An implementation and analysis of the ECPP algorithm

Gyöngyvér Kiss

PhD Thesis

Department of Computer Algebra

Eötvös Loránd University, Faculty of Informatics

Supervisor:
Antal Járai D.Sc.

PhD School of Computer Science

Dr. András Benczúr

PhD Program of Numeric and Symbolic Calculus

Dr. Antal Járai

Budapest, 2016

Contents

Acknowledgements **vi**

1 Introduction **1**

1.1 AKS test 2

1.2 Elliptic curves 4

1.3 Goldwasser–Kilian test 7

1.4 ECPP test 8

2 Theoretical results **14**

2.1 Asymptotic running time analysis of ECPP 14

2.2 Heuristics 21

2.3 Strategies 25

2.4 Experiments 29

2.5 Conclusions 40

3 Practical results **43**

3.1 Magma Computational Algebra System 43

3.2 Modified-ECPP 44

3.2.1 The first version of Modified-ECPP 46

3.2.2 The second version of Modified-ECPP 49
3.3 Experiments 54
3.3.1 Running times 54
3.3.2 Experiments on the strategy 57
3.4 Conclusions and future improvements 61

Bibliography **64**

List of Figures

2.1 $\sqrt{D}/h(D)$ as a function of D 30

2.2 $\sqrt{D}/h(D)$ as a function of D on finer scale 31

2.3 $e(n)$ and its mean as a function of D for 3400 digit numbers . 31

2.4 The mean of $e(n)$ as a function of D for $500-3400$ digit numbers 32

2.5 $e(n)$ and its mean as a function of S for 3400 digit numbers . 32

2.6 The mean of $e(n)$ as a function of S for $500-3400$ digit numbers 33

2.7 Precision of $\bar{e}$ 34

2.8 $l(m)$ as a function of $\lambda(m)$ for 3400 digit numbers using actual values e 35

2.9 $l(n)$ as a function of $\lambda(n)$ for $500-3400$ digit numbers using estimated values $\bar{e}(n)$ 35

2.10 l as a function of T for 3400 digit numbers 36

2.11 l as a function of e for 3400 digit numbers 37

2.12 $G(n)$ as the function of $\bar{G}(n)$ for 3400 digit numbers 38

2.13 $G(n)$ as the function of $\bar{G}'(n)$ for 3400 digit numbers 39

2.14 $I_b(n)$ and $I_n(n)$ as the function of $\log_{10} n$ up to 7000 digit numbers 39

2.15 Average $I_b(n)$ and $I_n(n)$ as the function of $\log_{10} n$ up to 7000 digit numbers 40

3.1 Running times 54

3.2 Average running times on logarithmic scale 55

3.3 Running time as function of k for 3000 digits numbers 55

3.4 The proportion of the execution time compared to the total running time 56

3.5 The number of backtracks and repetitions as a function of the length of the paths 58

3.6 The level and size differences of backtracks 58

3.7 The length of the repetition sequences 59

3.8 The proportion of the repetition sequences 60

3.9 The time of the repetition sequences 60

3.10 The length of the repetition sequences 61

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor, Professor Antal Járai for the continuous support of my PhD study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.

Besides my supervisor, I would like to thank Professor Wieb Bosma for his valuable help and insightful comments, but also for the hard question which incited me to widen my research from various perspectives.

Last but not the least, I would like to thank my family: my husband and my parents for supporting me spiritually throughout writing this thesis and my life in general.

Chapter 1

Introduction

The idea of prime numbers was already well known in ancient Greece. It has been a challenge since ancient times to find big prime numbers. In the recent past, more than ever, this line of research has been the focus of attention as more applications of big prime numbers emerged. They are often used in public key cryptography algorithms, hash functions and pseudo random number generators.

There are several algorithms, called primality tests, that determine whether a given number n is a prime or not. These tests can be probabilistic or deterministic.

Probabilistic tests are those primality tests for which a number passing such test is likely a prime. There are composite numbers that pass these tests but there are bounds on the probability of such event occurs. These tests frequently involve a randomly selected sample from a given space to test the primality of the input n. In general we can assert that the only possible erroneous situation can be that a composite number is reported as a prime, not vice versa. The probability of such errors can be reduced to a certain accuracy by repeating the test with independent random samples. If we reach the required accuracy by repeating the test with input n, and n is not reported as composite, then we say n is probably prime. As examples we can mention here the Miller-Rabin test and Solovay-Strassen test.

Deterministic tests deterministically distinguish whether the input number n is prime or not and provide a proof of primality. To implement a general, deterministic, unconditional primality test we need to involve time consuming computations. There is still no algorithm that fulfills all these adjectives, runs in polynomial time and is efficient in practice. If we drop some of our expectations, we can come up with solutions. For example the Pocklington test [19] is deterministic and unconditional, but requires a partial factorization of $n - 1$. The cyclotomy test of Adleman, Pomerance and

Rumely [1], APR in short, works much better in practice. However the running time of APR is $O\big((\ln n)^{c \ln \ln \ln n}\big)$. Despite of the later improvements it is still not possible to prove to have polynomial running time. In contrast, the running time of the version of the Miller–Rabin test [17] that is deterministic under the extended Riemann hypothesis, can be proven to be $\tilde{O}(\ln^4 n)$, but in practice, it is less efficient than the previous two.

There are two primality tests that are described in detail in the following. The first one is the relatively new AKS test [2], found in 2002 by Agrawal, Kayal and Saxena. The second one, that is the topic of this thesis, is the elliptic curve primality test [3], ECPP. It is the fastest algorithm in practice, though the worst case execution time of it is unknown.

1.1 AKS test

The significance of the AKS test is that it is an unconditional, deterministic and polynomial time algorithm, thus primality proving is in P follows from it. This section was written following the original paper of Agrawal, Kayal and Saxena [2]. The base idea of the algorithm is the following:

Lemma 1.1.1. *Suppose that $n \geq 2$, $n \in \mathbb{N}$, $a \in \mathbb{Z}$ and $\gcd(a, n) = 1$. Then n is a prime if and only if*

$$(x + a)^n \equiv x^n + a \pmod{n}. \tag{1.1}$$

Proof. The coefficient of x^i in $(x + a)^n - (x^n + a)$ is $\binom{n}{i} a^{n-i}$, where $0 < i < n$. If n is prime, $\binom{n}{i} = 0 \bmod n$, thus all the coefficients are 0. If n is composite, let p be a prime factor of n with $p^k \mid n$. Then $p^k \nmid \binom{n}{p}$ and is coprime to a^{n-p}, thus the coefficient of x^p modulo n is non-zero and $(x + a)^n - (x^n + a) \not\equiv 0 \pmod{n}$ □

Checking the condition 1.1 takes too much time because in the worst case we need to evaluate n coefficients. To reduce the number of the coefficients we evaluate 1.1 modulo $x^r - 1, n$ for an appropriate small r:

$$(x + a)^n \equiv x^n + a \pmod{x^r - 1, n}. \tag{1.2}$$

Now we run into the problem that there are composite n's that also satisfy the equation 1.2 for some values of a and r. However it can be proven for appropriate r, if condition 1.2 is satisfied for a number of a's, then n is a power of a prime. As the number of a's and the size of r are bounded by a polynomial in $\log n$, the algorithm is deterministic and polynomial time. The bound to the size of the appropriate r is:

Lemma 1.1.2. *There exists an* $r \leq \max\{3, \lceil \log^5 n \rceil\}$ *such that the order of* n *modulo* r, $\mathrm{ord}_r(n)$, *is greater than* $\log^2 n$.

The algorithm proceeds in the following stages with input n:

Algorithm 1.2.1. AKS(n)

(1) If $n = p^k$, where $k > 1$ and $p \in \mathbb{N}$ prime, return false.

(2) Find the smallest r such that $\mathrm{ord}_r(n) > \log^2 n$.

(3) If $1 < \gcd(a, n) < n$ for some $a \leq r$, return false.

(4) If $n \leq r$ return true.

(5) For $a = 1$ to $\lfloor \sqrt{\Phi(r)} \log n \rfloor$ check 1.2 with a and r. If for any a 1.2 does not hold, return false.

(6) Return true.

The running time of the algorithm is proved to be $\tilde{O}\left(\log^{21/2} n\right)$. It can be reduced heuristically to $\tilde{O}\left(\log^6 n\right)$ if the following two conjectures hold.

Conjecture 1.1.1. Artin's Conjecture. *Let* $n \in \mathbb{N}$ *an arbitrary number that is not a perfect square. The number of primes* p *such that* $p \leq m$ *and* $\mathrm{ord}_p(n) = p - 1$ *is asymptotically*

$$A(n) \cdot \frac{m}{\ln m},$$

where $A(n)$ *is the Artin constant with* $A(n) > 0.35$.

Definition 1.1.1. A prime p is a Sophie-Germain prime if $2p + 1$ is also a prime.

Conjecture 1.1.2. Sophie-Germain Prime Density Conjecture. *The number of Sophie-Germain primes* p *such that* $p \leq m$ *is asymptotically*

$$\frac{2C_2 m}{\ln^2 m},$$

where C_2 *is the twin prime constant that is estimated approximately* 0.66.

There is a variant of the algorithm though that runs in $\tilde{O}(\log^3 n)$ time if the following conjecture is true. Although Lenstra and Pomerance have given heuristic proof that suggests that it is false.

Conjecture 1.1.3. *If r is a prime such that $r \nmid n$ and*

$$(x-1)^n \equiv x^n - 1 \pmod{x^r - 1, n},$$

then either n is prime or $n^2 \equiv 1 \bmod r$.

There are many variants of the algorithm, for instance Pomerance and Lenstra suggested an improved algorithm too, that runs in $\tilde{O}(\log^6 n)$.

The algorithm has a huge impact on the theory of primality testing, it is not really used in practice though, because it needs a lot of storage. ECPP and APR works better in practice and they produce a primality certificate that can be verified fast and independently.

1.2 Elliptic curves

The elliptic curve primality test, ECPP for short, can prove primality for numbers with several thousand digits. To describe the algorithm in detail we need the following definitions.

Definition 1.2.1. An *algebraic curve* over a field F is a set

$$G = \{(x, y) : p(x, y) = 0\},$$

where p is a bivariate polynomial over the field F. The order of p is the order of G.

Definition 1.2.2. An *elliptic curve* E modulo n, where $n \in \mathbb{Z}$ and $\gcd(n, 6) = 1$, is a nonsingular cubic algebraic curve, that is given by the equation $y^2 = x^3 + ax^2 + b$, where $a, b \in (\mathbb{Z}/n\mathbb{Z})$ and the discriminant $\Delta = 4a^3 + 27b^2$ is nonzero. This is abuse of the language a bit, because $(\mathbb{Z}/n\mathbb{Z})$ is a field only if n is a prime.

Cubic curves of the form $ay^2 + bxy + cy = x^3 + ex^2 + fx + g$ over a field with characteristic $\neq 2, 3$ can be transformed (on the projective plain) into the form $y^2 = x^3 + ax + b$.

If point (x, y) is on the curve E, that is defined by $y^2 = x^3 + ax^2 + b$, it is clear that $(x, -y)$ will be on the curve too. If a non-vertical line crosses two points of a curve, it will cross a third one too. In case of a tangent we consider two intersections to be overlapping. If the two intersections are (x_1, y_1) and (x_2, y_2), the coordinates of the third one will be

$$x_3 = \lambda^2 - x_1 - x_2,$$

$$y_3 = \lambda (x_3 - x_1) + y_1$$

where $\lambda = (y_2 - y_1) / (x_2 - x_1)$, if $x_1 \neq x_2$ and $\lambda = (3x_1^2 + a) / (2y_1)$ otherwise.

Definition 1.2.3. Let (x_1, y_1) and (x_2, y_2) are two points of the elliptic curve E modulo n. We can define a *partial addition* $(x_1, y_1) + (x_2, y_2) = (x_3, -y_3)$ on E, where x_3 and y_3 are the same as defined above. We define the *zero element* O, as the point in infinity; $(x, y)+(x, -y) = (x, -y)+(x, y) = O$.

If n is prime, the set of points of the curve will form an Abelian group, with the addition, which is not partial in this case. If $\gcd(n, 6) = 1$, but n is composite, we still be able to define the partial addition, but then the points of the curve will not form a group. If n is coprime to 6, let $R = P + Q$, applying the partial addition on P, Q, that are points on an elliptic curve $E = E_{a,b}[\mathbb{Z}/n\mathbb{Z}]$. It can be shown that for any prime divisor p of n, $R_p = P_p + Q_p$ in the group $E_{\bar{a},\bar{b}}[\mathbb{Z}/p\mathbb{Z}]$, where R_p, Q_p and P_p are obtained by reducing the coordinates of R, Q and P modulo p, furthermore, $\bar{a} \equiv a \bmod p$ and $\bar{b} \equiv b \bmod p$.

Using the partial addition algorithm repeatedly, it is possible to obtain a partial multiplication algorithm by integers. If $P = (x, y)$, we can get the first coordinate of $2P = (x_2, y_2)$;

$$x_2 = \frac{(3x^2 + a)^2}{4y^2} - 2x.$$

Similarly, it is possible to get the first coordinate of $2iP = (x_{2i}, y_{2i})$ from the first coordinate of $iP = (x_i, y_i)$. Furthermore we can calculate the first coordinate of $(2i + 1) P = (x_{2i+1}, y_{2i+1})$ from the first coordinate of iP, $(i + 1) P = (x_{i+1}, y_{i+1})$ and P, if $x_i \neq x_{i+1}$ and $\gcd(x, n) = 1$;

$$x_{2i+1} = \frac{(a - x_i x_{i+1})^2 - 4b (x_i + x_{i+1})}{x (x_i - x_{i+1})^2}$$

These formulas make it possible to use the first coordinates only, while multiplying with integers.

Definition 1.2.4. A *projective plane* modulo n, $\mathbb{P}^2(\mathbb{Z}/n\mathbb{Z})$, where $n > 0$, $n \in \mathbb{Z}$, consists of equivalence classes $(X : Y : Z)$ of triplets $(X, Y, Z) \in (\mathbb{Z}/n\mathbb{Z})^3$, satisfying $\gcd(X, Y, Z, n) = 1$, under equivalence

$$(X, Y, Z) \sim (\lambda X, \lambda Y, \lambda Z)$$

for any unit $\lambda \in (\mathbb{Z}/n\mathbb{Z})$.

It is much more convenient to represent the points of an elliptic curve $E[\mathbb{Z}/n\mathbb{Z}]$ with equivalence classes $(X : Y : Z)$ of the projective plane. $P = (x, y)$, will be associated with the class $(x : y : 1) \in P^2(\mathbb{Z}/n\mathbb{Z})$, and let $O = (0 : 1 : 0)$. In this case the equation of E will be $ZY^2 = X^3 + aXZ^2 + bZ^3$. Using this representation it is possible to avoid the division while applying the partial multiplication algorithm;

$$X_{2i} = \left(X_i^2 - aZ_i^2\right)^2 - 8bX_iZ_i^3,$$

$$Z_{2i} = 4Z_i\left(X_i^3 + aX_iZ_i^2 + bZ_i^3\right),$$

$$X_{2i+1} = Z\left(\left(X_iX_{i+1} - aZ_iZ_{i+1}\right)^2 - 4bZ_iZ_{i+1}\left(X_iZ_{i+1} + X_{i+1}Z_i\right)\right),$$

$$Z_{2i+1} = X\left(X_{i+1}Z_i - X_iZ_{i+1}\right)^2.$$

After a brief outline of the necessary background, two theorems follow; the first theorem is needed to prove the second theorem that is the basic idea of the Goldwasser–Kilian test and the ECPP test. In the rest of this study, writing about the sum and multiplies of points on elliptic curves modulo n, we mean the result of the partial addition and multiplication, which in exceptional cases means that a divisor of n is found, rather than a point, as n is only a probable prime.

Theorem 1.2.1. Hasse's theorem. *The order m of an elliptic curve E over $\mathbb{Z}/p\mathbb{Z}$, where $p > 3$ prime, will be $p + 1 - 2\sqrt{p} < m < p + 1 + 2\sqrt{p}$.*

Theorem 1.2.2. *Let $n \in \mathbb{N}$, such that $\gcd(6, n) = 1$, and E an elliptic curve over $\mathbb{Z}/n\mathbb{Z}$. Let $m, n' \in \mathbb{Z}$ with $n' \mid m$. Suppose that we have found a point $P \in E$ such that for every prime factor q of n' holds $mP = O$ and $(m/q) \cdot P \neq O$. Then for every prime factor p of n holds $\#E[\mathbb{Z}/p\mathbb{Z}] \equiv 0 \bmod n'$. Furthermore, if $n' > (\sqrt[4]{n} + 1)^2$, then n is prime.*

Proof. Let p be a prime factor of n, and let

$$Q = \frac{m}{n'}P_p \in E[\mathbb{Z}/p\mathbb{Z}].$$

Then

$$n'Q = mP_p = (mP)_p = O,$$

therefore the order of Q is a divisor of n'. If q is a prime factor of n', then

$$\frac{n'}{q}Q = \frac{m}{q}P_p = \left(\frac{m}{q}P\right)_p \neq O,$$

as

$$\frac{m}{q}P \neq O,$$

therefore the order of Q is not a divisor of n'/q. As q was arbitrary, the order of Q is n', thus $\#E[\mathbb{Z}/p\mathbb{Z}] \equiv 0 \bmod n'$

According to Hasse's theorem $\#E[\mathbb{Z}/p\mathbb{Z}] = p+1-t$, with $t \in \mathbb{Z}$, $|t| < 2\sqrt{p}$. From this $\left(\sqrt{p}+1\right)^2 > \#E[\mathbb{Z}/p\mathbb{Z}]$. If $n' > (\sqrt[4]{n}+1)^2$, then $\left(\sqrt{p}+1\right)^2 > (\sqrt[4]{n}+1)^2$, thus $p > \sqrt{n}$. □

1.3 Goldwasser–Kilian test

To prove the primality of n probable prime using Theorem 1.2.2 we need to choose an elliptic curve E over $\mathbb{Z}/n\mathbb{Z}$ and determine the order of the curve, m. If m can be written in the form $m = fn'$, where the factors of f are known and n' is a probable prime - we call such m's *almost smooth* - furthermore $n' > (\sqrt[4]{n}+1)^2$ then the proof of primality of n follows from Theorem 1.2.2. Namely if we find a point $P \in E$ that fulfills $fP \neq O$ and $mP = O$, then n will be prime if n' is prime, as in this case the only prime factor of n' that we have to check is n' itself. Then apply the same procedure to $n_1 = n'$, and so forth. This way we are generating a sequence of n_i, where $i = 1, \ldots, l$ and n_l is below a limit, L, where we can prove primality easily. This idea comes from Goldwasser and Kilian.

Given a probable prime n as input the Goldwasser–Kilian algorithm proceeds in the following stages (see [8]):

Algorithm 1.2.2. Goldwasser–Kilian(n)

(1) If $n < L$ test n for primality. If n is prime, return true.

(2) Select random $a, b \in \mathbb{Z}/n\mathbb{Z}$ until $\gcd(4a^3 + 27b^2, n) = 1$ and $m = \#E_{a,b}[\mathbb{Z}/n\mathbb{Z}]$ is even.

(3) Let $n' = m/2$. If 2 or 3 is a factor of n', go back to 2. Test the primality of n' with a probabilistic primality test. If that returns composite, go back to 2.

(4) Select random $P \in E_{a,b}[\mathbb{Z}/n\mathbb{Z}]$

(5) If $(m/n')\,P = O$, go back to 4.

(6) Store (n, a, b, P, m, n'), let $n = n'$ and go back to 1.

Goldwasser and Kilian used only 2 to divide the curve orders. In practice it is better to use a bigger set of primes, we find the small factors of the curve orders using trial division. The bottleneck of this algorithm is to determine the order of the elliptic curve; Goldwasser and Kilian prefer Schoof's algorithm, which is still very cumbersome. The running time of Schoof's algorithm is $O\left(\ln^8 n\right)$, using fast arithmetic it can be reduced to $\tilde{O}(\ln^5 n)$. Elkies and Atkin improved the algorithm and reduced the heuristic running time to $\tilde{O}(\ln^4 n)$ using fast arithmetic, but it is probabilistic and the complexity in practice is still too high ([20]).

