Hard Mathematical Problems in Cryptography and Coding Theory

Srimathi Varadharajan

Thesis for the degree of Philosophiae Doctor (PhD)
University of Bergen, Norway
2020

UNIVERSITY OF BERGEN

Hard Mathematical Problems in Cryptography and Coding Theory

Srimathi Varadharajan

Thesis for the degree of Philosophiae Doctor (PhD)
at the University of Bergen

Date of defense: 28.04.2020

Year: 2020

Title: Hard Mathematical Problems in Cryptography and Coding Theory

Name: Srimathi Varadharajan

Print: Skipnes Kommunikasjon / University of Bergen

Abstract

In this thesis, we are concerned with certain interesting computationally hard problems and the complexities of their associated algorithms. All of these problems share a common feature in that they all arise from, or have applications to, cryptography, or the theory of error correcting codes. Each chapter in the thesis is based on a stand-alone paper which attacks a particular hard problem. The problems and the techniques employed in attacking them are described in detail.

The first problem concerns *integer factorization*: given a positive integer N, the problem is to find the unique prime factors of N. This problem, which was historically of only academic interest to number theorists, has in recent decades assumed a central importance in public-key cryptography. We propose a method for factorizing a given integer using a graph-theoretic algorithm employing *Binary Decision Diagrams* (BDD).

The second problem that we consider is related to the classification of certain naturally arising classes of error correcting codes, called *self-dual additive codes* over the finite field of four elements, $GF(4)$. We address the problem of classifying self-dual additive codes, determining their *weight enumerators*, and computing their *minimum distance*. There is a natural relation between self-dual additive codes over $GF(4)$ and graphs via isotropic systems. Utilizing the properties of the corresponding graphs, and again employing Binary Decision Diagrams (BDD) to compute the weight enumerators, we can obtain a theoretical speed up of the previously developed algorithm for the classification of these codes.

The third problem that we investigate deals with one of the central issues in cryptography, which has historical origins in the theory of geometry of numbers, namely *the shortest vector problem* in lattices. One method which is used both in theory and practice to solve the shortest vector problem is by enumeration algorithms. Lattice enumeration is an exhaustive search whose goal is to find the shortest vector given a lattice basis as input. In our work, we focus on speeding up the lattice enumeration algorithm, and we propose two new ideas to this end. The shortest vector in a lattice can be written as $\boldsymbol{s} = v_1\boldsymbol{b}_1 + v_2\boldsymbol{b}_2 + \ldots + v_n\boldsymbol{b}_n$, where $v_i \in \mathbb{Z}$ are integer coefficients and $\boldsymbol{b}_i$ are the lattice basis vectors. We propose an enumeration algorithm, called *hybrid enumeration*, which is a greedy approach for computing a short interval of possible integer values for the coefficients v_i of a shortest lattice vector. Second, we provide an algorithm for *estimating the signs* ($+$ or $-$) of the coefficients $v_1, v_2, \ldots, v_n$ of a shortest vector $\boldsymbol{s} = \sum_{i=1}^{n} v_i\boldsymbol{b}_i$. Both of these algorithms results in a reduction in the number of nodes in the search tree.

Finally, the fourth problem that we deal with arises in the arithmetic of the *class groups* of imaginary quadratic fields. We follow the results of Soleng and Gillibert pertaining to the class numbers of some sequence of imaginary quadratic fields arising in the arithmetic of *elliptic and hyperelliptic curves* and compute a

bound on the effective estimates for the orders of class groups of a family of imaginary quadratic number fields. That is, suppose $f(n)$ is a sequence of positive numbers tending to infinity. Given any positive real number L, an *effective estimate* is to find the smallest positive integer $N = N(L)$ depending on L such that $f(n) > L$ for all $n > N$.

In other words, given a constant $M > 0$, we find a value N such that the order of the ideal class I_n in the ring R_n (provided by the homomorphism in Soleng's paper) is greater than M for any $n > N$.

In summary, in this thesis we attack some hard problems in computer science arising from arithmetic, geometry of numbers, and coding theory, which have applications in the mathematical foundations of cryptography and error correcting codes.

Contents

Chapter 1

Introduction

1.1 Hard problems and algorithms

In the following, we introduce some basic notions on computational problems, algorithm, and their complexity; see [5] for a more detailed treatment of the topic.

The most basic construct, in theoretical computer science is the concept of an *algorithm*. By definition, an algorithm is a *finite* sequence of operations carried out in a particular order to solve a given problem. The keyword here is *finite*, implying that an algorithm must eventually terminate, however long its running time may be.

In computer science, a problem is said to be *solvable* if one can find an algorithm which produces the desired output for a given set of inputs; here, intuitively, the inputs represent the statement of a concrete instance of the problem, while the outputs represent its solution. The inputs for a given algorithm could consist of numerical or non-numerical parameters like numbers, data structures or even other algorithms.

We are interested in solving problems efficiently, and how efficiently a problem can be solved depends on the algorithm that is used to solve it. Clearly, there can be several algorithms which solve a given problem, and this makes it necessary to compare the relative efficiencies of all algorithms designed to solve the same problem. For example, there are many algorithms for sorting a given list of elements: Bubble sort [122], merge sort [122], quick sort [122], etc. All of these exhibit a different efficiency, i.e. a different worst-case, best-case and average-case computational complexity. These considerations lead to the theory of algorithmic *complexity*, wherein algorithms are classified into *complexity classes* in accordance with the amount of resources that they consume (usually by representing a resource parameter, like running time or storage space, as a function of the length of the input).

All these intuitive notions of inputs, efficiency and complexity need to be formalized in order to be applied to the analysis of algorithms. One way to handle these concepts in a rigorous and uniform manner is by means of so-called *Turing machines*.

The Turing machine is a conceptual device which models the process of computation. Informally speaking, a Turing machine consists of an infinite tape of cells (from which the Turing machine can read and onto which it can write), an initial state, a set of intermediate states, and a set of final states. Prior to the computation, the input is placed on the tape. At any given moment, the Turing machine is in

some particular state; it begins in the initial state, proceeds to different states in the course of computation, and, finally, terminates once it reaches one of the final states. At any moment, the Turing machine is processing one particular cell of the tape. Which state it proceeds to, how it modifies the contents of the cell, and which cell it moves to next, depends on the contents of the cell that is being processed and the state that the Turing machine is in. Once computation has finished, the output is what is written on the tape.

Any algorithm can be implemented on a Turing machine by constructing an appropriate set of rules dictating how the Turing machine acts on different combinations of states and symbols. Let n be an integer and $C(n)$ be the maximum number of moves made by the Turing machine before coming to a halt on any input of length n. Then $C(n)$ is a measure of the *complexity* of the algorithm as implemented on the Turing machine.

The number of moves $C(n)$ performed by the Turing machine is a measure of the *time complexity* of the algorithm. However, there can be other measures of complexity, such as the *space complexity* which, intuitively, measures how much memory the implementation of an algorithm will consume; in the case of a Turing machine, the space complexity is measured by the maximum number of cells $S(n)$ on the tape that are utilized during the computation for any input of length n.

Of course, the ultimate goal of the framework introduced above is to classify different algorithms into *complexity classes* based on how efficient they are with respect to time complexity or space complexity. Since the magnitude of $C(n)$ and $S(n)$ is most important for large values of n, when analyzing the efficiency of an algorithm, it is important to consider the asymptotic behaviour of these complexity functions. The following notion allows us to formally compare the asymptotic behavior of two functions.

Definition 1.1.1. Let $f, g : \mathbb{N}^+ \to \mathbb{R}^+$ be two functions. We say $g(n) = \mathcal{O}(f(n))$ if there exists some $c \in \mathbb{R}^+$ and a positive integer m such that for all $n \geq m$, $g(n) \leq c \cdot f(n)$, i.e. $c \cdot f(n)$ bounds $g(n)$ from above as n tends to infinity.

We frequently encounter algorithms whose complexity can be expressed as a polynomial in n. Since this so-called *Big-O* notation measures asymptotic growth, one only needs to consider the fastest growing term of some given polynomial $g(n)$. For example, if $g(n) = 5n^2 + \log(n)$ then $g(n) = \mathcal{O}(n^2)$, because as n approaches infinity, $\log(n)$ becomes negligible compared to $5n^2$.

The generally accepted convention is that an algorithm is "efficient" if its time complexity is polynomial in the length of its input. We say that an algorithm is in the class **P** (*Polynomial*) if it is solvable in time $\mathcal{O}(n^k)$, for some constant k. That is, **P** is the complexity class containing decision problems which can be solved by a Turing machine using polynomially many computational steps.

The class **NP** (*Non-deterministic Polynomial*) is the set of decision problems solvable in polynomial time on a non-deterministic Turing machine, i.e. an unbounded number of Turing machines running in parallel. This implies that the class **NP** consists of problems for which solutions might be hard to find, but where any given solution can be *verified* by a deterministic Turing machine in polynomial time. Every problem in this class can still be solved in exponential time using exhaustive search, also known as brute-force. Examples include the Boolean satisfiability problem (SAT), the Vertex cover problem, etc.

