UNIVERSITY OF CALIFORNIA, SAN DIEGO

Algorithms For The Closest And Shortest Vector Problems On General Lattices

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy

in

Computer Science

by

Panagiotis Voulgaris

2011

DEDICATION

To my family and my advisor.

EPIGRAPH

Stop wasting your time being sorry,
just do it.
—Daniele Micciancio

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

ACKNOWLEDGEMENTS

I would like to thank all the people that made my graduate life one of the most creative and fulfilling periods of my life.

First and foremost, I would like to thank my advisor Daniele Micciancio, for believing in me and setting high expectations, for giving me the freedom and time to pursue these goals and for having the energy and patience to be there and listen to a lot of random (and unsuccessful) ideas. I was fortunate to work, learn and grow both as a researcher and person under his guidance.

I am thankful to Ramamohan Paturi for giving me the opportunity to be his TA numerous times and both Ramamohan and Russell Impagliazzo for teaching a number of excellent theory classes that inspired my work. I am also thankful to Alexander Vardy, Nolan Wallach, Russell and Ramamohan for being members of my thesis proposal committee.

While on my Ph.D. program I had the pleasure to collaborate with many great people. First I would like to thank all the UCSD people I have worked with: Manuel Cebrian, Lorenzo Coviello, Daniel Lokshtanov, Petros Mol and Andrea Vattani. It was an honor and above all fun to work with you. I would like to thank Vinod Vaikuntanathan, for making my internship in Microsoft Research one of the best learning experiences I ever had. I have also greatly benefited from discussions and feedback from the following researchers: Daniel Dadush, Phong Nguyen, Chris Perikert, Oded Regev and Damien Stehlé.

It would be very hard to accomplish anything without the help and advice of my friends. I would like to thank my labmates (Andrea, Daniel, Daniel, Karyn, Petros, Rafael, Scott, Shanshan, Sarah), people from the cryptogroup (Panju, Sarah, Sriram, Todor, Tom, Vadim), friends I met in UCSD (Alex, Andreas, Andreas, Bill and Banu, Christos, Christophe, Dionysis, Elio, Emiran, Lorenzo and Emanuele, Marios), Greek friends (Albert, Antonis) and the Takens (Patrick, Didem, Gjergji, Levon and Mary, Nikos and Marisol, Tikir, Yuvraj). Special thanks go to my fiancée Wen Yi for making my life happier with her smile.

Chapter 3 and Section A.1, in part, are a reprint of the paper "Large Faster exponential time algorithms for the shortest vector problem" co-authored with Daniele Micciancio and appearing in the proceedings of SODA 2010. The dissertation author was the primary investigator and author of this paper.

Chapter 4, Chapter 5 and Chapter 6, in part, are a reprint of the full version of the paper "A Deterministic Single Exponential Time Algorithm for Most Lattice Problems based on Voronoi Cell Computations" co-authored with Daniele Micciancio. An extended abstract of this paper was presented in STOC 2010. The dissertation author was the primary investigator and author of this paper.

Finally, I am thankful to Phong Nguyen and Thomas Vidick for providing the implementation of the sieve algorithm of [NV08] which we used for the experimental comparison of Section 3.4 and Damien Stehlé for pointing out some problems in earlier drafts of Chapter 6.

VITA

2006	Diploma (B. S.) in Electrical and Computer Engineering National Technical University of Athens, Greece
2011	Doctor of Philosophy in Computer Science University of California, San Diego, CA, USA

ABSTRACT OF THE DISSERTATION

Algorithms For The Closest And Shortest Vector Problems On General Lattices

by

Panagiotis Voulgaris

Doctor of Philosophy in Computer Science

University of California, San Diego, 2011

Professor Daniele Micciancio, Chair

The shortest vector problem (SVP) and closest vector problem (CVP) are the most widely known problems on point lattices. SVP and CVP have been extensively studied both as purely mathematical problems, being central in the study of the geometry of numbers and as algorithmic problems, having numerous applications in communication theory and computer science.

There are two main algorithmic techniques for solving exact SVP and CVP: enumeration and sieving. The best enumeration algorithm was given by Kannan in 1983 and solves both problems in $n^{O(n)}$ time, where n is the dimensionality of the lattice. Sieving was introduced by by Ajtai, Kumar and Sivakumar in 2001 and lowered the time complexity of SVP to $2^{O(n)}$, but required $2^{O(n)}$ space and randomness. This result posed a number of important questions: Could we get a *deterministic* $2^{O(n)}$ algorithm for SVP? Is it possible to acquire a $2^{O(n)}$ algorithm for CVP?

In this dissertation we give new algorithms for SVP and CVP and resolve these questions in the affirmative. Our main result is a *deterministic* $\tilde{O}(2^{2n})$ time and $\tilde{O}(2^n)$ space that solves both SVP and CVP and by reductions of (Micciancio, 2008) most other lattice problems in NP considered in the literature. In the case of CVP the algorithm improves the time complexity from $n^{O(n)}$ to $2^{O(n)}$, while for SVP we achieve single exponential time as sieving, but without using randomization and improving the constant in the exponent from 2.465 (Pujol and Stehlé, 2010) to 2. The core of the algorithm is a new method to solve the closest vector problem with preprocessing (CVPP) that uses the Voronoi cell of the lattice (described as intersection of half-spaces) as the result of the preprocessing function. We also present our earlier results on sieving algorithms. Although the theoretical analysis of the proposed sieving algorithm gives worse complexity bounds than our new Voronoi based approach, we show that in practice sieving can be much more efficient. We propose a new heuristic sieving algorithm that performed quite well in practice, with estimated running time $2^{0.52n}$ and space complexity $2^{0.2n}$.

Chapter 1

Introduction

1.1 Lattice Problems

A d-dimensional lattice Λ can be defined as the set of all integer linear combinations of $n \leq d$ linearly independent basis vectors $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{d \times n}$. There are many famous algorithmic problems on point lattices, the most important of which are:

- The shortest vector problem (SVP): given a basis $\mathbf{B}$, find a shortest nonzero vector in the lattice generated by $\mathbf{B}$.
- The closest vector problem (CVP): given a basis $\mathbf{B}$ and a target vector $\mathbf{t} \in \mathbb{R}^d$, find a lattice vector generated by $\mathbf{B}$ that is closest to $\mathbf{t}$.

SVP and CVP have been extensively studied both as purely mathematical problems, being central in the study of the geometry of numbers [Cas71] and as algorithmic problems, having numerous applications in communication theory [CS98] and computer science. SVP and CVP have been used to solve major algorithmic problems in combinatorial optimization (integer programming [Len83, Kan87b], solving low density subset-sum problems [CJL$^+$92]), algorithmic number theory and geometry of numbers (factoring polynomials over the rationals [LLL82], checking the solvability by radicals [LM85], and many more [Kan87a]) and cryptanalysis (breaking the Merkle-Hellman cryptosystem [Odl89], and many other applications

[JS98, NS01]). Finally understanding the hardness of approximating SVP is fundamental in evaluating the security (and applicability) of many recent lattice based cryptosystems [Ajt04, AD97, Dwo97, Reg04a, Reg09, MR07, PW08, GPV08].

The complexity of lattice problems has been investigated intensively. SVP and CVP have been shown to be NP-hard both to solve exactly [vEB81, Ajt98, BS99], or even approximate within small (constant or sub-polynomial in n) approximation factors [BS99, ABSS97, DKRS03, CN99, Mic01b, Kho05, HR07]. Much effort has gone into the development and analysis of algorithms both to solve these problems exactly [Kan87b, Hel85, HS07, Blö00, AKS01, AKS02, BN09] and to efficiently find approximate solutions [LLL82, Sch88, Sch87, SE94, Sch06, GHGKN06, GN08b].

In this dissertation we focus on the complexity of finding exact solutions to these problems. Of course, as the problems are NP-hard, no polynomial-time solution is expected to exist. Still, the complexity of solving lattice problems exactly is interesting both because many applications (e.g., in mathematics and communication theory [CS98]) involve lattices in relatively small dimension, and because approximation algorithms for high dimensional lattices [Sch87, SE94, GHGKN06, GN08b] (for which exact solution is not feasible) typically involve the exact solution of low dimensional subproblems.

1.2 Prior Work

There have been two main approaches for solving exact lattice problems: enumeration and sieving. *Enumeration algorithms*, given a lattice basis $\mathbf{B}$, systematically explore a region of space (centered around the origin for SVP, or around the target vector for CVP) that is guaranteed to contain a nonzero shortest lattice vector. The running time of these algorithms is roughly proportional to the number of lattice points in that region, which, in turn depends on the quality of the input basis. Using an LLL reduced basis [LLL82], the Fincke-Pohst enumeration algorithm [Poh81] achieves complexity of $2^{O(n^2)}$ for both problems. This was improved to $n^{O(n)}$ by Kannan in 1983 [Kan87b] using a smart basis reduction algorithm. A careful analysis of the algorithm in [Hel85, HS07] further reduced the complexity to $n^{0.184n}$ for SVP and $n^{0.5n}$ for CVP and SIVP. Kannan's algorithm from 1983 remains the best algorithm for CVP and the best *deterministic* algorithm for SVP. Several other enumeration variants have been proposed (see [AEVZ02] for a survey,) including the Schnorr-Euchner enumeration method [SE94], currently used in state of the art practical lattice reduction implementations [Sho03, PS08].

The AKS Sieve, introduced by Ajtai, Kumar and Sivakumar in [AKS01], lowers the running time complexity of SVP to a simple exponential function $2^{O(n)}$ using randomization and $2^{O(n)}$ space. We refer collectively to the algorithm of [AKS01] and its variants as proposed in [NV08] and in this dissertation, as *sieve algorithms*. In [NV08, MV10b, PS09], improved analysis and variants of the AKS sieve are studied, but still using the same approach leading to randomized algorithms. Unfortunately, the best practical variant of sieving [NV08] is outperformed by the asymptotically inferior Schnorr-Euchner enumeration [SE94]. This discrepancy between asymptotically faster algorithms and algorithms that perform well in practice is especially unsatisfactory in the context of lattice based cryptography, where one needs to extrapolate the running time of the best known algorithms to ranges of parameters that are practically infeasible in order to determine appropriate key sizes for the cryptographic function.

Sieving algorithms posed numerous fundamental theoretical questions. Is randomization and exponential space necessary to lower the time complexity of SVP from $n^{O(n)}$ to $2^{O(n)}$? Perhaps an even more interesting question is: can we achieve similar time complexity improvements for CVP? Extensions of the AKS sieve algorithm for CVP have been investigated in [AKS02, BN09], but only led to approximation algorithms which are not guaranteed (even probabilistically) to find the best solution, except for certain very special classes of lattices [BN09]. A possible explanation for the difficulty of extending the result of [AKS01] to the exact solution of SIVP and CVP was offered by Micciancio in [Mic08], where it is shown (among other things) that CVP, SIVP and all other lattice problems considered in [BN09], with the exception of SVP, are equivalent in their exact version under deterministic polynomial-time dimension-preserving reductions. So, either all of them are solvable in single exponential time $2^{O(n)}$, or none of them admits such an algorithm.

1.3 Our Results

In this dissertation we give new algorithms for SVP and CVP and resolve most of the open theoretical open questions discussed above. Our main result [MV10a] is a *deterministic* $\tilde{O}(2^{2n})$ time and $\tilde{O}(2^n)$ space that solves both SVP and CVP and by reductions of [GMSS99, Mic08] most other lattice problems in NP considered in the literature. In the case of CVP the algorithm improves the time complexity from $n^{O(n)}$ [Kan87b] to $2^{O(n)}$, while for SVP we achieve single exponential time as sieving, but without using randomization and improving the time and space complexity from $2^{2.465n}$ and $2^{1.233n}$ [PS09] to $\tilde{O}(2^{2n})$ and $\tilde{O}(2^n)$. The core of the algorithm is a new method to solve the closest vector problem with preprocessing (CVPP) that uses the Voronoi cell of the lattice (described as intersection of half-spaces) as the result of the preprocessing function. We remark that all our algorithms, just like [AKS01], use exponential space. So, the question whether exponential space is required to solve lattice problems in single exponential time remains open.

Additionally we present some of our earlier work [MV10b] on sieving algorithms. We give a $2^{3.2n}$ time and $2^{1.325n}$ space sieving algorithm, improving the $\tilde{O}(2^{5.9n})$-time and $\tilde{O}(2^{2.95n})$-space complexity bounds of the asymptotically best previously known algorithm [AKS01, NV08]. The algorithm is relatively simpler than AKS and gives an explicit connection to sphere packings which allows us to relate the performance of sieve algorithms to well studied quantities in the theory of spherical codes, and make use of the best bounds known to date [KL78]. Although our new Voronoi algorithm gives better theoretical guarantees than sieving, the results remain relevant because sieving can be very effective in practice. In particular we give a heuristic variant of our sieving algorithm which in our experiments outperformed standard enumeration techniques used in practice, with running time $2^{0.52n}$ and space complexity $2^{0.2n}$

In the following subsections we give a fast overview of our sieving algorithms and the basic Voronoi algorithm.

1.3.1 Sieving Algorithms

In order to describe the main idea behind our algorithms, we first recall how the sieve algorithm of [AKS01] works. The algorithm starts by generating a large (exponential) number of random lattice points P within a large (but bounded) region of space. Informally, the points P are passed through a sequence of finer and finer "sieves", that produce shorter and shorter vectors, while "wasting" some of the sieved vectors along the way. (The reader is referred to the original article [AKS01] as well as the recent analysis [NV08] for a more detailed and technical description of the AKS sieve).

While using many technical ideas from [AKS01, NV08], our algorithms depart from the general strategy of starting from a large pool P of (initially long) lattice vectors, and obtaining smaller and smaller sets of shorter vectors. Instead, our algorithms start from an initially empty list L of points, and increase the length of the list by appending new lattice points to it. In our first algorithm *List Sieve*, the points in the list never change: we only keep adding new vectors to the list. Before a new point $\mathbf{v}$ is added to the list, we attempt to reduce the length of $\mathbf{v}$ as much as possible by subtracting the vectors already in the list from it. Reducing new lattice vectors against the vectors already in the list allows us to prove a lower bound on the angle between any two list points of similar norm. This lower bound on the angle between list points allows us to apply the linear programming bound for spherical codes of Kabatiansky and Levenshtein [KL78] to prove that the list L cannot be too long. The upper bound on the list size then easily translates to corresponding upper bounds on the time and space complexity of the algorithm.

Similarly to previous work [AKS01, NV08], in order to prove that the algorithm produces non-zero vectors, we employ a now standard perturbation technique. Specifically, instead of generating a random lattice point $\mathbf{v}$ and reducing it against the vectors already in the list, we generate a perturbed lattice point $\mathbf{v} + \mathbf{e}$ (where $\mathbf{e}$ is a small error vector), and reduce $\mathbf{v} + \mathbf{e}$ instead. The norm of the error

$\mathbf{e}$ is large enough, so that the lattice point $\mathbf{v}$ is not uniquely determined by $\mathbf{v}+\mathbf{e}$. This uncertainty about $\mathbf{v}$ allows to easily prove that after reduction against the list, the vector $\mathbf{v}$ is not too likely to be zero. Unfortunately the introduction of errors reduces the effectiveness of sieving and increases the space complexity.

In practice, as shown in [NV08], variants of sieving algorithms without errors, perform much better, but lack theoretical time bounds. *Gauss Sieve* is a practical variant of *List Sieve* without errors which incorporates a new heuristic technique. Beside reducing new lattice points $\mathbf{v}$ against the points already in the list L, the algorithm also reduces the points in L against $\mathbf{v}$, and against each other. As a result, the list L has the property that any pair of vectors in L forms a Gauss reduced basis. It follows from the properties of Gauss reduced bases that the angle between any two list points is at least $\pi/3$, that is the list forms a good spherical code. In particular, the list length never exceeds the kissing constant τ_n, which is defined as the highest number of points that can be placed on a sphere, while keeping the minimal angle between any two points at least $\pi/3$.[1] As already discussed, this allows to bound the space complexity of our second algorithm by $2^{0.402n}$ in theory, or $2^{0.21n}$ in practice. Unfortunately, we are unable to bound the running time of this modified algorithm, as we don't know how to prove that it produces nonzero vectors. However, the algorithm seems to work very well in practice, and outperforms the best previously known variants/implementations of the AKS Sieve [NV08] both in theory (in terms of provable space bounds,) and in practice (in terms of experimentally observed space and time requirements).

1.3.2 Voronoi Algorithms

At the core of all our results is a new technique for the solution of the closest vector problem with preprocessing (CVPP). We recall that CVPP is a variant of CVP where some side information about the lattice is given as a hint together with the input. The hint may depend on the lattice, but not on the target vector. Typically, in the context of polynomial time algorithms, the hint is restricted to

[1] The name "kissing" constant originates from the fact that $\pi/3$ is precisely the minimal angle between the centers of two nonintersecting equal spheres that touch (kiss) a third sphere of the same radius.

have polynomial size, but since here we study exponential time algorithms, one can reasonably consider hints that have size $2^{O(n)}$. The hint used by our algorithm is a description of the Voronoi cell of the lattice. We recall that the (open) Voronoi cell of a lattice Λ is the set $\mathcal{V}$ of all points (in Euclidean space) that are closer to the origin than to any other lattice point. The Voronoi cell $\mathcal{V}$ is a convex body, symmetric about the origin, and can be described as the intersection of half-spaces $H_{\mathbf{v}}$, where for any nonzero lattice point $\mathbf{v}$, $H_{\mathbf{v}}$ is the set of all points that are closer to the origin than to $\mathbf{v}$. It is not necessary to consider all $\mathbf{v} \in \Lambda \setminus \{\mathbf{0}\}$ when taking this intersection. One can restrict the intersection to the so-called *Voronoi relevant* vectors, which are the lattice vectors $\mathbf{v}$ such that $\mathbf{v}/2$ is the center of a facet of $\mathcal{V}$. Since the Voronoi cell of a lattice can be shown to have at most $2(2^n - 1)$ facets, $\mathcal{V}$ can be expressed as a finite intersection of at most $2(2^n - 1)$ half-spaces. Throughout this dissertation, we assume that the Voronoi cell of a lattice is always described by such a list of half-spaces.

The relation between the Voronoi cell and CVPP is well known, and easy to explain. In CVPP, we want to find the lattice point $\mathbf{v}$ closest to a given target vector $\mathbf{t}$. It is easy to see that this is equivalent to finding a lattice vector $\mathbf{v}$ such that $\mathbf{t} - \mathbf{v}$ belongs to the (closed) Voronoi cell of the lattice. In other words, CVP can be equivalently formulated as the problem of finding a point in the set $(\Lambda + \mathbf{t}) \cap \bar{\mathcal{V}}$, where $\bar{\mathcal{V}}$ is the topological closure of $\mathcal{V}$. The idea of using the Voronoi cell to solve CVP is not new. For example, a simple greedy algorithm for CVPP based on the knowledge of the Voronoi cell of the lattice is given in [SFS09]. The idea behind this algorithm (called the Iterative Slicer) is to make $\mathbf{t}$ shorter and shorter by subtracting Voronoi relevant vectors from it. Notice that if $\mathbf{t} \notin \bar{H}_{\mathbf{v}}$, then the length of $\mathbf{t}$ can be reduced by subtracting $\mathbf{v}$ from $\mathbf{t}$. So, as long as $\mathbf{t}$ is outside $\bar{\mathcal{V}}$, we can make progress and find a shorter vector. Unfortunately, this simple strategy to solve CVPP using the Voronoi cell is not known to perform any better than previous algorithms. The work [SFS09] only proves that the algorithm terminates after a finite number of iterations, and a close inspection of the proof reveals that the best upper bound that can be derived using the methods of [SFS09] is of the form $n^{O(n)}$: the running time of the Iterative Slicer is bound by a volume argument, counting the number of lattice points within a sphere of radius $\|\mathbf{t}\|$, and this can be well above $2^{O(n)}$ or even $n^{O(n)}$.

In the next two paragraphs we first sketch our new algorithm to solve CVPP using the Voronoi cell $\mathcal{V}$ in time $2^{O(n)}$, and then we show how to use the CVPP algorithm to recursively implement the preprocessing function and compute the Voronoi cell $\mathcal{V}$. Since both the preprocessing and CVPP computation take time $2^{O(n)}$, combining the two pieces gives an algorithm to solve CVP (and a host of other lattice problems, like SVP, SIVP, etc.) without preprocessing.

The CVPP algorithm As already remarked, the goal of CVPP with target vector $\mathbf{t}$ can be restated as the problem of finding a point $\mathbf{t}' \in (\Lambda + \mathbf{t}) \cap \bar{\mathcal{V}}$ inside the Voronoi cell. (This point is also characterized as being a shortest vector in the set $\Lambda + \mathbf{t}$.) Therefore we need to find the shortest vector of the set $\Lambda + \mathbf{t}$. We follow an approach similar to the Iterative Slicer of [SFS09]. Given the list of relevant vectors, the algorithm generates a sequence of shorter and shorter vectors from $\Lambda + \mathbf{t}$, until it finds the shortest vector of the set. However, in order to bound the number of iterations, we introduce two important modifications to the greedy strategy of [SFS09]. First we reduce the general CVPP to a special case where the target vector is guaranteed to belong to twice the Voronoi cell $2\bar{\mathcal{V}}$. This can be done very easily by a polynomial time Turing reduction. Next, using properties of the geometry of the Voronoi cell, we show that it is possible to generate a sequence of shorter and shorter vectors in $\Lambda + \mathbf{t}$, with the additional property that all the vectors are inside $2\bar{\mathcal{V}}$. This allows to bound the length of the sequence by 2^n. For each vector of the sequence the algorithm spends $\tilde{O}(2^n)$ time, which gives a total time complexity of $\tilde{O}(2^{2n})$.

Computing the Voronoi cell We have sketched how to solve CVPP, given the Voronoi cell of the lattice. This leaves us with the problem of computing the Voronoi cell, a task typically considered even harder than CVP. To this end, we use a method of [AEVZ02] to compute the Voronoi cell of a lattice Λ, making 2^n calls to a CVPP oracle for the same lattice Λ. We combine this with a standard rank reduction procedure implicit in enumeration algorithms [Kan87b, HS07]. This procedure allows to solve CVPP in a lattice Λ of rank n making only $2^{O(n)}$ calls to a CVPP oracle for a properly chosen sub-lattice Λ' of rank $n-1$, which can be found in polynomial time. Combining all the pieces together we obtain an algorithm that computes the Voronoi cell of a lattice Λ by building a sequence of lattices $\Lambda_1 \subset \Lambda_2 \subset \cdots \subset \Lambda_n = \Lambda$ with rank $\text{rank}(\Lambda_i) = i$, and iteratively computing the Voronoi cell of Λ_{i+1} using the previously computed Voronoi cell of Λ_i. Since each $\mathcal{V}(\Lambda_i)$ can be computed from $\mathcal{V}(\Lambda_{i-1})$ is time $2^{O(n)}$ (by means of $2^{O(n)}$ CVPP computations, each of which takes $2^{O(n)}$ time), the total running time is $2^{O(n)}$.

1.4 Related Work

Most relevant work has already been described in the introduction. Here we mention a few more related papers and results that were inspired by our work. The closest vector problem with preprocessing has been investigated in several papers [Mic01a, FM03, Reg04b, CM06, AKKV05], mostly with the goal of showing that CVP is NP-hard even for fixed families of lattices, or devising polynomial time approximation algorithms (with super-polynomial time preprocessing). In summary, CVPP is NP-hard to approximate for any constant (or certain sub-polynomial) factors [AKKV05], and it can be approximated in polynomial time within a factor $\sqrt{n}$ [AR05], at least in its distance estimation variant. In this dissertation we use CVPP mostly as a building block to give a modular description of our CVP algorithm. We use CVPP to recursively implement the preprocessing function, and then to solve the actual CVP instance. It is an interesting open problem if a similar bootstrapping can be performed using the polynomial time CVPP approximation algorithm of [AR05], to yield a polynomial time solution to $\sqrt{n}$-approximate CVP.

The problem of computing the Voronoi cell of a lattice is of fundamental importance in many mathematics and communication theory applications. There are several formulations of this problem. In this dissertation we consider the problem of generating the list of facets ($(n-1)$-dimensional faces) of the Voronoi cell, as done also in [AEVZ02, SFS09]. Sometimes one wants to generate the list of vertices (i.e., zero dimensional faces), or even a complete description including all faces in dimension 1 to $n-1$. This is done in [VB96, SSV09], but it is a much more complex problem, as in general the Voronoi cell can have as many as $(n+1)! = n^{\Omega(n)}$ vertices, so they cannot be computed in single exponential time. In [SSV09] it is also shown that computing the number of vertices of the Voronoi cell of a lattice is $\#P$-hard.

There have been a number of papers that have used or extended our results. In [PS09] the authors give a variation of our List Sieve algorithm with time and space complexity $2^{2.465n}$ and $2^{1.233n}$. A promising approach for faster heuristic sieving is given in [WLTB], but without any experimental results. Finally in [DPV10] the authors give an *deterministic* $2^{O(n)}$ algorithm for SVP in any ℓ_p norm and n^n algorithm for integer programming using our Voronoi algorithm as a subroutine.

1.5 Open Problems

We have shown that CVP, SVP, SIVP and many other lattice problems can be solved in deterministic single exponential time. Many open problems remain. Here we list those that we think are most important or interesting.

Our algorithm uses exponential space. It would be nice to find an algorithm running in exponential time and polynomial, or even sub-exponential space.

Our algorithms are specific to the ℓ_2 norm. Many parts of the algorithm easily adapt to other norms as well, but it is not clear how to extend our results to other norms. The main technical problem is that the Voronoi cells in norms other than ℓ_2 are not necessarily convex. So, extending the algorithm to any other norm is likely to require some new idea. (Convexity of the Voronoi cell is used implicitly in several parts of our proof.) An important application of extending our algorithm to other norms is that it would immediately lead to single exponential time algorithms for integer programming [Kan87b]. There has been some advances towards this direction in [DPV10] where they give a *deterministic* $2^{O(n)}$ SVP algorithm for any norm, however it is still unclear if CVP in other norms (or integer programming) can be solved in $2^{O(n)}$.

