Practical polynomial factoring in polynomial time

William Hart*
University of Warwick
Mathematics Institute
Coventry CV4 7AL, UK
W.B.Hart@warwick.ac.uk

Mark van Hoeij†
Florida State University
Tallahassee, FL 32306
hoeij@math.fsu.edu

Andrew Novocin
CNRS-INRIA-ENSL
46 Allée d'Italie
69364 Lyon Cedex 07, France
andy@novocin.com

ABSTRACT

There are several algorithms for factoring in $\mathbb{Z}[x]$ which have a proven polynomial complexity bound such as Schönhage of 1984, Belabas/Klüners/van Hoeij/Steel of 2004, and van-Hoeij/Novocin of 2009. While several other algorithms can claim to be comparable to the best algorithms in practice such as Zassenhaus on restricted inputs, van Hoeij of 2002, and Belabas of 2004. We present here the first algorithm for factoring polynomials in $\mathbb{Z}[x]$ which is both comparable to the best algorithms in practice and has a proven polynomial complexity bound. We show that the algorithm has a competitive runtime on many worst-case polynomials and can be made significantly faster on a wide class of common polynomials. Our algorithm makes its practical gains by requiring less Hensel lifting.

1. INTRODUCTION

Most practical factoring algorithms in $\mathbb{Z}[x]$ use a structure similar to [23]: factor modulo a small prime, Hensel lift this factorization, and use some method of recombining these local factors into integer factors. Zassenhaus performed the recombination step by an exhaustive search which can be made effective for as many as 40 local factors as is shown in [1]. While quite fast for many cases, the exponential complexity of this exhaustive technique is realized on many polynomials. In 2002 [8] used lattice reduction, the famous LLL algorithm of [12], to directly search for the right combinations. Because this algorithm was so fast it quickly became the standard in computer algebra systems. However, attempts to prove a polynomial time complexity bound for [8] and its efficient variants proved quite difficult. In 2004 [4] proved a polynomial time bound for a variant which required far too much Hensel lifting, and a far too large lattice reduction, to be considered practical (the same is true of the algorithm in [9]). In [17] an algorithm, called the r^3 algorithm, was presented that made two claims which can only be proved with a practical implementation:

Claim 1 The algorithm could be implemented in a practical way while maintaining a polynomial complexity bound.

Claim 2 It gave theoretical evidence that a practical improvement could be made on a large class of common polynomials while maintaining comparable behavior on worst-case polynomials.

The primary task of this paper is to provide a report on the first implementation of the r^3 algorithm. We demonstrate that this algorithm is indeed practical, making it the first algorithm which is both comparable to the best practical algorithms and polynomial time. Our implementation is done in FLINT, an open-source C library for number theory (see [7]) and compared with the much more polished implementation in NTL, an open-source C++ library (see [20]). We make this comparison because the two algorithms perform quite similar tasks (in a similar language) with the primary differences being the differences between the r^3 algorithm and the van Hoeij algorithm. Merely comparable times would be sufficient to demonstrate claim 1.

For claim 2, the class of polynomials for which we show practical savings over van Hoeij is any polynomial which would require more CPU time for the Hensel lifting phase than the recombination phase and has at least one factor of higher degree than the rest. To the best of our knowledge, all of the best practical algorithms for factoring use at least as much Hensel lifting as the Zassenhaus algorithm (as this could be required to reconstruct the integer factors). However, for polynomials which have one factor of large degree and zero or more small degree factors (including irreducible polynomials), a lower Hensel lifting bound is often sufficient to prove the irreducibility of the potential factors. In this case the low degree factors (if there are any) can be reconstructed at lower local precision than the large degree factors, and the large factor can be constructed by dividing away the smaller factors. Indeed, a large percentage of commonly arising polynomials that are also difficult for the Zassenhaus algorithm fall into this category. We demonstrate with our implementation the reduced level of Hensel lifting which is needed to solve several of these types of polynomials. The theoretical work of [17] encourages this aggressive approach to minimizing Hensel lifting, but only an implementation can verify these claims. We also present the algorithm in

*Author was supported by EPSRC Grant number EP/G004870/1

†Author was supported by NSF Grant number 0728853

a more implementable format than [17] with every effort to keep the algorithm simple to implement in any sufficiently advanced computer algebra system.

Roadmap Necessary background information will be included in section 2. The algorithm is laid out in an implementable fashion in section 3. A brief assertion of the polynomial complexity is laid out in section 4. Practical notes, including running time and analysis are included in section 5.

2. BACKGROUND

In this section we will outline necessary information from the literature. The primary methods for factoring polynomials which we address can all be said to share a basic structure with the Zassenhaus algorithm of 1969 [23].