The most difficult problems in **NP** form the so-called class of **NP-complete** problems. If one were to find an efficient algorithm to solve any given **NP**-complete problem, then that algorithm could be used to efficiently solve *all* problems in **NP**, which would imply $\mathbf{P} = \mathbf{NP}$. At present, all known algorithms for **NP**-complete problems require time that is super-polynomial in the input size. To solve an **NP**-complete problem in practice for inputs of a nontrivial size, generally one of the following approaches is used: approximation, probabilistic computing, heuristic techniques, etc. Examples of **NP**-complete problems include: the Boolean satisfiability problem (SAT), the Knapsack problem, the Traveling salesman problem, the Subgraph isomorphism problem, the Subset sum problem, the Clique problem, the Minimum vertex cover problem, and the Graph coloring problem.

There is ample evidence that seems to indicate that a polynomial-time algorithm for solving an **NP**-complete problem will never be found. However, a formal proof that $\mathbf{P} \neq \mathbf{NP}$ has thus far eluded mathematicians, and indeed stands as one of the great open problems of the century.

1.2 Cryptography and hard problems

1.2.1 A brief history of cryptography

The art of cryptography dates back several thousand years, to a time when kings and queens had to rely on secret communication to get their messages across while ensuring that they would not fall into the wrong hands. As such, they developed techniques to disguise messages so that only the intended recipient could read them. *Cryptology* is the art and science of transmitting secret messages. It includes both the areas of *cryptography*, which is the art of hiding the meaning of the message by a process known as *encryption*; and *cryptanalysis*, which is the art of unscrambling the encrypted message without knowledge of the secret key.

There are several ways of achieving secrecy. Early secret communication usually consisted of simply finding clever ways of hiding messages, known as *steganography* [118] (from the Greek words *steganos*, meaning "covered", and *graphein*, meaning "to write"). *Cryptography*, derived from the Greek word *kryptos* meaning "hidden", evolved in parallel with the development of steganography.

The history of cryptography is a fascinating subject. As mentioned above, the aim of cryptography is not to hide the existence of the message, but rather its meaning. The message is scrambled according to a particular protocol agreed upon beforehand by the communicating parties. This protocol usually involves the sharing of some secret information (the *key*) needed to unscramble the message. The advantage of cryptography is that even if the enemy intercepts the encrypted message, he will find it extremely difficult or impossible to retrieve the original message from the encrypted text without knowing the key.

Classically, encryption was divided into two branches, known as *transposition* and *substitution*. In transposition, the letters of the message are simply rearranged. It is clear that for short messages, this method will be insecure, because there are only a limited number of ways one can rearrange a set of letters. A random transposition could still offer a high level of security for longer messages, but the drawback was that it was harder for the intended recipient to unscramble them as well. Transposition techniques are known to have been used as early as the 5th century B.C. for military

communication. Called *Scytales* [117], these were objects around which one would wrap a strip of parchment, resulting in an intelligible message if and only if the Scytale was of the correct diameter. Thus, one could say that the *diameter* of the Scytale served as the key to the cipher, though as with most early methods of encryption, the security of the message crucially relied on the encryption method being kept secret.

On the other hand, substitution is a method by which each letter of the alphabet is replaced by a different letter, or symbol. The earliest known use of a substitution cipher appears in Julius Caesar's "Gallic Wars" [117]. Julius Caesar encrypted the message by shifting each letter of the alphabet by 3 positions: for instance, the letter "a" is replaced with "D", the letter "b" is replaced with "E", and so on. Such simple substitution ciphers are called *shift ciphers*, and they provided some basic security in a time when cryptography was unknown to most people. But as these ciphers rose in popularity, people quickly figured out how to break them, and cryptographers had to invent better ways to secure their messages. However, even substituting every single letter of the alphabet with a different symbol (which need not even be a letter of the alphabet itself) turned out to be insecure: clever cryptanalysts realized that the different letters of a given language show up in written form with a particular frequency, and that even with a rather small sample, the cipher can usually be broken simply by matching the symbol frequencies of the text with the known frequencies of the language. This is known as *frequency analysis* [91].

Substitution ciphers continued to evolve throughout the years, with new methods, such as the Vigenère cipher, long thought unbreakable. Even though its simplicity meant that anyone could easily use it, it withstood three centuries of attacks, before an algorithm for breaking it was finally discovered in 1863. The Vigenère cipher operates as follows: pick an arbitrary codeword, say "CAT", which will serve as the key to the cipher. The message is then encrypted by applying a separate shift cipher to each letter of the message. Since the first letter of the codeword in this example is "C", and the letter "C " is two positions in front of the letter "A", the first letter of the message is shifted by two positions; since the second letter in the key is "A", the second letter of the message remains unchanged; the third letter of the key is "T", and so the third letter of the message is shifted by 19 positions. If there are more letters in the message than the length of the codeword, the codeword is simply repeated, and thus the fourth letter is shifted by two positions, the fifth letter remains the same, and so forth.

At the beginning of the 20th century, as communication was revolutionized by the introduction of the telegram, the need for secrecy across open channels became more important than ever before. But with the dawn of electrical machines, cryptographers also saw opportunities for unprecedented levels of secrecy. The most famous example of such a cryptographic device is the *Enigma* [51, 117], used by Nazi Germany during the Second World War. These were devices with keyboards, that through intricate wiring, cogwheels, and a combination of short term and long term key settings, automatically encrypted each letter of a message that the operator would type on the keyboard. On the receiving end, another machine with the correct settings would be used to decrypt the ciphertext back into plaintext. Even if an enemy agent were to intercept a given encrypted message, it was assumed to be impossible for him to learn anything about its contents unless he had a working copy of the machine and knew the exact settings that were used when encrypting

the message.

1.2.2 Cryptography based on hard problems

As history had shown, any cryptographic technique tended to be broken sooner or later. After the Second World War, cryptographers started to ask the question of what exactly made an encryption algorithm secure, and whether its security could be proven.

The only encryption scheme now known to be perfectly secure is called the *One-Time Pad* [91]. Here, the key is a set of random bits at least as long as the message. Encryption is then achieved by XOR-ing the bits of the key with the bits of the message; decryption is performed in the same way, i.e. by XOR-ing the bits of the key with those of the ciphertext. Given that the key, or "pad", is used only once, the One-Time Pad was proven by Claude Shannon in 1945 to achieve *perfect secrecy* [24]. The requirement that the key should only be used once is crucial, and reusing the same key, or even parts of it, already leads to an encryption which can be broken. Furthermore, this encryption may actually be weaker than that provided by other encryption schemes that do not possess perfect secrecy.

All historical examples of encryption and decryption schemes thus far have been what we now call *symmetric* schemes. The encryption key has to be shared between the sender and receiver prior to communication, and the same key is used both for encryption and decryption, meaning that neither can be achieved without access to it. How to share the key, however, has always been a major obstacle to efficiently achieving secure communication, requiring the establishment of a separate secure channel (such as a trusted mail courier). The seminal 1976 paper by Diffie and Hellman [30] finally proposed a solution to these problems by showing that secure key sharing can be achieved across insecure channels by means of a new type of encryption scheme that relies on the computational hardness of certain mathematical problems. The original problem described by Diffie and Hellman is the following: let G be a finite cyclic group of order N and g a generator of G. Exponentiation, that is, computing g^n given an integer n, is easy and can be efficiently done with $O(\log n)$ multiplications using the square and multiply algorithm in any cyclic group. On the other hand, given an element g^n, computing n (the so-called *discrete logarithm* of g^n) might vary from easy to more difficult depending upon the selection of the group G [55].

In the same paper, Diffie and Hellman introduced the concept of a new type of cryptosystem called a *public-key cryptosystem* based on the discrete logarithm problem. Here, each user has a public key pk, corresponding to an encryption scheme E_{pk}, and a secret key sk corresponding to a decryption scheme D_{sk}. Unlike all previously known cryptographic schemes which are symmetric (in the sense that the same key is used for both encryption and decryption), in a public-key cryptosystem one only needs to know the public key pk in order to encrypt, while the private key sk is needed in order to decrypt. The assumption is that the secret key should be computationally hard to derive from the public key. Thus, the public keys of multiple users may be collected in a public directory so that anyone may encrypt a message using another user's public key, without the need for any prior sharing of keys. Once the message is encrypted, however, the only one who can decrypt is the one who possesses the secret key.

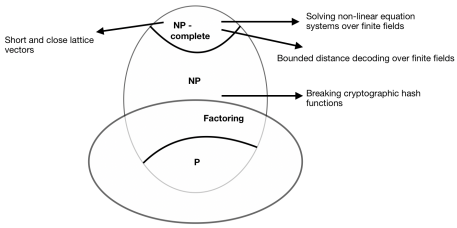


Figure 1.1: Complexity classes and some hard problems

Diffie and Hellman also showed how any such public key encryption scheme could be adapted into an unforgeable, yet publicly verifiable, *signature scheme* [30]. Here, a user may sign a message using their secret key, and any user wanting to verify that the message indeed originates from the purported source, may "decrypt" it using their public key.

Soon afterwards, Rivest, Shamir, and Adelman in 1977 invented another public-key cryptosystem, named RSA after the initials of its inventors [100]. Their cryptosystem is still widely used today.

Figure 1.1 shows the complexity of some hard problems. Other computationally hard problems which have been used to design public-key cryptosystems include solving non-linear equations over finite fields (leading to multivariate cryptography), finding short and close vectors of lattices (lattice-based cryptography), decoding general linear codes (code-based cryptography).

In 1981, the celebrated phycisist Richard Feynman suggested the idea of a *quantum computer*, which he posited might be necessary in order to properly simulate quantum physics [32]. In 1985, David Deutsch [28] gave the first example of a simple combinatorial problem whose solution would have a significant speed-up on such a computer.