As we have already discussed we can solve CVP in time $\tilde{O}(2^{2n})$. It is an open question if one can further decrease the constant in the exponent, with a better analysis of the CVPP algorithm and faster algorithms for $\textsc{Enum}_{2\bar{\mathcal{V}}}$. However, it is clear that our approach cannot possibly lead to constants in the exponent smaller than 1 (as achieved for example by randomized heuristics for SVP [NV08, MV10b]) just because the Voronoi cell of an n-dimensional lattice can have as many as 2^n facets. Still, it may be possible to extend our ideas to develop an algorithm with running time proportional to the number of Voronoi relevant vectors. This may give interesting algorithms for special lattices whose Voronoi cell has a small description. Another possible research direction is to develop practical variants of our algorithm that use only a sublist of Voronoi relevant vectors, at the cost of producing only approximate solutions to CVP.

It would be nice to extend our algorithm to yield a single exponential time solution to the covering radius problem, or equivalently, the problem of computing the diameter of the Voronoi cell of a lattice. In principle, this could be done by enumerating the vertices of the Voronoi cell, and selecting the longest, but this would not lead to a single exponential time algorithm because the number of such vertices can be as large as $n^{\Omega(n)}$. No NP-hardness proof for the covering radius problem in the ℓ_2 norm is known (but see [HR06] for NP-hardness results in ℓ_p norm for large p). Still, the problem seems quite hard: the covering radius problem is not even known to be in NP, and it is conjectured to be Π_2-hard [Mic04, GMR05] for small approximation factors. Counting the number of vertices of the Voronoi cell [SSV09] or the number of lattice points of a given length [Cha07] is also known to be $\#P$-hard.

Concerning sieve based algorithms we identify two possible lines of research. Firstly, improving the current algorithms. Bounding the running time of Gauss Sieve, or getting a faster heuristic would be very interesting. Another interesting question is whether the bound of Kabatiansky and Levenshtein [KL78] can be improved when the lattice is known to be cyclic, or has other interesting structure.

The second line of research is to use sieving as a subroutine for other algorithms that currently use enumeration techniques. Our early experimental results hint that sieving could solve SVPs in higher dimensions than we previously thought possible. It is especially interesting for example, to examine if such a tool can give better cryptanalysis algorithms.

Chapter 2

Preliminaries

In this chapter we give a fast exposition of the notation and the theoretical background required for the following chapters. We review basic facts and definitions about packing bounds, lattices, the Voronoi cells of lattices and the algorithmic problems studied in this thesis. For a more in-depth discussion of the background material the reader is referred to [MG02] for lattice theory and [CS98] for theory on the Voronoi cells of lattices and packings.

2.1 General Notation

The d-dimensional Euclidean space is denoted $\mathbb{R}^d$. We use bold lower case letters (e.g., $\mathbf{x}$) to denote vectors, and bold upper case letters (e.g., $\mathbf{M}$) to denote matrices. The ith coordinate of $\mathbf{x}$ is denoted x_i. For a set $S \subseteq \mathbb{R}^d$, $\mathbf{x} \in \mathbb{R}^d$ and $a \in \mathbb{R}$, we let $S + \mathbf{x} = \{\mathbf{y} + \mathbf{x} \colon \mathbf{y} \in S\}$ denote the translate of S by $\mathbf{x}$, and $aS = \{a\mathbf{y} \colon \mathbf{y} \in S\}$ denote the scaling of S by a. The Euclidean norm (also known as the ℓ_2 norm) of a vector $\mathbf{x} \in \mathbb{R}^d$ is $\|\mathbf{x}\| = (\sum_i x_i^2)^{1/2}$, and the associated distance is $\text{dist}(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|$. The linear space spanned by a set of vectors S is denoted $\text{span}(S) = \{\sum_i x_i \mathbf{s}_i \ : \ x_i \in \mathbb{R}, \mathbf{s}_i \in S\}$. The affine span of a set of vectors S is defined as $\mathbf{x} + \text{span}(S - \mathbf{x})$ for any $\mathbf{x} \in S$, and does not depend on the choice of $\mathbf{x}$. We will use $\phi_{\mathbf{x},\mathbf{y}}$ to refer to the angle between the vectors $\mathbf{x}, \mathbf{y}$.

The distance function is extended to sets in the usual way: $\text{dist}(\mathbf{x}, S) = \text{dist}(S, \mathbf{x}) = \min_{\mathbf{y} \in S} \text{dist}(\mathbf{x}, \mathbf{y})$. We often use matrix notation to denote sets of vectors. For example, matrix $\mathbf{S} \in \mathbb{R}^{n\times m}$ represents the set of n-dimensional vectors $\{\mathbf{s}_1, \ldots, \mathbf{s}_m\}$, where $\mathbf{s}_1, \ldots, \mathbf{s}_m$ are the columns of $\mathbf{S}$. We denote by $\|\mathbf{S}\|$ the maximum length of a vector in $\mathbf{S}$. The linear space spanned by a set of m vectors $\mathbf{S}$ is denoted $\text{span}(\mathbf{S}) = \{\sum_i x_i\mathbf{s}_i : x_i \in \mathbb{R} \text{ for } 1 \le i \le m\}$. For any set of n linearly independent vectors $\mathbf{S}$, we define the half-open parallelepiped $\mathcal{P}(\mathbf{S}) = \{\sum_i x_i\mathbf{s}_i : 0 \le x_i < 1 \text{ for } 1 \le i \le n\}$. Finally, we denote by $\mathcal{B}_n(\mathbf{x}, r)$ the closed n-dimensional Euclidean ball of radius r and center $\mathbf{x}$, $\mathcal{B}_n(\mathbf{x}, r) = \{\mathbf{w} \in \mathbb{R}^n : \|\mathbf{w} - \mathbf{x}\| \le r\}$. If no center is specified, then the center is zero $\mathcal{B}_n(r) = \mathcal{B}_n(\mathbf{0}, r)$.

We write log for the logarithm to the base 2, and $\log_q$ when the base q is any number possibly different from 2. We use $\omega(f(n))$ to denote the set of functions growing faster than $c \cdot f(n)$ for any $c > 0$. A function $e(n)$ is *negligible* if $e(n) < 1/n^c$ for any $c > 0$ and all sufficiently large n. We write $f = \tilde{O}(g)$ when $f(n)$ is bounded by $g(n)$ up to polylogarithmic factors, i.e., $f(n) \le \log^c g(n) \cdot g(n)$ for some constant c and all sufficiently large n.

2.2 Packing Bounds

The following classical packing bound is required for the analysis of our SVP algorithm of Chapter 3.

Theorem 2.2.1. *(Kabatiansky and Levenshtein [KL78]). Let $A(n, \phi_0)$ be the maximal size of any set C of points in $\mathbb{R}^n$ such that the angle between any two distinct vectors in C is at least ϕ_0. If $0 < \phi_0 < 63°$, then for all sufficiently large n, $A(n, \phi_0) \le 2^{cn}$ for*

$$c = -\frac{1}{2}\log(1 - \cos(\phi_0)) - 0.099.$$

Notice that when $\phi_0 = 60°$ this is equivalent to the kissing constant: $\tau_n = A(n, 60°) \le 2^{0.401n}$. We remark that these upper bounds are probably not tight, as there is no known matching lower bound. For example, for the case of the kissing constant, the best currently known lower bounds only proves that $\tau_n > 2^{0.2075+o(1)}$ [CS98].

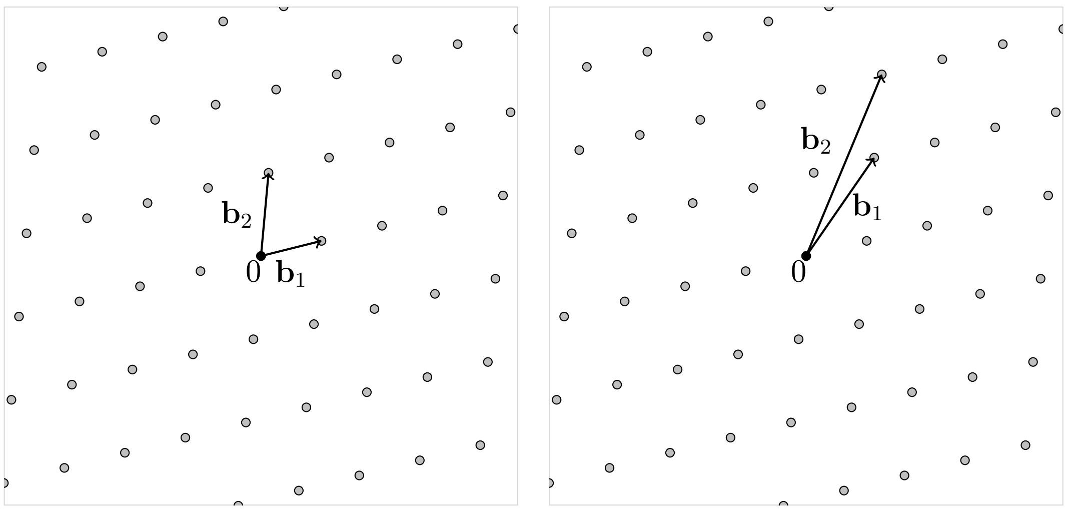

Figure 2.1: A two dimensional lattice and two different basis

2.3 Lattices

A d-dimensional *lattice* Λ of rank n is the set of all integer combinations

$$\left\{\sum_{i=1}^{n} x_i\mathbf{b}_i \colon x_i \in \mathbb{Z} \text{ for } 1 \leq i \leq n\right\}$$

of n linearly independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n$ in $\mathbb{R}^d$. The set of vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n$ is called a *basis* for the lattice. A basis can be represented by the matrix $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{d\times n}$ having the basis vectors as columns. The lattice generated by $\mathbf{B}$ is denoted $\mathcal{L}(\mathbf{B})$. Notice that $\mathcal{L}(\mathbf{B}) = \{\mathbf{Bx} \colon \mathbf{x} \in \mathbb{Z}^n\}$, where $\mathbf{Bx}$ is the usual matrix-vector multiplication. For any lattice basis $\mathbf{B}$ and point $\mathbf{x}$, there exists a unique vector $\mathbf{y} \in \mathcal{P}(\mathbf{B})$ such that $\mathbf{y} - \mathbf{x} \in \mathcal{L}(\mathbf{B})$. This vector is denoted $\mathbf{y} = \mathbf{x} \bmod \mathbf{B}$, and it can be computed in polynomial time given $\mathbf{B}$ and $\mathbf{x}$. A sub-lattice of $\mathcal{L}(\mathbf{B})$ is a lattice $\mathcal{L}(\mathbf{S})$ such that $\mathcal{L}(\mathbf{S}) \subseteq \mathcal{L}(\mathbf{B})$. The Gram-Schmidt orthogonalization of a basis $\mathbf{B}$ is the sequence of vectors $\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*$, where $\mathbf{b}_i^*$ is the component of $\mathbf{b}_i$ orthogonal to $\operatorname{span}(\mathbf{b}_1, \ldots, \mathbf{b}_{i-1})$. The *determinant* of a lattice $\det(\mathcal{L}(\mathbf{B}))$ is the (n-dimensional) volume of the fundamental parallelepiped $\mathcal{P}(\mathbf{B})$ and is given by $|\det(\mathbf{B})|$.

The *minimum distance* of a lattice Λ, denoted $\lambda(\Lambda)$, is the minimum distance between any two distinct lattice points, and equals the length of a nonzero shortest lattice vector:

$$\begin{aligned}\lambda(\Lambda) &= \min\{\operatorname{dist}(\mathbf{x},\mathbf{y}) : \mathbf{x} \neq \mathbf{y} \in \Lambda\} \\ &= \min\{\|\mathbf{x}\| : \mathbf{x} \in \Lambda \setminus \{\mathbf{0}\}\}\ .\end{aligned}$$

We often abuse notation and write $\lambda(\mathbf{B})$ instead of $\lambda(\mathcal{L}(\mathbf{B}))$.

The covering radius $\mu(\Lambda)$ of a lattice Λ, is the minimum distance r such that, for any point $\mathbf{p}$ in the linear span of Λ, $\operatorname{dist}(\mathbf{p}, \Lambda) \leq r$. The dual $\Lambda^\times$ of a lattice Λ is the set of all the vectors $\mathbf{x}$ in the linear span of Λ, that have integer scalar product $\langle \mathbf{x}, \mathbf{y} \rangle = \sum_i x_i \cdot y_i \in \mathbb{Z}$ with all lattice vectors $\mathbf{y} \in \Lambda$. Banaszczyk's transference theorem [Ban93] gives a very useful relation between the primal and dual lattices, namely $\mu(\Lambda) \cdot \lambda(\Lambda^\times) \leq n$.

2.4 Lattice Problems

In this thesis we are mostly concerned with the following problems:

- The shortest vector problem (SVP): given a basis $\mathbf{B}$, find a shortest nonzero vector in the lattice generated by $\mathbf{B}$.
- The closest vector problem (CVP): given a basis $\mathbf{B}$ and a target vector $\mathbf{t} \in \mathbb{R}^d$, find a lattice vector generated by $\mathbf{B}$ that is closest to $\mathbf{t}$.

Our results give algorithms of several other lattice problems like the Shortest Independent Vector Problem (SIVP), Subspace Avoiding Problem (SAP), the Generalized Closest Vector Problem (GCVP), and the Successive Minima Problem (SMP) considered in the lattice algorithms literature [BN09, Mic08]. The results for all problems other than CVP are obtained in a black-box way by reduction to CVP [Mic08], and we refer the reader to [GMSS99, BN09, Mic08] for details.

For simplicity we consider only inputs to lattice problems where all the entries in the basis matrix $\mathbf{B}$ have bit size polynomial in n, i.e., $\log(\|\mathbf{B}\|) = \operatorname{poly}(n)$. This allows to express the complexity of lattice problems simply as a function of a single parameter, the lattice rank n. All the results in this thesis can be easily adapted to the general case by introducing an explicit bound $\log \|\mathbf{B}\| \leq M$ on the size of the entries, and letting the time and space complexity bound depend polynomially in M.

2.5 Useful Lattice Algorithms

The following classical lattice algorithms are used in this thesis. The Nearest Plane algorithm [Bab86], on input a basis $\mathbf{B}$ and a target vector $\mathbf{t}$, finds a lattice point $\mathbf{v} \in \mathcal{L}(\mathbf{B})$ such that $\|\mathbf{v} - \mathbf{t}\| \leq (1/2)\sqrt{\sum_i \|\mathbf{b}_i^*\|^2}$. The LLL basis reduction algorithm [LLL82] runs in polynomial time, and on input a lattice basis, outputs a basis for the same lattice such that $\|\mathbf{b}_{i+1}^*\|^2 \geq \|\mathbf{b}_i^*\|^2/2$. (LLL reduced bases have other properties, but this is all we need here.)

We also use stronger basis reduction algorithms than LLL. In particular, we consider block basis reduction [GN08b, Sch87] of the dual $\Lambda^\times$ of the input lattice Λ. This algorithm on input a lattice basis, outputs a basis for the same lattice such that $1/\|\mathbf{b}_n^*\| \leq k^{n/k}\lambda(\Lambda^\times)$, using polynomially many queries to an SVP oracle for k-rank lattices, with $1 < k < n$. By Banaszczyk's transference theorem [Ban93], the inequality above gives: $\mu(\Lambda)/\|\mathbf{b}_n^*\| \leq nk^{n/k}$. This algorithm is required for Chapter 6 where we use directly the last inequality, without any further reference to the dual lattice or Banaszczyk's theorem. For a full description of these algorithms and the relation between primal and dual lattice bases the reader is referred to [GN08b].

2.6 The Voronoi Cell of a Lattice

2.6.1 Definitions and Facts

The (open) Voronoi cell of a lattice Λ is the set

$$\mathcal{V}(\Lambda) = \{\mathbf{x} \in \mathbb{R}^n : \forall \mathbf{v} \in \Lambda \setminus \{\mathbf{0}\}. \|\mathbf{x}\| < \|\mathbf{x} - \mathbf{v}\|\}$$

of all points that are closer to the origin than to any other lattice point. We also define the closed Voronoi cell $\bar{\mathcal{V}}$ as the topological closure of $\mathcal{V}$

$$\bar{\mathcal{V}}(\Lambda) = \{\mathbf{x} \in \mathbb{R}^n : \forall \mathbf{v} \in \Lambda \setminus \{\mathbf{0}\}. \|\mathbf{x}\| \leq \|\mathbf{x} - \mathbf{v}\|\}.$$

We omit Λ, and simply write $\mathcal{V}$, when the lattice is clear from the context. The Voronoi cell of a lattice point $\mathbf{v} \in \Lambda$ is defined similarly, and equals $\mathbf{v} + \mathcal{V}$. For any (lattice) point $\mathbf{v}$, define the half-space

$$H_{\mathbf{v}} = \{\mathbf{x} \colon \|\mathbf{x}\| < \|\mathbf{x} - \mathbf{v}\|\}.$$

Clearly, $\mathcal{V}$ is the intersection of $H_{\mathbf{v}}$ for all $\mathbf{v} \in \Lambda \setminus \{\mathbf{0}\}$. However, it is not necessary to consider all $\mathbf{v}$. The minimal set of lattice vectors V such that $\mathcal{V} = \bigcap_{\mathbf{v} \in V} H_{\mathbf{v}}$ is called the set of *Voronoi relevant* vectors. The Voronoi cell $\mathcal{V}$ is a polytope, and the Voronoi relevant vectors are precisely the centers of the ($(n-1)$ dimensional) facets of $2\mathcal{V}$.

The following observation connects the Voronoi cell, solving CVP and finding shortest vectors on cosets of the form $\Lambda + \mathbf{t}$. These problems will be used interchangeably throughout the paper.

Observation 2.6.1. *Let Λ be a lattice, $\mathcal{V}$ its Voronoi cell and $\mathbf{t}$, $\mathbf{t}_S$ vectors in the linear span of Λ. The following statements are equivalent:*

1. *$\mathbf{t}_S$ is a shortest vector in the coset $\Lambda + \mathbf{t}$*
2. *$\mathbf{t}_S$ belongs to $(\Lambda + \mathbf{t}) \cap \bar{\mathcal{V}}$*
3. *$\mathbf{c} = \mathbf{t} - \mathbf{t}_S$ is a lattice vector closest to $\mathbf{t}$.*

Proof. We prove that the three statements are equivalent by demonstrating the chain of implications $1 \to 2$, $2 \to 3$, $3 \to 1$. We start with the implication $1 \to 2$. Notice that for all $\mathbf{v} \in \Lambda$, $\mathbf{t}_S - \mathbf{v} \in \Lambda + \mathbf{t}$. Therefore from 1 we get that $\forall \mathbf{v} \in \Lambda . \|\mathbf{t}_S\| \leq \|\mathbf{t}_S - \mathbf{v}\|$ which is exactly the definition of the Voronoi cell. So $\mathbf{t}_S \in \bar{\mathcal{V}}$.

Next, we prove $2 \to 3$. If $\mathbf{t}_S \in \Lambda + \mathbf{t} \cap \bar{\mathcal{V}}$, then $\mathbf{c} = \mathbf{t} - \mathbf{t}_S$ is a lattice vector and $\mathbf{t} \in \bar{\mathcal{V}} + \mathbf{c}$. By definition of the Voronoi cell, this gives that $\mathbf{c}$ is a lattice vector closest to $\mathbf{t}$.

Finally, we show that $3 \to 1$. If $\mathbf{c}$ is a lattice vector closest to $\mathbf{t}$, then $\forall \mathbf{v} \in \Lambda . \|\mathbf{t} - \mathbf{c}\| \leq \|\mathbf{t} - \mathbf{v}\|$. Notice that the set $\cup_{\mathbf{v} \in \Lambda} \{\mathbf{t} - \mathbf{v}\}$ is exactly the coset $\Lambda + \mathbf{t}$, so $\mathbf{t}_S = \mathbf{t} - \mathbf{c}$ is a shortest vector in the coset $\Lambda + \mathbf{t}$. □

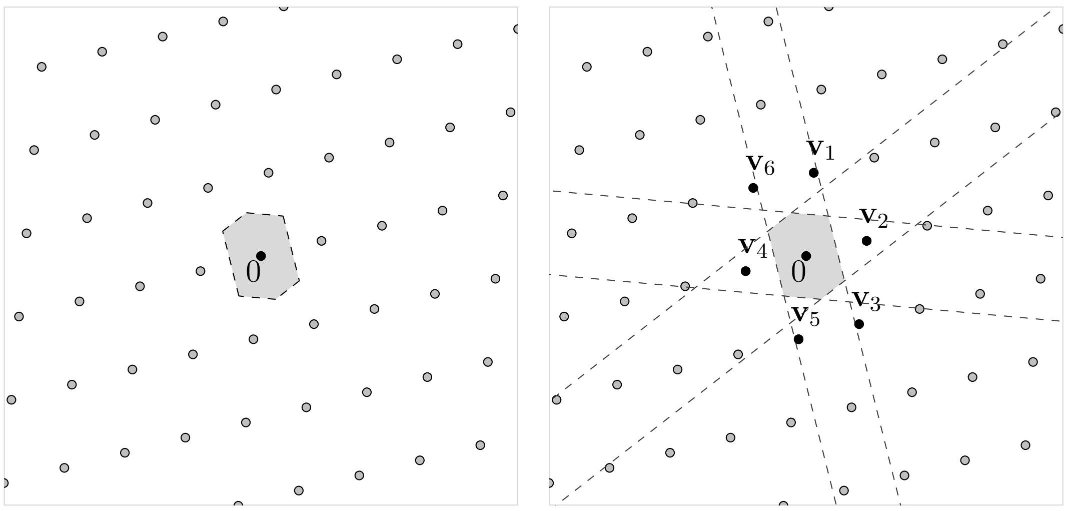

Figure 2.2: The Voronoi cell and the corresponding relevant vectors

We also need the following observation on the geometry of the Voronoi cell.

Observation 2.6.2. *Let $\bar{\mathcal{V}}$ be the closed Voronoi cell of a lattice Λ and $\mathbf{p}$ a point on the facet of $\bar{\mathcal{V}}$ defined by the relevant vector $\mathbf{v}$, namely $\{\mathbf{x} \in \bar{\mathcal{V}}\colon \|\mathbf{x}\| = \|\mathbf{x}-\mathbf{v}\|\}$. Then $\mathbf{p} - \mathbf{v}$ also belongs to the Voronoi cell.*

Proof. By Observation 2.6.1 we get that $\mathbf{p}$ is a shortest vector in the coset $\Lambda + \mathbf{p}$. Notice that $\|\mathbf{p}-\mathbf{v}\| = \|\mathbf{p}\|$ and the cosets $\Lambda+\mathbf{p}-\mathbf{v}$ and $\Lambda+\mathbf{p}$ are identical. We conclude that $\mathbf{p}-\mathbf{v}$ is a shortest vector in the coset $\Lambda+\mathbf{p}-\mathbf{v}$ and by Observation 2.6.1 it is a point in $\bar{\mathcal{V}}$. □

2.6.2 Relevant Vectors

In this work we describe the Voronoi cell by the set of its relevant vectors. In order to identify these vectors we use Voronoi's classical theorem:

Theorem 2.6.3 (Voronoi, see [CS98]:)**.** *Let Λ be a lattice and $\mathbf{v}$ a non-zero lattice vector. Then $\mathbf{v}$ is Voronoi relevant if and only if $\pm\mathbf{v}$ are the unique shortest vectors of the coset $2\Lambda + \mathbf{v}$.*

We use a number of techniques and observations derived directly from Voronoi's work. One of these techniques is partitioning the lattice Λ into 2^n cosets modulo 2Λ.

Observation 2.6.4. *Let* $\mathbf{B}$ *be a basis for an* n*-rank lattice* Λ*.* Λ *can be partitioned to exactly* 2^n *cosets of the form* $C_{\mathbf{B},\mathbf{p}} = 2\Lambda + \mathbf{B} \cdot \mathbf{p}$ *with* $\mathbf{p} \in \{0,1\}^n$*.*

Combining Voronoi's theorem with the observation above gives the following corollary.

Corollary 2.6.5. *Let* $\mathbf{B}$ *be a basis for an* n*-rank lattice* Λ *and the cosets* $C_{\mathbf{B},\mathbf{p}}$ *as defined above. If* $\mathbf{s_p}$ *is a shortest vector for* $C_{\mathbf{B},\mathbf{p}}$ *then the set* $R = \bigcup_{\mathbf{p}\in\{0,1\}^n\setminus\{\mathbf{0}\}}\{\pm\mathbf{s_p}\}$ *contains all the relevant vectors.*

Proof. Let $\mathbf{v}$ be a relevant vector of Λ. By Observation 2.6.4 there exists a $\mathbf{p} \in \{0,1\}^n$ such that $\mathbf{v} \in C_{\mathbf{B},\mathbf{p}}$. If $\mathbf{p} \neq \mathbf{0}$ by Voronoi's theorem $\pm\mathbf{v}$ are the unique shortest vectors of $C_{\mathbf{B},\mathbf{p}}$, so they are exactly the vectors $\pm\mathbf{s_p}$ included in R. If $\mathbf{p} = \mathbf{0}$, the shortest vector of $C_{\mathbf{B},\mathbf{0}}$ is $\mathbf{0}$ and $\mathbf{v}$ cannot be a relevant vector. □

Finally we need a theorem from [Hor96] that characterizes the lattice points on the boundary of $2\bar{\mathcal{V}}$.