2.1 The Zassenhaus algorithm

In some ways this is the first algorithm for factoring polynomials over $\mathbb{Z}$ which properly uses the power of a computer. For background on the evolution of factoring algorithms see the fine treatment in [10].

The algorithm utilizes the fact that the irreducible factors of f over $\mathbb{Z}$ are also factors of f over the p-adic numbers $\mathbb{Z}_p$. So if one has both a bound on the size of the coefficients of any integral factors of f and an irreducible factorization in $\mathbb{Z}_p[x]$ of sufficient precision then one can find the integral factors via simple tests (e.g. trial division). For coefficients of factors of f we can use the Landau-Mignotte bound (see [6, Bnd 6.33]). For the p-adic factorization it is common to choose a small prime p to quickly find a factorization over $\mathbb{F}_p$ then use the Hensel lifting method to increase the p-adic precision of this factorization. Due to a comprehensive search of all combinations of local factors the algorithm has an exponential complexity bound which is actually reached by application to the Swinnerton-Dyer polynomials.

ALGORITHM 1. *Description of Zassenhaus algorithm*

Input: *Square-free*[1] *and monic*[2] *polynomial* $f \in \mathbb{Z}[x]$ *of degree* N

Output: *The irreducible factors of* f *over* $\mathbb{Z}$

1. Choose a prime, p, such that $\gcd(f, f') \equiv 1$ modulo p.
2. **Modular Factorization:** Factor f modulo $p \equiv f_1 \cdots f_r$.
3. Compute the Landau-Mignotte bound $L \in \mathbb{R}$ and $a \in \mathbb{N}$ such that $p^a > 2L$
4. **Hensel Lifting:** Hensel lift $f_1 \cdots f_r$ to precision p^a.
5. **Recombination:** For each $v \in \{0,1\}^r$ (and in an appropriate order) decide if $g_v := \prod f_i^{v[i]}$ mods p^a divides f over $\mathbb{Z}$.

[1]Assumed square-free for simplicity. A standard gcd-based technique can be used to obtain a square-free factorization (see [6])

[2]We have assumed the polynomial is monic for simplicity. Each algorithm we present can be adapted to handle non-monic polynomials as well.

It is common to perform steps 1 and 2 several times to attempt to minimize, r, the number of local factors. Information from these attempts can also be used in clever ways to prove irreducibility or make the recombination in step 5 more efficient, see [1] for more details on these methods. Algorithms for steps 2, 3, and 4 have been well studied and we refer interested readers to a general treatment in [6]. Our primary interests in this paper lie in the selection of a and the recombination of the local factors in step 5.

2.2 Overview of the LLL algorithm

In 1982 Lenstra, Lenstra, and Lovasz devised an algorithm, of a completely different nature, for factoring polynomials. Their algorithm for factoring had a polynomial time complexity bound but was not the algorithm of choice for most computer algebra systems as Zassenhaus was more practical for the majority of everyday tasks. At the heart of their algorithm for factoring polynomials was method for finding 'nice' bases of lattices now known as the LLL algorithm. The LLL algorithm for lattice reduction proved to be of a great deal of practical interest in many areas of computational number theory and cryptography, as it (amongst other things) gives an approximate solution to the shortest vector problem, which is NP-hard [2], in polynomial time. In fact, the van Hoeij algorithm for factoring polynomials can be thought of as the application of the LLL lattice reduction algorithm to the Zassenhaus recombination step. The purpose of this section is to present some facts from [12] that will be needed throughout the paper. For a more general treatment of lattice reduction see [13].

A lattice, L, is a discrete subset of $\mathbb{R}^n$ that is also a $\mathbb{Z}$-module. Let $\mathbf{b}_1, \ldots, \mathbf{b}_d \in L$ be a basis of L and denote $\mathbf{b}_1^*, \ldots, \mathbf{b}_d^* \in \mathbb{R}^n$ as the Gram-Schmidt orthogonalization over $\mathbb{R}$ of $\mathbf{b}_1, \ldots, \mathbf{b}_d$. Let $\delta \in (1/4, 1]$ and $\eta \in [1/2, \sqrt{\delta})$. Let $l_i = \log_{1/\delta} \| \mathbf{b}_i^* \|^2$, and denote $\mu_{i,j} = \frac{\mathbf{b}_i \cdot \mathbf{b}_j^*}{\mathbf{b}_j^* \cdot \mathbf{b}_j^*}$. Note that $\mathbf{b}_i, \mathbf{b}_i^*, l_i, \mu_{i,j}$ will change throughout the algorithm sketched below.

