Acta Univ. Sapientiae, Informatica **7,** 2 (2015) 125–142

DOI: 10.1515/ausi-2015-0015

A strategy for elliptic curve primality proving

Gyöngyvér KISS
Eötvös Loránd University
Budapest
email: kissgyongyver@gmail.com

Abstract. This paper deals with an implementation of the elliptic curve primality proving (ECPP) algorithm of Atkin and Morain. As the ECPP algorithm is not deterministic, we are developing a strategy to avoid certain situations in which the original implementation could get stuck and to get closer to the situation where the probability that the algorithm terminates successfully is 1. We apply heuristics and tricks in order to test the strategy in our implementation in Magma on numbers of up to 7000 decimal digits and collect data to show the advantages over previous implementations in practice.

1 Introduction

The elliptic curve primality proving (ECPP) algorithm of Atkin and Morain [1] starts from an input probable prime n and is called recursively on probable primes of decreasing size in order to reach a probable prime whose primality can be easily verified. In the paper of Atkin and Morain an implementation of the ECPP algorithm is described. Further descriptions can be found in the work of Lenstra, Lenstra [6] and Morain [7]. The aim of one recursive step of these implementations is to find a new probable prime: the input for the next recursive step. As not all numbers are equally suitable, it is useful to produce

Computing Classification System 1998: F.2.1
Mathematics Subject Classification 2010: 11Y11, 11A51, 14H52
Key words and phrases: ECPP, elliptic curves, primality proving

more than one probable prime in one step and select 'the best one'. In this paper we describe an implementation in the Computational Algebra System Magma [2] that applies a strategy to control the selection of the input for the next step. We refer to the original implementations (see in [7], [2]) as ECPP and our modified implementation as Modified-ECPP in the rest of the paper.

This is the second implementation of Modified-ECPP within the confines of a project that, besides investigating different strategies on the 'Downrun', deals with the heuristic running time, and questions and assumptions related to this topic. The details of this project can be found in the work of Farkas, Kallós, Kiss [4], Járai, Kiss [5] and Bosma, Cator, Járai and Kiss [3]. One goal is to replace the actual ECPP implementation that is used in Magma.

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

2 Implementation of ECPP: details and tricks

The description, analysis, and implementation details of ECPP and the theoretical background of the statements below can be found in our paper [3]. Here we give a brief outline.

The input of the algorithm is a large probable prime n; this is a positive integer that has passed some probabilistic primality tests. The aim is to provide a *proof* for its primality.

Algorithm 1 : ECPP

(P) Starting with $n_0 = n$*, build up a sequence of probable primes* $n_0, n_1, \ldots, n_l$*, such that* n_l *is small enough for the primality of* n_l *to be recognized easily.*

(F) For each of the integers n_i *with* $i = 0, 1, \ldots, l-1$*, build up the elements of the proof (consisting of pairs* (E_i, P_i) *of an elliptic curve and a point on it).*

(V) Verify that n_l *is prime and verify the other steps of the primality proof (by showing that* P_i *has order* n_{i+1} *on* E_i *modulo* n_i *and that* n_{i+1} *exceeds* $(n_i^{\frac{1}{4}} + 1)^2$*, for* $i = l-1, l-2, \ldots, 0$*).*

In this paper we only deal with stage (P). On one hand the other two stages are not a risk in the sense that they always terminate successfully, on the other hand the other stages can be highly parallelized.

Next we describe the stages of algorithm (D), with input set n_{i_j}, $j = 1 \dots t$ briefly; this is part of stage (P), but the actual relation to it is described later on.

Algorithm 2 : Downrun

(D) For each n_{i_j} select a set of negative discriminants D that is suitable for n_{i_j} such that for integers u (determined by D), $n_{i_j} + 1 - u$ is the product of small primes and a probable prime q that exceeds $(\sqrt[4]{n_{i_j}} + 1)^2$. This is done as follows:

(1) Find a list of discriminants D for each n_{i_j} for which the binary quadratic form $n_{i_j}x^2 + Bxy + \frac{B^2-D}{4n_{i_j}}y^2$, where $B^2 \equiv D \bmod n_{i_j}$, provides v with $v \cdot \overline{v} = n_{i_j}$ (cf. [3, 6]). Store the pairs $(D, \pm u)$ for each n_{i_j} where $u = v + \overline{v}$.

(2) From the list of $(D, \pm u)$ from the previous step, select those for which $m = n_{i_j} + 1 - u$ can be factored as $m = q \cdot f$ with q a probable prime exceeding $(n_{i_j}^{\frac{1}{4}} + 1)^2$ and f is completely factored.

(3) Store the set of tuples (D, u, q) for each n_{i_j}.

