Int. J. Appl. Comput. Math (2015) 1:327–342
DOI 10.1007/s40819-014-0023-5

ORIGINAL PAPER

An Experimental Comparison of Some LLL-Type Lattice Basis Reduction Algorithms

Dimitris G. Papachristoudis · Spyros T. Halkidis · George Stephanides

Published online: 13 January 2015
© Springer India Pvt. Ltd. 2015

Abstract In this paper we experimentally compare the performance of the L^2 lattice basis reduction algorithm, whose importance recently became evident, with our own Gram-based lattice basis reduction algorithm, which is a variant of the Schnorr–Euchner algorithm. We conclude with observations about the algorithms under investigation for lattice basis dimensions up to the theoretical limit. We also reexamine the notion of "buffered transformations" and its impact on performance of lattice basis reduction algorithms. We experimentally compare four different algorithms directly in the Sage Mathematics Software: our own algorithm, the L^2 algorithm and "buffered" versions of them resulting in a total of four algorithms.

Keywords LLL · L^2 · Buffered transformations · Lattice basis reduction

Introduction

The importance of lattice theory has been profound during the last decades [5]. Researchers have started reexamining the LLL algorithm [14] and introduced variants of it like the Schnorr–Euchner algorithm [21], algorithms that are based entirely on the Gram matrix of the basis vectors [23], floating point versions of reduction algorithms (see Chapter 1 of [18]), and also an algorithm running quadratically with respect to $\log M$, where an integer d-dimensional lattice basis with vectors of norm less than M is originally given [16,17]. Finally, a recent algorithm with quasi-linear time complexity with respect to β, where $\beta = \log \max \|\underline{b}_i\|$ is

D. G. Papachristoudis · S. T. Halkidis (✉) · G. Stephanides
Computational Systems and Software Engineering Laboratory, Department of Applied Informatics, University of Macedonia, Egnatia 156, 54006 Thessaloniki, Greece
e-mail: halkidis@java.uom.gr

D. G. Papachristoudis
e-mail: mai1223@uom.gr

G. Stephanides
e-mail: steph@uom.gr

presented in [19] and $\underline{b}_i$ the basis vectors of the lattice. However, this algorithm is currently only theoretically justified and has not yet been implemented in practice.

The goal of lattice basis reduction is to find a basis of the same lattice, whose vectors are short and nearly orthogonal to each other. The first algorithm to achieve this was the Lenstra, Lenstra, Lovász [14], also known as the LLL algorithm, in 1982. Schnorr and Euchner [21] introduced the first lattice basis reduction algorithm that could efficiently be used in practice in 1991. Since then, many variants of the LLL algorithm have been proposed [2,5,16–18,23]. A survey of different aspects of the LLL algorithm is given in [18].

Furthermore, during the mid-1990s, several cryptosystems were introduced whose underlying hard problem was the shortest vector problem (SVP) and/or the closest vector problem (CVP) [10] in a lattice L of large dimension d. The most important of these, in alphabetical order, were the Ajtai–Dwork cryptosystem [1], the GGH cryptosystem of Goldreich, Goldwasser and Halevi [9] and the NTRU cryptosystem proposed by Hoffstein, Pipher and Silverman [11]. However, it has not been until recently that lattice basis reduction algorithms have started to gain attention. The main reason for this is that it has been shown that cryptosystems based on classical problems are prone to various attacks, and most importantly if the quantum computer were to be built in the near future, we will have to move to post-quantum cryptography [4] which includes lattice-based cryptography as one of its main techniques, in order to ensure the privacy of data.

In our work we examine floating point arithmetic lattice basis reduction algorithms in contrast to exact arithmetic lattice basis reduction algorithms. In order to be able to compare the different basic lattice basis reduction algorithms, we have chosen to implement four of them, namely our variant of the Schnorr–Euchner algorithm, the recent L^2 [16,17] algorithm and their "buffered" versions. We implemented them as Sage [6,22] scripts, based on the Python [20] programming language so that no algorithm would gain an advantage due to its implementation in a more efficient language. Moreover, in this way we make the algorithms available in their simplest form to the Sage community. We are not interested in choosing an optimal programming environment in terms of efficiency, since our aim is not to produce the fastest possible implementation but to compare the different algorithms chosen in the same environment. Moreover, we have chosen the Sage environment because it is an open–source computer algebra system (CAS) to which we wanted to contribute. Additionally, it is appropriate for expensive mathematical computations and is supported by a large community.

In this paper, apart from the comparison of the different lattice basis reduction algorithms, we propose a new algorithm. Our effort to design this new algorithm was initiated by [2].

The rest of the paper is organized as follows: In "Preliminaries" section we describe some mathematical preliminaries. "A Novel Algorithm Initiated by the Bu ered LLL Gram" section analyzes our proposed algorithm. "Experimental Results" section describes the experimental results on the comparison of the four lattice basis reduction algorithms that we implemented. Finally, in "Conclusions and Future Work" section we draw some conclusions and propose future work.

Preliminaries

A *lattice* $L \subset \mathbb{R}^n$ is an additive discrete subgroup of $\mathbb{R}^n$ such that $L = \left\{ \sum_{i=1}^{d} x_i \underline{b}_i | x_i \in \mathbb{Z}, 1 \leq i \leq d \right\}$ with linearly independent vectors $\underline{b}_1, \ldots, \underline{b}_d \in \mathbb{R}^n (d \leq n)$. $B = (\underline{b}_1, \ldots, \underline{b}_d) \in \mathbb{R}^{n \times d}$ is the *lattice basis* of L with dimension d. The basis for a lattice is not unique. Different lattice bases B and B' for the same lattice L are related to each other

by means of a unimodular transformation U, i.e., $B' = BU$ with $U \in \mathbb{Z}^{d\times d}$ and $|\det(U)| = 1$. Unimodular transformations include the exchange of two base vectors, usually referred to as a swap, adding an integer multiple of a basis vector to another, usually referred to as a reduction or translation, and multiplying a basis vector by -1. The goal of lattice basis reduction is to start from an original lattice basis and reach a lattice basis whose vectors are relatively short and nearly orthogonal to each other via unimodular transformations.

The Gram Matrix G of a lattice L with basis $B = (\underline{b}_1, \ldots, \underline{b}_d) \in \mathbb{R}^{n\times d}$ is defined as $B^T B$.

Unlike the lattice basis, the *determinant* of a lattice is an invariant $\left(\det(L) = \sqrt{\det(B^T B)}\right)$, independent of the particular basis and has a natural geometric representation [5]. Its geometrical meaning is that it represents the n-dimensional volume of the parallelepiped whose edges are the basis vectors.

All lattice basis reduction algorithms are essentially based on the Gram–Schmidt orthogonalization (GSO) algorithm [5] or an equivalent technique like Cholesky factorization [16,17]. Let $\underline{b}_1, \ldots, \underline{b}_n$ be a basis of $\mathbb{R}^n$. The Gram–Schmidt orthogonalization of $\underline{b}_1, \ldots, \underline{b}_n$ produces the following basis $\underline{b}_1^*, \ldots, \underline{b}_n^*$:

$$\underline{b}_1^* = \underline{b}_1$$

$$\underline{b}_i^* = \underline{b}_i - \sum_{j=1}^{i-1} \mu_{i,j}\underline{b}_j^*, \quad (2 \le i \le n)$$

$$\mu_{i,j} = \frac{\underline{b}_i \cdot \underline{b}_j^*}{\underline{b}_j^* \cdot \underline{b}_j^*}, \quad (1 \le j < i \le n)$$

The $\mu_{i,j}$'s are called orthogonal projection coefficients [5].