DEFINITION 1. $\mathbf{b}_1, \ldots, \mathbf{b}_d$ *is* LLL-reduced *if* $\| \mathbf{b}_i^* \|^2 \leq \frac{1}{\delta - \mu_{i+1,i}^2} \| \mathbf{b}_{i+1}^* \|^2$ *for* $1 \leq i < d$ *and* $|\mu_{i,j}| \leq \eta$ *for* $1 \leq j < i \leq d$.

In the original algorithm the values for (δ, η) were chosen as $(3/4, 1/2)$ so that $\frac{1}{\delta - \eta^2}$ would simply be 2.

ALGORITHM 2. *Rough sketch of LLL-type algorithms*

Input: *A basis* $\mathbf{b}_1, \ldots, \mathbf{b}_d$ *of a lattice* L.

Output: *An LLL-reduced basis of* L.

1. $\kappa := 2$
2. **while** $\kappa \leq d$ **do:**
   (a) *(Gram-Schmidt over* $\mathbb{Z}$*).* By subtracting suitable $\mathbb{Z}$-linear combinations of $\mathbf{b}_1, \ldots, \mathbf{b}_{\kappa-1}$ from $\mathbf{b}_\kappa$ make sure that $|\mu_{i,\kappa}| \leq \eta$ for $i < \kappa$.

(b) *(LLL Switch).* If interchanging $\mathbf{b}_{\kappa-1}$ and $\mathbf{b}_\kappa$ will decrease $l_{\kappa-1}$ by at least 1 then do so.

(c) *(Repeat).* If not switched $\kappa := \kappa + 1$, if switched $\kappa = \max(\kappa - 1, 2)$.

That the above algorithm terminates, and that the output is LLL-reduced was shown in [12]. There are many variations of this algorithm (such as [11, 18, 22, 15, 19]) and we make every effort to use it as a black box for ease of implementation. What we do require is an algorithm which returns an LLL-reduced basis and whose complexity is roughly linear in the number of times step 2a (and thus 2b) is called. In fact the central complexity result of [17] is that the r^3 algorithm has $\mathcal{O}(r^3)$ switches throughout the entire algorithm, in spite of many calls to LLL.

Intuitively the G-S lengths of an LLL-reduced basis do not drop as fast as a generic basis (for more on generic bases and LLL see [16]). In practice this property is used in factoring algorithms to separate vectors of small norm from the rest. The primary variations of the van Hoeij, Belabas, and r^3 algorithms is how often they call LLL, the types of input, and the timing of the calls.

2.3 Overview of the BHKS result

Let $f \in \mathbb{Z}[x]$ be a polynomial of degree N. Let H be a bound on the absolute value of the coefficients of f. Let p be a prime such that $f \equiv l_f f_1 \cdots f_r \bmod p^a$ a separable irreducible factorization of f in the p-adics lifted to precision a, the f_i are monic, and l_f is the leading coefficient of f.

We will make some minor changes to the All-Coefficients matrix defined in [4] to produce a matrix that looks like:

$$\begin{pmatrix} & & & & p^{a-b_N} \\ & & & \iddots & \\ & & p^{a-b_1} & & \\ 1 & & c_{1,1} & \cdots & c_{1,N} \\ & \ddots & \vdots & \ddots & \vdots \\ & 1 & c_{r,1} & \cdots & c_{r,N} \end{pmatrix}$$

Here $c_{i,j}$ represents a CLD (we'll use this term often) that is the j^{th} coefficient of the 'logarithmic derivative' $f_i' \cdot f / f_i$ mods p^a divided by p^{b_j} and p^{b_j} represents $\sqrt{N}$ times a bound on the j^{th} coefficient of $g' \cdot f/g$ for any true factor $g \in \mathbb{Z}[x]$ of f. In this way the targeted vectors in the span of the rows will be quite small. An empty spot in this matrix represents a zero entry. For large enough values of p^a a reduction of the rows of this matrix will solve the recombination problem by a similar argument to the one presented in [4] and reformulated in [17, lemma 11].

Seeing this matrix helps to understand the objective of the r^3 algorithm. The r^3 algorithm begins with the bottom left corner of this matrix and grows in size until it has enough data to solve the problem. The key difference is that not every column will be used, but each column will be considered.

3. THE MAIN ALGORITHM

In this section it is our intention to lay out the central algorithm as we have implemented it.

The following algorithm is the main wrapper of our factoring method. It follows closely the pattern of Algorithm 1 with the exception of computing a lower initial value for the p-adic precision, the method of recombination, and increasing the precision more frequently.

