Acta Univ. Sapientiae, Informatica, **5,** 1 (2013) 35–52

DOI: 10.2478/ausi-2014-0003

Finding suitable paths for the elliptic curve primality proving algorithm

Antal JÁRAI
Eötvös Loránd University
Faculty of Infomatics
email: `ajarai@moon.inf.elte.hu`

Gyöngyvér KISS
Eötvös Loránd University
Faculty of Infomatics
email: `kissgyongyver@gmail.com`

Abstract. An important part of the Elliptic Curve Primality Proving algorithm consists of finding a sequence of elliptic curves with appropriate properties. In this paper we consider a strategy to search for an improved sequence, as part of an implementation (implemented in Magma 2.19) to obtain improved heuristics and compare it to an implementation which does not use such heuristics, namely to a built-in Magma function.

1 Introduction

Although mathematicians have been interested in prime numbers since ancient times, there is still no general, deterministic, unconditional, practical, polynomial time algorithm for primality proving. If we are willing to drop some of these adjectives, the situation becomes different. There exist tests of Lucas-Lehmer type that can certify primes of very large size but only of a special form. The Miller-Rabin test has a version that is practical and runs in polynomial time but only provides primality proofs conditional on a generalized version of the Riemann hypothesis; the variant commonly used only produces *probable primes*, in the sense that with small probability a composite number will pass the tests. The now famous AKS test [1], on the other hand, is deterministic and proves primality in polynomial time, but has yet to be proven practical; for an improved randomized version see Bernstein [3].

Computing Classification System 1998: F.2.1, G.4
Mathematics Subject Classification 2010: 11Y11, 11A51, 14H52, 97R20
Key words and phrases: ECPP, elliptic curves, primality proving, Magma

Somewhere in between there are two algorithms that can prove primality in situations of practical importance (primes of hundreds or several thousands of decimal digits), of which the complexity analysis shows sub-exponential dependency on the size of the prime, but for which polynomial time bounds have not been proven. The significance of such primality tests has increased with the widespread use of primes for cryptographic purposes.

This paper aims to describe an implementation of one of the two successful practical tests for primality proving, *ECPP* see [2], written in Magma, a high performance software system. The test is based on elliptic curve arithmetic, by looking at heuristics for an optimal choice of parameters and next step in the recursion and to compare it to an implementation without heuristics, which is a part of Magma.

In what follows, we will always assume that n is the input of our algorithm, for which we want to construct a primality proof; also, we assume that n is a probable prime in the sense that it has passed some compositeness tests, and that it is free of small divisors. In particular, $\gcd(n, 6) = 1$. However of course, we do not *assume* that n is prime.

2 Elliptic curves

The main objective in the Elliptic Curve Primality Proving (*ECPP* for short) algorithm, which will be described in detail in the next section, is to construct a sequence of integers $n_0, n_1, \ldots, n_k$ that will be proved prime in reversed order, ending at $n_0 = n$. When the proof is completed, these numbers n_i will be (divisors of) orders of groups of points of elliptic curves over finite fields, as they are defined modulo n_{i-1}. However, during the construction we can not use yet that n_{i-1} is prime, and this means that we will have to be careful in defining elliptic curves modulo n, and their arithmetic; see [8].

Definition 1 *The projective plane modulo* n*, denoted* $\mathbb{P}^2(\mathbb{Z}/n\mathbb{Z})$*, for a positive integer* n*, consists of equivalence classes* $(x : y : z)$ *of triples* $(x, y, z) \in (\mathbb{Z}/n\mathbb{Z})^3$ *satisfying* $\gcd(x, y, z, n) = 1$*, under the equivalence* $(x, y, z) \sim (\lambda x, \lambda y, \lambda z)$ *for any* $\lambda \in (\mathbb{Z}/n\mathbb{Z})^*$.

Definition 2 *Let* n *be an integer with* $\gcd(n, 6) = 1$*. An elliptic curve* E *modulo* n *is a pair* $(a, b) \in (\mathbb{Z}/n\mathbb{Z})^2$ *for which* $\gcd(4a^3 + 27b^2, n) = 1$*. The set of points* $E[\mathbb{Z}/n\mathbb{Z}]$ *on an elliptic curve* E *modulo* n *consists of* $(x : y : z) \in \mathbb{P}^2(\mathbb{Z}/n\mathbb{Z})$ *for which*

$$y^2 z = x^3 + axz^2 + bz^3.$$

Definition 3 *Let n be an integer with $\gcd(n, 6) = 1$, and $a \in \mathbb{Z}/n\mathbb{Z}$. Define $V = V[\mathbb{Z}/n\mathbb{Z}]$ as the set of all $(x : y : 1) \in \mathbb{P}^2(\mathbb{Z}/n\mathbb{Z})$ together with $O = (0 : 1 : 0) \in \mathbb{P}^2(\mathbb{Z}/n\mathbb{Z})$. Given (V, a), the partial addition algorithm computes for any pair $P = (x_p : y_p : z_p)$, $Q = (x_q : y_q : z_q) \in V$ either an element $R = (x_r, y_r, z_r) \in V$ called the sum $P + Q$ of P and Q, or a non-trivial divisor d of n, as follows.*

(1) If $x_p = x_q$ and $y_p = -y_q$, then output $R = (0 : 1 : 0)$.