1.4 ECPP test

Atkin and Morain described an algorithm that is also recursively based on Theorem 1.2.2, but as opposed to the Goldwasser–Kilian algorithm, it avoids counting points of the elliptic curves. To give an outline of the algorithm, some definitions and details are necessary. This section follows the original paper of Atkin and Morain [3].

In the rest of the thesis $\ln^k n$ shall denote $(\ln n)^k$, $\ln \ln^k n$ shall denote $(\ln \ln n)^k$, and so on.

Definition 1.4.1. D is a *negative fundamental discriminant*, that means that $D < 0$, that $D \equiv 0 \bmod 4$ or $D \equiv 1 \bmod 4$ and D/f^2 is not a discriminant, for any $f > 1$ integer.

The cases $D = -3$ and $D = -4$ requires special treatment, thus we assume that $D \leq -7$.

Definition 1.4.2. The quadratic form $ax^2 + bxy + cx^2$ of a negative fundamental discriminant D, is given by the 3-tuple (a, b, c), where $a, b, c \in \mathbb{Z}$ and $b^2 - 4ac = D$. We identify the quadratic form with the 3-tuple.

For each quadratic form $Q = (a, b, c)$ there is a corresponding $2x2$ matrix

$$M(Q) = \begin{bmatrix} a & b/2 \\ b/2 & c \end{bmatrix}$$

Two forms Q and Q' are equivalent if there exists $N \in SL_2(\mathbb{Z})$ such that $M(Q') = N^{-1} M(Q) N$.

Definition 1.4.3. A form $Q = (a, b, c)$ is called *positive* if $a > 0$ and its discriminant is negative, it is called *primitive* if a, b and c are coprime and it is called *reduced* if $|b| \leq a \leq c$ and if $b \geq 0$ then $|b| = a$ otherwise $a = c$.

Theorem 1.4.1. Gauss's theorem. *Each equivalence class contains exactly one reduced form.*

Definition 1.4.4. The set of primitive reduced forms of discriminant D, denoted by $H(D)$, is a finite Abelian group for the operation called composition of classes. The order of $H(D)$ is denoted by $h(D)$. The neutral element is called the *principal* form and it is equal to $(1, 0, -D/4)$ if D is even, and to $\left(1, 1, (-D+1)/4\right)$ otherwise.

Let $C = (a, b, c) \in H(D)$, $(x, y) \in \mathbb{Z}$ and $C(x, y) = ax^2 + bxy + cy^2$ with $\gcd(a, D) = 1$. In general it is possible to replace the variables x, y with new variables. If we consider the quadratic form $C(x, y) = ax^2 + bxy + cy^2$, and we apply the substitutions

$$x' = -y, y' = x, \tag{1.3}$$

the new coefficients of the new form will be $a' = c$, $b' = -b$ and $c' = a$.

If we apply substitutions

$$x' = x + ky, y' = y, \tag{1.4}$$

where $k \in Z$, the new coefficients will be $a' = a$, $b' = b - 2ka$ and $c' = c - kb + k^2 a$.

Clearly applying these substitutions the range and the discriminant of the quadratic form will not change and every positive primitive form can be reduced by repeating them. An algorithm to reduce positive primitive form $C(x, y) = ax^2 + bxy + cx^2$ works as follows:

Algorithm 1.4.1. Reduction($C(x, y)$)

(1) If $C(x, y)$ is reduced then return (a, b, c).

(2) Apply 1.4 until $-a \leq b < a$.

(3) If $C(x, y)$ is reduced then return (a, b, c).

(4) If $C(x, y)$ is not reduced then apply 1.3.

(5) If $C(x, y)$ is not reduced go to 2.

(6) Return (a, b, c).

Consider the equation $n = ax^2 + bxy + cy^2$, where n is a prime. This equation has a solution only if

$$\left(\frac{D}{n}\right) = 1, \tag{1.5}$$

$$\left(\frac{n}{p_i}\right) = \left(\frac{a}{p_i}\right), \quad 1 < i \leq t, \tag{1.6}$$

We can write the factorization of D as $p_1^* \dots p_t^*$, where $p_i^* = (-1)^{\left(p_i^*-1\right)/2} p_i$ if p_i is an odd prime and -4 or -8 otherwise. Let $f_i(C) = \left(\frac{a}{p_i}\right)$ for $1 \leq i \leq t$ and let $F : H(D) \to \{\pm 1\}^t$, $F(C) = (f_1(C), \dots f_t(C))$.

Theorem 1.4.2. *F is a homomorphism and F is onto. The associated cosets, called the genera, are forming a group. The cardinality of the cosets is $e = h/g$, where $g = 2^{t-1}$.*

Definition 1.4.5. It is known that $\mathbb{Q}\left(\sqrt{D}\right)$ is quadratic field of degree two over $\mathbb{Q}$, where D is a negative fundamental discriminant. The algebraic integers ν of $\mathbb{Q}\left(\sqrt{D}\right)$ can be written in the form $\nu = x + y\omega$, where $x, y \in \mathbb{Z}$ and $\omega = \left(D + \sqrt{D}\right)/2$. The *conjugate* of ν is $\bar{\nu} = \tau(\nu) = x + y\tau(\omega)$, where τ is the complex conjugation. The *norm* of ν is $N_{\mathbb{Q}\left(\sqrt{D}\right)}(\nu) = \nu \cdot \bar{\nu}$

Definition 1.4.6. The *Hilbert-polynomial* $H_D(x)$ is a degree $h(D)$ polynomial with integer coefficients. It is defined by the product:

$$H_D(x) = \prod_{(a,b,c)} \left(x - j\left(\frac{b + \sqrt{D}}{2a}\right)\right),$$

where (a, b, c) runs through all the positive primitive reduced forms that belong to D and j is a fixed complex function:

$$j(z) = \frac{\left(1 + 240\sum_{k=1}^{\infty} k^3 q^k / \left(1 - q^k\right)\right)^3}{q\prod_{k=1}^{\infty}\left(1 - q^k\right)^{24}},$$

where $q = e^{2\pi i z}$. It can be proved that

$$j(z) = \frac{1}{q} + 744 + \sum_{k=1}^{\infty} c_k q^k,$$

where the coefficients $c_k \in \mathbb{Z}$, $c_k > 0$.

After providing the necessary background, we describe the ECPP algorithm in detail, given a probable prime n as input. As we saw in the previous section that determining the order of a random elliptic curve is quite cumbersome, the improvement is to find an appropriate curve order first and

determine the belonging elliptic curve later. It can be achieved by finding an algebraic integer $\nu \in \mathbb{Q}\left(\sqrt{D}\right)$, with $|\nu|^2 = n$, where D is a negative fundamental discriminant. If we have such ν, then it is relatively easy to find elliptic curves with order $m = |\nu \pm 1|^2$. In this case $|\nu|^2$ is the norm of ν, thus, if $\nu = x + y\omega$, we are looking for the solution of

$$\left(x + y\frac{D}{2}\right)^2 - y^2\frac{D}{4} = n$$

Reordering the equation we get

$$C(x,y) = x^2 + xyD + y^2\frac{D(D-1)}{4} = n \tag{1.7}$$

Equation 1.7 can be solved if

$$\left(\frac{D}{n}\right) = 1, \tag{1.8}$$

$$\left(\frac{n}{p}\right) = 1, \quad 1 < i \leq t, \tag{1.9}$$

where $D = p_1 \dots p_t$. This comes from the conditions 1.5 and 1.6, with $a = 1$. Note that these conditions are necessary but not sufficient conditions, but they make it possible to filter the discriminants in advance that are not appropriate for n.

Algorithm 1.9.1. Determine-Next-Input(n, D)

(1) If 1.7 has a solution, there will be a $C'(x,y) = a'x^2 + b'xy + c'y^2$ that is equivalent to $C(x,y)$, and $C'(1,0) = n$. Coefficient a' must be n, $b'^2 - D$ must be a factor of $4n$, otherwise c' is not an integer. Take $\bar{b}^2 \equiv D \bmod n$, $0 < \bar{b} < n$ and let $b' = \bar{b}$ if $\bar{b}$ and D has the same parity and $b' = \bar{b} + n$ otherwise. Let $c' = (b'^2 - D)/(4n)$.

(2) Reduction$(C(x,y))$.

(3) Reduction$(C'(x,y))$.

(4) If the reduced form of $C(x,y)$ and $C'(x,y)$ is not the same, 1.7 has no solution, return NULL.

(5) Follow the steps of the reduction of $C(x,y)$ backwards to get the required x and y.

(6) Determine $\nu = x + y\omega$, $m_+ = |\nu + 1|^2$ and $m_- = |\nu - 1|^2$.

(7) Try to factor m_+ and m_-.

(8) If neither m_+ nor m_- is almost smooth with $m_\pm = fn'$, and $n' > \left(\sqrt[4]{n} + 1\right)^2$ return NULL.

(9) Let the successful $m_\pm$ be m. Store (n, D, m, n'). Return n'.

In 2. reducing $C(x, y)$ is only one step. If D is even, the reduced form of $C(x, y)$ is $x^2 - (D/4)\, y^2$, otherwise it is $x^2 + xy + y^2 (1 - D)/4$.

Note that there three bottlenecks in this algorithm; extracting the square root of $\bar{b}$ modulo n from 1., the reduction algorithm from 3. and the factoring from 7. The success of the first two; the solubility of 1.7, is equivalent to the ideal $\left(n, (b - \sqrt{D})/2\right)$ being principal in the ring of integers of $\mathbb{Q}\left(\sqrt{D}\right)$. If the ideal class is assumed to be random, this happens with probability $1/(2h(D))$, where $h(D)$ is the ideal class number from 1.4.4.

Algorithm 1.9.2. Downrun(n)

(1) If $n < L$ test n for primality. If n is prime, return.

(2) Select a D that is appropriate for n from a relatively big set of negative fundamental discriminants.

(3) $n' =$Determine-Next-Input(n, D).

(4) If n' is NULL then go back to 2.

(5) Let $n = n'$ and go back to 1.

The algorithm Downrun(n) results in a list of tuples (n_i, D_i, m_i, n_i'), where $i = 0, \ldots, l$. Repeated call of the following function loops through the list of tuples (n_i, D_i, m_i, n_i') to determine the elliptic curves E_i with order m_i and the points P_i on the curves E_i that satisfy the conditions of Theorem 1.2.2.

Algorithm 1.9.3. Generate-Proof (n, D, m, n')

(1) Determine the Hilbert-polynomial H_D mod n for D.

(2) Determine an arbitrary root x_0 of H_D mod n.

(3) Elliptic curves with orders $m_\pm = |\nu \pm 1|^2$ will be $y^2 = x^3 + 3kx + 2k$ and $y^2 = x^3 + 3kc^2x + 2kc^3$, where $k \equiv x_0/(1728 - x_0) \bmod n$, $(c/n) = -1$. Find out which elliptic curve belongs to m and let it be E.

(4) Find a point P on the appropriate elliptic curve that satisfies the conditions in 1.2.2.

(5) Return (n, m, E, P, n')

The Generate-Proof (n, D, m, n') algorithm thus constructs the elements of the proof; a list of tuples $(n_i, m_i, E_i, P_i, n_i')$. This list will serve as a prime certificate that can be verified by showing that the given points lie on the given curves and have the given orders.

After determining the proof, we need to verify it. It is done as follows for one element of the proof list.

Algorithm 1.9.4. Proof-Verification(n, m, E, P, n')

(1) Verify that $n' \mid m$, $(m/n')P \in E$, $mP \in E$, $(m/n')P \neq O$ and $mP = O$, where E is the given elliptic curve mod n.

The heuristic running time of ECPP is $O\left(\ln^{6+\varepsilon}\right)$ [16], but it can be reduced to $O\left(\ln^{4+\varepsilon}\right)$ [18], [16]. In the next chapter we prove that it is possible to decrease the heuristic running time to $o(\ln^4 n)$ using the fastest known algorithms for various parts and applying refinements.

Chapter 2

Theoretical results

2.1 Asymptotic running time analysis of the ECPP algorithm

In this chapter we give a detailed analysis of the running time of the ECPP algorithm. The following three sections are written according to our paper [5]. The content of these sections are mostly joint results of Antal Járai and Gyöngyvér Kiss unless it is stated otherwise.

We drop the index of n_i in the course of this chapter and use only n, unless it is necessary to keep it, because we are not talking about a list of n_i for the time being. Our notations are according to Theorem 1.2.2.

As the analysis strongly depends on the complexity of the integer multiplication, $m(k)$ will denote the time that we need to multiply arbitrary k-bit numbers, to keep our calculations independent from the selected fast multiplication method. Multiplication is of the same order as division and squaring and exceed the complexity of addition, subtraction and multiplications by powers of 2. Applying the Fast Fourier Transformation and the Schönhage-Strassen method for fast multiplication will allow us to keep this time at $O(k \ln k \ln \ln k)$.

There are three main parameters that we use during the Downrun (see 1.9.2). Two of them have been already present in the implementation of Atkin and Morain [3], but they are used differently.

Parameter d – In stage 2 of the algorithm Downrun(n) we select a set of fundamental discriminants D. In order to control the size of this set, we apply an upper bound d on the size of the discriminants. In stage 3 we need to perform a reduction for essentially every discriminant that is suitable for the current input, as well as a factorization and a primality test on each curve order and n' that were produced processing the suitable discriminants; thus

the number of the selected discriminants has a huge impact on the running time.

Parameter s – In stage 3 we also have to extract the square root of discriminants D modulo n, which can be done faster if we extract the modular square roots of all the possible prime divisors of the D's instead. An upper bound s on the size of the factors of the discriminants can control the size of the set on which we have to perform the square root extraction modulo n. The size of s also has an effect on the number of the discriminants, as we throw away everything that is not s-smooth.

Parameter b – One of the bottlenecks is factoring the curve orders, performed in stage 3. There are two ways to control the running time of the factoring. The first one, mentioned above, is to control the size of the discriminant set through d and s, but we can also restrict the set of primes that we use to factor the curve orders. The bound on these primes is b.

Most of the ECPP implementations use these parameters as fixed limits. For example in [3], d is taken to be 10^6, for practical purposes. In our case they are of the form

$$a \ln^{c_1} n \ln\ln^{c_2} n,$$

where $n = n_i$ is the input probable prime of the i^{th} iteration. The three parameters shall be denoted by $d(n)$, $s(n)$ and $b(n)$, to indicate that they depend on n.

First we give a detailed analysis of the algorithm ECPP.

***D*$_1$ - Selection of the discriminants.** In this stage we need to select D's from a set of fundamental discriminants. As we need the square root of these discriminants modulo n in stage 3, which is one of the bottlenecks of the algorithm, we have to keep the number of square root calculations reasonable. To achieve that we will consider only $s(n)$-smooth discriminants below $d(n)$ for some $d(n)$ and $s(n)$.

As a standard example take $d(n) \asymp \ln^2 n$. We consider only discriminants of the classes -3, -4, -7, -8, -11 and -15 from the residue classes modulo 16, as the numbers from these classes that are free from a square of any odd primes, are fundamental discriminants. It is possible to prove that only these are the fundamental discriminants. We estimate the density of such numbers. The density of the squarefree numbers is $\prod_{p\in\mathbb{P}}(1-1/p^2)$ having reciprocal

$$\prod_{p\in\mathbb{P}}(1-\frac{1}{p^2})^{-1} = \prod_{p\in\mathbb{P}}(1+\frac{1}{p^2}+\frac{1}{p^4}+\cdots) = \sum_{n\in\mathbb{N}^+}\frac{1}{n^2} = \frac{\pi^2}{6}.$$

If we take x as the density of numbers that are squarefree and odd, we have $6/\pi^2 = 3/4 \cdot x$ as all such numbers reside only in 3 of the residue classes modulo 4. Additionally the fact that a number is odd and squarefree is independent from its residue class modulo 4, thus the density of such numbers in a residue class is the same as in all the integers. Thus the density of odd squarefree numbers x is

$$\frac{6}{\pi^2} \cdot \frac{4}{3} = \frac{8}{\pi^2},$$

They reside in six residue classes modulo 16, we get asymptotically at least $3d(n)/\pi^2$ fundamental discriminants below $d(n)$. As computing the modular square root of all fundamental discriminants up to $d(n)$ is too time-consuming, we consider only those that are $s(n)$-smooth, with some $s(n) \asymp d(n)^c$, $0 < c < 1$. In this case we compute the modular square root of each prime p up to $s(n)$, for which $(n|p) = 1$. That is approximately half of the primes below $d(n)^c$, around $O\left(d(n)^c / \ln d(n)\right)$ primes. Let $d(n) \asymp \ln^2 n$, and the running time of computing a square root modulo n is $\tilde{O}(\ln^2 n)$ according to [6]. To determine the modular square roots of the discriminants up to $d(n)$ takes time

$$\frac{\ln^2 n \ln\ln n \ln\ln\ln n \ln^{2c} n}{\ln\ln n} = \ln^{2c+2} n \ln\ln\ln n.$$

Thus we need $O\left(\ln^3 n \ln\ln\ln n\right)$ bit operations if $c = 1/2$ and $o\left(\ln^3 n\right)$ if $c < 1/2$ and we still suppose that the number of successfully factored discriminants is $O\left(\ln^2 n\right)$.

To check all the discriminants up to $d(n)$ whether they are $d(n)^c$ smooth and to factor them with simple trial division takes $\tilde{O}(\ln^{2+2c} n)$ bit operations even if we use the classical division procedure.

Thus if we choose $c < 1/2$, the time of this stage can be neglected compared to the other stages.

***D_2* - Modular square roots.** First we check each $d(n)^c$-smooth discriminant D up to $d(n)$ whether it is appropriate for n, namely D satisfies the conditions 1.8 and 1.9. The probability that these conditions are satisfied is 2^{-t}, where $D = p_1 \dots p_t$, if we consider -4 and -8 as prime factors, but -1 not. The average number of factors up to x has normal distribution with expected value and variance $\ln\ln x$, see in [11] Erdős-Kac theorem, thus we obtain that this probability is

$$2^{-\ln\ln d(n)} = (e^{-\ln\ln d(n)})^{\ln 2} = \frac{1}{\ln^{\ln 2} d(n)}.$$

If $d(n) \asymp \ln^{c_1} n \ln\ln^{c_2} n$ with $c_1 > 0$, this probability is $\asymp 1/\ln\ln^{\ln 2} n$. The time of this calculation can be neglected as $(n|p)$ is already checked. One of the bottleneck of this stage is the next step; calculating the square root of the discriminants modulo n. This needs

$$O\big(m(\ln n)d(n)/\ln^{\ln 2} d(n)\big)$$

time, if we precalculate the square root modulo n of all $\pm k \in \mathbb{Z}$ for $k \leq \sqrt{d(n)}$, then at most three modular multiplications gives the modular square root of D; if D has four factors, the product of the least two is not greater than $\sqrt{d(n)}$.

D_3 - Reduction of quadratic forms. To get the curve orders we have to reduce the quadratic forms that we have gained for the remaining discriminants. This can be done with applying a refined version of the reduction algorithm 1.4.1 or the Cornacchia algorithm [6]. The Cornacchia algorithm consists of three main stages:

Algorithm 2.0.1.

(1) Compute Kronecker symbol $(D|n)$

(2) Compute the square root of D modulo n.

(3) Controlled Euclidean descent algorithm.

In our case stage 1 and 2 were already done in $\boldsymbol{D_1}$ and $\boldsymbol{D_2}$. In stage 3 we have to use controlled Euclidean descent algorithm of Schönehage [21] which runs in $O\left(m\left(\ln n\right)\ln\ln n\right)$ time instead of the usual Euclidean algorithm. If we apply the algorithm on all the remaining D's, we get running time

$$O\left(m\left(\ln n\right)d(n)\ln\ln n/\ln^{\ln 2} d(n)\right).$$

D_4 - Factorization. In this step we try to factor the curve orders that we have gained in step $\boldsymbol{D_3}$. Let $e(n)$ be the number of these curve orders. This is twice the number of the remaining discriminants. We remove the small factors of these curve orders with trial division up to $t(n)$. This requires time

$$O\left(e(n)t(n)\ln n\right).$$

If $t(n) = O(1)$, this time can be neglected.

After removing the small factors we can apply various methods to remove the large factors.

