Improvement to LLL Algorithm for γ-Approximate SVP in n-Dimensional Lattices

Naman Verma
Roll Number: 160428
Project Mentors: Professor SK Mehta, Mahesh Sreekumar Rajasree

CS498A, Odd Semester, 2018-19

Abstract

Lattices have become a topic of active research in Computer Science. They have many applications in cryptography and crypt-analysis. One of the most important problems related to Lattices is the Shortest Lattice Vector Problem (SVP). In this report, we propose certain improvements to the well-known LLL Algorithm, which tackles a variation of SVP: γ-Approximate SVP.

1 Lattices

In Chemistry, a lattice is defined as a regularly repeated three-dimensional arrangement of atoms, ions, or molecules in certain solids. This definition can be extended to general n dimensions. An n-dimensional Lattice is a periodically repeating n-dimensional arrangement of points in $\mathbb{R}^n$. In more precise terms, an n-dimensional lattice $\mathcal{L}$ can be defined as follows:

$$\mathcal{L} = \left\{ \sum_{i=1}^{K} a_i \mathbf{b}_i | a_i \in \mathbb{Z} \right\}, \text{ where } B = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_K\},\ \mathbf{b}_i \in \mathbb{R}^n, \text{ is called the generating set.}$$

In this report, it is assumed that the generating set is a set of linearly independent vectors. So, B can be called the **Basis** of $\mathcal{L}$. A lattice can have infinite such Bases.

Another way to define lattices is as follows:

DEFINITION: An n-dimensional lattice $\mathcal{L}$ is a discrete additive subgroup of $\mathbb{R}^n$. [1]

More recently, lattices have become a topic of active research in computer science due to their applications in cryptography, and their unique properties from a computational complexity point of view.

One of the most important problems related to Lattices is the Shortest Lattice Vector Problem (SVP). In this report, we first define the SVP, followed by its approximate variation, γ-Approximate SVP, in Section 2, followed by an overview of the well-known Lenstra-Lenstra-Lovász (LLL) Algorithm in Section 3. Then, we will propose a heuristic which adds to the power of the LLL Algorithm

and explain it in Section 4. We describe the experiments that we've run to test this heuristic in Section 5, followed by the results of the experiments and their inferences in Section 6.

2 Shortest Vector Problem (SVP)

The Shortest Lattice Vector Problem has a simple statement: Given a Lattice $\mathcal{L}$, find the smallest non-zero lattice vector in $\mathcal{L}$. For a Lattice $\mathcal{L}$, it can be stated as:

$$\boldsymbol{\lambda} = \mathbf{v} \in \mathcal{L} \backslash \{\mathbf{0}\} \text{ , such that } ||\mathbf{v}|| \text{ is minimum}$$

Finding the smallest vector is an NP-hard problem.

Another version of this problem is **Approximate SVP**.

2.1 γ Approximate SVP

Given a lattice $\mathcal{L}$ with basis $B = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n\}$, $\mathbf{b}_i \in \mathbb{R}^n$, find a non-zero vector $\mathbf{v} \in \mathcal{L}$ such that $0 < ||\mathbf{v}|| \leq \gamma ||\boldsymbol{\lambda}||$, where $\boldsymbol{\lambda}$ is the shortest vector in the lattice. For certain values of γ, it can be shown to be solvable in polynomial time. For $\gamma = 2^{\frac{n-1}{2}}$, the **LLL Algorithm** yields a polynomial time solution.

3 Lenstra–Lenstra–Lovász (LLL) ALgorithm[2]

DEFINITION: Given n linearly independent vectors $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n \in \mathbb{R}^n$, their **Gram-Schmidt orthogonalization** is defined by $\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \mathbf{b}_j$, where $\mu_{i,j} = \dfrac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle}$.

Let a Basis be $\mathbf{B} = \{\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_{n-1}\}$ and its Gram-Schmidt Orthogonal(GSO) basis be $\mathbf{B}^* = \{\mathbf{b}_0^*, \mathbf{b}_1^*, \ldots, \mathbf{b}_{n-1}^*\}$.