The Gram–Schmidt basis vectors $\underline{b}_1^*, \ldots, \underline{b}_n^*$ are an orthogonal basis but usually not in the lattice generated by $\underline{b}_1, \ldots, \underline{b}_n$. In order to obtain a basis of the lattice we use variants of the Gram–Schmidt process resulting in vectors that are nearly orthogonal to each other.

In order to measure the quality of a lattice basis we use the notion of *orthogonality defect*. The orthogonality defect of a lattice basis $B = (\underline{b}_1, \ldots, \underline{b}_d) \in \mathbb{R}^{n\times d}$ defined as $\mathrm{dft}(B) = \frac{\prod_{i=1}^{d} \|\underline{b}_i\|}{\det(L)}$ allows one to compare the quality of different lattice bases. It holds that $\mathrm{dft}(B) \ge 1$ in general and $\mathrm{dft}(B) = 1$ for an orthogonal basis. The goal of lattice basis reduction is to algorithmically determine a basis with minimal defect. A basis is said to be LLL–reduced with parameters δ and η [14] if it satisfies the following conditions:

$$|\mu_{i,j}| \le \eta \quad \text{for } 1 \le j < i \le d,$$

referred to as the "size condition", and

$$|\underline{b}_i^* + \mu_{i,i-1}\underline{b}_{i-1}^*|^2 \ge \delta|\underline{b}_{i-1}^*|^2 \quad \text{for } 2 \le i \le d,$$

referred to as the "exchange condition".

The reduction parameter δ is a real number such that $\frac{1}{4} < \delta < 1$ with standard value $\delta = \frac{3}{4}$ and the parameter η satisfies $0.5 \le \eta < \sqrt{\delta}$.

Two fundamental problems related to the security of lattice based cryptosystems are the shortest vector problem (SVP) and the closest vector problem (CVP) [10].

The Shortest Vector Problem (SVP) Find a shortest nonzero vector in a lattice L, i.e., find a nonzero vector $\underline{v} \in L$ that minimizes the Euclidean norm $\|\underline{v}\|$.

The Closest Vector Problem (CVP) Given a vector $\underline{w} \in \mathbb{R}^n$ that is not in L, find a vector $\underline{v} \in L$ that is closest to $\underline{w}$, i.e. find a vector $\underline{v} \in L$ that minimizes the Euclidean norm $\|\underline{w} - \underline{v}\|$.

If a lattice $L \subset \mathbb{R}^n$ has a basis $\underline{b}_1, \ldots, \underline{b}_d$ consisting of vectors that are pairwise orthogonal, i.e. such that

$$\underline{b}_i \cdot \underline{b}_j = 0 \quad \text{for all } i \neq j,$$

then it is easy to solve both SVP and CVP ([10] p. 379). Thus the reduction algorithms lead to bases that are appropriate to approximate the hard problems on which the lattice based cryptosystems rely on.

A Novel Algorithm Initiated by the Buffered LLL Gram

The original LLL algorithm [14] is almost never used in practice. Instead one applies floating-point variants of LLL, where the long-integer arithmetic required by Gram–Schmidt orthogonalization is replaced by floating-point arithmetic [17]. The reason for this is because the running time of floating-point arithmetic versions [2,16,17,21] is improved compared to long-integer arithmetic.

Additionally, in recent lattice-basis-reduction algorithms, Gram–Schmidt orthogonalization is replaced by Cholesky factorization [16,17]. This is due to an improvement to the stability of the algorithm. An effort to achieve this has been proposed by Backes and Wetzel in 2007 [2]. Moreover, they have proposed the use of buffered transformations where the transformations are not applied directly to the lattice basis but are stored in a transformation matrix instead. This matrix is flushed, when its maximum entry exceeds a specific limit related to the bit size of the machine-type integers used (typically 32 or 64 bits).

In this paper we propose a novel algorithm borrowing ideas both from [16,17], using a Gram-matrix variant to avoid recomputing costly inner products as in [2,16,17,23]. We also reexamine the buffered transformations idea proposed by Backes and Wetzel [2].

Our Proposed Algorithm

Our proposed algorithm is Algorithm 1. In the pseudocode below, the symbol $\diamond$ means the floating point approximation using the specified precision to the operations performed in the following parentheses.

Algorithm 1: LLL_Gram(B, l, δ, η)

Input: Lattice Basis $B = (\underline{b}_1, \ldots, \underline{b}_d) \in \mathbb{Z}^{n\times d}$ with floating point precision l [12] and reduction parameters (δ, η) as defined in "Preliminaries" section.
Output: (δ, η)–LLL–Reduced lattice basis B.

(1) $G \leftarrow$ COMPUTE_GRAM(B)
(2) $G' \leftarrow$ APPROX_BASIS_GRAM(G, l)
(3) $r_{1,1} \leftarrow G'_{1,1}$, $\mu_{1,1} \leftarrow 1$, $F_r \leftarrow$ false, $F_c \leftarrow$ false.
(4) $i \leftarrow 2$
(5) while $(i \leq d)$ do
(6) $\quad \mu_{i,i} \leftarrow 1$
(7) $\quad$ for j from 1 to $i-1$ do
(8) $\quad\quad r_{i,j} \leftarrow G'_{j,i}$
(9) $\quad\quad$ for k from 1 to $j-1$ do
(10) $\quad\quad\quad r_{i,j} \leftarrow \diamond(r_{i,j} - \diamond(\mu_{j,k} \cdot r_{i,k}))$
(11) $\quad\quad \mu_{i,j} \leftarrow \diamond(\frac{r_{i,j}}{r_{j,j}})$
(12) $\quad$ for j from $i-1$ down to 1 do

```
(13)        if (|μ_{i,j}| > η) then
(14)            F_r ← true
(15)            b_i ← b_i − ⌈μ_{i,j}⌋ · b_j
(16)            REDUCE_GRAM(G, i, j, ⌈μ_{i,j}⌋)
(17)            if (|⌈μ_{i,j}⌋| > 2^{l/2}) then
(18)              F_c ← true
(19)            for t from 1 to j do
(20)              μ_{i,t} ← ◇(μ_{i,t} − ◇(⌈μ_{i,j}⌋ · μ_{j,t}))
(21)    if (F_r = true) then
(22)        APPROX_VECTOR_GRAM(G', G, l, i)
(23)        for j from 1 to i − 1 do /* Recompute Cholesky Factorization */
(24)            r_{i,j} ← G'_{j,i}
(25)            for k from 1 to j − 1 do
(26)              r_{i,j} ← ◇(r_{i,j} − ◇(μ_{j,k} · r_{i,k}))
(27)            F_r ← false
(28)    S_1^{(i)} ← G'_{i,i} /* Compute the S_j^{(i)}'s of the Cholesky factorization */
(29)    for j from 2 to i do
(30)        S_j^{(i)} ← ◇(S_{j−1}^{(i)} − ◇(μ_{i,j−1} · r_{i,j−1}))
(31)    r_{i,i} ← S_i^{(i)}
(32)    if F_c = true then
(33)        i ← max{i − 1, 2}
(34)        F_c ← false
(35)    else
(36)        k ← i
(37)        while((i > 1) AND (δ · r_{i−1,i−1} > S_{i−1}^{(i)}) do
(38)            b_i ↔ b_{i−1}
(39)            SWAP_GRAM(G, i)
(40)            SWAP_GRAM(G', i)
(41)            i ← i − 1
(42)        if (i ≠ k) then
(43)            if (i = 1) then
(44)              r_{1,1} ← G'_{1,1}
(45)              i ← 1
(46)            else
(47)              for t from 1 to i − 1 do
(48)                μ_{i,t} ← μ_{k,t}
(49)                r_{i,t} ← r_{k,t}
(50)              r_{i,i} ← S_i^{(k)}
(51)        i ← i + 1
```

