UNIVERSITY OF OREGON

Sieving Algorithms for Lattice Problems

by

Matthew Jagielski

A thesis submitted to
Computer and Information Science
University of Oregon
in partial fulfillment for the
degree of Bachelor of Science
supervised by Xiaodi Wu

June 2016

Acknowledgements

I would like to acknowledge Xiaodi Wu for guiding me through the process of learning and writing. He has invested a lot of time helping prepare me for graduate school.
I would also like to thank all my teachers and professors for helping me become the student I am, and my friends and family for shaping me into the person I am.

Contents

Chapter 1

Introduction

A lattice is a mathematical object which takes a set of vectors in $\mathbb{R}^n$ and combines them in all possible integer linear combinations. In this way, they form a regularly spaced, infinite subset of $\mathbb{R}^n$. Mathematicians and computer scientists have been studying lattices for hundreds, if not thousands, of years [1–5]. Some uses of lattices that have made them interesting have been their use in integer programming, coding theory, polynomial factoring, and some areas in pure math including group theory and number theory. Recently, however, interest in these objects has been increased significantly, thanks to the advent of quantum computation.

Quantum computation weakens the security of commonly used cryptographic protocols, and lattices are thought to be able to strengthen this security. A common method is RSA encryption, which relies upon the hardness of factoring large integers. In 1994, Peter Shor published an algorithm which can factor an integer in polynomial time using quantum computing. In order to circumvent this, one must find a new problem that can not be solved efficiently with quantum computers. Some of these problems that are being investigated are lattice problems.

There are a couple lattice problems that, as of now, seem to be resistant to quantum attacks, as well as NP-hard on classical computers. Two such problems are the shortest vector problem and the closest vector problem. The shortest vector problem is to find the shortest nonzero vector in a lattice. The closest vector problem is to find the closest vector in a lattice to some target point. These are being increasingly more well studied, and are the two problems I will discuss in this document, with reference to algorithms developed to solve them.

Sieving methods and Voronoi cell preprocessing are two methods that have been used to solve these problems, and are the ones discussed in this document. Others are enumeration based techniques, which generate large amounts of lattice vectors until the goal vector is found [1, 2]. Another type that will not be discussed is a recent method called Discrete Gaussian Sampling [6, 7]. This new method has improved runtime over the other methods.

This document will contain preliminaries and follow with the two types of algorithms -sieving and Voronoi cell preprocessing. The preliminaries will introduce lattices in more formality, as well as provide some basic and important algorithms on lattices. It will follow with some history of the problems. The sieving chapter will contain the original algorithm presented with analysis, then proceed to discuss extensions that have been made to improve running time. The Voronoi cell preprocessing chapter will contain the original problem with proof and some analysis of runtime and correctness.

As far as we know these days, the exact shortest vector and closest vector problems can only be solved in exponential time. Every algorithm for these problems, presented or omitted from this document, achieve no better than 2^n time. Even quantum algorithms achieve no significant speedup for this problem, and so, as of now, these problems seem to be candidates for quantum resistant cryptographic protocols.

Chapter 2

Preliminaries

We want to start by defining some preliminaries that will be required to understand the later chapters. We will start with definitions, follow with important algorithms, and finish with a brief history of the computational landscape of the important problems.

2.1 Lattices

Definition 2.1. A rank d *lattice* $\mathcal{L} \subset \mathbb{R}^n$ is the set of all integer linear combinations of d linearly independent vectors from a basis $B = \{b_1, b_2, \cdots b_d\}, b_i \in \mathbb{R}^n$,. Then

$$\mathcal{L}(B) = \left\{ \sum_{1}^{d} a_i b_i \mid a_i \in \mathbb{Z} \right\}. \tag{2.1}$$

We may also write a basis as a matrix

$$\begin{bmatrix} | & | & \cdots & | \\ \boldsymbol{b}_1 & \boldsymbol{b}_2 & \cdots & \boldsymbol{b}_n \\ | & | & \cdots & | \end{bmatrix}. \tag{2.2}$$

We denote the lattice constructed from a basis B as $\mathcal{L}(B)$. Furthermore, if B contains n vectors, we call $\mathcal{L}(B)$ full rank. In this document, we will assume lattices are full rank.

Example 2.1. *Consider $\mathbb{Z}^2$ as an example of a lattice.*

This is a rank 2 lattice in $\mathbb{R}^2$, so it is a full rank lattice. An example of a basis for $\mathbb{Z}^2$ is $B_0 = \left\{ \boldsymbol{e}_0 = [1,0]^T, \boldsymbol{e}_1 = [0,1]^T \right\}$. Then any $[a,b]^T \in \mathbb{Z}^2 = a\boldsymbol{e}_0 + b\boldsymbol{e}_1$.

However, also note that any $[a,b]^T$ can also be written as $(a-2b)\,[3,1]^T+(-a+3b)\,[2,1]^T$, so another basis for $\mathbb{Z}^2$ is $B_1 = \left\{[3,1]^T, [2,1]^T\right\}$. This illustrates an interesting fact - lattice bases aren't unique. This is important for the rest of the paper. Changing between bases can make solving some problems more efficient.

We also want to define a couple other mathematical objects that each lattice has, as they will come up later in this paper.

Definition 2.2. For any lattice $\mathcal{L}$, we define

$$\lambda_1(\mathcal{L}) = \{\|v\| \mid \|v\| \leq \|x\|, v, x \in \mathcal{L}\backslash 0\}, \tag{2.3}$$

$$\lambda_i(\mathcal{L}) = \inf\{r \mid \dim(\text{span}(\mathcal{L} \cap B(0,r))) \geq i\}. \tag{2.4}$$

The first value is the length of a shortest non-zero vector $v \in \mathcal{L}$. The second value is a generalized way of talking about shortest vectors. In this way, $\lambda_i(\mathcal{L})$ is the radius of the smallest ball containing i linearly independent vectors.

Example 2.2. *Let us reconsider $\mathbb{Z}^2$ using this new concept.*

If we look at the first basis we came up with, $B_0 = \left\{\boldsymbol{e}_0 = [1,0]^T, \boldsymbol{e}_1 = [0,1]^T\right\}$, it is clear that both $\boldsymbol{e}_0, \boldsymbol{e}_1$ are shortest vectors of $\mathbb{Z}^2$.
If, instead, the goal was to find the shortest vector of the lattice $\mathcal{L}(B_1)$, where $B_1 = \left\{[3,1]^T, [2,1]^T\right\}$, the calculation becomes less trivial. Instead, we use the basis $B_2 = \left\{[571,209]^T, [418,153]^T\right\}$, and this problem becomes almost unapproachable. This is despite the fact that B_0, B_1, and B_2 are all bases for Z_2. This problem will be revisited several times during this document.

Definition 2.3. The fundamental parallelepiped of a lattice

$$\mathcal{P}(\mathcal{L}(B)) = \left\{\sum_1^n x_i \boldsymbol{b}_i \mid 0 \leq x_i < 1, \boldsymbol{b}_i \in B\right\}. \tag{2.5}$$

Note that this definition depends on the basis used to construct the lattice, while $\lambda_1(\mathcal{L})$, for example, depends only on the lattice.

Example 2.3. *We revisit $\mathbb{Z}^2$ with the new definition.*

We see that $\mathcal{P}(\mathcal{L}(B))$ is just the unit square: $[0,1) \times [0,1)$. If instead we used B_1, we would get a parallelogram. An interesting thing to note is that the volume of the fundamental parallelepiped does not depend on the basis used even though the shape does. On top of this, if we view the basis as a matrix, its determinant is actually the volume of the fundamental parallelepiped.

2.2 Lattice Problems

Two important problems on lattices are the search shortest vector problem (SVP) and the search closest vector problem (CVP).

Definition 2.4. The search shortest vector problem (SVP) - given an input of a lattice basis B, output a lattice vector $\boldsymbol{v} \in \mathcal{L}(B), \|\boldsymbol{v}\| = \lambda_1(\mathcal{L}(B))$.

We can also parameterize the shortest vector problem in order to only require an approximate solution. The language for this follows:

Definition 2.5. The approximate search problem γ-SVP, where $\gamma = \gamma(n) > 1$ is an approximation factor where we want to take the same input as SVP and output an approximate shortest vector, some $\boldsymbol{v} \in \mathcal{L}(B), \|\boldsymbol{v}\| \leq \gamma\lambda_1(\mathcal{L}(B))$.

Definition 2.6. The search closest vector problem, is a problem to, given an input of a lattice basis B and a target vector $\boldsymbol{t}$, output $\boldsymbol{v} \in \mathcal{L}(B), \forall \boldsymbol{x} \in \mathcal{L}(B), \|\boldsymbol{v} - \boldsymbol{t}\| \leq \|\boldsymbol{x} - \boldsymbol{t}\|$, so that $\boldsymbol{v}$ is closer to $\boldsymbol{t}$ than any other vector in $\mathcal{L}(B)$.