The GSO coefficients are: $\mu_{i,j} = \dfrac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle}$

This basis $\mathbf{B}$ is said to be `LLL-Reduced` if it satisfies the following conditions for some real number $\delta \in (0.25, 1]$:

1. $\mu_{i,j} \leq 0.5 \ \ \forall$ valid values of i, j. (Size reduced)
2. $\delta ||\mathbf{b}_{k-1}^*||^2 \leq ||\mathbf{b}_k^*||^2 + \mu_{k,k-1}^2 ||\mathbf{b}_{k-1}^*||^2 \quad$ for $1 \leq k < n$ (Lovász condition)

In an `LLL-Reduced` Basis, the following holds:

$$||\mathbf{b}_0|| \leq \Big(\frac{2}{\sqrt{4\delta - 1}}\Big)^{n-1} ||\boldsymbol{\lambda}||, \text{ where } \boldsymbol{\lambda} \text{ is the shortest non-zero vector in } \mathcal{L}$$

Now, the LLL Algorithm takes a Lattice Basis $\mathbf{B}'$ as input, and converts it into a basis $\mathbf{B}$ such that $\mathbf{B}$ is `LLL-Reduced` and $\mathbf{B}$ generates the same lattice as $\mathbf{B}'$. Hence, we can also see this as follows: for an input basis $\mathbf{B}'$, the LLL Algorithm returns a vector $\mathbf{b}_0 \in \mathbf{B}$ which satisfies

$||\mathbf{b}_0|| \leq \left(\frac{2}{\sqrt{4\delta - 1}}\right)^{n-1} ||\boldsymbol{\lambda}||$, that is, it returns a vector $\mathbf{b}_0$ which is the solution to the γ-Approximate SVP with the approximation factor being $\left(\frac{2}{\sqrt{4\delta - 1}}\right)^{n-1}$.

When the founders of this algorithm gave this result, they used the value $\delta = \frac{3}{4}$. For this value of delta δ, we get the $\gamma = 2^{\frac{n-1}{2}}$ approximation.

Note that, higher the value of δ, the better is the approximation. However, the time taken by the algorithm increases as the value of δ increases. The exact relation is:

$$\text{Time} \propto \frac{1}{\log(\frac{1}{\delta})}$$

4 Improvements to LLL

We put forward a heuristic that improves the performance of the LLL algorithm. Algorithm 1 describes this heuristic at a high level. It uses the helper function REDUCE, which is defined in Section 4.1, and another helper function CONVERGED, defined in Section 4.2.

We know that the LLL Algorithm reduces the given Lattice Basis into another Basis which is `LLL-Reduced`. Now, if we add some changes into this output basis such that it is no longer `LLL-Reduced`, another run of the LLL algorithm may give us a basis which has an even shorter vector than before.

What the REDUCE function does is that it 'dis-balances' the Basis so that it is not `LLL-Reduced` anymore. Moreover, it changes the basis in such a way that all the basis vectors are at a bigger angle from each other than before. This is indeed a desirable property to ensure that a Lattice Basis has short vectors in it. We describe what the REDUCE function does in the proceeding subsection.

Indeed we see that when we run the LLL Algorithm on this dis-balanced Basis once again, that run gives an even shorter of the lattice than before. We keep on doing this until this dis-balancing doesn't give any better results. At that point the algorithm is said to have *converged*.

Algorithm 1 Modified LLL

Input: Lattice Basis $\mathbf{B} = [\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_{n-1}]$

1: $\mathbf{B}' \leftarrow \text{LLL}(\mathbf{B})$
2: **while NOT** CONVERGED$(\mathbf{B}', \mathbf{B})$ **do**
3: $\quad \mathbf{B} \leftarrow \mathbf{B}'$
4: $\quad \mathbf{B}' \leftarrow$ REDUCE$(\mathbf{B})$
5: $\quad \mathbf{B}' \leftarrow \text{LLL}(\mathbf{B}')$.
6: **end while**

4.1 REDUCE Function

Suppose we're given an n-dimensional lattice basis $\mathbf{B}$, in which the last vector, $\mathbf{b}_{n-1}$, is fixed. Our goal is to find an alternating basis $\mathbf{B}'$ of the same lattice which contains $\mathbf{b}_{n-1}$, but the hyperplane formed by the remaining $n-1$ vectors of $\mathbf{B}'$ has a normal that makes a smaller angle with $\mathbf{b}_{n-1}$ as compared to the hyperplane formed by the remaining vectors in $\mathbf{B}$.

Intuitively, we can do this by subtracting a certain (integral) component of $\mathbf{b}_{n-1}$ on each of the remaining vectors from themselves. The way we proceeded is as follows: for all $i = 0, 1, \ldots, n-2$, do the following:

1. We have a fixed vector $\mathbf{b}_{n-1}$ and a currently chosen vector, say $\mathbf{b}_i$. The remaining $n-2$ vectors form a vector space S. Find out the perpendicular component vectors of $\mathbf{b}_{n-1}$ and $\mathbf{b}_i$ on S, and call these $\mathbf{g}_{n-1}$ and $\mathbf{g}_i$ respectively.

