur 4,5 entre le meilleur temps obtenu et le moins bon temps. La valeur médiane est d'un facteur 2 au dessus du meilleur temps. De plus, on voit ici que c'est en effet le temps passé dans la reconstruction des logarithmes qui joue le rôle le plus important dans le temps CPU total. Une autre différence intéressante est que alors que le temps passé dans les Schirokauer maps est négligeable devant les temps passés dans la collecte de relations ou l'algèbre linéaire par exemple dans les meilleurs cas, alors que lorsque le temps passé dans la reconstruction des logarithmes est élévé, le temps passé dans les Schirokauer maps est équivalents au deux étapes citées précédemment.



Dans ce graphique, on s'intéresse au degré des facteurs irréductibles de f dans $\mathbb{F}_\ell[X]$, par exemple si $f(x) = 240x^4 + 36022x^3 + 23184677953x^2 + 97580662656722x + 486154...3186$ et sa factorisation modulo ℓ est $240(x + 663696...59669)(x^3 + 268099...30574x^2 + 374630...377109x + 213505...573617)$, alors f se décompose en une racine et un polynôme de degré 3. En utilisant le résultat du théorème de Dirichlet sur les Schirokauer maps on peut de même calculer le nombre de Schirokauer maps que CADO-NFS a utilisé pour ses calculs. On observe que le nombre de Schirokauer maps à utiliser pour f est 1 ou 2 dans les exemples observés. De plus en regardant le graphique précédent donnant les temps observés, on peut remarquer que les paliers semblent en effet avoir un lien avec les degrés des facteurs de f. Nous expliquerons le lien théorique entre le nombre de Schirokauer maps et la taille des facteurs de f dans $\mathbb{F}_\ell[X]$ dans la section suivante pour comprendre en quoi le cas où les facteurs de f sont de plus grand degré augmente le temps passé dans le calcul des Schirokauer maps.

14.2 Explication théorique du problème observé

Je donnerai une explication théorique assez simpliste pour se donner une idée du problème observé. Le calcul des Schirokauer maps se fait dans l'anneau fini $\mathbb{F}_\ell[X]/f(X)$. Dans le cas où f peut se factoriser en $f_i \in \mathbb{F}_\ell[X]$ polynômes irreductibles, on peut écrire $f = \prod_i f_i$. Le résultat est que dans ce cas $\mathbb{F}_\ell[X]/f(X)$ est isomorphe à $\prod_i \mathbb{F}_\ell[X]/f_i(X)$. Si par exemple f_i est de degré 1, alors on a alors que $\mathbb{F}_\ell[X]/f_i(X) \simeq \mathbb{F}_\ell$ où en effet les calculs vont se faire rapidement. Mais dans le cas où f est irréductible dans $\mathbb{F}_\ell[X]$ le corps fini dans lequel s'effectuent les calculs est beaucoup plus grand d'où des temps de calcul plus longs.

14.3 Comparaison des performances des solutions proposées

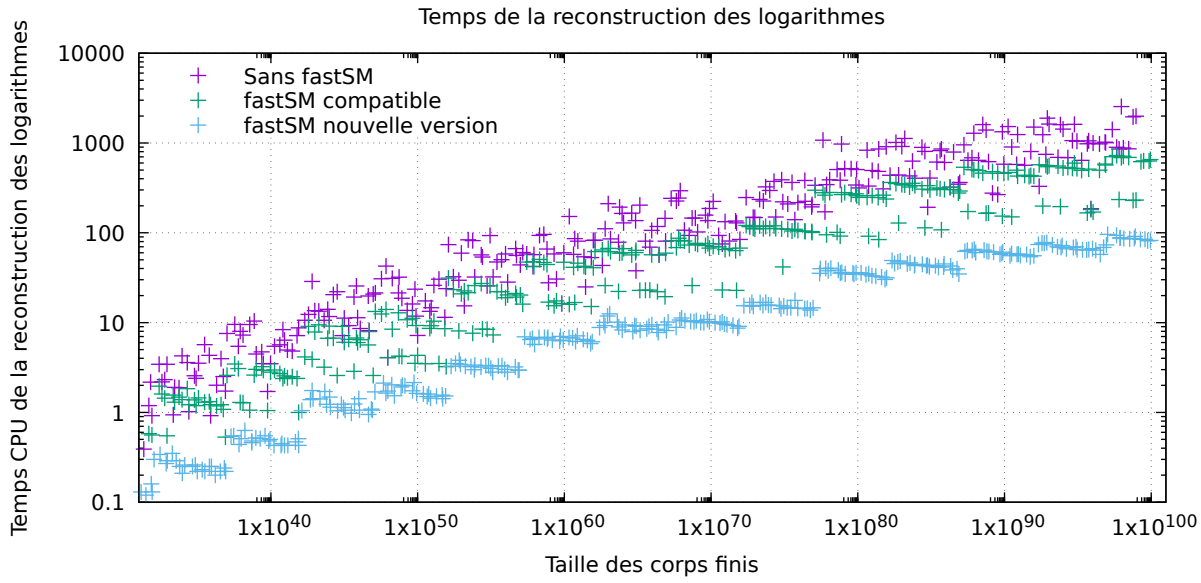
La solution proposée est de pouvoir sélectionner les polynômes f et g dès la sélection polynomiale avec un critère supplémentaire, ici c'est l'option `fastSM` qui n'est implémentée que pour le cas Joux-Lercier pour des raisons de compatibilité qui la rendrait difficile à ajouter dans le cas de la sélection polynomiale de Kleinjung. En effet la sélection polynomiale de Kleinjung est commune à la factorisation d'entiers dans CADO-NFS alors que le cas Joux-Lercier n'est proposé que dans le cas de la résolution du logarithme discret. Le critère en plus qu'on impose à f et g est de ne demander le calcul que d'une Schirokauer map et d'avoir au moins une racine modulo ℓ. Dans l'idée qu'il existe $f_1 \in \mathbb{F}_\ell[X]$ de degré 1 tel qu'il soit facteur de f dans $\mathbb{F}_\ell[X]$ ce qui serait le cas lorsque f a au moins une racine modulo ℓ et que le calcul s'effectue réellement que dans un corps fini $\mathbb{F}_\ell[X]/f_1(X) \simeq \mathbb{F}_\ell$ car on ne demanderait qu'un calcul de Schirokauer map.