(2) If $x_p \neq x_q$ and $y_p = -y_q$, then let $v = x_p - x_q$, otherwise let $v = y_p + y_q$; then use the extended Euclidean algorithm to compute $s, t \in \mathbb{Z}/n\mathbb{Z}$ such that $sv + tn = d = \gcd(v, n)$. If $d > 1$ then output d.

(3) Let $\lambda = s(y_p - y_q)$ if $x_p \neq x_q$ and $\lambda = s(3x_p^2 + a)$ if $x_p = x_q$. Output $R = (\lambda^2 - x_p - x_q : \lambda(\lambda^2 - 2x_p - x_q) + y_p : 1)$.

Remark 1 *If $n = p$ is prime, the set $E[\mathbb{Z}/p\mathbb{Z}]$ forms an Abelian group for any elliptic curve $E = E_{a,b}$, with unit element O. In this case, the partial addition algorithm, which will now always produce a sum of two points on E, is equivalent to the usual addition algorithm.*

Moreover, it can be shown that for a prime divisor p of arbitrary n coprime to 6, the sum R produced by the partial addition algorithm for any two points P, Q on an elliptic curve $E_{a,b}$ modulo n, has the property that R_p (obtained by reducing the coordinates of R modulo p) is the sum of (the similarly defined) points P_p and Q_p in the group $E_{\bar{a},\bar{b}}[\mathbb{Z}/p\mathbb{Z}]$, where $\bar{a} \equiv a \bmod p$, and $\bar{b} \equiv b \bmod p$.

Using the partial addition algorithm repeatedly, it is of course possible to obtain a partial multiplication algorithm, which computes either $k \cdot P$ or finds a divisor of n, for any positive integer k, given any $P \in V$ and any a as before. However, there are various ways to speed up this computation of $k \cdot P$, using partial doubling, and the fact that it is not necessary to keep track of the y-coordinate.

In the next sections we will occasionally be sloppy, and write about the sum and multiples of points on elliptic curves modulo n; we mean the result of application of the partial addition and multiplication algorithms, which in exceptional cases means that a divisor of n is found, rather than a point.

3 Elliptic Curve Primality Proving

We give an outline of the Elliptic Curve Primality Proving algorithm; some of the necessary definitions and details will be given in the subsequent sections. The algorithm is based on the following theorem.

Theorem 2 *Let* $n_0 \in \mathbb{N}$ *with* $\gcd(6, n_0) = 1$. *Let* E *be an elliptic curve modulo* n_0, *and let* $m, n_1 \in \mathbb{N}$ *with* $n_1 \mid m$. *Suppose that for every prime factor* q *of* n_1 *there exists* $P \in E$ *such that* $mP = 0_E$ *and* $\frac{m}{q}P \neq 0_E$. *Then for all prime factors* p *of* n_0 *holds* $\#E[\mathbb{Z}/p\mathbb{Z}] \equiv 0 \bmod n_1$.

Corollary 3 *Suppose that the hypotheses of Theorem 2 are satisfied. Then:*

$$n_1 > (n_0^{\frac{1}{4}} + 1)^2 \quad \Rightarrow \quad n_0 \textit{ is prime.}$$

Note that the requirement is that n_1 exceeds a bound slightly larger than $\sqrt{n_0}$.

Essential in the proof of the Corollary is the Theorem of Hasse, stating that the number of points on any elliptic curve modulo a prime p equals $p+1-t$ for some integer t with $|t| \leq 2\sqrt{p}$. Theorem 2 easily follows from the observation that, modulo any prime divisor p of n_0 the conditions imply that $\#E[\mathbb{Z}/p\mathbb{Z}]$ can not be a proper divisor of n_1.

Starting point for the application of *ECPP*, will always be a probable prime $n_0 = n$; it is assumed that n will be free of small prime factors (in particular 2 and 3), and that n has passed certain compositeness tests (of Miller-Rabin type). This will make it very likely that n is indeed prime; the objective is to prove that.

Given such an integer n, the basic Elliptic Curve Primality Proving algorithm proceeds roughly in these three stages:

(D) starting with $n_0 = n$, find a sequence of probable primes $n_0, n_1, \ldots, n_k$, such that n_{i+1} divides the order of some elliptic curve modulo n_i, such that $n_{i+1} > (\sqrt[4]{n_i} + 1)^2$, and such that n_k is so small that primality can be verified by easy inspection (or trial division).