The $\mu_{i,j}$'s in this algorithm correspond directly to the $\mu_{i,j}$'s used in the Gram–Schmidt orthogonalization (GSO). The $r_{i,j}$'s and the relation to the $\mu_{i,j}$'s are given by the following equations [16,17]:

$$r_{i,j} = < \underline{b}_i, \underline{b}_j > - \sum_{k=1}^{j-1} \mu_{j,k} \cdot r_{i,k}$$

and

$$\mu_{i,j} = \frac{r_{i,j}}{r_{j,j}}$$

The $S_j^{(i)}$'s are given by the following equations [16,17]:

$$S_j^{(i)} = \|\underline{b}_i\|^2 - \sum_{k=1}^{j-1} \mu_{i,k} \cdot r_{i,k}$$

and

$$S_i^{(i)} = \|\underline{b}_i^*\|^2 = r_{i,i}$$

where $\underline{b}_i^*$ corresponds to the i-th vector of the GSO.

We note that both the exact computation and the update of the $\mu_{i,j}$'s, $r_{i,j}$'s and $S_j^{(i)}$'s is important to ensure correctness of the algorithm. In our initial approach to achieve this, $r_{i,j}$'s and $S_j^{(i)}$'s have to be recomputed every time a reduction is performed, since the basis has been modified. However, we can implement an additional optimization in order to avoid recomputing these quantities needlessly. The optimization is performed as follows: We initially compute the $\mu_{i,j}$'s and $r_{i,j}$'s for $j \leq i$ (see lines 3–11 of Algorithm 1). We then proceed to size-reduce vector $\underline{b}_i$ while keeping only the $\mu_{i,j}$'s up-to-date (see lines 19–20). Once $\underline{b}_i$ has been size-reduced, we recompute the $r_{i,j}$'s (see lines 23–25) and $S_j^{(i)}$'s for $j \leq i$. That way, these quantities will be computed $O(1)$ times instead of $O(i)$ times, as would be the worst case of our initial approach. The computation of the $S_j^{(i)}$'s is stalled since they are not really needed until the loop condition of line (37).

The algorithm COMPUTE_GRAM(B) in Algorithm 1 simply computes the inner-products of the basis vectors for each pair resulting to the computation of the Gram matrix. Taking advantage of the symmetric property of the Gram matrix it suffices to compute and use only its upper triangular part. Also, the APPROX_BASIS_GRAM(G), creates an approximate version of the Gram matrix by converting its entries to floating point approximations. Finally, the APPROX_VECTOR_GRAM(G', G, k) algorithm updates the approximate Gram matrix G' after a vector has been size-reduced [2].

Next, we present the algorithm REDUCE_GRAM which is applied when the scalar multiplication of an integer with a basis vector is subtracted from another basis vector.

Algorithm 2: REDUCE_GRAM($G, i, j, \lceil\mu_{i,j}\rfloor$)

Input: Gram matrix $G \in \mathbb{Z}^{d\times d}$, the index i of the basis vector affected by the reduction, the index j of the basis vector whose scalar product with $\lceil\mu_{i,j}\rfloor$ is being subtracted from $\underline{b}_i$.
Output: The updated Gram matrix after the reduction.

(1) for t from 1 to $j-1$ do:
(2) $\quad G_{t,i} \leftarrow G_{t,i} - \lceil\mu_{i,j}\rfloor \cdot G_{t,j}$
(3) for t from j to $i-1$ do:
(4) $\quad G_{t,i} \leftarrow G_{t,i} - \lceil\mu_{i,j}\rfloor G_{j,t}$
(5) for t from $i+1$ to d do:
(6) $\quad G_{i,t} \leftarrow G_{i,t} - \lceil\mu_{i,j}\rfloor G_{j,t}$
(7) $G_{i,i} \leftarrow G_{i,i} - 2\cdot\lceil\mu_{i,j}\rfloor G_{j,i} - \lceil\mu_{i,j}\rfloor^2 \cdot G_{j,j}$

The Swap Gram Algorithm

The final algorithm we examine is the one used for the Gram matrix, when a swap is performed:

Algorithm 3: SWAP_GRAM(G, i)

Input: Gram matrix $G \in \mathbb{Z}^{d\times d}$, index i of the basis column vector swapped with its previous vector.
Output: Updated Gram matrix after swapping $\underline{b}_i$ and $\underline{b}_{i-1}$

(1) for t from 1 to $i-2$ do:
(2) $\quad G_{t,i-1} \leftrightarrow G_{t,i}$
(3) for t from $i+1$ to d do:

(4) $G_{i-1,t} \leftrightarrow G_{i,t}$
(5) $G_{i-1,i-1} \leftrightarrow G_{i,i}$

This algorithm is an adapted version of the algorithm in page 35 of [23]. We adapted S. Wetzel's algorithm in order to work with the upper triangular part of the Gram matrix.

For a short proof of why this algorithm correctly updates the Gram matrix see *Appendix*.

A Revised Buffered Transform Algorithm

In [2], Backes and Wetzel, introduced an algorithm that gradually performs the transformations needed to LLL-reduce a lattice basis by using an auxiliary matrix T as a buffer. The mathematical background behind this, is elementary (unimodular) matrices. In addition, a $\underline{T_max}$ vector is used for keeping track of the maximum entries of T and that is used to determine whether the buffer needs to be flushed or not. In our work we propose a revisited version of this algorithm, in Algorithm 4, shown below.

Furthermore, a number of modifications to Algorithm 1 are required in order to incorporate this technique. In line (15), the transformation: $\underline{b}_i \leftarrow \underline{b}_i - \lceil \mu_{i,j} \rfloor \cdot \underline{b}_j$ is buffered instead of being applied directly to B. In addition every time the vectors of B are interchanged, this transformation is also buffered. This is done by replacing $\underline{b}_i \leftrightarrow \underline{b}_{i-1}$ in line (38) with the swap operations: $\underline{t}_{i-1} \leftrightarrow \underline{t}_i$ and $\underline{T_max}_{i-1} \leftrightarrow \underline{T_max}_i$. Finally, the buffer needs to be flushed right before the termination of Algorithm 1, as it may still contain unapplied transformations.