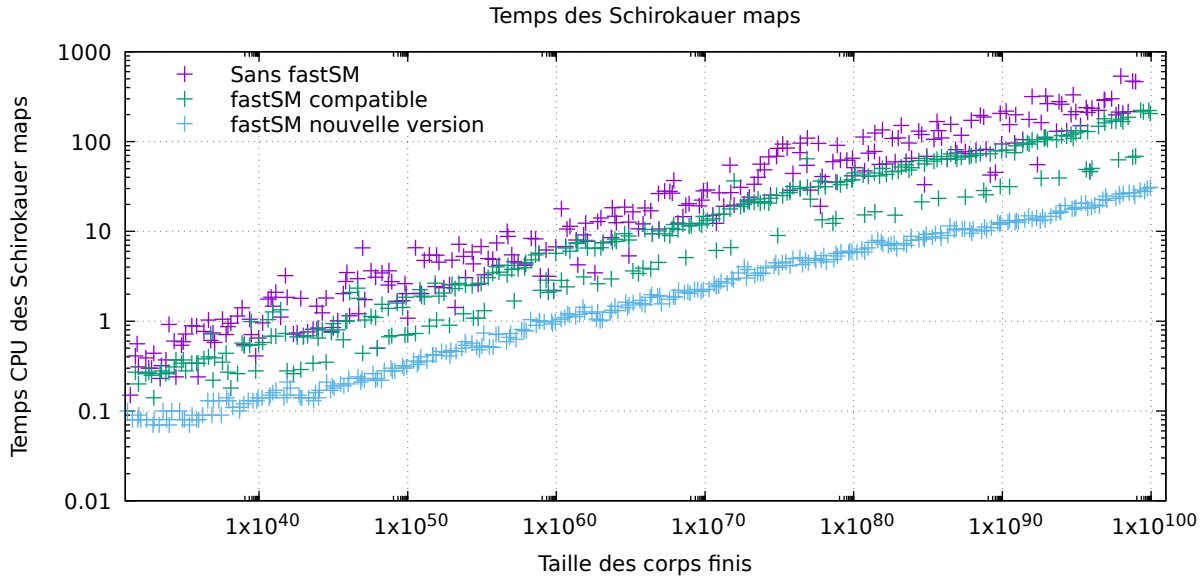
Ici, nous allons comparer les temps de CADO-NFS Joux-Lercier dans les cas de figure suivants :

— La version utilisée lorsque le problème est observé (`CADO-NFS 2.3.0`)