(F) For each of the integers n_i with $i = 0, 1, \ldots, k-1$, construct an elliptic curve E_i of order a multiple of n_{i+1} modulo n_i, together with a point P_i of order n_{i+1} on the curve modulo n_i.

(P) Verify that the conditions of Theorem 2 hold for the given probable primes n_i, curves E_i and points P_i, for $i = k-1, k-2, \ldots, 0$.

The original idea came from Goldwasser and Kilian, who designed such an algorithm, which uses random elliptic curves over the integers n_i, computes the order of them and factors the orders. Computing the order of a random elliptic curve over n_i is very cumbersome. It is yet a faster way to determine the curve order first and construct a curve with such order. Besides, we get two elliptic curves for each integers, that increases the possibility of success. m order has to be selected from the algebraic integer of an imaginary quadratic field $\mathbb{Q}(\sqrt{D})$. D is a *negative fundamental discriminant*, and as such, it has certain properties: $D \equiv 0 \pmod 4$, or $D \equiv 1 \pmod 4$, for every k (> 1) D/k^2 is not a fundamental discriminant, $D \leq 0$ (from practical point of view we use $D \leq -7$). Moreover D must be appropriate for n, which means: $(D|n) = 1$, where $(D|n)$ is the Jacobi symbol and there exist such $x, y \in \mathbb{Z}$ for which

$$4n = (2x + yD)^2 - y^2 D. \tag{1}$$

An appropriate D provides two possible orders: $m = |\nu \pm 1|^2$, where

$$\nu = x + y\frac{D + \sqrt{D}}{2}.$$

If (1) is valid, with the help of an x_0 root of the *Hilbert polynomial* $\pmod n$ we get two elliptic curves with order $m = |\nu \pm 1|^2$. Refer to [2]

We will refer to this algorithm as *ECPP* in the rest of the paper.

3.1 Downrun

The *ECPP* algorithm consists of two parts. The first part of the algorithm will be called recursively with input n_i. The main objective is to find n_{i+1}. This is what happens at level i:

(D) select a pair D, m of negative discriminant D and integer $m = m_{i+1}$ such that m is the product of small primes and a probable prime n_{i+1} that exceeds $(\sqrt[4]{n_i} + 1)^2$.

In practice this is what happens:

(D_0) Prepare a list of primes up to some bound $s = s_i$, as well as list of negative fundamental discriminants up to a bound $d = d_i$ that factor completely in a product of primes from the prime list, together with their full prime factorization.

(D_1) Select one discriminant D in the list, find the reduction of the binary quadratic form $Ax^2 + Bxy + Cy^2$ of discriminant D, where $A = n$, $B^2 \equiv$

$-D \bmod n$, and $C = (B^2 + D)/(4n)$. This requires the modular square root of $-D$ modulo n, which is obtained as a product of the square roots of the prime factors of $-D$. If this provides ν with $|\nu|^2 = n$, then $m_1 = |\nu - 1|^2$, $m_2 = |\nu + 1|^2$.

(D_2) From the two pairs D, m_1, D, m_2 found in the previous step, for which a probably prime q_1 dividing $m_1 = |\nu - 1|^2$ can be found such that m_1/q_1 is the product of small primes only, similarly for m_2, one is selected.

(D_3) Let n_{i+1} be the probable prime $q = q_1$ or q_2, for which $m = m_1$ or m_2, according to the selection in the previous step, m/q is the product of small primes, if that satisfies the conditions, otherwise select another D from the list.

Several comments are in order.

Usually a 'master-list' of primes up to some bound B is prepared in advance; the bound s_i (and hence the list) in step (D_0) may depend on i (the level of the recursion arrived at), but should be at most B. Similarly for the list of discriminants, and the bound d_i. This means that step (D_0) will mainly consist of the selection of sub-lists, from precompiled lists that are computed once for all n up to a fixed size N. In Step (D_2) the probable factorization of possible curve order m_i has to be found; one uses a smoothness bound $b = b_i$, that is, all prime factors smaller than b are removed (and considered small).

Note that backtracking may be necessary: it is possible that at some level no D provides a new n_i!

The output of the first phase of the algorithm will consist of a triple (n_i, D_i, m_i) for $i = 0$ to $i = k - 1$ such that m_i is the product of small primes and a probable prime n_{i+1} that exceeds $(\sqrt[4]{n_i} + 1)^2$.

3.2 Finding elliptic curves

The second phase is executed after the recursive call of the first part, which results in a list of n_i, D_i, m_i such triples, where n_i is the input of the recursive step produces n_{i+1}. This phase in the primality testing algorithm can be done as follows. Again, we describe the steps to be taken at level i.