The initializations required for Algorithm 1 to operate properly are as follows:

$$T \leftarrow I_d$$
$$\underline{T_max} \leftarrow (1, \ldots, 1)^T$$
$$pos_min \leftarrow d$$

Algorithm 4: BUFFER_TRANSFORMATION($B, T, \underline{T_max}, i, j, \lceil \mu_{i,j} \rfloor$)

Input: Lattice Basis $B = (\underline{b}_1, \cdots, \underline{b}_d) \in \mathbb{Z}^{n \times d}$, Transformation Buffer $T \in \mathbb{Z}^{d \times d}$, vector $\underline{T_max} \in \mathbb{Z}^d$, indices: i, j and $\lceil \mu_{i,j} \rfloor$
Output: Lattice Basis B

(1) if $(\underline{T_max}_i + |\lceil \mu_{i,j} \rfloor| \cdot \underline{T_max}_j) \geq 2^{k-1}$ then /* flush the buffer, where k represents the machine int limit */
(2) $C \leftarrow B$
(3) for x from pos_min to d do:
(4) for z from 1 to n do:
(5) $C_{z,x} \leftarrow 0$
(6) for y from pos_min to d do:
(7) if $(T_{y,x} \neq 0)$ then:
(8) if $(T_{y,x} = 1)$ then:
(9) for z from 1 to n do:
(10) $C_{z,x} \leftarrow C_{z,x} + B_{z,y}$
(11) else
(12) if $(T_{y,x} = -1)$ then:
(13) for z from 1 to n do:
(14) $C_{z,x} \leftarrow C_{z,x} - B_{z,y}$
(15) else
(16) for z from 1 to n do:
(17) $C_{z,x} \leftarrow C_{z,x} + B_{z,y} \cdot T_{y,x}$
(18) $B \leftarrow C$
(19) $T \leftarrow I_{d \times d}$

(20) for x from 1 to d do:
(21) $\underline{T_max}_i \leftarrow 1$
(22) $pos_min \leftarrow j$
(23) if $(|\lceil \mu_{i,j} \rfloor| \geq 2^{k-1})$ then:
(24) $\underline{b}_i \leftarrow \underline{b}_i - \lceil \mu_{i,j} \rfloor \cdot \underline{b}_j$
(25) return
(26) $\underline{t}_i \leftarrow \underline{t}_i - \lceil \mu_{i,j} \rfloor \cdot \underline{t}_j$
(27) $\underline{T_max}_i \leftarrow \underline{T_max}_i + |\lceil \mu_{i,j} \rfloor| \cdot \underline{T_max}_j$
(28) if $(j < pos_min)$ then:
(29) $pos_min \leftarrow j$ /* Update pos_min */

Experimental Results

Lattice Bases Generation

For our experiments we focus on unimodular lattice bases. The reason for this choice is that they require an unusually high number of size-reductions compared to other types of lattice bases [3,7]. In contrast to the experiments performed by [2] we allow the determinant of a lattice basis to have the value -1 apart from the unique value 1 to examine a broader class of unimodular lattices.

The techniques we have used are standard ones for generating unimodular lattice bases [2,8]. Specifically, we generate three types of unimodular matrices, from now on referred to as “Type A”, “Type B” and “Type C” accordingly based on the following formulas:

$$\text{“Type A”} : B = (L_1 \cdot U_1)$$
$$\text{“Type B”} : B = (L_1 P_1 \cdot U_1 P_2)$$
$$\text{“Type C”} : B = (L_1 P_1 \cdot U_1 P_2) \cdot (L_2 P_3 \cdot U_2 P_4)$$

where the L matrices are lower-triangular, the U matrices upper-triangular and the P matrices permutation matrices.

The explanation of the selection of these three specific types is as follows: The first category generates unimodular matrices with significantly “small” entries, thus examining the performance of the algorithms for “simple” cases. The other two categories use permutation matrices and matrix multiplication, to generate unimodular matrices with much larger entries that require significantly more unimodular operations to achieve an LLL-reduced basis.

Environment of the Experiments and Actual Experiments Performed

In our work we have chosen to compare four algorithms. First of all we have chosen to compare our proposed algorithm with the L^2 [16,17] algorithm by Nguyen and Stehlé. The first reason for this choice is that the L^2 algorithm gives the same approximation quality for an SVP as LLL. Secondly, L^2 has an improved worst case running time analysis (see [18] Ch. 10). Additionally, we have implemented two variants of the aforementioned algorithms using the buffered transformations heuristic [2].

We have implemented all the algorithms as Sage scripts, because we wanted to compare them under the same setting and also contribute to the Sage community. Since at this high level of coding, timings may not be so relevant, we also examine more qualitative properties. Thus, following the advice by McGeoch [15] we have compared code counters for dominant operations, which in our case are the number of reductions and deep insertions performed.

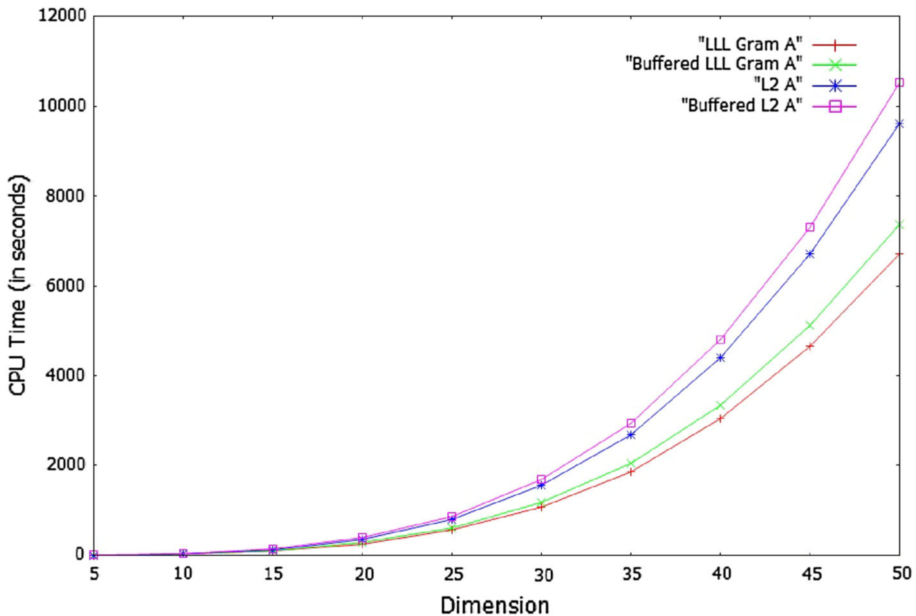

Fig. 1 Total CPU times for the four different algorithms executed for 1,000 "type A" bases

Our paper belongs to category c of [13], which means that our aim is to better understand the strengths, weaknesses and operation of interesting algorithmic ideas in practice, rather than an optimized implementation.

In order to ensure comparability, a pre-condition emphasized in [13], we have performed all our experiments on the same machine, namely a workstation with Intel Core i5 CPU, of 2.40 GHz clock speed and 4GB RAM.

The maximum bit–size of the matrix entries used was 80 bits.

We have compared the CPU times for 1,000 bases for each of the three types defined previously, and for each dimension in order to have a sufficiently large sample of lattice bases. We have not used average time but total execution time to compare the different algorithms. Average times can directly be derived from them.