ALGORITHM 3. *The main algorithm*

Input: *Square-free, monic, polynomial $f \in \mathbb{Z}[x]$ of degree N*

Output: *The irreducible factors of f over $\mathbb{Z}$*

1. Choose a prime, p, such that $\texttt{gcd}(f, f') \equiv 1$ modulo p.
2. **Modular Factorization:** Factor f modulo $p \equiv f_1 \cdots f_r$.
3. **if** $r \leq 10$ **return** Zassenhaus(f)
4. Compute first target precision a with `Algorithm 4`
5. **until solved**:
   (a) **Hensel Lifting:** Hensel lift $f_1 \cdots f_r$ to precision p^a.
   (b) **Recombination:** `Algorithm 6`$(f, f_1, \ldots, f_r, p^a)$
   (c) **if** not solved: $a := 2a$

Because step 5a is potentially repeated, care should be taken not to recompute necessary information. Our implementation uses a balanced factor tree approach as seen in [6, Sect. 15.5]. To minimize overhead costs of Hensel lifting multiple times our implementation stores the lifting tree, intermediate modular inverses computed by the extended gcd, and the intermediate products of the lifting tree itself. This encourages choosing an aggressively low value of a so that, with some luck, the Hensel costs can be minimized, and without luck, there is little harm.

We now outline our suggested heuristic for selecting an initial p-adic precision, a. This heuristic is designed so that we lift only far enough to guarantee at least one useful call to LLL in Algorithm 6 which is sometimes enough to solve the problem. Provided the other parts of the algorithm are careful, this heuristic minimizes the amount of Hensel lifting performed in practice.

ALGORITHM 4. *Heuristic for initial precision*

Input: *$f \in \mathbb{Z}[x]$, p*

Output: *Suggested target precision a*

1. Use Algorithm 5 to find, b, the lower coefficient bound of either x^0 or x^{N-1}.
2. **return** $a := \left\lceil \frac{2.5r + \log_2 b + (\log_2 N)/2}{\log_2 p} \right\rceil$

This heuristic is based on the observation that many polynomials have their largest coefficients near the highest degree terms, the lowest degree terms, or near the middle degree terms. If this observation holds true then using only the bounds for the highest and lowest terms will give a rough estimate for when at least several coefficients will be usable. In the worst case we still have one usable coefficient using this method. As a sanity check, we recommend using the minimum of this value for a and the value corresponding with the Landau-Mignotte bound used by Algorithm 1.

The quality of Algorithm 4 relies heavily on the quality of bounds for the CLDs. The following method (an analogous bound for the bivariate case is given in [4, Lemma 5.8]) quickly gives fairly tight bounds in practice. The method is based on the fact that $g'f/g = \prod_{\alpha|g(\alpha)=0} \frac{f}{x-\alpha}$ summed over all roots of the potential factor.

ALGORITHM 5. *CLD bound*

Input: $f = a_0 + \cdots + a_N x^N$ *and* $c \in \{0, \ldots, N-1\}$

Output: X_c*, a bound for the absolute value of the coefficient of* x^c *in the polynomial* fg'/g *for any* $g \in \mathbb{Z}[x]$ *dividing* f*.*

1. Let $B1(r) := \frac{1}{r^{c+1}}(|a_0| + \cdots + |a_c|r^c)$
2. Let $B2(r) := \frac{1}{r^{c+1}}(|a_{c+1}|r^{c+1} + \cdots + |a_N|r^N)$
3. Find $r \in \mathbb{R}^+$ such that $\text{MAX}\{B1(r), B2(r)\}$ is roughly minimized
4. return $X_c := N \cdot \text{MAX}\{B1(r), B2(r)\}$

In this method, for any positive real number r, one of $B1(r)$ or $B2(r)$ will be an upper bound for the coefficient of x^c in $\frac{f}{x-\alpha}$ for each possible complex root α (because of the monotonicity of $B1$ and $B2$, if $r \leq |\alpha|$ then $B1(r)$ is the upper bound and if $r \geq |\alpha|$ then $B2(r)$ will be). Thus the maximum of $B1(r)$ and $B2(r)$ is an upper bound for every root. The smaller one can make $\text{MAX}\{B1(r), B2(r)\}$ the better the bound you return. Since the CLD is summed over every root of g we use N as a bound for the number of roots of g (if you have extra information about the factors then perhaps you can use that here).

We leave step 3 intentionally vague as many computer algebra systems have fast solvers for such situations. Our method was to begin with a value of $r = 1$, compute $B1(r)$ and $B2(r)$, then replace r by either $2r$ or $r/2$. Repeat this until the one bound overtakes the other then replace 2 by $\sqrt{2}$ and continue like that. We did this because the bound need not be optimal and it works well enough in practice. As anecdotal evidence we note that our code had a bug which caused overly large CLD bounds, once this bug was fixed the running times in many cases were nearly cut in half.