(F) Find elliptic curves E_i and points P_i on $E_i[\mathbb{Z}/n_{i-1}\mathbb{Z}]$ with the property that if n_{i-1} is prime, then the order of P_i is n_i.

This is done as follows.

(F_0) Compute an auxiliary polynomial $G_i \in \mathbb{Z}[x]$; see the comments below.

[F$_1$) Find a root j_i of G_i mod n_{i-1} in $\mathbb{Z}/n_{i-1}\mathbb{Z}$, as well as an integer t_i such that the Jacobi symbol $\left(\frac{t_i}{n_{i-1}}\right)$ equals -1.

(F$_2$) Define elliptic curves E_i' and E_i'' by

$$E_i' : \quad y^2 = x^3 + 3kx + 2k$$

and

$$E_i'' : \quad y^2 = x^3 + 3kt_i^2 x + 2kt_i^3,$$

where $k = \dfrac{j}{1728 - j}$.

(F$_3$) Find (for example by randomly choosing) a point on E_i' or E_i'' that has order n_i, if n_{i-1} is prime.

Remark 4 *The auxiliary polynomial* G_i *is the Hilbert (or Weber) polynomial or a variant of this. The two elliptic curves are the twists of the elliptic curve with* j*-invariant* j_i*. We refer to [2] and [8] for more details, as this part of the algorithm plays no major role in what follows.*

4 Magma

Magma [4] is a large computer algebra system, with high-performance computations in number theory as one of its specializations, including very advanced integer factorization and primality proving algorithms. Since its launch (London, August 1993) a large body of intrinsic functions (implemented in the C language), have been supplemented by packages developed on top of this, making use of the Pascal-like user language and the programming environment that is provided.

4.1 Magma-ECPP

Magma has a built-in primality test, which uses a combination of the Miller-Rabin compositness test, and *ECPP*. It can be invoked by

```
IsPrime(n: parameter) : RngIntElt ↦ BoolElt
```

function. By default, this function proves primality using *ECPP* (after a quick test to throw out composites), but it is possible to set the optional Boolean parameter `Proof` to FALSE, in which case the function only uses the probabilistic

Miller-Rabin test, with the default number of bases (20). In the rest of the paper we refer to the function in Magma v2.19 (December 2012) as Magma-ECPP. We would like to compare our *ECPP* implementation, Modified-ECPP to this function.

This function is based on F. Morain's implementation of *ECPP* in the C language, and works more or less according to the description above as one can tell from the verbose printing (although the details of the source code are hidden from the user). It has a list of base discriminants in a file, likely fully factored and ordered by the value of their field h, where it loops through in each iteration. In the ith iteration it first applies a trial division sieve on n_i with bound `Pmax`, then on each discriminant, it performs (D) until it either finds n_{i+1} or runs out of discriminants. In the case of success it goes into the $(i+1)$th iteration with the new input; in the second case it loops through the same set of discriminants, but applies stronger factoring methods to each of the numbers m_i. It applies trial division, Pollard's ρ and Pollard's $p-1$ in the first round and if (D) is not successful, supposedly ECM is used too. If the second round still yields no n_{i+1}, it has to backtrack to input n_{i-1}. As after each iteration the information on which discriminants it succeeded is stored, on backtracking it starts from the next discriminant in the list.

4.2 Modified-ECPP

In the Modified-ECPP version, one does not abort executing the ith iteration as soon as a good discriminant is found, but only after a certain range of discriminants are scanned, thus we could gain a *range* of $n_{i+1,j}$'s instead of a single one in each (recursive) call of step (D), from which the input n_{i+1} for the next call is to be selected. Choosing one n_{i+1} out of the range of possibilities can be done based on criteria depending on certain properties of the numbers.

4.2.1 Theoretical observations

The following observation plays an important role in the running time analysis: if we are able to find all prime factors of curve cardinalities m_{i+1} up to a bound b_i in the ith iteration, the probability that one such curve cardinality m_{i+1} leads to a new node will be the probability that the second largest prime factor of m_{i+1} is less then b_i; this is approximately

$$e^{\gamma}\frac{\log b_i}{\log n_i}.$$

It is then reasonable to suppose that, if we have e_i such curve orders m_{i+1}, the number of new nodes has a probability distribution with average approximately

$$\lambda_i = e^{\gamma} \frac{\log b_i}{\log n_i} e_i.$$

Refer to [7].

For each negative discriminant $D \leq -7$ the probability of success is

$$\frac{1}{2h(D)},$$

where $h(D)$ is the ideal class number of $\mathbb{Q}(\sqrt{D})$. As each successful case results in 2 m_{i+1}'s, we expect

$$e_i \approx \sum_D \frac{1}{h(D)}.$$

Refer to [2].