(4) From the possible choices of (D, u, q) select a subset that will be the input for the next iteration.

An iteration step (D) in our Modified-ECPP differs slightly from an iteration in the original ECPP.

In the original ECPP, in general the i-th iteration of (P) consists of running (D) with only one input n_i and the outcome is just one q. At the first successful q, the i-th iteration returns and q becomes n_{i+1}, the input of the $(i+1)$-th iteration. If there is no new q after running through a predefined discriminant set, it backtracks and returns n_{i-1} as n_{i+1}. What actually happens after backtracking is that it goes further on the discriminants starting (D_{i+1}) right after the last successful D.

In the case of Modified-ECPP, the i-th iteration of (P) can have one or more inputs n_{i_j}. An initial (D) runs on the input set n_{i_j}. This can result in zero, one or more q-s. If there is at least one q, the iteration returns all of them and they become the inputs of the $(i+1)$-th iteration. If there is no new q after processing the discriminants up to certain limits, we select the *best* from a list of n_k, the results of the previous iterations. The selected n_k becomes the only input of a new run of (D), running on a new set of discriminants, or factoring

the existing m-s harder. Note that this is still the i-th iteration. The iteration only returns when there is at least one new q produced. If not ambiguous, we will denote n_{i_j} with n_k.

We have chosen this generic way above to describe (D), because it is applicable to both Modified-ECPP that runs more (D)-s on a set of numbers and returns a set of q-s and to ECPP that runs one (D) on one input and either returns the first successful q or backtracks.

2.1 Parameters

There are three main parameters that we use in the algorithm to control the Downrun, introduced in the work of Atkin and Morain [1]. As they play a major role in our strategy we describe them here briefly.

Parameter d – In (D1) 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. Unlike ECPP, an iteration of Modified-ECPP does not stop at the first successful discriminant, but processes all of them up to d. In stages (D1) and (D2) 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 m and q 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 (D1) we also have to extract the square root of discriminants D modulo n_i, which can be done faster if we extract the modular square roots of all the 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 root extraction and 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 m-s, performed in stage (D2). 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 m-s. The bound on these primes is b.

Most of the ECPP implementations use these parameters as fixed limits. For example in [1], d is taken to be 10^6, for practical purposes.

As we mentioned earlier, Modified-ECPP deals with a set of q-s during the Downrun. In our case, we control the size of this set with the above parameters.

The parameters depend on n_k; by our choice these parameters will all be taken of the form

$$a \ln^{c_1} n_k \ln\ln^{c_2} n_k.$$

For this reason, in the rest of the paper $d(n_k)$, $s(n_k)$ and $b(n_k)$ shall denote them. Initially we provide the values a, c_1 and c_2 for them. Further on if an iteration does not provide new q-s, backtracking and repetition (when we force the same n_k with bigger and bigger parameters) is also possible. In these cases the parameters are increased by multiplying with a constant c, which is also a parameter, while the exponents remain unchanged.

2.2 Tricks

Some data used in the algorithm is independent of the choice of n_k, so it is possible to collect it in advance.

In stage (D1) we need a list of the $s(n_k)$-smooth discriminants up to $d(n_k)$, that will be suitable for n_k. We check two necessary but not sufficient conditions; Jacobi symbols $\left(\frac{D}{n_k}\right) = 1$ and $\left(\frac{n_k}{p}\right) = 1$ for each prime divisor p of D, cf. [1], in order to reduce the possibility of failure when reducing the quadratic form in (D1). We will refer to this check further on as the Jacobi symbol filter. We also have to extract the square root of the D-s $\mod n_k$; doing this for the prime divisors of the discriminants will be more efficient as many primes will occur for several discriminants. Both of these points suggest to store the discriminants together with their prime factors and the primes themselves in preprocessed files. For estimation purposes [3] we also need the class number of the discriminants, which will be preprocessed too.

This way we can run through the prime file up to $s(n_k)$, checking $\left(\frac{n_k}{p}\right) = 1$ and extract the square roots $\mod n_k$, and then collect those discriminants up to $d(n_k)$ which have only appropriate prime divisors and build up the square roots of them after checking $\left(\frac{D}{n_k}\right) = 1$. We have such a file of discriminants up to 10^9. As our initial value of $d(n_k)$ is

$$\frac{1}{16e^{2\cdot\gamma}} \ln^2 n_k \ln\ln^{-2} n_k,$$

running out of discriminants should not be a problem. The primes are collected up to $2.7 \cdot 10^9$.