2. Update $\mathbf{b}_i$ as follows:
$$\mathbf{b}_i = \mathbf{b}_i - \left\lfloor \frac{\langle \mathbf{g}_{n-1}, \mathbf{g}_i \rangle}{\langle \mathbf{g}_{n-1}, \mathbf{g}_{n-1} \rangle} \right\rceil \times \mathbf{b}_{n-1}$$
What's happening here is the following: we find the projection-ratio of $\mathbf{g}_i$ on $\mathbf{g}_{n-1}$, round it off to the nearest integer to get an integer η, multiply the rounded off value to $\mathbf{b}_{n-1}$ to get the vector $\eta\mathbf{b}_{n-1}$, and subtract this from $\mathbf{b}_i$.

Now, what the REDUCE function does is as follows: it fixes $\mathbf{b}_{n-1}$ and modifies the aforementioned hyperplane as mentioned in the above two steps. Then, it goes into the modified sub-lattice of these new $n-1$ vectors, fixes $\mathbf{b}_{n-2}$ and changes the sub-lattice formed by the remaining $n-2$ vectors. Its keeps on going down recursively until $\mathbf{b}_0$ is the fixed vector.

Algorithm 2 describes how we get the new hyperplane, and Algorithm 3 describes the REDUCE function.

4.2 Convergence

We mentioned before that when the dis-balancing provided by the REDUCE function doesn't give any better results, we break from the while loop in Algorithm 1, and that is when we say that the algorithm has converged. Now, one way of seeing this is that if a particular step doesn't give a shorter smallest vector, then we have converged. However, in our testings, we found out that

Algorithm 2 REDUCE-PERP

Input: Lattice Basis $\mathbf{B} = [\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_{n-1}]$

1: **for** i `in` $0 \ldots (n-2)$ **do**
2: $S = \text{PLANE}(\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_{i-1}, \mathbf{b}_{i+1}, \ldots \mathbf{b}_{n-2})$ `//Sub-plane formed by the input vectors`
3: $\mathbf{g}_i = \text{PERP-COMP}(\mathbf{b}_i, S)$ `//perpendicular component of` $\mathbf{b}_i$ `on` S
4: $\mathbf{g}_{n-1} = \text{PERP-COMP}(\mathbf{b}_{n-1}, S)$
5: $\eta = \text{ROUNDOFF}\big(\text{DOTP}(\mathbf{g}_i, \mathbf{g}_{n-1}) / \text{NORM-SQUARE}(\mathbf{g}_{n-1})\big)$
6: $\mathbf{b}_i = \mathbf{b}_i - \eta \times \mathbf{b}_{n-1}$
7: **end for**

Algorithm 3 REDUCE function

Input: Lattice Basis $\mathbf{B} = [\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_{n-1}]$

$m \leftarrow n$
while $m \geq 0$ **do**
  $\mathbf{B}' \leftarrow \mathbf{B}[0 : m]$
  $\mathbf{B}' \leftarrow$ REDUCE-PERP($\mathbf{B}'$). //See Algorithm 2
  $m \leftarrow m - 1$.
end while

checking the smallest vector's norm alone is not enough. It might happen that the shortest norm in the Basis doesn't change, but other vectors might do, which gives improvements in a later step. Hence, we need to check the other vectors in the Basis as well.

An easy way to incorporate all the norms into consideration without adding any major overheads is to just compare the change in sum of norms of all the vectors between the previous Basis $\mathbf{B}$ and the new Basis $\mathbf{B}'$. Algorithm 4 describes this simple step of both checking the shortest norm and the sum of norms.

Algorithm 4 CONVERGED function

Input: New Basis $\mathbf{B}' = [\mathbf{b}'_0, \mathbf{b}'_1, \ldots, \mathbf{b}'_{n-1}]$, Old Basis $\mathbf{B} = [\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_{n-1}]$

$s \leftarrow$ SHORTEST-NORM($\mathbf{B}$) //returns the norm of the shortest vector in the input Basis
$s' \leftarrow$ SHORTEST-NORM($\mathbf{B}'$)
$sum \leftarrow$ SUM-NORMS($\mathbf{B}$) //returns the sum of norms of all vectors in the input Basis
$sum' \leftarrow$ SUM-NORMS($\mathbf{B}'$)
if $s' > s$ **AND** $sum' > sum$ **then**
  return False
else
  return True
end if

4.3 Expected Improvements in Results

The shortest vector found using LLL Algorithm depends upon the δ used in the Lovász Condition. Higher the delta, smaller is the resultant vector. However, the time taken by the algorithm increases as the value of δ increases.