The next several algorithms form the core of our approach. While similar to the 'fine-tuning' of [8] found in [3], which introduced the concept of gradually adding data to the lattice, we have two primary differences. First we make use of CLDs which tend to have tighter bounds than traces. This allows us to attempt solving the problem with less Hensel lifting, which is key to our improved running times. The second crucial difference is the decision making process in Algorithm 7, which decides if a given CLD has enough data to justify a call to LLL. This step is important to the complexity analysis in [17] as it guarantees that the work done in the next call to LLL will help solve the problem more than it might undo previous work. One of the important contributions of this paper is demonstrating, via actual implementation, that the added cost of this decision does not impact the practicality of the algorithm. Another important contribution is demonstrating that in many common cases the reduced amount of Hensel lifting offers a practical advantage.

The following algorithm is the wrapper of our recombination process. It is designed to efficiently use all possible data from the most recent round of Hensel lifting. In the algorithm we take new CLD data from the local factors and decide if it is worth calling LLL with this data. The rows of M form the basis of our lattice and we gradually adjoin new columns of data to M and occasionally new rows.

ALGORITHM 6. *Attempt Reconstruction*

Input: f*,* $f_1, \ldots, f_r$ *the lifted factors, their precision* p^a*, and possibly* $M \in \mathbb{Z}_{s\times(r+c)}$*.*

Output: *If solved then the irreducible factors of* f *over* $\mathbb{Z}$ *otherwise an updated* M*.*

1. If this is the first call let $M := I_{r\times r}$
2. Until solved or all data used do:
   (a) Compute some new column vectors:
   $\mathbf{x}_j := (c_{1,j}, \ldots, c_{r,j})^T$
   Here $c_{i,j}$ is the coefficient of x^j in $f \cdot f_i'/f_i$
   (b) For each new vector $\mathbf{x}_j$ until it is fully used:
      i. Use Algorithm 5 to compute X_j
      ii. Use Algorithm 7 to possibly update M with $\mathbf{x}_j$
      iii. If updated then run LLL(M)
      iv. Compute G-S lengths of rows of M
      v. Decrease the number of rows of M until the final G-S length $\leq \sqrt{r+1}$
      vi. Use Algorithm 8 to test if solved

For the computation of new data in step 2a it should be noted that to compute the top (or bottom) l coefficients of ff_i'/f_i modulo p^a only the top (or bottom) l coefficients of f and f_i are needed. So in practice we compute 5-20 of the top and bottom coefficients first as this is often enough to either solve the problem or determine that more Hensel lifting is needed. Indeed when all of these computed coefficients are checked we use a heuristic to decide if computing more

CLDs would be worth it or if more Hensel lifting is needed. It is often the case that the top coefficients or the bottom coefficients have the lowest bounds and thus the most information so computing middle terms might not be worth the time. For a proven polynomial time complexity bound this heuristic must always decide to compute more data when the level of p-adic precision is large enough (see section 4 for more details). Although, to the best of our knowledge, no examples have ever been recorded which require that level of p-adic precision to be solved.

The following algorithm outlines the decision making process when deciding if a given CLD justifies a call to LLL. Its function is mostly theoretical as it allows the complexity work of [17] to work, although it does work well in practice for keeping the number of columns in M low and quickly handling the useless CLDs. The idea of the check is to see what the new entries would be at full precision then to determine if enough bits lie above the CLD bound to make the vectors significantly larger than target vectors. There is also a theoretical check to see if a vector of the form $(0, \ldots, 0, p^a)$ is needed to keep the entries reduced modulo p^a (for detail on the theory of this check see [17, Sect 3.12]). It should be noted that each column could be used multiple times adjoining a new entry each time. In [17] this was handled in a different way which complicated the algorithm and analysis, here we see no difference between a CLD used multiple times and multiple CLDs.