In stage (D2) we have to factor the m-s in order to acquire the input for the next iteration. In our implementation we use Batch Trial Division [3] up to $2.7 \cdot 10^9$ and the Pollard ρ method after that. In Batch Trial Division one takes

GCDs of products of number that are to be factored with products of primes, a list of prime products is stored to avoid the time consuming multiplication of primes on the fly. We store pairs of prime products with size $2^t \cdot 10^6$ where $t = 0, \ldots, 12$. If we use factorization limit b, the size of the product of the primes up to b is around e^b. As $2^{2^t \cdot 10^6} \asymp e^b$, we have $b \asymp 2^t \cdot 10^6 \cdot \ln 2$. This way we have a product from 0 up to around $2^t \cdot 10^6 \cdot \ln 2$ and from around $2^t \cdot 10^6 \cdot \ln 2$ up to around $2^{t+1} \cdot 10^6 \cdot \ln 2$ and they are both of size $2^t \cdot 10^6$ bits. This is useful, because depending on $b(n_k)$, we just have to find the right place in the file and get the appropriate product. We store pairs to have the possibility to start from 0 if it is the first factoring attempt, or from $2^t \cdot 10^6 \cdot \ln 2$ if we have already tried to factor below that.

2.3 Strategy

The detailed theoretical background of the strategy that the program uses is described in [3]. Here we list the most important notation and facts.

By $e_k = e\big(s(n_k), d(n_k)\big)$ we denote the number of m-s that we gain in stage (D2) after processing a set of $s(n_k)$-smooth discriminants up to $d(n_k)$. The expected value of e_k is

$$\bar{e}\big(s(n_k), d(n_k)\big) = \sum_D \frac{1}{h(D)},$$

where $h(D)$ denotes the class number of discriminant D.

After applying the Jacobi symbol filter, that uses arithmetic properties of n_k and its prime factors, this expected value changes to

$$\bar{e_k} = \bar{e}\big(n_k, s(n_k), d(n_k)\big) = \sum_D \frac{2^t}{h(D)},$$

where t is the number of different prime factors of D.

The expected number of m-s that split as required (that is, for which the second largest prime divisor is less than $b(n_k)$) is

$$\lambda_k = \lambda\big(s(n_k), d(n_k), b(n_k)\big) = e^\gamma \frac{\ln b(n_k)}{\ln n_k} e_k.$$

The progress we make is measured by the size difference between n_k and the q that is produced by (D) with n_k as input; the expected value of that 'gain' is

$$G_k = G\big(b(n_k)\big) = \ln b(n_k).$$

In the rest of the paper we denote by q numbers that have been produced already, and by q' we denote numbers that are not produced yet, but for which an estimation has been made for the amount of work needed to produce them for a given n_k. Note that q-s become n_k-s, when they become one of the inputs of the next iteration.

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.

Choosing the best q

We saw that λ_k is the expected number the successful m-s, so the expected number of new q-s. From [3] we know that we are likely to succeed when λ_k exceeds 1.

On the other hand all the new q-s become the input of the next iteration and we run an initial (D) on them with certain (small) values of parameters s, d and b. If this does not provide at least one new q, then we select the *best* n_k as the input of the next (D).

The reason why we cannot choose the *best* n_k right away and have to run an initial (D) on the newly produced numbers, lies in our definition of *best*. There are two aspects to this.

First of all, we have to consider that the numbers are not equally appropriate. Applying the Jacobi symbol filters on n_{k_1} could filter out more discriminants than on n_{k_2}, so to produce the same number of new q-s we would have to process a larger number of discriminants or use bigger factoring bound for n_{k_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 q-s on input n_k.

The second aspect is the size of the produced q-s. The smaller they are, the faster we get to the small primes.

The first aspect we can estimate with the help of λ; for estimating the running time to produce certain amount of q-s, we collect some actual running times to see how the numbers behave. This information can be collected from the initial (D) runs. The parameters of these initial runs are chosen as follows:

$$s_0(n_k) = \lambda_0 \cdot \frac{1}{2 \cdot e^{\gamma}} \ln n_i \ln \ln^{-1} n_k$$

$$d_0(n_k) = \lambda_0^2 \cdot \frac{1}{4 \cdot e^{2\cdot\gamma}} \ln^2 n_i \ln\ln^{-2} n_k$$

$$b_0(n_k) = \ln^2 n_k$$

The choice for $s_0(n_k)$ comes from taking $\lambda = e^{\gamma \frac{\ln b(n_k)}{\ln n_k}} e_k$ if we suppose that $\lambda = \lambda_0$ is a parameter and $s(n_k) \approx e_k$; we take $d(n_k)$ equal to $s^2(n_k)$ despite of [3], where we state that it is better to keep the value of s below $\sqrt{d}$, because in practice, on small numbers, taking $s < \sqrt{d}$ led to some difficulties.