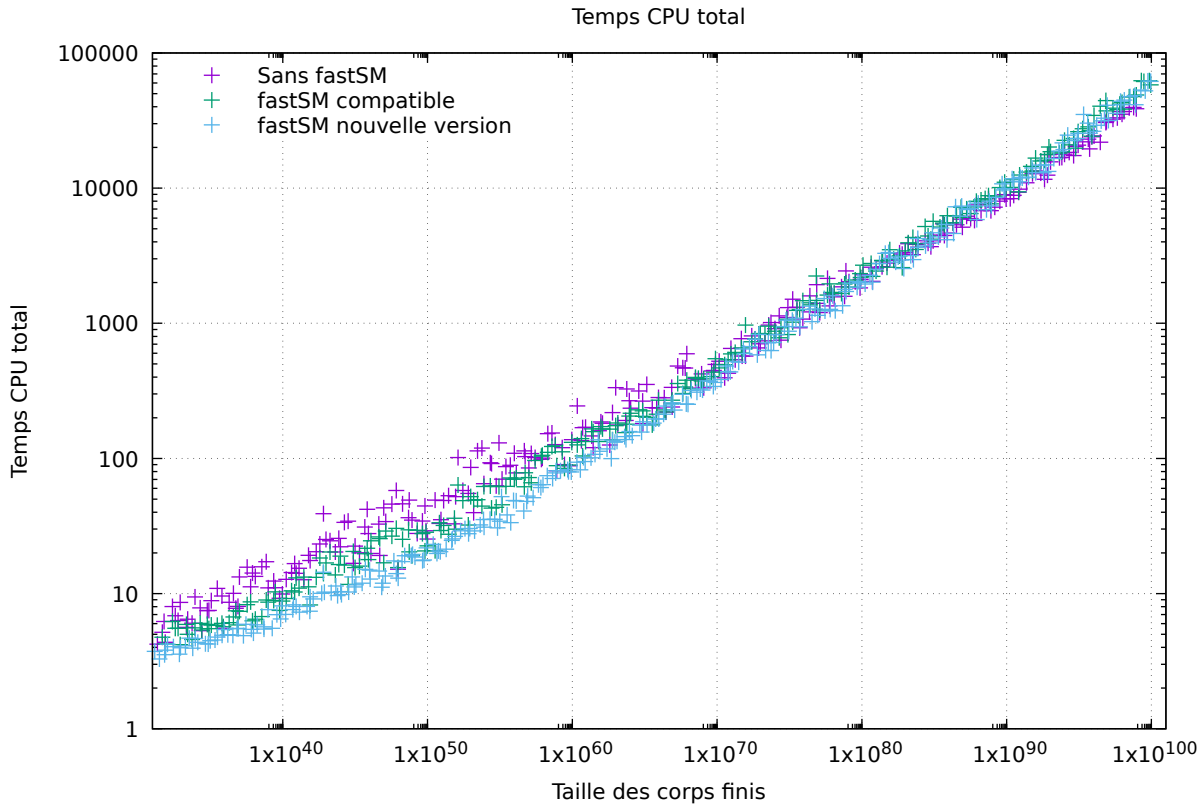
— Une première proposition de l'option `fastSM` qui sélectionne les polynômes sur les critères énoncés précedemment mais gardant les Schirokauer maps de la version `CADO-NFS 2.3.0` pour des problèmes de compatibilité.

— Une dernière version qui propose une nouvelle version du calcul des Schirokauer maps dans le cas Joux-Lercier.