In 1994, Peter Shor published a quantum algorithm [113] which would be able to factor large numbers in polynomial time. If a quantum computer with a sufficient number of qubits can be built, Shor's algorithm could be used to break the widely used RSA public key encryption scheme. Furthermore, the same principle could be used to break the discrete logarithm problem and therefore the Diffie-Hellmann key exchange protocol. Until this point, quantum computers had been viewed as a highly hypothetical curiosity of limited practical interest. Now, however, it was clear that if it were possible to build such a computer, a number of important and widely used cryptosystems would be vulnerable to attack. This kickstarted a study of quantum computing, and the first prototypes of such computers were built already in 1995. Improvements have occurred and larger quantum computers are now being built in several facilities around the world.

Meanwhile, cryptographers have been busy finding new problems which may withstand attacks by quantum computers, an area of research known as post-quantum cryptography. Potential candidates for cryptographic schemes that can withstand quantum attacks include multivariate cryptography, code-based cryptography, and lattice-based cryptography, the latter of which will be explored in more detail in Chapter 4.

1.3 Papers in this thesis

We now take a closer look at the papers making up this thesis. For each paper, we explain the problem being attacked, the tools used to investigate it, and the results obtained.

1.3.1 Integer factorization using Binary Decision Diagrams

In the first paper we deal with one of the oldest difficult problems in mathematics - integer factorization. The fundamental theorem of arithmetic states that any natural number can be expressed uniquely as a product of prime numbers. This theorem also exhibits the structure of the natural numbers in relation to multiplication – it shows that primes are the elements out of which all natural numbers can be built.

Finding prime factors of a given number N has been one of the most fascinating and formidable problems in mathematics, and dates back to the time of Euclid. Once the unique factorization property was established, a natural problem was to find all the prime factors of a given natural number N. Several mathematicians have developed step by step procedures to factor a given integer N, that is, to find all distinct prime factors p_is such that $N = p_1^{e_1} p_2^{e_2} \dots p_n^{e_n}$, where e_i are positive integers. An account of early and modern factorization algorithms is given in Chapter 2, Section 2.2.

Given a large integer N, there is no known efficient algorithm that can factorize N in the classical computation model. Nevertheless, there is a *quantum* algorithm, namely Shor's algorithm which was already mentioned above [113], which can factorize N in polynomial time. The fastest methods for factoring a number N in the classical model take sub-exponential time in the number of bits of N [19]. As some cryptographic schemes rely on the hardness of finding the factors of large integers, it is interesting to apply different methods and explore the possibility of factoring such numbers.

Many problems in computer science, related to the design of both software and hardware, involve representations and manipulations of Boolean functions, i.e. functions that assume either 0 or 1 as their output value. It is essential that these processes are carried out in an efficient manner. One natural method which is widely used to this end is to represent such functions by means of Binary Decision Diagrams (BDD) [59]. In our paper, we apply a particular type of BDD to the integer factorization problem. We focus primarily on integers that have only two prime factors p and q of the same bit size. Despite its simplicity, this case is of significant practical interest since a similar situation occurs in RSA. Here we give a very brief outline of the paper; please refer to Section 2.4 for more details.

The statements of many problems in digital logic circuits, especially in hardware and networking, involve switching circuits. Switching circuits exhibit binary state values ON and OFF. It is natural to represent these conditions in terms of elements of binary logic. This connection between switching theory and Boolean algebra was first formulated by Shannon in 1938 [23]. Following his methodology, any switching circuit can be completely described in terms of Boolean functions of several variables.

Although the Boolean representation of switching circuits has been the foundation on which switching theory was built, ways of compactly representing these functions in an efficient manner remains an open problem. In 1958, Lee [65] in-

troduced Binary Decision Programs, which he used to represent Boolean functions. Compared to Shannon's method, which is algebraic in nature, binary decision programs provide a new representation of switching circuits, with the advantage that they are more suitable for representing circuits with a large number of decision processes, or transfers.

Binary Decision Diagrams (BDD) were first introduced and analysed by Akers in 1978 [4]. He was motivated by the same considerations as Lee, that is, to find representations more compact than the known ones at the time. The purpose of a BDD is to represent a Boolean function using a diagram or a graph which enables one to evaluate the output value of the function for given input values.

A BDD is a directed acyclic graph which contains a root-node at the top and a true-node at the bottom. Every node has at most two outgoing directed edges labelled 0 or 1. The nodes in a BDD are arranged in horizontal *levels*, where each level corresponds to a single input variable or a linear combination of the input variables. A *path* in a BDD is a sequence of consecutive edges, where the end node of one edge is the start node for the next. A *complete* path starts in the root node and ends in a true-node. Since each edge is labelled either with 0 or 1, and each level corresponds to a linear combination of variables, every path naturally corresponds to an instantiation of the linear combinations. A complete path can be seen as giving a system of linear equations which may or may not have a solution. Finding a solution to an equation system represented by a BDD equates to finding a particular path in the BDD.

We start by building the BDD corresponding to the set of equations obtained from the binary multiplication of two integers. Having done this, we notice that the BDD duplicates the input variables. In an attempt to resolve this, we define two basic operations on BDD: *swapping levels* and *adding levels*. The former consists of exchanging, or swapping, the variables corresponding to two adjacent levels. The latter involves the addition of variables from a given level to the level directly below it. The main purpose of these operations is to remove the duplication and thereby resolve the dependencies among the variables in order to obtain a consistent system of linear equations and facilitate the solution of the original problem. This process of removing dependencies is called *linear absorption*. In the final step, we apply an operation called *reduction* to the BDD [104]. This operation removes all the nodes which have no incoming or outgoing edges, and also the ones where the 0-edge and 1-edge point to the same node. The BDD that we obtain after applying all of these operations then gives us the solution to the set of equations that we stared with.

Our contribution: [99] We model the problem of integer factorization using a system of Boolean equations which naturally arises from the multiplication of two integers represented in binary. Let $(p_{n-1}, p_{n-2}, \ldots, p_1, p_0)$ and $(q_{n-1}, q_{n-2}, \ldots, q_1, q_0)$ be the binary representations of the integers p and q, respectively. We represent the number $N = pq$ as the product $N = (p_{n-1}, p_{n-2}, \ldots, p_1, p_0) \times (q_{n-1}, q_{n-2}, \ldots, q_1, q_0)$. We treat the bits p_i and q_i in the unknown factors as variables, and we obtain equations by relating the unknown bits of p and q with the known bits of N from the multiplication operation. We then build the BDD corresponding to these equations.

The constructed BDD encodes a set of binary vectors, which are precisely the solutions to all linear systems generated by all complete paths in the BDD. To facilitate the solution of this system of equations, we apply the operations of adding levels, swapping levels, and reduction described above, and then solve the resulting

system of equations.

We show that the number of nodes in the BDD representing $N = pq$ prior to applying the operations is $\mathcal{O}(n^3)$, where n is the number of bits in one of the factors p and q. We have run experiments to determine the complexity of the proposed factorization algorithm. The best known algorithm for factoring is the GNFS, which has complexity $L_N[\frac{1}{3}, \sqrt[3]{\frac{64}{9}}]$ [70]. The observed complexity of the algorithm in our work is $\mathcal{O}(2^n)$. Asymptotically, this would be comparable only to the trial division method. Practically, the trial division is faster than the proposed method. Thus, although using BDD for factoring integers is a natural idea, we have shown that it does not lead to an improvement in computational efficiency.

1.3.2 Graphs and self-dual additive codes over $GF(4)$

Our second paper deals with the classification of the family of a particular type of codes called *self-dual additive codes* over $GF(4)$. The 1948 publication of Claude Shannon's landmark paper "A mathematical theory of communication" [111] signified the beginning of coding theory. The main theorem of this paper essentially guarantees the existence of error-correcting codes with certain good properties. This result was only existential in nature, meaning that it did not show how such codes could be constructed. This motivated later researchers to look for more and better codes which could be applied in practice.

The crux of the main problem in coding theory is to find codes with rate R and minimum distance d as large as possible. The greater the rate R, the more efficiently information can be transmitted. The greater the minimum distance d, the more errors the code can correct. Clearly, these two are in conflict, hence the goal is to find a good trade-off.

Self-dual codes are an interesting family of codes which has rich mathematical history and connections to lattices and sphere packings [120], and modular forms [78, 112]. Self-dual additive codes over $GF(4)$ under the Hermitian trace inner product (which will be defined later) became of interest because of their correspondence to additive (or stabilizer) quantum error-correcting codes [7, 20, 27, 35, 52].

Let $GF(4) = \{0, 1, \omega, \omega^2\}$ be the finite field of 4 elements. An *additive code* C over $GF(4)$ of length n is an additive subgroup of $GF(4)^n$. In other words, let C be a code over $GF(4)$ with generator matrix G. We say that C is an *additive code* if all codewords in C are $GF(2)$-linear combinations of the rows of G.

There is a natural inner product arising from the trace map. The trace map $Tr : GF(4) \to GF(2)$ is given by $Tr(x) = x + x^2$. In particular, $Tr(0) = Tr(1) = 0$ and $Tr(\omega) = Tr(\omega^2) = 1$. The *conjugate* of an element $x \in GF(4)$, denoted as $\overline{x}$ is the image of x under the Frobenius automorphism; in other words, $\overline{0} = 0$, $\overline{1} = 1$, $\overline{\omega} = \omega^2$, and $\overline{\omega^2} = \omega$.