During the 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 new q-s are produced. With this information, we can estimate the time needed when we increase s, d or b separately. For the first two we can estimate the increment in the value of $\bar{e_k}$ when the smoothness bound for the discriminants up to $d_0(n_k)$ is increased from $s_0(n_k)$ to $c \cdot s_0(n_k)$, and separately, the effect of including the $s_0(n_k)$-smooth discriminants up to $c \cdot d_0(n_k)$ rather than $d_0(n_k)$. We filter out the discriminants that are appropriate for n_k and determine $\bar{e_k}$ in both cases. We can directly determine the value of $b(n_k)$ using the actual e_k.

Now we can compute λ_k in all three cases and we can also estimate the time t_k it would take in all three cases to execute (D) with the estimated values of one parameter while the other two remain unchanged. Then we can store the different $\frac{t_k}{\lambda_k}$ values.

We now know the expected number of q'-s resulting from increasing one of the parameters, but we do not know the expected size of them. This can be determined with the help of the gain function G_k, which depends only on $b(n_k)$, the factorization effort on m. From this we can see that we gain q'-s with the smallest expected size if we increase b, also if we increase s or d, the expected size of the q' that we gain are the same. After incrementing s or d, we want to know how much effort it takes to reduce q' further: what is the average work per bit needed to decrease q'? This we can estimate from the previous iterations. After multiplication by the estimated size differences, we obtain a value, a_k-s for the expected effort of reducing q'-s to the size of the smallest one. Then compare the values $\frac{t_k}{\lambda_k} + a_k$ and select that parameters for which this value is minimal. We denote this minimal value by mt_k, for the given n_k. Now the *best* number is the n_k for which this value mt_k is the smallest.

Note that the running time of the three bottleneck subroutines (extracting square roots modulo n_k, quadratic form reduction, integer factoring) depends

(via the three parameters) only on n_k, so it is possible to express these running time as a function of n_k. Estimating the running time of square root extraction modulo a prime and of form reduction is fairly easy, because we can measure the running time of a single such operation and multiply by the number of times we need to perform them (which is the number of primes in n_k 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 m-s nor to the number of primes used, but as we do not use huge amount of m-s or primes simultaneously, linear approximation works well in practice. For $b(n_k) = 2^t \cdot 10^6 \cdot \ln 2$ we double the expected time if we increase t to $t+1$ as we have to deal with products of twice the size.

If no new q-s are produced, we need to be able to backtrack. We keep a window with a certain number of n_k-s for which we store all the data that is necessary to continue using this value of n_k if turns out to be the *best*. Newly found n_k-s are always going to the window, and if the number of the n_k-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 the expected work to decrease one bit for values n_k in this window and update it after each iteration.

Note that while processing a number, the running times that are stored will be updated with the new data; estimation takes place directly after processing, so we can base our decision on an up-to-date set of data.

Resulting algorithm

Now we describe briefly how we make our estimates and decisions, followed by the description of one iteration. For the notation used we refer to the previous subsection.

Algorithm 3 : Estimation Algorithm

(E) The Estimation Algorithm determines the value of mt_{i_j} *for a value of* n_{i_j}*. The input is* $s(n_{i_j})$*,* $d(n_{i_j})$ *and* $b(n_{i_j})$*, the values of the main parameters of the previous call of* (D) *on* n_{i_j} *as input.*

(1) Determine the effect of increasing s *or* d *by computing the values of* $e\big(n_{i_j}, c \cdot s(n_{i_j}), d(n_{i_j})\big)$*, and* $e\big(n_k, s(n_{i_j}), c \cdot d(n_{i_j})\big)$*: collect the* $c \cdot s(n_{i_j})$*-smooth discriminants up to* $d(n_{i_j})$ *and* $s(n_{i_j})$*-smooth discriminants up to* $c \cdot d(n_{i_j})$*, that are appropriate for* n_{i_j}*.*

(2) *From the actual running times stored for* n_{i_j}*, determine* t_{i_j} *for* $c \cdot s(n_{i_j})$, $c \cdot d(n_{i_j})$, $2^t \cdot b(n_{i_j})$.

(3) *Compute the expected gain* G_{i_j}*, and compute the values* a_{i_j} *for each parameter.*

(4) *Determine and store the value of* mt_{i_j} *together with the corresponding lists of discriminants and primes.*

Algorithm 4 : Modified-ECPP Iteration Step

(P) *With a set of one or more* n_{i_j}*-s as input, one iteration of the Modified-ECPP Algorithm runs until it finds one ore more new* q*-s that become the inputs for the next iteration. The following steps are carried out.*

(1) *Run* (D) *on each* n_{i_j} *with* $s_0(n_{i_j})$, $d_0(n_{i_j})$ *and* $b_0(n_{i_j})$.