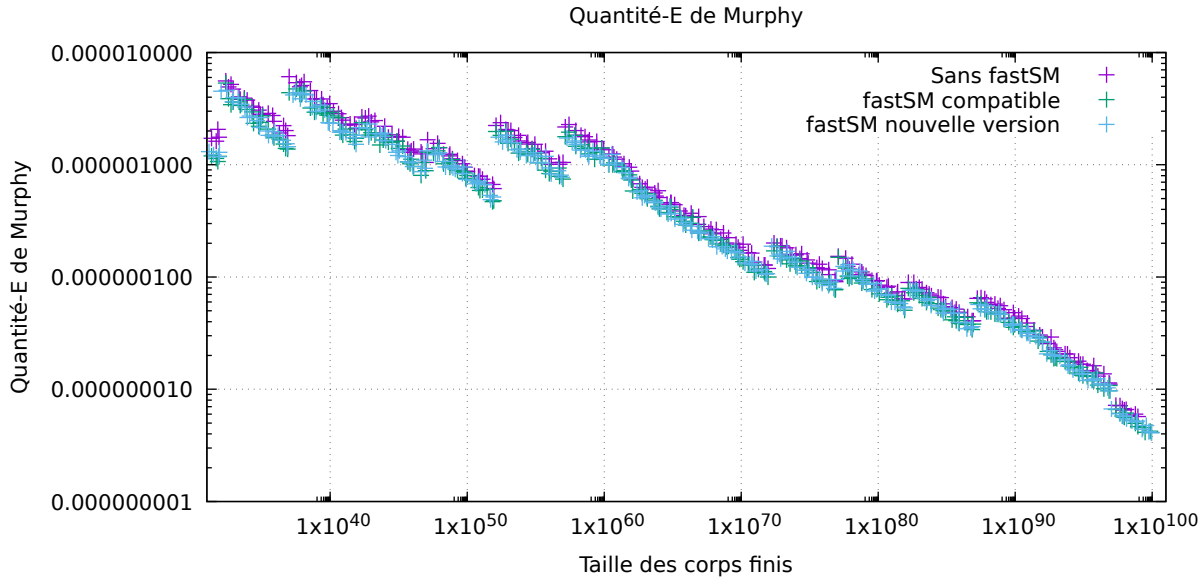


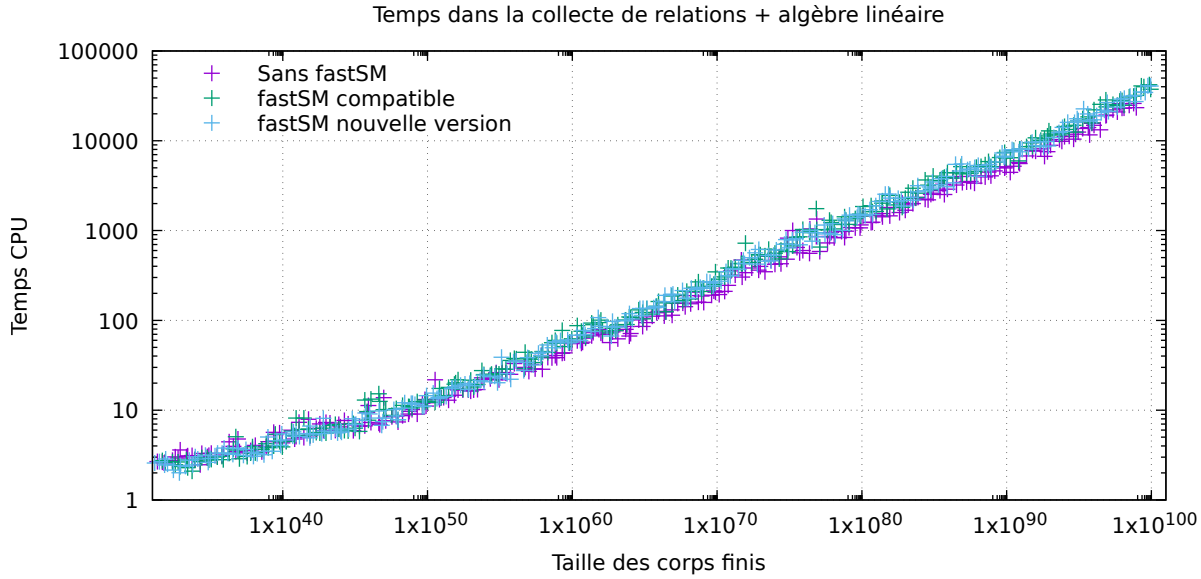


Le graphique suivant l'illustre l'impact des différentes solutions sur le temps total. On remarque que la première solution proposée fait déjà gagner de l'ordre d'au moins un facteur 3 en moyenne. Mais que l'amélioration qui décide de proposer une nouvelle version du calcul des Schirokauer maps fait gagner de l'ordre d'un facteur 6 en changeant seulement les Schirokauer maps (comparé à l'ancienne version `fastSM`).



Cependant pour les plus grandes tailles le temps passé dans la reconstruction des logarithmes semble négligeable à côté du temps CPU total bien que les graphiques précédents montrent que le gain sur le temps passé dans la reconstruction des logarithmes et le calcul des Schirokauer maps est bien plus important. Il semble même que corriger le problème des Schirokauer maps sur les grandes tailles serait moins intéressant car en effet on impose des critères de sélection de f et g plus stricts ce qui demande plus d'efforts dans les tâches telles que la collecte de relations et l'algèbre linéaire. Pour l'illustrer mon hypothèse, le graphique suivant donne les quantité-E de Murphy, puis la somme des temps passés dans la collecte de relations et l'algèbre linéaire :





On peut voir ici que la quantité de Murphy est meilleure sans l'option `fastSM`, ce qui semble en effet se confirmer lorsqu'on regarde le temps de la collecte des relations et de l'algèbre linéaire. Il semble en effet que même si la différence semble légère, un choix de critères plus stricts pour corriger le problème des Schirokauer maps fait malgré tout perdre un peu de temps sur le temps total passé un certain seuil. À l'aide du graphique donnant les temps totaux, je pense que l'option `fastSM` est vraiment pertinente pour les tailles de 80 chiffres et moins, mais que pour les plus grandes tailles il semble préférable de ne pas l'utiliser, le temps de la reconstruction des logarithmes et des Schirokauer maps devenant négligeables sur le temps total.