It is expected that the for a particular value of δ, the modified LLL Algorithm will give a way smaller shortest vector than the original LLL Algorithm. Moreover, the time it takes to give this smaller vector should be less than the time taken by the original LLL with higher values of delta.

5 Experiments

The two goals of the experiment are to test the modified LLL in these two conditions:

1. Original LLL vs Modified LLL with same $\delta = 0.251$: Both the Algorithms run till termination.
2. Original LLL with $\delta = 0.9$ vs Modified LLL with $\delta = 0.251$: The former runs till termination, while the latter either stops when it finds the smallest vector found by the former, or if it converges before finding that vector.

Given a Lattice Basis, we compare the performance of original and modified conditions in both the experiments. In the first one, the primary concern is that the modified LLL should show improvement in the resultant smallest vector, while in the latter, the concern is to get the same smallest vector with the modified version taking lesser time.

The Lattice basis used in these experiments were of this type:

$$\begin{bmatrix} x_0 & 0 & 0 & 0 & \dots & 0 \\ x_1 & 1 & 0 & 0 & \dots & 0 \\ x_2 & 0 & 1 & 0 & \dots & 0 \\ x_3 & 0 & 0 & 1 & \dots & 0 \\ & \dots & \dots & & \ddots & \\ x_{n-1} & 0 & 0 & 0 & \dots & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_0^T \\ \mathbf{b}_1^T \\ \mathbf{b}_2^T \\ \mathbf{b}_3^T \\ . \\ . \\ \mathbf{b}_{n-1}^T \end{bmatrix}$$

The zeroth coordinate of $\mathbf{b}_i$ is x_i, and for $i \neq 0$, the ith coordinate of $\mathbf{b}_i$ is 1. All the other coordinates are $0 \; \forall \; i$. For all i, x_i had approximately 240 digits.

These tests were run on 600 different Lattices. Of these 600 Lattices, there were 100 Lattices each of dimensions $20, 22, 24, 26, 28, \& \; 30$. All these Lattices had basis of the form mentioned above.

5.1 Code

This experiments were carried out in C++ using the `gmpxx` library. This library provides various functions to handle very large numbers, be it integers, rationals, or floating point numbers. This helped in doing arithmetic operations involving the 240 digit x_i mentioned above.

The code used to run this experiment can be found here: https://github.com/namanv3/Lattices. This repository has a detailed readme describing all the files in the repo, and has all the instructions for running and testing the code.

5.2 Lattice Basis Instances

These Lattice Bases were created using the generator from the SVP Challenge Website:
https://www.latticechallenge.org/svp-challenge/

6 Results of The Experiment

6.1 Original LLL vs Modified LLL with same $\delta = 0.251$

As expected, the improved LLL Algorithm gives huge improvements in the resultant shortest vector. The table below quantifies how much better the improved version performs and how much more time in takes on average.

In the table below, $\eta_{original}$ means the shortest norm found by the original LLL Algorithm and $\eta_{improved}$ means the shortest norm found by the modified LLL Algorithm.

Table 1: Comparison of Original LLL vs Modified LLL with same $\delta = 0.251$.

Dimension	**Average Improvement** $\left(\frac{\eta_{original}}{\eta_{improved}}\right)$	**Average Speed-down** $\left(\frac{\text{average time taken by original LLL}}{\text{average time taken by modified LLL}}\right)$
20	17.602	0.300
22	30.100	0.262
24	50.245	0.222
26	73.363	0.188
28	99.461	0.172

6.2 Original LLL with $\delta = 0.9$ vs Improved LLL with $\delta = 0.251$

In this test, say the Original LLL with $\delta = 0.9$ returns $\mathbf{v}$ as the shortest vector it could find in the Lattice formed by the input basis. Our objective is to check if the modified LLL Algorithm with $\delta = 0.251$ can find $\mathbf{v}$ in lesser time than the original LLL Algorithm. Hence, we add another condition for termination of the original LLL for this test. The improved LLL Algorithm terminates in these two cases:

1. It finds the smallest vector found by the original LLL with $\delta = 0.9$.
2. It is unable to find that vector because it converges before doing so.

These conditions for termination help in observing as to how much speedup the improved LLL gives with weaker δ value as compared to the original LLL with stronger δ.

The table below quantifies how much better the improved version performs. Success percentage refers to the number of times the improved LLL was able to find the vector found by the original LLL with stronger δ.