4.2.2 The tree structure

The process of *ECPP* may be envisaged as choosing a path through a (directed) tree of which the nodes represent probable primes. (Note that strictly speaking the graph is not a tree, as it is possible, but not really likely that different paths lead to the same node!) The root of this tree is n_0, the leaves correspond to probable primes that are small enough to be recognized as primes by some direct method. The aim is to find a relatively short and "easy" path from the root n_0 to a leaf n_k as fast as possible; in particular, one would like to avoid computing too many nodes explicitly.

By the latter we mean that we store certain information with the nodes that we compute explicitly: for possible descendants $n_{i,j}$, $j = 1, \ldots, t_i$ of n_i, we store some information to base our choice of n_{i+1} on it. (For the rest of the paper with n_{i+1} we mean the selected $n_{i,j}$.) This includes the value $n_{i,j}$, as well as $s_{i,j}$, $d_{i,j}$ and $b_{i,j}$, (respectively: the smoothness bound for the discriminants, the bound on the size of the discriminants and the smoothness bound on the curve order), the level i in the tree and a parameter measuring the *suitability*. The choice of node n_{i+1} is based on this *suitability* parameter, which is determined when the node is produced. The value of the *suitability* is initialized to the value of the field $d_{i,j}$ stored with the nodes. Backtracking is unfavorable as, besides the useless work decreasing the level in the tree, the size of the primes in the nodes is likely to increase, therefore there is a fixed, global penalty value

p added to the *suitability* of each node when a new level starts up, except for the nodes of the new level.

The idea is that the computations for smaller discriminants will be cheaper, and hence the algorithm will be completed faster. The field $d_{i,j}$ that gives the initial value of *suitability* of the node, is determined based on the function λ, searching for the minimal power of $d_{i,j} = \log(n_{i,j})^{\delta_{i,j}}$ allowing the value of λ to exceed a certain bound given $s_{i,j}$, $b_{i,j}$. The power $\delta_{i,j}$ is stored as the initial value of *suitability.* The nodes are stored in an array, and a certain penalty p is added, if necessary. The value of p depends on how hard we want to punish backtrack steps. Order by *suitability* and select the smallest one and call algorithm (T) recursively with that node as input. If (T) is successful the new nodes are added to the array and the sorting process starts again. If there is no new node found the value of *suitability* and the field $d_{i,j}$ of the selected node is increased. The power $d_{i,j}$ can reach a given bound where the node falls out from the array of possible nodes. The nodes are reordered. Repeat this procedure, until the size of the nodes reaches a limit which is small enough to recognize the prime.

4.2.3 The path finding algorithm

The path finding algorithm (T) in the 'tree' then has three main stages:

(T_0) Step (D) is being applied for n_0 looking for the minimal choice of $d(n_0) = \log(n_0)^{\delta_0}$ for which D is successful with a kind of brute force strategy, using a loop in which the value of d_0 is incremented until there exists at least one descendant $n_{1,j}$, as the next step requires a non-empty list of $n_{1,j}$'s. This part is called just once at the beginning.

The next steps are repeated for $i = 1, 2, \ldots, k$, where n_k is the first probable prime that can be proven prime directly.

(T_1) Keeping the value of $\lambda = \lambda_{i,j}$ above 1.5, determine the value of $d_{i,j}$ for given $s_{i,j}$, $b_{i,j}$ for each newly found $n_{i,j}$, and store a list of (at most 100 of) the best values according to *suitability.* Sort the list of the best hundred nodes and select the best as n_{i+1}.

(T_2) Apply Step (D) again on n_{i+1}, with the parameter $d_{i+1} = \log^{\delta_{i+1}}(n_{i+1})$; if no new node is found increase the δ_{i+1} by 0.1, repeat ordering and selection until at least one new $n_{i+1,j}$ is found. Once Step (D) is successful, go back to (T_1) with the new list of nodes as input.

5 Experiments

In this section we are going to point out some typical situations that occur during the process of *ECPP* and that are handled differently by Magma-ECPP and Modified-ECPP. We describe the effects of these differences with respect to outcome and running time in one particular example.

We ran Magma-ECPP and Modified-ECPP on a probable prime with more than one thousand digits and show the output that describes the first three steps of the process. We present the number of digits and the first 10 and last 10 digits of big numbers occuring during the process. Describing the whole process or presenting whole numbers in the process for probable primes of this size would be far too space-consuming, as it usually consists of more than a hundred iterations and contains numbers in the size of the input probable prime. The output of the whole process for both implementations and the proof provided by Modified-ECPP can be found on one of the authors' homepage. Follow the links: `http://www.math.ru.nl/~kiss/Modified-ECPP_Proof.pdf`, `http://www.math.ru.nl/~kiss/Modified-ECPP_Output.pdf` and `http://www.math.ru.nl/~kiss/Magma-ECPP_Output.pdf`. For more big probable primes (coming from the generalized Pascal triangles), tested by Modified-ECPP refer to [6]. Note that G. Farkas and G. Kallós already have dealt with such numbers and tested them with the *ECPP*. Refer to [5] about the details.