The *Hermitian trace inner product* of two vectors $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$ is defined as

$$x * y = \sum_{i=1}^{n} Tr(x_i \overline{y_i}) = \sum_{i=1}^{n} (x_i {y_i}^2 + x_i^2 y_i).$$

If C is an additive code, its *dual*, denoted as $C^\perp$, is the additive code $\{x \in GF(4)^n \mid x * c = 0 \text{ for all } c \in C\}$. If C is an $(n, 2^k)$ code, then $C^\perp$ is an $(n, 2^{2n-k})$

code. We say that C is *trace self-orthogonal* if $C \subseteq C^{\perp}$ and we say that C is *trace self-dual* if $C = C^{\perp}$. In particular, if C is self-dual, then C is an $(n, 2^n)$ code. We remark that additive self-dual codes exist for any length n since the identity matrix I_n generates a self-dual $(n, 2^n, 1)$ code.

The *weight* of a codeword $c \in C$ is the number of nonzero components of c. The *minimum weight* d of a code C is the smallest weight of any nonzero codeword of C. Let C be an $[n, k, d]$ code. For $i \in \{0, 1, 2, \ldots, n\}$, let A_i denote the number of code words in C of weight i. The collection $\{A_i : i \in \{0, 1, 2, \ldots, n\}$ is called the *weight distribution* of the code C. The *weight enumerator* of C is defined to be the homogeneous polynomial $W_C(x, y) = A_0 x^n + A_1 x^{n-1} y + A_2 x^{n-2} y^2 + \ldots + A^n y^n$.

Given the weight distribution of a code, it is easy to determine its *minimum distance*. It is hard to approximate the minimum distance of a given code, so the computation of the weight enumerator is also a hard problem [127].

Two self-dual additive codes C_1 and C_2 over $GF(4)$ are equivalent if and only if the codewords of C_1 can be mapped onto the codewords of C_2 by a map that preserves self-duality. Such a map consists of a permutation of coordinates, followed by a scaling of coordinates by the elements of $GF(4)$, followed by conjugation of some of the coordinates. Two equivalent self-dual codes have the same weight enumerator.

All additive self-dual codes over $GF(4)$ of length $n \leq 5$ have previously been classified up to equivalence by Calderbank et al. [20]. For $n \leq 7$, the classification was done by Höhn [52] and Hein et al. [46], and for $n \leq 9$, it was done by Glynn et al. [40]. Danielsen and Parker [27] classified all self-dual additive codes over $GF(4)$ of length $n \leq 12$ by using a graph based algorithm for general graphs.

Our contribution: [62] We follow the classification of Danielsen and Parker. We study self-dual additive codes over $GF(4)$ and give a method to classify them by using a graph based algorithm for graphs having a fixed value of a property called rankwidth (which we will define later in Section 3.3.1). This leads to a significantly faster method for classifying the codes corresponding to these graphs. Even though this method is applicable only to a subset of codes, the branching strategy for classifying graphs corresponding to self-dual additive codes on n vertices takes at most $3(n-1)$ ways for graphs having rankwidth 1. For comparison, in Danielsen and Parker's work [27], the branching is done in $2^{n-1} - 1$ ways. It is proposed to determine for how large values of n classification can still be performed in practice by conducting experiments in the future.

There are two computationally heavy steps in the classification algorithm: testing graph isomorphism and weight enumeration. For a fixed k, testing graph isomorphism for graphs of rankwidth k is polynomial in the size of the graph [42], and it is in fact linear for graphs of rankwidth 1 [124]. Hence, looking at the problem of classification of such codes in terms of rankwidth may have additional advantages.

Another important step in the classification algorithm from [27] was computing weight-enumerators for a given code. The algorithm given in [27] for computing the weight-enumerator of a linear $[n, k]$ code is essentially a brute-force search with complexity $\mathcal{O}(2^k)$. If $k > n/2$, the weight enumeration of the dual code is computed instead for the sake of efficiency. We use Binary Decision Diagrams (BDD) to compute the weight-enumerators instead of performing a brute-force search. The algorithm using BDD for weight enumeration has complexity similar to brute force, but has the benefit that we automatically get complexity $\mathcal{O}(2^{\min\{k, n-k\}})$ without needing to consider the dual code.

We show that the minimum distance of a code is at least 4if and only if the corresponding graph does not contain any pendent vertex or any twin-pairs (both of these terms are defined later).

1.3.3 Reducing lattice enumeration search trees

With the current interest in post-quantum cryptography, lattice based cryptographic primitives are among the most promising candidates for achieving secure and efficient post-quantum encryption. Given two positive integers m and n with $m \geq n$, the *lattice* generated by a set $\{\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n\}$ of n linearly independent vectors in $\mathbb{R}^m$ is the set $\{\sum_{i=1}^{n} x_i \boldsymbol{b}_i : x_i \in \mathbb{Z}\}$ of integer combinations of the vectors $\boldsymbol{b}_i$. One of the most basic computational problems concerning lattices is the *shortest vector problem* (SVP): given a lattice basis as an input, find a nonzero lattice vector of smallest norm.

The first step towards attempting to solve the SVP is through *lattice reduction*, the goal of which is to transform the input basis into one containing very short and nearly orthogonal vectors. This is closely related to the reduction theory of quadratic forms developed by Lagrange [64], Gauss [38] and Hermite [50]. Lattice reduction algorithms have numerous practical applications, notably public-key cryptanalysis (for instance, breaking knapsack cryptosystems [88], or special cases of RSA and DSA [86, 81]).

Algorithms for solving the SVP fall into two general categories:

- **exact algorithms**: these provably find a shortest vector, but their complexity is exponential in the dimension of the lattice. Exact algorithms fall into two sub-categories:

  - **polynomial-space exact algorithms**: these are based on enumeration which, in its simplest form, is an exhaustive search for finding the best integer combinations of the basis vectors such that the resulting vector is the shortest vector.

    The standard enumeration technique has a worst-case complexity of $2^{\mathcal{O}(n^2)}$, where n is the dimension of the lattice. In the 1980s Fincke, Pohst and Kannan studied how to improve the complexity of the standard enumeration algorithm for solving SVP [33, 56, 34]. These algorithms are deterministic and are based on an exhaustive search of lattice points within a small convex set. In general, the running time of an enumeration algorithm heavily depends on the quality of the input basis. So, suitably pre-processing the input lattice using a basis reduction algorithm is essential before applying a lattice enumeration method.

    In the 90s Schnorr, Euchner and Hörner introduced the *pruning* technique, by which these algorithms obtained substantial speedups [107, 108]. The rough idea is to prune away sub-trees where the probability of finding the desired short lattice vectors is small. This restricts the exhaustive search to a subset of all solutions. Although there is a chance of missing the desired vector, the probability of this is small compared to the reduction in running time.

    The pruning strategy was later studied more rigorously by Gama, Nguyen and Regev [37], introducing what they called *extreme pruning*. In extreme

pruning, very large parts of the search tree are cut away. This makes the search very fast, but the probability of finding the shortest vector on a given run is very small. However, the authors showed that the search tree is reduced by a larger factor than the reduction in the probability of finding the shortest vector, so one obtains a speed-up by just permuting the basis and repeating the process a number of times. The performance of pruned enumeration leads asymptotically to an exponential speed up of about $2^{n/2}$. The algorithm using extreme pruning is the fastest known and today's state of the art when it comes to enumeration.

- **exponential-space exact algorithms**: these have a better running time than the polynomial-space exact algorithms, but have exponential space complexity. The first such algorithm is the randomized sieve algorithm of Ajtai, Kumar and Sivakumar (AKS) [3], with a worst-case time complexity of $2^{\mathcal{O}(n)}$. An alternative deterministic algorithm was later presented by Micciancio and Voulgaris [84] with time complexity $2^{2n+\mathcal{O}(n)}$. It is worth noting that heuristics can be used to improve the running time of the AKS algorithm to e.g. $2^{0.3836n}$ [130].

- **approximation algorithms**: these are much faster than the exact algorithms, but do not guarantee that their output vector is indeed a shortest one. Their output is typically an entire reduced basis, and thus they are essentially lattice reduction algorithms. The best known lattice reduction algorithms are the famous LLL algorithm [69] and the BKZ algorithm [87]. Both of these algorithms work by applying successive transformations to the input basis in an attempt to make the basis vectors as short and as orthogonal as possible.

Our contribution: [61] We propose two new ideas, and show their potential in speeding up lattice enumeration. First, we propose a new enumeration algorithm called *hybrid enumeration*. Second, we provide an algorithm for estimating the signs (+ or −) of the coefficients $v_1, v_2, \ldots, v_n$ in the lattice element $\sum_{i=1}^{n} v_i \boldsymbol{b}_i$.

One disadvantage of the standard enumeration technique is that the algorithm depends on the computed Gram-Schmidt (GS) orthogonal basis for computing the intervals in which the coefficients v_i giving a short vector can be found. Once the GS orthogonal basis is computed, it fixes the order of the coefficients to be guessed.

In our work, the hybrid enumeration takes a greedy approach, where the basis vectors are not bound by any particular order and we are free to choose which of the coefficients v_i that we have not tried before to branch for at any given point in the search tree. We show that dynamically changing the order of the coefficients v_i that we guess lowers the number of nodes in the search tree as compared to the standard enumeration algorithm.