In the same way as with SVP, we can parameterize CVP in the following way:

Definition 2.7. The approximate search problem γ-CVP, with $\gamma = \gamma(n) > 1$ an approximation factor takes as input a lattice basis and a target vector $\boldsymbol{t}$. It will instead output $v \in \mathcal{L}(B), \forall x \in \mathcal{L}(B), \|v - t\| \leq \gamma \|x - t\|$.

If these problems seem to be very similar, they are. Most algorithms in this paper have been developed for one or the other and can, with modifications, be applied to another. In addition, there is a polynomial reduction from CVP to SVP.

Theorem 2.8. *Given an oracle for CVP, we can, using polynomially many queries to the oracle, solve SVP.*

Proof. Suppose we are tasked with solving SVP on a basis $B = \{\boldsymbol{b}_1, \boldsymbol{b}_2, \cdots, \boldsymbol{b}_n\}$. Then define $B_i = \{\boldsymbol{b}_1, \cdots 2\boldsymbol{b}_i, \cdots, \boldsymbol{b}_n\}$ so all basis vectors are the same as those for B, except $\boldsymbol{b}_i$ is replaced by $2\boldsymbol{b}_i$. Then we compute CVP on each B_i with target $\boldsymbol{b}_i$, calling the output $\boldsymbol{v}_i$. We will return the shortest vector $\boldsymbol{v}_i - \boldsymbol{b}_i$.
First, observe that $\boldsymbol{b}_i$ cannot be in $\mathcal{L}(B_i)$. If it were, then $\boldsymbol{b}_i \in \mathcal{L}(B/\boldsymbol{b}_i)$. But this cannot happen as then B would be a linearly dependent set. So we can proceed.
Write the shortest vector of the lattice $\boldsymbol{v} = \sum_{i=1}^{n} c_i \boldsymbol{b}_i \in \mathcal{L}(B)$. We want to show that at least one of the $\boldsymbol{v}_i$ satisfies $\boldsymbol{v}_i - \boldsymbol{b}_i = \boldsymbol{v}$. In order to do this, we want to show that, if c_i is odd, then $\boldsymbol{v}_i = \boldsymbol{v} + \boldsymbol{b}_i$.
First, note that if c_i is odd, $\boldsymbol{v} + \boldsymbol{b}_i \in \mathcal{L}(B_i)$. The coefficient of $\boldsymbol{b}_i$ in $\boldsymbol{v} + \boldsymbol{b}_i$ will be $c_i + 1$, which is even. Then $\boldsymbol{v} + \boldsymbol{b}_i \in \mathcal{L}(B_i)$. Now, suppose there is some other vector $\boldsymbol{w} \in \mathcal{L}(B_i)$ closer to $\boldsymbol{b}_i$. Then

$$\|\boldsymbol{w} - \boldsymbol{b}_i\| < \|(\boldsymbol{v} + \boldsymbol{b}_i) - \boldsymbol{b}_i\| = \|\boldsymbol{v}\|.$$

But $\boldsymbol{w}$, which is in $\mathcal{L}(B)$, has smaller norm than our shortest lattice vector, and is nonzero (as $\boldsymbol{b}_i \notin \mathcal{L}(B_i)$). Then we have a contradiction and we can calculate $\boldsymbol{v} = \boldsymbol{v}_i - \boldsymbol{b}_i$ if c_i odd. All c_i cannot be even, as then $\frac{1}{2}\boldsymbol{v} \in \mathcal{L}(B)$, which would contradict $\boldsymbol{v}$ being the shortest lattice vector. □

2.3 Basis algorithms

Suppose we have a lattice vector $\boldsymbol{v} \in \mathcal{L}(B)$, and we want to know its coefficients as vectors in B. This is a pretty fundamental computation for a lattice - as one can imagine, several algorithms (including LLL and AKS) do this many times. It would be appealing to work in a basis where this computation is efficient. In order to make this an efficient operation, an algorithm for the computation is needed. Treating the basis B as a matrix, the vector of coefficients $\boldsymbol{c}$ is just the solution to $B\boldsymbol{c} = \boldsymbol{v}$. So we just need to calculate the inverse of the basis matrix and calculate $\boldsymbol{c} = B^{-1}\boldsymbol{v}$.
One way that is known to make a calculation like this more efficient is by using a QR decomposition. The Gram-Schmidt process takes in a basis and returns a new basis from which the original basis can be produced by multiplication by an upper diagonal matrix, mimicking the QR decomposition.

2.3.1 Gram-Schmidt Orthogonalization

The Gram-Schmidt orthogonalization of a set of linearly independent vectors, such as a lattice basis $B = \{\boldsymbol{b}_1, \boldsymbol{b}_2, \cdots, \boldsymbol{b}_n\}$, is an output $\tilde{B} = \{\tilde{\boldsymbol{b}}_1, \tilde{\boldsymbol{b}}_2, \cdots, \tilde{\boldsymbol{b}}_n\}$ where all $\tilde{\boldsymbol{b}}_i$ are mutually orthogonal. The Gram-Schmidt orthogonalization is constructed using the Gram-Schmidt process, described in Algorithm 1.

Algorithm 1 The Gram-Schmidt Process

function PROJ$_{\boldsymbol{u}}(\boldsymbol{v})$ **return** $\frac{\boldsymbol{u}\cdot\boldsymbol{v}}{\boldsymbol{u}\cdot\boldsymbol{v}}\boldsymbol{u}$
end function
function GRAM-SCHMIDT(B)
  for $i = 1 \cdots n$ **do**
    $\tilde{\boldsymbol{b}}_i = \boldsymbol{b}_i$
    for $j = 0 \cdots i-1$ **do** $\tilde{\boldsymbol{b}}_i = \tilde{\boldsymbol{b}}_i - \text{proj}_{\boldsymbol{b}_j}\boldsymbol{b}_i$
    end for
  end for
  return $\{\boldsymbol{b}_i\}$
end function

The idea behind this algorithm is to incrementally build a basis by extracting the orthogonal part of each basis vector when compared against the span of all previously used vectors. Note, however, that the values of each $\frac{\boldsymbol{u}\cdot\boldsymbol{v}}{\boldsymbol{u}\cdot\boldsymbol{v}}$ might not be integers, in which case the lattice for the Gram-Schmidt orthogonalization of a basis will not be the same as the original lattice.

One interesting thing about this construction is that the basis transformation from $\tilde{B}$ to B is the matrix

$$G = \begin{bmatrix} 1 & \text{proj}_{b1}(\boldsymbol{b}_2) & \text{proj}_{b1}(\boldsymbol{b}_3) & \cdots & \text{proj}_{b1}(\boldsymbol{b}_n) \\ 0 & 1 & \text{proj}_{b2}(\boldsymbol{b}_3) & \cdots & \text{proj}_{b2}(\boldsymbol{b}_n) \\ 0 & 0 & 1 & \cdots & \text{proj}_{b3}(\boldsymbol{b}_n) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}.$$

This means that if we treat these bases as matrices as in 2.2, we have $B = \tilde{B}G$ This is an upper triangular matrix, with all diagonal entries 1. Then $\det(B) = \det(\tilde{B}G) = \det(\tilde{B})\det G = \det(\tilde{B})$. But then consider the parallelepiped for $\mathcal{L}(\tilde{B})$. The basis $\tilde{B}$ is an orthogonal basis, which means that the parallelepiped looks just like a rectangular prism in $\mathbb{R}^n$, and has volume $\prod_{i=1}^{n} \left\|\tilde{\boldsymbol{b}}_i\right\|$. Because the determinants of $\tilde{B}$ and B are equal, any basis of $\mathcal{L}(B)$ will have this volume.

Theorem 2.9. *Given a lattice $\mathcal{L}(B)$, we can bound the length of the shortest vector from below as:* $\lambda_1(\mathcal{L}(B)) \geq \min_i \left\|\tilde{\boldsymbol{b}}_i\right\|$.

Proof. Consider a vector $\boldsymbol{v} \in \mathcal{L}(B)/0$. Let $\tilde{B}$ be the Gram-Schmidt orthogonalization of B. For each $1 \leq i \leq n$, consider v projected onto $\tilde{\boldsymbol{b}}_i$.
This projection could be one of the hyperplanes $a \cdot \tilde{\boldsymbol{b}}_i + \mathcal{L}(\tilde{B}/\tilde{\boldsymbol{b}}_i), a \in \mathbb{Z}$. If it is, then $\|\boldsymbol{v}\| \geq a \cdot \left\|\tilde{\boldsymbol{b}}_i\right\|$.
If this projection is 0, then we can project the lattice onto $\tilde{B}/\tilde{\boldsymbol{b}}_i$. Then recursively consider the Gram-Schmidt orthogonalization of this new lattice and our new v. Project this new v on another one of these Gram-Schmidt vectors and continue on smaller and smaller subspaces until we find a nonzero projection.
One of these projections is not zero, or we have a zero vector. Then $\|\boldsymbol{v}\| \geq \min_j \left\|\tilde{\boldsymbol{b}}_j\right\|$.
□

2.3.2 LLL basis

While the Gram-Schmidt process provides some information to help handle the shortest vector problem, just performing Gram-Schmidt doesn't provide too much help in actually computing more accurate answers to SVP or CVP. Fortunately, Lenstra, Lenstra, and Lovász presented an algorithm for turning an arbitrary lattice basis into one with smaller vectors [1]. This will provide an approximate answer to SVP, and will also be a good starting point for other algorithms later in the paper.