ALGORITHM 7. *Decide if adjoining data*

Input: *$M \in \mathbb{Z}_{s\times(r+c)}$, data vector $\boldsymbol{x}_j$, p^a, X_j the CLD bound for x^j*

Output: *A potentially updated M*

1. Let $B := r + 1$ and s be the number of rows of M
2. If $p^a < X_j \cdot B \cdot \sqrt{N} \cdot 2^{(1.5)r}$ then exit
3. Find U the first r columns of M
4. Compute $\mathbf{y}_j := U \cdot \mathbf{x}_j$
5. If $\| \mathbf{y}_j \|_\infty < X_j \cdot B \cdot \sqrt{N} \cdot 2^{(1.5)r}$ then exit
6. If $p^a - Bp^a/2^{(1.5)r} > \| \mathbf{y}_j \|_\infty \cdot (2(3/2)^{s-1} - 2)$ then no_vec :=True otherwise False
7. Find new column weight 2^k roughly $\frac{\|\mathbf{y}_j\|_\infty}{X_j \cdot B \cdot \sqrt{N} \cdot 2^{(1.5)r}}$ if no_vec is True and $\frac{p^a}{X_j \cdot B \cdot \sqrt{N} \cdot 2^{(1.5)r}}$ if False
8. Embed $\mathbf{x}_j$ and $p^a/2^k$ into $\mathbb{Z}/2^r$ by rounding or truncating say $\tilde{\mathbf{x}_j}$ and $\tilde{P}$
9. If no_vec is True then augment M with new column $\tilde{\mathbf{y}_j} = U \cdot \tilde{\mathbf{x}_j}$

   If no_vec False then also adjoin a new row so

   $$M := \left[\begin{array}{c|c} \mathbf{0} & \tilde{P} \\ \hline M & \tilde{\mathbf{y}_j} \end{array}\right]$$

This algorithm is rather technical so we attempt to shed some light now. Step 2 catches any CLD which is eliminated by insufficient p-adic precision, as a result this step would likely be more practical when you choose which CLDs to compute in Algorithm 6step 2a. Step 4 computes what the new column would look like at full precision in our lattice, so in the next step if these values are large enough to give some interesting data we just skip this CLD. Step 6 is a technical condition from [17, sect.3.12] that tests if the new entries are so much smaller than p^a that LLL would simply move the potential vector $(0, \ldots, 0, p^a)$ to the final spot and then have it removed. In practice the condition is very accurate for detecting the utility of this extra vector. Finally one crucial difference between this algorithm and the r^3 algorithm in [17] is the fact that the new entries are not full precision rational numbers. We allow r bits of decimals for the sake of stability in LLL, but in practice we want to avoid having huge decimal entries. Since some implementations of LLL only allow integer entries we note that the embedding of step 8 can be virtually accomplished by scaling up the entries in U by 2^r. Such a scaling requires replacing $B = r+1$ with $2^{2r}(r+1)$ whenever needed.

Finally we briefly mention the new method in which we check for true factors. One of the central novelties to the algorithm is a reduced level of Hensel lifting when attempting to solve the problem. It has been observed that the Landau-Mignotte bound is often too pessimistic and that even Zassenhaus' algorithm could potentially terminate at a lower level of Hensel lifting. This is also true of our lattice based attack, as we can often prove the irreducibility of potential factor before we can fully reconstruct each of its coefficients. This is seen most frequently in polynomials which turn out to have one large degree factor and zero or more smaller degree factors. In these cases we must check for true factors in a way that will recover the large factor by dividing away any small irreducible factors. Such cases arise naturally and frequently, see section 5 for examples.

We will begin by using a short-cut for detecting a 0-1 basis of our lattice. Such a basis, if it exists, could potentially solve the recombination problem.

ALGORITHM 8. *Check if solved*

Input: *M, f, $f_1, \ldots, f_r$ to precision p^a*

Output: *If possible the irreducible factors of f in $\mathbb{Z}[x]$*

1. Attempt to partition the first r columns of M under the equivalence relation $=$
2. If a partition exists and has less classes than there are rows of M we have a potential solution otherwise exit
3. For each class multiply the polynomials matching the columns in that class and reduce with symmetric remainder modulo p^a to find the potential factors
4. In order, from the lowest degree to the highest degree, perform trial divisions of f
5. If any two polynomials fail to divide f then no solution

6. Otherwise we have a solution, if a single failed polynomial then recover it by division of f by the successful factors

If we have the correct lattice than a row reduced echelon basis of M will have a single 1 in each column. So we detect this basis by seeing which columns match, and if we can achieve a basis with exactly one 1 in each column then we move on. The symmetric remainder of step 3 is required for polynomials with negative coefficients. By moving from the lowest degree to the highest degree we maximize the chances of solving the problem with less Hensel lifting than the Landau-Mignotte bound used in Zassenhaus.

4. POLYNOMIAL COMPLEXITY BOUND

A precise complexity analysis would require an in-depth treatment of lattice reduction in this setting such as [9] and [17]. In this report we merely assert a polynomial time complexity bound and reserve an in-depth treatment for future work.

ASSERTION 1. *Algorithm 3 has a polynomial-time complexity bound.*

We outline a simplistic, and non-rigorous, argument. Each component of the algorithm (local factoring, hensel lifting, LLL, and some additional matrix multiplications) has some classical polynomial time complexity bound in the degree N, number of local factors r, and height of the input polynomial H (for an introduction to any of these algorithms see [6]).

