Complexity of factoring polynomials with rational number coefficients

Mark van Hoeij
Florida State University

JA'2007 Edinburgh

July 6, 2007

Papers discussed in this talk

- [Zassenhaus 1969]. Algorithm that is usually very fast, but can take exponential time for certain types of polynomial.
- [LLL 1982]. Lattice reduction (LLL algorithm) = key tool for solving combinatorial problems.
- [LLL 1982]. First polynomial-time factoring algorithm, though Zassenhaus is usually faster.
- [vH 2002]. New algorithm, outperforms prior algorithms on all tests, but no complexity bound is given.
- [Belabas 2004] Gave the best-tuned version of [vH 2002].
- [Belabas, vH, Klüners, Steel 2004] (in the JA'2007 notes). Gave poly-time bound for the slowest version of [vH 2002], however, gave a worse bound for the best tuned version!
- [vH and Andrew Novocin, 2007] An asymptotically sharp bound for the fastest version.

Zassenhaus' algorithm

Let $f \in \mathbb{Z}[x]$ separable and monic. **Goal:** the factors of f in $\mathbb{Z}[x]$.

Idea 1: If $g \in \mathbb{Z}[x]$ divides f then the coefficients of g are smaller than some **bound** L that we can compute.

Idea 2: If $g \in \mathbb{Z}[x]$ divides f then g can be **reconstructed** when g mod p^a is known for some $p^a > 2L$.

Idea 3: Factor $f = f_1 \cdots f_r$ over $\mathbb{Z}_p$ (the p-adic integers). There are only **finitely many** monic factors of f in $\mathbb{Z}_p[x]$. Each is of the form

$$g_v := \prod f_i^{v_i}$$

for some 0–1 vector $v = (v_1, \ldots, v_r)$.

Idea 4: $f_1, \ldots, f_r$ (and hence g_v) are not known exactly, but are only known mod p^a. That's enough using idea 2.

Features of Zassenhaus' algorithm

Let L = bound for coefficients of factors in $\mathbb{Z}[x]$.
Let $f_1, \ldots, f_r \in \mathbb{Z}_p[x]$ be the p-adic factors.
Compute the p-adic factors mod p^a for some $p^a > 2L$
(first compute the f_i mod p, and then mod p^a by Hensel lifting).

1. Given some 0–1 vector $v \in \{0,1\}^r$ then one can rapidly decide if $g_v := \prod f_i^{v_i}$ is in $\mathbb{Z}[x]$ or not.
2. A factor in $\mathbb{Z}[x]$ can be computed efficiently if its 0–1 vector v is known: Take the f_i with $v_i = 1$ and multiply them mod p^a.

If f is irreducible we end up trying 2^r (actually 2^{r-1}) cases. Then the CPU time will be roughly:

Cost(factoring f mod p) $+$ Cost(Hensel lifting) $+$ $2^r \cdot$tiny.

Complexity of Zassenhaus' algorithm

Cost(factoring f mod p) $+$ Cost(Hensel lifting) $+$ $2^r \cdot$tiny

1. Cost(factoring mod p) depends polynomially on the degree N.
2. Cost(Hensel lifting) depends polynomially on N, $\log(\| f \|_\infty)$ where $\| f \|_\infty =$ largest absolute value of coefficients of f.
3. With some tricks, testing one $v \in \{0, 1\}^r$ usually takes only a tiny amount of CPU time, regardless N and $\log(\| f \|_\infty)$

Given some polynomial $f \in \mathbb{Z}[x]$ of degree N, the algorithm tries several primes p, and then chooses the one for which f has the fewest p-adic factors $f_1 \cdots f_r$.

Usually $r << N$ and Zassenhaus' algorithm is fast, with Hensel lifting dominating the CPU time. But for polynomials that have large r at each p the algorithm suddenly takes exponential time.

Timings on an example

Cost $\leqslant$ Polynomial(N, $\log(\| f \|_\infty)$) $+$ 2^r·tiny

Suppose for example f has degree $N \approx 200$, and each coefficient has about 200 digits.

For the best implementations of Zassenhaus' algorithm, as long as $r < 20$ then the precise value of r has little impact on the CPU time, it will take about a second either way. Make examples with larger r, and the CPU time suddenly starts to go up exponentially.

Zassenhaus' algorithm is usually much faster than [LLL 1982] (for such N, H one second instead of a day, if $r < 20$).