Results of the Experiments

We have executed the four examined algorithms up to dimension 50, with parameters $(\delta, \eta) = (0.99, 0.51)$. The reason for focusing on small dimensions is that according to [16] the correctness proof given by Nguyen and Stehlé [16] holds for dimensions up to 50, for the specific parameters (δ, η) and quadruple precision (which is the highest precision for most programming languages). Since we have observed that there exists no major difference in the running times for consecutive dimensions we have chosen 5 as the dimension step. We did not examine total time, since the execution time apart from CPU-time for our experiments has proved to be negligible.

The results of the running times of the four algorithms for the three different lattice basis types are presented in Figs. 1–3, where the horizontal axis represents the currently examined dimension while the vertical axis illustrates the execution time in seconds.

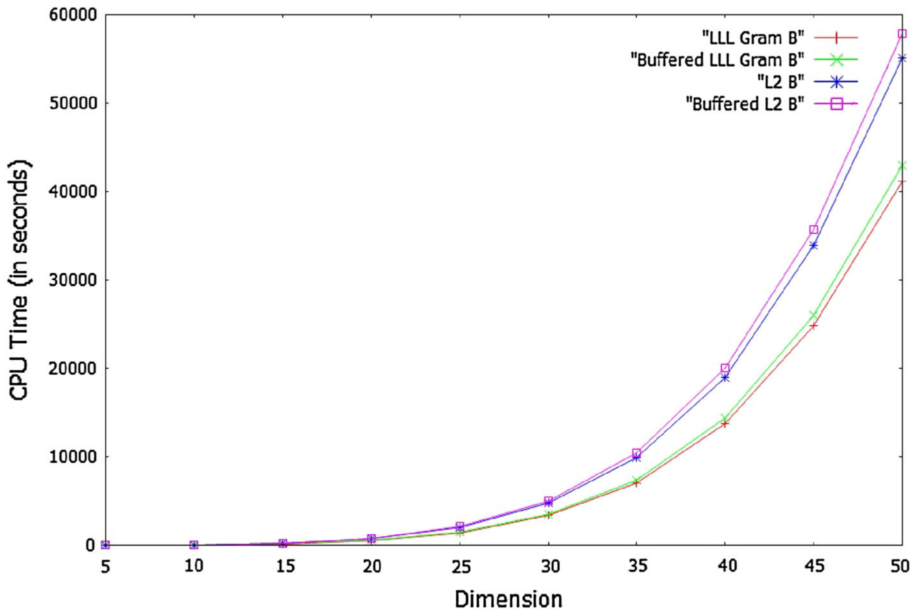

Fig. 2 Total CPU times for the four different algorithms executed for 1,000 "type B" bases

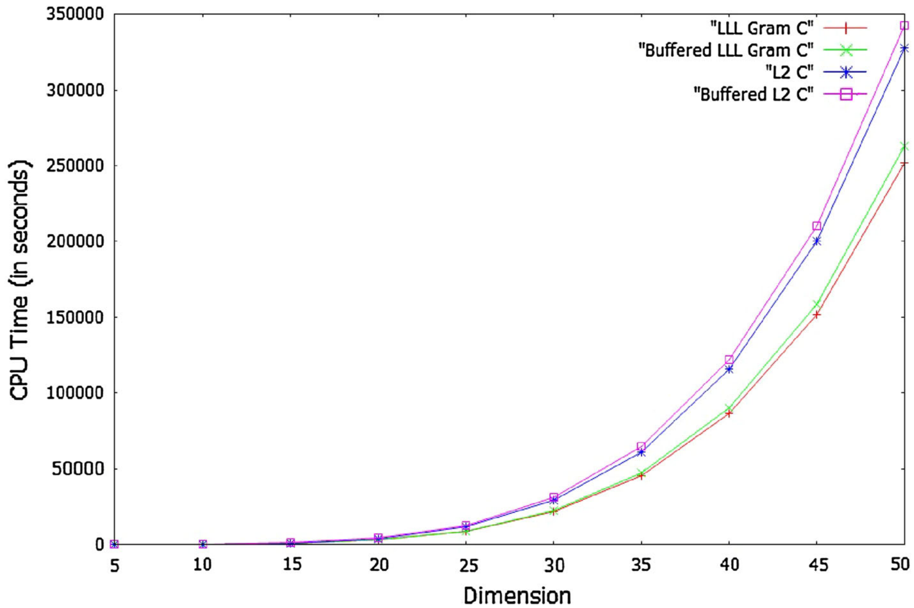

Fig. 3 Total CPU times for the four different algorithms executed for 1,000 "type C" bases

As we can easily see the running times for the three different types of bases, increases as we move from "Type A" to "Type B" and consecutively to "Type C". Accordingly, the number of reductions and to a lower extent the number of deep insertions shows a similar increase. This can be attributed to the larger entries of the bases for more complex types.

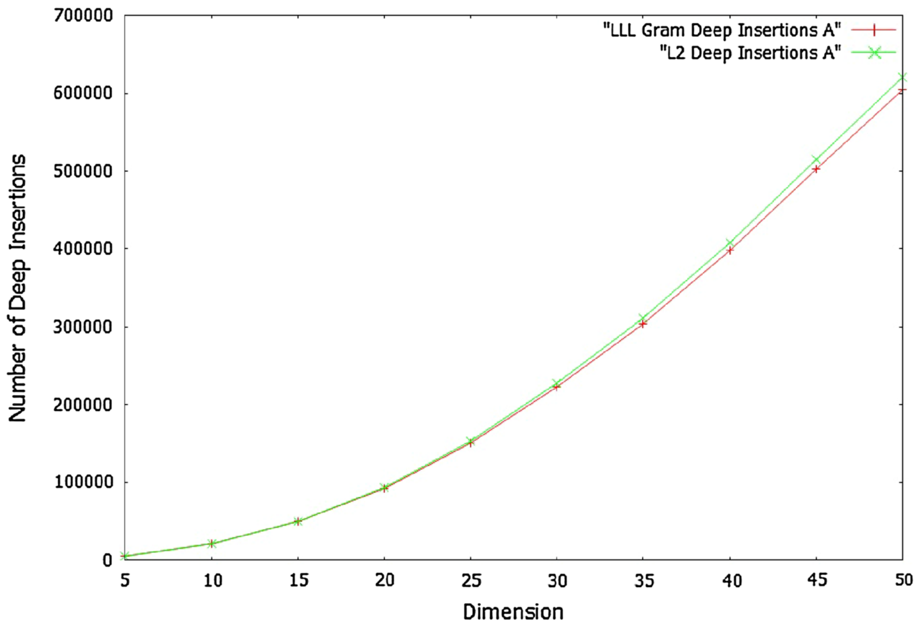

Fig. 4 Total number of deep insertions for the four different algorithms executed for 1,000 “type A” bases