(a) Trial division. Using only simple trial division to remove the factors up to $b(n)$ takes time

$$O\left(e(n)b(n)\ln n\right).$$

(b) Batch trial division. This method is based on trial division and it is efficient in practice if we want to use a set of primes to factor a set of integers. The method was suggested by Antal Járai several years ago and appeared independently by other authors; for example [9]. Let P be the set of primes up to $b(n)$ and M be the set of integers; curve orders in our case. First we have to calculate the product of curve orders $\bar{m}$. This is done by the following algorithm.

Algorithm 2.0.2. Batch-Multiplication(M)

(1) If M has one element m return m.

(2) Pair up the elements of M and multiply them with their pairs. Then we get $\#M/2$ products. Let M_2 be the set of these products.

(3) Store M and let $M = M_2$.

(4) Go back to stage 1.

We also need the product of the primes $\bar{p} = \prod_{p\in P} p$, but as it can be reused, it is precomputed and stored in a file. We take the $\gcd(\bar{m}, \bar{p})$ then the gcd calculations are distributed to the partial products of curve orders, that were computed running Batch-Multiplication, finally to the curve orders themselves. This method surprisingly depends only really weakly on the size of the curve orders. The algorithm seems to work better here in practice, compared to other factorization methods as Pollard ρ, $p-1$, or ECM, but finding the best limit needs further investigation.

If $b(n) \geq e(n)\ln n$, splitting the product $\bar{p}$, which is of magnitude $e^{b(n)}$, to parts with the same size as the product of curve orders, we get

$$\frac{b(n)}{e(n)\ln n}$$

parts. Then we take the gcd of each part with the product of curve orders m. This takes time

$$O\left(\frac{b(n)}{e(n)\ln n} m\big(e(n)\ln n\big)\ln\big(e(n)\ln n\big)\right)$$

and size

$$O\big(e(n)\ln n\big)$$

of core space.

We can divide $\bar{p}$ with the $\bar{m}$ and take the gcd of the remainder and $\bar{m}$. This needs

$$O\left(e(n)\ln n\right)$$

core space too, but the time drops to

$$O\left(m\left(e(n)\ln n\right)\left(\frac{b(n)}{e(n)\ln n}+\ln\left(e(n)\ln n\right)\right)\right).$$

(c) Pollard ρ method. Applying this method to find factors up to $b(n)$ we need roughly $O\left(\sqrt{b(n)}\right)$ iterations and the iterations need $O\left(m\left(\ln n\right)\right)$ time. There are also gcd calculations, which take $O\left(m\left(\ln n\right)\ln\ln n\right)$ bit operations, but it might be enough to use gcd only after each $\ln\ln n$ iterations. In this case the total running time is

$$O\left(e(n)\sqrt{b(n)}m(\ln n)\right).$$

(d) Pollard $p-1$ method. If we apply the Pollard $p-1$ method, we can choose parameters to find all factors $p < b(n)$ for which $p-1$ is $\sqrt{b(n)}$-smooth. In this case it is very likely to find prime factors p such that $p \ll b(n)$, but $p \approx b(n)$ only with probability ≈ 0.34, [15]. The running time is similar to the Pollard ρ method.

(e) ECM. If we use the elliptic curve factorization method, the optimal choice to find prime factors below $b(n)$ is to choose $L_{b(n)}\left(1/\sqrt{2}\right)$ as the parameter of ECM and to use $L_{b(n)}\left(1/\sqrt{2}\right)$ elliptic curves, where

$$L_k(\beta)=e^{\beta\sqrt{\ln(k)\ln\ln(k)}}.$$

In this case we probably find prime factors below $b(n)$. The total time to factor $e(n)$ curve orders is

$$O\left(e(n)\ln b(n)m(\ln n)L_{b(n)}(\sqrt{2})\right).$$

See [16] for details.

D_5 - Miller–Rabin test. To test $e(n)$ remaining unfactored parts we apply

Miller–Rabin test [6] with a fixed number of bases. It requires

$$O\big(e(n)m(\ln n)\ln n\big)$$

bit operations.

After analyzing one step of the algorithm Downrun(n) we give details to the complexity of the algorithm Generate-Proof (n, D, m) 1.9.3.

F_1 - Hilbert polynomial. To build up the certificate of primality of n, first we have to calculate the Hilbert polynomial. The running time of this process can be neglected.

F_2 - Root of the Hilbert polynomial. To calculate the appropriate curves with the given curve orders, we have to find a root of the Hilbert polynomial. The degree of this polynomial is $h(D)$. To find a root of the polynomial we can use the splitting procedure, until a degree 1 part is obtained. To do this we have to calculate the n^{th} power of a random polynomial with degree 1 modulo the Hilbert polynomial modulo n. There is a refinement to this part of the algorithm, then the degree is $h^*(D) = h(D)/2^{t-1}$, where t is the number of factors of D. Using Hilbert polynomials, the degree $h(D)$ is certainly $\leq 2\sqrt{d(n)}\ln d(n)$, but might be $\asymp \sqrt{d(n)}$ [16]. In the refined version, the degree $h^*(D)$ might be $\asymp \sqrt{d(n)}/\ln^{\ln 2} d(n)$.

(a) In the first case the running time of one step is

$$O\big(m\,(h(D)\ln n)\ln n\big),$$

(b) in the second case it is

$$O\big(m\,(h^*(D)\ln n)\ln n\big).$$

F_3 - Find elliptic curve. From the parameters determined above we have to calculate the two elliptic curves. The only difficulty of this step is to find out which elliptic curve belongs to the selected curve order. But the time of this step still can be neglected compared to the other steps.

$$O\big(m(\ln n)\ln n\big)$$

F_4 - Find point on the curve. If we have the elliptic curve, we still have to find an appropriate point on it. As the expected number of the points to try is bounded this step can be neglected too.

P_1 - Proof verification. One step of verification of the proof needs time

$$O\big(m(\ln n)\ln n\big).$$

This can be neglected.

2.2 Heuristics

As we saw from the previous section, the number of discriminants has a huge impact on the running time and on the success of the algorithm. This amount determines the number of reductions, factorizations, produced curve orders and the probability of getting at least one almost-smooth curve order. We have to be careful. Would we process too few discriminants, the algorithm will fail, on the other hand, too many discriminants diminishes efficiency. Thus the question is, what is the optimal number of discriminants. There are well-known estimations listed in this section, from probability theory, the work of Hendrik and Arjen Lenstra [16] and the original Atkin–Morain paper [3], that can be applied to answer this question at least partially. First, we can give an estimation on the number of curve orders we get if we process a certain set of discriminants. Of course this number does not depend only on the number of the discriminants, but also on the discriminants themselves. We have given the definition of the class number $h(D)$ of the discriminant D, in 1.4.4 and also that the possibility that equation 1.7 can be solved for given $D \leq -7$ is $1/\left(2 \cdot h(D)\right)$. Thus if we consider a set of negative fundamental discriminants $\urcorner$, where the discriminants $D \leq -7$, then the expected $e(n)$; the number of the curve orders that we get is

$$\sum_{D \in \urcorner} \frac{1}{h(D)}.$$

Note that for each successful discriminant D we get two curve orders.

This expected number is independent from n so far. If we filter the discriminants for given n with the Jacobi symbol tests (1.8, 1.9), the expected $e(n)$ will be

$$\bar{e}(n) = \sum_{D \in \urcorner} \frac{2^t}{h(D)}, \tag{2.1}$$

where $D = p_1 \dots p_t$. The asymptotic behavior of this function is still unknown. We assume that for $s(n) \asymp d(n)^c$ we have $\bar{e}(n) \asymp \sqrt{d(n)}$ for some $c < 1/2$ or at least for $c = 1/2$.

We can also estimate the number of the almost smooth curve orders $l(n)$

from $e(n)$ curve orders. If we are able to find all prime factors of m that are less than $b(n)$, were m is the order of an elliptic curve modulo n, then m is a suitable candidate if the second largest prime factor of m is less than $b(n)$. If we use trial division, or batch trial division, this probability supposed to be approximately

$$e^{\gamma}\frac{\ln b(n)}{\ln n} \approx 1.7811\frac{\ln b(n)}{\ln n}.$$

Note that if we use trial division with bound $b(n)$, we can guarantee that all prime factors below $b(n)$ will be found. Thus if we factor $e(n)$ curve orders m, then the number of successfully factored m's has a probability distribution with mean approximately equal to

$$\lambda(n) = e^{\gamma}\frac{\ln b(n)}{\ln n}e(n). \tag{2.2}$$

This means that we can estimate the number of the almost smooth curve orders that we gain after processing a set of discriminants. Now the question is, what is the minimal value of $\lambda(n)$ for which the algorithm still terminates successfully.

To give partial answer to this question, first we set up a simple model of the Downrun process. If we want to optimize the algorithm then we face with difficulties, namely in the choices that we have to make at a given iteration i; assume that we can produce more than one new primes in one iteration to make it more likely that the algorithm gets to the small primes and it is also possible to backtrack to some previous prime if there are no suitable new ones. This process is similar to traversing a decision *tree*, as in each iteration we create a set of descendants (the big factors of the elliptic curve orders), and choose the one that is considered the best, which will be the input of the next iteration.

The proof of the following theorem and the statements below that are concerning our model are the results of Eric Cator (see [5]).

Theorem 2.2.1. *If the probability that the given curve order is almost smooth is small and independent from the given elliptic curve, these probabilities are independent for different curves and do not vary too much, the number of descendants is asymptotically distributed according to a Poisson-λ distribution, where λ is equal to the average ratio of suitable curves to all elliptic curves.*

Proof. Let X_j be stochastic variables, where $1 \leq j \leq N$, representing that the it^{th} curve order m_j, is almost smooth. Our assumption is that the probability of success

$$p_j = \mathbb{P}(X_j = 1)$$

is small. Let $p_j = \frac{u_j}{N}$. If X_j are independent, the expected number of almost smooth curve orders will be

$$\mathbb{E}(\sum_{j=1}^{N} X_j) = \sum_{j=1}^{N} \frac{u_j}{N} = \bar{u}.$$

We assume that this average $\bar{u}$ converges to some limit μ as $N \to \infty$.

If we want to show that the distribution of $\sum X_j$ converges to a Poisson distribution, it is enough to show that $\mathbb{E}(e^{t\sum_{j=1}^{N} X_j})$, with variable $t \in \mathbb{R}$ converges to $\mathbb{E}(e^{tY})$ for stochastic variable Y with Poisson-λ distribution.

As X_j are supposed to be independent, $P(X_j = 1) = u_j/N$ and $P(X_j = 0) = 1 - u_j/n$,

$$\mathbb{E}(e^{t\sum_{j=1}^{N} X_j}) = \prod_{j=1}^{N} \mathbb{E}(e^{tX_j}) = \prod_{j=1}^{N} \left((1 - \frac{u_j}{N})e^0 + \frac{u_j}{N}e^t\right) =$$

$$= \prod_{j=1}^{N} \left(1 + \frac{u_j(e^t - 1)}{N}\right).$$

If we take the logarithm

$$\sum_{j=1}^{N} \log(1 + u_j\frac{e^t - 1}{N}) = \sum_{j=1}^{N} \left(\frac{u_j(e^t - 1)}{N} + O(u_j^2/N^2)\right) \to \overline{u}(e^t - 1),$$

for $N \to \infty$, if we assume that $\sum_{j=1}^{N} \frac{u_j^2}{N} = o(N)$. Thus $\mathbb{E}(e^{t\sum_{j=1}^{N} X_j}) \to e^{\mu(e^t-1)}$ as $N \to \infty$.

For a stochastic variable Y with Poisson-λ distribution

$$\mathbb{E}(e^{tY}) = \sum_{j=0}^{\infty} \frac{\lambda^j}{j!}e^{-\lambda}e^{tj} = \sum_{j=0}^{\infty} \frac{(\lambda e^t)^j}{j!}e^{-\lambda} = e^{\lambda(e^t-1)}.$$

Thus $\sum X_j$, the number of almost smooth curve orders, behaves like a Poisson-distributed stochastic variable with parameter $\lambda = \mu$. □

If we still stick to our model, suppose that at each node in the decision tree, the number of the almost smooth curve orders has Poisson distribution with parameter μ_i, where i means now the i^{th} level. We would like to see that our tree has an infinite branch, which means that we can get to the small primes.

Let p_0 be the probability that our tree has an infinite branch.

$$p_0 = \mathbb{P}_{\mu_0\mu_1\cdots}(\infty-\text{branch}).$$

This probability depends on the parameters $d(n)$, $s(n)$ and $b(n)$ in the nodes. If we are at level k, we have

$$p_k = \mathbb{P}_{\mu_k\mu_{k+1}\cdots}(\infty-\text{branch}).$$

This leads to a recursion

$$p_k = \left(\sum_{\ell=1}^{\infty} \frac{\mu_k^\ell}{\ell!} e^{-\mu_k}\right)\left(1-(1-p_{k+1})^\ell\right) = 1 - e^{-\mu_k p_{k+1}},$$

where the first factor represents the probability of having at least one descendant at the k^{th} node, the second represents the probability that at least one of these descendants will have a further descendant.This is roughly the probability of such scenario when we do not have to backtrack at all; there will be always suitable descendants.

If we assume that $\mu_j = \mu$, for all j and some fixed μ, then $p = p_0 = p_k = 1 = e^{-\mu p}$. If $\mu \geq 1$, we will get a unique solution $p > 0$ and increasing the value of μ, the value of p is also increasing; the more descendants we produce, the larger the probability is, that we do not have to backtrack. This simple model is missing the concept of *cost* though. Sometimes it is not beneficial to increase μ, because the values of the parameters are so high already that it would cost less to backtrack. Therefore the aim of our second, more sophisticated model will be to introduce the concept of cost and gain and their ratio would give us a more realistic view on the process of the Downrun.

But we can still draw the conclusion; if the number of almost smooth curve orders is large enough we will succeed with a fixed positive probability. As $\lambda(n)$ from 2.2 is the parameter of the Poisson distribution, we have

$$\lambda(n) = e^\gamma \frac{\ln b(n)}{\ln n} e(n) > 1.$$

It is also possible to give an estimation of the *gain* $G(n)$; the size difference between n and n', if we use trial division or batch trial division. In these cases, if n and we can guarantee to find all prime factors below $b(n)$, the expected gain will be $\bar{G}(n) = \ln b(n)$; as for each prime p the probability that p divides a curve order m is $1/p$ with a gain $\ln p$, the expected total gain

is

$$\sum_{p\in\mathbb{P},p\leq b(n)} \frac{\ln p}{p} \sim \int_2^{b(n)} \ln x \frac{1}{\ln x}\frac{1}{x}\,dx \sim \ln b(n).$$

Using other factorization methods we may still expect that the gain only differ by a bounded factor, thus it will be also $\asymp \ln b(n)$.

The expected size of the gain gives us the opportunity to estimate the length of the path from $n = n_0$ to n_l, where n_l is sufficiently small. Let $I(n_i)$ be the length of the path from n_i to n_l. It is reasonable to assume that the expected value of $I(n)$ is

$$\bar{I}(n) = \frac{\ln n}{G(n)}.$$

2.3 Strategies

After given a detailed running time analysis of one step of the algorithm as a function of $\ln n$, $d(n)$, $s(n)$ and $b(n)$, we compute the running time of the whole algorithm. The aim of this section is to verify that with certain choices of the three parameters the running time can be reduced to $o\left(\ln^4 n\right)$.

We suppose from the previous section that:

- $s(n) \asymp d(n)^c$ for some appropriate $c < 1/2$,
- $e(n) \asymp \sqrt{d(n)}$,
- $h(D) = O\left(\sqrt{d(n)}\right)$
- $h^*(D) = O\left(\sqrt{d(n)}/\ln^{\ln 2} d(n)\right)$,
- $G(n) \asymp \ln b(n)$, and
- $I(n) \asymp \ln n/G(n)$,

From the running time analysis it was clear that the most time consuming steps are $\boldsymbol{D_3}$, $\boldsymbol{D_4}$, $\boldsymbol{D_5}$ and $\boldsymbol{F_2}$ thus we will consider only these steps analyzing the following strategies, as they determine the running time.

Strong factorization strategy. Let $d(n) \asymp \ln^2 n$. In this case $h(D) = O(\ln n)$, $h^*(D) = O(\ln n/\ln\ln^{\ln 2} n)$ and the running time of the most time consuming steps will be the following:

$$\boldsymbol{D_3}: O\big(d(n)m(\ln n)\ln\ln n/\ln^{\ln 2} d(n)\big) = O\big(\ln^3 n \ln\ln^{2-\ln 2} n \ln\ln\ln n\big),$$

$\boldsymbol{D_5}$ **:** $O\big(e(n)m(\ln n)\ln n\big) = O\big(\ln^3 n \ln\ln n \ln\ln\ln n\big)$,

$\boldsymbol{F_2(a)}$ **:** $O\big(m(h(D)\ln n)\ln n\big) = O\big(\ln^3 n \ln\ln n \ln\ln\ln n\big)$,

$\boldsymbol{F_2(b)}$ **:** $O\big(m(h^*(D)\ln n)\ln n\big) = O\big(\ln^3 n \ln\ln^{1-\ln 2} n \ln\ln\ln n\big)$,

$\boldsymbol{D_4}$ **:** In this step we have several choices of factoring algorithms, that are mentioned when describing step $\boldsymbol{D_4}$. The running time of each method depends on the choice of $b(n)$, now we choose such $b(n)$ for each of them that the running time of them will not exceed step $\boldsymbol{D_3}$, the reduction, thus we can take the running time of $\boldsymbol{D_3}$ as the total running time of one step. As $\bar{G}(n)$ and $\bar{I}(n)$ also depend on $b(n)$, we determine the total running time of the algorithm here too for each factoring method.

(a): Trial division. If we choose $b(n)$ such that it is between $\asymp \ln n$ and $\asymp \ln n \ln\ln^{2-\ln 2} n \ln\ln\ln n$ then the running time

$$O\big(e(n)b(n)\ln n\big)$$

will be between

$$O\big(\ln^3 n\big)$$

and

$$O\big(\ln^3 n \ln\ln^{2-\ln 2} n \ln\ln\ln n\big).$$

In both cases $G(n) \asymp \ln\ln n$ and $I(n) \asymp \ln n/\ln\ln n$, thus the total running time is

$$O\big(\ln^4 n \ln\ln^{1-\ln 2} n \ln\ln\ln n\big).$$

(b): Batch trial division. If we choose $b(n)$ to be between $\asymp \ln n$ and $\asymp \ln^3 n \ln\ln^{1-\ln 2} n$, then the running time

$$O\left(m\big(e(n)\ln n\big)\Big(\frac{b(n)}{e(n)\ln n} + \ln\big(e(n)\ln n\big)\Big)\right)$$

will be between

$$O\big(\ln^2 n \ln\ln^2 n \ln\ln\ln n\big)$$

and

$$O\big(\ln^3 n \ln\ln^{2-\ln 2} n \ln\ln\ln n\big).$$

The running time of the factoring does not exceed $\boldsymbol{D_3}$, $G(n) \asymp \ln\ln n$ and $I(n) \asymp \ln n/\ln\ln n$ again, therefore the total running time is the same as in **(a)**.

(c): Pollard ρ. If $b(n)$ is between $\asymp \ln n$ and $\asymp \ln^2 n \ln\ln^{2-2\ln 2} n$, then the running time

$$O\big(e(n)\sqrt{b(n)}m(\ln n)\big)$$

will be between

$$O\big(\ln^{2.5} n \ln\ln n \ln\ln\ln n\big)$$

and

$$O\big(\ln^{3} n \ln\ln^{2-\ln 2} n \ln\ln\ln n\big).$$

The time of $\boldsymbol{D_3}$ is not exceeded again, and $\bar{G}(n)$, $\bar{I}(n)$ are the same as previously thus the total running time is the same as in **(a)** or **(b)**.

(d): Pollard $p-1$. As the running time of this method is the same as the running time of Pollard ρ, this case is similar to **(c)**.