To start with, consider a modified Gram-Schmidt orthogonalization process which saves the scaling factors in each step, presented in Algorithm 2.

Algorithm 2 The Modified Gram-Schmidt Process

```
function MODIFIED GRAM-SCHMIDT(B)
    for i = 1 ··· n do
        b̃_i = b_i
        for j = 0 ··· i − 1 do
            μ_{i,j} = (b_i · b_j) / (b_j · b_j)
            b̃_i = b̃_i − μ_{i,j} b_j
        end for
    end for
    return {b̃_i}
end function
```

In this way, each $\boldsymbol{b}_i = \mu_{i,1}\boldsymbol{b}_1 + \cdots + \mu_{i,i-1}\boldsymbol{b}_{i-1} + \tilde{\boldsymbol{b}}_i$.

Definition 2.10. Let $\delta \in [\frac{1}{4}, 1]$. A basis $B = [\boldsymbol{b}_1, \boldsymbol{b}_2, \cdots, \boldsymbol{b}_n]$ is called δ-*LLL reduced* if

1. $\forall 1 \leq i < j \leq n, \mid \mu_{i,j} \mid \leq \frac{1}{2}$,
2. $\forall 1 \leq i < j \leq n-1 \left\|\tilde{\boldsymbol{b}}_{i+1}\right\|^2 \geq (\delta - \mu_{i+1,i}^2) \left\|\tilde{\boldsymbol{b}}_i\right\|^2$.

The algorithm for constructing an LLL basis is presented in Algorithm 3.

Algorithm 3 The LLL Basis Reduction Algorithm

```
function LLL(B = b_1, b_2, ··· , b_n)
    while True do
        B̃ =Gram-Schmidt(B)
        for i = 2 ··· n do
            for j = i − 1 ··· 1 do
                b_i = b_i − int(μ_{j,i}) b_j
            end for
        end for
        if ∃i, ‖b̃_i + 1‖ < √(δ − μ²_{i+1,i}) ‖b̃_i‖ then
            Swap b_i, b_{i+1}
        else
            return B = b_1, b_2, ··· , b_n
        end if
    end while
end function
```

Theorem 2.11. *The LLL basis reduction algorithm presented in Algorithm 3 provides a $2^{O(n)}$ approximation to the shortest lattice vector in polynomial time.*

Termination: In order to prove that the LLL algorithm terminates, we define a function $\phi_i(B)$:

$$\phi_i(B) = \prod_{j=1}^{i} \left\|\tilde{\boldsymbol{b}}_j\right\|. \tag{2.6}$$

We also define $\phi(B)$ as follows:

$$\phi(B) = \prod_{i=1}^{n} \phi_i(B). \tag{2.7}$$

This is a potential function - we want to see how $\phi(B)$ changes as the LLL basis reduction algorithm is run on B. When we start the algorithm on B, we have $\phi(B) = \prod_{i=1}^{n} \prod_{j=1}^{i} \left\|\tilde{\boldsymbol{b}}_j\right\| \leq \max_i \left\|\tilde{\boldsymbol{b}}_i\right\|^{O(n^2)}$.
We also want to bound this function from below. We know $\phi_n(B) = \prod_{i=1}^{n} \left\|\tilde{\boldsymbol{b}}_i\right\| = \det(B)$. But B is a linearly independent set (so determinant is nonzero) and B is

an integer lattice, so the determinant must be an integer. Then $|\det(B)| \geq 1$, and therefore $\phi(B) \geq 1$.

Now we consider what happens after each step of the computation. When we are subtracting basis vectors from each other to reduce their size, the Gram-Schmidt vectors do not change. This is because we are only subtracting vectors that appeared earlier in the basis. Then $\phi(B)$ does not change in this step.

In the next step, where we swap basis vectors, then $\phi(B)$ will change. But if $\boldsymbol{b}_i$ is swapped with $\boldsymbol{b}_{i+1}$, then the only factor $\phi_j(B)$ that will change will be $\phi_i(B)$. This is because either one of two things will happen in the other two cases: $\phi_j(B)$ has neither $\left\|\tilde{\boldsymbol{b}}_i\right\|$ or $\left\|\tilde{\boldsymbol{b}}_{i+1}\right\|$ as factors and so this swap will leave the product invariant, or $\phi_j(B)$ has both $\left\|\tilde{\boldsymbol{b}}_i\right\|$ and $\left\|\tilde{\boldsymbol{b}}_{i+1}\right\|$ as factors and so the swap will leave the product invariant. Then if before the swap our basis is B_0 and after it is B_1, then

$$\frac{\phi_i(B_0)}{\phi_i(B_1)} = \frac{\left\|\tilde{\boldsymbol{b}}_i\right\|}{\left\|\tilde{\boldsymbol{b}}_{i+1}\right\|}.$$

but the condition on the two swapping is that $\left\|\tilde{\boldsymbol{b}}_{i+1}\right\| < \sqrt{\delta - \mu_{i,i+1}^2}\left\|\tilde{\boldsymbol{b}}_i\right\|$. Then

$$\frac{\phi_i(B_0)}{\phi_i(B_1)} > \frac{1}{\sqrt{\delta - \mu_{i,i+1}^2}} > \frac{1}{\sqrt{\delta}}.$$

Now, $\delta < 1$ by assumption, so $\phi(B)$ decreases by a factor of $\sqrt{\delta}$ at each swap, and we have $1 \leq \phi(B) \leq \max_i \left\|\tilde{\boldsymbol{b}}_i\right\|^{O(n^2)}$, so there are polynomially many swaps that can happen: the maximum number of swaps should be $\frac{O(n^2)\log \max_i \left\|\tilde{\boldsymbol{b}}_i\right\|}{\log \sqrt{\delta}}$, which is polynomial in the size of the input. Then the algorithm will run in polynomial time.

Correctness: After the program terminates, we have some guarantees about what the result of the algorithm will be. We know that there have been no swaps, so all vectors are in the index they were in before the current iteration. Also, we know that all steps of the form $\boldsymbol{b}_i = \boldsymbol{b}_i - \text{int}(\mu_{j,i})\boldsymbol{b}_j$ have made it so that $\mu_{j,i} \leq \frac{1}{2}$, as the integer part has been subtracted. The other reason is that j is decremented from $i-1$ to 1, so $\mu_{j,i}$ will not be modified by earlier subtractions (recall that $\tilde{\boldsymbol{b}}_k$ is constructed by subtracting only vectors of the form $\tilde{\boldsymbol{b}}_m, m < k$).

2.4 History of Lattice Problems

Lattice problems have been researched for quite a long time. Thousands of years ago, Euclid came up with the Euclidean algorithm, which can be viewed as a solver for SVP in $\mathbb{Z}$. When we are considering algorithms that can be applied to any $\mathbb{R}^n$, however, the Euclidean algorithm is not very helpful. Thankfully, many others have contributed to improving solutions to the problem:

In 1982, Lenstra, Lenstra, and Lovász published their paper presenting the LLL algorithm for basis reduction [1]. I have already presented the algorithm and its analysis, so I only wish to highlight here that it is a polynomial time algorithm that can solve approximate $2^{n/2}$-SVP. This is not a very strict approximation, but the algorithm is very significant as the best approximation that can be done to SVP in polynomial time. It is also useful for cryptographers - knowing what approximation factors can be computed in polynomial time helps construct cryptosystems.

Perhaps the longest standing application of the LLL algorithm is its use as a basis reduction algorithm. It returns a basis that satisfies certain 'niceness' properties, which I have already described in 2.10. To review in plain language, the new basis vectors are in ascending order of size and there is a lower bound on the angle between any two basis vectors. This makes it so they can be approximately orthogonal. These properties, along with only a polynomial running time, make LLL a very reasonable preprocessing step in other algorithms which seek a better approximate solution or an exact solution to SVP. In fact, the analysis of both algorithms presented in following chapters relies on the basis being LLL-reduced.

The first method intended to solve exact SVP and CVP was presented in Kannan's 1987 paper - 'Minkowski's convex body theorem and integer programming.' [2]. It is an enumeration based technique, which means that it generates many lattice vectors with the goal of finding the shortest or closest to a target. Because it is exact, we can't expect it to run as fast as LLL, and it doesn't - it runs in $n^{O(n)}$ time but polynomial in space. This is a large runtime and has been surpassed by many algorithms, but it is still a significant algorithm. If not for its historical significance as the first exact lattice problem solver, it is still the fastest polynomial space algorithm for solving SVP/CVP. The algorithms that surpass it in running time all run in $2^{O(n)}$ time and space. It remains an open question whether an algorithm exists for exact SVP or CVP that runs

in $2^{O(n)}$ time but only uses $2^{o(n)}$ space.