The price to pay for this flexibility is increased work at each node of the search tree. Hence, the actual time taken to enumerate a lattice using the new method may be longer than the time taken by the standard GS enumeration. Therefore, we propose to only use the new enumeration technique at the nodes on the highest levels of the search tree, and then switch to standard enumeration for the remaining levels. This still leads to a reduction in the number of nodes in comparison with the standard enumeration method. The magnitude of the reduction depends on the type of lattice and the level where we switch to standard GS enumeration.

The second technique we provide is to estimate the signs of each coefficient v_i. The main idea behind the algorithm is to exploit the inner product function which contains information about the length and angle between the basis vectors. Given two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, if the angle between them is less than 90 degrees, then their sum $\boldsymbol{a}+\boldsymbol{b}$ is longer than both $\boldsymbol{a}$ and $\boldsymbol{b}$, and their difference $\boldsymbol{a}-\boldsymbol{b}$ is shorter than at least one of $\boldsymbol{a}$ and $\boldsymbol{b}$. To get a short vector we need to subtract one from another, which implies that the signs of these vectors should be opposite to one another. Similarly, when the angle between them is more than 90 degrees, then addition gives a short vector, so their relative signs should be the same.

We generalize this observation to n vectors, developing a method for estimating the signs of each v_i together with a confidence measure for each estimate. We then give a pruning strategy where the interval computed for each v_i is cut down using the estimate of the sign and the confidence factor. Unlike other pruning methods, this leads to a one-sided pruning where we only cut away a portion of possible values for v_i (values where the sign is believed to be wrong).

We ran experiments to compare the efficiency of hybrid enumeration to that of standard enumeration. Through our experiments, we observed that applying hybrid enumeration leads to a reduction in the number of nodes, but the reduction is asymptotically negligible and hence the algorithm has the same complexity as that of standard enumeration. When comparing hybrid enumeration with extreme pruning, we let enumeration using extreme pruning run for the same amount of time that hybrid enumeration uses to exhaustively search for the shortest vectors. What we observed is that in most cases extreme pruning will miss some of the short vectors that are found by hybrid enumeration. If it is important to find *all* vectors of length within some bound (we used the bound from [105] in the experiments), then it is faster to use hybrid enumeration.

We ran experiments for sign-based pruning on top of both standard enumeration and extreme pruning. In the case of running sign-based pruning on the top of standard enumeration, we observed a modest reduction in the number of nodes in the search tree and a reduction in the running time. However, we never failed to find the shortest vector using sign-based pruning. In the case of running sign-based pruning on top of extreme pruning, our method did not show any improvement over ordinary extreme pruning. The number of nodes in the search tree is indeed reduced, but it takes longer to find short vectors compared to the usual extreme pruning technique.

1.3.4 Applications of elliptic and hyperelliptic curves - effective estimates of class numbers

The arithmetic of quadratic fields, i.e. degree two extensions of $\mathbb{Q}$, has interesting applications in cryptography. It is well known that many public key cryptosystems are based on intractable computational problems in number theory like integer factorization, discrete logarithms, etc. A number of problems involving the structure of the class groups of these fields are believed to be intractable. Consequently, corresponding cryptographic implementations built on these problems could be considered secure given large enough parameters.

Instances of such problems include key exchange protocols using imaginary quadratic fields [18] and real quadratic number fields [103], the NICE (New Ideal Coset En-

cryption) cryptosystem based on the hidden kernel problem [92], one way functions based on ideal arithmetic in number fields and the Diffie-Hellman problem [16]. It is also pertinent to mention the discrete logarithm problem for class groups of imaginary quadratic fields, for which no efficient algorithm is known [17].

All of these problems involve the computation of the class numbers of imaginary quadratic extensions. The difficulty lies in the hardness of factoring the discriminant which in turn implies that computing the orders of the elements in the class group of imaginary quadratic fields is difficult if factoring the discriminant is hard.

Our contribution: [125] In this paper we consider the results of Soleng [121] and Gillibert [39] regarding certain families of imaginary quadratic extensions arising out of some natural homomorphisms in the arithmetic of elliptic and hyperelliptic curves. The results of [121, 39] show that the class groups will become arbitrarily big when a parameter n tends to ∞ or $-\infty$.

The objective of this paper is to compute a bound on the effective estimate (defined in Chapter 5) for the orders of the class groups of a family of imaginary quadratic number fields. That is, we estimate how small or large some parameter n needs to be in order for the associated class group to have size greater than some pre-defined value M.

Chapter 2

Integer Factorization using Binary Decision Diagrams

2.1 Introduction

Prime numbers have fascinated mathematicians since antiquity. The concept of a prime number was probably known to Pythagoras, and acquired a clear definition in Euclid's "Elements" (300 BC): a positive integer N is a *prime* if it does not have any non-trivial divisors, in other words, if the only integers that divide N are 1 and N.

The problem of finding the prime factors of an integer was an immediate consequence of the fundamental theorem of arithmetic enunciated in the "Elements": every positive integer N greater than 1 can be represented uniquely as a product of prime powers. Formally, if $p_1, p_2, \ldots, p_i$ are all distinct primes dividing N arranged in increasing order of magnitude, then N can be uniquely written as $N = p_1^{e_1} p_2^{e_2} \ldots p_i^{e_i}$, where e_i are positive integers.

It ought to be stressed that Euclid's theorem is only existential in nature. It is not constructive, in the sense that the proof does not give any method of finding the prime divisors of N.

We start this chapter by discussing some historical and modern factorization algorithms. We then proceed to a discussion of a method that we use to attack the factorization problem, which utilizes data structures called Binary Decision Diagrams (BDD). At the outset, it must be stated that BDD is a well known graph theoretic structure [59] used to represent Boolean functions. In this chapter, we consider the product $N = pq$ of two large unknown prime numbers p and q, and focus on how to recover p and q if N is known. We write all these integers in binary, with the bits of p and q corresponding to unknown variables. We then express N as the binary product of p and q and thereby obtain relations between the bits of p and q (which are unknown) and the bits of N (which are given).

We represent this natural Boolean system as a BDD, which we subsequently transform until we arrive at an equivalent, concise BDD from which the values of p and q can be read. The resulting complexity appears in practice to be of the order $\mathcal{O}(\sqrt{N})$. Asymptotically, this would be comparable only to the trial division method (described in the next section), though the latter is faster than the method proposed in this chapter.

2.2 Early factorization algorithms

In this section, we present some classical algorithms for solving the integer factorization problem.

1. *Trial Division:* Trial division is a brute force method which goes through all primes $p \leq \sqrt{N}$ and tests whether they are divisors of N. The algorithm tries to factor N by each prime in turn, removing each prime factor it finds. The algorithm stops when the next prime trial divisor exceeds the square root of the remaining cofactor. This method is impractical for large integers with large prime factors. In the worst case, it can take up to $\mathcal{O}(\sqrt{N})$ steps. There are some techniques such as quadratic residues, which can speed up trial division by skipping some primes that cannot be divisors of N. More details can be found on page 121 in [129].

2. *Fermat's Algorithm* [129]: Every odd integer N can be written as the difference of two squares, i.e. if N is odd, then it can be written as $N = a^2 - b^2$ for some integers a and b. Let N be a composite number that we want to factor; write $N = x \cdot y$ with $x \leq y$. Since N is odd, x and y must both be odd. Let $a = \frac{(x+y)}{2}$ and $b = \frac{(y-x)}{2}$. Notice that a and b are both integers. Then $x = a - b$ and $y = a + b$ so $N = x \cdot y = (a-b)(a+b) = a^2 - b^2$. Fermat's algorithm succeeds if one manages to express the number N as the difference of two squares, i.e. if one is able to find a and b such that $N = (a^2 - b^2) = (a+b)(a-b)$. In the worst case, this algorithm takes $\mathcal{O}(N)$ operations. If $N = x \cdot y$, and x, y have the same number of bits, then the algorithm takes $\mathcal{O}(\sqrt{N})$ operations.

   This algorithm was improved by Hart and Lehman. Hart proposed a variation of Fermat's factoring algorithm to factor N in $\mathcal{O}(N^{\frac{1}{3}+\epsilon})$ steps.

   Hart's algorithm [45] begins by checking whether N is a square. If N is not a square, then the algorithm performs trial division, but it halts when the prime p being tested for divisibility reaches $N^{1/3}$. If N has not yet been factored, it performs the following steps: for $i = 1, 2, 3, \ldots$, the algorithm tests whether $(\lceil\sqrt{Ni}\rceil)^2 \mod N$ is a square. If this number equals t^2 for some integer t, then $\gcd((\lceil\sqrt{Ni}\rceil - t, N))$ is a factor of N. Later, Lehman [66] reduced the complexity of Fermat's factoring algorithm to $\mathcal{O}(N^{1/3})$.

   In 1920, *Maurice Kraitchick* [131] improved upon Fermat's method as follows: instead of looking for integers a and b such that $a^2 - b^2 = N$, we ask when $a^2 - b^2$ is a multiple of N. In other words, we try to find as many solutions as possible to the congruence $a^2 - b^2 \equiv 0 \mod N$. We will come back to this method when discussing modern factorization algorithms.