(e): ECM. Let $b(n) \asymp \ln^{\ln\ln n/\ln\ln\ln^2 n} n$, then the running time of ECM is

$$O\big(e(n)\ln b(n) m(\ln n) L_{b(n)}(\sqrt{2})\big) =$$

$$= O\big(\ln^{2+2/\sqrt{\ln\ln\ln n}} n \ln\ln^3 n/\ln\ln\ln n\big),$$

as

$$L_{b(n)}(\sqrt{2}) = e^{\sqrt{2\ln b(n)\ln\ln b(n)}} = e^{\sqrt{4(\ln\ln\ln n - \ln\ln\ln\ln n)\ln\ln^2 n/\ln\ln\ln^2 n}} <$$

$$< e^{\sqrt{4\ln\ln^2 n/\ln\ln\ln n}} = e^{2\ln\ln n/\sqrt{\ln\ln\ln n}} = \ln^{2/\sqrt{\ln\ln\ln n}} n.$$

As $\bar{G}(n) = \ln\ln^2 n/\ln\ln\ln^2 n$, $\bar{I}(n) = \ln n \ln\ln\ln^2 n/\ln\ln^2 n$, the total running time is

$$O\big(\ln^4 n \ln\ln\ln^3 n/\ln\ln^{\ln 2} n\big).$$

Our aim is to prove that the heuristic running time of the ECPP can be reduced to $o(\ln^4 n)$. As we can see we were successful only in **(e)**. If we want to reduce the other cases also to $o(\ln^4 n)$, we have two choices; we can either reduce the number of steps, $I(n)$, or reduce the time of one step. Reducing $I(n)$ can be achieved by increasing the value of $b(n)$, but then the time of factoring will possibly exceed the time of $\boldsymbol{D_3}$. Thus it seems a better solution to reduce the time of one step, by decreasing the value of the other two parameters; $d(n)$ and $s(n)$.

Strong factorization and small discriminants strategy. If we want to decrease $d(n)$ and $s(n)$, we have to be careful about the value of $\lambda(n)$; the expected number of the new descendants, should be above 1. In order to

achieve this let $d(n) \asymp \ln^2 n / \ln\ln^2 n$. In this case $h(D) = O(\ln n / \ln\ln n)$ and $h^*(D) = O\big(\ln n / \ln\ln^{1+\ln 2} n\big)$ and the running time of the critical steps will be the following:

$\boldsymbol{D_3}$ **:** $O\big(d(n)m(\ln n)\ln\ln n / \ln^{\ln 2} d(n)\big) = O\big(\ln^3 n \ln\ln\ln n / \ln\ln^{\ln 2} n\big)$;

$\boldsymbol{D_5}$ **:** $O\big(e(n)m(\ln n)\ln n\big) = O\big(\ln^3 n \ln\ln\ln n\big)$;

$\boldsymbol{F_2(a)}$ **:** $O\big(m(h(D)\ln n)\ln n\big) = O\big(\ln^3 n \ln\ln\ln n\big)$;

$\boldsymbol{F_2(b)}$ **:** $O\big(m(h^*(D)\ln n)\ln n\big) = O\big(\ln^3 n \ln\ln\ln n / \ln\ln^{\ln 2} n\big)$.

$\boldsymbol{D_4}$ **:** We want to choose the $b(n)$ so that the running time of factoring will not become the most critical step, like in the previous strategy. In this case the bottleneck is surprisingly $\boldsymbol{D_5}$, the Miller–Rabin test.

(a): Trial division. If we choose $b(n)$ such that it is between $\asymp \ln n$ and $\asymp \ln n \ln\ln n \ln\ln\ln n$ then the running time

$$O\big(e(n)b(n)\ln n\big)$$

will be between

$$O\big(\ln^3 n / \ln\ln n\big)$$

and

$$O\big(\ln^3 n \ln\ln\ln n\big).$$

In both cases $G(n) \asymp \ln\ln n$ and $I(n) \asymp \ln n / \ln\ln n$, thus the total running time is

$$O\big(\ln^4 n \ln\ln\ln n / \ln\ln n\big).$$

(b): Batch trial division. If we choose $b(n)$ to be between $\asymp \ln n$ and $\asymp \ln^3 n / \ln\ln n$, then the running time

$$O\left(m\big(e(n)\ln n\big)\Big(\frac{b(n)}{e(n)\ln n} + \ln\big(e(n)\ln n\big)\Big)\right)$$

will be between

$$O\big(\ln^2 n \ln\ln n \ln\ln\ln n\big)$$

and

$$O\big(\ln^3 n \ln\ln\ln n\big).$$

As $\bar{G}(n)$, $\bar{I}(n)$ is the same as in **(a)**, the total running will be also the same.

(c): Pollard ρ. If $b(n)$ is between $\asymp \ln n$ and $\asymp \ln^2 n$, then the running time

$$O\big(e(n)\sqrt{b(n)}m(\ln n)\big)$$

will be between

$$O\big(\ln^{2.5} n \ln\ln\ln n\big)$$

and

$$O\big(\ln^3 n \ln\ln\ln n\big).$$

Here we have again the same $\bar{G}(n)$, $\bar{I}(n)$, thus the total running time is the same as in **(a)** or **(b)**.

(d): Pollard $p-1$. This case is similar to **(c)**.

(e): ECM. Let $b(n) \asymp \ln^{\ln\ln n/\ln\ln\ln^2 n} n$, then the running time of ECM is again as in the previous strategy:

$$O\big(e(n)\ln b(n) m(\ln n) L_{b(n)}(\sqrt{2})\big) =$$

$$= O\big(\ln^{2+2/\sqrt{\ln\ln\ln n}} n \ln\ln^3 n/\ln\ln\ln n\big),$$

$\bar{G}(n) = \ln\ln^2 n/\ln\ln\ln^2 n$, $\bar{I}(n) = \ln n \ln\ln\ln^2 n/\ln\ln^2 n$, the total running time is

$$O\big(\ln^4 n \ln\ln\ln^3 n/\ln\ln^2 n\big).$$

Applying this strategy, using any of the listed factoring algorithms, the total running time will be $o(\ln^4 n)$. We have to be careful when choosing constants for $d(n)$ and $b(n)$ as our assumption is that $e(n) \asymp \sqrt{d(n)}$ and the value of

$$\lambda = e^{\gamma}\frac{\ln b(n)}{\ln n}e(n)$$

must be > 1. In **(a)**, **(b)**, **(c)** and **(d)**

$$\lambda \asymp e^{\gamma},$$

while in **(e)**

$$\lambda \asymp e^{\gamma}\frac{\ln\ln n}{\ln\ln\ln^2 n}.$$

2.4 Experiments

In the previous section we saw a collection of estimations and heuristics that are useful if we want to implement an efficient ECPP algorithm. This section

will deal with such experiments that justifies some of the results in practice. This section is the result of Gyöngyvér Kiss and follows her paper [12]. The experiments were run in Magma Computational Algebra System [4], in later sections there will be more details about it.

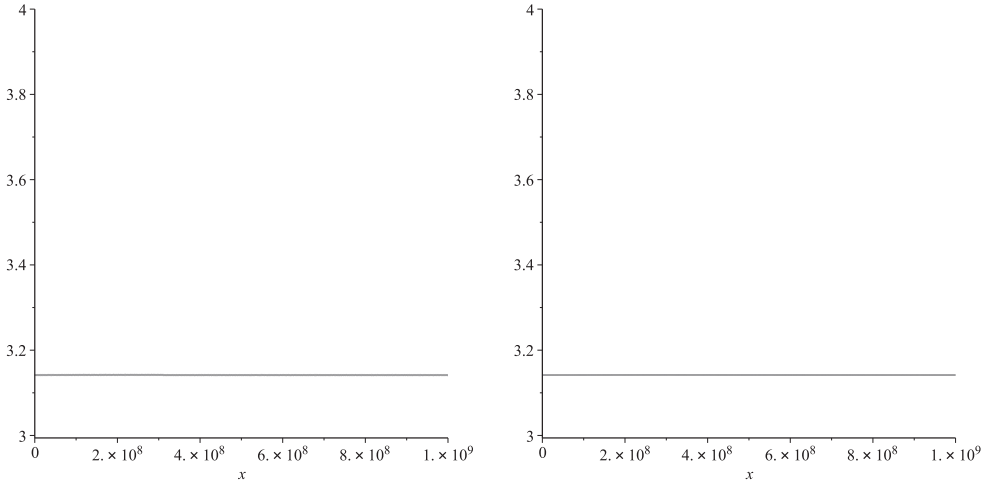

Figure 2.1: $\sqrt{D}/h(D)$ as a function of D

Our first statement is

$$h(D) = O\left(\sqrt{d(n)}\right). \tag{2.3}$$

We show the relation 2.3 between the class number $h(D)$ of D, and $\sqrt{D}$ in Figure 2.1 and 2.2, where D is a negative fundamental discriminant. The two graphs present the same data only Figure 2.2 is on a much finer scale. The graphs on the left present the average value of $\sqrt{D}/h(D)$ as a function of D. The graphs on the right present the cumulative average of the same function. The data on the discriminants is collected up to 10^9. As we expect, the curves are more or less straight lines, but on the finer scale of 2.2 we see that they seem to have a maximum at $3 \cdot 10^8$. This could be an effect of the implementation of the class number function in Magma, as the cut at that point seems to be very drastic.

As we saw it in the previous sections, the parameters $d(n)$ and $s(n)$ determine the number of the curve orders $e(n)$.

The standard example is to take $d(n) = \ln^2 n$, but it is clear from the analysis of the two strategies described above, that in this case it is hard to decrease the running time to $o(\ln^4 n)$. The second strategy suggests that we should go under $\ln^2 n$ when selecting the value of $d(n)$, but not too far. We

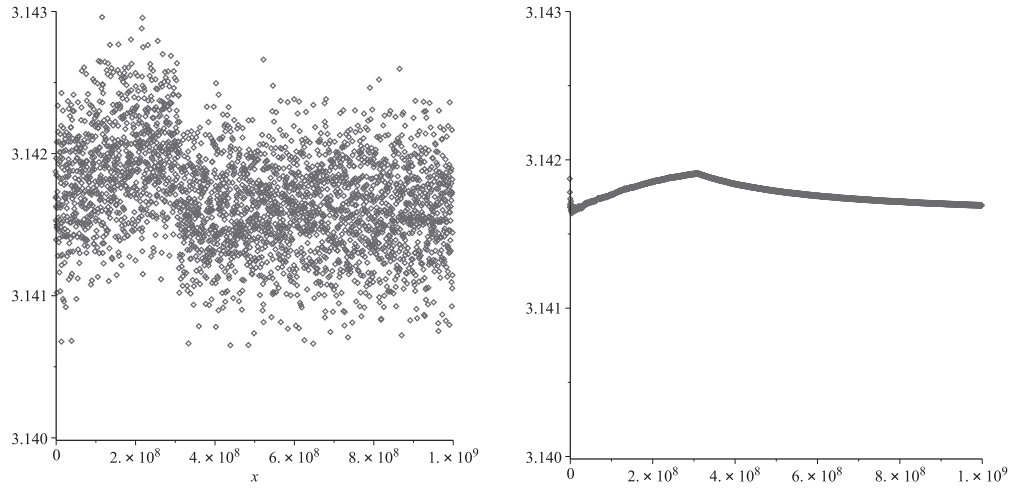

Figure 2.2: $\sqrt{D}/h(D)$ as a function of D on finer scale

would like to see, how far we can go below $\ln^2 n$ in practice and we can still make sure that the number of the curve orders does not drop considerably.

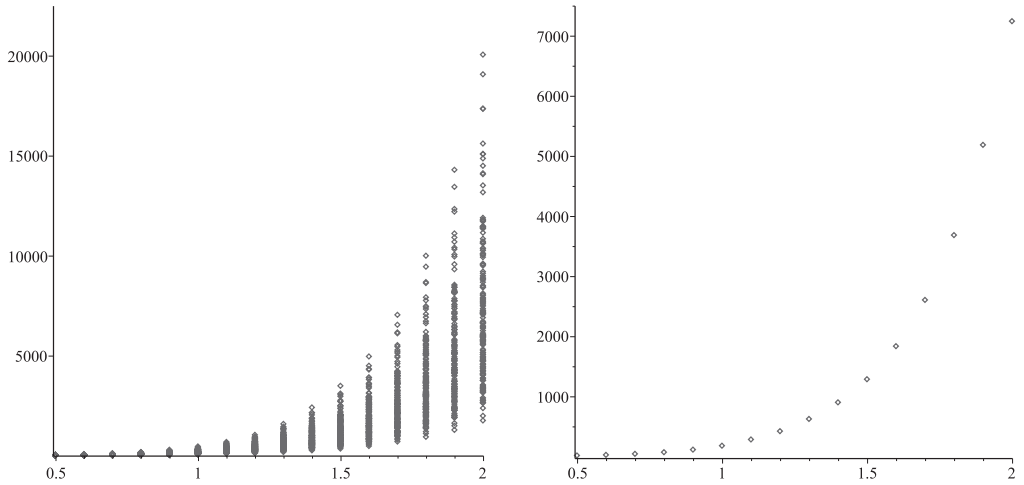

Figure 2.3: $e(n)$ and its mean as a function of D for 3400 digit numbers

We also take $s(n) = d(n)^c$, where $c \leq 1/2$. Should we select $c < 1/2$, the running time of extracting the modular square roots will drop from $O(\ln^3 n \ln\ln\ln n)$ to $o(\ln^3 n)$ and theoretically we do not lose too many curve orders. We wanted to see this behavior in practice.

In Figure 2.3 and 2.4 we can see the relation between $d(n) = \ln^D(n)$ and $e(n)$, where $D = 0.5, \ldots, 2$ and $s(n) = \ln^{1.1} n$ is fixed. On the first graph

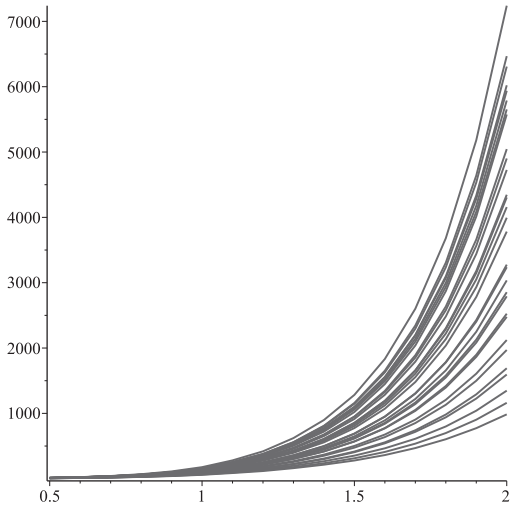

Figure 2.4: The mean of $e(n)$ as a function of D for $500-3400$ digit numbers

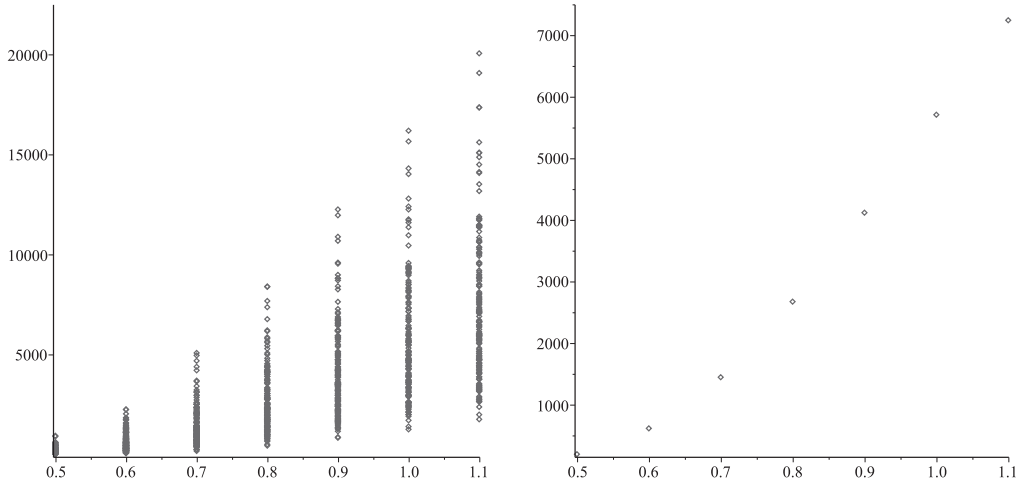

Figure 2.5: $e(n)$ and its mean as a function of S for 3400 digit numbers

of 2.3 $e(n)$ is presented as the function of D. The samples were taken from experiments ran on several numbers with 3400 digits (around 10000 bits), one point on the graph corresponds to the number of curve orders that we gain for a given n and D. On the second graph we can see the mean of the same function for the same numbers n. In Figure 2.4 we present the same mean values as in the second graph of 2.3, it is not only for 3400 digit numbers but from 500 up to 3400 digits. The topmost curve corresponds to the curve for 3400 digits numbers.

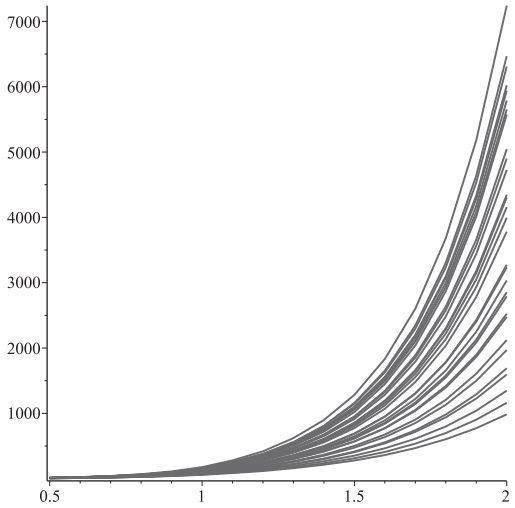

Figure 2.6: The mean of $e(n)$ as a function of S for $500-3400$ digit numbers

In Figure 2.5 and 2.6 we can see similar data presented as in 2.3 and 2.4, only in this case it is $e(n)$ as a function of S, where $S = 0.5, \ldots, 1.1$, $s(n) = \ln^S n$ and $d(n) = \ln^2 n$ fixed.

It is clear that decreasing the value of $d(n)$ has a drastic effect on the value of $e(n)$; if we decrease the value of D from 2 to 1.9, the value of $e(n)$ drops from around 7500 to just over 5000, thus it does not really worth to decrease the exponent of $\ln n$ in $d(n)$.

If we decrease the value of S from 1 to 0.9 though we still have almost 4500 curve orders from the original 6000. Thus taking $d(n) = \ln^2 n \ln\ln^{c_1} n$ and $s(n) = d(n)^{c_2}$, where $c_1 < 0$ and $c_2 < 1/2$, seems to be a good choice, because the time of extracting modular square roots will drop significantly but the number of curve orders will not.

The next statement that we would like to justify in practice is that the expected number of the curve orders $e(n)$ is

$$\bar{e}(n) = \sum_{D \in \urcorner} \frac{2^t}{h(D)}, \tag{2.4}$$

where $\urcorner$ is the set of $s(n)$-smooth discriminants up to $d(n)$, for some values of $s(n)$ and $d(n)$, that are appropriate for given n.

Figure 2.7 displays the relation between $\bar{e}(n)$ and $e(n)$ for numbers with 3400 digits; we would like to see in practice how reliable our estimation, $\bar{e}$ is, on the number of the curve orders. This graph presents the actual number of curve orders $e(n)$ as a function of $\bar{e}(n)$, our estimation. The experiments

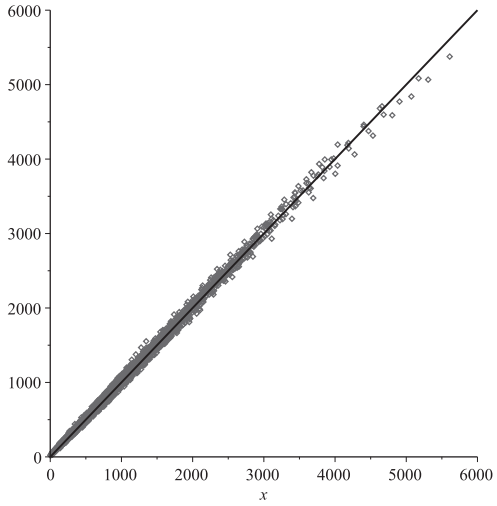

Figure 2.7: Precision of $\bar{e}$

were ran on numbers with 3400 digits with varying values of $s(n)$ and $d(n)$; in all experiments that we did in this topic we collected the $e(n)$'s and $\bar{e}(n)$'s. The graph also presents the identity function. We can see that all points of the function are close to it, indicating that $\bar{e}$ is a good estimator for e.