Theorem 2.6.6. *Any lattice point* $\mathbf{u}$ *in the boundary of* $2\bar{\mathcal{V}}$ *is a sum of relevant vectors that are orthogonal with each other.*

Chapter 3

Sieving Algorithms

In this chapter we present and analyze new algorithms for the shortest vector problem in arbitrary lattices that both improve the best previously known worst-case asymptotic complexity and also have the advantage of performing pretty well in practice, thereby reducing the gap between theoretical and practical algorithms. More specifically, we present:

- *List Sieve:* A new probabilistic algorithm that provably finds the shortest vector in any n dimensional lattice (in the worst case, and with high probability) in time $\tilde{O}(2^{3.199n})$ and space $\tilde{O}(2^{1.325n})$ (or space $2^{1.095n}$ and still $2^{O(n)}$ time), improving the $\tilde{O}(2^{5.9n})$-time and $\tilde{O}(2^{2.95n})$-space complexity bounds of the asymptotically best previously known algorithm [AKS01, NV08], and
- *Gauss Sieve:* A practical variant of List Sieve that admits much better space bounds, and outperforms the best previous practical implementation [NV08] of the AKS sieve [AKS01].

The space complexity of our second algorithm can be bounded by $\tilde{O}(\tau_n)$, where τ_n is the so called "kissing" constant in n-dimensional space, i.e., the maximum number of equal n-dimensional spheres that can be made to touch another sphere, without touching each other. The best currently known lower and upper bounds on the kissing constant are $2^{(0.2075+o(1))n} < \tau_n < 2^{(0.401+o(1))n}$ [CS98]. Based on these bounds we can conclude that the worst-case space complexity of our second algorithm is certainly bounded by $2^{0.402n}$. Moreover, in practice we should expect the space complexity to be near $2^{0.21n}$, because finding a family of lattices for which the algorithm uses more than $2^{0.21n}$ space would imply denser arrangements of hyperspheres than currently known, a long standing open problem in the study of spherical codes. So, input lattices for which our algorithm uses more than $2^{0.21n}$ space either do not exist (i.e., the worst-case space complexity is $2^{0.21n}$), or do not occur in practice because they are very hard to find. The practical experiments reported in Section 3.4 are consistent with our analysis, and suggest that the space complexity of our second algorithm is indeed near $2^{0.21n}$. Unfortunately, we are unable to prove any upper bound on the running time of our second algorithm, but our experiments suggest that the algorithm runs in time $2^{0.52n}$.

The rest of the chapter is organized as follows. In Section 3.1 we give an overview of our new algorithms and our contribution. In Section 3.2 we describe our new algorithms and state theorems about their complexity. Section 3.3 gives the proof for the time and space complexity asymptotic bounds, while Section 3.4 contains our experimental results.

3.1 Overview

In order to describe the main idea behind our algorithms, we first recall how the sieve algorithm of [AKS01] works. The algorithm starts by generating a large (exponential) number of random lattice points P within a large (but bounded) region of space. Informally, the points P are passed through a sequence of finer and finer "sieves", that produce shorter and shorter vectors, while "wasting" some of the sieved vectors along the way. (The reader is referred to the original article [AKS01] as well as the recent analysis [NV08] for a more detailed and technical description of the AKS sieve.)

While using many technical ideas from [AKS01, NV08], our algorithms depart from the general strategy of starting from a large pool P of (initially long) lattice vectors, and obtaining smaller and smaller sets of shorter vectors. Instead, our algorithms start from an initially empty list L of points, and increase the length of the list by appending new lattice points to it. In our first algorithm *List Sieve*,

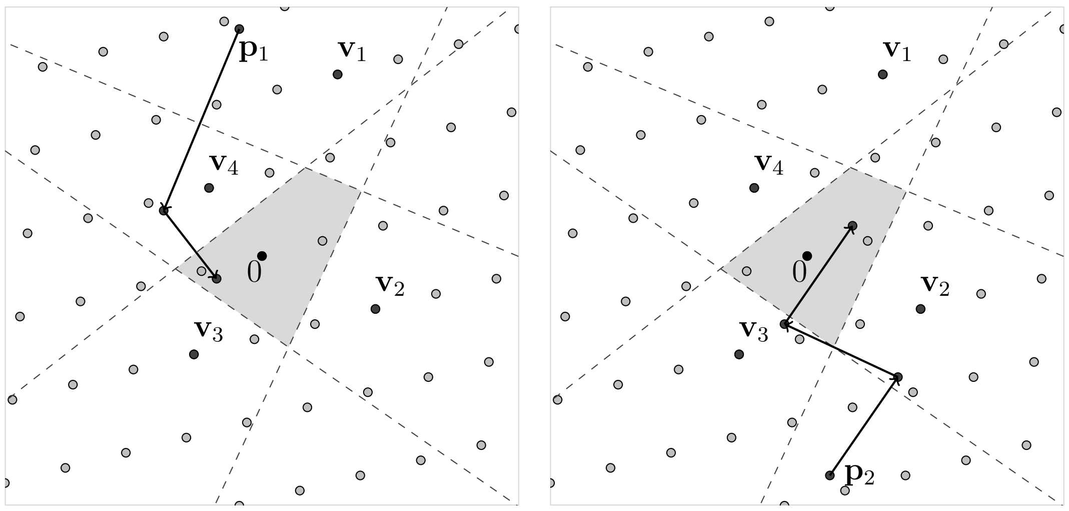

Figure 3.1: Reducing two points $\mathbf{p}_1, \mathbf{p}_2$ with the list $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4\}$

the points in the list never change: we only keep adding new vectors to the list. Before a new point $\mathbf{v}$ is added to the list, we attempt to reduce the length of $\mathbf{v}$ as much as possible by subtracting the vectors already in the list from it. Reducing new lattice vectors against the vectors already in the list allows us to prove a lower bound on the angle between any two list points of similar norm. This lower bound on the angle between list points allows us to apply the linear programming bound for spherical codes of Kabatiansky and Levenshtein [KL78] to prove that the list L cannot be too long. The upper bound on the list size then easily translates to corresponding upper bounds on the time and space complexity of the algorithm.

Similarly to previous work [AKS01, NV08], in order to prove that the algorithm produces non-zero vectors, we employ a now standard perturbation technique. Specifically, instead of generating a random lattice point $\mathbf{v}$ and reducing it against the vectors already in the list, we generate a perturbed lattice point $\mathbf{v} + \mathbf{e}$ (where $\mathbf{e}$ is a small error vector), and reduce $\mathbf{v} + \mathbf{e}$ instead. The norm of the error $\mathbf{e}$ is large enough, so that the lattice point $\mathbf{v}$ is not uniquely determined by $\mathbf{v} + \mathbf{e}$. This uncertainty about $\mathbf{v}$ allows to easily prove that after reduction against the list, the vector $\mathbf{v}$ is not too likely to be zero. Unfortunately the introduction of errors reduces the effectiveness of sieving and increases the space complexity.

In practice, as shown in [NV08], variants of sieving algorithms without errors, perform much better, but lack theoretical time bounds. *Gauss Sieve* is a practical variant of *List Sieve* without errors which incorporates a new heuristic technique. Beside reducing new lattice points $\mathbf{v}$ against the points already in the list L, the algorithm also reduces the points in L against $\mathbf{v}$, and against each other. As a result, the list L has the property that any pair of vectors in L forms a Gauss reduced basis. It follows from the properties of Gauss reduced bases that the angle between any two list points is at least $\pi/3$, that is the list forms a good spherical code. In particular, the list length never exceeds the kissing constant τ_n, which is defined as the highest number of points that can be placed on a sphere, while keeping the minimal angle between any two points at least $\pi/3$.[1] As already discussed, this allows to bound the space complexity of our second algorithm by $2^{0.402n}$ in theory, or $2^{0.21n}$ in practice. Unfortunately, we are unable to bound the running time of this modified algorithm, as we don't know how to prove that it produces nonzero vectors. However, the algorithm seems to work very well in practice, and outperforms the best previously known variants/implementations of the AKS Sieve [NV08] both in theory (in terms of provable space bounds,) and in practice (in terms of experimentally observed space and time requirements).

3.1.1 Contribution

Our contribution is both analytical and algorithmic. On the analytical side, we explicitly introduce the use of sphere packing bounds in the study of sieve algorithms for lattice problems. Such usage was already implicit in previous work, but somehow obfuscated by the complexity of previous algorithms and analyses. Our simpler algorithms and explicit connection to sphere packings, allows us to relate the performance of sieve algorithms to well studied quantities in the theory of spherical codes, and make use of the best bounds known to date [KL78]. We remark that these bound are broadly applicable, and can be used to improve the analysis of previous sieving algorithms [AKS01, NV08] as well. However, this is

[1] The name "kissing" constant originates from the fact that $\pi/3$ is precisely the minimal angle between the centers of two nonintersecting equal spheres that touch (kiss) a third sphere of the same radius.

not the only source of improvement. In Section A.1 we sketch how to apply sphere packing bounds to the analysis of the original sieve algorithm [AKS01, NV08], and show that, even using the powerful linear programming bound of [KL78], only yields provable space and time complexity bounds approximately equal to $2^{1.97n}$ and $2^{3.4n}$, which is still worse than the performance of our theoretical algorithm by an exponential factor.

On the algorithmic side, we introduce a new sieving strategy, as described in the previous section. While at first sight the new strategy may look like a simple reordering of the instructions executed by the original sieve of Ajtai, Kumar and Sivakumar [AKS01], there are deeper algorithmic differences that lead to a noticeable reduction in both the provable space and time complexity. The main source of algorithmic improvement is the way our new sieving strategy deals with useless points, i.e., samples that potentially yield zero vectors in the original sieve. In our algorithms these vectors are immediately recognized and discarded. In the original sieve algorithm these vectors are generated at the outset, and remain undetected during the entire execution, until the algorithm reaches its final stage. Both in the analysis of [AKS01, NV08] and in this chapter, such useless points are potentially an overwhelming fraction of all samples, leading to a noticeable difference in performance.

3.2 Algorithms

In this section we describe our two algorithms for the shortest vector problem. For simplicity we assume that these algorithms:

- take as input a *square* basis $\mathbf{B} \in \mathbb{N}^{n\times n}$,
- a parameter $\mu \in [\lambda_1(\mathbf{B}), 1.01 \cdot \lambda_1(\mathbf{B})]$ which approximates the length of the shortest nonzero lattice vector within a constant factor 1.01, and
- are only required to produce a nonzero lattice vector of length bounded by μ, possibly larger than λ_1.

This is without loss of generality because any such algorithm can be turned, using standard techniques, into an algorithm that solves SVP exactly by trying only polynomially many possible values for μ.

In Section 3.2.1, we describe *List Sieve*, while in Section 3.2.2 we give the *Gauss Sieve*, a practical variant of the List Sieve with much better *provable* worst-case space complexity bound $\tilde{O}(\tau_n)$, where τ_n is the kissing constant in dimension n.

3.2.1 The List Sieve

Algorithm 3.1: The List Sieve Algorithm

```
function ListSieve($\mathbf{B}$, $\mu$)
    $L \leftarrow \{\mathbf{0}\}$, $\delta \leftarrow 1 - 1/n$
    $\xi \leftarrow 0.685$, $K \leftarrow 2^{cn}$            ▷ The choice of $\xi$, $c$ is explained in the analysis
    for $i = 0$ to $K$ do
        $(\mathbf{p}_i, \mathbf{e}_i) \leftarrow$ Sample$(\mathbf{B}, \xi\mu)$
        $\mathbf{v}_i \leftarrow$ ListReduce $(\mathbf{p}_i, L, \delta) - \mathbf{e}_i$
        if $(\mathbf{v}_i \notin L)$ then
            if $\exists \mathbf{v}_j \in L : \|\mathbf{v}_i - \mathbf{v}_j\| \leq \mu$ then
                return $\mathbf{v}_i - \mathbf{v}_j$
            $L \leftarrow L \cup \{\mathbf{v}_i\}$
    return $\perp$
function Sample($\mathbf{B}$, $d$)
    $\mathbf{e} \overset{\$}{\leftarrow} \mathcal{B}_n(d)$
    $\mathbf{p} \leftarrow \mathbf{e} \bmod \mathbf{B}$
    return $(\mathbf{p}, \mathbf{e})$
function ListReduce($\mathbf{p}$, $L$, $\delta$)
    while $(\exists \mathbf{v}_i \in L : \|\mathbf{p} - \mathbf{v}_i\| \leq \delta\|\mathbf{p}\|)$ do
        $\mathbf{p} \leftarrow \mathbf{p} - \mathbf{v}_i$
    return $\mathbf{p}$
```

The *List Sieve* algorithm works by iteratively building a list L of lattice points. At every iteration, the algorithm attempts to add a new point to the list. Lattice points already in the list are never modified or removed. The goal of the algorithm is to produce shorter and shorter lattice vectors, until two lattice vectors within distance μ from each other are found, and a lattice vector achieving the target norm can be computed as the difference between these two vectors. At every iteration, a new lattice point is generated by first picking a (somehow random, in a sense to be specified) lattice point $\mathbf{v}$, and reducing the length of $\mathbf{v}$ as much as possible by repeatedly subtracting from it the lattice vectors already in the list L when appropriate. Finally, once the length of $\mathbf{v}$ cannot be further reduced, the vector $\mathbf{v}$ is included in the list.

The main idea behind our algorithm design and analysis is that reducing $\mathbf{v}$ with the vector list L ensures that no two points in the list are close to each other. [2] We use this fact to bound from below the angle between any two list points of similar norm and use Theorem 2.2.1 to prove an upper bound on the size of the list L. This immediately gives upper bounds on the space complexity of the algorithm. Moreover, if at every iteration we were to add a new lattice point to the list, we could immediately bound the running time of the algorithm as roughly quadratic in the list size, because the size of L would also be an upper bound on the number of iterations, and each iteration takes time proportional[3] to the list size $|L|$. The problem is that some iterations might give collisions, lattice vectors $\mathbf{v}$ that already belong to the list. These iterations leave the list L unchanged, and as a result they just waste time. So the main hurdle in the time complexity analysis is bounding the probability of getting collisions.

This is done using the same method as in the original sieve algorithm [AKS01]: instead of directly working with a lattice point $\mathbf{v}$, we use a perturbed version of it $\mathbf{p} = \mathbf{v} + \mathbf{e}$, where $\mathbf{e}$ is a small random error vector of length $\|\mathbf{e}\| \leq \xi\mu$ for an appropriate value of $\xi > 0.5$. As before the length of $\mathbf{p}$ is reduced using list points, but instead of adding $\mathbf{p}$ to the list we add the corresponding lattice vector

[2]This is because if $\mathbf{v}$ is close to a list vector $\mathbf{u} \in L$, then $\mathbf{u}$ is subtracted from $\mathbf{v}$ before $\mathbf{v}$ is considered for inclusion in the list.

[3]Each iteration involves a small (polynomial) number of scans of the current list L.

$\mathbf{v} = \mathbf{p} - \mathbf{e}$. We will see that some points $\mathbf{p} = \mathbf{v}_1 + \mathbf{e}_1 = \mathbf{v}_2 + \mathbf{e}_2$ correspond to two different lattice points $\mathbf{v}_1$, $\mathbf{v}_2$ at distance precisely $\|\mathbf{v}_1 - \mathbf{v}_2\| = \lambda_1(\mathbf{B})$ from each other. For example, if $\mathbf{s}$ is the shortest nonzero vector in the lattice, then setting $\mathbf{p} = -\mathbf{e}_1 = \mathbf{e}_2 = \mathbf{s}/2$ gives such a pair of points $\mathbf{v}_1 = \mathbf{0}, \mathbf{v}_2 = \mathbf{s}$. The distance between two points in L is greater than μ or else the algorithm terminates and as a result at most one of the possible lattice vectors $\mathbf{v}_1, \mathbf{v}_2$ is in the list. This property can be used to get an upper bound on the probability of getting a collision.

Unfortunately the introduction of perturbations comes at a cost. As we have discussed above, sieving produces points that are far from L and as a result we can prove a lower bound on the angles between points of similar norm. Indeed after sieving with L the point $\mathbf{p}$ will be far from any point in L. However the point that is actually added to the list is $\mathbf{v} = \mathbf{p} - \mathbf{e}$ which can be closer to L than $\mathbf{p}$ by as much as $\|\mathbf{e}\| \leq \xi\mu$. That makes the resulting bounds on the angles worse. This worsening gets more and more significant as the norm of the points gets smaller. Fortunately we can also bound the distance between points in L by μ, which gives a good lower bound on the angles between shorter points. The space complexity of the algorithm is determined by combining these two bounds to obtain a global bound on the angle between any two points of similar norm, for any possible norm.

The complete pseudo-code of the *List Sieve* is given as Algorithm 3.1. Here we explain the main operations performed by the algorithm.

Sampling. The pair $(\mathbf{p}, \mathbf{e})$ is chosen picking $\mathbf{e}$ uniformly at random within a ball of radius $\xi\mu$, and setting $\mathbf{p} = \mathbf{e} \bmod \mathbf{B}$. This ensures that, by construction, the ball $\mathcal{B}(\mathbf{p}, \xi\mu)$ contains at least one lattice point $\mathbf{v} = \mathbf{p} - \mathbf{e}$. Moreover, the conditional distribution of $\mathbf{v}$ (given $\mathbf{p}$) is uniform over all lattice points in this ball. Notice also that for any $\xi > 0.5$, the probability that $\mathcal{B}(\mathbf{p}, \xi\mu)$ contains more than one lattice point is strictly positive: if $\mathbf{s}$ is a lattice vector of length $\lambda_1(\mathbf{B})$, then the intersection of $\mathcal{B}(\mathbf{0}, \xi\mu)$ and $\mathcal{B}(\mathbf{s}, \xi\mu)$ is not empty, and if $\mathbf{e}$ falls within this intersection, then both $\mathbf{v}$ and $\mathbf{v} + \mathbf{s}$ are within distance $\xi\mu$ from $\mathbf{p}$.

List reduction. The vector $\mathbf{p}$ is reduced by subtracting (if appropriate) lattice vectors in L from it. The vectors from L can be subtracted in any order. Our analysis applies independently from the strategy used to choose vectors from L. For each $\mathbf{v} \in L$, we subtract $\mathbf{v}$ from $\mathbf{p}$ only if $\|\mathbf{p}-\mathbf{v}\| < \|\mathbf{p}\|$. Notice that reducing $\mathbf{p}$ with respect to $\mathbf{v}$ may make $\mathbf{p}$ no longer reduced with respect to some other $\mathbf{v}' \in L$. So, all list vectors are repeatedly considered until the length of $\mathbf{p}$ can no longer be reduced. Since the length of $\mathbf{p}$ decreases each time it gets modified, and $\mathbf{p}$ belongs to a discrete set $\mathcal{L}(\mathbf{B}) - \mathbf{e}$, this process necessarily terminates after a finite number of operations. In order to ensure *fast* termination, as in the LLL algorithm, we introduce a slackness parameter $\delta < 1$, and subtract $\mathbf{v}$ from $\mathbf{p}$ only if this reduces the length of $\mathbf{p}$ by at least a factor δ. As a result, the running time of each invocation of the list reduction operation is bounded by the list size $|L|$ times the logarithm (to the base $1/\delta$) of the length of $\mathbf{p}$. For simplicity, we take $\delta(n) = 1 - 1/n$, so that the number of iterations is bounded by a polynomial $\log(n\|\mathbf{B}\|)/\log(1-1/n)^{-1} = n^{O(1)}$.

Termination. When the algorithm starts it computes the maximum number K of samples it is going to use. If a lattice vector achieving norm at most μ is not found after reducing K samples, the algorithm outputs $\perp$. In Section 3.3 we will show how to choose K so that if $\lambda_1(\mathbf{B}) \leq \mu \leq 1.01\lambda_1(\mathbf{B})$ the algorithm finds a vector with norm bounded by μ with probability exponentially close to 1.

Now we are ready to state our main theorem.

Theorem 3.2.1. *Let ξ be a real number such that $0.5 < \xi < 0.7$ and $c_1(\xi) = \log(\xi + \sqrt{\xi^2+1}) + 0.401$, $c_2(\xi) = 0.5\log(\xi^2/(\xi^2 - 0.25))$. List Sieve solves* SVP *in the worst case, with probability exponentially close to 1, in space $\tilde{O}(2^{c_1 n})$, and time $\tilde{O}(2^{(2\cdot c_1 + c_2)n})$.*

Proof. Immediately follows from Theorem 3.3.3 and Theorem 3.3.1, proved in Section 3.3.

□

Setting the parameter ξ: A smaller ξ gives smaller perturbations and reduces the space complexity. On the other hand we need $\xi > 0.5$ to bound the collision probability and consequently the running time. By choosing ξ arbitrarily close to

0.5, we can achieve space complexity 2^{cn} for any $c > \log((1+\sqrt{5})/2) + 0.401 \approx 1.095$, and still keep exponential running time $2^{O(n)}$, but with a large constant in the exponent. At the cost of slightly increasing the space complexity, we can substantially reduce the running time. The value of ξ that yields the best running time is $\xi \simeq 0.685$ which yields space complexity $< 2^{1.325n}$ and time complexity $< 2^{3.199n}$.

3.2.2 The Gauss Sieve

Algorithm 3.2: The Gauss Sieve Algorithm

```
function GaussSieve($\mathbf{B}$, $\mu$)
    $L \leftarrow \{\mathbf{0}\}$, $S \leftarrow \{\ \}, K \leftarrow 0$
    while $K < c$ do
        if $S$ is empty then
            $S.push($ SampleKlein($\mathbf{B}$) $)$
        $\mathbf{v}_{new} \leftarrow$GaussReduce($S.pop(), L, S$)
        if $(\mathbf{v}_{new} = \mathbf{0})$ then
            $K \leftarrow K + 1$
        else
            $L \leftarrow L \cup \{\mathbf{v}_{new}\}$
    return Shortest vector of $L$.
function GaussReduce($\mathbf{p}$, $L$, $S$)
    while $(\exists \mathbf{v}_i \in L : \|\mathbf{v}_i\| \leq \|\mathbf{p}\| \wedge \|\mathbf{p} - \mathbf{v}_i\| \leq \|\mathbf{p}\|)$ do
        $\mathbf{p} \leftarrow \mathbf{p} - \mathbf{v}_i$
    while $(\exists \mathbf{v}_i \in L : \|\mathbf{v}_i\| > \|\mathbf{p}\| \wedge \|\mathbf{v}_i - \mathbf{p}\| \leq \|\mathbf{v}_i\|)$ do
        $L \leftarrow L \setminus \{\mathbf{v}_i\}$
        $S.push(\mathbf{v}_i - \mathbf{p})$
    return $\mathbf{p}$
```

As we have discussed in the previous section the introduction of perturbations substantially increases the space requirements. In an attempt to make sieving algorithms practical Nguyen and Vidick in [NV08] have proposed a heuristic variant of AKS that does not use perturbations. Their experiments show that in practice collisions are not common and the algorithm performs quite well. We also introduce a practical variant of our algorithm without perturbations which we call the *Gauss Sieve*.

The *Gauss Sieve* follows the same general approach of building a list of shorter and shorter lattice vectors, but when a new vector $\mathbf{v}$ is added to the list, not only we reduce the length of $\mathbf{v}$ using the list vectors, but we also attempt to reduce the length of the vectors already in the list using $\mathbf{v}$. In other words, if $\min(\|\mathbf{v} \pm \mathbf{u}\|) < \max(\|\mathbf{v}\|, \|\mathbf{u}\|)$, then we replace the longer of $\mathbf{v}, \mathbf{u}$ with the shorter of $\mathbf{v} \pm \mathbf{u}$. As a result, the list L always consists of vectors that are pairwise reduced, i.e., they satisfy the condition $\min(\|\mathbf{u} \pm \mathbf{v}\|) \geq \max(\|\mathbf{u}\|, \|\mathbf{v}\|)$. This is precisely the defining condition of reduced basis achieved by the Gauss/Lagrange basis reduction algorithm for two dimensional lattices, hence the name of our algorithm.

It is well known that if $\mathbf{u}, \mathbf{v}$ is a Gauss reduced basis, then the angle between $\mathbf{u}$ and $\mathbf{v}$ is at least $\pi/3$ (or 60 degrees). As a result, the maximum size of the list (and space complexity of the algorithm) can be immediately bounded by the kissing number τ_n. Unfortunately, we are unable to prove any bounds on the probability of collisions or the running time. Notice that collisions are problematic because they reduce the list size, possibly leading to nonterminating executions that keep adding and removing vectors from the list. In practice (see experimental results in Section 3.4) this does not occur, and the running time of the algorithm seems to be between quadratic and cubic in the list size, but we do not know how to prove any worst-case upper bound.

As in [NV08], we do not use perturbations, and just choose $\mathbf{p} = \mathbf{v}$ at random using Klein's randomized rounding algorithm [Kle00] (denoted *SampleKlein* in the description of the algorithm). However since we cannot prove anything about running time, the choice of sampling algorithm is largely arbitrary. In the context of the Gauss Sieve, not using perturbation has the main practical advantage of allowing to work with lattice points only. This allows an integer only implementation of the algorithm (except possibly for the sampling procedure which may still internally use floating point numbers).

The *Gauss Sieve* pseudo-code is shown as Algorithm 3.2. The algorithm uses a stack or queue data structure S to temporarily remove vectors from the list L. When a new point $\mathbf{v}$ is reduced with L, the algorithm checks if any point in L can be reduced with $\mathbf{v}$. All such points are temporarily removed from L, and inserted in S for further reduction. The *Gauss Sieve* algorithm reduces the points in S with the current list before inserting them in L. When the stack S is empty, all list points are pairwise reduced, and the *Gauss Sieve* can sample a new lattice point $\mathbf{v}$ for insertion in the list L. Unfortunately, we cannot bound the number of samples required to find a nonzero shortest vector with high probability. As a result we have to use a heuristic termination condition. Based on experiments a good heuristic is to terminate after a certain number $c(n)$ of collisions has occurred (see section 3.4).