3. *Euler's Algorithm* [89]: This algorithm depends on the possibility that the given odd composite integer N can be expressed as the sum of two squares in two different ways; in other words, on the possibility of writing $N = (a^2 + b^2) = (c^2 + d^2)$ for some integers a, b, c, d with $\{a, b\} \neq \{c, d\}$. From these expressions, Euler's method factorizes the given number N by the formula $N = (\frac{k}{2}^2 + \frac{h}{2}^2)(l^2 + m^2)$ where $m = \gcd(a + c, d - b)$, $l = \gcd(a - c, d + b)$, $k = (a - c)/l$ and $h = (a + c)/m$.

These methods were principally designed for calculation by hand for small values of N, and are not suitable if the number of digits of N is very large. The main challenge in factorizing a number is to formulate efficient algorithms without assuming anything about the input integer.

In the early 20th century, electrically driven machines were specifically constructed for the purpose of factoring large integers. For example, in 1927 D. H. Lehmer's "Bicycle-chain sieve" [68] produced the following examples of large integer factorizations: $10^{20}+1 = 73\times 137\times 1676321\times 5964848081$ and $2019210335106439 = 25709599 \times 78539161$.

Later, in 1932, Lehmer improved on this machine and built the "Photoelectric Number Sieve" [10] which was successful in factorizing even larger numbers: $2^{79}-1 = 2687 \times 202029703 \times 1113491139767$ and $2^{93}+1 = 3^2 \times 529510939 \times 715827883 \times 2903110321$.

2.3 Modern factorization algorithms

Factorization problems, which had so far attracted mainly mathematicians, had now acquired a position of central importance in view of their applications to cryptography. The security of the widely used RSA cryptosystem relies on the hardness of the RSA problem. Given an RSA public key (N, e) and a ciphertext $C = M^e \pmod N$, the RSA problem is to compute M. This problem is no harder than factoring integers, since if the adversary can factor the number N, they can compute the private key (N, d) from the public key (N, e). However, it is not clear whether an algorithm for integer factoring can be efficiently constructed from an algorithm for solving the RSA problem [101].

In the following, we mention a few instances of recent factorization algorithms.

1. *Pollard's rho method* [95]: This algorithm, also known as the Pollard Monte Carlo factorization method, was invented by John Pollard in 1975. Let N be a composite positive integer to be factored. The algorithm generates a sequence of integers $x_0, x_1, x_2, \ldots$ modulo N. We start with a random integer $x_0 \in \mathbb{Z}_N$, and a function $f(x)$, which is easily computable (the most common choice is $f(x) = x^2 + a \mod N$, where $a \neq 0$ is an element of $\mathbb{Z}_N$) and subsequently obtain a sequence of integers $x_i = f(x_{i-1})$ for $i \geq 1$.

   Since $\mathbb{Z}_N$ is finite, and x_i is generated from x_{i-1} deterministically, the sequence $x_0, x_1, x_2, \ldots$ must eventually be periodic. Now consider the sequence $x'_0, x'_1, x'_2, \ldots$ where each x'_i is the reduction of x_i modulo the unknown prime p. The reduced sequence $x'_0, x'_1, x'_2, \ldots$ is periodic as well (since each x'_i is an element of the finite set $\mathbb{Z}_p$).

   Let τ and τ' be the smallest period of the sequence $x_0, x_1, \ldots$ and $x'_0, x'_1, \ldots$, respectively. Then $\tau \mid \tau'$. If $\tau < \tau'$, then there exist integers i, j with $i < j$ such that $x_i \not\equiv x_j \mod N$ but $x'_i \equiv x'_j \mod p$, that is, $x_i \equiv x_j \mod p$. In that case, $d = \gcd(x_j - x_i, N)$ is a proper divisor of N. On the other hand, if $\tau = \tau'$, then for all i, j with $i < j$, we have that $\gcd(x_j - x_i, N)$ is either 1 or N. Note that computing $x_j - x_i$ for all i, j with $i < j$ is expensive. The complexity of the Pollard rho method is $\mathcal{O}(\sqrt{p})$, where p is the smallest prime factor of N.

2. *Pollard's* $(p-1)$ *method* [96]: Both trial division and Pollard's rho method are effective for factoring N if N has small prime factors. Pollard's $(p-1)$ method is effective if $p-1$ has small prime factors for some prime p dividing N.

   Pollard's $(p-1)$ method uses Fermat's little theorem, which states that for a prime p and for $a \in \mathbb{Z}_p^*$ we have $a^{p-1} \equiv 1 \mod p$. The theorem implies that for any integer k we have $a^{k(p-1)} \equiv 1^k \equiv 1 \mod p$. Therefore, for any multiple m of $(p-1)$, we have $a^m \equiv 1 \mod p$, that is $p \mid (a^m - 1)$. Thus, computing $\gcd((a^m-1), N)$ might reveal a factorization of N. If p divides N, then p divides $\gcd((a^m-1), N)$. We cannot compute $a^m \mod p$ because p is an unknown prime factor of N. However, we can compute $a^m \mod N$.

   Let B be a positive integer. To find $m > 1$ which is a multiple of $(p-1)$, let $p_1, p_2, \ldots, p_t$ be all primes between 2 and B. For each $i = 1, 2, \ldots, t$, define $e_i = \lfloor \log_{p_i} B \rfloor$. Consider the exponent $m = \prod_{i=1}^{t} p_i^{e_i}$. If $p-1$ is B-smooth, then the exponent m is a multiple of $(p-1)$. A number is called B-smooth if all its prime factors are less than or equal to B. It is important to note that m may be quite large so instead of computing $a^m \mod N$ directly, and then computing $\gcd(a^m - 1, N)$, one may sequentially compute the sequence $a^{p_1^{e_1}}, (a^{p_1^{e_1}})^{p_2^{e_2}}, ((a^{p_1^{e_1}})^{p_2^{e_2}})^{p_3^{e_3}} \ldots$, and so on. After this series of exponentiations, one can compute $\gcd(a^m - 1, N)$. Pollard's $(p-1)$ method can be used to discover factors p in time roughly proportional to the largest prime factor of $p-1$. The algorithm runs in time $\mathcal{O}(B \log^2 N)$. Evidently, this is only going to be efficient for small values of B.

3. *Continued fraction factorization* [67]: This factorization method was developed by D.H. Lehmer and R.E. Powers in 1931 and is based on the continued fraction expansion of $\sqrt{N}$. To factor a positive integer N which is not a square, we compute the partial continued fraction expansion $\{(P_r, Q_r)\}_{r=1}^{\infty}$ of $\sqrt{N}$ (or $\sqrt{kN}$ for some integer k). If P_r and Q_r are the r-th convergents to $\sqrt{N}$ for $r = 0, 1, 2, \ldots$, we have

$$P_r^2 - NQ_r^2 = (\sqrt{N}Q_r - P_r)^2 - 2\sqrt{N}Q_r(\sqrt{N}Q_r - P_r)$$

   so that

$$|P_r^2 - NQ_r^2| \leq |\sqrt{N}Q_r - P_r|^2 + 2\sqrt{N}Q_r|\sqrt{N}Q_r - P_r| < 2\sqrt{N}.$$

   Here $\sqrt{N}$ is irrational because we assume that N is not a square; that is, $\frac{P_r}{Q_r} \neq \sqrt{N}$ for all r. Then $0 < |P_r^2 - NQ_r^2| < 2\sqrt{N}$ for all $r \geq 1$.

   In order to factorize N, we fix a factor base of small primes $B = p_1, p_2, \ldots, p_t$ and compute the convergents P_r/Q_r for $r = 1, 2, 3, \ldots$. Let $t_r = P_r^2 - NQ_r^2$. Going modulo N, we have, $P_r^2 \equiv t_r \mod N$. We check the smoothness of t_r by trial division of t_r by the primes in B. Since some of the values of t_r are negative, we include -1 as an element in the factor base so that $B = -1, p_1, p_2, \ldots, p_t$. Every smooth t_r gives a relation. The crucial point is that $|t_r| \leq 2\sqrt{N}$, which is small relative to N; this follows from the theory of continued fractions. After sufficiently many relations are obtained, we combine

them to obtain congruences of the desired form, and solve them to obtain the factorization of N.

It is worth noting that the continued fraction factorization method was the first integer factorization algorithm with a sub-exponential running time. A sub-exponential expression in ln N is, in this context, given by an expression called L notation, and is defined as: $L_N[\alpha, c] = e^{(c+o(1))(\ln\ N)^{\alpha}(\ln\ \ln\ N)^{(1-\alpha)}}$ where $c > 0$ and $0 \leq \alpha \leq 1$. Plugging in $\alpha = 0$ in $L_N[\alpha, c]$ gives a polynomial expression in ln N. On the other hand, for $\alpha = 1$, the expression $L_N[\alpha, c]$ is exponential in ln N. For $0 < \alpha < 1$, the expression $L_N[\alpha, c]$ is sub-exponential in ln N. The time complexity of the algorithm is $L_N[\frac{1}{2}, \sqrt{2}]$. John Brillhart and Michael Morrison in 1970 used this algorithm to achieve the following factorization:

$$2^{128} + 1 = 59649589127497217 \times 5704689200685129054721.$$

4. *Elliptic curve factorization* [72]: This is the third fastest integer factorization method which makes use of elliptic curves and was found by Hendrik Lenstra in 1987. This factorization method is a clever adaptation of Pollard's $(p-1)$ method. The time complexity of elliptic curve factorization depends on the size of the smallest prime factor p in the factorization of N and is given by $L_p[\frac{1}{2}, \sqrt{2}]$. Therefore this method can be efficient if p is small.