Estimating the number of the almost smooth curve orders $l(n)$ for input n plays a mayor role in our theory and implementation, as this means the expected number of the new descendants that we can produce with input n.

$$\lambda(n) = e^{\gamma} \frac{\ln b(n)}{\ln n} e(n), \tag{2.5}$$

There are two aspects of this estimation. In the definition of $\lambda(n)$ we use $e(n)$ as the number of curve orders that are produced, but in practice, using this actual value would be too time consuming, thus we use $\bar{e}(n)$, which is relatively easy to determine. Figure 2.8 and 2.9 both show the relation between $\lambda(n)$ and $l(n)$.

In Figure 2.8 we generated e random integers m with 3400 digits and factored them with bound $b(m)$, for several different values of $b(m)$, and in each situation we determined the value of $\lambda(m)$ and $l(m)$. We drop the suffix (n) from e and changes the suffix of the other parameters to (m). This experiment was carried out on a set of random numbers, thus the values of e are fixed and the value of the other parameters do not depend on any parent node n, rather on the random integer m itself. We can do that as the input number n and a curve order m that we gain are of the same size. This is a

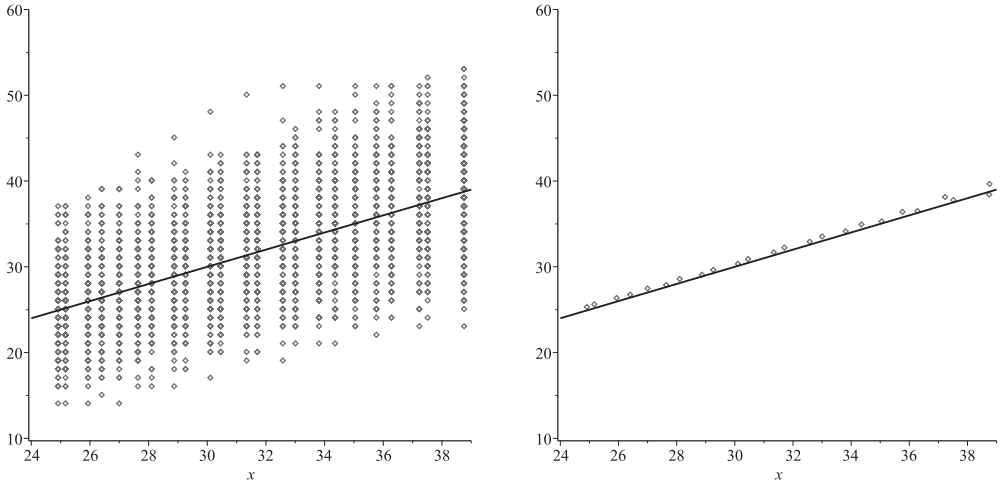

Figure 2.8: $l(m)$ as a function of $\lambda(m)$ for 3400 digit numbers using actual values e

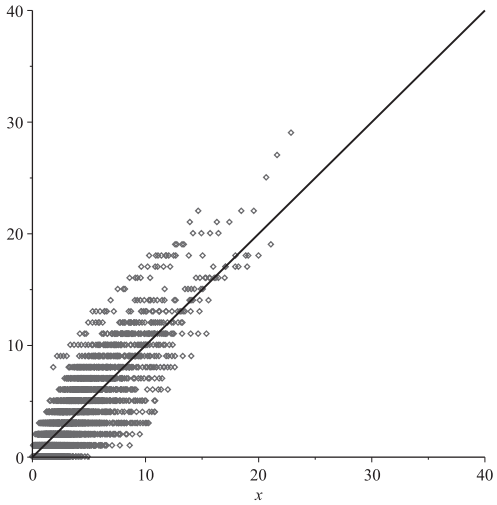

Figure 2.9: $l(n)$ as a function of $\lambda(n)$ for $500 - 3400$ digit numbers using estimated values $\bar{e}(n)$

fast way to carry out this experiment, as in this particular case we do not use any special properties of curve orders, they could be ordinary integers. The first graph of Figure 2.8 presents l as a function of λ, the second graph presents the mean of the same function.

Figure 2.9 shows the same function; l as a function of λ, as it is in practice. While running the algorithm, instead of using an actual value $e(n)$, we use

the estimation $\bar{e}(n)$ while estimating the value of $\lambda(n)$. In this case we use the suffix (n) again, because the numbers to factor, are actual curve orders produced by the algorithm; the parameters are depending on the value of n. The set of data is similar to Figure 2.7 in the sense that all the $l(n)$'s and $\lambda(n)$'s are collected while running different kind of experiments on numbers with 3400 digits. Therefore it does not make sense to determine the mean value of the function here, as the $\lambda(n)$ values are all different.

In both Figure 2.8 and 2.9 we included the identity function. We can see that the points of the mean of the function on Figure 2.8 are close to the identity function, but as we expect, Figure 2.9 has a bigger deviation from it, but seems to be still of a plausible degree.

There is another important question concerning $l(n)$; the number of the almost smooth curve orders. In order to increase $l(n)$ is it better to increase $b(n)$ or $e(n)$? In Figure 2.10 and 2.11 we see the relation between l, b and e. The lack of the suffixes indicates that we are using again random numbers m in this experiment, but now the parameters depend on neither a parent node n, nor m itself.

The first graph of Figure 2.10 shows l as a function of T, where $b = 2^T \cdot s \ln 2$, with $s = 10^6$ and $T = 0, \ldots, 12$. This means all the values that occurred as l while increasing T are included. The second graph gives the average of the same function. In Figure 2.11 we present l as a function of e. We have the same situation here as in Figure 2.10; the left hand graph gives all the values of l, and the right hand graph only shows the average value of the l's.

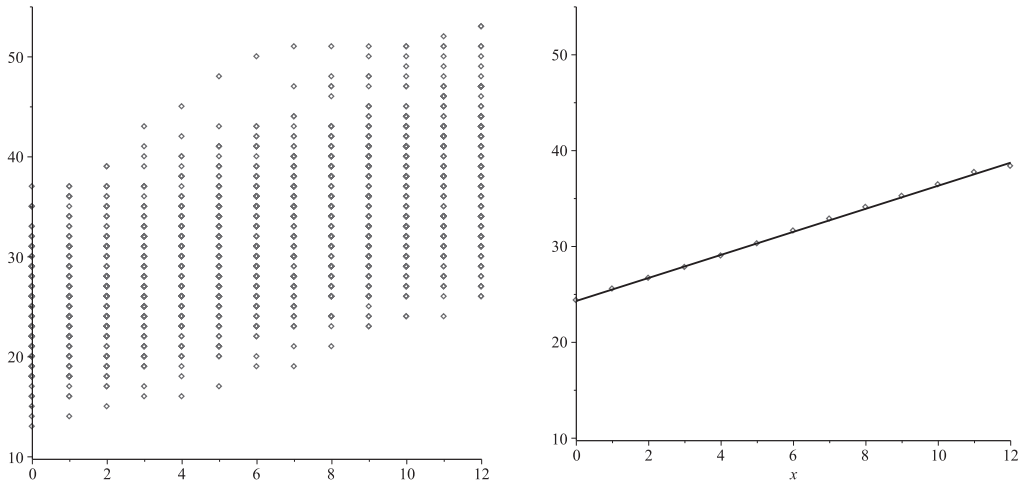

Figure 2.10: l as a function of T for 3400 digit numbers

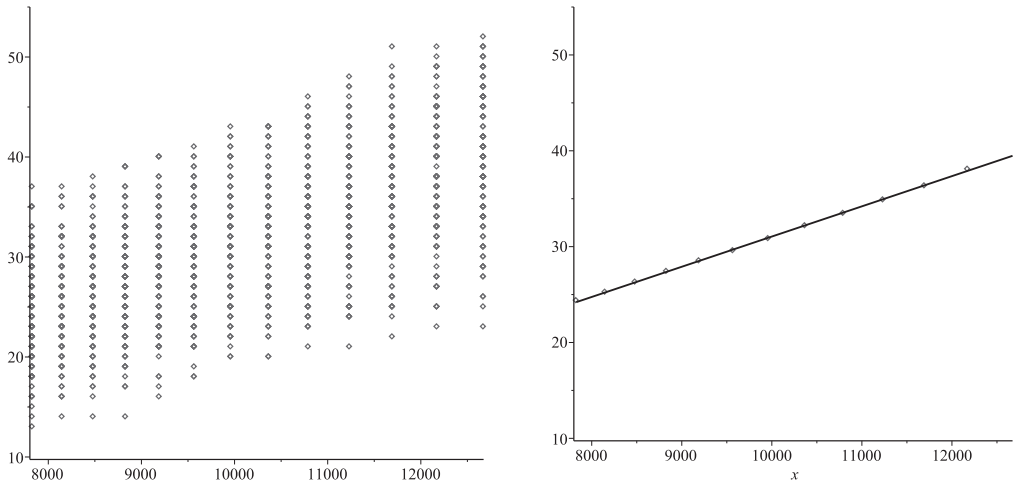

Figure 2.11: l as a function of e for 3400 digit numbers

The gradients of the curves in the right side graphs of 2.10 and 2.11 are similar, but we have to be aware that increasing e results in linear while increasing T results in logarithmic growth in the value of l. Consider the expected value

$$\lambda = e^{\gamma} \cdot e \cdot \frac{\ln b}{\ln m}$$

of l, where m is a random integer in this case. If we increase e 1.04-fold then the value of λ will be also 1.04-fold, while increasing the value of $b = 2^T \cdot s \cdot \ln 2$ to $2^{T+1} \cdot s \cdot \ln 2$ results in

$$\lambda(2 \cdot b) = e^{\gamma} \cdot \bar{e} \cdot \frac{\ln(2^{T+1} \cdot s \cdot \ln 2)}{\ln n} =$$

$$e^{\gamma} \cdot \bar{e} \cdot \left(\frac{\ln(2^T \cdot s \cdot \ln 2)}{\ln n} + \frac{\ln 2}{\ln n} \right) =$$

$$\lambda(b) \left(1 + \frac{\ln 2}{\ln(2^T \cdot s \cdot \ln 2)} \right),$$

thus on average λ will be also 1.04-fold if $T = 0, ..., 12$, but we had to increase b 2-fold. Of course there are many situations in which it can be still worthwhile, even necessary to increase b, for example if we have a huge amount of curve orders already, if we have a very efficient factoring algorithm or if we run out of discriminants.

In the rest of this section we deal with topics concerning the length of the path $I(n)$ from $n = n_i$ to n_l, where n_l small enough to be proved as

prime easily, and the gain $G(n)$; the size difference between n and n'. As we saw that in the analysis of the two strategies, if we have a fast factoring algorithm, increasing $b(n)$ will be very pleasant, because it shortens the path by increasing the gain, $G(n)$.

The two statements concerning this topic, that we would like to justify in practice are

$$\bar{G}(n) = \ln b(n) \tag{2.6}$$

and

$$\bar{I}(n) = \frac{\ln n}{G(n)}, \tag{2.7}$$

where $\bar{G}(n)$ is the expected gain and $\bar{I}(n)$ is the expected path length from n to n_l.

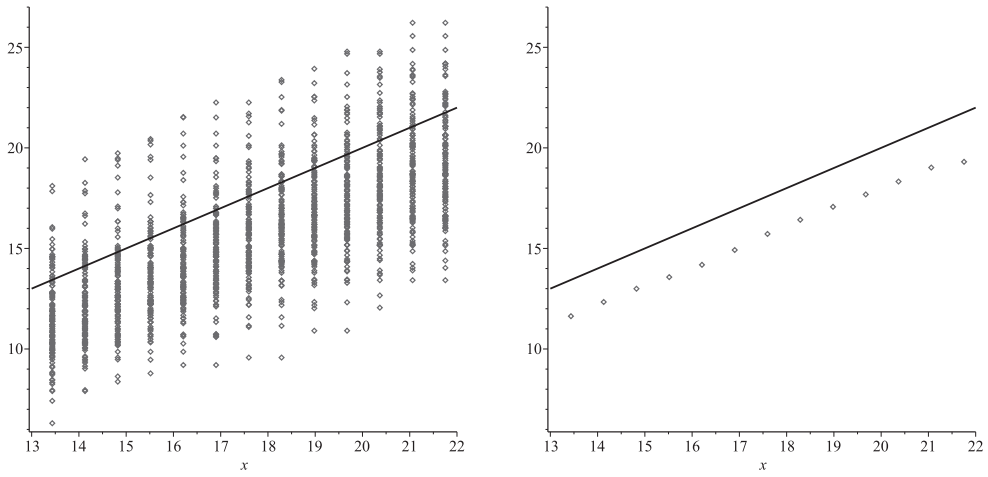

Figure 2.12: $G(n)$ as the function of $\bar{G}(n)$ for 3400 digit numbers

In Figure 2.12 we see $G(n)$ as a function of $\bar{G}(n)$ in the first graph and the mean of the function in the second graph. In both graphs the identity function is included. We observe a constant difference between $G(n)$ and $\bar{G}(n)$, due to the relatively significant difference between the sum and the integral for small primes and

$$G(n) \asymp \int_2^{b(n)} \frac{1}{x}\, dx = \ln b(n) - \ln 2.$$

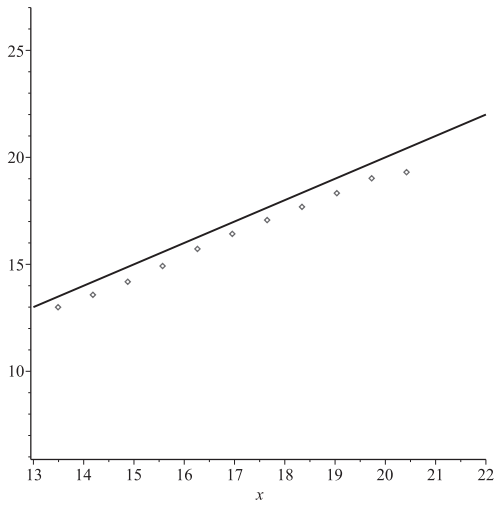

Figure 2.13: $G(n)$ as the function of $\bar{G}'(n)$ for 3400 digit numbers

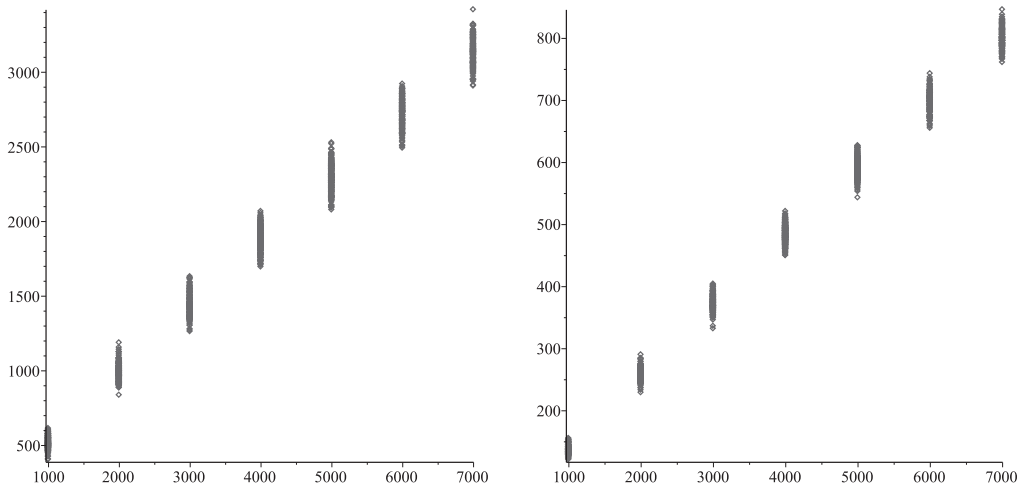

Figure 2.14: $I_b(n)$ and $I_n(n)$ as the function of $\log_{10} n$ up to 7000 digit numbers

If we consider the estimation

$$\bar{G}'(n) = \sum_{p \in \mathbb{P}, p \leq b(n)} \frac{\ln p}{p}$$

then the curve will be closer to the identity function (see Figure 2.13), in practice it is faster to use the equation 2.6 though.

The length $I(n)$ of the ECPP-path could refer to two different measures:

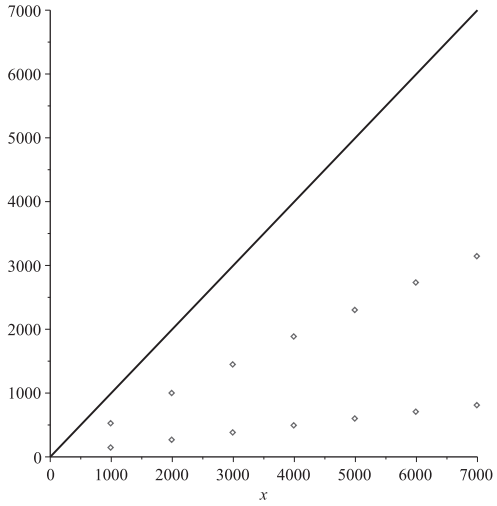

Figure 2.15: Average $I_b(n)$ and $I_n(n)$ as the function of $\log_{10} n$ up to 7000 digit numbers

on one hand we simply have a path from n to the small primes, along which we verify the primality of n; on the other hand, there is a sequence of probable primes that we used to get to the small primes, including backtracks and repetitions. We will denote the first one by $I_n(n)$ and the second by $I_b(n)$; one would expect a constant factor between them.

Figure 2.14 shows $I_b(n)$ on the left and $I_n(n)$ on the right, both as a function of $\log_{10} n$. We can see that the graph of $I_b(n)$ is much steeper. This we can see from Figure 2.15 too, that shows the average $I_b(n)$ and $I_n(n)$ as a function of $\log_{10} n$ together with the identity function. The graphs are consistent with a constant factor of around 3.95 between the two functions.

2.5 Conclusions

To summarize the results of this chapter we have given a detailed running time analysis of the ECPP algorithm and have proved that the heuristic running time can be reduced to $o(\ln^4 n)$ using refinements and fast algorithms to implement the different parts of the primality test. These are mostly joint results of Antal Járai and Gyöngyvér Kiss. The heuristics behind the refinements has the following main points, that are well-known estimations from probability theory, the work of Hendrik and Arjen Lenstra [16] and the original Atkin–Morain paper [3].

(1)
$$h(D) = O(\sqrt{d(n)}).$$

(2) The heuristic running time of extracting the modular square roots will drop from $O(\ln^3 n \ln\ln\ln n)$ to $o(\ln^3 n)$ if we choose a value of $s(n)$ that is below $\sqrt{d(n)}$ while retaining a reasonable number of discriminants by not decreasing the exponent of $\ln n$ in $d(n)$ below 2.

(3)
$$\bar{e}(n) = \sum_{D \in \urcorner} \frac{2^t}{h(D)},$$

where $\bar{e}(n)$ is the expected number of curve orders that we gain after processing all the $s(n)$ smooth discriminants D up to $d(n)$.

(4)
$$\lambda(n) = e^{\gamma} \frac{\ln b(n)}{\ln n} e(n),$$

where $\lambda(n)$ is the expected number of almost smooth curve orders.

(5)
$$\bar{G}(n) = \ln b(n),$$

where $\bar{G}(n)$ is the expected size of the difference between n and n'.

(6)
$$\bar{I}(n) = \frac{\ln n}{G(n)},$$

where $\bar{I}(n)$ is the expected length of the path from n to the small primes.

We have justified these estimations on a manageable set of experimental data; numbers with 500 digits up to 3400. The following is the own result of Gyöngyvér Kiss.

For (1) we presented a non cumulative and a cumulative average of $\sqrt{D}/h(D)$ up to 10^9 and in this range the function is close to a constant value of approximately 3.14.

To see (2) we looked at $e(n)$ as a function of $s(n)$ and $d(n)$ and the latter has a much bigger gradient, which means that if we want to decrease the running time by decreasing the number of discriminants, it is indeed much more efficient to decrease $s(n)$ below $\sqrt{d(n)}$.

For (3) we presented $e(n)$ as a function of $\bar{e}(n)$ and the graph is indeed close to the identity function.

Regarding (4), as we expected, the estimation of $l(n)$, the number of almost smooth curve orders, is much more precise if we use the actual $e(n)$ instead of $\bar{e}(n)$. However practice shows (see [13]) that it is not worthwhile to compute the curve orders for estimation purposes, as it takes too much time, and working with $\bar{e}(n)$ seems to be appropriate enough. We also examined the behavior of $l(n)$ if we increase $e(n)$ and $b(n)$. We have found that in general it is not worthwhile to increase $b(n)$ in big steps, but it seems more reasonable to collect a relatively big set of curve orders first and then increase $b(n)$ c-fold. In our case $c = 2$.