Another method of solving SVP has been with sieving algorithms. The idea behind a sieving algorithm is to randomly select lattice vectors, then compare them (often by subtracting close vectors to get smaller vectors) in order to end up getting the shortest lattice vector after running the algorithm for many steps. The first of this type of algorithm was the Ajtai-Kumar-Sivakumar (AKS) algorithm.

The AKS algorithm was first published in 2000 [3], with a proven running time of $2^{5.9n} = 2^{O(n)}$ and requiring $2^{2.95n} = 2^{O(n)}$ space. The constants for the complexity of the original AKS algorithm weren't calculated in the original paper, but Nguyen and Vidick managed to prove the above running time [8]. The demonstrated running time is quite large, but over the years, there have been many improvements made to the algorithm to reduce its time and space requirements. The results have had enormous speedups, although they still all run in $2^{O(n)}$ time and space with much smaller constants in the exponent.

One notable improvement to AKS was Micciancio and Voulgaris's ListSieve and GaussSieve algorithms [9]. The ListSieve Algorithm can be proven to run in $2^{3.199n}$ time and $2^{1.325n}$ space, a huge improvement over AKS, but still exponential time and space. GaussSieve goes farther than ListSieve's improvements, at the cost of not being able to prove runtime. GaussSieve heuristically reduces the running time to $2^{0.48n}$ and space of $2^{0.18n}$, but is, again, not proven. It seems to be much better in practice than previous algorithms, however. Another method that built upon this was using locality sensitive hashing.

Locality sensitive hashing is a hashing technique that will, with some probability, associate two vectors 'close' to each other to the same hash value. It has been applied to image recognition and clustering for data analysis in the past (see [10]), and Laarhoven had the idea to apply it to lattice vectors for sieving based algorithms [11]. Using this technique, we can search the list more quickly by only comparing a vector to other vectors with the same hash value. This faster searching technique reduces the heuristic (but not proven) sieving algorithm runtime to $2^{0.3366n+o(n)}$ and space complexity to $2^{0.2075n+o(n)}$. This increases the space complexity from GaussSieve slightly (we need to store hash tables), but reduces the runtime. We've reduced the exponent of the runtime by a factor of over 17x and space's by a factor of over 14x from the original AKS algorithm. However, let's also look at other techniques aside from sieving.

One other technique was using Voronoi cells for solving CVP. This algorithm was published in 2009 by the same Micciancio and Voulgaris as published ListSieve and GaussSieve [4]. The algorithm is technical, but the general idea is that there is a way to reduce a CVP computation to several CVP computations in lower dimension, and using some preprocessing, we can do these smaller CVP computations more quickly, ending in a CVP runtime of 4^n. Note that this is a deterministic runtime, the fastest current deterministic algorithm for CVP or SVP. Recall that SVP can be solved in polynomially many CVP calls 2.8, so this is also a 4^n runtime for SVP as well.

Chapter 3

AKS Sieving Algorithm

In this chapter, we will start by introducing some basic ideas necessary for the AKS Sieving algorithm. Next, we will present the algorithm. After this, some analysis will be done, to prove running time and correctness. And finally, some improvements upon AKS will be presented - those discussed in Section 2.4.

In 2000, Ajtai, Kumar, and Sivakumar published 'A Sieve Algorithm for the Shortest Lattice Vector Problem', which presented a $2^{O(n)}$ randomized algorithm for solving SVP [3]. The first paper presenting this technique as a possible method for solving SVP, it has spawned a large amount of publications modifying and improving the algorithm it presented. The presentation of this chapter is based on the lecture notes from Vaikuntanathan [12] and Regev [13] on the AKS algorithm.

Recall the fundamental parallelepiped 2.5. Any vector in the ambient space can be represented as the sum of a lattice vector and a member of a fundamental parallelepiped. For example, $r \in \mathbb{R}^2$ must be the sum of an integer lattice point (the integer part of r's coordinates) and a member of the fundamental parallelepiped for $\mathbb{Z}^2$ (the fractional part of r's coordinates).

The idea behind the AKS algorithm is to sample a large number of lattice vectors, all perturbed by a small amount. We do this by first sampling a large number of points from a small ball around 0. For each point $\boldsymbol{x}$ generated by this sampling, we can use them to calculate a lattice vector by first solving $B\boldsymbol{v} = \boldsymbol{x}$. This $\boldsymbol{v}$ will allow us to represent the points as linear combinations of the lattice vectors. Rounding the coefficients can give us a lattice vector $\boldsymbol{z}$, and the perturbation of $\boldsymbol{z}$ is y, given by $\boldsymbol{y} = \boldsymbol{z} + \boldsymbol{x}$.

As an example of this process, say we sampled $\boldsymbol{x} = [0.3, -0.1]^T$ with the basis for $\mathbb{Z}^2$ from earlier, $B_1 = \left\{[3,1]^T, [2,1]^T\right\}$. We treat B_1 as a matrix as in 2.2 and get the equation $B_1\boldsymbol{v} = \boldsymbol{x}$, which we can solve to get $\boldsymbol{v} = [0.5, -0.6]^T$. Then rounding gives us the lattice vector $\boldsymbol{z} = [1, 0]^T$, and the perturbation $\boldsymbol{y} = [1.3, -0.1]^T$. We remember each vector $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ of these vectors for the next steps.

After we have sampled many perturbed lattice vectors, we put them into locality-based buckets, based on some radius. This means that we will have one perturbed lattice vector at the center of some ball, and many other ones grouped based on the center they are closest to. When this procedure is done, we subtract from each perturbation the lattice vector corresponding to the center of its ball. This way all the new lattice vectors generated from this subtraction will be at most a small amount larger than the radius of the balls. We can make new balls of even smaller radius now, decreasing the maximum length of the lattice vectors in each step. The process of making smaller and smaller balls terminates when the radius of the ball is small enough that we can expect to be able to find the shortest lattice vector.

3.1 Algorithm

In this section, we present the algorithm. Before the algorithm, it is necessary to discuss a sampling subroutine.

3.1.1 Sampling

The first step of the algorithm is to sample a large number of points in $\mathbb{R}^n$, called Y. Then we need to refine this set of points according to some parameter R to produce a resulting set of points C with the following properties:

1. C contains at most 5^n points
2. For all $y \in Y$, there is some $y' \in C$ with $\|y - y'\| < R/2$

The algorithm that completes this refinement, call it Sieve(Y) presented in Algorithm 4.

Algorithm 4 Sieve step of AKS

function SIEVE(Y)
  $C \leftarrow \{\}$
  $\nu \leftarrow \{\}$
  for $y \in Y$ **do**
    if $\exists y' \in C$ with $\| y - y' \| < R/2$ **then**
      Set hash function $\nu(y) = y'$
    else
      $C = C \cup \{y\}$
    end if
  end for
  return C
end function

3.1.2 AKS algorithm

This algorithm assumes that the shortest vector will be in the range $[2, 3)$ and can be generalized for any $[2x, 3x)$. We can assume this, because the polynomial time LLL algorithm gives us a 2^n bound. This 2^n bound can be split into $[2, 3), [3, 4.5), \cdots, [2^n/1.5, 2^n)$. There are polynomially many of these intervals, so only a polynomial number of AKS algorithm runs will need to be executed if we do the polynomial time LLL algorithm as preprocessing.

Algorithm 5 The AKS Sieving Algorithm

function AKS
  Initialization
  $R_0 \leftarrow n \cdot \max \|\boldsymbol{b}_i\|, N \leftarrow 2^{8n} \log R_0$
  Sample $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_N\} \leftarrow B(0, 2)$
  Calculate $\forall i \in \{1, \cdots, N\}, \boldsymbol{y}_i = \boldsymbol{x}_i \mathrm{mod} \mathbb{P}(B)$
  Let $R \leftarrow R_0, X \leftarrow \{\boldsymbol{x}_i\}_{i \in \{1..N\}}, Y \leftarrow \{\boldsymbol{y}_i\}_{i \in \{1..N\}}, Z \leftarrow \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i \in \{1..N\}}$
  Sieving
  while $R > 6$ **do**
    $C \leftarrow Sieve(Y)$
    for $\boldsymbol{y}_i \in Y$ **do**
      if $\boldsymbol{y}_i \in C$ **then** Discard $\boldsymbol{y}_i$
      else$\boldsymbol{y}_i \leftarrow \boldsymbol{y}_i - (\boldsymbol{y}_{\nu(i)} - \boldsymbol{x}_{\nu(i)})$, where ν is the hashing function from Sieve
      end if
    end for
    $R \leftarrow \frac{R}{2} + 2$
  end while
  Output: **return** Shortest vector $(\boldsymbol{y}_j - \boldsymbol{x}_j) - (\boldsymbol{y}_k - \boldsymbol{x}_k) : (\boldsymbol{x}_j, \boldsymbol{y}_j), (\boldsymbol{x}_k, \boldsymbol{y}_k) \in Z$
end function

3.2 Analysis

In order to prove that this algorithm works, several invariants must be proven to be invariant. We start with one necessary for correctness.