However, if say $r = 64$ then [LLL 1982] is much faster (a day instead of an estimated 100,000 years for Zassenhaus).

The goal

Suppose f has degree $N \approx 200$, with ≈ 200 digit coefficients, and say $r = 64$ p-adic factors $f = f_1 \cdots f_{64}$.

For such a polynomials [LLL 1982] takes about 1 day.

Although that is much better than Zassenhaus, keep in mind that if we somehow knew which subset(s) of $f_1, \ldots, f_{64}$ to take, then Zassenhaus would only take 1 second which is much better than 1 day!

Thus, the only thing that stands in the way to reduce CPU time from 1 day to 1 second are objects with *only 64 bits of data* (namely the $v \in \{0, 1\}^r$ that encode the right subsets of $f_1, \ldots, f_r$).

The goal in [vH 2002] is a quick way to compute this data.

LLL

In [LLL 1982] Lenstra, Lenstra and Lovász gave a lattice reduction algorithm (the LLL algorithm), as well as a polynomial time factoring algorithm for $\mathbb{Q}[x]$ based on the LLL algorithm.

Suppose $L \subseteq \mathbb{Z}^n$ is a $\mathbb{Z}$-module.

The input of the LLL algorithm is an arbitrary basis of L.

The output is a new basis $b_1, \ldots, b_m$ of the same lattice L, but this basis has some very useful properties.

LLL separates short from long vectors if gap is big enough

Let $n = \dim(L)$ and let B be some positive number. Let L_B be the sublattice of L spanned by the B-short vectors

$$L_B := \mathrm{SPAN}\{v \in L \ : \| v \| \leqslant B\}$$

Suppose furthermore that all vectors outside of L_B are sufficiently much longer than B, i.e. suppose

$$\text{Big Gap Condition :} \quad \| v \| > 2^{\frac{n}{2}} B \ \text{ for all } v \in L \setminus L_B.$$

Then LLL allows us to compute a basis for L_B
(compute an LLL basis $b_1, \ldots, b_n$ for L, and as long as the Gram-Schmidt length of the last vector is $> B$ remove it).

If the Big Gap Condition does not hold, then instead of a basis of L_B we would get a basis of some lattice L' for which $L_B \subseteq L' \subseteq L$.

Factoring with LLL

Suppose $f \in \mathbb{Z}[x]$ has a non-trivial factor $g = c_0 + c_1 x + \cdots \in \mathbb{Z}[x]$. How to find g with LLL?

Idea: Construct a lattice L with these properties:

1. $w := (c_0, c_1, \ldots) \in L$.
2. Big Gap Condition: All vectors $\notin \mathrm{SPAN}(\{w\})$ are sufficiently much longer than than w.

Then compute an LLL reduced basis $b_1, \ldots, b_m$ of L, and find $w = \pm b_1$. Read off g from w.

This way one can find a factor g (or prove f is irreducible) in polynomial time, see [LLL 1982].

Back to the example

Suppose f has degree $N \approx 200$, with ≈ 200 digit coefficients, and say $r = 64$ p-adic factors $f = f_1 \cdots f_{64}$.

To construct an irreducible factor $g \in \mathbb{Z}[x]$ (worst case: $g = f$ if f is irreducible) with [LLL 1982] means finding $w = \text{vector}(g)$ with lattice reduction. This vector could contain as much as $200 \cdot \log_2 10^{200} \approx 132,000$ bits of data, and LLL could take a day.

However, if we had $r = 64$ bits of data, $v = (v_1, \ldots, v_r) \in \{0, 1\}^r$ then we could compute the corresponding factor

$$g = \prod f_i^{v_i}$$

in 1 second.

Main idea in [vH 2002]: Use LLL to compute $(v_1, \ldots, v_r)$ in a way that avoids computing any bits of information about the coefficients of g.

van Hoeij, 2002

Let $f = f_1 \cdots f_r \in Z_p[x]$. The map

$$v \mapsto g_v = \prod f_i^{v_i}$$

that sends a 0–1 vector $v = (v_1, \ldots, v_r)$ to the corresponding factor of f turns additions into multiplications. For lattice reduction we need something that is linear, so we have to turn multiplications back into additions. One way to do that is using the following map:

$$g \mapsto \mathrm{Tr}_1(g)$$

where $\mathrm{Tr}_1(g)$ is the sum of the roots (with multiplicity) of g. So we get an additive map

$$\phi : v \mapsto \mathrm{Tr}_1(g_v) = \sum v_i \mathrm{Tr}_1(f_i)$$

from $\mathbb{Z}^r$ to the p-adic integers $\mathbb{Z}_p$.