3.3 Analysis of *List Sieve*

In this section we prove time and space upper bounds for the *List Sieve* algorithm, assuming it is given as a hint a value μ such that $\lambda_1 \leq \mu \leq 1.01 \cdot \lambda_1$. In subsection 3.3.1 we use the fact that the list points are far apart to get an upper bound N on the list size. Then in subsection 3.3.2 we prove that collisions are not too common. We use this fact to prove that after a certain number of samples are processed, the algorithm finds a nonzero vector with norm less than or equal to μ with high probability.

3.3.1 Space complexity

Theorem 3.3.1. *The number of points in L is bounded from above by $N = \mathrm{poly}(n) \cdot 2^{c_1 n}$ where*

$$c_1 = \log(\xi + \sqrt{\xi^2 + 1}) + 0.401.$$

Proof. Let $\mathbf{B}$ be the input basis, and μ be the target length of the List Sieve algorithm. Notice that as soon as the algorithm finds two lattice vectors within distance μ from each other, the algorithm terminates. So, the distance between any two points in the list L must be greater than μ. In order to bound the size of L, we divide the list points into groups, according to their length, and bound the size of each group separately. Consider all list points belonging to a ball of radius $\mu/2$

$$S_0 = L \cap \mathcal{B}(\mu/2).$$

Clearly S_0 has size at most 1, because the distance between any two points in $\mathcal{B}(\mu/2)$ is bounded by μ. Next, divide the rest of the space into a sequence of spherical shells

$$S_i = \{\mathbf{v} \in L : \gamma^{i-1}\mu/2 < \|\mathbf{v}\| \leq \gamma^i \mu/2\}$$

for $i = 1, 2, \dots$ and $\gamma = 1+1/n$. Notice that we only need to consider a polynomial number of spherical shells $\log_\gamma(2n\|\mathbf{B}\|/\mu) = O(n^c)$, because all list points have length at most $n\|\mathbf{B}\|$. We will prove an exponential bound on the number of points in each spherical shell. The same bound holds (up to polynomial factors) for the total number of points in the list L.

So, fix a spherical shell $S_i = \{\mathbf{v} \in L\colon R < \|\mathbf{v}\| \leq \gamma R\}$ for some $R = \gamma^{i-1}\mu/2$. Consider two arbitrary points $\mathbf{v}_a, \mathbf{v}_b \in S_i$ and let $\phi_{\mathbf{v}_a,\mathbf{v}_b}$ be the angle between them. We will show that

$$\cos(\phi_{\mathbf{v}_a,\mathbf{v}_b}) \leq 1 - \frac{1}{2(\xi + \sqrt{\xi^2+1})^2} + o(1). \tag{3.1}$$

The upper bound on $\cos(\phi)$ is greater than 0.5. (Equivalently the minimum angle is less than 60°.) Therefore, we can safely use Theorem 2.2.1 with the bound (3.1), and conclude that the number of points in S_i is at most $2^{c_1 n}$ where

$$\begin{aligned} c_1 &= -\frac{1}{2}\log(1-\cos(\phi)) - 0.099 \\ &\leq \log\left(\sqrt{2}(\xi + \sqrt{\xi^2+1})\right) - 0.099 \\ &= \log(\xi + \sqrt{\xi^2+1}) + 0.401. \end{aligned}$$

as stated in the theorem. It remains to prove (3.1).

We use the fact that

$$\cos(\phi_{\mathbf{v}_a,\mathbf{v}_b}) = \frac{\mathbf{v}_a \cdot \mathbf{v}_b}{\|\mathbf{v}_a\|\|\mathbf{v}_b\|}, \tag{3.2}$$

to transform any upper bound on $\mathbf{v}_a \cdot \mathbf{v}_b$ into a corresponding upper bound on $\cos(\phi_{\mathbf{v}_a,\mathbf{v}_b})$. We give two bounds on $\mathbf{v}_a \cdot \mathbf{v}_b$. First remember that $\|\mathbf{v}_a - \mathbf{v}_b\| > \mu$. Squaring the terms yields $\mathbf{v}_a \cdot \mathbf{v}_b < (\mathbf{v}_a^2 + \mathbf{v}_b^2 - \mu^2)/2$ and we can use (3.2) to get

$$\begin{aligned}\cos(\phi_{\mathbf{v}_a,\mathbf{v}_b}) &\leq \frac{\mathbf{v}_a^2 + \mathbf{v}_b^2 - \mu^2}{2\|\mathbf{v}_a\|\|\mathbf{v}_b\|} \\ &\leq \frac{\mathbf{v}_a^2 + \mathbf{v}_b^2}{2\|\mathbf{v}_a\|\|\mathbf{v}_b\|} - \frac{\mu^2}{2\|\mathbf{v}_a\|\|\mathbf{v}_b\|}.\end{aligned}$$

Since $R < \|\mathbf{v}_a\|, \|\mathbf{v}_a\| \leq \gamma R = (1 + o(1))R$, we get

$$\cos(\phi_{\mathbf{v}_a,\mathbf{v}_b}) \leq 1 - \frac{\mu^2}{2R^2} + o(1). \tag{3.3}$$

Notice that this bound is very poor when R is large. So, for large R, we bound $\mathbf{v}_a \cdot \mathbf{v}_b$ differently. Without loss of generality, assume that $\mathbf{v}_b$ was added after $\mathbf{v}_a$. As a result the perturbed point $\mathbf{p}_b = \mathbf{v}_b + \mathbf{e}_b$ was reduced with $\mathbf{v}_a$, i.e., $\|\mathbf{p}_b - \mathbf{v}_a\| > \delta\|\mathbf{p}_b\|$. After squaring we get $\mathbf{p}_b \cdot \mathbf{v}_a < ((1 - \delta^2)\mathbf{p}_b^2 + \mathbf{v}_a^2)/2$ and therefore

$$\begin{aligned}\mathbf{v}_a \cdot \mathbf{v}_b &= \mathbf{v}_a \cdot \mathbf{p}_b - \mathbf{v}_a \cdot \mathbf{e}_b \\ &< ((1 - \delta^2)\mathbf{p}_b^2 + \mathbf{v}_a^2)/2 + \|\mathbf{v}_a\|\xi\mu.\end{aligned}$$

Using (3.2) gives

$$\cos(\phi_{\mathbf{v}_a,\mathbf{v}_b}) \leq \frac{(1 - \delta^2)\mathbf{p}_b^2 + \mathbf{v}_a^2}{2\|\mathbf{v}_a\|\|\mathbf{v}_b\|} + \frac{\|\mathbf{v}_a\|\xi\mu}{\|\mathbf{v}_a\|\|\mathbf{v}_b\|}.$$

Since $1 - \delta^2 = o(1)$, we get

$$\cos(\phi_{\mathbf{v}_a,\mathbf{v}_b}) \leq \frac{1}{2} + \frac{\xi\mu}{R} + o(1). \tag{3.4}$$

Combining the two bounds on $\cos(\phi_{\mathbf{v}_a,\mathbf{v}_b})$ we get

$$\cos(\phi_{\mathbf{v}_a,\mathbf{v}_b}) \leq \min\left\{1 - \frac{\mu^2}{2R^2}, \frac{1}{2} + \frac{\xi\mu}{R}\right\} + o(1). \tag{3.5}$$

As R increases, the first bound gets worse and the second better. So, the minimum (3.5) is maximized when

$$1 - \frac{\mu^2}{2R^2} = \frac{1}{2} + \frac{\xi\mu}{R}.$$

This is a quadratic equation in $x = \mu/R$, with only one positive solution

$$\frac{\mu}{R} = \sqrt{1 + \xi^2} - \xi = \frac{1}{\xi + \sqrt{1 + \xi^2}},$$

which, substituted in (3.5), gives the bound (3.1).

□

3.3.2 Time Complexity

In this subsection we prove bounds on the running time and success probability of our algorithm. We recall that the *List Sieve* samples random perturbations $\mathbf{e}_i$ from $\mathcal{B}_n(\xi\mu)$, sets $\mathbf{p}_i = \mathbf{e}_i \bmod \mathbf{B}$ and reduces $\mathbf{p}_i$ with the list L. Then it considers the lattice vector $\mathbf{v}_i = \mathbf{p}_i - \mathbf{e}_i$. At this point, one of the following (mutually exclusive) events occurs:

- Event **C:** $\mathbf{v}_i$ is a collision ($dist(L, \mathbf{v}_i) = 0$, i.e., $\mathbf{v}_i \in L$)
- Event **S:** $\mathbf{v}_i$ is a solution ($0 < dist(L, \mathbf{v}_i) \leq \mu$).
- Event **L:** $\mathbf{v}_i$ is a new list point ($dist(L, \mathbf{v}_i) > \mu$)

We will prove that if $\lambda_1(\mathbf{B}) \leq \mu \leq 1.01\lambda_1(\mathbf{B})$, event **S** will happen with high probability after a certain number of samples.

We first give a lower bound to the volume of a hypersphere cap (an alternative proof is given in [NV08]), which will be used to bound the probability of getting collisions.

Lemma 3.3.2. *Let* $\text{Cap}_{R,h}$ *be the n-dimensional spherical cap with height h of a hypersphere* $\mathcal{B}_n(R)$ *and set* $R_b = \sqrt{2hR - h^2}$. *Then we have that:*

$$\frac{\text{Vol}(\text{Cap}_{R,h})}{\text{Vol}(\mathcal{B}_n(R))} > \left(\frac{R_b}{R}\right)^n \cdot \frac{h}{2R_b n}.$$

Proof. The basis of $\text{Cap}_{R,h}$ is an $n-1$ dimensional hypersphere of radius $R_b = \sqrt{2hR - h^2}$. Therefore $\text{Cap}_{R,h}$ includes a cone C_1 with basis $\mathcal{B}_{n-1}(R_b)$ and height h. Also notice that a cylinder C_2 with basis $\mathcal{B}_{n-1}(R_b)$ and height $2 \cdot R_b$ includes $\mathcal{B}_n(R_b)$. Using the facts above we have:

$$\begin{aligned}\text{Vol}(\text{Cap}_{R,h}) > \text{Vol}(C_1) = \text{Vol}(\mathcal{B}_{n-1}(R_b))\frac{h}{n} &= \\ \text{Vol}(C_2)\frac{h}{2R_b n} > \text{Vol}(\mathcal{B}_n(R_b))\frac{h}{2R_b n}&.\end{aligned}$$

Therefore

$$\frac{\text{Vol}(\text{Cap}_{R,h})}{\text{Vol}(\mathcal{B}_n(R))} > \frac{\text{Vol}(\mathcal{B}_n(R_b))}{\text{Vol}(\mathcal{B}_n(R))} \cdot \frac{h}{2R_b n} = \left(\frac{R_b}{R}\right)^n \cdot \frac{h}{2R_b n}.$$

□

In the following theorem we assume $\xi < 0.7$ as this yields a slightly simpler proof, and the values of ξ that optimize the space and time complexity of our algorithm satisfy this constraint anyway. The theorem remains valid for larger values of ξ.

Theorem 3.3.3. *If* $\lambda_1(\mathbf{B}) \leq \mu \leq 1.01\lambda_1(\mathbf{B})$ *and* $0.5 < \xi < 0.7$ *then* List Sieve *outputs a lattice point with norm* $\leq \mu$ *with probability exponentially close to 1 as long as the number of samples used is at least* $K = \tilde{O}(2^{(c_1+c_2)n})$*, where*

$$c_1 = \log(\xi + \sqrt{\xi^2+1}) + 0.401,\ c_2 = \log\left(\frac{\xi}{\sqrt{\xi^2 - 0.25}}\right).$$

Proof. Let $\mathbf{s}$ be a shortest nonzero vector in $\mathcal{L}(\mathbf{B})$. Consider the intersection of two n-dimensional balls of radius $\xi\mu$ and centers $\mathbf{0}$ and $-\mathbf{s}$: $I_0 = \mathcal{B}_n(\mathbf{0}, \xi\mu) \cap \mathcal{B}_n(-\mathbf{s}, \xi\mu)$ and also $I_1 = \mathcal{B}_n(\mathbf{0}, \xi\mu) \cap \mathcal{B}_n(\mathbf{s}, \xi\mu) = I_0 + \mathbf{s}$. Notice that $\xi\mu \leq 0.707\lambda_1(\mathbf{B}) < \|\mathbf{s}\|$ and as a result I_0 and I_1 do not intersect.

Consider an error vector $\mathbf{e}_i$ in $I_0 \cup I_1$, and the corresponding perturbed point $\mathbf{p} = \mathbf{v} + \mathbf{e}_i$ generated by the list reduction procedure. The conditional distribution of $\mathbf{v}$ given $\mathbf{p}$, is uniform over the lattice points in $\mathbf{p} - I_0 \cup I_1$. We know that this set contains at least one lattice point $\mathbf{v} = \mathbf{p} - \mathbf{e}_i$. Moreover, if $\mathbf{v} \in \mathbf{p} - I_b$, then $\mathbf{v} - (-1)^b\mathbf{s} \in \mathbf{p} - I_{1-b}$. Finally, notice that the diameter of each $\mathbf{p} - I_b$ is bounded by

$$2\sqrt{(\xi\mu)^2 - \lambda_1^2/4} \leq 2\sqrt{0.707^2 - 0.25}\lambda_1 < \lambda_1.$$

It follows that $\mathbf{v}$ and $\mathbf{v}' = \mathbf{v} - (-1)^b\mathbf{s}$ are the only two lattice vectors in $\mathbf{p} - I_0 \cup I_1$, and at most one of them belongs of L because $\|\mathbf{v} - \mathbf{v}'\| = \lambda_1 \leq \mu$. This proves that, conditioned on $\mathbf{e}_i \in I_0 \cup I_1$, the probability that $\mathbf{v} \in L$ is at most 1/2:

$$Pr[\mathbf{C}|\mathbf{e}_i \in I_0 \cup I_1] \leq 1/2.$$

Now notice that $I \cup I'$ contains four disjoint caps with height $h = \xi\mu - \|\mathbf{s}\|/2 \geq \xi\mu - \mu/2$ on an n-dimensional ball of radius $\xi\mu$. We use Lemma 3.3.2 to bound from below the probability that $\mathbf{e}_i \in I_0 \cup I_1$:

$$\begin{aligned} Pr[\mathbf{e}_i \in I_0 \cup I_1] &= \frac{4\text{Vol}(\text{Cap}_{\xi\mu,\xi\mu-0.5\mu})}{\text{Vol}(\mathcal{B}_n(\xi\mu))} \\ &> \left(\frac{\sqrt{\xi^2-0.25}}{\xi}\right)^n \cdot \frac{\xi-0.5}{n\sqrt{\xi^2-0.25}} \\ &= 2^{-c_2 n} \cdot \frac{\xi-0.5}{n\sqrt{\xi^2-0.25}}. \end{aligned}$$

Therefore the probability of not getting a collision is bounded from below by

$$\begin{aligned} Pr[\mathbf{C'}] &\geq Pr[\mathbf{C'}|\mathbf{e}_i \in I \cup I']Pr[\mathbf{e}_i \in I \cup I'] \\ &\geq 2^{-c_2 n} \cdot \frac{\xi-0.5}{2n\sqrt{\xi^2-0.25}} = p. \end{aligned}$$

Now given that the probability of event **C'** is at least p, the number of occurrences of **C'** when K samples are processed is bounded from below by a random variable X following binomial distribution $Binomial(K,p)$. Let $F(N;K,p) = Pr[X \leq N]$ the probability of getting no more than N occurrences of **C'** after K samples. If we set $K = 2Np^{-1} = \tilde{O}(2^{(c_1+c_2)n})$ we can use Chernoff's inequality:

$$\begin{aligned} F(N;K,p) &\leq exp\left(-\frac{1}{2p}\frac{(Kp-N)^2}{K}\right) \\ &= exp\left(-\frac{N}{4}\right) \leq \frac{1}{2^{O(n)}}. \end{aligned}$$

As a result if List Sieve uses K samples, then, with exponentially high probability, at least $N+1$ of them will not satisfy event **C**. These events can either be **L** or **S** events. However the list size cannot grow beyond N, and as a result the number of **L** events can be at most N. So event **S** will happen at least once with high probability.

$\square$

Theorem 3.3.3 is used to set the variable K in the *List Sieve* algorithm, in order to ensure success probability exponentially close to 1. A bound on the running time of the algorithm immediately follows.

Corollary 3.3.4. *The total running time of the algorithm is* $2^{(2\cdot c_1+c_2)n}$ *where* c_1, c_2 *are as in theorem 3.3.3.*

Proof. Let us consider the running time of *List Reduce.* After every pass of the list L the input vector $\mathbf{p}_i$ to *List Reduce* gets shorter by a factor δ. Therefore the total running time is $\log_\delta(\|\mathbf{B}\|n) = poly(n)$ passes of the list and each pass costs $O(N)$ vector operations, each computable in polynomial time. Now notice that our algorithm will run *List Reduce* for K samples. This gives us total running time of $\tilde{O}(K \cdot N) = \tilde{O}(2^{(2c_1+c_2)n})$.

□

3.4 Practical performance of Gauss Sieve

In this section we describe some early experimental results on *Gauss Sieve.* We will describe the design of our experiments and then we will discuss the results on the space and time requirements of our algorithm.

Experiment setting: For our experiments we have generated square $n \times n$ bases corresponding to random knapsack problems modulo a prime of $\simeq 10 \cdot n$ bits. These bases are considered "hard" instances and are frequently used in the experimental evaluation of lattice algorithms [NV08, GN08a]. In our experiments we used *Gauss Sieve*, the *NV Sieve* implementation from [NV08] and the NTL library for standard enumeration techniques [SE94]. For every $n = 35, \ldots, 63$ we generated 6 random lattices, reduced them using BKZ with window 20, and measured the average space and time requirements of the algorithms. For sieving algorithms the logarithms of these measures grow almost linearly with the dimension n. We use a simple model of 2^{cn} to fit our results and we use least squares estimation to compute c. We run our experiments on a Q6600 Pentium, using only one core, and the algorithms were compiled with exactly the same parameters. The experiments are certainly not exhaustive, however we believe that they are enough to give a sense of the practical performance of our algorithm, at least in comparison to previous sieving techniques.

Termination of Gauss Sieve: As we have already discussed we cannot bound the number of samples required by *Gauss Sieve* to find a nonzero shortest vector with high probability. A natural termination condition is to stop after a certain number c of collisions. In order to get an estimate for a good value of c, we measured the number of collisions before a nonzero shortest vector is found. Although the number of collisions vary from lattice to lattice we have found that setting c to be let's say a $\simeq 10\%$ of the list size, is a conservative choice. The results that follow are based on this termination condition. *Gauss Sieve* found the nonzero shortest vector for 173 out of the 174 of the lattices we tested. Interestingly enough the failure was for dimension 38, and can be probably attributed to the very short list size used by the Gauss Sieve in small dimension.

Size complexity: To evaluate the size complexity of the *Gauss Sieve* we measure the maximum list size. (We recall that in the *Gauss Sieve* the list can both grow and shrink, as list points collide with each other. So, we consider the maximum list size during each run, and then average over the input lattice.) Our experiments show that the list size grows approximately as $2^{0.2n}$. This is consistent with our theoretical *worst-case* analysis, which bounds the list size by the kissing constant τ_n. Recall that τ_n can be reasonably conjectured to be near $2^{0.21n}$. The actual measured exponent 0.2 may depend on the input lattice distribution, and it would be interesting to run experiments on other distributions. However, in all cases, we expect the space complexity to be below $2^{0.21n}$. *Gauss Sieve* improves *NV Sieve* results in two ways:

- *Theory:* Our τ_n bound is proven under no heuristic assumptions, and gives an interesting connection between sieving techniques and spherical coding.
- *Practice:* In practice *Gauss Sieve* uses far fewer points (e.g., in dimension $n \simeq 40$, the list size is smaller approximately by a factor $\simeq 70$). See Figure 3.4.

Running time: Fitting the running time with our simple model 2^{cn} gives $c = 0.52$, which is similar to the experiments of *NV Sieve*. However once more *Gauss Sieve* is better in practice. For example, in dimension $\simeq 40$, the 70-fold improvement on the list size, gives a $250\times$ running time improvement. In Figure 3.4 we also give the running time of the Schnorr-Euchner (SE) enumeration algorithm as

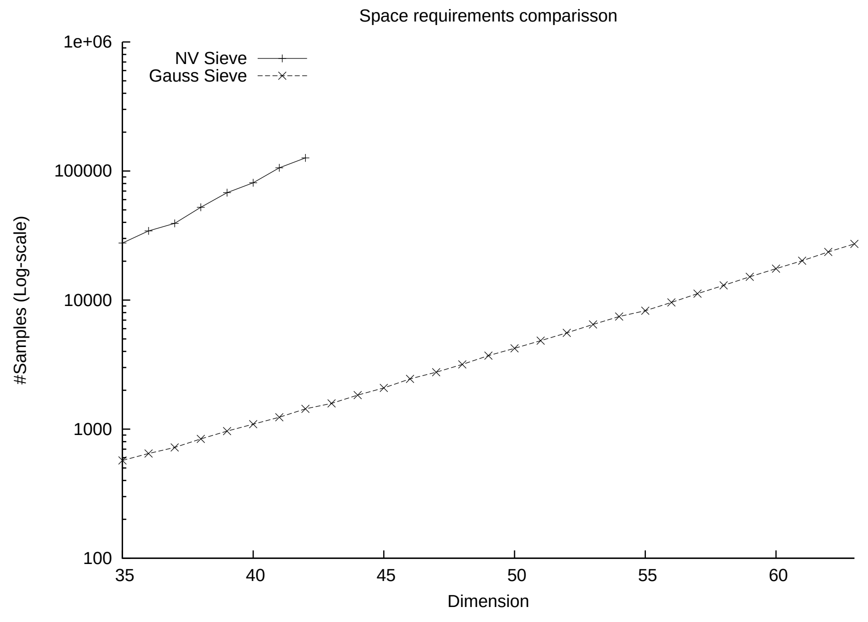

Figure 3.2: Space Requirements of Gauss Sieve

implemented in NTL.[4] This preliminary comparison with SE is meant primarily to put the comparison between sieve algorithms in perspective. In [NV08], Nguyen and Vidick had compared their variant of the sieve algorithm with the same implementation of SE used here, and on the same class of random lattices. Their conclusion was that while sieve algorithms have better asymptotics, the SE algorithm still reigned in practice, as the cross-over point is way beyond dimension $n \simeq 50$, and their algorithm was too expensive to be run in such high dimension. Including our sieve algorithm in the comparison, changes the picture quite a bit: the crossover point between the *Gauss Sieve* and the Schnorr-Euchner reference implementation used in [NV08] occurs already in dimension $n \simeq 40$.

This improved performance shows that sieve algorithms have the potential to be used in practice as an alternative to standard enumeration techniques. Although currently sieving algorithms cannot compare with more advanced enumeration heuristics (e.g., pruning), we hope that it is possible to develop similar

[4]The running time of enumeration algorithms, is greatly affected by the quality of the initial basis. To make a fair comparison we have reduced the basis using BKZ with window 20.

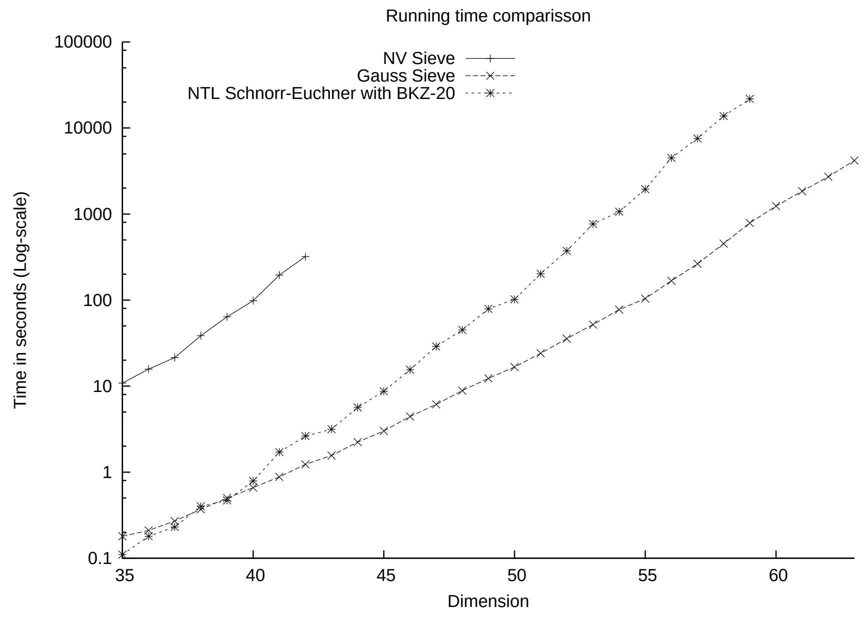

Figure 3.3: Running Time of Gauss Sieve

heuristics for sieve algorithms. The development of such heuristics, and a thorough study of how heuristics affect the comparison between enumeration and sieving, both in terms of running time and quality of the solution, is left to future research.

3.5 Extensions

An interesting feature common to all sieve algorithms is that they can be slightly optimized to take advantage of the structure of certain lattices used in practical cryptographic constructions, like the NTRU lattices [HPS98], or the cyclic lattices of [Mic07]. The idea is the following. The structured lattices used in this constructions have a non-trivial automorphism group, i.e., they are invariant under certain (linear) transformations. For example, cyclic lattices have the property that whenever a vector $\mathbf{v}$ is in the lattice, then all n cyclic rotations of $\mathbf{v}$ are in the lattice. When reducing a new point $\mathbf{p}$ against a list vector $\mathbf{v}$, we can use all rotations of $\mathbf{v}$ to decrease the length of $\mathbf{p}$. Effectively, this allows us to consider each list point as the implicit representation of n list points. This approximately translates to a reduction of the list size by a factor n. While this reduction may not be much from a theoretical point of view because the list size is exponential, it may have a noticeable impact on the practical performance of the algorithm.