Table 2: Comparison of Original LLL with $\delta = 0.9$ vs Improved LLL with $\delta = 0.251$

Dimension	Success Percentage	Average Speedup		Average Ratio of Shortest Norms in Failures
		Overall	Successful Cases	
20	90%	2.590	2.620	1.065
22	80%	2.530	2.545	1.263
24	74%	2.399	2.392	2.200
26	59%	2.233	2.247	2.059
28	46%	2.099	1.947	9.431

7 Conclusions and Further Improvements

The improved LLL algorithm shows promising results. It greatly increases the power of lower δ-valued LLL algorithm, and while increasing the power, it takes lesser time than higher δ-valued pure LLL algorithm.

We also observe that as the number of Basis vectors increase (that is, as the dimensions of the Lattice increase), the modified LLL Algorithm performs even more better than the original LLL algorithm for the same δ value.

On the path to reach this modification to the LLL Algorithm, we tried various other heuristics as well, each having a particular intuition or the other. That being said, there are many ways in which the work done here can be built upon.

1. Firstly, we can find a more robust criteria for convergence.
2. Secondly, there are many libraries heavily system-optimised implementations of the LLL Algorithm available for various languages. Implementing the REDUCE function at such levels of optimisation can certainly provide a clearer picture as to how much less time does the modified LLL take.

Appendix: Code Snippets

Below are the implementations of the algorithms defined in this report. These functions are of the class `Lattice`.

```
class Lattice {
  private:
    int n, m;
  public:
    mpz_class original[MAXSIZE][MAXSIZE];          //MAXSIZE is a marco variable
    mpz_class B[MAXSIZE][MAXSIZE];
    Lattice(int rows, int cols);
    .
    .
    .
    mpz_class shortest();
    mpz_class sum_of_norms();
    void LLL (int deltaNum, int deltaDen);
    void reduce_perp(int numRows, int i);
    void reduce(int numRows);
    void reduce_main();
    void newLLL (int deltaNum, int deltaDen);

};
```

In all the snippets below, various functions are used that are not a part of the `Lattice` class, but have been defined in `functions.h` file in the github repository mentioned in Section 5.1. All the functions can't be written here as the report might get too long, so we've described the relevant functions in the comments.

A1. REDUCE-PERP

```
void Lattice::reduce_perp (int numRows, int i) {
    mpz_class val, temp[MAXSIZE];
    mpq_class proj, num, den;
    mpq_class G[numRows][MAXSIZE], qtemp[m];

    gso(B, G, numRows, m);                                  // G has the GSO Basis of B

    //projection of B[numRows-1] on G[numRows-2] stored in qtemp
    projection (B[numRows-1], G[numRows-2], qtemp, m);
    //sum of G[numRows-1] and qtemp stored in G[numRows-1]
    addvector(G[numRows-1], qtemp, G[numRows-1], m);

    num = dotp(G[numRows-1], G[numRows-2], m);
    // returns ||G[numRows-1]||^2
    den = norm(G[numRows-1], m);
    // takes a rational number as input and rounds it to the nearest integer
    val = roundtoInt(num * mpq_class(den.get_den(), den.get_num()));
    // multiplies the integer val to the vector B[numRows-1] and stores the product
    in temp
    intmult(B[numRows-1], val, temp, m);
    // subtracts B[numRows-2] by temp, stores the result in B[numRows-2]
    subvector(B[numRows-2], temp, B[numRows-2], m);
}
```

A2. REDUCE

```
void Lattice::reduce (int numRows) {
    mpq_class dist, val;

    for (int j = 0; j < 2; j++) {
        for (int i = 0; i < numRows-1; i++) {
            swapArr(B[numRows-2], B[i], m);
            reduce_perp(numRows, i);
            swapArr(B[numRows-2], B[i], m);
      }
    }
    return;
}

void Lattice::reduce_main() {
    for(int i = 0; i < n-1; i++) {
        reduce(n-i);
    }
    return;
}
```

A3. Modified LLL

```
void Lattice::newLLL (int deltaNum, int deltaDen) {
  mpz_class shortnorm,prev;
  mpz_class new_sum, prev_sum;
  shortnorm = shortest();

  LLL(deltaNum, deltaDen);

  shortnorm = prev = shortest();
  prev_sum = sum_of_norms();
  while (1) {
    reduce_main();

    LLL(deltaNum, deltaDen);

    shortnorm = shortest();
    new_sum = sum_of_norms();
    if (new_sum >= prev_sum && shortnorm >= prev) {
      cout << "No improvement, breaking from newLLL.." << endl;
      break;
    }
    prev_sum = new_sum;
    prev = shortnorm;
  }

}
```

References

[1] Chris Peikert *Lattices in Cryptography,* Lecture 1, Georgia Tech, Fall 2013

[2] Oded Regev *Lattices in Computer Science,* Lecture 2, Tel Aviv University, Fall 2004