van Hoeij, 2002

So lets take $t_i := \mathrm{Tr}_1(f_i) \in \mathbb{Z}_p$ for $i = 1, \ldots, r$ and look at this map

$$\phi : v = (v_1, \ldots, v_r) \mapsto \mathrm{Tr}_1(g_v) = v_1 t_1 + \cdots + v_r t_r$$

from $\mathbb{Z}^r$ to $\mathbb{Z}_p$.

If $g_v \in \mathbb{Z}[x]$ then $\mathrm{Tr}_1(g_v)$ is a integer bounded by some b (assume for now that f is monic. For b we can take N times a bound for the absolute values of the complex roots of f).
Set

$$\tilde{t}_i := (t_i \bmod p^a) \in \mathbb{Z}$$

Then

$$\mathrm{Tr}_1(g_v) = v_1 \tilde{t}_1 + \cdots + v_r \tilde{t}_r \ + \ \text{small multiple of } p^a$$

for any of our target v's (the v's for which $g_v \in \mathbb{Z}[x]$).

van Hoeij, 2002

For any of our target v's (i.e. $g_v \in \mathbb{Z}[x]$) we have:

$$\mathrm{Tr}_1(g_v) = v_1\tilde{t}_1 + \cdots + v_r\tilde{t}_r \ + \ \text{small multiple of } p^a.$$

Now $\mathrm{Tr}_1(g_v)$ is a coefficient of the factor g_v, but for efficiency we want to compute $(v_1, \ldots, v_r)$ without computing any coefficients of factors of f. So we take

$$s_i := \frac{\tilde{t}_i}{b} \in \mathbb{Q}$$

(the implementation rounds this to an integer for efficiency, but we'll skip that for simplicity).
Now let L be the lattice generated by:

$$(1, 0, \ldots, 0, s_1), \quad (0, 1, \ldots, 0, s_2), \ \ldots \ (0, 0, \ldots, 1, s_r)$$

and

$$(0, 0, \ldots, 0, \frac{p^a}{b}).$$

van Hoeij, 2002

Any target $v = (v_1, \ldots, v_r)$ corresponds to a vector

$$v' = (v_1, \ldots, v_r,\ \mathrm{Tr}_1(g_v)/b) \in L.$$

All entries of v' are bounded by 1, so

$$\| v' \| \leqslant B := \sqrt{r+1} \qquad (B \text{ is a bit higher if we rounded})$$

So if let L_B be the span of all vectors in L of length $\leqslant B$, and we let π be the projection on the first r coordinates, then all our target v's are in $\pi(L_B)$.

If the Big Gap Condition holds, then we can compute L_B with LLL. But we make no effort to ensure this condition, so we get some lattice L' such that

$$L_B \subseteq L'.$$

van Hoeij, 2002

Denote W as the span of our target v's (the 0–1 vectors corresponding to the irreducible factors of f in $\mathbb{Q}[x]$ form the reduced echelon basis of W).

Solving combinatorial problem $\Longleftrightarrow$ computing W.

Now

$$W \subseteq \pi(L_B) \subseteq \pi(L')$$

W is the lattice we want, and L' is the lattice we can get from LLL.

Given L' we can quickly test whether $\pi(L')$ equals W or not.

(check if the reduced echelon basis of $\pi(L')$ consists of 0–1 vectors, and if so, check like in Zassenhaus if those 0–1 vectors correspond to factors in $\mathbb{Z}[x]$ or not).

van Hoeij, 2002

If $\pi(L')$ equals W then we are done, and the resulting factors are irreducible regardless how many p-adic digits were used.

Prior factoring algorithms need some lower bound on the p-adic precision in order to prove that the factors are irreducible. Our algorithm does not need such a bound, because of the following

- Our algorithm only terminates if it finds $\dim(\pi(L'))$ factors in $\mathbb{Z}[x]$, whose product equals f.
- Any set of $\geqslant \dim(W)$ factors with product f are automatically irreducible.
- $\pi(L') \supseteq W$ is true for any p-adic precision.