Chapter 3 and Section A.1, in part, are a reprint of the paper "Large Faster exponential time algorithms for the shortest vector problem" co-authored with Daniele Micciancio and appearing in the proceedings of SODA 2010. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Voronoi Algorithms

In this chapter we give some common background for both the basic Voronoi algorithm and the optimized variant. In particular in Section 4.1 we give an overview of the algorithms and our contribution, while in Section 4.2 we define the Voronoi graph of a lattice and prove useful lemmas for both algorithms. The basic Voronoi algorithm is presented in Chapter 5, while the optimized variant in Chapter 6.

4.1 Overview

At the core of all our results is a new technique for the solution of the closest vector problem with preprocessing (CVPP). We recall that CVPP is a variant of CVP where some side information about the lattice is given as a hint together with the input. The hint may depend on the lattice, but not on the target vector. Typically, in the context of polynomial time algorithms, the hint is restricted to have polynomial size, but since here we study exponential time algorithms, one can reasonably consider hints that have size $2^{O(n)}$. The hint used by our algorithm is a description of the Voronoi cell of the lattice. We recall that the (open) Voronoi cell of a lattice Λ is the set $\mathcal{V}$ of all points (in Euclidean space) that are closer to the origin than to any other lattice point. The Voronoi cell $\mathcal{V}$ is a convex body, symmetric about the origin, and can be described as the intersection of half-spaces $H_{\mathbf{v}}$, where for any nonzero lattice point $\mathbf{v}$, $H_{\mathbf{v}}$ is the set of all points that are closer to the origin than to $\mathbf{v}$. It is not necessary to consider all $\mathbf{v} \in \Lambda \setminus \{\mathbf{0}\}$ when taking this intersection. One can restrict the intersection to the so-called *Voronoi relevant* vectors, which are the lattice vectors $\mathbf{v}$ such that $\mathbf{v}/2$ is the center of a facet of $\mathcal{V}$. Since the Voronoi cell of a lattice can be shown to have at most $2(2^n - 1)$ facets, $\mathcal{V}$ can be expressed as a finite intersection of at most $2(2^n - 1)$ half-spaces. Throughout this dissertation, we assume that the Voronoi cell of a lattice is always described by such a list of half-spaces.

The relation between the Voronoi cell and CVPP is well known, and easy to explain. In CVPP, we want to find the lattice point $\mathbf{v}$ closest to a given target vector $\mathbf{t}$. It is easy to see that this is equivalent to finding a lattice vector $\mathbf{v}$ such that $\mathbf{t} - \mathbf{v}$ belongs to the (closed) Voronoi cell of the lattice. In other words, CVP can be equivalently formulated as the problem of finding a point in the set $(\Lambda + \mathbf{t}) \cap \bar{\mathcal{V}}$, where $\bar{\mathcal{V}}$ is the topological closure of $\mathcal{V}$. The idea of using the Voronoi cell to solve CVP is not new. For example, a simple greedy algorithm for CVPP based on the knowledge of the Voronoi cell of the lattice is given in [SFS09]. The idea behind this algorithm (called the Iterative Slicer) is to make $\mathbf{t}$ shorter and shorter by subtracting Voronoi relevant vectors from it. Notice that if $\mathbf{t} \notin \bar{H}_{\mathbf{v}}$, then the length of $\mathbf{t}$ can be reduced by subtracting $\mathbf{v}$ from $\mathbf{t}$. So, as long as $\mathbf{t}$ is outside $\bar{\mathcal{V}}$, we can make progress and find a shorter vector. Unfortunately, this simple strategy to solve CVPP using the Voronoi cell is not known to perform any better than previous algorithms. The work [SFS09] only proves that the algorithm terminates after a finite number of iterations, and a close inspection of the proof reveals that the best upper bound that can be derived using the methods of [SFS09] is of the form $n^{O(n)}$: the running time of the Iterative Slicer is bound by a volume argument, counting the number of lattice points within a sphere of radius $\|\mathbf{t}\|$, and this can be well above $2^{O(n)}$ or even $n^{O(n)}$.

In the next two paragraphs we first sketch our new algorithm to solve CVPP using the Voronoi cell $\mathcal{V}$ in time $2^{O(n)}$, and then we show how to use the CVPP algorithm to recursively implement the preprocessing function and compute the Voronoi cell $\mathcal{V}$. Since both the preprocessing and CVPP computation take time $2^{O(n)}$, combining the two pieces gives an algorithm to solve CVP (and a host of other lattice problems, like SVP, SIVP, etc.) without preprocessing.

The CVPP algorithm As already remarked, the goal of CVPP with target vector $\mathbf{t}$ can be restated as the problem of finding a point $\mathbf{t}' \in (\Lambda + \mathbf{t}) \cap \mathcal{V}$ inside the Voronoi cell. (This point is also characterized as being a shortest vector in the set $\Lambda + \mathbf{t}$.) Therefore we need to find the shortest vector of the set $\Lambda + \mathbf{t}$. We follow a greedy approach similar to the Iterative Slicer of [SFS09]. Given the list of relevant vectors, the algorithm generates a sequence of shorter and shorter vectors from $\Lambda + \mathbf{t}$, until it finds the shortest vector of the set. However, in order to bound the number of iterations, we introduce two important modifications to the greedy strategy of [SFS09]. First we reduce the general CVPP to a special case where the target vector is guaranteed to belong to twice the Voronoi cell $2\bar{\mathcal{V}}$. This can be done very easily by a polynomial time Turing reduction. Next, using properties of the geometry of the Voronoi cell, we show that it is possible to generate a sequence of shorter and shorter vectors in $\Lambda + \mathbf{t}$, with the additional property that all the vectors are inside $2\bar{\mathcal{V}}$. This allows to bound the length of the sequence by 2^n. For each vector of the sequence the algorithm spends $\tilde{O}(2^n)$ time, which gives a total time complexity of $\tilde{O}(2^{2n})$.

Computing the Voronoi cell We have sketched how to solve CVPP, given the Voronoi cell of the lattice. This leaves us with the problem of computing the Voronoi cell, a task typically considered even harder than CVP. This is where the basic and optimized versions of the algorithm differ. The basic algorithm is more modular and uses already known techniques in an (almost) black-box way, while the optimized uses properties of the Voronoi cell to make the computation faster. Here we only give an overview of the modular but slower approach. In order to compute the Voronoi cell of a lattice Λ we use a method of [AEVZ02] that solves the problem by making 2^n calls to a CVPP oracle for the same lattice Λ. We combine this with a standard rank reduction procedure implicit in enumeration algorithms [Kan87b, HS07]. This procedure allows to solve CVPP in a lattice Λ of rank n making only $2^{O(n)}$ calls to a CVPP oracle for a properly chosen sub-lattice Λ' of rank $n-1$, which can be found in polynomial time. Combining all the pieces together we obtain an algorithm that computes the Voronoi cell of a lattice Λ by building a sequence of lattices $\Lambda_1 \subset \Lambda_2 \subset \cdots \subset \Lambda_n = \Lambda$ with rank $\text{rank}(\Lambda_i) = i$, and iteratively computing the Voronoi cell of Λ_{i+1} using the previously computed Voronoi cell of Λ_i. Since each $\mathcal{V}(\Lambda_i)$ can be computed from $\mathcal{V}(\Lambda_{i-1})$ is time $2^{O(n)}$ (by means of $2^{O(n)}$ CVPP computations, each of which takes $2^{O(n)}$ time), the total running time is $2^{O(n)}$.

4.1.1 Contribution

In this work we give a deterministic $\tilde{O}(2^{2n})$ time and $\tilde{O}(2^n)$ space algorithm for CVP, and therefore by the reductions in [GMSS99, Mic08], also to SVP, SIVP and several other lattice problems in NP considered in the literature. This improves the time complexity of the best previously known algorithm for CVP, SIVP, etc. [Kan87b] from $n^{O(n)}$ to $2^{O(n)}$. In the case of SVP, we achieve single exponential time as in [AKS01, NV08, MV10b, PS09], but without using randomization and decreasing the complexity from $\tilde{O}(2^{2.465n})$ [PS09] to $\tilde{O}(2^{2n})$. In the process, we also provide deterministic single exponential time algorithms for various other classic computational problems in lattices, like computing the kissing number, and computing the list of all Voronoi relevant vectors. Finally, in order to achieve these results we introduce the notion of the Voronoi graph of a lattice, which might be of independent interest.

4.2 The Voronoi Graph

In this section we define graphs where nodes are vectors from the coset $\Lambda + \mathbf{t}$. In particular we are considering the following families of graphs.

Definition 4.2.1. *Let Λ an n-dimensional lattice and $\mathbf{t}$ a vector on the span of Λ. The Voronoi Graph of the coset $\Lambda + \mathbf{t}$, denoted by $G_{\Lambda+\mathbf{t}}$ is an infinite graph with:*

- *Nodes $U = \{\mathbf{u}_i : \mathbf{u}_i \in \Lambda + \mathbf{t}\}$, all the vectors of the coset $\Lambda + \mathbf{t}$* [1]

[1] The set of nodes is denoted by U instead of the usual V. The letter V is used to denote the relevant vectors.

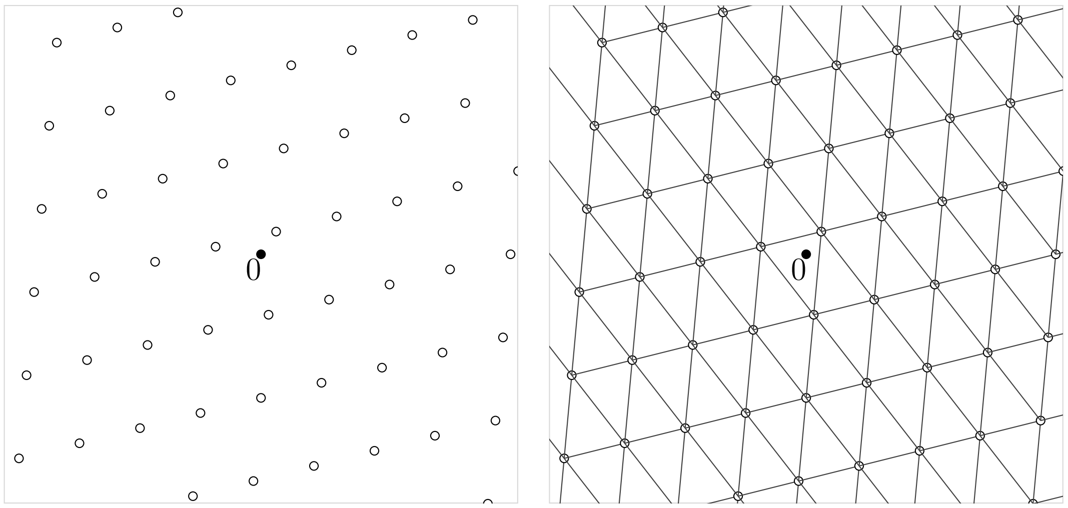

Figure 4.1: A coset $\Lambda + \mathbf{t}$ and the corresponding Voronoi graph

- *Edges between the nodes* $\mathbf{u}_i, \mathbf{u}_j$ *if and only if* $\mathbf{u}_i - \mathbf{u}_j$ *is a relevant vector.*

For each node $\mathbf{u}$ in the graph $G_{\Lambda+\mathbf{t}}$ we consider the following two quantities:

1. The *norm*, which is simply the euclidean norm of the vector $\|\mathbf{u}\|$.

2. The parity $\mathbf{p} \in \{0,1\}^n$. We say that a node $\mathbf{u}$ has parity $\mathbf{p}$ with respect to some basis $\mathbf{B}$, if and only if $\mathbf{u} \in 2\Lambda + \mathbf{B} \cdot \mathbf{p} + \mathbf{t}$.

The parities effectively partition $\Lambda + \mathbf{t}$ to 2^n cosets of the form $2\Lambda + \mathbf{B} \cdot \mathbf{p} + \mathbf{t}$. Notice that the partition acquired is independent of the choice of $\mathbf{B}$. The choice of basis simply assigns parity vectors $\mathbf{p} \in \{0,1\}^n$ to each partition.

We define the following finite subgraphs of $G_{\Lambda+\mathbf{t}}$:

1. The *core* subgraph, denoted by $G_{(\Lambda+\mathbf{t})\cap\bar{\mathcal{V}}}$ that is induced by the nodes in $(\Lambda + \mathbf{t}) \cap \bar{\mathcal{V}}$.

2. The open *relevant* subgraph, denoted by $G_{(\Lambda+\mathbf{t})\cap 2\mathcal{V}}$ that is induced by the nodes in $(\Lambda + \mathbf{t}) \cap 2\mathcal{V}$.

3. The closed *relevant* subgraph, denoted by $G_{(\Lambda+\mathbf{t})\cap 2\bar{\mathcal{V}}}$ that is induced by the nodes in $(\Lambda + \mathbf{t}) \cap 2\bar{\mathcal{V}}$.

We outline some simple facts about these graphs. By definition the core subgraph consists of the shortest vectors of $\Lambda + \mathbf{t}$. Notice that there is a straightforward correspondence between shortest vectors of $\Lambda + \mathbf{t}$ and closest vectors to $\mathbf{t}$. In particular if $\mathbf{s}$ is such a shortest vector then $\mathbf{t} - \mathbf{s}$ is a closest vector to $\mathbf{t}$. If the shortest vector is unique then the core subgraph consists of a single node in the interior of $\mathcal{V}$. If there are multiple shortest vectors then they reside in the boundary of $\bar{\mathcal{V}}$. If a node $\mathbf{u}$ is in $G_{(\Lambda+\mathbf{t})\cap 2\bar{\mathcal{V}}}$ then it is a shortest vector among all the vectors that share the same parity. Additionally if it is in $G_{(\Lambda+\mathbf{t})\cap 2\mathcal{V}}$ then it is a *unique* shortest vector for its parity. Recall that there are exactly 2^n parities, therefore the number of nodes in $G_{(\Lambda+\mathbf{t})\cap 2\mathcal{V}}$ is bounded by 2^n. (However the number of nodes in $G_{(\Lambda+\mathbf{t})\cap 2\bar{\mathcal{V}}}$ can be as big as 3^n, for example consider $\mathbb{Z}^n$.)

We show that all three subgraphs are connected. We start with the *core* subgraph.

Lemma 4.2.2. *For any lattice Λ and target vector $\mathbf{t}$ the core subgraph $G_{(\Lambda+\mathbf{t})\cap\bar{\mathcal{V}}}$ is connected.*

Proof. We show that any two nodes $\mathbf{u}_1, \mathbf{u}_2$ in $G_{(\Lambda+\mathbf{t})\cap\bar{\mathcal{V}}}$ are connected. By definition both $\mathbf{u}_1, \mathbf{u}_2$ are shortest vectors of $\Lambda + \mathbf{t}$ so $\|\mathbf{u}_1\| = \|\mathbf{u}_2\|$. Their difference $\mathbf{u}_2 - \mathbf{u}_1$ is in $\Lambda \cap 2\bar{\mathcal{V}}$ and by theorem 2.6.6 there exists a subset M of relevant vectors such that: $\mathbf{u}_2 = \mathbf{u}_1 + \sum_{\mathbf{v}_i \in M} \mathbf{v}_i$. Using the orthogonality of these relevant vectors:

$$\|\mathbf{u}_1\| = \|\mathbf{u}_2\| = \|\mathbf{u}_1 + \sum_{\mathbf{v}_i \in M} \mathbf{v}_i\| \Rightarrow \sum_{\mathbf{v}_i \in M} \|\mathbf{u}_1 + \mathbf{v}_i\|^2 = |M| \cdot \|\mathbf{u}_1\|^2$$

Notice that the vectors $\mathbf{u}_1 + \mathbf{v}_i$ are in $\Lambda + \mathbf{t}$ and cannot be shorter than $\mathbf{u}_1$ so: $\|\mathbf{u}_1 + \mathbf{v}_i\| \geq \|\mathbf{u}_1\|$. Combining the above we conclude that for every $\mathbf{v}_i$: $\|\mathbf{u}_1 + \mathbf{v}_i\| = \|\mathbf{u}_1\|$, so the vectors $\mathbf{u}_1 + \mathbf{v}_i$ are also shortest vectors of $\Lambda + \mathbf{t}$ and consequently nodes of $G_{(\Lambda+\mathbf{t})\cap\bar{\mathcal{V}}}$. Applying this argument iteratively we conclude that for any subset $M' \subseteq M$, the vectors $\mathbf{u}_1 + \sum_{\mathbf{v}_i \in M'} \mathbf{v}_i$ are also nodes of $G_{(\Lambda+\mathbf{t})\cap\bar{\mathcal{V}}}$. Finally, notice that the set of these vectors form a connected component of $G_{(\Lambda+\mathbf{t})\cap\bar{\mathcal{V}}}$. The vectors $\mathbf{u}_1, \mathbf{u}_2$ are in this component and therefore they are connected. $\square$

In order to show that the relevant subgraphs are connected we need the following technical lemma.

Lemma 4.2.3. *Let $\mathbf{u} \in \Lambda + \mathbf{t}$, such that it is exactly on the boundary of the scaled voronoi cell $k\bar{\mathcal{V}}$, with $k > 1$ and in particular on a facet corresponding to the scaled relevant vector $k\mathbf{v}$ ($\|\mathbf{u} - k\mathbf{v}\| = \|\mathbf{u}\|$). Then the vector $\mathbf{u}' = \mathbf{u} - \mathbf{v}$ has the following properties:*

- *It is in $(\Lambda + \mathbf{t}) \cap k\bar{\mathcal{V}}$,*
- *It is strictly shorter than $\mathbf{u}$ ($\|\mathbf{u}'\| < \|\mathbf{u}\|$).*

Proof. In order to prove the first property we use Observation 2.6.2, if $\mathbf{u}$ is in the facet of $k\bar{\mathcal{V}}$ corresponding to the relevant $\mathbf{v}$, then $\mathbf{u} - k\mathbf{v}$ is also in $k\bar{\mathcal{V}}$. Given that both $\mathbf{u}$ and $\mathbf{u} - k\mathbf{v}$ are in $k\bar{\mathcal{V}}$ and by convexity of the Voronoi cell we have that $\mathbf{u}' = \mathbf{u} - \mathbf{v}$ is also in $k\bar{\mathcal{V}}$. For the second property we have that: $\|\mathbf{u} - k\mathbf{v}\| = \|\mathbf{u}\| \Rightarrow 2\langle \mathbf{u}, \mathbf{v}\rangle = k\|\mathbf{v}\|^2$ and

$$\|\mathbf{u}'\|^2 = \|\mathbf{u} - \mathbf{v}\|^2 = \|\mathbf{u}\|^2 + \|\mathbf{v}\|^2 - 2\langle \mathbf{u}, \mathbf{v}\rangle = \|\mathbf{u}\|^2 - (k-1)\|\mathbf{v}\|^2 < \|\mathbf{u}\|^2.$$

□

Finally we show that both the open and closed relevant subgraphs are connected, by using the connectivity of the core subgraph.

Theorem 4.2.4. *There exist a path in $G_{(\Lambda+\mathbf{t})\cap 2\bar{\mathcal{V}}}$ ($G_{(\Lambda+\mathbf{t})\cap 2\mathcal{V}}$) from any node $\mathbf{u}$ of the graph to a node in $(\Lambda + \mathbf{t}) \cap \bar{\mathcal{V}}$. Additionally we can construct such a path where the nodes have strictly decreasing norm and the length of the path is bounded by 2^n.*

Proof. We construct such a path $\mathbf{u} = \mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_N$ by iteratively subtracting appropriate relevant vectors $\mathbf{u}_{i+1} = \mathbf{u}_i - \mathbf{v}_i$ chosen as shown in Lemma 4.2.3. Notice that as long as $\mathbf{u}_i$ is not in $\bar{\mathcal{V}}$, the new vector $\mathbf{u}_{i+1}$ is strictly shorter than $\mathbf{u}_i$ and is still in $2\bar{\mathcal{V}}$ (or $2\mathcal{V}$ if we started with a point in the open relevant subgraph) and consequently is a node of the graph. What remains to be shown is that the length of the path is at most 2^n nodes. Recall that every node in $2\bar{\mathcal{V}}$ is a shortest node for its parity. Given that there are only 2^n parities and that the norms of the vectors of the path are strictly decreasing we conclude that the length of the path is bounded by 2^n. □

Corollary 4.2.5. *For any lattice Λ and $\mathbf{t}$ on the span of Λ, the open and closed relevant subgraphs of $G_{\Lambda+\mathbf{t}}$ are connected.*

Chapter 4, Chapter 5 and Chapter 6, in part, are a reprint of the full version of the paper "A Deterministic Single Exponential Time Algorithm for Most Lattice Problems based on Voronoi Cell Computations" co-authored with Daniele Micciancio. An extended abstract of this paper was presented in STOC 2010. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Basic Voronoi Algorithm

5.1 Overview

In this chapter we describe and analyze a $\tilde{O}(2^{3.5n})$ time and $\tilde{O}(2^n)$ space algorithm for CVP and related lattice problems. This is improved to a $\tilde{O}(2^{2n})$ time and $\tilde{O}(2^n)$ space algorithm in Chapter 6. We describe the slower algorithm first as it allows for a more modular description and illustrates the connections with previous work. The CVP algorithm presented in this chapter has three components:

1. An $\tilde{O}(2^{2n})$-time algorithm to solve the *closest vector problem with preprocessing* (CVPP), where the output of the preprocessing function is the Voronoi cell of the input lattice, described as the intersection of half-spaces.

2. A *rank reduction* procedure that solves CVP on a k-rank lattice $\Lambda_k = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_k)$ with $2^{0.5k}$ calls to a CVP oracle for $\Lambda_{k-1} = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_{k-1})$.

3. A procedure for computing the *Voronoi cell* of a lattice Λ by 2^n calls to a CVP oracle for Λ and 2^{2n} polynomial time computations.

These three components are described and analyzed in Sections 5.2, 5.3, 5.4. In Section 5.5 we show how these components can be combined together into an algorithm for CVP with running time $\tilde{O}(2^{3.5n})$.

Notice that the rank reduction procedure immediately gives a recursive algorithm to solve CVP in arbitrary lattices. However, the obvious way to turn the rank reduction procedure into a recursive program leads to an algorithm with $2^{O(n^2)}$ running time, because each time the rank of the input lattice is reduced by 1, the number of recursive invocations gets multiplied by $2^{O(n)}$. We use the CVPP and Voronoi cell computation algorithms to give a more efficient transformation. The idea is to compute the Voronoi cells of all sub-lattices $\Lambda_k = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_k)$ sequentially, where $\mathbf{b}_1, \ldots, \mathbf{b}_n$ is the lattice basis produced by the rank reduction procedure. Each Voronoi cell $\mathcal{V}(\Lambda_k)$ is computed by (rank) reduction to $2^{O(k)}$ CVP computations in the lattice Λ_{k-1}. In turn, these CVP computations are performed using the CVPP algorithm and the previously computed Voronoi cell $\mathcal{V}(\Lambda_{k-1})$. Once $\mathcal{V}(\Lambda_n)$ has been computed, the input CVP instance can be solved using CVPP. We show that:

Theorem 5.1.1. *There is a deterministic $\tilde{O}(2^{3.5n})$-time algorithm to solve CVP.*

The proof of the theorem is postponed to Section 5.5, after we have described and analyzed the building blocks of the algorithm. The pseudocode of the algorithm is given in Algorithms 5.1, 5.2, 5.3, 5.4.

5.2 CVP with preprocessing

We give an algorithm that on input an n-rank lattice Λ, a list V of the Voronoi relevant vectors of Λ and a target vector $\mathbf{t}$, computes a lattice point closest to $\mathbf{t}$ in time $\tilde{O}(2^{2n})$. We proceed as follows:

1. We present CVPP$_{2\bar{\mathcal{V}}}$, a $\tilde{O}(2^{2n})$ algorithm for the special case where $\mathbf{t} \in 2\bar{\mathcal{V}}$,
2. then we solve the general CVPP problem with polynomially many calls to CVPP$_{2\bar{\mathcal{V}}}$.

For the following we assume without loss of generality that $\mathbf{t}$ belongs to the linear span of Λ, otherwise, we simply project $\mathbf{t}$ orthogonally to that subspace.

First we present the CVPP$_{2\bar{\mathcal{V}}}$ algorithm. Recall that in order to solve CVPP for $\mathbf{t}$ it is enough to find a vector $\mathbf{u} \in (\Lambda + \mathbf{t}) \cap \bar{\mathcal{V}}$ (then $\mathbf{t} - \mathbf{u}$ is a closest

Algorithm 5.1: Single exponential time for CVPP

```
function CVPP($\mathbf{t}, \mathbf{B}, V$)
    $\mathbf{t} \leftarrow$ Project $\mathbf{t}$ on the linear span of $\mathbf{B}$
    $\mathbf{t}_p \leftarrow \mathbf{t}$                    ▷ Choose smallest $p \in \mathbb{Z}$ that $\mathbf{t} \in 2^p\bar{\mathcal{V}}$
    for $i = p$ downto 1 do
        $\mathbf{t}_{i-1} \leftarrow \mathbf{t}_i - \mathrm{CVPP}_{2\bar{\mathcal{V}}}(\mathbf{t}_i, 2^{i-1}V)$
    return $\mathbf{t} - \mathbf{t}_0$

function CVPP$_{2\bar{\mathcal{V}}}$($\mathbf{t}, V$)
    $i \leftarrow 1$
    $\mathbf{u}_1 \leftarrow \mathbf{t}$
    while $\mathbf{u}_i \notin \bar{\mathcal{V}}$ do
        Find $\mathbf{v}_i \in V$ that maximizes $\langle \mathbf{u}_i, \mathbf{v} \rangle / \|\mathbf{v}\|^2$
        $\mathbf{u}_{i+1} \leftarrow \mathbf{u}_i - \mathbf{v}_i$
        $i \leftarrow i + 1$
    return $\mathbf{t}_0 - \mathbf{u}_i$
```

vector to $\mathbf{t}$). Consider the graph $G_{(\Lambda+\mathbf{t})\cap 2\bar{\mathcal{V}}}$, by assumption $\mathbf{t}$ is in $2\bar{\mathcal{V}}$ and therefore a node in the graph. By Theorem 4.2.4 we can construct a path in $G_{(\Lambda+\mathbf{t})\cap 2\bar{\mathcal{V}}}$ from $\mathbf{t} = \mathbf{u}_1$ to a node $\mathbf{u}_N \in (\Lambda + \mathbf{t}) \cap \bar{\mathcal{V}}$. The algorithm computes such a path $\mathbf{t} = \mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_N$ as follows:

- As long as $\mathbf{u}_i$ is not in $\bar{\mathcal{V}}$, it finds the relevant vector $\mathbf{v}_i$ that maximizes the quantity $2\langle \mathbf{u}_i, \mathbf{v} \rangle / \|\mathbf{v}\|^2$.
- It sets $\mathbf{u}_{i+1} = \mathbf{u}_i - \mathbf{v}_i$.