We could justify (5) too, apart from the constant difference between the functions $G(n)$ and $b(n)$ that was explained above, as the two lines are parallel; the same holds for (6).

The overall conclusion is that the experiments support our assumptions in practice for numbers in the range of $500, \ldots, 3400$ digits. Running experiments on numbers beyond this range would take too much time.

Chapter 3

Practical results

In the previous chapter we listed a couple of refinements based on the heuristics described there, that should make it possible to give an efficient implementation of the ECPP algorithm. This chapter deals with two implementations of the algorithm that uses the given heuristics and refinements. They are both implemented by Gyöngyvér Kiss. This chapter follows the content of two papers, refer to [10] and [13]. The implementations are written in Magma Computational Algebra System (see [4]).

In this chapter we cannot drop the index of n_i, except for the description of the algorithms for simplicity and consistency reasons, because our explanations would become unclear.

3.1 Magma Computational Algebra System

Magma [4] is a large software system, that has several highly optimized modules in different fields of computer algebra e.g. number theory, group theory, and geometry. It was launched in 1993 at the First Magma Conference on Computational Algebra in London. It consists of a large amount of built-in functions implemented in C language but also supports implementation of new functions in a command line interface environment. It is efficient, well documented and easy to use both under Linux and Windows.

Magma has a large scale of built-in primality tests, both probabilistic and deterministic. We are describing one of them here in detail, as this function is a combination of the Miller-Rabin test and ECPP. It can be invoked by the following function:

`IsPrime(`n`:` *parameter*`) :` `RngIntElt` $\mapsto$ `BoolElt`

This test is deterministic by default using ECPP after a quick Miller-

Rabin test, but it is possible to set a `Boolean` parameter `Proof` to false to use only the Miller-Rabin test with the default number of bases. We will refer to this function of Magma v2.17 (2011) as Magma-ECPP. This function is based on the implementation of F. Morain and works more or less according to the description above (1.9.2) as precise as it is possible to determine it from the verbose output, as the source code is not public. It operates on a set of predefined discriminants D, that are possibly fully factored and ordered by their class number $h(D)$. In the i^{th} iteration it first applies a trial division sieve on n_i then it loops through the set of discriminants performing one step of the algorithm Downrun(n) until it either finds $n_{i+1} = n_i'$ or runs out of discriminants. In case of success it goes to the $(i+1)^{th}$ iteration with the new input, otherwise it loops through the same set of discriminants, possibly increasing $b(n_i)$ or applies stronger factorization methods. If this effort is still unsuccessful, it has to backtrack to n_{i-1} starting from the next discriminant of n_{i-1} as after each iteration the last successful discriminant is stored together with the input n_i.

This implementation works well in most cases, but there are situations when it gets stuck, because of the small amount of discriminants that it uses. The numbers are not equally appropriate, some of them filter out more discriminants than others, thus changing only one parameter, in this case only $b(n_i)$, sometimes will fail. We can draw the conclusion from these situations; if we want to get closer to probability 1 to reach the small primes, we have to apply a strategy in the algorithm Downrun(n), when selecting the input n_i of the next attempt and also when choosing the value of $d(n_i)$, $s(n_i)$ and $b(n_i)$. To achieve this we implemented our version of ECPP, called Modified-ECPP.

3.2 Modified-ECPP

To describe the algorithm we have to slightly modify our view on the algorithms Downrun and Determine-Next-Input.

Algorithm 3.0.1. Determine-Next-Inputs($n, \urcorner, b(n)$)

(1) $N' = \emptyset$.

(2) If $\urcorner \neq \emptyset$ select and remove D from $\urcorner$, otherwise return N'.

(3) Determine $C'(x,y) = a'x^2 + b'xy + c'y^2$.

(4) Reduction($C(x,y)$).

(5) Reduction($C'(x,y)$).

(6) If the reduced form of $C(x,y)$ and $C'(x,y)$ is not the same, 1.7 has no solution, go back to 2.

(7) Follow the steps of the reduction of $C(x,y)$ backwards to get the required x and y.

(8) Determine $\nu = x + y\omega$, $m_+ = |\nu + 1|^2$ and $m_- = |\nu - 1|^2$.

(9) Try to factor m_+ and m_- up to $b(n)$.

(10) If neither m_+ nor m_- is almost smooth with $m_\pm = fn'$, and $n' > (\sqrt[4]{n} + 1)^2$, go back to 2.

(11) Let the successful $m_\pm$ be m. Store (n, D, m, n'). Add n' to N'. Go back to 2.

Algorithm 3.0.2. Downrun(n)

(1) $N = \{n\}$

(2) If $n < L$ test n for primality. If n is prime, return.

(3) Select a set of discriminants $\urcorner$ that is appropriate for n. Determine the value of $b(n)$.

(4) N' =Determine-Next-Inputs($n, \urcorner, b(n)$).

(5) $N = N \cup N'$. Select the *best* n from N and go back to 2.

The point of the modification of the two algorithms is that we do not stop at the first produced n' but we process a given set of discriminants and in an attempt on the number n we return a set N' of n'. The main question is the following; how do we choose the *best* n from N. Of course the *best* n can be a new n', can be a result of a previous iteration, we consider this *backtracking* and it can be the same n as the input of the current iteration, we call this situation *repetition*.

There are two versions of Modified-ECPP. They both work based on this interpretation of the algorithm Downrun(n), but there is a major difference between them; their definition of the *best*.

From this point of the chapter we cannot use one dimensional index anymore to indicate the current input and output, because the input and output of an iteration are both sets. The i^{th} iteration means a sequence of operation

from Downrun(n) that has one or more input $n_{i,j}$ and has one or more output $n'_{i,j}$. Note that the $n'_{i,j}$'s must be new descendants, thus if we backtrack or repeat an $n_{i,j}$, that is not considered the end of an iteration. One run of Determine-Next-Inputs (see 3.0.1), on an input $n_{i,j}$ is called an attempt. There are successful and failed attempts. In an iteration we can have several failed attempts and one successful attempt, which ends the iteration.

3.2.1 The first version of Modified-ECPP

As we already mentioned, a single iteration can have one or more inputs and outputs. A larger number of inputs increases the probability of reaching the small primes, but on the other hand, it slows down the computation. Our aim is to find a balanced situation, where the implementation is reasonably fast but we still have a good chance to terminate successfully.

The definition of *best* in this case is determined based on the function $\lambda(n_{i,j})$. The idea is that the computations for smaller discriminants will be cheaper, and hence the algorithm will be completed faster. Thus we are searching for the minimal power of $d(n_{i,j}) = \ln^{D(n_{i,j})} n_{i,j}$ for which the value of $\lambda(n_{i,j})$ is still above a certain bound, with $s(n_{i,j})$ and $b(n_{i,j})$ fixed. These $D(n_{i,j})$'s are stored in an array and a certain penalty p_i is added to the $D(n_{i,j})$'s, that is computed from the level i. If a value $\delta_{i,j} = D(n_{i,j}) + p_i$ reaches a given bound, $n_{i,j}$ is excluded from the array of the possible choices. Then the array is ordered and the smallest one is selected. Thus in this case the *best* $n_{i,j}$ is the one with the minimal $\delta_{i,j}$.

Algorithm 3.0.3. Process-First-N($n, b(n)$)

(1) $D(n) = D_0$, $s(n) = \ln^S n$.

(2) $d(n) = \ln^{D(n)} n$.

(3) Let $\urcorner$ be the set of $s(n)$-smooth discriminants up to $d(n)$ that is appropriate for n.

(4) $N' =$Determine-Next-Inputs($n, \urcorner, b(n)$).

(5) $D(n) = D(n) + \Delta$. If $N' = \emptyset$, then go back to 2.

(6) Add $(n, D(n))$ to N, return N'

The function Process-First-N(n) is kind of a brute force strategy to find the descendants of n_0, the input of the algorithm. We have to force n_0 because at that point of the algorithm it is our only choice. It runs only at the beginning of the algorithm. As D_0 and S are independent from the choice of $n_{i,j}$, it is stored as a global variable. In our case $D_0 = 1$ and $S = 1$;

Algorithm 3.0.4. Determine-Discriminant-Limit$(n, b(n))$

(1) $D(n) = D_0$, $s(n) = \ln^S n$.

(2) $d(n) = \ln^{D(n)} n$.

(3) Let $\urcorner$ be the set of $s(n)$-smooth discriminants up to $d(n)$ that is appropriate for n.

(4) Determine $\lambda(n) = e^\gamma e(n) \ln b(n) / \ln n$ for $\urcorner$.

(5) If $\lambda(n) < \lambda_0$, then $D(n) = D(n) + \Delta$. Go back to 2.

(6) Let $\delta(n) = D(n)$, add $(n, \delta(n))$ to N.

This function increases the discriminant set by incrementing $D(n_{i,j})$ with Δ, until the value of $\lambda(n_{i,j})$ reaches the bound λ_0. In this case λ_0, Δ, S are global variables, and independent from $n_{i,j}$. We have chosen $\lambda_0 = 1.3$, $\Delta = 0.1$, $S = 1$.

Algorithm 3.0.5. Modified-Downrun(n)

(1) $N = \emptyset$, $b(n) = \ln^B n$, $N' =$Process-First-N$(n, b(n))$.

(2) Select and remove an n from N', $b(n) = \ln^B n$.

(3) If $n < L$ test n for primality. If n is prime, return.

(4) Determine-Discriminant-Limit$(n, b(n))$

(5) If $N' \neq \emptyset$ go back to 2.

(6) For each $n \in N$ if n is not on the latest level, $\delta(n) = \delta(n) + p$

(7) Sort N by δ and let n be the one with the minimal δ.

(8) Determine $\urcorner$ for n and $d(n)$. Determine $b(n)$.

(9) $N' =$Determine-Next-Inputs$(n, \urcorner, b(n))$.

(10) $\delta(n) = \delta(n) + \Delta$. If $N' = \emptyset$, then go back to 7.

(11) Go back to 2.

Note that the value p added to $\delta(n_{i,j})$'s in each iteration except for the one in which the associated $n_{i,j}$'s are produced. The value of p and B is also global; $p = 0.8$ and $B = 2$ are our choices. The size of the set N; the possibilities to backtrack to, is also a global variable, we use a set of 100.

Modified-ECPP uses simple trial division up to 1000 and batch trial division up to $b(n_{i,j})$ computing the prime products on the fly. As we do not change the value of $b(n_{i,j}) = \ln^2 n_{i,j}$ this approach seems sufficient.

We ran several experiments on numbers with more than 1000 digits to see the advantages and disadvantages of this implementation in practice. We managed to prove big prime numbers that are determined from the Pascal's triangle, see [7]. We have found an example when Magma-ECPP got stuck and Modified-ECPP terminated successfully, see [10].

There are certain advantages of Modified-ECPP compared to Magma-ECPP, although in most of the situations the latter will terminate successfully too.

First of all the bigger set of discriminants; Modified-ECPP uses a list of predefined and factored discriminants together with their class number up to 10^9. Obviously if we have more discriminants, the possibility of failure is smaller.

There is a bigger range of eligible choices and freedom in the decision in case of a failed attempt. Modified-ECPP keeps track of a hundred $n_{i,j}$'s and in case of failure backtracking from n_i to n_{i-1} is not the only option in the i^{th} iteration; there are possibly more $n_{i,j}$'s on the same level, but stepping back more levels is also possible, thus we can avoid getting stuck in an unfortunate branch.

The goal of producing more nodes in an iteration is not only making the implementation more robust against getting stuck but to avoid running multiple attempts on the same $n_{i,j}$ by switching between the sibling nodes on the same level. In addition, if it turns out that an $n_{i,j}$ is not as successful as estimated, and thus produces no descendants after the first iteration on the expected interval, we become more careful and try to take small steps at a time and reestimate. We try to keep backtracking fast and flexible.

We try to minimize the number of backtracks to the previous levels though, because that might make our whole effort on the current level useless, and also because the size of the numbers is growing going up in the tree. The goal of the estimation of the initial value of $\delta(n_{i,j})$ is to reduce the probability of backtracking and repetition by predicting the minimal interval where an attempt will run successfully.

As the predictions give us no hundred percent certainty to avoid such situations, we introduced the penalty p. The default value of the penalty is high, 0.8. After one unsuccessful attempt we increase the value $\delta(n_{i,j})$ with

0.1, thus roughly speaking 0.8 would mean 8 unsuccessful attempts on an $n_{i,j}$. This is a tough condition, does not occur frequently. Of course different $n_{i,j}$'s starts from different $\delta(n_{i,j})$, thus in practice we do not always need 8 unsuccessful iterations, if none of the $n_{i,j}$'s on the current level is suitable enough. The numbers are just to get a feeling about the size of the penalty.

The strategy gives also the basis to predict, on which $n_{i,j}$ we have to process the smallest amount of discriminants to provide at least one descendant to avoid unnecessary efforts.

Modified-ECPP was tested with several numbers with around 1000 digits on a machine with 8001 Mb RAM and eight 2.5 GHz Intel Xeon processors, and it provided proof in each situation in $2000 - 3000$ seconds. Magma-ECPP was running on the same numbers, failed on one number (see [10]), and provided proof in $2000 - 8000$ seconds in every other cases. Depending on whether we backtrack or not there are bigger differences in running time. It seems that the running times of Modified-ECPP are more balanced on numbers with similar size.

There are still defects of this implementation though; it takes advantage of only one parameter $d(n_{i,j})$. If we consider a strategy that controls only the number of the discriminants, we would not get far by changing only $d(n_{i,j})$, because after a while the size of the prime factors, that is fixed in this case, becomes too small compared to the discriminants themselves, thus we will not find any more $s(n_{i,j})$-smooth discriminants. Therefore it is not enough to change only $d(n_{i,j})$, at certain points we have to increase $s(n_{i,j})$ too. It would be also good to know which parameter is more efficient to increase at a certain point.

On the other hand controlling only the size of the discriminants seems also insufficient as we could have so many processed discriminants without finding new descendants, that it would be worthwhile to increase the factoring bound $b(n_{i,j})$, or we can even run out of discriminants. If we change $b(n_{i,j})$, the gain $G(n_{i,j})$ will also change, thus we cannot consider the expected descendants equal anymore. We have to apply a strategy that determines the *best* number depending on all the parameters; $d(n_{i,j})$, $s(n_{i,j})$ and $b(n_{i,j})$ and takes the expected gain and work into account too.

3.2.2 The second version of Modified-ECPP

This section is written according to the paper of Gyöngyvér Kiss [13] that describes her implementation of this version of Modified-ECPP in detail.

The definition of *best* has two aspects in this case;

1- First we have to consider that the numbers are not equally appropri-

ate. Applying the Jacobi symbol filters on n_{i_1,j_1} could filter out more discriminants than on n_{i_2,j_2}, so to produce the same number of descendants we would have to process a larger number of discriminants or use bigger factoring bound for n_{i_1,j_1}. Higher bounds imply more execution time, as we need to deal with bigger discriminants, primes. So the first aspect is the time it takes to produce a given number of descendants on input $n_{i,j}$.

2- The second aspect is the size of the descendants; the smaller they are, the faster we get to the small primes.

To estimate 1, we have to acquire running time information on the $n_{i,j}$'s. This information can be collected from running algorithm Determine-Next-Inputs (see 3.0.1), on each newly produced $n_{i,j}$ with certain (small) values of parameters $s(n_{i,j})$, $d(n_{i,j})$ and $b(n_{i,j})$:

$$b_0(n_{i,j}) = \ln^2 n_{i,j}$$

$$s_0(n_{i,j}) = \lambda_0 \cdot \frac{\ln n_{i,j}}{e^\gamma \cdot \ln b_0(n_{i,j})}$$

$$d_0(n_{i,j}) = s_0(n_{i,j})^2$$

The choice for $s_0(n_{i,j})$ comes from $\lambda(n_{i,j}) = e^\gamma e(n_{i,j}) \ln b(n_{i,j}) / \ln n_{i,j}$. If we suppose that $e(n_{i,j}) \asymp \sqrt{d(n_{i,j})}$ and $\lambda(n_{i,j}) = \lambda_0$ is a parameter, we take $s_0(n_{i,j}) = e(n_{i,j})$ and $d_0(n_{i,j}) = s(n_{i,j})^2$ despite of [5], where we suggest keeping the value of $s(n_{i,j})$ below $\sqrt{d(n_{i,j})}$, but in practice, on small numbers, taking $s(n_{i,j}) < \sqrt{d(n_{i,j})}$ led to some difficulties.

In these initial runs we store the time needed for extracting the modular square roots, for the reduction algorithm and for factoring, and we also see how many actual descendants are produced. With this information, we can estimate the time needed if we increase $s(n_{i,j})$, $d(n_{i,j})$ k-fold or $b(n_{i,j})$ 2^c-fold separately; the running time of the three subroutines depends (via the three parameters) only on $n_{i,j}$, so it is possible to express these running times as functions of $n_{i,j}$. Estimating the running time of square root extraction modulo a prime and of form reduction is fairly easy. We can measure the running time of a single operation and multiply by the number of times we need to perform them (which is the number of appropriate primes and the number of successful discriminants, respectively). In the case of factoring the running time of batch trial division is linear neither to the number of curve orders nor to the number of primes used, but as we do not use huge amount of curve orders or primes simultaneously, linear approximation works well in

practice. For $b(n_{i,j}) = 2^t \cdot 10^6 \cdot \ln 2$ we double the expected time if we increase t to $t+1$ as we have to deal with products of twice the size.

If we have the initial running time we will determine the value of $\lambda_s(n_{i,j})$ for $k \cdot s(n_{i,j})$, $d(n_{i,j})$ and $\lambda_d(n_{i,j})$ for $s(n_{i,j})$, $k \cdot d(n_{i,j})$ where k is a parameter, by collecting the discriminants in both cases and compute $\bar{e}_s(n_{i,j})$, $\bar{e}_d(n_{i,j})$. Using the original $b(n_{i,j})$ we can compute $\lambda_s(n_{i,j})$ and $\lambda_d(n_{i,j})$.

Parameter $b(n_{i,j})$ must be treated differently. We do not have to evaluate the discriminants to see the change of the value of $\lambda(n_{i,j})$, but vice versa, it is possible to compute directly how far we have to increase $b(n_{i,j})$ to induce a certain rise in the value of $\lambda(n_{i,j})$. First we determine $\lambda_b(n_{i,j}) = \lambda(n_{i,j}) + \alpha$, where α is the minimal excepted increase of the value of $\lambda(n_{i,j})$, the previous estimation of λ for $n_{i,j}$ that we get from the last attempt on $n_{i,j}$. We certainly have a last attempt from the initial runs. From $\lambda_b(n_{i,j})$ with the help of the actual $e(n_{i,j})$ of the last attempt, we get the minimal required t from $b(n_{i,j}) = 2^t \cdot 10^6 \cdot \ln 2$.

Now we have $\lambda_s(n_{i,j})$, $\lambda_d(n_{i,j})$ and $\lambda_b(n_{i,j})$ and also the estimated times $t_s(n_{i,j})$, $t_d(n_{i,j})$ and $t_b(n_{i,j})$ that it necessary to increase $\lambda(n_{i,j})$. Then we can store the different $t_{s|d|b}(n_{i,j})/\lambda_{s|d|b}(n_{i,j})$ values.

We have determined the expected work for each parameter that we have to invest to obtain a given increase of the value $\lambda(n_{i,j})$, but we do not know the expected size of the descendants that we would get. This can be determined with the help of the gain function $G(n_{i,j})$, which depends only on $b(n_{i,j})$, the factorization effort on the curve orders. From this we can see that we gain descendants with the smallest expected size if we increase $b(n_{i,j})$. Furthermore if we increase $s(n_{i,j})$ or $d(n_{i,j})$, the expected size of the descendants that we gain is the same. After incrementing $s(n_{i,j})$ or $d(n_{i,j})$, we want to know how much effort it takes to reduce the descendants further; what is the average work per bit, w, that we needed to decrease them? This we can estimate from the previous iterations. After multiplication by the estimated size differences, we obtain a value, $a_{s|d|b}(n_{i,j})$ for the expected effort of reducing the descendants to the size of the smallest one, of course $a_s(n_{i,j}) = a_d(n_{i,j})$ and $a_b(n_{i,j}) = 0$. Then compare the values $a_{s|d|b}(n_{i,j}) + t_{s|d|b}(n_{i,j})/\lambda_{s|d|b}(n_{i,j})$ and select that parameter for which this value is minimal. We denote this minimal value by $mt(n_{i,j})$, for the given $n_{i,j}$.