  (if we didn't use any digits at all we'd get $L' = \mathbb{Z}^r$. Using more digits brings L' closer to W, but $L' \supseteq W$ will always hold, and termination only happens when $L' = W$).

van Hoeij, 2002

Since no bounds on the p-adic accuracy are needed to prove that the output is irreducible, we can be very flexible with how many p-adic digits to use. However, we only find the factors when $L' = W$, so in order for the algorithm to terminate, we do need that L' eventually becomes W.

So what if $\pi(L') \neq W$? We can gradually add more and more p-adic digits, but that may not be enough. Additional data may be needed. For instance, instead of Tr_1 (= sum of roots) we can also use Tr_2 (= sum of squares of roots), Tr_3 (= sum of cubes) etc.

One can prove that L' will eventually become W if we keep using more and more "traces" Tr_i and p-adic digits, see [vH 2002].

van Hoeij, 2002

If we had an oracle that told us exactly how many p-adic digits to use, and which traces Tr_i to use, in order to reach $L' = W$ in just one lattice reduction, and if we used this oracle, it would

- Be very helpful for determining a complexity bound for the algorithm (no bound is given in [vH 2002], only a termination proof).
- But it would not speed up the algorithm. In fact, it can even slow it down in certain types of examples. Gradually going from $\mathbb{Z}^r$ to W with a number of calls to LLL is not slower, and sometimes faster, than getting there with one LLL call. Understanding why this is so is the key to the new complexity result [vH and Novocin, 2007].

van Hoeij, 2002

Suppose we have $W \subseteq L \subseteq \mathbb{Z}^r$. The idea was to append some data to the vectors in L such that the target vectors will still have length $\leqslant B$ while most other vectors get longer. If they get sufficiently much longer than B then LLL can separate them from the B-short vectors so that we get an $L' \subseteq L$ of lower dimension, bringing us closer to our target W.

However, even if we get $L' = L$ after running LLL, we may still have made progress, because we're working a basis of L, and the result of running LLL can be that we now have a better basis of the same L, which will save time during the next LLL call.

So undershooting (not finding W after an LLL call) is better than overshooting (using way more digits than were needed to find W).

Belabas 2004

Strategy B in [Belabas 2004] organizes the adding of p-adic digits in such a way that each next call benefits maximally from the LLL-work done in the previous call.

This way the number of calls to LLL has very little impact on the total CPU time, because whichever work was done in one call will save the same amount of work for the remaining calls.

The advantage of this is the following:

- It allows him to add only few p-adic digits at a time without hurting the running time (adding few digits at a time means that more LLL calls will have occurred before the required number of digits was reached).
- The advantage of adding few p-adic digits at a time is that he can never overshoot the required number of digits by much. This way he prevents spending much more time than needed.

Belabas, vH, Klüners, Steel (arXiv '04 and JA'07 notes)

At the time, no complexity bound for the [vH 2002] algorithm was known. Now [LLL 1982] does have a bound, but to mimic this proof, we need to take a resultant, and for that, we need a polynomial instead of numbers $\mathrm{Tr}_i(g)$.
Now Tr_i is not the only thing that sends $*$ to $+$, the logarithmic derivative $g \mapsto g'/g$ does this too. The main idea of the paper was that all we have to do to get a polynomial (so we can take a resultant and get a complexity bound) is to multiply that by f.

So we switched from traces $\mathrm{Tr}_i(g)$ (sum of i'th power of roots) to coefficients of the polynomial

$$f \cdot \frac{g'}{g}$$

and this was the key idea for getting a polynomial time complexity result for a version of [vH 2002].

Belabas, vH, Klüners, Steel (arXiv '04 and JA'07 notes)

About this version of [vH 2002] for which we proved a polynomial time complexity.

Belabas' version works great in practice, but makes it a lot harder to bound the complexity because there is no reasonable bound on the number of LLL calls. So to get a complexity bound, we moved to the opposite direction: use enough p-adic digits and enough coefficients($f \cdot f_i'/f_i$) so that 1 call to LLL will provably be enough.

In other words: we're way overshooting!

This means that the version for which we got the poly-time complexity result is way slower than any of the implemented versions of [vH 2002].

Belabas, vH, Klüners, Steel (arXiv '04 and JA'07 notes)

This meant that we now had a polynomial-time complexity result for a version that nobody will ever use because it is much slower than the implemented versions.

Much effort was made to get a complexity result for a fast version of the algorithm (i.e. one that is actually used).