When it finds a $\mathbf{u}_N \in \bar{\mathcal{V}}$ it returns $\mathbf{t} - \mathbf{u}_N$. We show that the relevant vector subtracted from each node $\mathbf{u}_i$ is exactly the relevant vector chosen in Lemma 4.2.3 and therefore the path is identical with the path of Theorem 4.2.4.

Lemma 5.2.1. *Given a list V of the relevant vectors of lattice Λ and a target point $\mathbf{t} \in 2\bar{\mathcal{V}}$ the* CVPP$_{2\bar{\mathcal{V}}}$ *algorithm (Algorithm 5.1) finds a closest vector to $\mathbf{t}$ in time $\tilde{O}(2^{2n})$, where n is the rank of the lattice.*

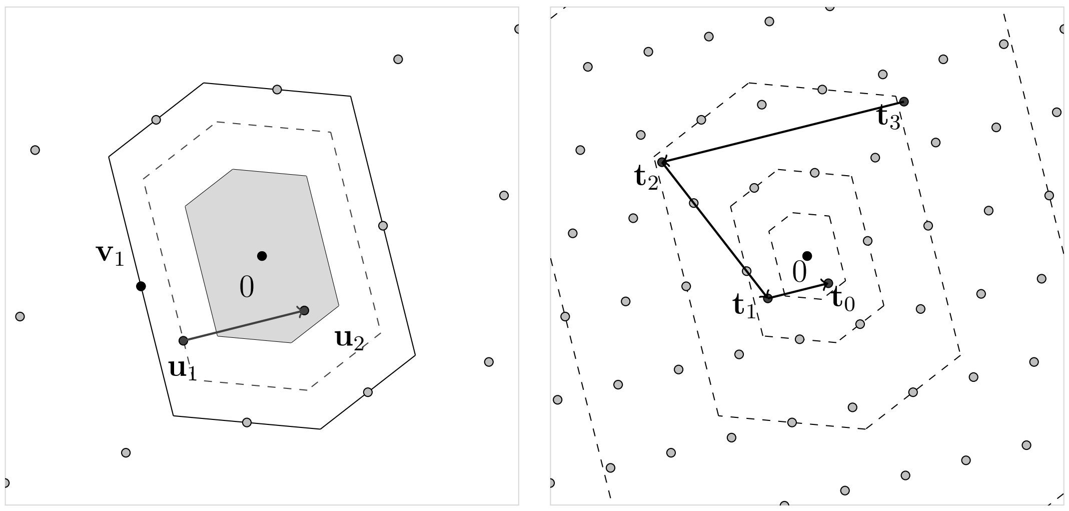

Figure 5.1: One step of CVPP$_{2\bar{\mathcal{V}}}$ (left), Succesive steps of CVPP(right)

Proof. Let $\mathbf{v}_i$ be the relevant vector that maximizes the quantity $2\langle \mathbf{u}_i, \mathbf{v}\rangle/\|\mathbf{v}\|^2$, let k be this maximum value. Then for every relevant vector $\mathbf{v}$, $\|\mathbf{u}_i - k\mathbf{v}\| \geq \|\mathbf{u}_i\|$ and for $\mathbf{v}_i$, $\|\mathbf{u}_i - k\mathbf{v}_i\| = \|\mathbf{u}_i\|$. Equivalently $\mathbf{u}_i$ is in $k\bar{\mathcal{V}}$ and exactly in the facet corresponding to the (scaled) relevant vector $k\mathbf{v}_i$. Therefore the chosen relevant vectors are identical with Lemma 4.2.3 and by Theorem 4.2.4 the path $\mathbf{u}_1, \ldots, \mathbf{u}_N$ is in $G_{(\Lambda+\mathbf{t})\cap 2\bar{\mathcal{V}}}$ and after at most 2^n nodes the algorithm should find a node $\mathbf{u}_N \in \bar{\mathcal{V}}$.

Concluding, the CVPP$_{2\bar{\mathcal{V}}}$ algorithm finds the required vector $\mathbf{u}_N$, and solves CVPP, after at most 2^n iterations. Recall that each iteration examines all the relevant vectors in order to find an appropriate $\mathbf{v}_i$, therefore the complexity of each such iteration is poly$(n)|V| \leq \tilde{O}(2^n)$, which gives a total complexity of $\tilde{O}(2^{2n})$. $\square$

The next theorem shows how to solve CVP for any target vector.

Theorem 5.2.2. *On input a list V of the relevant vectors of a lattice Λ and a target point $\mathbf{t}$, the* CVPP *algorithm (Algorithm 5.1) finds a closest vector to $\mathbf{t}$ in* $\log\|\mathbf{t}\| \cdot \tilde{O}(2^{2n})$ *time, where n is the rank of the input lattice.*

Proof. The CVPP algorithm computes a factor α such that $\mathbf{t}$ is exactly on the boundary of $\alpha\bar{\mathcal{V}}$. Recall that α can be computed as $2\max_{\mathbf{v}\in V}(\langle \mathbf{t}, \mathbf{v}\rangle/\|\mathbf{v}\|^2)$, so $\alpha < 2\|\mathbf{t}\|$. Then it computes the smallest integer p such that $2^p > \alpha$, so that $\mathbf{t} \in 2^p\bar{\mathcal{V}}$.

The algorithm sets $\mathbf{t}_p = \mathbf{t}$ and computes a sequence of vectors $\mathbf{t}_p, \mathbf{t}_{p-1}, \ldots, \mathbf{t}_0$ in $\Lambda + \mathbf{t}$ such that each $\mathbf{t}_i$ is in $2^i\bar{\mathcal{V}}$. The algorithm obtains $\mathbf{t}_i$ from $\mathbf{t}_{i+1}$ with a call to $\mathrm{CVPP}_{2\bar{\mathcal{V}}}$. Notice that p is bounded by $\log \|\mathbf{t}\| + 2$ and each $\mathrm{CVPP}_{2\bar{\mathcal{V}}}$ computation costs $\tilde{O}(2^{2n})$ by Lemma 5.2.1. Therefore the total complexity is $\log \|\mathbf{t}\| \cdot \tilde{O}(2^{2n})$. The vector $\mathbf{t}_0$ is in $\bar{\mathcal{V}}$ and consequently a shortest vector in $\Lambda + \mathbf{t}$. As a result the algorithm can simply output $\mathbf{t} - \mathbf{t}_0$ which is a closest vector to $\mathbf{t}$.

Finally, we show how to obtain $\mathbf{t}_i$ from $\mathbf{t}_{i+1}$ with a call to $\mathrm{CVPP}_{2\bar{\mathcal{V}}}$. The algorithm scales the lattice Λ and the corresponding relevant vectors V by 2^i and runs $\mathrm{CVPP}_{2\bar{\mathcal{V}}}$ on $2^i\Lambda$, with target vector $\mathbf{t}_{i+1}$. Notice that $\mathbf{t}_{i+1}$ is in $2 \cdot 2^i\mathcal{V}$, therefore it is a valid input for $\mathrm{CVPP}_{2\bar{\mathcal{V}}}$ on the lattice $2^i\Lambda$. Finally, given the closest vector $\mathbf{c}_{i+1}$ the algorithm simply sets $\mathbf{t}_i = \mathbf{t}_{i+1} - \mathbf{c}_{i+1}$. The new vector $\mathbf{t}_i$ is in $\Lambda + \mathbf{t}$ and also in $2^i\mathcal{V}$ by Observation 2.6.1. □

5.3 Rank reduction

Algorithm 5.2: Rank reduction procedure for CVP

```
function RankReduceCVP($\mathbf{t}, \mathbf{B}_k, V_{k-1}, H$)
    $h_t \leftarrow \langle \mathbf{t}, \mathbf{b}_k^* \rangle / \langle \mathbf{b}_k^*, \mathbf{b}_k^* \rangle$
    $\mathbf{c} \leftarrow \mathbf{0}$
    for all $h$ that: $|h - h_t| < H$ do
        $\mathbf{c}_h \leftarrow \mathrm{CVPP}(\mathbf{t} - h\mathbf{b}_k, V_{k-1}) + h\mathbf{b}_k$
        if $\|\mathbf{c}_h - \mathbf{t}\| < \|\mathbf{c} - \mathbf{t}\|$ then
            $\mathbf{c} \leftarrow \mathbf{c}_h$
    return $\mathbf{c}$
```

We present a procedure that on input a basis $\mathbf{B}_k = \{\mathbf{b}_1, \ldots, \mathbf{b}_k\}$ for a k-rank lattice Λ_k and an integer H such that $\mu(\Lambda_k)/\|\mathbf{b}_k^*\| \leq H$, solves a CVP instance on Λ_k, with $2H+1$ calls to CVP on the $(k-1)$-rank sub-lattice $\Lambda_{k-1} = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_{k-1})$ (where $\mu(\Lambda_k)$ is the covering radius of Λ_k and $\mathbf{b}_k^*$ the component of $\mathbf{b}_k$ orthogonal to Λ_{k-1}). The efficiency of this transformation depends on the bound H of the ratio $\mu(\Lambda_k)/\|\mathbf{b}_k^*\|$ and therefore on the quality of the input basis. To address this issue the rank reduction consists of two steps:

1. A polynomial time preprocessing step that outputs a basis $\mathbf{B}$ for the input lattice such that the ratio $\mu(\Lambda_k)/\|\mathbf{b}_k^*\|$ is bounded by $2^{0.5k}$ for every $k = 2, \ldots, n$. We show that this can be achieved by simply running the LLL algorithm[LLL82]. This step is done only once at the beginning of the algorithm and all further computations use the higher quality basis.

2. The actual rank reduction procedure that given an integer H such that $\mu(\Lambda_k)/\|\mathbf{b}_k^*\| \le H$ solves a CVP instance on Λ_k with $2H+1$ calls to CVP on Λ_{k-1}.

Combining the two steps gives a reduction from a CVP in Λ_k to $2^{0.5k}$ CVPs in Λ_{k-1}. The two procedures are analyzed in the following lemmas.

Lemma 5.3.1. *The LLL algorithm, on input a basis for an n-rank lattice Λ, outputs a new basis $\mathbf{B}$ for the same lattice with the following property: for every sub lattice $\Lambda_k = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_k)$, $k \le n$, we have $\mu(\Lambda_k)/\|\mathbf{b}_k^*\| \le 2^{0.5k}$.*

Proof. Notice that:

- If $\mathbf{B}$ is a basis for Λ, then $\mu(\Lambda) \le \rho = \frac{1}{2}\sqrt{\sum_{i=1}^n \|\mathbf{b}_i^*\|^2}$, as the set $\{\sum \mathbf{x}_i \mathbf{b}_i^* : -\frac{1}{2} < x_i \le \frac{1}{2}\}$ is a fundamental region of Λ and it is contained in a ball with radius ρ centered at $\mathbf{0}$.

- If $\mathbf{B}$ is LLL reduced then $\|\mathbf{b}_{i+1}^*\|^2 \ge 2^{-1}\|\mathbf{b}_i^*\|^2$ and consequently $\|\mathbf{b}_k^*\|^2 \ge 2^{i-k}\|\mathbf{b}_i^*\|^2$. Thus:
$$\rho = \frac{1}{2}\sqrt{\sum_{i=1}^n \|\mathbf{b}_i^*\|^2} \le \frac{1}{2}\sqrt{\sum_{i=1}^n 2^{n-i}\|\mathbf{b}_n^*\|^2} = \frac{1}{2}\sqrt{2^n - 1}\|\mathbf{b}_n^*\| \le 2^{0.5n}\|\mathbf{b}_n^*\|.$$

From the above inequalities we conclude that if $\mathbf{B}$ is an LLL reduced basis for Λ then $\mu(\Lambda)/\|\mathbf{b}_n^*\| \le 2^{0.5n}$. Finally, notice that if $\mathbf{B}$ is LLL reduced then, for all $k \le n$, the basis $\mathbf{B}_k = \{\mathbf{b}_1, \ldots, \mathbf{b}_k\}$ is also LLL reduced and consequently $\mu(\Lambda_k)/\|\mathbf{b}_k^*\| \le 2^{0.5k}$. □

Lemma 5.3.2. *On input a basis $\mathbf{B}_k = \{\mathbf{b}_1, \ldots, \mathbf{b}_k\}$ for a k-rank lattice Λ_k and an integer H such that $\mu(\Lambda_k)/\|\mathbf{b}_k^*\| \le H$, the algorithm* RankReduceCVP *(Algorithm 5.2) solves a* CVP *instance on Λ_k with $2H+1$ calls to* CVP *on the $(k-1)$-rank sub-lattice $\Lambda_{k-1} = \mathcal{L}\{\mathbf{b}_1, \ldots, \mathbf{b}_{k-1}\}$.*

Proof. Let $\mathbf{t}$ be the target vector of CVP on Λ_k. We can assume without loss of generality that $\mathbf{t}$ belongs to the linear span of Λ_k, otherwise, we simply project $\mathbf{t}$ orthogonally to that subspace. Partition the lattice Λ_k into layers of the form $h\mathbf{b}_k + \Lambda_{k-1}$, where $h \in \mathbb{Z}$. Notice that:

- by the definition of the covering radius, the closest vector to $\mathbf{t}$ has distance at most $\mu(\Lambda_k)$.
- the distance of all lattice points in the layer $h\mathbf{b}_k + \Lambda_{k-1}$ from $\mathbf{t}$ is at least $|h - h_t| \cdot \|\mathbf{b}_k^*\|$, where $h_t = \langle \mathbf{t}, \mathbf{b}_k^* \rangle / \langle \mathbf{b}_k^*, \mathbf{b}_k^* \rangle$, because this is the distance between $\mathbf{t}$ and the entire affine space generated by the layer.

It follows from the above observations that the lattice points in Λ_k closest to $\mathbf{t}$ belong to layers $h\mathbf{b}_k + \Lambda_{k-1}$ such that $|h - h_t| \leq \mu(\Lambda_k)/\|\mathbf{b}_k^*\|$. Recall that the algorithm is given an H such that $\mu(\Lambda_k)/\|\mathbf{b}_k^*\| \leq H$. Therefore, in order to find a lattice point closest to $\mathbf{t}$ the algorithm iterates over all $2H+1$ layers with h such that $|h - h_t| \leq H$, and for each of them finds a point in $h\mathbf{b}_k + \Lambda_{k-1}$ closest to $\mathbf{t}$. Notice that this is equivalent to finding a point in Λ_{k-1} closest to $\mathbf{t} - h\mathbf{b}_k$, i.e., a CVP computation in Λ_{k-1}. A lattice point closest to $\mathbf{t}$ is found selecting the best solution across all layers. □

5.4 Computing the Voronoi cell

This component computes the relevant vectors of a lattice Λ by $2^n - 1$ calls to CVP, all for the same lattice, plus 2^{2n} polynomial time computations. In order to achieve this we use a variant of the RelevantVectors algorithm of [AEVZ02] [1]. Our variant computes the relevant vectors in two steps:

1. First it runs the FindRelevant procedure that, given a lattice Λ, outputs a list V of size $2(2^n - 1)$ that contains all the relevant vectors, with the help of $2^n - 1$ CVP queries. Notice that the list might contain non relevant vectors, but all vectors in the list are lattice points.

[1] The original algorithm as described in [AEVZ02] does not require the additional step of RemoveNonRelevant, but it requires a CVP oracle that finds *all* closest points. We describe this variant in order to avoid giving one more algorithm for solving the AllCVP problem.

Algorithm 5.3: Voronoi cell computation

```
function FindRelevant($\mathbf{B}_k, V_{k-1}, H$)
    for $\mathbf{p}$ in $\{\{0,1\}^k \setminus \mathbf{0}\}$ do
        $\mathbf{t} \leftarrow -\mathbf{B}_k \cdot \mathbf{p}/2$
        $\mathbf{c} \leftarrow$ RankReduceCVP($\mathbf{t}, \mathbf{B}_k, V_{k-1}, H$)
        $V_k \leftarrow V_k \cup \{\pm 2(\mathbf{c} - \mathbf{t})\}$
    return $V_k$
function RemoveNonRelevant($V_k$)
    for $\mathbf{v}_i \in V_k$ do
        for $\mathbf{v}_j \in V_k : \mathbf{v}_j \neq \mathbf{v}_i$ do
            if $\|\mathbf{v}_j - \mathbf{v}_i/2\| \leq \|\mathbf{v}_i/2\|$ then
                $V_k \leftarrow V_k \setminus \{\mathbf{v}_i\}$
    return $V_k$
```

2. Then it feeds the output list V to the RemoveNonRelevant procedure. RemoveNonRelevant discards the non-relevant points from a list of *lattice points* V that contains all the relevant vectors in time poly$(n)|V|^2$. Notice that $|V| < 2^{n+1}$, therefore this step takes $\tilde{O}(2^{2n})$ time.

The pseudocode is given in Algorithm 5.3. We go on to describe and analyze the two procedures.

Lemma 5.4.1. *Given a basis* $\mathbf{B}$ *for an* n*-rank lattice, the* FindRelevant *procedure (Algorithm 5.3) computes a set of at most* 2^{n+1} *lattice vectors that contains all the relevant vectors of* $\Lambda = \mathcal{L}(\mathbf{B})$ *using* $2^n - 1$ CVP *queries for the same lattice.*

Proof. On input a basis $\mathbf{B}$ for a n-rank lattice Λ, the FindRelevant procedure iterates over all $\mathbf{p} \in \{0,1\}^n \setminus \{\mathbf{0}\}$, and for each of them, does the following:

1. Find a closest vector $\mathbf{c}$ of Λ to $\mathbf{t} = -\mathbf{B} \cdot \mathbf{p}/2$.
2. Include $\pm \mathbf{s}_{\mathbf{p}} = \pm 2(\mathbf{c} - \mathbf{t})$ to the list V.

It is clear that the above procedure generates a list of $2(2^n - 1)$ vectors using $2^n - 1$ calls to a CVP oracle. Now in order to show that it contains all the relevant vectors, notice that if $\mathbf{c} \in \Lambda$ is a vector closest to $\mathbf{t} = -\mathbf{B}\mathbf{p}/2$, then by Observation 2.6.1, $2(\mathbf{c} - \mathbf{t})$ is a shortest vector of the coset $C_{\mathbf{B},\mathbf{p}} = 2\Lambda + \mathbf{B} \cdot \mathbf{p}$. Therefore the list V contains a pair of shortest vectors for all the cosets $C_{\mathbf{B},\mathbf{p}}$, with $\mathbf{p} \in \{0,1\}^n \setminus \{\mathbf{0}\}$, and by Corollary 2.6.5 all the relevant vectors. □

Lemma 5.4.2. *Given a superset of the relevant vectors V, the* RemoveNonRelevant *procedure (Algorithm 5.3) outputs a set that contains only the relevant vectors in time poly$(n)|V|^2$.*

Proof. Given a list of vectors $V = \{\mathbf{v}_1, \dots, \mathbf{v}_k\}$, RemoveNonRelevant iterates over the vectors $\mathbf{v}_i$ and if there exists $\mathbf{v}_j \neq \mathbf{v}_i \in V$ such that $\|\mathbf{v}_j - \mathbf{v}_i/2\| \leq \|\mathbf{v}_i/2\|$, removes $\mathbf{v}_i$ from the list. Finally it outputs the list V without the discarded vectors. It is clear that the algorithm runs in time poly(n)$|V|^2$. For correctness we need the following two observations:

- If $\mathbf{v}_i$ is a relevant vector, for every $\mathbf{u} \neq \mathbf{v}_i \in \Lambda$, $\|\mathbf{u} - \mathbf{v}_i/2\| > \|\mathbf{v}_i/2\|$. For the proof, notice that if $\mathbf{v}_i$ is a relevant vector, $\pm\mathbf{v}_i$ should be unique shortest vectors of $2\Lambda + \mathbf{v}_i$. The vector $2\mathbf{u} - \mathbf{v}_i$ is in $2\Lambda + \mathbf{v}_i$ therefore $\|2\mathbf{u} - \mathbf{v}_i\| > \|\mathbf{v}_i\|$ which gives the required inequality.

- If $\mathbf{v}_i$ is not relevant, there exists a relevant vector $\mathbf{r}$ such that $\|\mathbf{r} - \mathbf{v}_i/2\| \leq \|\mathbf{v}_i/2\|$. We prove the statement by contradiction. Assume that $\mathbf{v}_i$ is not relevant and that for all relevant vectors $\mathbf{r}$, $\|\mathbf{r} - \mathbf{v}_i/2\| > \|\mathbf{v}_i/2\|$. Equivalently the vector $\mathbf{v}_i/2$ is in the interior of $\mathcal{V}$ and $\mathbf{0}$ is its *unique* closest vector. This is a contradiction because $\mathbf{0}$ and $\mathbf{v}$ have the same distance from $\mathbf{v}/2$.

Now, by the first observation none of the relevant vectors is going to be removed from the list. Therefore the second observation gives that all the non-relevant vectors are going to be discarded (because all the relevant vectors remain in the list). □

5.5 Combining the blocks

In this section we give a $\tilde{O}(2^{3.5n})$ algorithm for computing the Voronoi cell by combining the three blocks of the previous sections. First we describe RankRe-

Algorithm 5.4: The Voronoi Cell Algorithm

```
function BasicVoronoiCell($\mathbf{B}$)
    $\mathbf{B} \leftarrow$ Preprocess($\mathbf{B}$)
    $V_1 = \{(\mathbf{b}_1, -\mathbf{b}_1)\}$
    for $k = 2$ to $n$ do
        $\mathbf{B}_k \leftarrow [\mathbf{b}_1, \ldots, \mathbf{b}_k]$
        $V_k \leftarrow$ RankReduceVCell($\mathbf{B}_k, V_{k-1}, 2^{0.5k}$)
    return $V_n$
function Preprocess($\mathbf{B}$)
    return LLL($\mathbf{B}$)
function RankReduceVCell($\mathbf{B}_k, V_{k-1}, H$)
    $V_k \leftarrow$ FindRelevant($\mathbf{B}_k, V_{k-1}, H$)
    $V_k \leftarrow$ RemoveNonRelevant($V_k$)
    return $V_k$
```

duceVCell, an algorithm that computes the Voronoi cell of $\Lambda_n = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_n)$ with the the Voronoi cell of Λ_{n-1}.

Lemma 5.5.1. *Given a basis $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ for an n-rank lattice Λ_n, an integer H such that $\mu(\Lambda_n)/\|\mathbf{b}_n^*\| \leq H$ and the Voronoi cell of $\Lambda_{n-1} = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_{n-1})$ the* RankReduceVCell *procedure (Algorithm 5.4) deterministically computes the Voronoi cell of Λ_n in time $H \cdot \tilde{O}(2^{3n})$.*

Proof. In order to compute the Voronoi cell of Λ_n it is enough to call the procedure FindRelevant followed by RemoveNonRelevant (see Section 5.4). However, notice that FindRelevant requires access to a CVP oracle for Λ_n. RankReduceVCell instantiates a CVP oracle for Λ_n as follows. For each query to the CVP oracle it calls RankReduceCVP to reduce the given CVP instance on Λ_n to $2H + 1$ CVP instances on Λ_{n-1} (Lemma 5.3.2). Then it solves every such instance with a call to the CVPP algorithm for Λ_{n-1} (notice that the Voronoi cell of Λ_{n-1} is given as input). Consequently we have the following derivations:

1. FindRelevant computes a superset of the relevant vectors of Λ_n with $2^n - 1$ CVP computations in Λ_n (Lemma 5.4.1).

2. RankReduceCVP reduces each such CVP computation to at most $2H+1$ (Lemma 5.3.2) CVP computations in Λ_{n-1} (Lemma 5.3.2).

3. CVPP solves each CVP computation in Λ_{n-1} with the help of the precomputed Voronoi cell in time $\tilde{O}(2^{2n})$ (Lemma 5.2.2).

4. RemoveNonRelevant discards vectors that are non relevant from the list produced by FindRelevant in time $\tilde{O}(2^{2n})$ (Lemma 5.4.2).

We conclude that computing the Voronoi cell of Λ_n is reduced to $2H+1$ CVP computations in Λ_{n-1} each solved in $\tilde{O}(2^{2n})$ time by CVPP plus 2^{2n} polynomial time computation by RemoveNonRelevant. Therefore the total time complexity for computing V_{n-1} from V_n is $H \cdot \tilde{O}(2^{3n})$. □

The following theorem describes the main algorithm of this chapter. An algorithm that computes the Voronoi cell of a given lattice with the help of two subroutines. A preprocessing subroutine (LLL) and a subroutine for computing the Voronoi cell of Λ_{k+1} using the Voronoi cell of Λ_k (RankReduceVCell). The pseudocode is given as Algorithm 5.4.