The test was running in Magma v2.19 on a machine with 8001 Mb RAM and eight 2.5 GHz Intel Xeon processors.

Modified-ECPP is currently using different variants of trial divisions to factor the curve orders. With bound $t = 1000$ it applies normal trial division, and after that a batch trial division with bound b_i working on a list of $m_{i,j}$'s instead of factoring them one by one. The value of the penalty p is 0.8.

On an iteration we mean in the case of Magma-ECPP that we run (D) on one discriminant and in the case of Modified-ECPP that we run (D) on a set of discriminants up to $\log^{d_i}(n_i)$ (starting from a previously reached limit, or from the beginning of the discriminant list). On one step in both cases we mean a sequence of iterations, which either results in at least one new node, or runs out of discriminants and has to backtrack to another node. Going back to the same node is not considered as backtracking. In the implementation of Modified-ECPP one step also contains (T1) running on the new nodes, if there is any.

5.1 Example

The input number

```
n= 8565190451...2658848547
```

was used. This number has 1015 digits.

5.1.1 Magma-ECPP

In the first step Magma-ECPP needs 58 iterations to provide a new node and uses trial division to factor m_1.

```
% Pmax=4000000
% N_0=8565190451...683952658848547 1015 digits
% next D is 0 at 1.950000s
% next D is 7 at 10.180000s
...
% next D is 14683 at 495.940000s
% next D is 14083 at 513.280000s
% Cofactor after sieve is a probable prime
% D[[0]]=14083
%
% End of depth 0 at 513.990000 s
```

In the second step, Magma-ECPP fails to find an n_2 after running through a fixed list of discriminants twice, supposedly using trial division, Pollard's $p-1$ and Pollard's ρ in the first turn, and other, possibly harder and more time consuming, methods in the second, thus backtracking to n_0.

```
% Pmax=4000000
% N_1=8985609131...1613020571 1009 digits
% next D is 0 at 514.760000s
% next D is 7 at 522.780000s
...
% next D is 991 at 1140.930000s
% next D is 19963 at 1151.780000s
%!% No next D
%!% Forced to retry, snif...
% next D is 0 at 1163.620000s
% next D is 7 at 1184.430000s
...
% next D is 991 at 2754.460000s
% next D is 19963 at 2778.880000s
%!% No next D
%!% One redo was not enough...
% Backtracking
% End of depth -1 at 2803.580000 s
```

In the third step Magma-ECPP, as it backtracked in the previous step, uses n_0 again and starts from the first discriminant that was not used in the first step. It does not find a new n_1 just using trial division on the rest of the discriminants, so it applies stronger factorization methods running through the same set of discriminants from the beginning. As there is only one successful discriminant in the set, it finds the n_1 produced also by the first step. Therefore it gets to an endless loop finding the same n_1 and backtracking to n_0 again.

```
% Pmax=4000000
% N_0=8565190451...683952658848547 1015 digits
% next D is 19963 at 2805.250000s
% next D is 2339 at 2814.730000s
%!% No next D
%!% Forced to retry, snif...
% next D is 0 at 2825.470000s
% next D is 7 at 2847.600000s
...
% next D is 14683 at 4084.320000s
% next D is 14083 at 4114.750000s
% Cofactor after sieve is a probable prime
% D[[0]]=14083
% End of depth 0 at 4115.470000 s
```

Magma-ECPP needs 58 iterations to produce a new node in the first step and has to backtrack after the second step, and as it does not find another node where it could continue, gets stuck in an endless loop.

5.1.2 Modified-ECPP

By default, Modified-ECPP uses parameters $b_i = \log^2(n_i)$ and $s_i = \log(n_i)$, where b_i is the bound to the primes used to factor the m_i-s and s_i is the bound to primes used to factor the discriminants (refer to section 4.2.2). This configuration found a path to the leaves in 1246.99 seconds after 139 steps and it finished the proof using this path in 977.65 seconds; thus the total time was 2224.64 seconds.

Below we provide more information on the process in a configuration where we put $s_i = \log^{1.3}(n_i)$, as the first configuration does not find the node where Magma-ECPP gets stuck (because limit s_i is too low to factor 14083, a prime in fact), and it is therefore complicated to compare the default case with the Magma-ECPP output.