For each iteration of the main loop (step 5 of Algorithm 3 there are no more than $\mathcal{O}(Nlog(H)/r)$ calls to the primary procedures, including LLL, making the cost of each loop at most polynomial. So the central threat to a polynomial time complexity bound for our algorithm is the number of iterations of the main loop.

Each iteration of the main loop ends with a single call of Hensel lifting to double the precision. We merely show that the number of calls to Hensel lifting is polynomial thus the number of calls to the main loop is polynomial and since the cost of each loop is polynomial this is sufficient.

The complexity work in Theorem 4.3 of [4] and adapted to our format in Lemma 11 of [17] can be used to prove that the number of iterations of the main loop is only $\mathcal{O}(\log N + \log\log H)$. This was done there by showing that when a lattice contains all target 0-1 vectors, does not solve the factorization problem, but does not give any CLDs which are usable by Algorithm 7 then there is one vector which corresponds with a polynomial, h, with $0 < \text{Res}(h,f) \leq 2^{\mathcal{O}(N^2+N\log H)}$ and $\text{Res}(h,f) \equiv 0 \bmod p^a$. Thus if p^a is larger than the upper bound on $\text{Res}(h,f)$ then no such vector exists so at least one coefficient will be usable. In [17] great care was taken to show that solving the problem on $\mathcal{O}(r^2)$ CLDs would be sufficient to solve the problem. In our case, to show polynomial time complexity, it would be sufficient to have each of the $\mathcal{O}(NlogH/r)$ CLDs included at this largest Hensel lifting bound as in [4].

As the Hensel lifting begins at precision p^1 and doubles the exponent each time the number of times this can happen (and thus a bound for the number of iterations of the main loop) is bounded by $\mathcal{O}(\log N + \log\log H)$. Of course a much more enlightening bound is possible in an extended analysis.

5. RUNNING TIMES AND PRACTICAL OBSERVATIONS

In this section we assert the practicality of our algorithm. We do this by providing running times of our implementation side by side with a more polished implementation in NTL version 5.5.2. We provide timings on a collection of polynomials made available on NTL's website which was collected by Paul Zimmerman and Mark van Hoeij, as well as two other polynomials mentioned below. All but T1 and T2 were designed to test the recombination phase of knapsack-based factoring techniques, so those can be considered worst-case polynomials. The goal of these times is to show that our algorithm is comparable on these worst case polynomials.

A highly polished factoring algorithm will include many tricks for special cases, techniques, heuristics, and optimizations which our code is still developing. We expect these times to improve as some rather naive code is replaced and better heuristics are developed. The central contribution of these times is illustrating that this algorithm is indeed practical making it the first efficiently practical algorithm with a provably polynomial complexity bound.

Poly	r	NTL	H-bnd	Alg 3	H-bnd
P1	60	.276	29^{311}	.152	89^{33}
P2	20	.424	11^{437}	.196	11^{22*}
P3	28	1.08	11^{629}	.444	11^{31*}
P4	42	2.048	13^{745}	1.976	7^{80*}
P5	32	.104	19^{51}	.056	23^{26}
P6	48	.340	19^{152}	.220	23^{38*}
P7	76	1.188	37^{78}	.956	19^{74}
P8	54	3.472	13^{324}	1.812	11^{84}
M12_5	72	12.437	13^{1171}	4.856	11^{180}
M12_6	84	21.697	13^{1555}	10.277	13^{190*}
S7	64	.428	29^{78}	.512	47^{41}
S8	128	3.856	47^{140}	6.224	53^{79}
T1	30	3.896	7^{495}	1.636	7^{40}
T2	32	3.18	7^{200}	1.676	7^{43}

These timings are measured in seconds and were made on a 2400MHz AMD Quad-core Opteron processor, using gcc version 4.4.1 with the -O2 optimization flag, although the processes did not utilize all four cores. Our algorithm was limited to values of r of around 150 or less as our floating point LLL, using so-called double extended precision entries, becomes numerically unstable around dimension 150 (for more details see [16, Sect. 6]). Also for the sake of comparison NTL's default 'power hack' strategy[3] was deactivated in these timings as we had not yet implemented our own, we did not include timings for H1 and H2 without this power hack.