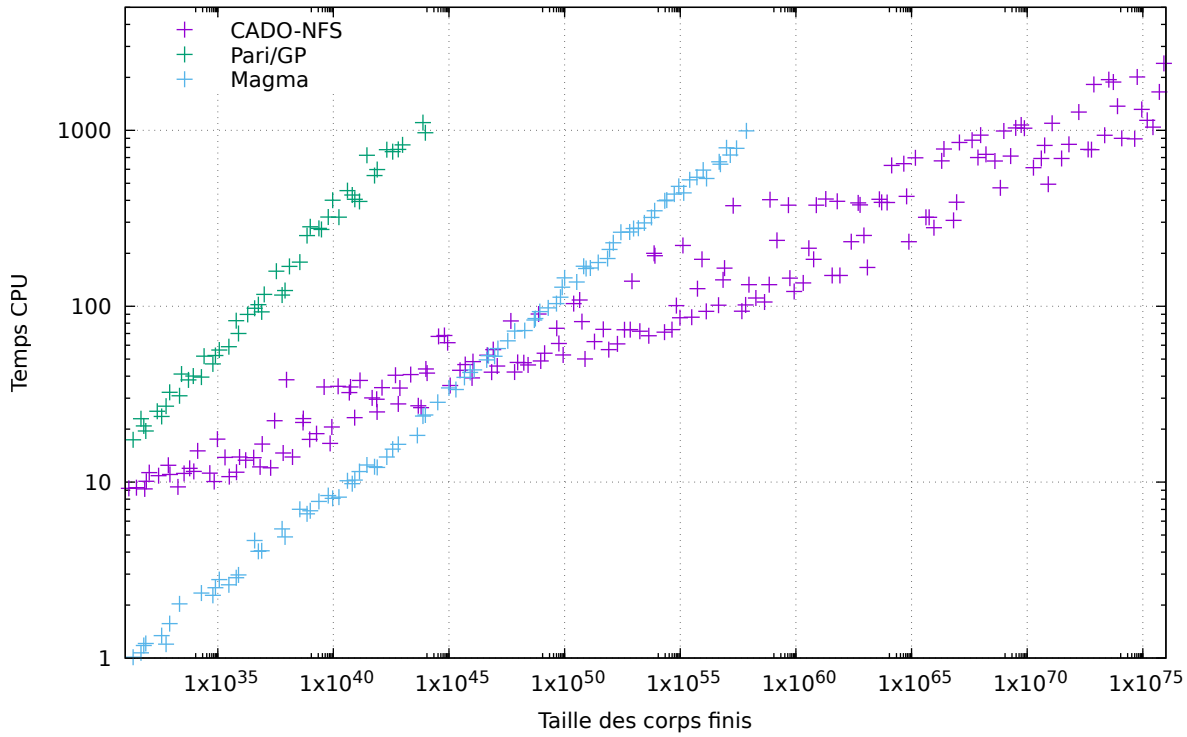
15 Comparaison avec autres implémentations de résolution du problème du logarithme discret

Il est maintenant intéressant de comparer les performances de CADO-NFS avec d'autres implémentations de la résolution de problème du logarithme discret. Ce qui nous intéresse ici est le temps CPU obtenu sur un cœur machine. J'ai effectué mes tests sur 4 machines de même configuration qui est la suivante :

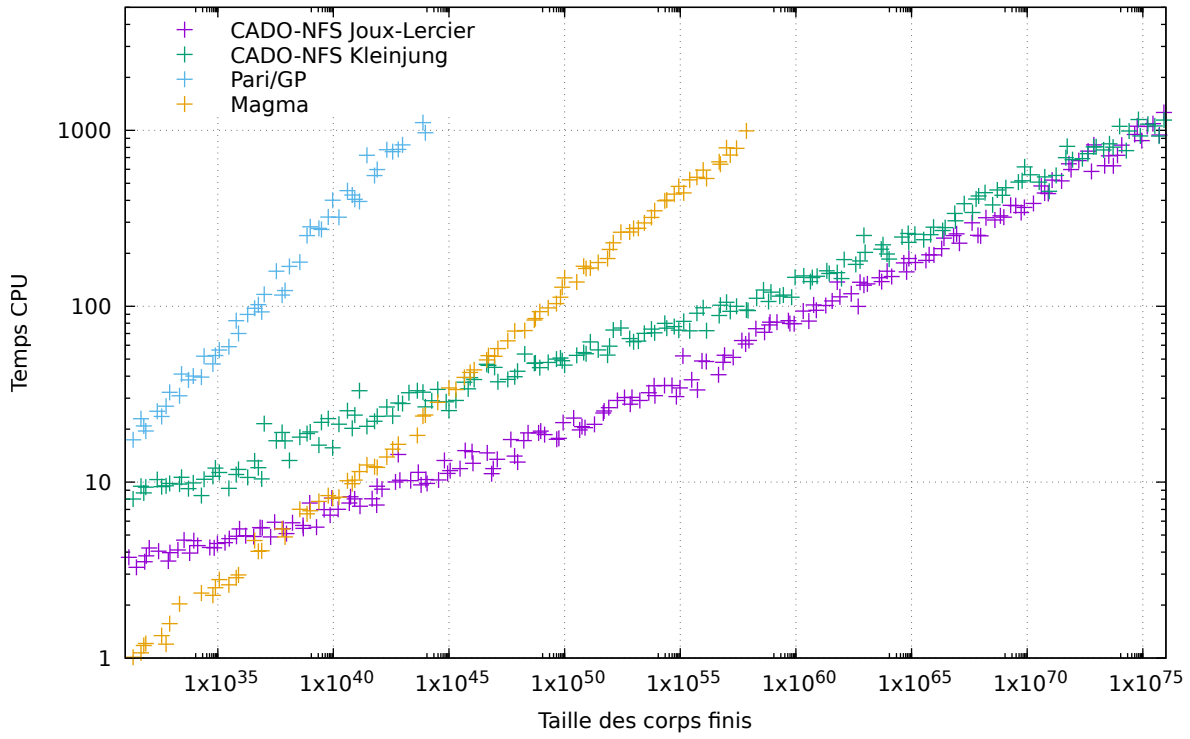
```
$ uname -r
4.16.0-2-amd64

$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 45
Model name:            Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz
Stepping:              7
CPU MHz:               1197.024

gcc version 8.2.0 (Debian 8.2.0-4)
GP/PARI CALCULATOR Version 2.9.5
Magma V2.23-8
SageMath version 8.0
```