A good choice is version [Belabas 2004] because it is well defined (Belabas spelled out precisely which p-adic digits to add for each LLL call).

The cost of each individual call to LLL in [Belabas 2004] is very low (bounded by a polynomial depending solely on r, completely independent of both degree and coefficient size!)

However, ...

Belabas, vH, Klüners, Steel (arXiv '04 and JA'07 notes)

The bound we got for the number of LLL calls for the fast version [Belabas 2004] is huge.

So the bound we get in [BHKS] for the fast version is much worse than the bound for the slow version
(Theorem 4.6 only says "polynomially bounded" but does not spell out this polynomial in order to avoid embarrassment).

It is very unsatisfactory that the faster version should have a worse bound. The problem is that the key advantage of the fast version did not contribute at all to the complexity bound in [BHKS].

(A key advantage of [Belabas 2004] was that it is designed in such a way that the number of LLL calls does not matter much, which makes it easy to avoid overshooting)

van Hoeij and Novocin, 2007

The product "bound for number of LLL calls" times "bound for each LLL call" can not give a good bound because

- the number of calls in [Belabas 2004] can indeed be large
- the bound for each LLL call can not be improved (The cost of each LLL call is determined by the number of LLL switches it makes. The switch-complexity, i.e. the bound for the number of LLL switches, is $O(r^3)$, which is sharp.)

[vH&N 2007] proves a bound with the following property:

The bound for all of these LLL calls combined is the same as the bound for each of the individual calls.

The switch-complexity for all LLL calls combined is the same $O(r^3)$ as it is for any of the individual LLL calls.

van Hoeij and Novocin, 2007

And this describes the observed behavior of the algorithm perfectly.

There is an example in [Belabas 2004] that takes 62 LLL calls, with the bulk of the CPU time spent on just a handful of them. So experimentally, the cost for an individual LLL call is of the same magnitude as the cost for the total. The complexity result in [vH&N 2007] explains this observation perfectly.

This $O(r^3)$ is independent both of the degree and the coefficient size of f. How could a complexity bound possibly be independent of those?

- Here we are not yet bounding the cost of factoring f, at the moment we are only bounding the number of LLL switches used to solve the combinatorial problem, because this is what dominated the worst-case complexity.
- The cost for the other steps in factoring (like Hensel lifting) do of course depend on degree and coefficient size.
- Digits are fed gradually to LLL, so the LLL input never has vectors whose length depends on the coefficient size of f.
- We will give a lattice problem and show that it can be solved at a switch-complexity that is independent of coefficient-size.
- Applying this to the combinatorial problem shows its independence of coefficient size (text completed this week).
- Independence of degree is being written down right now by my student Andrew Novocin, this should soon be added to preprint [vH&N 2007].

Rough sketch of Lattice Reduction Algorithms

Let $b_1, \ldots, b_r \in L$ be a basis of L and denote $b_1^*, \ldots, b_r^* \in \mathbb{R}^m$ as the Gram-Schmidt orthogonalization over $\mathbb{R}$ of $b_1, \ldots, b_r$.

Let $l_i = \log_{4/3}(\| b_i^* \|^2)$, and $\mu_{i,j} = \frac{b_i \cdot b_j^*}{b_j^* \cdot b_j^*}$.

Input: A basis $b_1, \ldots, b_r$ of a lattice L.
Output: A LLL-reduced basis of L.

1. *(Gram-Schmidt over $\mathbb{Z}$)*. By subtracting suitable $\mathbb{Z}$-linear combinations of $b_1, \ldots, b_{i-1}$ from b_i make sure that $|\mu_{i,j}| \leq 1/2$ for all $j < i$.
2. *(LLL Switch)*. If there is a k such that interchanging b_{k-1} and b_k will decrease l_{k-1} by at least 1 then do so.
3. *(Repeat)*. If there was no such k in Step 2, then the algorithm stops. Otherwise go back to Step 1.

What LLL does

Let $b_1, \ldots, b_r$ be the current sequence of vectors.

Let $l_i = \log_{4/3}(\| b_i^* \|^2)$ be the logarithmic Gram-Schmidt lengths of our vectors.