Now we now which parameter to increase for each $n_{i,j}$, but it is still unclear which is the *best* $n_{i,j}$. The selected parameter determines $mt(n_{i,j})$ for each $n_{i,j}$, but we cannot yet compare them as the expected size of the predicted descendants are different. We would like to know the size of the smallest because we would still have to work on the other descendants to reduce their size so far. Thus we have to add the average work per bit, w

to $mt(n_{i,j})$ after multiplying with the estimated size differences, then we get $mt^*(n_{i,j})$. Now the *best* number is the $n_{i,j}$ for which this value $mt^*(n_{i,j})$ is the smallest.

If no new descendants are produced, we need to be able to backtrack. We keep a window with a certain number of $n_{i,j}$'s for which we store all the data that is necessary to continue using this value of $n_{i,j}$ if turns out to be the *best*. Newly found $n_{i,j}$'s are always going to the window. If the number of the $n_{i,j}$'s in the window exceeds a limit, we throw away the *worst* ones. It is not possible to backtrack to a number that is not in the window anymore. We compute w for values $n_{i,j}$ in this window and update it after each iteration.

Algorithm 3.0.6. Estimate$(n, d(n), s(n), b(n))$

(1) Determine $\lambda_d(n)$ by computing the value of $e_d(n)$: collect the $s(n)$-smooth discriminants up to $k \cdot d(n)$, that are appropriate for n.

(2) Determine $\lambda_s(n)$ by computing the values of $e_s(n)$: collect the $k \cdot s(n)$-smooth discriminants up to $d(n)$, that are appropriate for n.

(3) Determine the new value of $b(n_{j,i})$ from $\lambda_b(n) = \lambda(n) + \alpha$.

(4) From the actual running times stored for n, determine $t_s(n)$, $t_d(n)$ and $t_b(n)$ for $k \cdot s(n)$, $k \cdot d(n)$, $2^c \cdot b(n)$.

(5) Compute the expected gain $G(n)$, and compute the value $a_s(n) = a_d(n)$. Let $a_b(n) = 0$.

(6) Determine the value of $mt(n)$ together with the corresponding lists of discriminants and primes, add them to the window together with n.

Algorithm 3.0.7. Modified-Downrun(n)

(1) $N = \{n\}$

(2) If $N \neq \emptyset$ select and remove n from N, otherwise go to 5.

(3) N' =Determine-Next-Inputs$(n, \urcorner, b_0(n))$, where $\urcorner$ is the set of $s_0(n)$-smooth discriminants up to $d_0(n)$.

(4) Estimate$(n, d_0(n), s_0(n), b_0(n))$. $N = N \cup N'$ and go back to 2.

(5) Reorder the window by the values mt^* (all of them are up to date).

(6) Pick the best as n. N' =Determine-Next-Inputs$(n, \urcorner, b(n))$; $\urcorner$ and $b(n)$ come from the window.

(7) Estimate$(n, d(n), s(n), b(n))$, update the window.

(8) If $N' \neq \emptyset$ then $N = N'$ and go back to 2. Otherwise go back to 5.

There is a list of additional parameters that we use in the algorithm.

Parameter λ_0 – This parameter provides the initial value of λ in the initial run. We have to keep in mind that this part of the algorithm runs only because we would like to collect some data on the new descendants. Thus we keep the value of λ_0 relatively small to avoid spending too much time here. We also have to be aware of that if it runs with a big value of λ_0 the tree would expand too much as this part of the algorithm runs on each $n_{i,j}$. On the other hand if it is too small, we will not be able to collect realistic data about the running time. In practice we keep this value between 1/3 and 1/2.

Parameter k – In algorithm Estimate$(n, d(n), s(n), b(n))$ we take $k \cdot s(n_{i,j})$, $k \cdot d(n_{i,j})$ as the new values of these parameters that we want to examine. We have to be careful with our choice here as if it is too big, we take too big steps and the algorithm becomes slow and we also loose the flexibility of it (reestimating after small steps to see if the selected $n_{i,j}$ is as good as it was predicted). On the other hand it cannot be too small because then we spend too much time estimating the $n_{i,j}$'s compared to the time to process them. In practice we usually use $k = 2$, but there are experiments described later, that use $k = 1.5, \ldots, 4$.

Parameter α – As it is mentioned above α is the minimal increase of value $\lambda(n_{i,j})$. Increasing the parameters k-fold does not give us any certainty of the degree of increase of $\bar{e}(n_{i,j})$, as the exact connection between $s(n_{i,j})$, $d(n_{i,j})$ and $e(n_{i,j})$, even $\bar{e}(n_{i,j})$ is unknown. Thus it is possible that a k-fold rise in the value of the parameters will not increase the value of $\lambda(n_{i,j})$ enough. If $\lambda_d(n_{i,j}) - \lambda(n_{i,j}) < \alpha$ or $\lambda_s(n_{i,j}) - \lambda(n_{i,j}) < \alpha$ then we have to increase the value of k for that particular parameter as one step would be too small. The new value of $b(n_{i,j})$ we determine directly from α. We use $\alpha = 0.25$;

Parameter $wSize$ – This parameter denotes the size of the window of the $n_{i,j}$'s that we keep track of. It is possible to backtrack only in the window. This window cannot be too small, to make sure that it contains all the $n_{i,j}$'s that the strategy would choose as *best*, but if we store too much of them, the program becomes slow. This parameter is of the form $\ln n/(c \ln \ln n)$. Experiments confirm that it would be more suitable to express the size of the window in terms of level instead of number of $n_{i,j}$'s.

3.3 Experiments

In the rest of this study we will refer to the second version of Modified-ECPP as Modified-ECPP.

Implementing Modified-ECPP allowed us to carry out experiments on numbers up to 7000 digits to study the behavior of the strategy and the running time of the algorithm. In this section we will describe these experiments. This section is entirely the work of Gyöngyvér Kiss and is written following her paper [13].

We ran various experiments on the running time and on the strategy with around 200 numbers each for $k \cdot 1000$ decimal digits, with $k = 1, 2, \ldots, 7$. We have produced many graphs, but presenting all of them here is impossible thus we will display the graph for the largest size for which data are available for each type of experiment. The others, that are not displayed here can be downloaded from the page `http://www.math.ru.nl/~gykiss`. The experiments were run on computers with Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz processors and 8 GB memory.

3.3.1 Running times

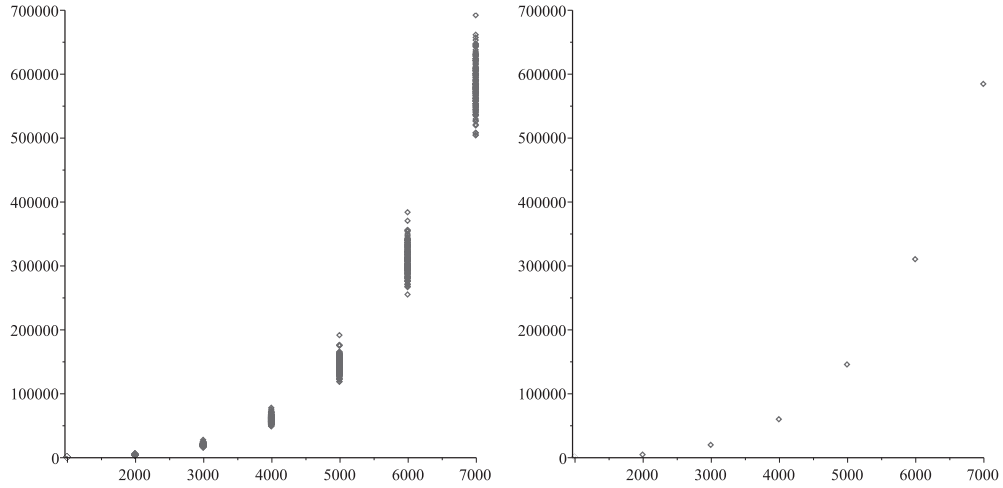

Figure 3.1: Running times

Figure 3.1 shows the running times of the algorithm Modified-ECPP. On the first graph of Figure 3.1 the total running time is presented as a function of the decimal digits of the input numbers of $k \cdot 1000$ digits for $k = 1, 2, 3, \ldots, 7$. A single dot indicates the running time for a given number.

On the second graph the average running times of the algorithm on the same numbers is indicated. A single dot here shows the average running time of the numbers with the same size.

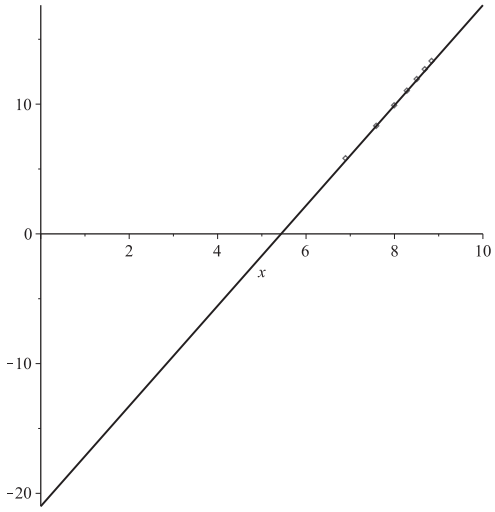

Figure 3.2: Average running times on logarithmic scale

We have applied least squares linear approximation method, to find out the running time in the examined range and we found that the best fit for the running time on a logarithmic scale is given by the line $y = 3.86 \cdot x - 21.00$. See Figure 3.2.

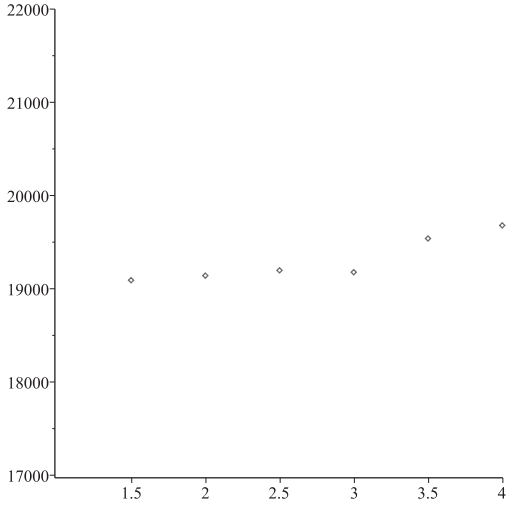

Figure 3.3: Running time as function of k for 3000 digits numbers

We indicated in the previous section that at each attempt we increase the size of the parameters $s(n_{i,j})$, $d(n_{i,j})$ or $b(n_{i,j})$ and the increase in the first two cases is k-fold. We state that in practice we use $k = 2$ and that $k = 1.5, \ldots, 3$ seems to be a useful range. We tested this behavior on numbers with 3000 digits with $k = 1.5, 2, 2.5, 3, 3.5, 4$ and the result can be seen in Figure 3.3. The experiments were also done on numbers with 1000 and 2000 digits (see `http://www.math.ru.nl/~gykiss/graphs/S.html`). We can see that for smaller numbers, bigger steps seem to work well in practice, but as the numbers are growing, the range $k = 1.5, \ldots, 3$ becomes more preferable. It is also visible that the difference is rather insignificant.

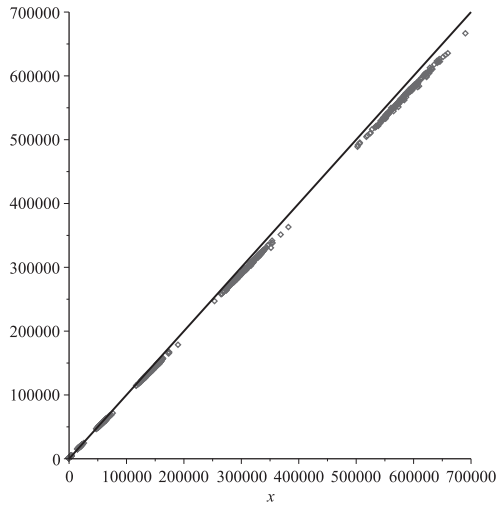

Figure 3.4: The proportion of the execution time compared to the total running time

We also indicated that the size of the steps k cannot be too small, as in this case the time of estimations and such administrative jobs would become too big compared to the time of executional jobs such as extracting modular square roots modulo $n_{i,j}$, reduction of binary forms, factoring and Miller-Rabin tests on the new descendants. We have to emphasize that in one iteration there will typically be several execution and administration steps, and these experiments are not meant to measure the time used in an iteration. Figure 3.4 shows the time of the executional tasks as a function of the total running time for numbers from 1000 digits up to 7000 with $k = 2$ and the identity function for comparison. The clusters for the data for the numbers of the same size are clearly visible. As expected, the time spent on administration rather than execution is negligible.

3.3.2 Experiments on the strategy

In this section we would like to study the behavior of the strategy of Modified-ECPP mainly via the analysis of the number and size of backtracks and repetitions compared to the length of the path. Repetitions and backtracks occur if the initial runs on the new $n_{i,j}$'s as well as one attempt on the best available input do not provide new descendants; we would not consider the first attempt after the initial runs as backtrack or repetition, as the initial runs are only precomputation and in general we do not expect new descendants running them (of course in practice in many cases they produce descendants), thus in both cases we have to select an $n_{i,j}$ from existing ones. The only difference between the two is when we backtrack, we select a different number, whereas in a repetition the same $n_{i,j}$ is selected, because it is still the *best*. We expect that repetitions occur frequently as we increase $\lambda(n_{i,j})$ with around $\alpha = 0.25$ after each attempt on $n_{i,j}$ instead of 1. This means that one may expect to repeat the procedure as much as 4 times before producing new descendants.

Note that when we are talking about backtracking, we do not only mean stepping back to a previous level, as we may step either backwards or forward.

Note that the length of the path $I_b(n_0)$ from n_0, the input of the algorithm, is not equal to the number of iterations, as we include each number on which an attempt was ever made, and one iteration likely consists of several attempts. However, the number of the iterations is equal to the maximum *level*, the i from the index of $n_{i,j}$, as we consider numbers produced in the same iteration to be on the same level.

Figure 3.5 shows the proportion of the number of backtracks and repetitions to the length of the path, for numbers of various sizes. In the first graph of Figure 3.5 we can see the number of backtracks as the function of the length of the path from n_0, $I_b(n_0)$, in the second graph the number of repetitions are indicated as the function of $I_b(n_0)$. The length of the path is of order $O(\ln(n))$, and the graphs clearly show the 7 clusters corresponding to numbers of size $k \cdot 1000$, for $k = 1, \ldots, 7$. Furthermore it is also clear that we need backtracks during the Downrun, as around one of each 30 steps is a backtrack on the path. As we expected, the repetitions are far more frequent. Although around half of the length of the path are repetitions, it is less than our expectations; 4 repetitions before producing descendants. This means that it is worthwhile to take small steps only, because besides the flexibility that we gain with small steps, in general it seems that we do not have to have $\lambda(n_{i,j}) = 1$ in practice to produce descendants.

Besides knowing how often we backtrack in a run, we also want to know how far we backtrack in terms of level and size. If these differences are not

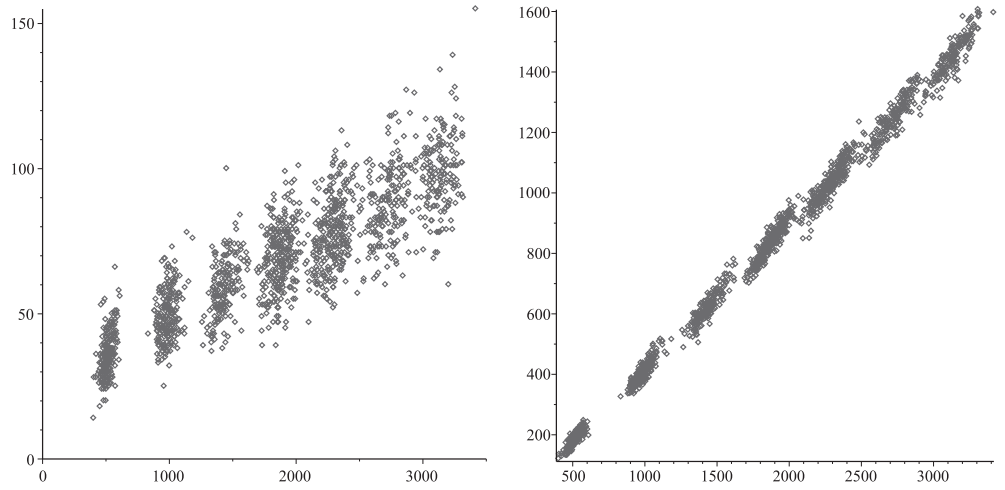

Figure 3.5: The number of backtracks and repetitions as a function of the length of the paths

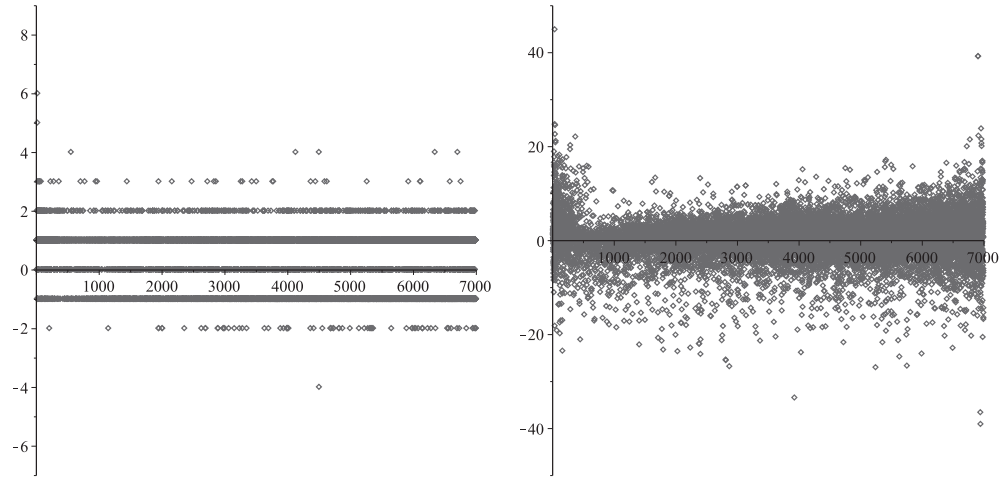

Figure 3.6: The level and size differences of backtracks

too big, it would mean that the strategy is balanced as it aims to avoid these situations as taking a big step backwards could mean that our efforts on a big branch will be lost. We present these size and level differences in Figure 3.6. The data was collected during several runs of the algorithm on numbers with 7000 digits, which means that along the path, several attempts were made on numbers with size between $0 - 7000$ digits. The first graph of Figure 3.6 shows the level differences of backtracks as the function of the size of the numbers (in decimal digits) from which we backtracked. The second graph

shows the size differences of the backtracks as the function of the size of the start-up points. A positive number means that we stepped back, a negative number means that we stepped forward, 0 indicates that we stayed on the same level but we have selected a different number. These graphs and the rest of them (see `http://www.math.ru.nl/~gykiss/graphs/B.html`), where the input numbers are smaller, suggest the same conclusion: neither the level nor the size differences depend on the size of the input, except for really small numbers, when the work that we loose with backtracking is negligible, and there are only a very few outliers. This result supports our idea to express the size of the window of stored $n_{i,j}$ in terms of level instead of cardinality. According to the first graph we could just say, store the numbers of the last 7 levels.