3.2.1 Invariants

Theorem 3.1. *The vector $\boldsymbol{y}_j - \boldsymbol{x}_j$ is at every step a lattice vector.*

Proof. First, observe that during initialization, $\boldsymbol{x}_i \equiv \boldsymbol{y}_i \bmod \mathcal{P}(B)$ - this means $\boldsymbol{y}_i - \boldsymbol{x}_i \equiv 0 \bmod \mathcal{P}(B)$. This is equivalent to saying that $\boldsymbol{y}_i - \boldsymbol{x}_i$ is a lattice vector. During the computation, $\boldsymbol{y}_i$ may be reduced $\boldsymbol{y}_i \to \boldsymbol{y}_i - (\boldsymbol{y}_{\nu(i)} - \boldsymbol{x}_{\nu(i)})$, which means that $\boldsymbol{y}_i - \boldsymbol{x}_i \to (\boldsymbol{y}_i - \boldsymbol{x}_i) - (\boldsymbol{y}_{\nu(i)} - \boldsymbol{x}_{\nu(i)})$. Because both $\boldsymbol{y}_i - \boldsymbol{x}_i$ and $(\boldsymbol{y}_{\nu(i)} - \boldsymbol{x}_{\nu(i)})$ are lattice vectors, their difference is a lattice vector. Then $\boldsymbol{y}_i - \boldsymbol{x}_i$ is at every step a lattice vector for any i. This means the return value must also be a lattice vector. □

Some other invariants we need will help us understand the running time of the algorithm.

Theorem 3.2. *After one iteration through 0 Algorithm 5's while loop, each $\|\boldsymbol{y}_i\| \leq \frac{R}{2} + 2$.*

Proof. Due to the sieve step, we have $\left\|y_i - y_{\nu(i)}\right\| \leq \frac{R}{2}$. Also, each $\boldsymbol{x}_i$ satisfies $\|\boldsymbol{x}_i\| \leq 2$ by the sampling procedure. As a result, the new $\boldsymbol{y}_i$ value $\boldsymbol{y}_i \leftarrow \boldsymbol{y}_i - (\boldsymbol{y}_{\nu(i)} - \boldsymbol{x}_{\nu(i)})$ has norm

$$\left\|\boldsymbol{y}_i - (\boldsymbol{y}_{\nu(i)} - \boldsymbol{x}_{\nu(i)})\right\| = \left\|(\boldsymbol{y}_i - \boldsymbol{y}_{\nu(i)}) + \boldsymbol{x}_{\nu(i)})\right\| \leq \left\|\boldsymbol{y}_i - \boldsymbol{y}_{\nu(i)}\right\| + \left\|\boldsymbol{x}_{\nu(i)})\right\| \leq \frac{R}{2} + 2$$

by the triangle inequality. □

Theorem 3.3. *The algorithm 0 runs through at most $2\log(R_0)$ iterations of the while loop.*

Proof. Consider two iterations of the while loop, starting with radius R. This will take $R \to \frac{R}{2} + 2 \to \frac{R/2+2}{2} + 2 = \frac{R}{4} + 3$. Because we can assume $R > 6$ (this is the condition for the while loop to continue), the radius after two steps is $\frac{R}{4} + 3 < \frac{R}{2}$. Then radius is reduced by at least a factor of 2 every 2 iterations of the while loop, giving at most $2\log(R_0)$ iterations total. □

We want to also calculate how many points will remain at the end of the while loop - those that haven't been discarded. This will help make sure we have a very small probability of error.

Theorem 3.4. *The number of remaining points in Z after the while loop ends is at least $2^{7n}\log(R_0)$.*

Proof. We start with $N = 2^{8n}\log(R_0)$ points. The points that are removed are all part of the set of center points C at some step in the algorithm. We knew that we could bound the size of this set as $|C| \leq 5^n$. Then after at most $2\log(R_0)$ iterations of the while loop (and therefore $2\log(R_0)$ Sieves), we have removed at most $2\log(R_0)5^n$ points. This leaves

$$(2^{8n} - 2 \cdot 5^n)\log(R_0) \geq 2^{7n}\log(R_0)$$

points, because $2 \cdot 5^n \leq 2^{7n}$. □

3.2.2 Running Time

The initialization step will take $2^{O(n)}$ time. We need to make $2^{O(n)}$ samples, and calculate the corresponding lattice vectors for each of them. These operations total $2^{O(n)}$ time, as sampling is constant time and calculating lattice vectors can be done in polynomial time.
The inside of the while loop should also take $2^{O(n)}$ time. This is because every point is processed twice - in the Sieve algorithm 4, and in the reducing step. The sieving step should take at the most $2^{O(n)}$ time for any point - it needs to find if there is a center point near it, and the center point set can be of size 5^n. In the reduction step, we need to find $\nu(i)$ and do some vector arithmetic. The hashing step should be no more than $2^O(n)$ time. As a result, the entire loop body runs in $2^O(n)$ time.
Putting it all together, the inside of the while loop will be run only $2\log(R_0)$ times maximally, which is $O(\log(n))$ times. Then the running time is $2^{O(n)} + O(\log(n))2^{O(n)} = 2^{O(n)}$ time.

3.2.3 Correctness

We wish to show that this algorithm correctly returns the shortest vector of the lattice. We see that there are at least 2^{7n} perturbed lattice points remaining after the while loop finishes. Each point $\boldsymbol{x}_i, \boldsymbol{y}_i \in Z$, represented as $\boldsymbol{z}_i = \boldsymbol{y}_i - \boldsymbol{x}_i$ has norm $\|\boldsymbol{z}_i\| \leq 8$, because $\|\boldsymbol{y}_i\| \leq 6$ and $\|\boldsymbol{x}_i\| \leq 2$.

Then there are 2^{7n} lattice points fit into a ball of radius 8 around the origin. Recall that we assumed $\lambda_1(\mathcal{L}(B)) \in [2, 3)$ (we made this assumption right before presenting the algorithm). But then around each lattice point in this ball, there is a ball of radius 1 where there can be no lattice points. Then we can do the same calculation as for enumerating C in the sieving piece for enumerating the number of possible remaining lattice points:

$$\frac{Vol(B(0,9))}{Vol(B(0,1))} = 9^n.$$

So there are at most than 9^n possible lattice points, but 2^{7n} total points. This seems very likely that the shortest vector will show up at least once, but it's also possible that it doesn't. We want to figure out the probability that this does happen.

First, call the shortest vector $\boldsymbol{v}$. Observe that, because $2 \leq \|v\| < 3$, it is possible to find pairs of points $\boldsymbol{x}_i, \boldsymbol{x}_j \in B(0,2)$, such that $\boldsymbol{x}_i = \boldsymbol{v} + \boldsymbol{x}_j$. Then $\boldsymbol{x}_i \equiv \boldsymbol{x}_j mod \mathcal{P}(B)$, so $\boldsymbol{y}_i = \boldsymbol{y}_j$, when their result modulo the fundamental parallelepiped are calculated. Because $\boldsymbol{x}_i$ values for the vectors remaining after the while loop are never used, the algorithm will run exactly the same, even if these $\boldsymbol{x}_i$ are replaced with the other vector mapping to the same $\boldsymbol{y}_i$ value. We can formalize this by defining a function

$$flip(\boldsymbol{x}) = \begin{cases} \boldsymbol{x} + \boldsymbol{v} & : \boldsymbol{x} \in B(0,2) \cap B(-\boldsymbol{v}, 2) \\ \boldsymbol{x} - \boldsymbol{v} & : \boldsymbol{x} \in B(0,2) \cap B(\boldsymbol{v}, 2) \\ \boldsymbol{x} & else \end{cases}, \tag{3.1}$$

and noticing that the algorithm runs the exact same way for values that remain after all iterations through the loop body, whether the original $\boldsymbol{x}$ value is used or $flip(\boldsymbol{x})$ is used instead. First, let us decide how frequently we will have a point that is changed by flip. This will enlighten us to the probability that our true shortest vector is obtained.