   Given a number N to be factored, we choose an arbitrary elliptic curve $E : Y^2 = X^3 + aX + b \mod N$ and a non-trivial point $P = (\alpha, \beta)$ on the curve. Fix a factor base B as a finite set of small primes and define $M = \sqrt{N}$ and $m = \prod {p_i}^{e_i}$ where $e_i = \lfloor \log\ M / \log\ p_i \rfloor$. As per the addition law on elliptic curves, if the attempt to determine $mP \mod N$ fails, i.e. if the difference in the X coordinates $(X_2 - X_1)$ is not relatively prime to N, then it will be successful in factoring N as $\gcd(X_2 - X_1, N)$ will be a proper divisor of N.

5. *Dixon's factorization method* [31]: This algorithm was designed by John D. Dixon in 1981. Unlike Fermat's method which attempts to express N as a difference of two squares, here the factorization method finds differences of squares modulo N of the form $a^2 \equiv b^2 \mod N$ (as in the Kraitchick method mentioned above) which can be transformed into the factorization $N = \gcd(a + b, N) \times (N/\gcd(a + b, N))$.

   This method works as follows: choose a bound B, and calculate a factor base $P = \{p_1, p_2, \ldots, p_m\}$, which is the set of all primes less than or equal to B. Search for positive integers $\{a_1, a_2, \ldots, a_{m+1}\}$ such that ${a_i}^2 \equiv \prod p_j^{e_{ij}} \mod N$ with $p_j \in P$. After generating enough relations, usually a few more than the size of P, we can use linear algebra (for example, Gaussian elimination) to multiply together various relations in such a way that the exponents e_j for all j are even. This gives us the congruence $a^2 \equiv b^2 \mod N$, which can be transformed into a factorization of N, namely $N = \gcd(a+b, N) \times (N/\gcd(a + b, N))$. If the factorization turns out to be trivial, which happens when $a \equiv b \mod N$, then we have to try out other relations until we get a non-trivial pair of factors of N. Dixon's factorization algorithm has sub-exponential time complexity $L_N[\frac{1}{2}, 2\sqrt{2}]$.

6. *Quadratic sieve algorithm* [97]: This is another sub-exponential integer factoring algorithm, invented by Carl Pomerance in 1981. Many of its ideas go back to Kraitchick and the continued fraction factorization method. The complexity of this method is $L_N[\frac{1}{2}, \sqrt{2}]$, i.e. the same as the continued fraction factorization method.

   However, the quadratic sieve offers the possibility of sieving, a process that replaces trial division in the continued fraction factorization method. As a result, the quadratic sieve method achieves a better running time. Another advantage of the quadratic sieve method over the continued fraction method is that with the quadratic sieve it is possible to distribute the task of factoring to many computers such that each computer can be given its own set of quadratic polynomials to sieve.

7. *General number field sieve algorithm* [70]: This algorithm is the most sophisticated and efficient integer factorization algorithm known to date. It was first proposed for the so-called Fermat numbers, i.e. for numbers of the form $2^{2^k}+1$, where k is some positive integer; later, it was extended to general composite numbers by Pomerance et al [19].

   The number field sieve algorithm is based on the earlier quadratic sieve method but is more involved because all computations are done in the ring of algebraic integers in a suitable number field. The key idea is to exploit the factorization of smooth numbers in a well chosen algebraic number field. The difference over the complexity of the quadratic sieve is that the quantity in the exponent, the power of $\log N$, has its exponent reduced from $1/2$ to $1/3$, giving the complexity $L_N[\frac{1}{3}, \sqrt[3]{\frac{64}{9}}]$.

8. *Quantum factoring* [113]: In 1994, Peter Shor invented a polynomial time quantum factoring algorithm for integer factorization. Shor's algorithm consists of two parts: (i) a reduction of the factoring problem to the problem of finding the order of an element a in the group $(\mathbb{Z}_N^*)$, where N is the integer to be factored; this problem can be solved on a classical computer, and (ii) a quantum algorithm for solving the order-finding problem. In general, quantum algorithms are probabilistic, which means that they find the correct answer with high probability, and the probability of failure can be decreased by repeating the algorithm. Shor's algorithm is significant because given a large (that is, with a sufficient number of qubits) quantum computer, it is possible to break the RSA public key cryptosystem.

2.4 Binary Decision Diagrams

Researchers have studied different ways of transforming the integer factorization problem into the problem of solving systems of Boolean equations. Once we have a system of Boolean equations, there are different types of techniques that can be applied in order to solve them. For example, SAT solving and algorithms using binary decision diagrams have all been tried. In [54, 6, 73], the problem of integer factorization is translated into SAT instances. In [6, 73], the authors also apply

SAT solvers to the SAT instance of the factorization problem and measure their complexity.

Researchers have also considered how to use BDD for the integer factorization problem. In [132] it is shown that the BDD that verifies a multiplication circuit on n bits has node complexity of $\Omega(2^{\frac{n}{2}})$. Such a BDD can be used to factor an integer in polynomial time in the number of nodes. Recently, the authors of [119] looked at a different way to transform integer multiplication into a BDD.

In this chapter, we transform the integer factorization problem into a BDD in a different way compared to the previous works in [119, 132]. Moreover, [119] does not give a factorization algorithm to be used on their BDD. We use the technique of linear absorption to factor RSA numbers using our BDD. The motivation for using the linear absorption technique comes from [98], where it is shown that using this method to solve a Boolean system of equations outperformed SAT solvers on a particular class of problems. Out of academic curiosity, we wanted to see how well this linear absorption technique performs on the integer factorization problem. We ran experiments and got similar behaviour and complexity as in [6, 73], but we tested larger instances. Our experiments suggest that the complexity for factoring $2n$ bit RSA numbers using the method of linear absorption is $\mathcal{O}(2^n)$.

2.4.1 Description of Binary Decision Diagrams (BDD)

Binary Decision Diagrams are data structures used to represent a Boolean function [59]. There are different variants of BDD such as OBDD [14], ZBDD [57] and πBDD [85]. In this chapter, we focus on a particular type of BDD which will be described below.

We focus primarily on representing integers N that have only two prime factors p and q of the same bit size as a system of Boolean equations. By solving the system, one can obtain the prime factors p and q. We believe that our method can be generalized to numbers with arbitrarily many prime factors.

In our work, we define a *Binary Decision Diagram (BDD)* as a directed acyclic graph with a unique root node at the top and a true-node at the bottom. The nodes in a BDD are arranged in horizontal *levels*. In Figure 2.1 we give an example of a BDD, and it may be useful to refer to this figure when reading the description below.

Edges in the BDD connecting two adjacent nodes are labelled as *0-edges* or as *1-edges*. Edges are drawn downwards only, which means that no edges are drawn between the nodes on the same level. The 0-edges are drawn as dotted lines, while the 1-edges are drawn as solid lines.

Each level in a BDD is associated with a single variable or linear combination of variables. A *path* in a BDD is a sequence of consecutive edges, where the end node of one edge is the start node of the next edge. A *complete path* starts in the top node and ends in the bottom node. Since each edge is labelled either as a 0-edge or a 1-edge, the path can be seen as a sequence of binary assignments. A complete path and its corresponding linear system are shown in Figure 2.1.

Each edge in a path can be seen as an assignment of a value (0 or 1) to a variable or linear combination of variables associated with that level. If an edge labelled with $e \in \{0, 1\}$ starts from a node on a level associated with a linear combination l, it yields the linear equation $l = e$. Thus, a complete path gives a system of

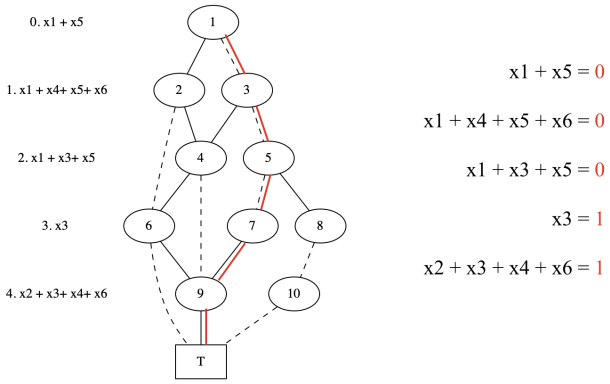


Figure 2.1: A binary decision diagram with 6 levels and linear combinations from a set of 6 variables. The red path corresponds to the linear equation system on the right.

linear equations. The whole BDD after construction encodes a set of binary vectors, namely the solutions to all linear systems generated by all complete paths in the BDD. A linear system may or may not have a solution. If a path gives a linear system that has a solution, we call it a *consistent* path, and, otherwise, we say that it is an *inconsistent* path. Two BDD are equivalent if they encode the same set of binary vectors, i.e. the same set of solutions to all linear systems given by the paths.

An edge need not go to a node on the level directly below. In that case we say that the edge *jumps* over some levels. For an edge that jumps over levels, we can always insert nodes on each level that it jumps over. For example, if an edge jumps from level i to level $i+2$, we can insert a node at level $i+1$ such that the newly inserted node has outgoing 0- and 1-edges, with both of these edges pointing down to the same node in level $i+2$ that the original edge (which jumped over level $i+1$) pointed to. Both of these BDD representations are equivalent, i.e. they encode the same Boolean function.