The columns labelled 'H-bnd' give the p-adic precision at

[3]The power hack is an attempt to detect structure of the form $g(x^k)$ in f so that smaller polynomials can be reduced in some cases.

which both algorithms solved the combinatorial problem. No matter what the current quality of our implementation these Hensel bounds show the potential for faster times yet. We expect that NTL's bounds are similar to other implementations of van Hoeij and that our algorithm will, in general, use less Hensel lifting than other versions as well. In the case of the five polynomials which include a $*$, the level of Hensel lifting shown was sufficient to solve the combinatorial problem but one more round of Hensel lifting was needed before all of the factors could be successfully reconstructed.

The polynomials S7 and S8 are Swinnerton-Dyer polynomials, and the factorization of such polynomials is dominated by the LLL costs. We see that in those examples our aggressive approach to Hensel lifting is of little significance and the less strong parts of our code dominate. On the other end of the spectrum the polynomials T1 and T2, which arose from rather standard computations in Trager's Algorithm, can be solved with as little p-adic precision as 7^{40} which is significantly less than the Landau-Mignotte bound. While our implementation is still young, such examples are plentiful and arise naturally in many applications. In a later implementation we hope to produce a highly optimized version. The SAGE [21] computer algebra system (which also uses FLINT) has a specially patched version of NTL which can beat the generic version of NTL that we have used. MAGMA [5] also has a highly optimized van Hoeij implementation which we believe performs very well. For example, the MAGMA implementation of factorization over $\mathbb{Z}/p\mathbb{Z}$ is already more than twice as fast as our more naive implementation. As our code develops we will have more in-depth and up-to-date timings, which we will make available online at `http://andy.novocin.com/timings`.

Acknowledgements The authors would like to Damien Stehlé for very generous assistance and many interesting conversations.

6. REFERENCES

[1] J. Abbott, V. Shoup and P. Zimmermann, *Factorization in $\mathbb{Z}[x]$: The Searching Phase*, ISSAC'2000 Proceedings, 1–7 (2000).

[2] M. Ajtai *The shortest vector problem in L2 is NP-hard for randomized reductions*, STOC 1998, pp. 10–19.

[3] K. Belabas *A relative van Hoeij algorithm over number fields*, J. Symb. Comp. **37** 2004, pp. 641–668.

[4] K. Belabas, M. van Hoeij, J. Klüners, and A. Steel, *Factoring polynomials over global fields*, preprint arXiv:math/0409510v1 (2004).

[5] J. J. Cannon, W. Bosma (Eds.) *Handbook of Magma Functions*, Edition 2.13 (2006)

[6] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[7] W. Hart *FLINT*, open-source C-library. `http://www.flintlib.org`

[8] M. van Hoeij, *Factoring polynomials and the knapsack problem*, J. Num. The., **95** 2002, pp. 167–189.

[9] M. van Hoeij and A. Novocin, *Gradual sub-lattice reduction and a new complexity for factoring polynomials*, accepted for proceedings of LATIN 2010.

[10] E. Kaltofen, *Polynomial factorization.* In: Computer Algebra, 2nd ed„ editors B. Buchberger et all, Springer Verlag, 95–113 (1982).

[11] E. Kaltofen,*On the complexity of finding short vectors in integer lattices*, EUROCAL'83,LNCSv.162,pp.235–244

[12] A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász, *Factoring polynomials with rational coefficients*, Math. Ann. **261** 1982, pp. 515–534.

[13] L. Lovász, *An Algorithmic Theory of Numbers, Graphs and Convexity*, SIAM 1986

[14] I. Morel, D. Stehlé, and G. Villard, *H-LLL: Using Householder Inside LLL*, ISSAC'09, pp. 271–278.

[15] P. Nguyen and D. Stehlé, *Floating-point LLL revisited*, Eurocrypt 2005, v. 3494 LNCS, pp. 215–233.

[16] P. Nguyen and D. Stehlé, *LLL on the Average*, ANTS VII 2006, v. 4076 LNCS, pp. 238–256.

[17] A. Novocin, *Factoring Univariate Polynomials over the Rationals*, Ph.D. Flor. St. Univ. 2008.

[18] C. P. Schnorr, *A more efficient algorithm for lattice basis reduction*, J. of Algo. **9** 1988, pp. 47–62.

[19] A. Schönhage, *Factorization of univariate integer polynomials by Diophantine approximation and an improved basis reduction algorithm*, ICALP'84, LNCS **172**, pp. 436–447.

[20] V. Shoup *NTL*, open-source C++ library. `http://www.shoup.net/ntl/`

[21] W. Stein *SAGE Mathematics Software.* `http://www.sagemath.org`

[22] A. Storjohann, *Faster algorithms for integer lattice basis reduction*, Technical report 1996, ETH Zürich.

[23] H. Zassenhaus, *On Hensel factorization I*, Journal of Number Theory (1969), pp. 291–311.