À noter que les algorithmes de Sage, Magma et GP/PARI sont différents de l'algorithme NFS implémenté par CADO-NFS. Magma et Pari/GP sont des implémentations d'algorithmes en $L_N\left(\frac{1}{2}, c\right)$ alors que NFS est en $L_N\left(\frac{1}{3}, c\right)$. Ce qui fait que CADO-NFS est asymptotiquement plus rapide que Magma et Pari/GP, nous nous attendons alors à ce que pour des grandes tailles CADO-NFS devienne meilleur.

Le graphique suivant illustre la comparaison des temps obtenus pour les différentes implémentations. Les tests sont réalisés sur les mêmes tailles de corps finis $\mathbb{F}_p^*$ avec p un nombre premier de Sophie Germain où p varie de 30 à 75 chiffres. Nous nous intéressons aux taille au dessus de 30 chiffres car CADO-NFS n'est pas adapté aux très petites tailles. Ma version de Sage n'a pas pu être illustré sur la graphique suivant car à partir des tailles de 29 chiffres, Sage change d'algorithme pour le calcul du logarithme discret donc il n'est plus possible de le comparer avec les autres implémentations listées. Le problème est corrigé dans la version 8.3, c'est-à-dire que Sage utilise Pari/GP. Dans le cas de CADO-NFS nous illustrons le cas de l'algorithme de Kleinjung pour donner une idée des temps obtenus à l'aide des jeux de paramètres travaillés pendant mon stage. Le graphique suivant est tel qu'on pourrait l'observer en comparant les temps obtenus sans biais.



Mais il peut être intéressant de comparer les performances de Magma et Pari/GP avec la version Joux-Lercier en cours de développement et le cas où la sélection polynomiale de Kleinjung choisirait le couple de polynôme évitant le problème des Schirokauer maps, autrement dit ici je propose un biais où les $p_i \in \mathbb{S}$ sont choisis tels que f se factorise en produit de racines modulo ℓ. Dans le cas de Joux-Lercier le calcul du logarithme individuel n'est pas effectué, mais le temps passé pour un logarithme individuel étant négligeable à côté du temps passé dans les précalculs, il est malgré tout possible de se faire une idée des performances dans cette configuration par rapport aux autres implémentations.



Il est possible d'observer ici que Magma est le plus adapté pour les plus petites tailles. À partir de 30 chiffres CADO-NFS est déjà plus intéressant que Pari/GP. Dans le cas de l'algorithme de Kleinjung, il semble plus intéressant d'utiliser CADO-NFS pour les tailles à partir de 45 chiffres, même si avec le problème des Schirokauer maps le crossover est plutôt entre 45 et 50 chiffres et dans le cas Joux-Lercier le crossover pourrait être pour les tailles entre 38 et 39 chiffres. On peut de même voir que pour un calcul durant 1000 secondes CPU, on aurait par exemple un résultat pour une taille de 44 chiffres avec Pari/GP, une taille 58 avec Magma et une taille 75 avec CADO-NFS.

Cinquième partie
Conclusion

Synthèse de mes travaux

Le logarithme discret est en effet très présent dans le monde de la cryptographie moderne et c'est sur la difficulté de de le retrouver que repose principalement les systèmes actuels d'échanges de données. Nous avons eu ici l'occassion de parler en détails de l'algorithme NFS et plus particulièrement d'une implémentation de celui-ci qu'est CADO-NFS. Nous avons pu voir que la compréhension de CADO-NFS demande des connaissances en mathématiques assez fines, principalement en l'algèbre qui peuvent même aller bien plus loin que mes connaissances universitaires actuelles comme dans le cas des Schirokauer maps.