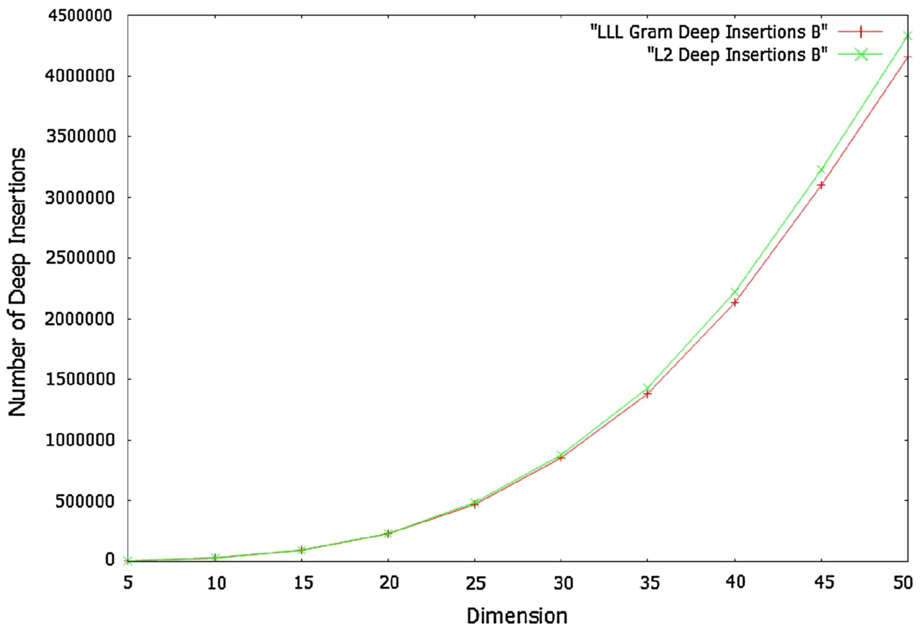

Fig. 5 Total number of deep insertions for the four different algorithms executed for 1,000 “type B” bases

From these figures we can observe that the buffered heuristic does not benefit either of these two algorithms. The second observation one can make is that the “LLL Gram” algorithm outperforms L^2 when unimodular lattices are examined for the

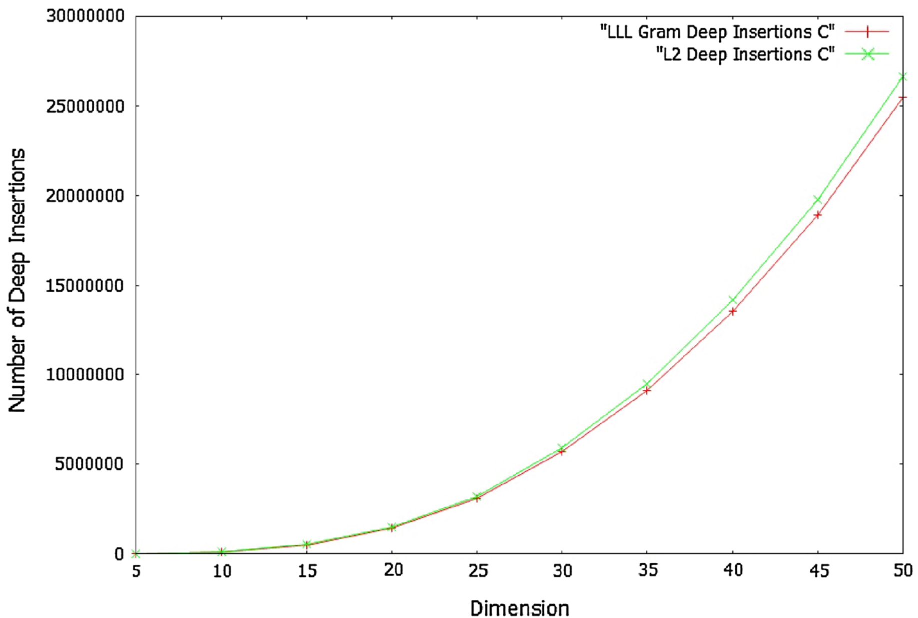

Fig. 6 Total number of deep insertions for the four different algorithms executed for 1,000 “type C” bases

specific dimensions. Additionally, the difference in CPU times widens as dimension increases.

The number of deep insertions for each type of lattice basis are presented in Figs. 4–6, while the number of reductions for the three different types of lattice bases are depicted in Figs. 7–9. Since the number of reductions and/or deep insertions required to obtain an LLL-reduced basis is not affected by whether they are buffered or not, we omit them from Figs. 4–9. Although the difference in the total number of deep insertions of the two algorithms is negligible, from Figs. 7–9 it is obvious that the number of reductions required to obtain an LLL-reduced basis using L^2 is overwhelming compared to the ones required by our algorithm. This becomes much more evident when comparing the number of reductions for the highest dimensions examined, where the difference exceeds 200 %. This can be attributed to the fact that the L^2 algorithm performs the reductions progressively, thus requiring many more reductions. This can be attributed to the use of “lazy size reduction” [16,17] of Babai’s Nearest Plane algorithm. The proposed algorithm performs significantly faster, due to the fact that it requires a much lower number of reductions.

Also, given that the total number of reductions required by the L^2 algorithm compared to our variant of Schnorr–Euchner is significantly bigger, we expected the “buffered” version of L^2 to have a better running time compared to its “raw” counterpart, since the buffer needs to be flushed more frequently. However, that is not the case, as Figs. 1–3 suggest.

Additionally, every time a deep insertion is performed both L^2 and our algorithm need to backtrack to the point of insertion in order to ensure correctness, which also adds an overhead of reductions needed to obtain an LLL-reduced basis. As Figs. 4–6 suggest, the difference in the number of deep insertions required between L^2 and our algorithm widens (albeit slowly) thus burdening L^2 with an additional number of reductions.

However, the L^2 algorithm is asymptotically better, as the complexity analysis of [16] proves.

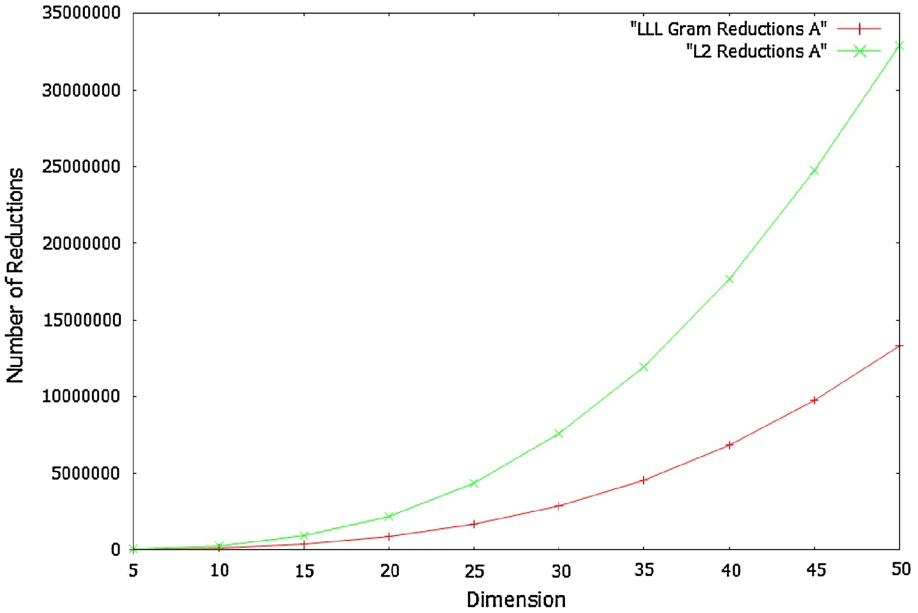

Fig. 7 Total number of reductions for the four different algorithms executed for 1,000 "type A" bases