(2) *Run* (E) *on each* n_{i_j} *in order to determine* mt_{i_j}*, with the data collected in the previous step on the running times. Add the* n_{i_j}*-s to the window. If we obtain new* q*-s go to* (1)*, else to* (3)

(3) *Reorder the window by the values* mt_k *(all of them are up to date).*

(4) *Pick the best as* n_{i_j} *and run* (D) *on it.*

(5) *Run* (E) *on the selected* n_{i_j} *with the data collected in the previous step.*

(6) *If we have new* q*-s go to* (1)*, else to* (3)

We list the additional important parameters, besides the ones mentioned in the previous sections.

Parameter λ_0 – In (1) this parameter provides the initial value of λ. If it is too big, (1) becomes too slow, and also the process would result in too many new q-s and the tree of n_k-s would expand too much. On the other hand if it is too small, we cannot collect realistic data about the running time. It seems to be appropriate to keep this value between $1/3$ and $1/2$.

Parameter c – In (E) we take $c \cdot s(n_k)$, $c \cdot d(n_k)$ as next values of these parameters. If this is too big, running (D) becomes too slow, if it is too small, we spend to much time on administration because we take too small steps. The value seems to be appropriate between $3/2$ and 3.

Parameter α – In (E) we increase the value of λ with $\delta\lambda$. We require a certain lower bound on $\delta\lambda$, that is α. If $\delta\lambda$ is above α the value of c is acceptable, below it we have to multiply c with $\alpha/(\delta\lambda)$. We use $\alpha = 0.25$;

Parameter *wSize* – This parameter denotes the size of the window, that we have mentioned above. We have to be careful with it because if we store too big window, the program becomes slow, otherwise, if it is too small, it is possible that the strategy would select a node that is outside of the window, so we restrain it. This parameter is of the form $\ln n/(c \ln \ln n)$.

3 Analysis of the strategy and running times

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

Running times

Figure 1 shows the running times for the main experiment of the algorithm. The number of decimal digits of the input numbers appears on the x-axis on both graphs, while the y-axis indicates the running time in seconds on the first graph, the average running times of the algorithm on the same numbers; for $k = 1, 2, 3, \ldots, 7$ vertically the average of the running times for the numbers of $k \cdot 1000$ digits is indicated as a single dot on the second graph.

Applying a Least Squares linear approximation method, we found that the best fit for the running time on a logarithmic scale is given by the line $y = 3.86 \cdot x - 21.00$. This is displayed in Figure 2.

It is possible to play around with the running time, for example by changing the parameter c that is responsible for the size of the increase of the parameters s and d. We tested numbers from 1000 up to 3000 digits with $c = 1.5, 2, 2.5, 3, 3.5, 4$.

The effect on the running time can be seen in Figure 3. On the x-axis the different values of c are given, and on the y-axis the running time. We see that effect of changing c in this range is insignificant.

The time spent in a single iteration can be separated into administration time and the execution time. We consider filtering the discriminants for given n_k and the estimation process as part of the 'administration', and time spent on extracting modular square roots $\bmod n_k$, reduction of forms, factoring

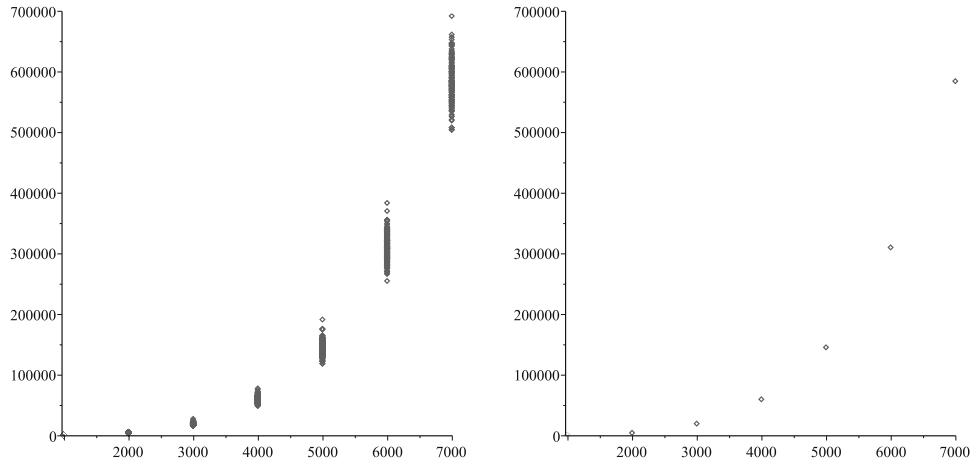

Figure 1: Running times

and Miller-Rabin tests on the new q-s as part of the 'execution'. 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 4 shows the proportion of the total execution time per run to the total running time per run. The total running times are indicated on the x-axis, and on the y-axis the execution times for numbers from 1000 digits up to 7000 digits are displayed (together with the line $y = x$, 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.