Theorem 5.5.2. *There is a deterministic $\tilde{O}(2^{3.5n})$-time and $\tilde{O}(2^n)$ space algorithm that on input a lattice $\mathbf{B}$, outputs the relevant vectors of $\Lambda_n = \mathcal{L}(\mathbf{B})$.*

Proof. The BasicVoronoiCell algorithm (Algorithm 5.4) combines the two subroutines as follows. First it runs LLL on the input basis, and acquires a new basis $\mathbf{B}$ for the same lattice where for all $k = 2, \ldots, n$ we have $\mu(\Lambda_k)/\|\mathbf{b}_k^*\| \leq 2^{0.5k}$, by Lemma 5.3.1. The rest of the algorithm works on the new basis $\mathbf{B}$. The algorithm sets the Voronoi cell of the one dimensional lattice $\Lambda_1 = \mathcal{L}(\mathbf{b}_1)$ to $V_1 = \{\mathbf{b}_1, -\mathbf{b}_1\}$. Then for $k = 2, \ldots, n$ it uses the RankReduceVCell algorithm with $H = 2^{0.5k}$, in order to compute the Voronoi cell of Λ_{k+1} using the Voronoi cell of Λ_k. Each such call has a cost of $\tilde{O}(2^{3.5k})$, therefore the total time complexity of the algorithm is $\sum_{k=2,\ldots,n}(\tilde{O}(2^{3.5k})) = \tilde{O}(2^{3.5n})$. Finally the space requirements are at most $\tilde{O}(2^n)$, as none of the called procedures requires more space. □

Combining the BasicVoronoiCell and CVPP algorithms gives a $\tilde{O}(2^n)$ space and $\tilde{O}(2^{3.5n})$ time algorithm for CVP and a proof for the main theorem of this chapter (Theorem 5.1.1).

Proof. In order to solve CVP for target $\mathbf{t}$ on a lattice Λ, the algorithm first computes the Voronoi relevant vectors V of Λ with BasicVoronoiCell (Theorem 5.5.2) in $\tilde{O}(2^{3.5n})$ time and $\tilde{O}(2^n)$ space. Then it solves the CVP instance using the $\tilde{O}(2^{2n})$ CVPP algorithm and the precomputed relevant vectors of Λ (Theorem 5.2.2). $\square$

Chapter 4, Chapter 5 and Chapter 6, in part, are a reprint of the full version of the paper "A Deterministic Single Exponential Time Algorithm for Most Lattice Problems based on Voronoi Cell Computations" co-authored with Daniele Micciancio. An extended abstract of this paper was presented in STOC 2010. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Optimized Voronoi Algorithm

6.1 Overview

In this chapter we present the ImprovedVoronoiCell algorithm, a deterministic $\tilde{O}(2^{2n})$ algorithm for computing the Voronoi cell and consequently solving CVP, SVP, and other hard lattice problems. The algorithm follows the general outline of the BasicVoronoiCell algorithm. First it preprocesses the input lattice to acquire a better quality basis $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ and sets the Voronoi cell of Λ_1 to $\{-\mathbf{b}_1, \mathbf{b}_1\}$. Then it computes the Voronoi cells of $\Lambda_2, \ldots, \Lambda_n$ iteratively, using a subroutine that computes the Voronoi cell of Λ_{k+1} given the Voronoi cell of Λ_k. The improved time complexity comes from improved subroutines for both the preprocessing and Voronoi cell computation. In particular the Improved-VoronoiCell algorithm uses the following subroutines:

1. The OptPreprocess, a $\tilde{O}(2^n)$ basis reduction subroutine that on input an n-rank lattice Λ_n outputs a basis $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ for the same lattice, such that for all $k = 2, \ldots, n$ we have $\mu(\Lambda_k)/\|\mathbf{b}_k^*\| \leq k^5$, (where $\Lambda_k = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_k)$).

2. The OptRankReduceVCell, a subroutine that on input a basis $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_{k+1}\}$, an integer H such that $\mu(\Lambda_{k+1})/\|\mathbf{b}_{k+1}^*\| \leq H$ and the Voronoi cell of Λ_k outputs the Voronoi cell of Λ_{k+1} in time $H^2 \cdot \tilde{O}(2^{2k})$.

The full description and analysis of the subroutines is given in Sections 6.2, 6.3, 6.4. In the following paragraphs we briefly revisit the initial subroutines PREPROCESS and RANKREDUCEVCELL and show why it is possible to improve upon them.

The goal of the preprocessing step is to output a basis $\mathbf{B} = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n\}$ for the input lattice, such that the quantities $\mu(\Lambda_k)/\|\mathbf{b}_k^*\|$ are relatively small. The basic PREPROCESS subroutine simply runs LLL and achieves $\mu(\Lambda_k)/\|\mathbf{b}_k^*\| < 2^{0.5k}$ in poly(n) time, where n is the rank of the lattice. It is natural to ask if there are algorithms that provide stronger guarantees on the output basis, ideally bounding $\mu(\Lambda_k)/\|\mathbf{b}_k^*\|$ by a polynomial of k. In Section 6.2 we show that OPTPREPROCESS is such an algorithm. In particular it achieves $\mu(\Lambda_k)/\|\mathbf{b}_k^*\| \leq k^5$ in $\tilde{O}(2^n)$ time. This is possible using block reduction techniques from [GN08b, Sch87]. Notice that the new preprocessing algorithm is much slower than LLL. However, in our case this is not an issue. The preprocessing step is run only once at the beginning of the algorithm and as long as its time complexity is lower (or equal) with the Voronoi cell computation step it cannot affect the total time complexity of the algorithm.

Finally we show why it is possible to improve the Voronoi cell computation subroutine. Recall that RANKREDUCEVCELL computes the Voronoi cell of Λ_{k+1} with the help of $H \cdot \tilde{O}(2^k)$ calls to CVPP for the lower rank lattice Λ_k, where H an integer such that $\mu(\Lambda_{k+1})/\|\mathbf{b}_{k+1}^*\| \leq H$. It turns out that the CVP instances solved by CVPP have a very special structure that allows for more efficient strategies to compute them. In particular, it is possible to group them in such a way that solving a group of 2^k CVP instances on Λ_k can be reduced to a single instance of the following problem on Λ_k:

Definition 6.1.1. ENUM$_{2\bar{\mathcal{V}}}$ *for target* $\mathbf{t}$ *on* Λ*: Given a basis* $\mathbf{B}$ *for an* n*-rank lattice* Λ *and a target vector* $\mathbf{t}$ *in the span of* Λ*, enumerate all the vectors of the set* $\Lambda + \mathbf{t} \cap 2\mathcal{V}$*, where* $\mathcal{V}$ *is the (open) Voronoi cell of* Λ.

In Section 6.3 we give an algorithm that solves ENUM$_{2\bar{\mathcal{V}}}$ given the Voronoi cell of Λ in $\tilde{O}(2^{2n})$, (while using directly CVPP results a $\tilde{O}(2^{3n})$ algorithm). In Section 6.4 we show how to use this algorithm for ENUM$_{2\bar{\mathcal{V}}}$ to acquire a faster Voronoi cell computation subroutine. In particular the OPTRANKREDUCEVCELL subroutine computes the Voronoi cell of Λ_{k+1} given the Voronoi cell of Λ_k in time

$H^2 \cdot \tilde{O}(2^{2k})$, compared with the $H \cdot \tilde{O}(2^{3k})$ of the RankReduceVCell. We remark that the increased dependence on H is not an issue, after running OptPreprocess H can be a polynomial in k.

In Section 6.5 we combine the improved subroutines to acquire the OptimizedVoronoiCell algorithm and give a proof for the main theorem of this chapter:

Theorem 6.1.2. *There is a deterministic $\tilde{O}(2^{2n})$-time and $\tilde{O}(2^n)$ space algorithm that on input an n-rank lattice Λ with basis $\mathbf{B}$, outputs the relevant vectors of Λ.*

From the description of the Voronoi cell, we immediately get a solution to many other lattice problems, e.g., the shortest vector problem (SVP) can be solved simply by picking the shortest vector in the list of lattice points describing the Voronoi cell, and the kissing number of the lattice can be computed as the number of vectors in the list achieving the same length as the shortest vector in the lattice.

Corollary 6.1.3. *There is a deterministic $\tilde{O}(2^{2n})$ time algorithm to solve SVP, and to compute the kissing number of a lattice.*

Once the Voronoi cell of Λ_n has been computed, then we can solve CVP using the CVPP algorithm.

Corollary 6.1.4. *There is a deterministic $\tilde{O}(2^{2n})$ time, $\tilde{O}(2^n)$ space algorithm to solve* CVP*.*

Proof. First we run the OptimizedVoronoiCell to acquire the relevant vectors V_n of the n-rank input lattice Λ in time $\tilde{O}(2^{2n})$ and space $\tilde{O}(2^n)$. Then we solve the given instance of CVP using CVPP and the precomputed relevant vectors in time $\tilde{O}(2^{2n})$. □

Algorithms for other lattice problems, like SIVP, SAP, GCVP, SMP, can be obtained by a reduction to CVP.

Corollary 6.1.5. *There is a deterministic $\tilde{O}(2^{2n})$ time, $\tilde{O}(2^n)$ space algorithm to solve SIVP, SAP, GCVP and SMP.*

Proof. See [Mic08] for reductions of the above problems to CVP. □

6.2 Preprocessing with block reduction

Algorithm 6.1: Optimized Preprocessing

```
function OptPreprocess($\mathbf{B}$)
    for $k = n$ downto 2 do
        Run dual block reduction with
        block size $k/4$ on $\mathbf{B}_k = \{\mathbf{b}_1, \ldots, \mathbf{b}_k\}$
        and replace $\mathbf{B}_k$ with the output basis.
    return $\mathbf{B}$
```

In this section we describe and analyze the optimized preprocessing subroutine. The results are summarized in the following lemma.

Lemma 6.2.1. *On input a basis for an n-rank lattice Λ, the* OptPreprocess *subroutine outputs new basis $\mathbf{B}$ for the same lattice with the following property: For every sub-lattice of the form $\Lambda_k = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_k)$, $k \le n$, we have $\mu(\Lambda_k)/\|\mathbf{b}_k^*\| \le k^5$. The subroutine is deterministic and its complexity is $\tilde{O}(2^n)$.*

Proof. First we give a deterministic $\tilde{O}(2^n)$ procedure that on input a basis for an n-rank lattice Λ, outputs a basis $\mathbf{B}$ for the same lattice, such that $\mu(\Lambda)/\|\mathbf{b}_n^*\| \le n^5$. We achieve this by running block basis reduction algorithms from [GN08b, Sch87] to the dual of the input lattice. The dual block reduction, on input a basis for an n-rank lattice Λ, outputs a basis $\mathbf{B}$ for Λ such that $\mu(\Lambda)/\|\mathbf{b}_n^*\| \le n\beta^{n/\beta}$ (see Preliminaries chapter). This is achieved with the help of polynomially many calls to an SVP oracle for β-rank lattices. (The reader is referred to [GN08b] for a thorough analysis of block reduction and and the connections between dual and primal lattice bases).

For our purposes set $\beta = n/4$ and instantiate the SVP oracle with our unoptimized $\tilde{O}(2^{3.5n})$ algorithm, in order to avoid circular dependencies. [1] We show that for this instantiation of block reduction the required property is achieved. First, observe that for $\beta = n/4$, dual block reduction gives a basis with $\mu(\Lambda)/\|\mathbf{b}_n^*\| \le n^5$.

[1]We could instantiate the oracle with the optimized variant, but it would make the proof more complicated because we would need an induction argument.

This is achieved with polynomially many calls to our algorithm for rank $n/4$ which gives a total complexity of $\tilde{O}(2^{3.5n/4}) < \tilde{O}(2^n)$ and concludes the analysis.

Now we can just apply this procedure for all Λ_k for $k = n, n-1, \ldots, 2$ and get a basis with $\mu(\Lambda_k)/\|\mathbf{b}_k^*\| \leq k^5$ for all k. The time complexity is essentially the same $\sum_{i=2}^{i=n} \tilde{O}(2^i) = \tilde{O}(2^n)$. $\square$

6.3 Enumeration of $(\Lambda + \mathbf{t}) \cap 2\mathcal{V}$

In this section we describe and analyze a procedure that given a basis $\mathbf{B}$ of an n-rank lattice Λ, the relevant vectors V of Λ and a target $\mathbf{t}$ on the linear span of Λ, outputs a set of at most 2^n vectors that contain all the points in $\Lambda + \mathbf{t} \cap 2\mathcal{V}$.

Notice that traversing the nodes of the open relevant graph $G_{(\Lambda+\mathbf{t})\cap 2\mathcal{V}}$ is equivalent with enumerating the set $\Lambda + \mathbf{t} \cap 2\mathcal{V}$. We have already proved that this graph is connected, therefore we could traverse it by first computing a vector in $\Lambda + \mathbf{t} \cap \bar{\mathcal{V}}$ using CVPP and then use standard graph traversal algorithms (e.g. DFS). Given any node $\mathbf{u}$ and the relevant vectors V we can easily compute all the accessible nodes on $G_{\Lambda+\mathbf{t}}$ from $\mathbf{u}$, and then discard all the nodes that are not in $G_{(\Lambda+\mathbf{t})\cap 2\mathcal{V}}$. Unfortunately checking if a node is in $2\mathcal{V}$ costs $\tilde{O}(2^n)$ and a simple depth first search would give a $\tilde{O}(2^{3n})$ algorithm (for each of 2^n nodes of the graph, for each of their 2^n neighbors we should run a check). We show that using simple facts about the parity of the nodes and Theorem 4.2.4 we can lower the complexity to $\tilde{O}(2^{2n})$.

Given the relevant vectors of an n-rank lattice Λ, the $\textsc{Enum}_{2\bar{\mathcal{V}}}$ algorithm traverses all the nodes of $G_{(\Lambda+\mathbf{t})\cap 2\mathcal{V}}$ in time $\tilde{O}(2^{2n})$, the complete pseudocode is given in Algorithm 6.2 [2]. The algorithm keeps a list of visited nodes and a list of accessible nodes, both indexed by parity. At any point the algorithm keeps only one vector for each parity that might be either visited or accessible, therefore the space requirements are $\tilde{O}(2^n)$. The first step of the algorithm is to compute a vector $\mathbf{u}_s \in (\Lambda + \mathbf{t}) \cap \bar{\mathcal{V}}$ using one CVPP computation that costs $\tilde{O}(2^{2n})$ and add it to the list of accessible nodes. ($\mathbf{u}_s = \mathbf{t} - \mathrm{CVPP}(\mathbf{t}, \mathbf{B}, V)$). At each subsequent step

[2]The algorithm might also visit nodes outside $2\mathcal{V}$, but on the boundary of $2\bar{\mathcal{V}}$.

Algorithm 6.2: $\textsc{Enum}_{2\bar{\mathcal{V}}}$ Algorithm

```
function $\textsc{Enum}_{2\bar{\mathcal{V}}}(\mathbf{B}, V, \mathbf{t})$
        ▷ The following arrays are indexed by $\mathbf{p} \in \{0,1\}^n$ and initialized to none.
    Visited[] $\leftarrow$ Array of $2^n$ vectors
    Accessible[] $\leftarrow$ Array of $2^n$ vectors
    $\mathbf{u}_s \leftarrow \mathbf{t} - CVPP(\mathbf{t}, \mathbf{B}, V)$
    Accessible[parity($\mathbf{u}_s, \mathbf{B}$)] $= \mathbf{u}_s$
    while no more Accessible vectors do
        $\mathbf{u} \leftarrow$ Shortest Accessible vector
        Visit($\mathbf{u}, \mathbf{B}$, Visited, Accessible)
    return Visited
function $\textsc{Visit}(\mathbf{u}, \mathbf{B}$, Visited, Accessible)
    $\mathbf{p} \leftarrow$ parity($\mathbf{u}, \mathbf{B}$)
    Visited[$\mathbf{p}$] $= \mathbf{u}$
    Accessible[$\mathbf{p}$] = none
    for $\mathbf{v} \in V$ do
        $\mathbf{a} \leftarrow \mathbf{u} + \mathbf{v}$
        $\mathbf{p}_a \leftarrow$ parity($\mathbf{a}, \mathbf{B}$)
        if Visited[$\mathbf{p}_a$] = none then
            $\mathbf{a}_{\text{prev}} \leftarrow$ Accessible[$\mathbf{p}_a$]
            if $\mathbf{a}_{\text{prev}}$ =none OR $\|\mathbf{a}_{\text{prev}}\| > \|\mathbf{a}\|$ then
                Accessible[$\mathbf{p}_a$] $= \mathbf{a}$
Where:
$\mathbf{p} \leftarrow$parity$(\mathbf{u}, \mathbf{B}) \Leftrightarrow \mathbf{u} \in 2\Lambda + \mathbf{B} \cdot \mathbf{p}, \mathbf{p} \in \{0,1\}^n$
```

the algorithm visits the shortest vector on the accessible list. When the algorithm visits a node $\mathbf{u}$ the following computations take place:

1. The vector $\mathbf{u}$ is removed from the Accessible list and added to the Visited list to the corresponding parity index.

2. Then the algorithm considers all the accessible nodes in G from $\mathbf{u}$ by adding relevant vectors. It computes the vectors $\mathbf{a}_i = \mathbf{u} + \mathbf{v}_i$ for all $\mathbf{v}_i \in V$ and for each of them does the following:

   (a) If there is a visited node with the same parity as $\mathbf{a}_i$, or an accessible node with shorter norm for the same parity, it discards $\mathbf{a}_i$.

   (b) Else it inserts $\mathbf{a}_i$ to the accessible list, discarding possible accessible nodes with the same parity (but larger norm).

The algorithm terminates when there are no more accessible nodes, and outputs the vectors of the Visited list.

Lemma 6.3.1. *Given a basis* $\mathbf{B}$ *for an* n*-rank lattice* Λ*, the relevant vectors* V *for* Λ *and a target vector* $\mathbf{t}$*, the* $\text{Enum}_{2\bar{\mathcal{V}}}$ *algorithm (Algorithm 6.2) computes a set of at most* 2^n *vectors that includes all the vectors of* $\Lambda + \mathbf{t} \cap 2\mathcal{V}$*, in time* $\tilde{O}(2^{2n})$*.*

Proof. For the time complexity of the algorithm, notice that the initial CVPP computation costs $\tilde{O}(2^{2n})$ and that for each visited node the algorithm does $\tilde{O}(2^n)$ additional computations. It is not hard to validate that the algorithm will visit at most 1 vector from each parity. Therefore there are at most 2^n visited nodes and the complexity is $\tilde{O}(2^{2n})$.

We prove correctness of the algorithm in two steps. First we show that the algorithm visits *all* the nodes in the core subgraph $G_{(\Lambda+\mathbf{t})\cap\bar{\mathcal{V}}}$ before visiting any other node. In order to see this notice that the first visited node is in the core subgraph, the core subgraph is connected and the algorithm always visits the shortest accessible node.

Now we prove by contradiction that any node $\mathbf{u}$ in $G_{(\Lambda+\mathbf{t})\cap 2\mathcal{V}}$ is visited. Assume that a node $\mathbf{u} \in G_{(\Lambda+\mathbf{t})\cap 2\mathcal{V}}$ is not visited. By Theorem 4.2.4 there exists

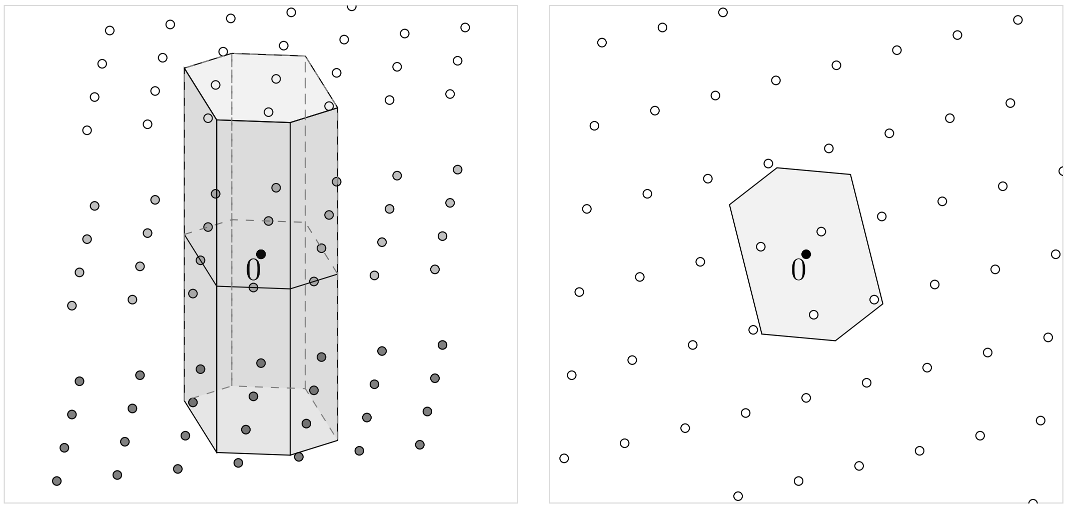

Figure 6.1: Relevants are in a cylinder with base $2\bar{\mathcal{V}}$

a path in $G_{(\Lambda+\mathbf{t})\cap 2\mathcal{V}}$ with vectors of decreasing norms $\mathbf{u}, \mathbf{u}_1, \ldots, \mathbf{u}_N$. Where $\mathbf{u}_N$ is a node of the core subgraph and is surely visited. Therefore there exist two nodes $\mathbf{u}_k, \mathbf{u}_{k+1}$ in the sequence such that $\mathbf{u}_k$ is not visited, while $\mathbf{u}_{k+1}$ is. When $\mathbf{u}_{k+1}$ was visited, the node $\mathbf{u}_k$ was considered for inclusion in the list of accessible nodes. Notice that $\mathbf{u}_k$ is in $G_{(\Lambda+\mathbf{t})\cap 2\mathcal{V}}$ and consequently the shortest node of $G_{\Lambda+\mathbf{t}}$ for its parity. As a result $\mathbf{u}_k$ would be included in the accessible list and eventually visited, unless a node $\mathbf{x}$ with the same parity was visited first. However this is impossible because $\mathbf{u}_k$ is in $2\mathcal{V}$ and therefore it is the shortest vector of its parity, so shorter than $\mathbf{x}$. Also all the nodes in the path from $\mathbf{u}_{k+1}$ to $\mathbf{u}_N$ are shorter that $\mathbf{u}_k$. Consequently the algorithm should visit the nodes of the path leading to $\mathbf{u}_k$ instead of $\mathbf{x}$. We conclude that the assumption is false and all the points in $G_{(\Lambda+\mathbf{t})\cap 2\mathcal{V}}$ are visited. □

6.4 Relevant vectors using Enum of $(\Lambda + \mathbf{t}) \cap 2\mathcal{V}$

In this section we give an algorithm that on input a basis $\mathbf{B}_n = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ for an n-rank lattice, the relevant vectors V_{n-1} of $\Lambda_{n-1} = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_{n-1})$ and an integer H such that $\mu(\Lambda_n)/\|\mathbf{b}_n^*\| \leq H$, computes the relevant vectors of V_n with

Algorithm 6.3: Optimized Voronoi cell computation

```
function OptRankReduceVCell($\mathbf{B}_k, V_{k-1}, H$)
    $V_k \leftarrow V_{k-1}$
    for all $h$ that: $|h| < H$ do
        $V_{k,h} \leftarrow \text{Enum}_{2\bar{\mathcal{V}}}(\mathbf{B}_k, V_k, h(\mathbf{b}_k - \mathbf{b}_k^*))$
        Add $h\mathbf{b}_k^*$ to every element of $V_{k,h}$
        $V_k \leftarrow V_k \bigcup V_{k,h}$
    $V_k \leftarrow$ RemoveNonRelevant($V_k$)
    return $V_k$
```

$4H + 1$ calls to the algorithm $\text{Enum}_{2\bar{\mathcal{V}}}$ described in the previous section. First we give two useful lemmas for the relevant vectors of Λ_n. The lemmas restrict the search space for relevant vectors.

Lemma 6.4.1. *Set* $\mathbf{B}_n$*,* Λ_n*,* Λ_{n-1} *as defined above. Let* $\mathbf{b}_n^*$ *be the component of* $\mathbf{b}_n$ *orthogonal to* Λ_{n-1}*. If* $\mathbf{u}$ *is a relevant vector of* Λ_n*, then it belongs to some layer* $\Lambda_{n-1} + h\mathbf{b}_n$ *with* h *an integer and* $|h| \leq 2\mu(\Lambda_n)/\|\mathbf{b}_n^*\|$*.*

Proof. Partition the lattice Λ_n in layers of the form $\Lambda_{n-1} + h\mathbf{b}_n$ with $h \in \mathbb{Z}$. Notice the following:

- If $\mathbf{u}$ is a relevant vector then it is in $2\bar{\mathcal{V}}$.
- For all the points in $\bar{\mathcal{V}}$ the vector $\mathbf{0}$ is a closest vector. Therefore the norm of any point in $\bar{\mathcal{V}}$ is at most $\mu(\Lambda_n)$.
- The norm of all lattice points in the layer $\Lambda_{n-1} + h\mathbf{b}_n$ is at least $|h| \cdot \|\mathbf{b}_n^*\|$, because this is the distance between $\mathbf{0}$ and the entire affine space generated by the layer.