L'algorithme NFS se décompose en plusieurs étapes où chacune peut demander un paramétrage particulier qui impacte les étapes suivantes. Une première approche a été de s'intéresser uniquement au temps total, le problème était qu'elle est bien trop coûteuse en temps et qu'il est difficile de justifier le choix de paramètres obtenus. Je pense donc qu'il est plus pertinent de vraiment chercher à optimiser chaque étape indépendamment tout en prenant en compte les impacts sur le temps total ce qui demande de connaître NFS, l'effet de faire varier chaque paramètre et le résultat à regarder pour déterminer la pertinence du paramétrage. Malgré tout le paramétrage de CADO-NFS reste un problème difficile, car en effet il demande pour les petites tailles de passer par l'expérimentation et l'attente de temps non négligeables pour essayer de déterminer une solution pertinente pour un grand nombre de cas possibles,

car en effet un jeu de paramètres doit éviter de n'être optimal que pour un seul cas de corps fini. Bien que ma méthode ne donne pas la garantie d'être optimale ou proche de l'optimal, je pense malgré tout qu'elle produit des jeux de paramètres intéressants et me permet de plus facilement justifier mes choix de paramètres par une approche plus scientifique.

De plus nous avons pu voir que paramétrer les petites tailles est bien différent ce qui peut être fait pour de plus grandes tailles. En effet les temps passés dans des étapes comme la sélection polynomiale, le logarithme individuel ou la reconstruction des logarithmes deviennent négligeables pour les grandes tailles, mais le temps passé dans ces étapes a son importance pour les petites tailles, ce qui a par exemple été mis en évidence dans le cas des Schirokauer maps.

J'ai aussi eu l'occasion d'expérimenter le paramétrage de CADO-NFS dans le cas Joux-Lercier et en le comparant avec la sélection polynomiale actuellement issue de la factorisation, il semble que cette nouvelle sélection semble plus pertinente pour les petites tailles en particulier car le choix de degré pour les polynômes pour ces tailles semble être optimal à la différence du cas Kleinjung où le changement de degré semble affaiblir ses performances pour les petites tailles.

J'ai finalement pu comparer les résultats obtenus à l'aide de mes jeux de paramètres à d'autres implémentations du logarithme discret, montrant que CADO-NFS est une implémentation tout aussi intéressante pour les tailles de 30 à 100 chiffres que celles déjà existantes et utilisées et devient même plus pertinente pour les grandes tailles.

Ce que mon travail a apporté à l'équipe

Mon travail par le paramétrage du logarithme discret des tailles allant de 30 à 100 chiffres fait de CADO-NFS initialement utilisé pour des records sur tailles de corps très grandes un outil utilisable par un plus grand nombre d'utilisateurs en particulier que les calculs pour ces tailles peuvent tourner sur une machine de bureau en un temps raisonnable. En effet mes jeux de paramètres ont été ajouté au projet CADO-NFS (commits c77fe4b3b5f2 et 1cd07513de565).

De plus, en travaillant sur les petites tailles j'ai pu mettre en évidences des phénomènes qui ont un impact négligeable sur les grandes tailles mais qui ont leur importance ici, comme le problème des Schirokauer maps auxquels l'équipe CARAMBA a pu apporter des solutions. Dans le même thème, j'ai aussi pu rapporter plusieurs bugs comme un premier dans l'étape du filtre (corrigé par le commit 3d78465), un autre dans la théorie des nombres ([#21706] dans le Bug Tracking) ou encore dans la reconstruction des logarithmes ([#21707]).

J'ai aussi eu l'occasion de comparer les performances de différentes implémentations du logarithme discret à celle de CADO-NFS montrant que CADO-NFS est une implémentation pertinente du logarithme discret aussi pour les petites tailles. De même, j'ai aussi pu comparer les performances qu'on obtient avec la sélection polynomiale actuelle qu'est celle de Kleinjung à la sélection polynomiale Joux-Lercier encore expérimentale et envisagée pour le logarithme discret.

Ce que le stage m'a apporté

Ce stage a été pour moi l'occasion de m'initier à la recherche et découvrir en quoi consiste le travail d'un chercheur en devant moi-même chercher des jeux de paramètres pertinents tout en m'intéressant un peu plus à des notions de cryptographie introduites en cours de première année de master informatique comme le logarithme discret et pratiquer l'algèbre d'un point par une utilisation concrète en informatique. J'ai aussi eu l'occassion de découvrir et utiliser de nouveaux outils tels que Sage, mais aussi Magma ou Pari/GP.

En effet c'était pour moi l'occassion d'essayer une approche différente, car en effet alors qu'en temps normal je résous de manière assez indépendante des problèmes d'informatique et de mathématiques, j'ai ici eu l'occassion de mettre à profit en même temps les connaissances à la fois en informatique et en mathématiques acquises de mon cursus universitaire. En effet pour aborder le paramétrage de CADO-NFS j'ai dû tatonner, expérimenter, analyser et comprendre mes résultats pour finalement trouver des méthodes qui s'affinent et deviennent plus pertinentes avec l'évolution de ma compréhension de l'algorithme NFS.

Références

[1] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, and Luke Valenta. Imperfect forward secrecy : How Diffie-Hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 5–17. ACM, 2015.