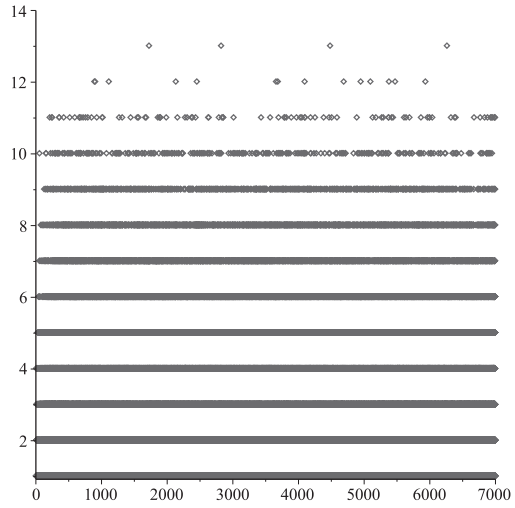

Figure 3.7: The length of the repetition sequences

Information on the length of the repetition sequences is also important, that gives an indication on the reliability of the estimations. These results can be seen in Figure 3.7, Figure 3.8. In Figure 3.7 we can see the length of the repetition sequences as the function of the size of the number repeated in decimal digits. We can see that the length of the repetition sequences does not go above 13. It is visible from this graph that even 12 and 13 long sequences occur rarely, but the proportion of the smaller lengths are unclear as the dots are to dense. We also want to see for which size of numbers the repetitions are more frequent. In the first chart of Figure 3.8 we can see the number of the repetitions for the number of digits indicated on the pie chart; the number of the repetitions seems to be independent from the size of the numbers. The second chart shows the proportion of the repetition sequences

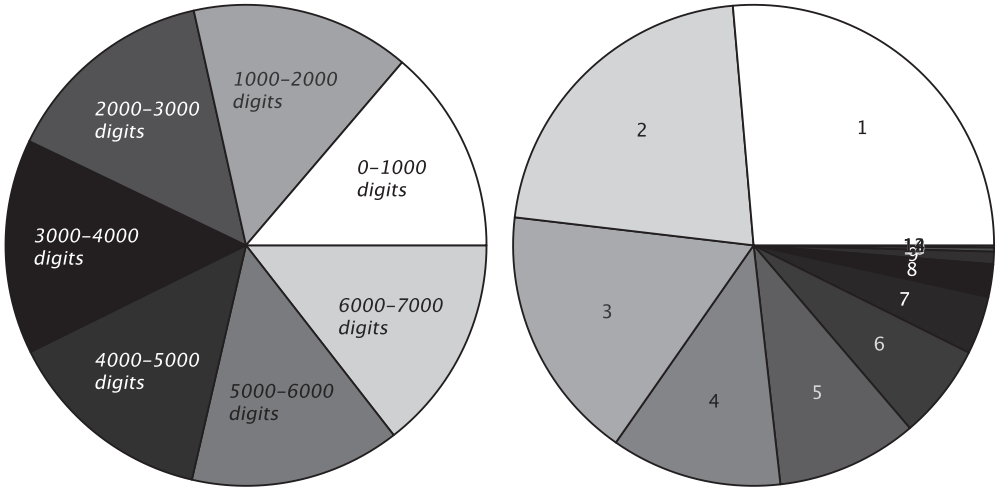

Figure 3.8: The proportion of the repetition sequences

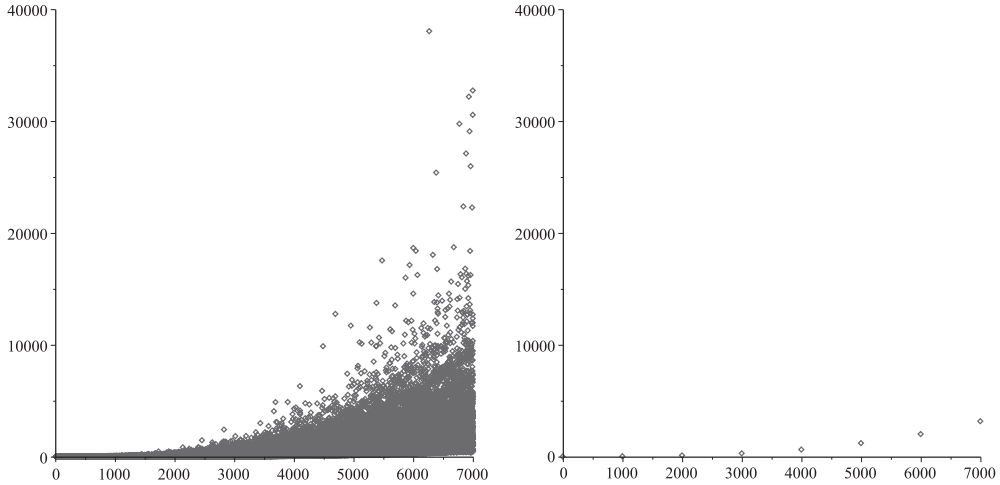

Figure 3.9: The time of the repetition sequences

with given lengths that are indicated on the fields of the chart for numbers with $6000-7000$ digits. The pie chart of smaller numbers look really similar. We can see that the number of repetition sequences with length 1, 2 and 3 is around 2/3 and with length 1, 2, 3 and 4 is around 3/4 compared to the total number of repetitions. Longer sequences are relatively rare.

In Figure 3.9 we can see the time of the repetition sequences as the function of the size of the numbers that are repeated in decimal digits in the first graph and the average of the same function in the second graph. The time of course does depend on the size of the numbers, as for bigger numbers

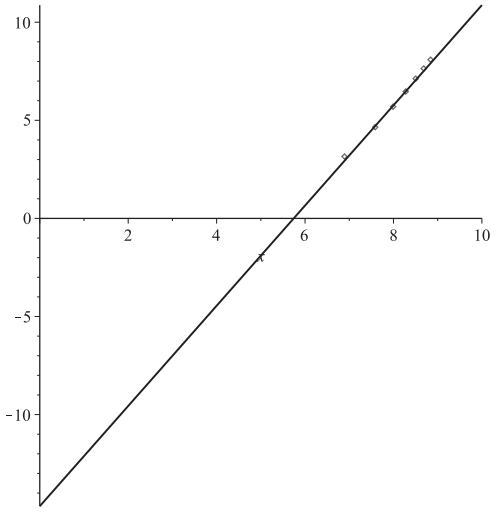

Figure 3.10: The length of the repetition sequences

an attempt takes longer; still the dependence is rather controlled with few outliers. We have applied the least squares method again, and the best fit for the running time on a logarithmic scale in this range is given by the line $y = 2.55 \cdot x - 14.67$. See Figure 3.10. Note that it is not equal to the running time of one iteration, we just wanted to check that the time of repetitions does not become unmanageable.

3.4 Conclusions and future improvements

To sum up the results of this chapter we describe two implementations of the ECPP primality test that uses refinements and a strategy based on the heuristics from the previous chapter and justify in practice, that it has advantages compared to an implementation without heuristics. The first implementation uses a light weight strategy that is appropriate to test numbers around 1000 digits. We showed a situation in our paper [10], when an implementation without a strategy, Magma-ECPP get stuck on a number, while for our implementation, the first version of Modified-ECPP, we could not find such number. Despite of the fact that the strategy requires additional time, the running time of Magma-ECPP ($2000 - 8000$ seconds), and the first version of Modified-ECPP ($2000 - 3000$ seconds) on numbers with 1000 digits, is similar, our implementation runs even more balanced from running time point of view.

The strategy of the first implementation was based only on the parameter

$d(n_{i,j})$ and the exponent of $\ln n_{i,j}$ in the other two parameters were fixed and it also did not take gain into account, thus we introduced another strategy, that finds the parameter for each $n_{i,j}$ that requires the least work to produce descendants and then selects the $n_{i,j}$ for which the quotient of the invested work and the expected gain is the smallest. We ran experiments on this implementation of the Modified-ECPP on numbers up to 7000 decimal digits and the running time in this range seems to be below $o\big(\ln^4(n)\big)$ for input n. Of course there is no experimental way to show that this is asymptotically correct. The amount of the running time needed for administration of the strategy to be applied is small, which is necessary for the strategy to be useful.

Experiments show that in around half of the cases the strategy chooses to increment $d(n_{i,j})$ (allow the use of larger discriminants) and in almost all the other cases it chooses to increase $s(n_{i,j})$ (allow larger primes in the discriminants). For bigger numbers enlarging $d(n_{i,j})$ is a bit more frequent, for smaller numbers enlarging $s(n_{i,j})$. Selection of $b(n_{i,j})$ (allow larger primes in the factorization of the m-s) hardly ever happens, although in some situations, e.g. when we run out of discriminants, or if we have already a huge set of curve orders, it is necessary to have the possibility to increase this parameter. As we saw that the number of repetitions does not exceed 13, the implementation should be able to work with numbers up to 10000 digits without running out of discriminants. After running out of discriminants it would be still possible to continue with increasing $b(n_{i,j})$ (there is no upper limit to that parameter). Of course testing such big numbers would take very long.

The number of the backtracks and repetitions are proportional to the length of the path. The maximum level of backtracks and the lengths of repetitions seem to be similar for different sizes of inputs. That is what we expect as the input selection depends on the estimated running times + the work that we have to do to reduce the new descendants to the same size. The size differences for backtracks are growing with bigger inputs, but the differences are negligible. According to the experiments there are negligible amount of extreme cases concerning the repetitions and backtracks.

The overall conclusion is that the implementation seems to be working as it was intended, but there is still space for improvement; speeding up the factoring would make it more likely that the strategy selects $b(n_{i,j})$. This has a pleasant effect on the size of the descendants and the length of the path. Currently we are using trial division up to 10830, batch trial division up to $2 \cdot 10^9$ and Pollard ρ method further.

One goal is to replace the current implementation of ECPP in Magma with the Modified-ECPP. Another goal is to provide an optimized implemen-

tation of the ECPP algorithm that combines the strategy of Modified-ECPP with a collection of highly optimized packages written in C and Assembly in order to be able to find bigger primes than the current general prime record.

Bibliography

[1] **Adleman, M.A., Pomerance, C., Rumely, R.S.,** On distinguishing prime numbers from composite numbers, *Annals of Mathematics* **117** (1983), 173–206

[2] **Agrawal, M., Kayal, N., Saxena, N.,** Primes is in P, *Annals of Mathematics* **160** (2004), 781–793

[3] **Atkin, A.O.L., Morain, F.,** Elliptic curves and primality proving, *Math. of Computation* **61** (1993), 29–68.

[4] **Bosma, W., Cannon, J., Playoust, C.,** The Magma algebra system. I. The user language, *J. Symbolic Comput.*, **24, 3-4**, (1997) 235–265.

[5] **Bosma, W., Cator, E., Járai, A., Kiss, Gy.** Primality proofs with elliptic curves: heuristics and analysis, *Ann. Univ. Sci. Budapest. Sect. Comput.* **44** (2015), 3–27.

[6] **Cohen, H.,** *A course in computational algebraic number theory*, Springer Verlag, 1993.

[7] **Farkas, G., Kallós, G., Kiss, Gy.,** Large primes in generalized pascal triangles, *Acta Univ. Sapientiae, Informatica* **3,** 2 (2011) 158–171.

[8] **Goldwasser, S., Kilian, J.,** Primality testing using elliptic curves, *Journal of the ACM* **46,** 4 (1999) 450–472

[9] **Gowers, T.,** *The Princeton Companion to Mathematics*, Princeton University Press, Princeton and Oxford, 2008.

[10] **Járai, A., Kiss, Gy.,** Finding suitable paths for the elliptic curve primality proving algorithm, *Acta Univ. Sapientiae, Informatica* **5,** 1 (2013) 35–52.

[11] **Kac, M.,** Statistical Independence in Probability, *Analysis and Number Theory*, Wiley, New York, 1959.

[12] **Kiss, Gy.,** Primality proofs with elliptic curves: experimental data, *Ann. Univ. Sci. Budapest. Sect. Comput.* **44** (2015), 197–201.

[13] **Kiss, Gy.,** A strategy for elliptic curve primality proving, *Acta Univ. Sapientiae, Informatica* **7,** 2 (2015) 125–142.

[14] **Knuth, D.E.,** *The Art of Computer Programming, Vol. 1–3.*, Addison-Wesley, 1968.

[15] **Knuth, D.E.,** *The Art of Computer Programming, Vol. 1–3. Second Edition*, Addison-Wesley, 1981.

[16] **Lenstra, A. K., Lenstra Jr., H. W.,** Algorithms in Number Theory, in: van Leeuwen, J., (Ed.), *Algorithms in Complexity, Vol.A*, Elsevier, 1990, 673–716.

[17] **Miller, G.L.,** Riemann's hypothesis and tests for primality, *Journal of Computer and System Sciences* **13** (1976), 300–317

[18] **Morain, F.,** Implementing The Asymptotically Fast Version Of The Elliptic Curve Primality Proving Algorithm, *Mathematics of Computation* **76**, 2007, 493–505

[19] **Pocklington, H.C.,** The determination of the prime or composite nature of large numbers by Fermat's theorem, *Proc. Cambridge Philosophical Society* **18** (1914), 29-30

[20] **Schoof, R.,** Counting points on elliptic curves over finite fields, *Journal de thorie des nombres de Bordeaux* **7,** 1 (1995) 219–254.

[21] **Schönhage, A., Grotefeld, A.F.W., Vetter, E.,** *Fast Algorithms: A Multitape Turing Machine Implementation*, B.I.Wissenschaftverlag, Mannheim, 1994.

Summary

This thesis collects the results of Gyöngyvér Kiss. She deals with the heuristic running time analysis and implementation of the elliptic curve primality proving (ECPP) algorithm of Atkin and Morain. ECPP is a recursive algorithm, that starts from an input probable prime $n = n_0$ and traverses a graph to get down to such n_l that is small enough to be easily proven prime.

In the work of Atkin and Morain the background and an exact implementation of the ECPP algorithm is described. Lenstra, Lenstra and Morain have found that, using asymptotically fast methods for multiplication, division, polynomial calculation, etc., and a trick attributed to J. O. Shallit, the heuristic running time of the algorithm can be to reduced to $O(\ln^{4+\varepsilon} n)$ for input possible prime n. We show that it is possible to reduce the heuristic running time to $o(\ln^4 n)$ using refinements based on certain assumptions. We give a detailed running time analysis of each step using these refinements. This is mostly result of a joint effort of Antal Járai and Gyöngyvér Kiss. As own result, the assumptions are also justified in practice on a manageable set of possible primes.

Two ECPP implementations of the author in Magma Computational Algebra System are described too. Based on the theory in the first part of the thesis, it possible to converge to the optimal choices of parameters. A strategy is applied to produce more then one descendants in each step and to determine the best of them. The first implementation uses a light weight strategy. It is best to apply it on numbers up to 1000 digits. The second implementation is rather designed for numbers with $1000 - 10000$ digits. In earlier implementations that we know of, it is only dealt with one descendant to go further down in this graph. The author managed to justify that in practice it is worthwhile to use strategies when traversing such graph.

This research is part of a project. The goal of the project is to combine a traversing strategy with a collection of highly optimized package collection written in C and Assembly.

Összefoglalás

Ez a disszertáció Kiss Gyöngyvér eredményeit gyűjti össze. A szerző az Atkin–Morain féle elliptikus görbés prímteszt (ECPP) futási idő elemzésével és implementációjával foglalkozik. Az ECPP egy rekurzív algoritmus, ami kiindulva egy $n = n_0$ valószínű prím inputból bejár egy gráfot, hogy elérjen egy olyan n_l számig, ami elég kicsi ahhoz, hogy könnyen bizonyithatjuk hogy prím.

Atkin és Morain munkájában az ECPP algoritmus elméleti háttere és pontos megvalósítása megtalálható. Lenstra, Lenstra és Morain megállapította, hogy aszimptotikusan gyors szorzást, osztást, polinom műveleteket, stb., használva, ezen kívül J. O. Shallit trükkjét alkalmazva az algoritmus heurisztikus futási ideje lecsökkenthető $O(\ln^{4+\varepsilon} n)$-ra n valószínű prím input esetén. Mi megmutatjuk, hogy a heurisztikus futási idő tovább csökkenthető $o(\ln^4 n)$-re, ha bizonyos feltevéseken alapuló finomításokat használunk. Meghatározzuk az algoritmus minden lépésének részletes futási idő elemzését a finomítások alkalmazásával. Ez a rész többnyire Járai Antal és Kiss Gyöngyvér közös munkájának eredménye. Saját eredményként a szerző igazolja a fenti feltevéseket gyakorlatban, valószínű prímek egy kezelhető méretű halmazán.

A tanulmányban a szerző két ECPP implementációja is található, Magma Számítógépes Algebra Rendszerben. Felhasználva az első rész elméleti eredményeit, lehetővé válik a paraméterek optimális értékéhez való konvergálás. A szerző által alkalmazott stratégia minden lépésben több leszármazottat állít elő és eldönti, hogy melyik a legjobb. Az első implementáció egy könnyű súlyú stratégián alapszik, amit 1000 jegyű számokig érdemes használni. A második implementáció inkább $1000 - 10000$ jegyű számokra lett tervezve. Általunk ismert előző implementációk csak egy leszármazottat kezeltek lépésenként a bejárás során. A szerző kísérletekkel támasztotta alá, hogy gyakorlatban érdemes stratégiát alkalmazni a gráfbejáráshoz.

Ez a kutatás egy projekt része, aminek célja hogy egy stratégiát kombináljon egy C és Assembly nyelven írott optimalizált programcsomaggal.

[1]**ADATLAP**
a doktori értekezés nyilvánosságra hozatalához

I. A doktori értekezés adatai
A szerző neve: **Kiss Gyöngyvér**
MTMT-azonosító: **10046284**
A doktori értekezés címe és alcíme: **An implementation and analysis of the ECPP algorithm**
DOI-azonosító[2]: **10.15476/ELTE.2016.124**
A doktori iskola neve: **ELTE Informatika Doktori Iskola**
A doktori iskolán belüli doktori program neve: **Numerikus és szimbolikus számítások**
A témavezető neve és tudományos fokozata: **Járai Antal az MTA doktora**
A témavezető munkahelye: **ELTE Informatika Kar (professor emeritus)**

II. Nyilatkozatok
1. A doktori értekezés szerzőjeként[3]

a) hozzájárulok, hogy a doktori fokozat megszerzését követően a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az ELTE Digitális Intézményi Tudástárban. Felhatalmazom az Informatika Doktori Iskola hivatalának ügyintézőjét Boda Annamáriát, hogy az értekezést és a téziseket feltöltse az ELTE Digitális Intézményi Tudástárba, és ennek során kitöltse a feltöltéshez szükséges nyilatkozatokat.

b) kérem, hogy a mellékelt kérelemben részletezett szabadalmi, illetőleg oltalmi bejelentés közzétételéig a doktori értekezést ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;[4]

c) kérem, hogy a nemzetbiztonsági okból minősített adatot tartalmazó doktori értekezést a minősítés (*dátum*)-ig tartó időtartama alatt ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;[5]

d) kérem, hogy a mű kiadására vonatkozó mellékelt kiadó szerződésre tekintettel a doktori értekezést a könyv megjelenéséig ne bocsássák nyilvánosságra az Egyetemi Könyvtárban, és az ELTE Digitális Intézményi Tudástárban csak a könyv bibliográfiai adatait tegyék közzé. Ha a könyv a fokozatszerzést követőn egy évig nem jelenik meg, hozzájárulok, hogy a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban.[6]

2. A doktori értekezés szerzőjeként kijelentem, hogy

a) az ELTE Digitális Intézményi Tudástárba feltöltendő doktori értekezés és a tézisek saját eredeti, önálló szellemi munkám és legjobb tudomásom szerint nem sértem vele senki szerzői jogait;

b) a doktori értekezés és a tézisek nyomtatott változatai és az elektronikus adathordozón benyújtott tartalmak (szöveg és ábrák) mindenben megegyeznek.

3. A doktori értekezés szerzőjeként hozzájárulok a doktori értekezés és a tézisek szövegének plágiumkereső adatbázisba helyezéséhez és plágiumellenőrző vizsgálatok lefuttatásához.

Kelt: Budapest, 2016.08.23.

a doktori értekezés szerzőjének aláírása

Kiss Gyöngyvér

---

[1] Beiktatta az Egyetemi Doktori Szabályzat módosításáról szóló CXXXIX/2014. (VI. 30.) Szen. sz. határozat. Hatályos: 2014. VII.1. napjától.
[2] A kari hivatal ügyintézője tölti ki.
[3] A megfelelő szöveg aláhúzandó.
[4] A doktori értekezés benyújtásával egyidejűleg be kell adni a tudományági doktori tanácshoz a szabadalmi, illetőleg oltalmi bejelentést tanúsító okiratot és a nyilvánosságra hozatal elhalasztása iránti kérelmet.
[5] A doktori értekezés benyújtásával egyidejűleg be kell nyújtani a minősített adatra vonatkozó közokiratot.
[6] A doktori értekezés benyújtásával egyidejűleg be kell nyújtani a mű kiadásáról szóló kiadói szerződést.