What each LLL switch does is to move some of this G-S length from b_i's to later vectors in the sequence.

$$
\begin{array}{ccccccccc}
l_1 & \Longrightarrow & l_2 & \Longrightarrow & l_3 & \Longrightarrow & \cdots & \Longrightarrow & l_r \\
b_1^* & & b_2^* & & b_3^* & & \cdots & & b_r^* \\
b_1 & \leftrightarrow & b_2 & \leftrightarrow & b_3 & \leftrightarrow & \cdots & \leftrightarrow & b_r
\end{array}
$$

A random basis $b_1, \ldots, b_r$ has big l_1 and small l_r. Each LLL switch brings us closer to a good basis (small l_1 and big l_r).

(in our application, if $\| b_r^* \| > \sqrt{r+1}$ then $W \subseteq \pi(L')$ where $L' := \mathbb{Z}b_1 + \cdots \mathbb{Z}b_{r-1}$ and we get one step closer to finding W).

Solving the combinatorial problem in factoring

Take this matrix, which is matrix A from [BHKS] with the last N columns scaled down a factor $c := n \cdot B(f)$.

$$\begin{pmatrix} & & & & & p^a/c \\ & & & & \cdot & \\ & & & p^a/c & & \\ 1 & & & * & \cdots & * \\ & \ddots & & \vdots & \ddots & \vdots \\ & & 1 & * & \cdots & * \end{pmatrix}$$

- Let $b_1, \ldots$ be an LLL reduced basis for the rows of this matrix.
- As long as the G.S. length of the last one is greater than $B := \sqrt{r+1}$, remove it. Let $b_1, \ldots, b_s$ = remaining vectors.

Then [BHKS] Theorem 4.3 shows that $W = \pi(\mathbb{Z}b_1 + \cdots \mathbb{Z}b_s)$ where π = projection on $\mathbb{Z}^r$.

Solving the combinatorial problem in factoring

The entries of this $r+N$ by $r+N$ matrix depend on the coefficient size. Our task is to show that we can compute $W = \pi(\mathbb{Z}b_1 + \cdots \mathbb{Z}b_s)$ with a number of LLL-switches $O(r^3)$ that is independent of both N and the coefficient size.

Lets start with a basis for $\mathbb{Z}^r$ (certainly $W \subseteq \mathbb{Z}^r$ so we're still OK). That's the left lower corner of our matrix. Now add one row and column:

$$\begin{pmatrix} & & & p^a/c \\ 1 & & & * \\ & \ddots & & \vdots \\ & & 1 & * \end{pmatrix}$$

LLL this matrix will lead to $L' := \mathbb{Z}b_1 + \cdots \mathbb{Z}b_s$ with $W \subseteq \pi(L')$. If $W = \pi(L')$ then done, if $W \subsetneq \pi(L')$ we have to then we have to look at the next row/column.

van Hoeij and Novocin, 2007

$$\begin{pmatrix} & & & p^a/c \\ 1 & & & * \\ & \ddots & & \vdots \\ & & 1 & * \end{pmatrix}$$

Entries in the last column could be huge. So we do this

1. Scale down last column a factor 2^{rd} where d is big enough that this makes the last column of size $O(1)$.
2. Repeat d times:
   - Scale up last column a factor 2^r.
   - LLL (the vectors are the rows of the matrix)
   - Remove (if any) last vector(s) with G.S. length $> B = \sqrt{r+1}$.

Output = LLL reduced $b_1, \ldots, b_s$ with $W \subseteq \pi(\mathbb{Z}b_1 + \cdots \mathbb{Z}b_s)$.
We now have to show that the switch-complexity of this strategy is independent of d (the number of LLL calls).

van Hoeij and Novocin, 2007

We start with $r+1$ vectors, and at any given time we have $b_1, \ldots, b_s$ remaining vectors and $r+1-s$ removed vectors. Again $l_1, \ldots, l_s$ are the logarithmic Gram-Schmidt lengths. We now assign a value to the current configuration as:

$$\mu(b_1, \ldots, b_s) = 0 \cdot l_1 + 1 \cdot l_2 + \cdots + (s-1) \cdot l_s + (r+1-s) \cdot r \cdot \log_{4/3}(2^{3r} B^2)$$

The key to the proof is now that

- $\mu = 0$ at the beginning.
- No step in the algorithm decreases μ.
- Each LLL switch increases μ by at least 1, regardless in which LLL call that switch was made.
- μ can never become more than $(r+1-0) \cdot r \cdot 10r = 10(r+1)r^2$

Details on the blackboard.