Experiments on the strategy

In the analysis of the strategy we put emphasis on backtracks and repetitions. Repetitions and backtracks are very similar and occur for the same reason: they indicate that we could not provide a new q after executing the initial (D) as well as a run of (D) on the best available input (the initial run of D is only used to collect data on the number and is considered only as precomputation). In both cases we have to select a new value of n_k to continue on. The only difference is that when we backtrack we select a different number, whereas in a repetition the same number is selected again (because it is still 'best'). It is natural that repetitions occur frequently as we increase λ with around $\alpha = 0.25$ instead of 1, which means that one may expect to repeat the procedure as much

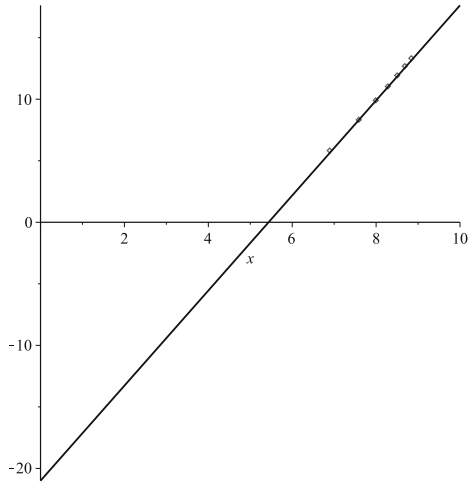

Figure 2: Average Running times on logarithmic scale

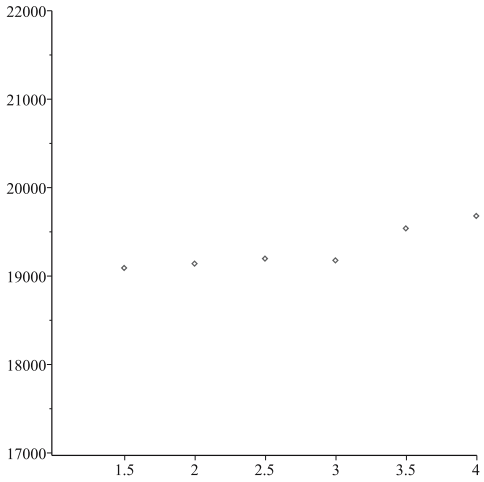

Figure 3: Changes of the running time if we change parameter c on 3000 digit numbers

as 4 times before producing a new q.

Note that when we backtrack, we may step either backwards or forward, as the window may contain numbers for both situations.

Note that the length of the path is not equal to the number of iterations, as we include each number on which (D) was ever called in the path, and in

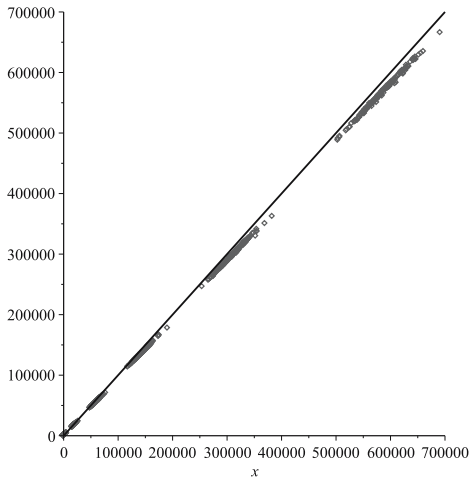

Figure 4: The proportion of the execution time compared to the total running time

one iteration (D) may be called several times. However, the number of the iterations is equal to the maximum *level*, as we consider numbers produced in the same iteration to be on the same level.