Theorem 3.5. *For an $\boldsymbol{x}$ value sampled from $B(0,2)$, $Pr[\boldsymbol{x} \in B(0,2) \cap B(-\boldsymbol{v}, 2) \vee B(0,2) \cap B(\boldsymbol{v}, 2)] \geq 2 \cdot 2^{-2n}$.*

Proof. First, let us consider $B(0,2) \cap B(-\boldsymbol{v}, 2)$. The same argument will be applicable to $B(0,2) \cap B(\boldsymbol{v}, 2)$.
Because we assumed $\|\boldsymbol{v}\| < 3$, we can inscribe a ball of radius $\frac{1}{2}$ in $B(0,2) \cap B(-\boldsymbol{v}, 2)$, given by $B(\frac{\boldsymbol{v}}{2}, 0.5) \in B(0,2) \cap B(-\boldsymbol{v}, 2)$. The volume of this ball relative to the entire sampling space $B(0,2)$ is

$$\frac{Vol(B(v/2, 0.5))}{Vol(B(0,2))} = \frac{0.5^n}{2^n} = 2^{-2n}.$$

So we have $Pr[\boldsymbol{x} \in B(0,2) \cap B(-\boldsymbol{v},2)] \geq 2^{-2n}$ and, similarly, that $Pr[\boldsymbol{x} \in B(0,2) \cap B(v,2)] \geq 2^{-2n}$. But these are disjoint sets (except for maybe the origin if $\|\boldsymbol{v}\| = 2$), because we assumed $\|\boldsymbol{v}\| \geq 2$. Then the total probability $Pr[\boldsymbol{x} \in B(0,2) \cap B(-\boldsymbol{v},2) \vee B(0,2) \cap B(\boldsymbol{v},2)] \geq 2 \cdot 2^{-2n}$, as we wanted to show. □

Then with high probability (we can apply a Chernoff bound here), there are about $2^{7n-2n} = 2^{5n}$ total $\boldsymbol{x}, \boldsymbol{y}$ pairs that fell into either of the two sets. With these 2^{5n} points and only 9^n possible lattice points, we know that there is at least 1 lattice point $\boldsymbol{w}$ with $2^{5n}/9^n \approx 2^{1.8n}$ remaining points which could have been flipped near it. We just need that two of the $\boldsymbol{x}$ values fell into different sets (around $\boldsymbol{v}$ or $-\boldsymbol{v}$), which there is overwhelming probability of happening, given that there is a pool of $2^{1.8n}$ $\boldsymbol{x}$ values.
In fact, suppose they were all zero - we could have instead flipped some of $\boldsymbol{x}$ values prior (randomly), and been returned $\boldsymbol{w} + \boldsymbol{v}$ or $\boldsymbol{w} - \boldsymbol{v}$ instead of $\boldsymbol{w}$. Because some of the values would have given some $\boldsymbol{w} \pm \boldsymbol{v}$ and some values would have resulted in $\boldsymbol{w}$, the AKS algorithm would be guaranteed to return $\boldsymbol{v}$.
As a result, we have that the probability of error is in fact the probability that all $\boldsymbol{x}$ values belong to the same set $\boldsymbol{x} \in B(0,2) \cap B(-\boldsymbol{v},2)$ or $B(0,2) \cap B(\boldsymbol{v},2)$, which is very unlikely - there are $2^{1.8n}$ vectors all needing to belong to the same set, giving a probability of $\approx 2^{-2^{1.8n}}$.

3.3 Improvements

While this is a $2^{O(n)}$ time and space algorithm, the constants are larger than they need to be. The algorithm presented in the original paper can be modified to run in $2^{5.9n}$ time and $2^{2.95n}$ space, as shown by Nguyen and Vidick [8]. Let us consider the first step of the algorithm - sampling. The algorithm samples a very large number of vectors just in order to start the algorithm. One might consider this a large investment, and conjecture that the algorithm's runtime could be improved by interleaving sampling and reducing, stopping when as many vectors as are needed are generated. This should improve both the time and space complexities. It is this insight that led to Daniele Micciancio's and Panagiotis Voulgaris' paper: 'Faster exponential time algorithms for the shortest vector problem.'[9]
In Micciancio and Voulgaris' paper, they present two algorithms - ListSieve 6 and GaussSieve 7, which sample as the algorithm runs instead of all before the algorithm starts. ListSieve is proven in the paper to run in $2^{3.199n}$ time and $2^{1.325n}$ space. GaussSieve was not presented with a runtime proof (it was proven to use $2^{0.41n}$ space at most), but heuristics were given to show that it runs in $2^{0.48n}$ time and $2^{0.18n}$ space. The end condition for these algorithms is different from the end of AKS, though. While

AKS finishes when R gets to be small enough, these algorithms finish when there is a vector whose length is smaller than a predetermined cutoff length μ. The similarity, though, is that AKS assumes that the shortest vector is in a range $[2x, 3x)$, so the top of the range for running AKS can be used as the μ value for ListSieve or GaussSieve. The algorithms are shown below.

Algorithm 6 The Modified Sieving Algorithm - ListSieve

function SAMPLE(B, d)
  Sample $\boldsymbol{e} \leftarrow B(0, d)$
  $\boldsymbol{p} \leftarrow \boldsymbol{e} mod \mathcal{P}(B)$
  return $(\boldsymbol{p}, \boldsymbol{e})$
end function

function LISTREDUCE(p, L, δ)
  while $\exists \boldsymbol{v}_i \in L$ such that $\|\boldsymbol{v}_i - \|\boldsymbol{p}\|\| \leq \delta \|\boldsymbol{p}\|$ **do**
    $\boldsymbol{p} \leftarrow \boldsymbol{p} - \boldsymbol{v}_i$
  end while
  return $\boldsymbol{p}$
end function

function LISTSIEVE(B, μ)
  Initialization
  List $L \leftarrow \{\mathbf{0}\}$
  $\delta \leftarrow 1 - \frac{1}{n}$
  $i \leftarrow 0$
  $\zeta \leftarrow 0.685$
  $MaxIterations \leftarrow 2^{cn}$
  Sampling and Reducing
  while $i < MaxIterations$ **do**
    $i \leftarrow i + 1$
    $(\boldsymbol{p}_i, \boldsymbol{e}_i) \leftarrow$ Sample$(B, \zeta\mu)$
    $\boldsymbol{p}_i \leftarrow$ ListReduce$(\boldsymbol{p}_i, L, \delta)$
    $\boldsymbol{v}_i \leftarrow \boldsymbol{p}_i - \boldsymbol{e}_i$
    done up to here
    if $\boldsymbol{v}_i \notin L$ **then**
      if $\exists \boldsymbol{v}_j \in L$ such that $\|\boldsymbol{v}_i - \boldsymbol{v}_j\| < \mu$ **then**
        return $\boldsymbol{v}_i - \boldsymbol{v}_j$
      end if
      L.append$(\boldsymbol{v}_i)$
    end if
  end whilereturn No correct vector
end function

One very slow piece of even these improved algorithms is the step where we determine if there exists a $\boldsymbol{v}_i \in L$: such that $\|\boldsymbol{v}_i\| \leq \|\boldsymbol{p}\| \wedge \|\boldsymbol{v}_i - \|\boldsymbol{p}\|\| \leq \delta \|\boldsymbol{p}\|$. Finding close vectors in high dimensional space is in general a very difficult problem. We know how to do it effectively in one dimension - sorting and using binary search, or constructing a binary

Algorithm 7 The Modified Sieving Algorithm - GaussSieve

function GaussReduce($\boldsymbol{p}, L, S$)
  while $\exists \boldsymbol{v}_i \in L$: such that $\|\boldsymbol{v}_i\| \leq \|\boldsymbol{p}\| \wedge \|\boldsymbol{v}_i - \|\boldsymbol{p}\|\| \leq \delta \|\boldsymbol{p}\|$ **do**
    $\boldsymbol{p} \leftarrow \boldsymbol{p} - \boldsymbol{v}_i$
  end while
  while $\exists \boldsymbol{v}_i \in L$ such that $\|\boldsymbol{v}_i\| > \|\boldsymbol{p}\| \wedge \|\boldsymbol{v}_i - \boldsymbol{p}\| \leq \|\boldsymbol{v}_i\|$ **do**
    L.remove($\boldsymbol{v}_i$)
    S.push($\boldsymbol{v}_i - \boldsymbol{p}$)
  end while
  return $\boldsymbol{p}$
end function
function GaussSieve(B, μ)
  Initialization
  List $L \leftarrow \{\mathbf{0}\}$
  Stack $S \leftarrow \{\}$
  $Collisions \leftarrow 0$
  Sieving
  while $Collisions < MaxCollisions$ **do**
    if S.notEmpty() **then**
      $\boldsymbol{v}_{new} \leftarrow S$.pop()
    else$\boldsymbol{v}_{new} \leftarrow$ SampleGaussian(B)
    end if
    $\boldsymbol{v}_{new} \leftarrow$ GaussReduce($\boldsymbol{v}_{new}, L, S$)
    if $\boldsymbol{v}_{new} = 0$ **then**
      $Collisions \leftarrow Collisions + 1$
    else$L \leftarrow L$.append($\boldsymbol{v}_{new}$)
    end if
  end while
end function

tree and searching in the tree. These are both methods of preprocessing a list of size N in $O(N \log(N))$ time, allowing for a $O(\log(N))$ search. But in multiple dimensions, we don't really know how to do this effectively, and so a linear search is how we'd implement it for these algorithms, giving an $O(N) = 2^{(O(n))}$ runtime *for each search.* There is an algorithm, however, that has been used for image recognition and other machine learning applications, that can do this. It is called Locality Sensitive Hashing - the idea is to create a hash function that has a high probability of collisions if vectors are close together, and then instead of searching the whole list, we can just search through vectors with the same hash value.

In 2014, Thijs Laarhoven published 'Sieving for shortest vectors in lattices using locality-sensitive hashing.' It takes GaussSieve as I've described above, and applies this locality sensitive hashing technique to reduce search space. The result is an algorithm that runs, heuristically, but not provably, in $2^{0.3366n}$ time and $2^{0.2075n}$ space. In order to reproduce

the algorithm here, I would need to explain locality sensitive hashing in more detail (and it is quite technical), so one may find the original paper here: [11].