By the above it is not hard to validate that the norm of the relevant vectors is at most $2\mu(\Lambda)$ and consequently they belong to layers with $|h| < 2\mu(\Lambda_n)/\|\mathbf{b}_n^*\|$. □

Lemma 6.4.2. *Set* $\mathbf{B}_n$*,* Λ_n*,* $\mathbf{B}_{n-1}$*,* Λ_{n-1} *as defined above. Let* $\mathcal{V}_{n-1}$ *be the open Voronoi cell of* Λ_{n-1} *and* V_{n-1} *be the set of the corresponding relevant vectors. If* $\mathbf{u}$ *is a relevant vector of* Λ_n *then:*

- *If* $\mathbf{u} \in \Lambda_{n-1}$*,* $\mathbf{u}$ *is relevant for* Λ_{n-1}*,*
- *If* $\mathbf{u} \in h \cdot \mathbf{b}_n + \Lambda_{n-1}$ *for* $h \neq 0$*, then* $\mathbf{u} \in h \cdot \mathbf{b}_n^* + 2\mathcal{V}_{n-1} + h(\mathbf{b}_n - \mathbf{b}_n^*)$*.*

Proof. Let $\mathbf{u}$ be a relevant vector of Λ_n, equivalently $\pm\mathbf{u}$ are the only shortest vectors of the coset $2\Lambda_n + \mathbf{u}$. If $\mathbf{u} \in \Lambda_{n-1}$ then $\pm\mathbf{u}$ are also the only shortest vectors of the coset $2\Lambda_{n-1} + \mathbf{u}$ therefore $\mathbf{u}$ is a relevant vector of Λ_{n-1}. Now assume $\mathbf{u} \notin \Lambda_{n-1}$. The vectors $\pm\mathbf{u}$ should be the only shortest vectors of $2\Lambda + \mathbf{u}$ therefore:

$$\forall \mathbf{v} \in \Lambda_n,\ \mathbf{v} \neq \mathbf{u}\ : \|\mathbf{u} - 2\mathbf{v}\| > \|\mathbf{u}\|.$$

But $\mathbf{u} \notin \Lambda_{n-1}$, therefore for all the relevant vectors $\mathbf{v}_i$ of Λ_{n-1}: $\mathbf{u} \neq \mathbf{v}_i$. Combining this fact with the equation above we have:

$$\forall \mathbf{v}_i \in V_{n-1} : \|\mathbf{u} - 2\mathbf{v}_i\| > \|\mathbf{u}\|.$$

Notice that the equation above describes exactly the points inside a cylinder with base $2\mathcal{V}_{n-1}$. □

Finally we are ready to state the lemma that summarizes the results of this section.

Lemma 6.4.3. *Set* $\mathbf{B}_n$*,* Λ_n*,* $\mathbf{B}_{n-1}$*,* Λ_{n-1}*,* $\mathbf{b}_n^*$*,* $\mathcal{V}_{n-1}$*,* V_{n-1} *as defined above. On input the Voronoi cell of* Λ_{n-1} *and an integer* H *such that* $\mu(\Lambda_n)/\|\mathbf{b}_n^*\| \leq H$*, the* OptRankReduceVCell *algorithm (Algorithm 6.3) computes the relevant vectors of* Λ_n *in time* $H^2 \cdot \tilde{O}(2^{2n})$ *and space* $H \cdot \tilde{O}(2^n)$*.*

Proof. In order to compute the relevant vectors the algorithm partitions Λ_n in layers of the form $\Lambda_{n-1} + h\mathbf{b}_n$. For each layer with $|h| \leq 2H$, the algorithm computes a set of at most 2^n vectors that contains all the relevant vectors of the layer. By Lemma 6.4.1 the union of these sets is a superset of the relevant vectors. The additional vectors are discarded with the procedure RemoveNonRelevant. We go on to show how to find appropriate sets for each layer.

For the layer with $h = 0$ the precomputed relevant vectors of Λ_{n-1} are sufficient (by Lemma 6.4.2). Now, for layers with $h \neq 0$ the intersection of $\Lambda_{n-1} + h\mathbf{b}_n$ with the cylinder with base $2\mathcal{V}_{n-1}$ is an appropriate set (by Lemma 6.4.2).

Notice that projecting vectors from $\Lambda_{n-1} + h\mathbf{b}_n$ to the linear span of Λ_{n-1} gives a one to one mapping between vectors $\mathbf{v}$ on the layer and their projections on the linear span of Λ_{n-1} $\mathbf{v}^{\|} = \mathbf{v} - h\mathbf{b}_n^*$. In order to enumerate the intersection of $\Lambda_{n-1} + h\mathbf{b}_n$ with the cylinder with base $2\mathcal{V}_{n-1}$ we project it to Λ_{n-1}, enumerate the projections and acquire the original vectors by adding $h\mathbf{b}_n^*$. Notice that this projection is the set $\Lambda_{n-1} + h(\mathbf{b}_n - \mathbf{b}_n^*) \cap 2\mathcal{V}_{n-1}$ and enumerating these vectors is an instance of $\textsc{Enum}_{2\bar{\mathcal{V}}}$.

For the complexity of the algorithm notice that it enumerates at most $H \cdot \tilde{O}(2^n)$ vectors using $4H + 1$ calls to $\textsc{Enum}_{2\bar{\mathcal{V}}}$ that requires $\tilde{O}(2^{2n})$ time. Finally it runs RemoveNonRelevant to discard non relevant vectors. Therefore the total time complexity is $H^2 \cdot \tilde{O}(2^{2n})$ and space complexity $H \cdot \tilde{O}(2^n)$. □

We remark that we can reduce the space complexity of the above algorithm to $\tilde{O}(2^n)$ and time complexity to $H \cdot \tilde{O}(2^{2n})$ but it does not change the total asymptotics of the final algorithm because H is going to be polynomial in n.

6.5 Combining the blocks

Algorithm 6.4: The Optimized Voronoi Cell Algorithm

```
function OptimizedVoronoiCell($\mathbf{B}$)
    $\mathbf{B} \leftarrow$ OptPreprocess($\mathbf{B}$)
    $V_1 = \{(\mathbf{b}_1, -\mathbf{b}_1)\}$
    for $k = 2$ to $n$ do
        $\mathbf{B}_k \leftarrow [\mathbf{b}_1, \ldots, \mathbf{b}_k]$
        $V_k \leftarrow$ OptRankReduceVCell($B_k, V_{k-1}, k^5$)
    return $V_n$
```

Finally we present $\textsc{OptimizedVoronoiCell}$, a deterministic $\tilde{O}(2^{2n})$-time and $\tilde{O}(2^n)$-space algorithm for computing the Voronoi cell of an n- rank lattice Λ and prove the main theorem of this chapter (Theorem 6.1.2).

Proof. Given a basis for an n-rank lattice the $\textsc{OptimizedVoronoiCell}$ algorithm does the following:

1. First it applies the $\tilde{O}(2^n)$ OptPreprocessing algorithm to get a basis $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ for the same lattice such that $\mu(\Lambda_k)/\|\mathbf{b}_k^*\| \leq k^5$ for all $k = 2, \ldots, n$ (Lemma 6.2). The rest of the algorithm is using the new basis.

2. It sets the Voronoi cell of $\Lambda_1 = \mathcal{L}(\mathbf{b}_1)$ as $V_1 = \{\mathbf{b}_1, -\mathbf{b}_1\}$.

3. Then for $k = 2, \ldots, n$ it runs the OptRankReduceVCell algorithm to acquire V_k from V_{k-1} in time $\tilde{O}(2^{2k})$ and space $\tilde{O}(2^k)$ (Lemma 6.4.3, with $H \leq k^5$).

Notice that the total time complexity of the algorithm is $\tilde{O}(2^n) + \sum_{k=1}^{n} \tilde{O}(2^{2k}) = \tilde{O}(2^{2n})$ and the space complexity is $\tilde{O}(2^n)$. $\square$

Chapter 4, Chapter 5 and Chapter 6, in part, are a reprint of the full version of the paper "A Deterministic Single Exponential Time Algorithm for Most Lattice Problems based on Voronoi Cell Computations" co-authored with Daniele Micciancio. An extended abstract of this paper was presented in STOC 2010. The dissertation author was the primary investigator and author of this paper.

Appendix A

Additional Proofs

A.1 Analysis of AKS with Packing Bounds

In this section we show how to improve the analysis of AKS using the packing bounds of Theorem 2.2.1 and a number of other observations. Our analysis shows that the time and space bounds of AKS can be improved to $2^{3.4n}$ and $2^{1.97n}$. However even after the improved analysis, AKS is outperformed by the conceptually simpler *List Sieve*. Intuitively there are two reasons for this:

1. AKS algorithm has to generate all the points from the beginning. As a result the points that will cause collisions (which in a worst case analysis are exponentially more) increase the space bounds. *List Sieve* on the other hand discards collisions as early as possible.

2. AKS sieving cannot provably continue to produce shorter vectors after a certain norm. As a result it generates enough "short" vectors so that the difference of two of them will give a nonzero shortest vector. The point arrangements we get out of this procedure are slightly worse than the sieving of *List Sieve*.

In the following, we assume the reader has some familiarity with [NV08] and only highlight the elements of the algorithm that are directly relevant to the comparison to our work. Apart from the parameters ξ, μ used in *List Sieve*, AKS requires a parameter γ the shrinking factor. The core procedure of AKS takes as input a set of perturbed points P_i (the norm of the perturbations is bounded by $\xi\mu$) with maximum norm R_i. It generates a maximal subset L_i of P_i with the property that the distance between any two points in L_i is greater than γR_i. Then it uses L_i to reduce the norms of the remaining points in $P_i \setminus L_i$ to at most $\gamma R_i + \xi\mu$. The algorithm starts with a large pool of points P_0 with maximum norm R_0. Applying the core procedure described above it generates a sequence of sets $P_0, P_1, P_2, \ldots$ with shorter and shorter maximum norms $R_0, R_1, R_2, \ldots$. Notice that the maximum norm of the sets P_i can be easily reduced near $R_k \simeq \xi\mu/(1-\gamma)$ but not further. The algorithm runs a polynomial number k of steps and acquires a set P_k of perturbed points with maximum norm $R_k \simeq \xi\mu/(1-\gamma)$. After the removal of the perturbations these points correspond to a set of lattice vectors V_k with maximum norm $R_\infty \simeq \xi\mu(1 + 1/(1-\gamma))$. Then AKS computes the pairwise differences of all the points in V_k to find a nonzero shortest vector.

We bypass most of the details (the reader is referred to [NV08] for the complete analysis), to show directly how to improve the space and time bounds of AKS. Let the total number of generated points $|P_0| = 2^{c_0 n}$, the probability of a point not being a collision $p = 2^{-c_u n}$, the maximum number of points used for sieving $|L_i| \leq 2^{c_s n}$, and the number of points required in V_k to find a vector with the required norm is $2^{c_R n}$. After k steps of the algorithm the points in P_k will be $2^{c_0 n} - k2^{c_s n}$ and the points in V_k at most $(2^{c_0 n} - k2^{c_s n})/2^{c_u n}$ because of collisions. Notice that we need enough points in P_0 to generate the lists L_i, $c_0 > c_s$ and acquire enough points in V_k, $c_0 - c_u > c_R$. Therefore the space complexity is $2^{c_0 n}$ with $c_0 = max\{c_s, c_R + c_u\}$. Every sieving step reduces $2^{c_0 n}$ points with the $2^{c_s n}$ points of L_i, as a result the sieving procedure needs $2^{(c_0+c_s)n}$ time. Finally we need to acquire the pairwise differences of at least $2^{c_R n}$ points in V_k, but for each point we might need to use $2^{c_u n}$ points from P_k. As a result the total running time for the last step is $2^{2c_R n + c_u n}$. Totally the time complexity is $2^{c_T n}$ with $c_T = max\{c_0 + c_s, 2c_R + c_u\}$.

Now we are ready to introduce the linear programming bounds. Let A a set of points inside an n-dimensional ball of radius R and pairwise distance $\geq r$. Using similar techniques with our main theorems we can show that the angle between any two points with similar norm is $\cos\phi < 1 - r^2/(2R^2)$ and using theorem 2.2.1 we get that the total number of points in A are bounded by 2^{cn} with $c = 0.401 + \log(R/r)$. Now we can give bounds to all the constants defined above. First $c_u = \log\left(\frac{\xi}{\sqrt{\xi^2 - 0.25}}\right)$ which is identical to c_2 in our analysis. Notice that the points in any set L_i are in a ball of radius R_i and have pairwise distance γR_i therefore $c_s = 0.401 + \log(1/\gamma)$. On the other hand the set V_k has minimum distance $> \mu$ while the maximum radius of the points is $R_\infty \simeq \xi\mu(1 + 1/(1-\gamma))$ therefore we need $c_R = 0.401 + \log(\xi(1 + 1/(1-\gamma)))$. To minimize the space constant $c_0 = max\{c_s, c_R + c_u\}$ we set $\xi = 0.71, \gamma = 0.382$ which gives space requirements $2^{1.79n}$ and time $2^{3.58n}$. On the other hand if we want to minimize the time complexity good values are $\xi = 0.69$ and $\gamma = 0.49$ which give space $2^{1.97n}$ and time $2^{3.4n}$.

Chapter 3 and Section A.1, in part, are a reprint of the paper "Large Faster exponential time algorithms for the shortest vector problem" co-authored with Daniele Micciancio and appearing in the proceedings of SODA 2010. The dissertation author was the primary investigator and author of this paper.

Bibliography

[ABSS97] Sanjeev Arora, László Babai, Jacques Stern, and Elizabeth Z Sweedyk. The hardness of approximate optima in lattices, codes, and systems of linear equations. *Journal of Computer and System Sciences*, 54(2):317–331, April 1997. Preliminary version in FOCS'93.

[AD97] M. Ajtai and C. Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, page 293. ACM, 1997.

[AEVZ02] E. Agrell, T. Eriksson, A. Vardy, and K. Zeger. Closest point search in lattices. *IEEE Transactions on Information Theory*, 48(8):2201–2214, August 2002.

[Ajt98] Miklós Ajtai. The shortest vector problem in L2 is NP-hard for randomized reductions (extended abstract). In *Proceedings of STOC '98*, pages 10–19. ACM, May 1998.

[Ajt04] Miklós Ajtai. Generating hard instances of lattice problems. *Complexity of Computations and Proofs, Quaderni di Matematica*, 13:1–32, 2004. Preliminary version in STOC 1996.

[AKKV05] Michael Alekhnovich, Subhash Khot, Guy Kindler, and Nisheeth Vishnoi. Hardness of approximating the closest vector problem with pre-processing. In *Proceedings of FOCS 2005*. IEEE, October 2005.

[AKS01] Miklós Ajtai, Ravi Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *Proceedings of STOC '01*, pages 266–275. ACM, July 2001.

[AKS02] Miklós Ajtai, Ravi Kumar, and D. Sivakumar. Sampling short lattice vectors and the closest lattice vector problem. In *Proceedings of CCC '02*, pages 53–57. IEEE, May 2002.

[AR05] Dorit Aharonov and Oded Regev. Lattice problems in NP intersect coNP. *Journal of the ACM*, 52(5):749–765, 2005. Preliminary version in FOCS 2004.

[Bab86] László Babai. On Lovasz' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.

[Ban93] W. Banaszczyk. New bounds in some transference theorems in the geometry of numbers. *Mathematische Annalen*, 296(1):625–635, 1993.

[Blö00] Johannes Blömer. Closest vectors, successive minima and dual HKZ-bases of lattices. In *Proceedings of ICALP '00*, volume 1853 of *LNCS*, pages 248–259. Springer, July 2000.

[BN09] Johannes Blömer and Stefanie Naewe. Sampling methods for shortest vectors, closest vectors and successive minima. *Theoretical Computer Science*, 410(18):1648–1665, April 2009. Preliminary version in ICALP 2007.

[BS99] Johannes Blömer and Jean-Pierre Seifert. On the complexity of computing short linearly independent vectors and short bases in a lattice. In *Proceedings of STOC '99*, pages 711–720. ACM, May 1999.

[Cas71] John William Scott Cassels. *An introduction to the geometry of numbers.* Springer-Verlag, New York, 1971.

[Cha07] Denis Xavier Charles. Counting lattice vectors. *Journal of Computer and System Sciences*, 73(6):962 – 972, 2007.

[CJL$^+$92] Matthijs J. Coster, Antoine Joux, Brian A. LaMacchia, Andrew M. Odlyzko, Claus-Peter Schnorr, and Jacques Stern. Improved low-density subset sum algorithms. *Computational Complexity*, 2(2):111–128, 1992. Preliminary versions in Eurocrypt '91 and FCT '91.

[CM06] Wenbin Chen and Jiangtao Meng. The hardness of the closest vector problem with preprocessing over ℓ_∞ norm. *IEEE Transactions on Information Theory*, 52(10):4603–4606, 2006.

[CN99] Jin-Yi Cai and Ajay P. Nerurkar. Approximating the SVP to within a factor $(1+1/dim^\epsilon)$ is NP-hard under randomized reductions. *Journal of Computer and System Sciences*, 59(2):221–239, October 1999. Prelim. version in CCC 1998.

[CS98] John H. Conway and Neil J. A. Sloane. *Sphere packings, lattices and groups.* Springer Verlag, 3rd edition, 1998.

[DKRS03] Irit Dinur, Guy Kindler, Ran Raz, and Shmuel Safra. Approximating CVP to within almost-polynomial factors is NP-hard. *Combinatorica*, 23(2):205–243, 2003. Preliminary version in FOCS 1998.

[DPV10] D. Dadush, C. Peikert, and S. Vempala. Enumerative algorithms for the shortest and closest lattice vector problems in any norm via M-ellipsoid coverings. *Arxiv preprint arXiv:1011.5666*, 2010.

[Dwo97] C. Dwork. Positive applications of lattices to cryptography. *Mathematical Foundations of Computer Science 1997*, pages 44–51, 1997.

[FM03] Uriel Feige and Daniele Micciancio. The inapproximability of lattice and coding problems with preprocessing. *Journal of Computer and System Sciences*, 69(1):45–67, 2003. Preliminary version in CCC 2002.

[GHGKN06] Nicolas Gama, Nick Howgrave-Graham, Henrik Koy, and Phong Nguyen. Rankin's constant and blockwise lattice reduction. In *Advances in Cryptology – Proceedings of CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 112–130. Springer, August 2006.

[GMR05] Venkatesan Guruswami, Daniele Micciancio, and Oded Regev. The complexity of the covering radius problem. *Computational Complexity*, 14(2):90–121, jun 2005. Preliminary version in CCC 2004.

[GMSS99] Oded Goldreich, Daniele Micciancio, Shmuel Safra, and Jean-Pierre Seifert. Approximating shortest lattice vectors is not harder than approximating closest lattice vectors. *Information Processing Letters*, 71(2):55–61, 1999.

[GN08a] N. Gama and P.Q. Nguyen. Predicting lattice reduction. In *Proceedings of the theory and applications of cryptographic techniques 27th annual international conference on Advances in cryptology*, pages 31–51. Springer-Verlag, 2008.

[GN08b] Nicolas Gama and Phong Q. Nguyen. Finding short lattice vectors within mordell's inequality. In *Proceedings of STOC '08*, pages 207–216. ACM, May 2008.

[GPV08] C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *Proceedings of the 40th annual ACM Symposium on Theory of Computing*, pages 197–206. ACM, 2008.

[Hel85] Bettina Helfrich. Algorithms to construct Minkowski reduced and Hermite reduced lattice bases. *Theoretical Computer Science*, 41(2–3):125–139, December 1985.

[Hor96] A.G. Horvath. On the DirichletVoronoi cell of unimodular lattices. *Geometriae Dedicata*, 63(2):183–191, 1996.

[HPS98] J. Hoffstein, J. Pipher, and J.H. Silverman. NTRU: A ring-based public key cryptosystem. *Algorithmic number theory*, page 267, 1998.

[HR06] Ishay Haviv and Oded Regev. Hardness of the covering radius problem on lattices. In *Proceedings of CCC '06*, pages 145–158. IEEE, July 2006.

[HR07] Ishay Haviv and Oded Regev. Tensor-based hardness of the shortest vector problem to within almost polynomial factors. In *Proceedings of STOC '07*, pages 469–477. ACM, June 2007.

[HS07] Guillaume Hanrot and Damien Stehlé. Improved analysis of kannan's shortest lattice vector algorithm. In *Proceedings of CRYPTO '07*, volume 4622 of *LNCS*, pages 170–186. Springer, August 2007.

[JS98] Antoine Joux and Jacques Stern. Lattice reduction: A toolbox for the cryptanalyst. *Journal of Cryptology*, 11(3):161–185, 1998.

[Kan87a] R. Kannan. Algorithmic geometry of numbers. *Annual review of computer science*, 2(1):231–267, 1987.

[Kan87b] Ravi Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of operation research*, 12(3):415–440, August 1987. Prelim. version in STOC 1983.

[Kho05] Subhash Khot. Hardness of approximating the shortest vector problem in lattices. *Journal of the ACM*, 52(5):789–808, September 2005. Preliminary version in FOCS 2004.

[KL78] G.A. Kabatiansky and V.I. Levenshtein. On Bounds for Packings on a Sphere and in Space. *Problemy peredachi informatsii*, 14(1):3–25, 1978.

[Kle00] P. Klein. Finding the closest lattice vector when it's unusually close. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 937–941. Society for Industrial and Applied Mathematics, 2000.

[Len83] Hendrik W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8(4):538–548, November 1983.

[LLL82] Arjen K. Lenstra, Hendrik W. Lenstra, Jr., and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:513–534, 1982.

[LM85] Susan Landau and Gary Lee Miller. Solvability by radicals is in polynomial time. *Journal of Computer and System Sciences*, 30(2):179–208, April 1985. Preliminary version in STOC 1983.

[MG02] Daniele Micciancio and Shafi Goldwasser. *Complexity of Lattice Problems: a cryptographic perspective*, volume 671 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, Massachusetts, March 2002.

[Mic01a] Daniele Micciancio. The hardness of the closest vector problem with preprocessing. *IEEE Transactions on Information Theory*, 47(3):1212–1215, March 2001.

[Mic01b] Daniele Micciancio. The shortest vector problem is NP-hard to approximate to within some constant. *SIAM Journal on Computing*, 30(6):2008–2035, March 2001. Prelim. version in FOCS 1998.

[Mic04] Daniele Micciancio. Almost perfect lattices, the covering radius problem, and applications to Ajtai's connection factor. *SIAM Journal on Computing*, 34(1):118–169, 2004. Preliminary version in STOC 2002.

[Mic07] D. Micciancio. Generalized compact knapsacks, cyclic lattices, and efficient one-way functions. *Computational Complexity*, 16(4):365–411, 2007.

[Mic08] Daniele Micciancio. Efficient reductions among lattice problems. In *Proceedings of SODA 2008*, pages 84–93. ACM/SIAM, January 2008.

[MR07] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on Gaussian measure. *SIAM Journal on Computing*, 37(1):267–302, 2007. Preliminary version in FOCS 2004.

[MV10a] D. Micciancio and P. Voulgaris. A deterministic single exponential time algorithm for most lattice problems based on voronoi cell computations. In *Proceedings of the 42nd ACM symposium on Theory of computing*, pages 351–358. ACM, 2010.

[MV10b] Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *Proceedings of SODA 2010*. ACM/SIAM, January 2010.

[NS01] Phong Nguyen and Jacques Stern. The two faces of lattices in cryptology. In *Proceedings of CaLC '01*, volume 2146 of *LNCS*, pages 146–180. Springer, March 2001.

[NV08] Phong Nguyen and Thomas Vidick. Sieve algorithms for the shortest vector problem are practical. *J. of Mathematical Cryptology*, 2(2):181–207, jul 2008.

[Odl89] Andrew M. Odlyzko. The rise and fall of knapsack cryptosystems. In Carl Pomerance, editor, *Cryptology and computational number theory*, volume 42 of *Procedings of Symposia in Applied Mathematics*, pages 75–88, Boulder, Colorado, 1989. AMS.

[Poh81] M. Pohst. On the computation of lattice vectors of minimal length, successive minima and reduced bases with applications. *ACM SIGSAM Bulletin*, 15(1):37–44, 1981.

[PS08] X. Pujol and D. Stehlé. Rigorous and efficient short lattice vectors enumeration. *Advances in Cryptology-ASIACRYPT 2008*, pages 390–405, 2008.

[PS09] Xavier Pujol and Damien Stehlé. Solving the shortest lattice vector problem in time $2^{2.465n}$. *Cryptology ePrint Archive, Report 2009/605*, 2009.

[PW08] C. Peikert and B. Waters. Lossy trapdoor functions and their applications. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 187–196. ACM, 2008.

[Reg04a] O. Regev. New lattice-based cryptographic constructions. *Journal of the ACM (JACM)*, 51(6):899–942, 2004.

[Reg04b] Oded Regev. Improved inapproximability of lattice and coding problems with preprocessing. *IEEE Transactions on Information Theory*, 50(9):2031–2037, 2004. Preliminary version in CCC 2003.

[Reg09] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.

[Sch87] Claus-Peter Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical Computer Science*, 53(2–3):201–224, August 1987.

[Sch88] Claus-Peter Schnorr. A more efficient algorithm for lattice basis reduction. *Journal of Algorithms*, 9(1):47–62, March 1988.

[Sch06] Claus Peter Schnorr. Fast LLL-type lattice reduction. *Information and Computation*, 204(1):1–25, January 2006.

[SE94] Claus-Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical programming*, 66(1-3):181–199, August 1994. Preliminary version in FCT 1991.

[SFS09] Naftali Sommer, Meir Feder, and Ofir Shalvi. Finding the closest lattice point by iterative slicing. *SIAM J. Discrete Math.*, 23(2):715–731, April 2009.

[Sho03] V. Shoup. NTL: A library for doing number theory, 2003.

[SSV09] Mathieu Dutour Sikirić, Achill Schürmann, and Frank Vallentin. Complexity and algorithms for computing Voronoi cells of lattices. *Mathematics of Computation*, 78(267):1713–1731, July 2009.

[VB96] Emanuele Viterbo and Enzo Biglieri. Computing the Voronoi cell of a lattice: the diamond-cutting algorithm. *IEEE Trans. on Information Theory*, 42(1):161–171, January 1996.

[vEB81] Peter van Emde Boas. Another NP-complete problem and the complexity of computing short vectors in a lattice. Technical Report 81-04, Mathematische Instituut, University of Amsterdam, 1981. Available on-line at URL `http://turing.wins.uva.nl/~peter/`.

[WLTB] X. Wang, M. Liu, C. Tian, and J. Bi. Improved Nguyen-Vidick Heuristic Sieve Algorithm for Shortest Vector Problem. Technical report, Cryptology ePrint Archive, 2010. http://eprint. iacr. org/2010/647.