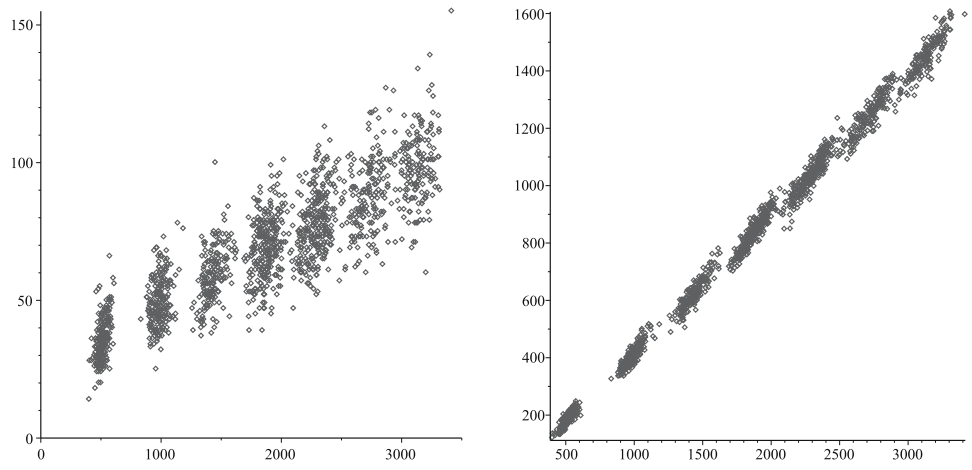

Figure 5: The number of backtracks as a function of the length of the paths

Figure 5 show the proportion of the number of backtracks and repetitions to the length of the path, for numbers of various sizes. On the x-axis the

path lengths are indicated, on the y-axis, the number of backtracks (on the first graph) or repetitions (on the second graph). 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 $k = 1, \ldots, 7$. From Figure 5 it is clear that we need backtracks during the Downrun, and as a rule of thumb, approximately once every 30 steps on the path. We can also see that around half of the path is repetition.

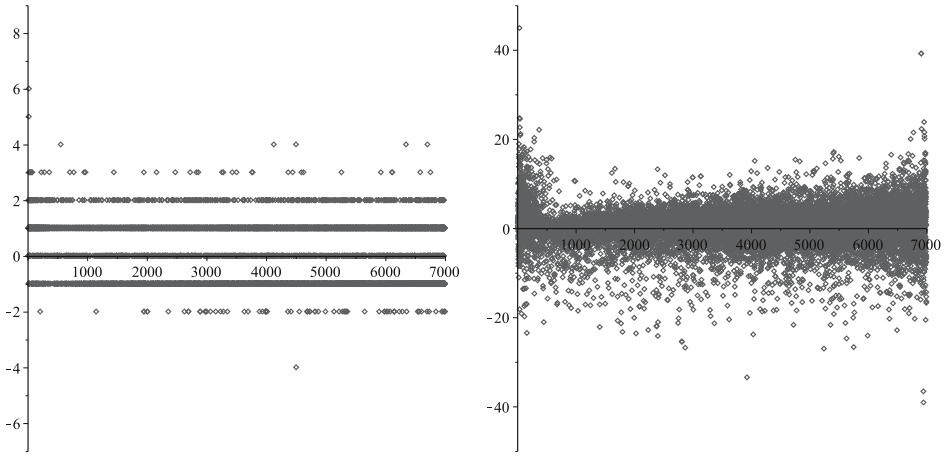

Figure 6: The level and size differences of backtracks

Besides knowing how often we backtrack in a run, we would like information on how far back we step (in terms of level or size) when backtracking. There should not be too big size or level differences, as in this case the effort invested in decreasing the size of the n_k-s is lost. In Figure 6 we present the level and size differences that occurred up to 7000 digits. The x-axis on both graphs indicates the size of the numbers from which we backtracked. On the y-axis on the graph on the left the level differences are indicated, and on the graph on the right the size differences (in number of digits). 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. It is clearly suggested by these graphs that neither the level nor the size differences depend on the size of the input; only for really small numbers, when the work that we loose with backtracking is small, the size differences can be relatively large. There are very few outliers for larger sizes.