Chapter 4

CVP with Preprocessing using Voronoi Cells

In this chapter, we will see a new method of attacking the Closest Vector Problem, Preprocessing with Voronoi cells. We start by defining some terms which will be important in describing and discussing the algorithm. Next, I will present each step of the algorithm, with some discussion accompanying each step. The first step is dimension reduction - converting CVP computations into smaller ones. The second step is preprocessing - coming up with a way to solve SVP fairly quickly. The third step is the SVP solver, using the preprocessing.

In 2010, Daniele Micciancio and Panagiotis Voulgaris (who you may recognize from an improvement upon AKS) presented 'A Deterministic Single Exponential Time Algorithm for Most Lattice Problems based on Voronoi Cell Computations.'[4] This paper contained an algorithm as fast (asymptotically) as AKS, but has a few key differences. First, the algorithm is deterministic, while AKS is randomized. This means that AKS has a very small probability of not producing the correct shortest lattice vector, while the MV algorithm will always produce the correct shortest vector. Another difference is that MV is not a sieving algorithm. It instead uses a different technique - its goal is to use the connection between three different problems on a lattice to come up with the shortest vector. In order to discuss this in more detail, we need to define some terms.

4.1 Preliminaries

In this chapter, we will write B_i to represent the basis B but restricted to only the first i basis vectors. That is, if $B = \{\boldsymbol{b}_1, \boldsymbol{b}_2, \cdots, \boldsymbol{b}_n\}$, then $B_i = \{\boldsymbol{b}_1, \boldsymbol{b}_2, \cdots, \boldsymbol{b}_i\}$.

Definition 4.1. The Closest Vector Problem with Preprocessing is the problem, when given some $\pi(B)$ and a target vector $\boldsymbol{t}$, to output the closest lattice vector $\boldsymbol{v}$ to the target vector $\boldsymbol{t}$. That is, $\|\boldsymbol{v} - \boldsymbol{t}\| \leq \|\boldsymbol{x} - \boldsymbol{t}\|$ for any $\boldsymbol{x} \in \mathcal{L}(B)$.

The function π can be thought of as giving a hint for solving CVP. The particular hint used in this algorithm is the Voronoi cell:

Definition 4.2. The Voronoi cell of a lattice point $\boldsymbol{v} \in \mathcal{L}(B)$, denoted $\mathcal{V}(\boldsymbol{v})$, is the set of all points

$$\{\boldsymbol{x} | \, \|\boldsymbol{x} - \boldsymbol{v}\| \leq \|\boldsymbol{x} - \boldsymbol{w}\| \, \forall \boldsymbol{w} \in \mathcal{L}(B)\} \tag{4.1}$$

It is now possible to see a connection between Voronoi cells and CVP. If we are searching for the closest vector to $\boldsymbol{t}$, we want to find the lattice vector $\boldsymbol{v}$ such that $\boldsymbol{t} \in \mathcal{V}(\boldsymbol{v})$. So there is some intuition for the MV algorithm. It uses, as a hint, a Voronoi cell $\mathcal{V}(\boldsymbol{0})$. However, before we are able to give a Voronoi cell as a hint, we need to come up with a way to describe it. A Voronoi cell consists of an uncountably infinite number of points, so instead of describing its interior, we should look at its boundary. To start:

Definition 4.3. Let $\boldsymbol{v}$ and $\boldsymbol{w}$ be lattice vectors in the lattice $\mathcal{L}(B)$. A half space $\mathcal{H}_{\boldsymbol{w}}(\boldsymbol{v})$ is the set of all points

$$\{\boldsymbol{x} | \, \|\boldsymbol{x} - \boldsymbol{v}\| \leq \|\boldsymbol{x} - \boldsymbol{w}\|\} \, . \tag{4.2}$$

These are all the points that are closer to $\boldsymbol{v}$ than $\boldsymbol{w}$.

Note that the Voronoi cell $\mathcal{V}(\boldsymbol{v}) = \cap_{\boldsymbol{w} \in \mathcal{L}(B)} H_{\boldsymbol{w}}(\boldsymbol{v})$. This is because the Voronoi cell is the set of all points closer to $\boldsymbol{v}$ than any other lattice vector. However, not all of these half spaces are needed to determine the Voronoi cell.
If we have a minimal set V of half spaces such that $\mathcal{V}(\boldsymbol{v}) = \cap_{\boldsymbol{w} \in V} H_{\boldsymbol{w}}(\boldsymbol{v})$, we call V a set

of Voronoi-relevant vectors. This Voronoi-relevant set will be the way we describe the Voronoi cell to be passed into our CVP with Preprocessing algorithm. It is important for our running time analysis to know how many vectors there can possibly be in the Voronoi-relevant set. The correct maximum number is $2(2^n - 1)$, as shown by Minkowski [5].

Now we can discuss the algorithm itself. It relies on the interaction between two problems - calculating the Voronoi cell, and solving the Closest Vector Problem using the Voronoi cell. The interaction is not just that solving CVP helps you find a Voronoi cell, and vice versa. That would get us stuck in an infinite loop. The interaction is described in Algorithm 8.

Algorithm 8 Recursively Solving CVP($\boldsymbol{t}$,B_n)

function CVP($\boldsymbol{t}$, B_{i+1})
  Calculate $\mathcal{V}$ for B_i, then solve $2^{i/2}$ instances of CVPP($\boldsymbol{t}'$,$\mathcal{V}$)
end function
function VORONOI CELL CALCULATION(B_i)
  Solve 2^i instances of CVP(B_i) to generate Voronoi cell
end function
function CVPP($\boldsymbol{t}, \mathcal{V}$ in dimension i)
  $2^{O(n)}$ time algorithm using the Voronoi cell
end function

So in order to solve this one computation of CVP($\boldsymbol{t}, B_n$), we calculate the Voronoi cell using 2^n calculations of CVP in B_n. In order to do these CVP computations, we do $2^{n/2}$ CVP computations in B_{n-1}. Each of these is done using a Voronoi cell calculation in B_{n-1}, which is in turn built using smaller CVP calculations and so on.

Let us calculate the resulting time complexity, given that the computation of CVPP runs in $2^{O(n)}$ time. For each k from $1, 2, \cdots n$, we need to solve $2^k \cdot 2^{k/2}$ CVPP instances on B_k. This gives us a resulting runtime of $\sum_{k=1}^{n} 2^{1.5k} 2^{O(n)} = 2^{O(n)}$.

4.2 Algorithm

The goal is to solve CVP for a target vector $\boldsymbol{t}$ in a basis B. We first assume B is LLL-reduced. Recall that this is just a polynomial time algorithm, and so does not contribute significantly to the runtime of the algorithm. Additionally, if B is LLL-reduced, so is any B_i. Also, as usual, the notation $\tilde{\boldsymbol{b}}_i$ represents the ith Gram-Schmidt vector of the basis B.

I will first discuss the dimension reduction piece of the algorithm. Next, I will show how to calculate a Voronoi cell. And finally, I will demonstrate how knowledge of the Voronoi cell can be used to solve CVP.

4.2.1 Dimension Reduction

Our goal is to be able to calculate the closest lattice vector $\boldsymbol{v} \in B_{i+1}$ to a target vector $\boldsymbol{t}$ using several solutions to the CVP problem v_j in B_i. The way we do this is by considering several planes that our target vector could lie on, and find the closest vector on each of these planes. Each plane has a smaller dimension than the original space, so we have a smaller CVP computation as a result.
First, we assume that $\boldsymbol{t}$ is in the subspace spanned by B_{i+1}. Otherwise, we just project $\boldsymbol{t}$ onto the subspace. We also define the value $c_t = \langle \boldsymbol{t}, \tilde{\boldsymbol{b}}_{i+1} \rangle / \langle \tilde{\boldsymbol{b}}_{i+1}, \tilde{\boldsymbol{b}}_{i+1} \rangle$, the part of $\boldsymbol{t}$ parallel to $\tilde{\boldsymbol{b}}_{i+1}$. The algorithm, assuming $\boldsymbol{t}$ has been projected onto the subspace spanned by B_{i+1}, is the following:

Algorithm 9 CVP Dimension Reduction

function DIMENSION REDUCTION($\boldsymbol{t}$, B_{i+1})
  for each c such that $|c - c_t| \leq \frac{1}{2}\sqrt{2^{i+1} - 1}$ **do**
    $\boldsymbol{s}_{i,c} \leftarrow \text{CVP}(\boldsymbol{t} - c\boldsymbol{b}_{i+1}, B_i)$
  end for
  Output: **return** Vector $\boldsymbol{s}_{i,c} + c\boldsymbol{b}_{i+1} \in \mathcal{L}(B_{i+1})$ that minimizes $\|\boldsymbol{t} - (\boldsymbol{s}_{i,c} + c\boldsymbol{b}_{i+1})\|$
end function

Theorem 4.4. *The above algorithm will produce the closest vector to t from* B_{i+1}.