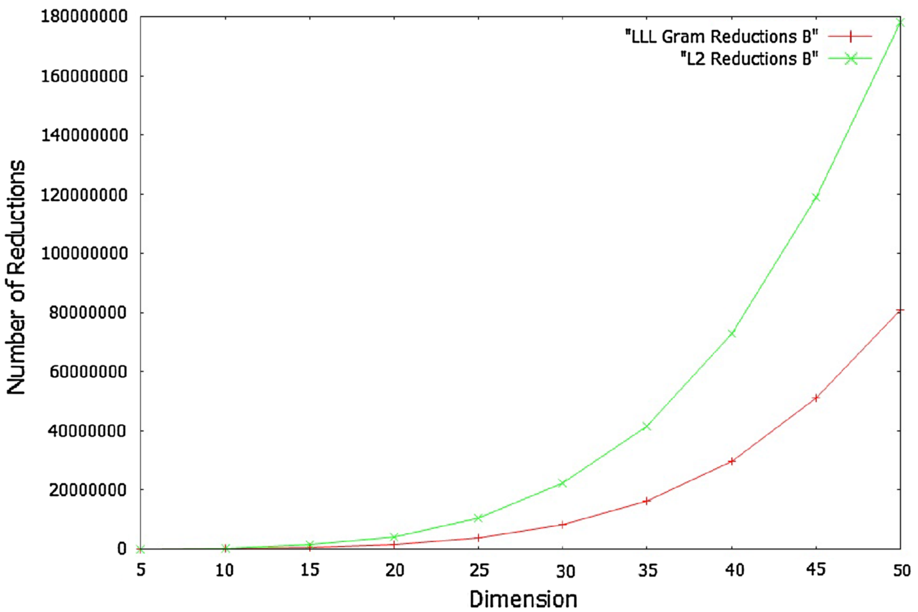

Fig. 8 Total number of reductions for the four different algorithms executed for 1,000 "type B" bases

Conclusions and Future Work

Our results show that our algorithm outperforms the L^2 algorithm for dimensions not exceeding the highest dimension available in order for the correctness proof by Nguyen and Stehlé

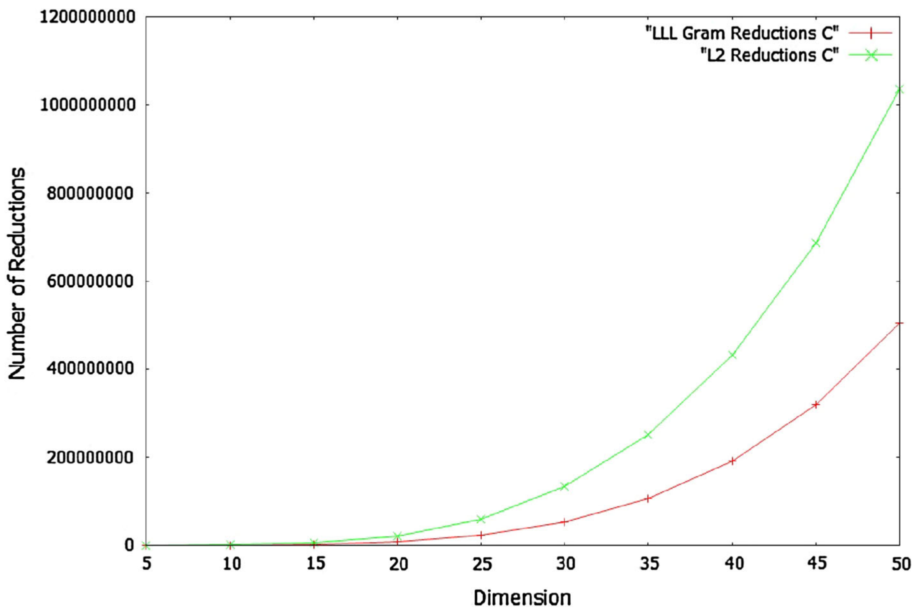

Fig. 9 Total number of reductions for the four different algorithms executed for 1,000 “type C” bases

to hold for quadruple precision. This does not cancel the effort to propose an algorithm that performs better asymptotically, since the L^2 algorithm could be more efficient if a better precision were available in standard programming languages (i.e. more than quadruple precision). As future work we would like to examine whether we could use fast matrix multiplication techniques used in the literature, in order to improve the results of the buffered transformations idea. Additionally, we would like to implement the quasi-linear time complexity reduction algorithm [19] and perform appropriate comparisons.

Acknowledgments The authors would like to thank Damien Stehlé, from ENS Lyon, France for his valuable answers to the questions they addressed to him.

Appendix: Proof of the update Gram matrix algorithm

In Algorithm 3 (SWAP_GRAM) we have provided the pseudocode that updates the Gram matrix when two consecutive basis vectors are swapped. In the following we prove the validity of the algorithm that properly updates the Gram matrix when a deep insertion is performed. This algorithm is simply a generalization of the SWAP_GRAM process.

Proof of Correctness of Algorithm 3 (SWAP_GRAM)

Let $\underline{b}_1, \underline{b}_2, \ldots, \underline{b}_{k-1}, \underline{b}_k, \underline{b}_{k+1}, \ldots, \underline{b}_{k'-1}, \underline{b}_{k'}, \underline{b}_{k'+1}, \ldots, \underline{b}_d$ be our initial lattice basis with a corresponding Gram matrix G.

Suppose we wish to perform a deep insertion of basis vector $\underline{b}_{k'}$ before basis vector $\underline{b}_k$, where $k \leq k^{'}$.

Let $\underline{b}_1^*, \underline{b}_2^*, \ldots \underline{b}_{k-1}^*, \underline{b}_k^*, \underline{b}_{k+1}^*, \ldots, \underline{b}_{k'-1}^*, \underline{b}_{k'}^*, \underline{b}_{k'+1}^*, \ldots, \underline{b}_d^*$ be the lattice basis after the deep insertion with a corresponding Gram matrix G^*. The following relations obviously hold for the new basis:

Fig. 10 The nine submatrices of the Gram matrix

$\underline{b}_i^* = \underline{b}_i, \forall i \notin \{k, k+1, \ldots, k^{'}\}, \underline{b}_k^* = \underline{b}_{k'}$ and $\underline{b}_i^* = \underline{b}_{i-1}, \forall i \in \{k+1, k+2, \ldots, k^{'}\}$

The above relations suggest that we should divide the Gram matrix into nine submatrices (see Fig. 10).

Lets consider arbitrary indices $i, j \in \{1, \ldots, d\}$.

Case 1: If $i \in \{1, 2, \ldots, k-1\} \bigcup \{k^{'}+1, k^{'}+2, \ldots, d\}$ and $j \in \{1, 2, \ldots, k-1\} \bigcup \{k^{'}+1, k^{'}+2, \ldots, d\}$ (submatrices 1–3–7–9) then $g_{i,j}^* = \underline{b}_i^* \cdot \underline{b}_j^* = \underline{b}_i \cdot \underline{b}_j = g_{i,j}$. So it becomes obvious that the entries of the submatrices 1–3–7–9 do not change.