[2] Shi Bai, Cyril Bouvier, Alexander Kruppa, and Paul Zimmermann. Better polynomials for GNFS. *Mathematics of Computation*, 85(298) :861–873, 2016.

[3] Cyril Bouvier. *Algorithmes pour la factorisation d'entiers et le calcul de logarithme discret*. PhD thesis, Université de Lorraine, 2015.

[4] Don Coppersmith. Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. *Mathematics of Computation*, 62(205) :333–350, 1994.

[5] Joshua Fried, Pierrick Gaudry, Nadia Heninger, and Emmanuel Thomé. A kilobit hidden SNFS discrete logarithm computation. In *Eurocrypt*, pages 202–231. Springer, 2017.

[6] Pierrick Gaudry. Integer factorization and discrete logarithm problems. Notes d'un cours donné aux Journées Nationales de Calcul Formel, 2014.

[7] Laurent Grémy. *Algorithmes de crible pour le logarithme discret dans les corps finis de moyenne caractéristique*. PhD thesis, Université de Lorraine, 2017.

[8] Antoine Joux and Reynald Lercier. Improvements to the general number field sieve for discrete logarithms in prime fields. a comparison with the Gaussian integer method. *Mathematics of Computation*, 72(242) :953–967, 2003.

[9] Thorsten Kleinjung, Claus Diem, Arjen K Lenstra, Christine Priplata, and Colin Stahlke. Computation of a 768-bit prime field discrete logarithm. In *Eurocrypt*, pages 185–201. Springer, 2017.

[10] Hendrik W Lenstra Jr. Factoring integers with elliptic curves. *Annals of mathematics*, pages 649–673, 1987.

[11] Brian Antony Murphy. *Polynomial selection for the number field sieve integer factorisation algorithm*. Australian National University, 1999.

[12] Oliver Schirokauer. Virtual logarithms. *Journal of Algorithms*, 57(2) :140–147, 2005.

[13] The CADO-NFS Development Team. CADO-NFS, An Implementation of the Number Field Sieve Algorithm, 2017. Release 2.3.0.

Sixième partie

Annexes

A Fichier de paramètres sélection polynomiale Kleinjung

```
################################################################################
# General parameters
################################################################################

name = p70

################################################################################
# Polynomial selection with Kleinjung's algorithm
################################################################################

tasks.polyselect.degree = 4

tasks.polyselect.admax = 2000
tasks.polyselect.incr = 60
tasks.polyselect.adrange = 1000
tasks.polyselect.P = 2000
tasks.polyselect.nq = 1000
tasks.polyselect.ropteffort = 0.01

################################################################################
# Sieve
################################################################################

tasks.I = 11
lim0 = 50000
lim1 = 50000
lpb0 = 18
lpb1 = 18
tasks.sieve.mfb0 = 36
tasks.sieve.mfb1 = 36
tasks.sieve.lambda0 = 2.2
tasks.sieve.lambda1 = 2.2
tasks.sieve.qmin = 15000
tasks.sieve.qrange = 2500
tasks.sieve.rels_wanted = 78000

################################################################################
# Filtering
################################################################################

tasks.filter.maxlevel = 19
tasks.reconstructlog.partial=false

################################################################################
# Individual log
################################################################################

tasks.descent.init_I = 11
tasks.descent.init_ncurves = 10
```

```
tasks.descent.init_lpb = 28
tasks.descent.init_lim = 50000
tasks.descent.init_mfb = 56
tasks.descent.init_tkewness = 50000
tasks.descent.I = 11
tasks.descent.lim0 = 50000
tasks.descent.lim1 = 50000
tasks.descent.lpb0 = 18
tasks.descent.lpb1 = 18
tasks.descent.mfb0 = 36
tasks.descent.mfb1 = 36
```

B Fichier de paramètres sélection polynomiale Joux-Lercier

```
################################################################################
# General parameters
################################################################################

name = p70

################################################################################
# Polynomial selection with Joux-Lercier's algorithm
################################################################################

jlpoly = true
fastSM = true

tasks.polyselect.degree = 3
tasks.polyselect.bound = 7
tasks.polyselect.modm = 11

################################################################################
# Sieve
################################################################################

tasks.I = 11
lpb0 = 18
lpb1 = 17
lim0 = 250000
lim1 = 12500
tasks.sieve.mfb0 = 18
tasks.sieve.mfb1 = 34
tasks.sieve.lambda0 = 1.1
tasks.sieve.lambda1 = 2.1
tasks.sieve.qmin = 25000
tasks.sieve.qrange = 5000
tasks.sieve.sqside = 0
tasks.sieve.rels_wanted = 62000

################################################################################
# Filtering
################################################################################

tasks.filter.maxlevel = 25
tasks.filter.target_density = 92.5

################################################################################
# Reconstructlog
################################################################################

tasks.reconstructlog.partial=false
tasks.reconstructlog.checkdlp=false
```