In the first step, Modified-ECPP provides two $n_{1,j}$-s for $n_0 = n$ after 4 iterations. The discriminant bound $d_0 = \log^{\delta_0}(n_0)$ is increased after each iteration, moving up from $\delta_0 = 1$ to 1.3, where it finds two appropriate nodes, $n_{1,1}$ and $n_{1,2}$. It initiates bound $\delta = 1.2$ for both of them. $n_{1,2}$ is the same number that Magma-ECPP gets stuck on and as it is smaller than $n_{1,1}$, it is selected at the end of this step.

```
SLimit:  23946.87 1.3
BLimit:  5461424.06 2.0

n0 8565190451...683952658848547 1015 digits
DLimit, delta:  2336.97, 1.00
Filtering discriminants and reduction takes 7.700 seconds for
711 D-s, where 56 was succesful On average that is 0.011
The time for 56 trial divisions is 0.000 seconds, 0.000 on average
Batch trial division takes 3.270 seconds for 56 D, 0.058 on average
Time of Miller-Rabin test for 55 is 2.280 seconds, 0.041 on average
This resulted 0 new nodes
...
delta: 1.30
Filtering discriminants and reduction takes 33.380 seconds for
3930 D-s, where 50 was succesful On average that is 0.009
The time for 50 trial divisions is 0.020 seconds, 0.000 on average
Batch trial division takes 3.220 seconds for 50 D, 0.064 on average
Time of Miller-Rabin test for 46 is 3.580 seconds, 0.078 on average
This resulted 2 new nodes

lambda 1.93
delta 1.20
The total time of estimation is  3.140
lambda 1.49
delta 1.20
The total time of estimation is  3.050

1 level completed in 97.930 seconds
```

In the second step Modified-ECPP neither provides new node in one iteration, with bound $\delta_1 = 1.2$, therefore the *suitability* values had to be reevaluated and the other child is selected, as after the failure of the iteration the *suitability* of $n_{1,2}$ is increased to 1.3.

```
n1:  898560913...903551613020571 1009 digits
```

```
SLimit:  23763.64 1.3
DLimit:  10947.23 1.2
BLimit:  5397264.7359025146158857635688 5 2.0
Filtering discriminants and reduction takes 27.250 seconds for
3331 D-s, where 130 was succesful On average that is 0.008
The time for 130 trial divisions is 0.030 seconds, 0.000 on average
Batch trial division takes 3.410 seconds for 130 D, 0.026 on average
Time of Miller-Rabin test for 127 is 4.960 seconds, 0.039 on average
This resulted 0 new nodes

2 level completed in 36.790 seconds
```

It has more luck with $n_{1,1}$; in the third step provides four new nodes and initiates $\delta_{2,j}, 1 \leq j \leq 4$ for them.

```
n1:  1928293492...0513347699 1010 digits
SLimit:  23773.79 1.3
DLimit:  10951.55 1.2
BLimit:  5400813.29 2
Filtering discriminants and reduction takes 28.930 seconds for
3333 D-s, where 166 was succesful On average that is 0.009
The time for 166 trial divisions is 0.040 seconds, 0.000 on average
Batch trial division takes 3.480 seconds for 166 D, 0.021 on average
Time of Miller-Rabin test for 153 is 9.190 seconds, 0.060 on average
This resulted 4 new nodes

lambda 1.95
delta 1.20
The time of estimation is  3.110
...
lambda 2.06
delta 1.20
The time of estimation is  3.140

3 level completed in 51.070
```

Modified-ECPP provides two new nodes after 4 iterations in the first step. Note that it does not apply any heuristics in (T_0). Then it picks up the same node $n_{1,2}$ where Magma-ECPP got stuck and also has bad luck with this number, as it produces no new node after one iteration. Therefore it had to backtrack to $n_{1,1}$. In this case λ does not help us very much as it underestimated the necessary amount of discriminants to provide a new node, but as there is another node produced on the same level, Modified-ECPP does not even have to go back to the previous level to provide a new node. In the third step it provides four new nodes in one iteration; λ seems to work better there. The algorithm reached the small primes in 1794.97 seconds after 146 steps. Completing the proof using this path took 1119.23 seconds, and thus the total running time was 2914.2 seconds.

Note that one iteration for Modified-ECPP takes a list of discriminants, this way the 4 iterations from the first step were running through discriminants up to 23946.

The first step took 513.99 seconds for Magma-ECPP and 97.93 seconds for Modified-ECPP. The difference probably comes from the fact that Magma-ECPP has to force the m_i-s with strong factoring methods and big factoring bounds to provide a new node while Modified-ECPP works with a much bigger set of discriminants (up to 10^{10}) and therefore it can use batch trial division and lower factoring bounds. In this case the bounds on lower levels in the tree are around 5300000. In the third step Modified-ECPP provides 4 new nodes with one iteration, where the predictions have been already applied on the value of D_1. The goal of the estimation of the initial value of δ_i using λ is to decrease the probability of backtracking and selecting the same node more often by predicting the minimal interval where one iteration will run successfully. This gives also the basis of the *suitability* value to predict on which node do we have to process the least number of discriminants to provide at least one new node. The total number of iterations in this case is 187 where 146 provides a new node. The other iterations provided the same node as the input or provided another node to backtrack. We do not count the four iterations of the first step in the number of the iterations, as there is no heuristics applied.