The length of repetition sequences and how long they take, is also interest-

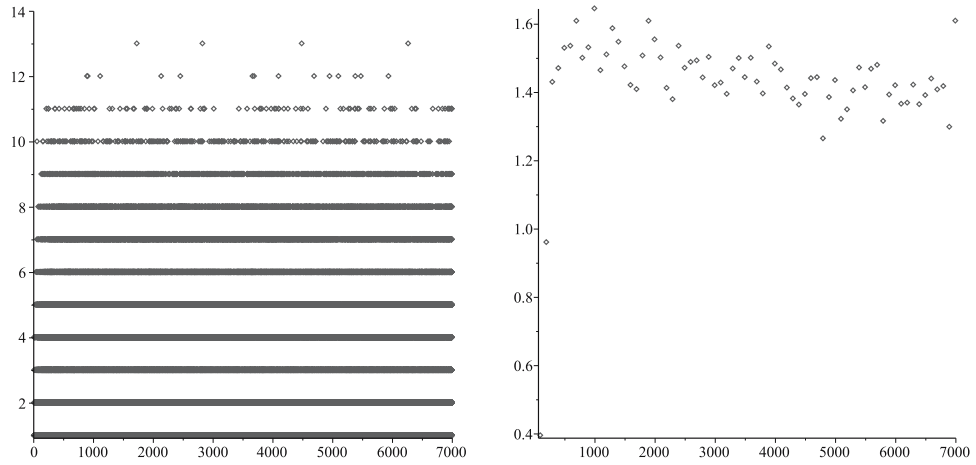

Figure 7: The length and proportion of the repetition sequences

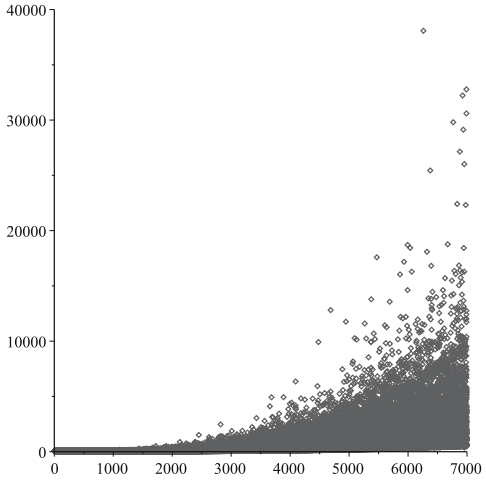

Figure 8: The time of the repetition sequences

ing. The result of our experiments on this can be seen in Figure 7 and Figure 8. In x-direction in each case the size of the number that is repeated is indicated. The y-axis in the first graph of Figure 7 indicates the length of the repetition sequence, in the second graph it indicates the percentage of the number of the repetitions on numbers of a certain size compared to the total amount of repetitions. In Figure 8 the y-axis indicates the time of the repetition sequences.

We see that the length of the repetition sequence does not really depend on the size of the input, it never exceeds 13. The percentage of the number of repetitions seems to be decreasing when the numbers are growing, though at 7000 digits the percentage is higher. The reason is that when we start on a number with 7000 digits we have to increase our effort on it until there are new q-s produced, as this is the only choice at that point. The time of course does depend on the size of the numbers, as for bigger numbers (D) takes longer; still the dependence is rather controlled with few outliers.

4 Conclusions and future improvements

To sum up the results from our experiments we make the following observations.

The evidence from experiments with numbers up to 7000 decimal digits is that the running time is below $o\left(\ln^4(n_0)\right)$. Of course there is no experimental way to show that this is asymptotically correct.

The proportion 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 (meaning: allow the use of larger discriminants) and in almost all other cases it chooses to increase s (that is: allow larger primes in the discriminants). For bigger numbers enlarging d is a bit more frequent, for smaller numbers enlarging s. Selection of b (that is: allow larger primes in the factorization of the m-s) hardly ever happens. 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 (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 and seems to be independent from the size of the input.

The maximum level of backtracks and the lengths of repetitions seem to be the 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 q-s to the same size. The work to reduce the new q-s; the avgWork, is growing for bigger inputs, but also the estimated running times, thus their relation should be the same. The size differences for backtracks are growing with bigger inputs, but the differences are negligible.

The time of the repetition sequences are growing also, but without too many extreme cases and of course for bigger input the execution time grows.

The overall conclusion is that the implementation seems to be working as it was intended, but there is still space for improvement. The goal is to provide an optimized implementation of the ECPP algorithm written in C that combines this strategy with a collection of highly optimized package written in C and Assembly.

References

[1] A. O. L. Atkin, F. Morain, Elliptic curves and primality proving, *Math. Comp.* **61,** 203 (1993) 29–68. ⇒125, 128, 129

[2] W. Bosma, J. Cannon, C. Playoust, The Magma algebra system I: the user language, *J. Symbolic Comput.*, **24,** 3 (1997) 235–265. ⇒126

[3] W. Bosma, E. Cator, A. Járai, Gy. Kiss, Primality proofs with elliptic curves: heuristics and analysis, *Ann. Univ. Sci. Budapest. Sect. Comput.*, in print ⇒126, 127, 129, 130, 131, 132

[4] G. Farkas, G. Kallós, Gy. Kiss, Large primes in generalized pascal triangles, *Acta Univ. Sapientiae*, *Informatica* **3,** 2 (2011) 158–171. ⇒126

[5] A. Járai, Gy. Kiss, Finding suitable paths for the elliptic curve primality proving algorithm, *Acta Univ. Sapientiae*, *Informatica* **5,** 1 (2013) 35–52. ⇒126

[6] A. K. Lenstra, H. W. Lenstra, Jr., Algorithms in number theory, in: *Handbook of Theoretical Computer Science, Vol. A. Algorithms and Complexity* (ed. J. van Leeuwen), Elsevier, 1990, pp. 673–716. ⇒125, 127

[7] F. Morain, Implementing the asymptotically fast version of the elliptic curve primality proving algorithm, *Math. Comp.* **76,** 257 (2007) 493–505. ⇒125, 126

Received: November 16, 2015 • Revised: December 13, 2015