Case 2: If $i \in \{1, 2, \ldots, k-1\} \bigcup \{k^{'}+1, k^{'}+2, \ldots, d\}$ and $k \leq j \leq k^{'}$ (submatrices 2 and 8) then $\underline{b}_i^* = \underline{b}_i$.

- if $j = k$, then $\underline{b}_j^* = \underline{b}_k^* = \underline{b}_{k'}$ and therefore: $g_{i,j}^* = \underline{b}_i^* \cdot \underline{b}_j^* = \underline{b}_i \cdot \underline{b}_{k'} = g_{i,k'}$.
- if $k < j \leq k^{'}$, then $\underline{b}_j^* = \underline{b}_{j-1}$ which implies that $g_{i,j}^* = \underline{b}_i^* \cdot \underline{b}_j^* = \underline{b}_i \cdot \underline{b}_{j-1} = g_{i,j-1}$.

Therefore, all entries in these submatrices horizontically shift by one column to the right cyclically in respect to their corresponding matrix row.

Case 3: If $k \leq i \leq k^{'}$ and $j \in \{1, 2, \ldots, k-1\} \bigcup \{k^{'}+1, k^{'}+2, \ldots, d\}$ (submatrices 4 and 6), then $\underline{b}_j^* = \underline{b}_i$.

- If $i = k$, then $\underline{b}_i^* = \underline{b}_k^* = \underline{b}_{k'}$ and therefore $g_{i,j}^* = \underline{b}_i^* \cdot \underline{b}_j^* = \underline{b}_{k'} \cdot \underline{b}_j = g_{k',j}$.
- If $k < i \leq k^{'}$, then $\underline{b}_i^* = \underline{b}_{i-1}$, which implies that $g_{i,j}^* = \underline{b}_i^* \cdot \underline{b}_j^* = \underline{b}_{i-1} \cdot \underline{b}_j = g_{i-1,j}$.

Therefore, all entries in these submatrices vertically shift by one row below cyclically in respect to their corresponding matrix column.

Case 4: If $k \leq i \leq k^{'}$ and $k \leq j \leq k^{'}$ (submatrix 5) then:

- If $i = j = k$, then $g_{i,j}^* = g_{k,k}^* = \underline{b}_k^* \cdot \underline{b}_k^* = \underline{b}_{k'} \cdot \underline{b}_{k'} = g_{k',k'}$.
- If $i = k$ and $k < j \leq k^{'}$ and $j = k$, then $g_{i,j}^* = \underline{b}_i^* \cdot \underline{b}_j^* = \underline{b}_k^* \cdot \underline{b}_j^* = \underline{b}_{k'} \cdot \underline{b}_{j-1} = g_{k',j-1}$.
- If $k < i < k^{'}$ and $j = k$, then $g_{i,j}^* = \underline{b}_i^* \cdot \underline{b}_j^* = \underline{b}_{i-1} \cdot \underline{b}_{k'} = g_{i-1,k'}$.
- If $k < i < k^{'}$ and $k < j < k^{'}$, then $g_{i,j}^* = \underline{b}_i^* \cdot \underline{b}_j^* = \underline{b}_{i-1} \cdot \underline{b}_{j-1} = g_{i-1,j-1}$.

Therefore, all entries are vertically shifted by one row and horizontally by one column, cyclically with respect to the bounds of submatrix 5. ◻

References

1. Ajtai, M., Dwork, C.: A public-key cryptosystem with worst-case/average-case equivalence. In: Proceedings of the 29th symposium on the theory of computing (STOC), pp. 284–293, ACM Press (1997)
2. Backes, W., Wetzel, S.: An efficient LLL Gram using buffered transformations. In: Proceedings of computer algebra in scientific computing, LNCS vol. 4770, pp 31–44 (2007)
3. Backes, W., Wetzel, S.: Heuristics on lattice basis in practice. J. Exp. Algorithmics **7**, 1–21 (2002)
4. Bernstein, D.J., Buchmann, J., Dahmen, E. (eds.): Post-Quantum Cryptography. Springer Verlag, Berlin-Heidelberg (2009)
5. Bremner, M.R.: Lattice Basis Reduction, An Introduction to the LLL Algorithm and its Applications. CRC Press, Boca Raton (2012)
6. Finch, C.: Sage Beginner's Guide. Packt Publishing, Birmingham (2011)
7. Hecker, C.: Automatische Parallelisierung und Parallelle Gitterbasisreduction, Ph.D. Thesis, Universität des Saarlandes, Saarbrücken (1994)
8. Generation of Unimodular Matrices with Bounded Elements, http://math.stackexchange.com/questions/336829/generation-of-unimodular-matrices-with-bounded-elements, (2014). Accessed 2014
9. Goldreich, O., Goldwasser, S., Halevi, S.: Public-key cryptosystems from lattice reduction problems. In: Proceedings of crypto 1997, LNCS, vol. 1294, pp. 112–131. Springer-Verlag (1997)
10. Hoffstein, J., Pipher, J., Silverman, J.H.: An Introduction to Mathematical Cryptography. Springer-Verlag, New York (2008)
11. Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: A ring based public key cryptosystem. In: Proceedings of the 3rd algorithmic number theory aymposium (ANTS III), LNCS, vol. 1423, pp. 267–288, Springer-Verlag (1998)
12. IEEE 754 Floating Point Standard, http://steve.hollasch.net/cgindex/coding/ieeefloat.html, (2014). Accessed 2014
13. Johnson, D.: A theoretician's guide to the experimental analysis of algorithms. Discret. Math. Theor. Comput. Sci. **59**, 215–250 (2001)
14. Lenstra, A., Lenstra, H., Lovász, L.: Factoring polynomials with rational coefficients. Math. Ann. **261**, 515–534 (1982)
15. McGeoch, C.C.: A Guide to Experimental Algorithmics. Cambridge University Press, Cambridge (2012)
16. Nguyen, P.Q., Stehlé, D.: An LLL algorithm with quadratic complexity. SIAM J. Comput. **39**(3), 874–903 (2009)
17. Nguyen, P.Q., Stehlé, D.: Floating-point LLL revisited, In: Cramer, R.J.F. (ed.), EUROCRYPT 2005, LNCS vol. 3494, pp 215–233, Springer, Heidelberg (2005)
18. Nguyen, P.Q., Vallée, B. (eds.): The LLL Algorithm. Survey and Applications. Springer Verlag, Heidelberg (2010)
19. Novocin, A., Stehlé, D., Villard, G.: An LLL-reduction algorithm with quasi-linear time complexity, In: Proceedings of the 43rd annual ACM symposium on theory of computing (2011)
20. http://www.python.org, Python home page, (2014). Accessed 2014
21. Schnorr, C., Euchner, M.: Lattice basis reduction: improved practical algorithms and solving subset sum problems. In: Budach, L. (ed.) FCT 1991, LNCS, vol. 529, pp. 68–85. Springer, Heidelberg (1991)
22. Stein W.A., et al.: Sage Mathematics Software (Version 5.13), The Sage Development Team, 2013, http://www.sagemath.org, (2013)
23. Wetzel, S.: Lattice basis reduction algorithms and their applications. PhD thesis, Universität des Saarlandes (1998)