6 Remarks and conclusions

We described a strategy applied during the process of *ECPP*; produce more nodes in a step in the recursion, estimate the *suitability* of the new nodes and select the most suitable. With this strategy we try to avoid backtracking (or repetition on the same node) by estimating the interval for discriminants to search through for each new node and by predicting which $n_{i,j}$ to pick in order to be able to successfully continue. 78% of the iterations ran in the test terminated successfully on the nodes and intervals selected this way.

As we can see, the predictions give us no hundred percent certainty to avoid such situation; for instance, in the second step from the example $n_{1,2}$ provided no new node, but at least we have an idea, what we can expect in theory from considering a node. Roughly speaking we could not avoid backtracking or repetition after 22% of the iterations in this case. What to do then?

First of all we are trying to avoid backtracking to the previous level 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. There we can use the penalty value, that is added to the nodes in the previous levels' *suitability*, influencing the probability of selecting a node from previous levels. The default value of the penalty is high, 0.8. After one unsuccessful iteration we increase the value *suitability* with 0.1, and thus 0.8 would mean 8 unsuccessful iterations on the node; this is a tough condition, does not occur frequently. Of course different nodes starts from different *suitability*, thus in practice we do not always need 8 unsuccessful iterations; if none of the nodes on the current level is suitable enough, the numbers are just to get a feeling about the size of the penalty. In this example backtracking to the previous level does not occur.

The goal of producing more nodes in a step is to make the implementation more robust against backtracking to the previous level, as the nodes in the current level are likely more suitable, and to avoid running multiple iteration on the same node by switching between the nodes on the same level. In addition, if it turns out that a node is not as successful as estimated, and thus produces no new node after the first iteration on the expected interval, we become more careful and try to take small steps at a time. We increase the value $\delta_{i,j}$ of the current node $n_{i,j}$ with 0.1 after each unsuccessful iteration and reorder the list of possible nodes to see whether our selected node is still the best. This way it tries to keep backtracking fast and flexible. That is what happens in the second step in the example.

Modified-ECPP was tested with several numbers with around 1000 digits and it provided proof in each situation in 2000–3000 seconds. Magma-ECPP was running on the same numbers failed on the number from the example, 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 for numbers with similar size.

The implementation works on a list of preprocessed discriminants up to 10^{10}, thus it can process the discriminants fast, without the need of factoring the discriminants or the computed m_i-s with high factoring bounds. By default Modified-ECPP runs with configuration $s_{i,j} = \log(n_{i,j})$ and $b_{i,j} = \log^2(n_{i,j})$, which for numbers with such size are around 2 300 and 5 300 000, and in our tests always terminated successfully.

As it still has to backtrack in a number of situations, there are further plans to tune the strategy to reduce this number by involving all the important parameters $s_{i,j}, b_{i,j}, d_{i,j}$ (refer to Section 4.2.2) to determine *suitability.* Also the implementation is not the final version, stronger factoring methods are going to be applied on the $m_{i,j}$-s in order to gain shorter paths and to be able to prove primality for bigger numbers.

Acknowledgements

We are greatly indebted to Prof. Wieb Bosma for his scientific consultations.

References

[1] M. Agrawal, N. Kayal, N. Saxena, PRIMES is in P, *Annals Math.* **160,** 2 (2004) 781–793. ⇒35

[2] A. O. L. Atkin, F. Morain, Elliptic curves and primality proving *Math. Comp.* **61,** 203 (1993) 29–68. ⇒36, 39, 41, 43

[3] D. Bernstein, Proving primality in essentially quartic random time, *Math. Comp.* **76,** 257 (2007) 389–403. ⇒35

[4] W. Bosma, J. Cannon, C. Playoust, The Magma algebra system. I. The user language, *J. Symbolic Comput.* **24,** 3-4 (1997) 235–265. ⇒41

[5] G. Farkas, G. Kallós, Prime numbers in generalized Pascal triangles, *Acta Tech. Jaur.* **1,** 1 (2008) 109–118. ⇒45

[6] G. Farkas, G. Kallós, G. Kiss, Large primes in generalized Pascal triangles, *Acta Univ. Sapientiae, Inform.* **3,** 2 (2011) 158–171. ⇒45

[7] J. L. Hafner, K. S. McCurley, On the Distribution of Running Times of Certain Integer Factoring Algorithms, *J. Algorithms* **10,** 4 (1989) 531–556. ⇒43

[8] A. K. Lenstra, H. W. Lenstra Jr., Algorithms in number theory in: *Handbook of Theoretical Computer Science* (vol. A) 1990, Elsevier, Amsterdam, pp. 673–715. ⇒36, 41

Received: March 10, 2013 • Revised: May 31, 2013