Proof. We can consider the lattice $\mathcal{L}(B_{i+1})$ as the union of all layers of the lattice $L_c = \{\boldsymbol{x} + c\boldsymbol{b}_{i+1} | \boldsymbol{x} \in B_i, c \in \mathbb{Z}\}$, so each L_c lies on a distinct i dimensional hyperplane. Then the minimum distance from any lattice point on fixed layer L_c to $\boldsymbol{t}$ is at least

$$|c - c_t| \cdot \left\| \tilde{\boldsymbol{b}}_{i+1} \right\|. \tag{4.3}$$

This is because the entire hyperplane $\{\boldsymbol{x} + c\boldsymbol{b}_{i+1} | \boldsymbol{x} \in B_i, c \in \mathbb{Z}\}$ has this distance from $\boldsymbol{t}$, and each lattice point lies on this hyperplane.

In addition, using Babai's nearest plane algorithm [14] could produce a lattice vector within distance $\rho = \frac{1}{2}\sqrt{\sum_{j=1}^{i+1} \left\| \tilde{\boldsymbol{b}}_j \right\|^2}$. Because the basis is LLL-reduced, it has the

property that for any k, $\left\|\tilde{\boldsymbol{b}}_{k+1}\right\| \geq \left\|\tilde{\boldsymbol{b}}_k\right\|/2$. We can use this to obtain

$$\rho = \frac{1}{2}\sqrt{\sum_{j=1}^{i+1}\left\|\tilde{\boldsymbol{b}}_j\right\|^2} \leq \frac{1}{2}\sqrt{\sum_{j=1}^{i+1} 2^{i-j+1}\left\|\tilde{\boldsymbol{b}}_{i+1}\right\|^2} = \frac{1}{2}\sqrt{2^{i+1}-1}\left\|\tilde{\boldsymbol{b}}_{i+1}\right\|. \tag{4.4}$$

Combining equations 4.1 and 4.2, we have that $|c - c_t| \leq \frac{1}{2}\sqrt{2^{i+1}-1}$. Then enumerating over each of these and finding the closest vector among all of them will return the shortest. □

Now we have reduced the dimension of our CVP computations, so now we should come up with our method of solving the CVP computations.

4.2.2 Calculating Voronoi Cell

Recall that calculating a Voronoi cell is equivalent to calculating the Voronoi-relevant vectors. We will construct points that are halfway between two lattice vectors, and find the closest vectors to these. Then the vector between these two should be a Voronoi relevant vector.
The subroutine used for this is the RelevantVectors algorithm from Agrell, Eriksson, Vardy, and Zeger in [15]. The method is presented in Algorithm 10.

Algorithm 10 Voronoi Cell Computation

```
function RelevantVectors(B_i)
    V ← {}
    for each c ∈ {0,1}^i/0 do
        t ← (1/2)cB_i
        s ← CVP(t, B_i)
        if 2 possible s are closest then
            V.add(2(s + t))
            V.add(−2(s + t))
        end if
    end for
    Output: return V
end function
```

Unfortunately, this subroutine can't be run as part of this algorithm. This is because the CVP solving does not know how many shortest vectors there are. In MV's algorithm, $\pm 2(\boldsymbol{s}+\boldsymbol{t})$ are just added for every $\boldsymbol{c}$. Because there are $2^i - 1$ possible values for $\boldsymbol{c}$, the description of the Voronoi cell will have fewer than 2^{i+1} vectors. While there may be some repetition, it will not be enough to significantly increase the running time of the algorithm.

Now that a Voronoi cell has been computed, we may use it to calculate instances of CVPP.

4.2.3 Voronoi Cell for CVPP

This piece of the algorithm takes Voronoi cells and attempts to find a path between 0 and $\boldsymbol{t}$ by moving between adjacent Voronoi cells.

Recall that, by definition of Voronoi cell, if the target vector $\boldsymbol{t}$ for the closest vector problem is in the Voronoi cell $\mathcal{V}(\boldsymbol{v})$ for some lattice vector $\boldsymbol{v}$, then $\boldsymbol{v}$ is a closest vector to $\boldsymbol{t}$. The target $\boldsymbol{t}$ may be at the intersection of multiple Voronoi cells, in which case it would have multiple closest lattice vectors. For now, we restrict ourselves to the case where $\boldsymbol{t} \in 2\mathcal{V}(\boldsymbol{0})$. Later, this will be extended.
To start, we define a graph G, with nodes N and edges E, with

$$N = \{\boldsymbol{t} + \boldsymbol{v} | \boldsymbol{v} \in \mathcal{L}(B_i), (\boldsymbol{t} + \boldsymbol{v}) \in 4\mathcal{V}\}, E = \{(\boldsymbol{x}, \boldsymbol{y}) | \boldsymbol{x} - \boldsymbol{y} \in \mathcal{V}\}. \tag{4.5}$$

So that nodes are lattice vectors shifted by $\boldsymbol{t}$ which fall inside of $4\mathcal{V}$, and nodes are connected if their Voronoi cells border each other.

Algorithm 11 CVPP given Voronoi cell

```
function CVPPSolver(V, t)
    G ← (N, E)
    return Shortest x such that x is connected in G to t
end function
```

The shortest $\boldsymbol{x}$ can be found by doing a graph traversal starting from $\boldsymbol{t}$ of G. This can be done in polynomial time in the size of N and E, which is $2^{O(n)}$ time because $|N|, |E| \in 2^{O(n)}$. Then two statements remain to be shown. First, we need that the closest vector to $\boldsymbol{t}$ is actually connected to $\boldsymbol{t}$ in the G. For the proof of this, one can read the original paper: [4]. The other statement is that we can reduce the problem of CVPP on arbitrary $\boldsymbol{t}$ to solving CVPP on $\boldsymbol{t} \in 2\mathcal{V}$.

Consider what the output of the CVPP solver is. It is an $\boldsymbol{x}$ such that $\boldsymbol{x} \in \mathcal{V}$, where $\boldsymbol{x} = \boldsymbol{t} + \boldsymbol{v}$, with $\boldsymbol{t} \in 2\mathcal{V}$ and $\boldsymbol{v} \in \mathcal{L}(B_i)$. What if $\boldsymbol{t}$ was not in $2\mathcal{V}$, but instead in some $2^k\mathcal{V}$? We could instead run the algorithm with the description of $2^{k-1}\mathcal{V}$, to find some vector $\boldsymbol{x} \in 2^{k-1}\mathcal{V}$ with $\boldsymbol{x} = \boldsymbol{t} + \boldsymbol{v}$, with $\boldsymbol{v} \in 2^{k-1}\mathcal{L}(B_i)$. But now we've reduced calculating the closest vector to $\boldsymbol{t}$ to instead calculating the closest vector to $\boldsymbol{x}$, then adding $-\boldsymbol{v}$. But

$\boldsymbol{x} \in 2^{k-1}\mathcal{V}$, so only k iterations of this procedure are required.

Bibliography

[1] Arjen Klaas Lenstra, Hendrik Willem Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[2] Ravi Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of operations research*, 12(3):415–440, 1987.

[3] Miklós Ajtai, Ravi Kumar, and Dandapani Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 601–610. ACM, 2001.

[4] Daniele Micciancio and Panagiotis Voulgaris. A deterministic single exponential time algorithm for most lattice problems based on voronoi cell computations. *SIAM Journal on Computing*, 42(3):1364–1391, 2013.

[5] Hermann Minkowski. Allgemeine lehrsätze über die convexen polyeder. *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse*, 1897:198–220, 1897.

[6] Divesh Aggarwal, Daniel Dadush, Oded Regev, and Noah Stephens-Davidowitz. Solving the shortest vector problem in 2 n time using discrete gaussian sampling. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, pages 733–742. ACM, 2015.

[7] Divesh Aggarwal, Daniel Dadush, and Noah Stephens-Davidowitz. Solving the closest vector problem in 2ˆ n time–the discrete gaussian strikes again! In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 563–582. IEEE, 2015.

[8] Phong Q Nguyen and Thomas Vidick. Sieve algorithms for the shortest vector problem are practical. *Journal of Mathematical Cryptology*, 2(2):181–207, 2008.

[9] Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 1468–1480. Society for Industrial and Applied Mathematics, 2010.

[10] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

[11] Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In *Advances in Cryptology–CRYPTO 2015*, pages 3–22. Springer, 2015.

[12] Vaikuntanathan. Ajtai-kumar-sivakumar algorithm for exact shortest vectors. http://people.csail.mit.edu/vinodv/6876-Fall2015/index.html, September 2015.

[13] Regev. A simply exponential algorithm for svp (ajtai-kumar-sivakumar). https://www.cims.nyu.edu/~regev/teaching/lattices_fall_2004/, September 2004.

[14] László Babai. On lovászlattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.

[15] Erik Agrell, Thomas Eriksson, Alexander Vardy, and Kenneth Zeger. Closest point search in lattices. *Information Theory, IEEE Transactions on*, 48(8):2201–2214, 2002.

[16] Chris Peikert. A decade of lattice cryptography. Technical report, 2015.

[17] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. *Preprint*, 2015.

[18] Noah Stephens-Davidowitz. Discrete gaussian sampling reduces to cvp and svp. *arXiv preprint arXiv:1506.07490*, 2015.