2.4.2 Operations in a BDD

Given a set of Boolean equations, we draw the corresponding binary decision diagram with nodes, edges and levels. Each level of the BDD is associated with a value, i.e. with a single variable or a linear combination of variables. The number of paths in the BDD can be exponential in the number of nodes; thus, even if the number of nodes is small, the BDD can have a very large number of paths. On a high level, the actual structure of a given BDD encodes all relations between the variables in the system of Boolean equations we are trying to solve. In other words, the structure of the BDD represents the system of Boolean equations. Finding the solution to an equation system represented as a BDD amounts to finding a consistent path in the BDD, and solving the associated linear system of equations in order to obtain a solution to the original system. The operations performed on a BDD are analogous to the elementary row operations used in Gaussian elimination for linear systems in Boolean variables. The operations used are the following:

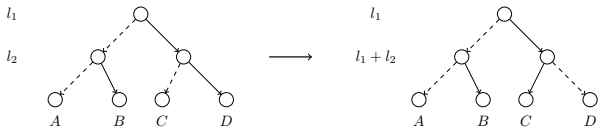


Figure 2.2: Adding levels in a BDD

1. *Adding levels in a BDD:*

   The add operation allows us to add the linear combination corresponding to a level in the BDD to the linear combination corresponding to the level directly below, and change the BDD accordingly to keep the set of solutions encoded by it the same. The procedure is shown in Algorithm 1, and we explain the general case of adding levels using Figure 2.2. The figure shows the case when all edges are present. In the cases where some of the edges are missing, we can imagine that the missing edges lead to some fictive nodes, change the edges according to Figure 2.2, and then remove the fictive nodes. In the case when there is an edge which jumps from level i directly to level $i+2$ (skipping level $i+1$), we can insert a node at level $i+1$ before performing the add operation.

   Let l_1 and l_2 be the linear combinations corresponding to two adjacent levels, with l_1 on the top. In the following, we will identify a linear combination with its corresponding level; in other words, we will refer to the level corresponding to the linear combination l_i simply as "level l_i". We want to add l_1 onto l_2. In the general case, there are two edges going out from the node on the top level: a 0-edge and a 1-edge. By choosing values for l_1 and l_2, we end up in one of the four nodes labelled A, B, C, D in Figure 2.2. When we add l_1 to l_2, the lower level becomes associated with the linear combination $l_1 + l_2$, and the choice of values for l_1 and l_2 must send us to the same node as in the original BDD. For instance, the combination $l_1 = 1$ and $l_2 = 0$ leads to node C in the BDD on the left-hand side of Figure 2.2. That choice of values gives $l_1 + l_2 = 1$, so after adding the levels together, the values $l_1 = 1$ and $l_1 + l_2 = 1$ must also end up in node C in the BDD on the right-hand side. To preserve the set of vectors encoded in the BDD when replacing l_2 by $l_1 + l_2$, we must "flip" edges going out from the node pointed to by the 1-edge originating at the node on level l_1.

   Let l_i be a level containing n nodes. The complexity of adding l_i to l_{i+1} is $\mathcal{O}(n)$ because we only flip 2 edges for every node in the given level, and this is linear in the number of nodes.

2. *Swapping levels in a BDD:*

   When swapping levels, the linear combination associated with level i is interchanged with the one corresponding to level $i+1$, without affecting the set of vectors encoded by the BDD. We explain the general case of swapping levels using Figure 2.3, with pseudo-code given in Algorithm 2.

   Let l_1 and l_2 be the linear combinations corresponding to two adjacent levels, with l_1 being above l_2. We want to swap these linear combinations without changing the set of vectors encoded in the BDD. For the node on level l_1, we have two outgoing edges. As seen previously, choosing values for l_1 and l_2, we

Algorithm 1 Add level (i)

Input: Nodes in the level i and nodes in the level $i+1$ with their associated linear combinations l_i and l_{i+1} respectively.
Output: Nodes in the level i and nodes in the level $i+1$ with their associated linear combinations l_i and $l_i + l_{i+1}$ respectively.

for Every node n on level i **do**
  $n_1 \leftarrow$ child along 1-edge from n.
  Swap 0- and 1-edges from n_1.
end for

Figure 2.3: Swapping levels in a BDD

end up in one of the four nodes labelled A, B, C, D in Figure 2.3. In the BDD on the left-hand side prior to swapping, the choice of values $l_1 = 0$ and $l_2 = 1$ leads to the node labeled B. After we swap l_1 and l_2, the choice of values $l_1 = 0$ and $l_2 = 1$ must still lead to the node labeled B. This is achieved by swapping edges on the lower level of the paths where l_1 and l_2 have different values.

The swap and add operations have linear time complexity in the number of nodes on the two affected levels, so it is very cheap to do when this number is small. The drawback is that the number of nodes on the lower of the two levels may, in the worst case, double during the operation. This happens, for instance, when there is only one node on the lower level prior to applying the operations, but after performing an add or swap operation, two nodes are needed in order to keep the paths intact. Hence, repeatedly applying swap or add operations to the BDD may lead to an exponential growth in the number of nodes. The number of nodes may also decrease after reduction, but finding an arrangement of the linear combinations giving the smallest reduced BDD is an NP-hard problem [9].

Algorithm 2 Swap level (i)

Input: Nodes in the level i and nodes in the level $i+1$ with their associated linear combinations l_i and l_{i+1} respectively.
Output: Nodes in the level i and nodes in the level $i+1$ with their associated linear combinations l_{i+1} and l_i respectively.

for Every node n on level i **do**
  $n_0 \leftarrow$ child along 0- edge from n.
  $n_1 \leftarrow$ child along 1- edge from n.
  Swap 0-edge from n_1 and 1-edge from n_0.
end for

3. *Linear absorption: removing inconsistent paths in a BDD*

   In constructing a BDD to represent a set of equations, there might be some dependencies among the linear combinations. As these dependencies lead to inconsistent paths, we would like to eliminate them in order to facilitate finding the consistent paths that lead to a solution of the associated linear system. This elimination process is called *linear absorption*. This operation is explained in Algorithm 3, and Figure 2.4 shows a small example of the steps carried out in applying linear absorption to a BDD.

   Let $l_0, l_1, \ldots, l_r$ be levels such that $l_0 + l_1 + \ldots + l_r = 0$, so that they are linearly dependent. We start with level l_0 and repeatedly use the swap operation to move it downwards in the BDD until l_0 is at the level just above l_1. We then apply the add operation, transforming level l_1 into a level associated with the linear combination $l_0 + l_1$. Using the swap operation again, and moving $l_0 + l_1$ down to the level just above l_2, we then apply the add operation to transform level l_2 into a level corresponding to $l_0 + l_1 + l_2$. We proceed in the same way until we have processed all $r + 1$ variables.

   The final add operation creates the sum $l_0 + \ldots + l_r$ for the level that was previously associated with l_r. Since $l_0 + l_1 + \ldots + l_r = 0$ by assumption, we have created a level with 0 as its linear combination. We call this level a 0-level. The dependency $l_0 + \ldots + l_r = 0$ has now been condensed into a single level, and whether a path is consistent or not with this dependency now only depends on the edges going out from this level. A path having a 1-edge going out from this 0-level gives a linear system containing the inconsistent "equation" $0 = 1$ directly. Hence, all 1-edges going out from the 0-level can be removed.

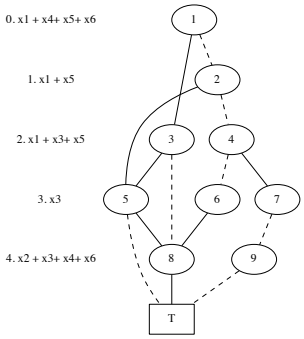
   After removing all 1-edges from the 0-level, all remaining paths in the BDD will be consistent with the particular dependency that we started with, regardless of how we transform the BDD using swap and add operations afterwards. When removing the inconsistent 1-edges, we say that the linear dependency $l_0 + \ldots + l_r = 0$ has been *absorbed* into the BDD.

   A 0-level with only outgoing 0-edges does not give any constraint or information that can be useful in finding a solution to the problem instance. We may therefore remove the entire 0-level. This is done by redirecting all edges pointing to nodes on the 0-level to their children along the 0-edge, and then deleting all nodes on the 0-level.

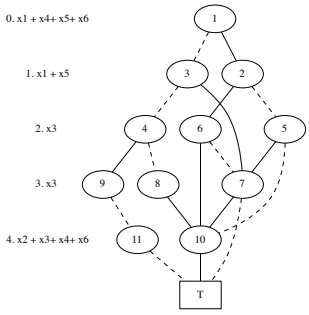
   Repeating the steps in this example, we note that a BDD admitting several linear dependencies can be restructured to a simpler BDD with a reduced number of levels.

   Linear absorption essentially amounts to repeatedly applying the add and swap operations. As we have pointed out earlier, the use of these operations (add and swap) at level i might double the number of nodes at level $i + 1$. Performing one linear absorption involves each level only once, so absorbing one linear dependency might in the worst case double the number of nodes in the BDD. An upper bound on the complexity of absorbing k dependencies is therefore $\mathcal{O}(n2^k)$, where n is the number of nodes in the initial BDD.

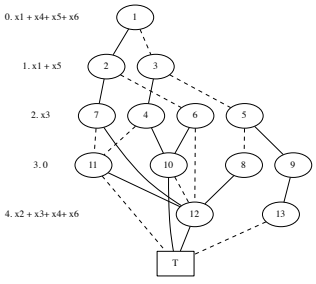
4. *Reducing a BDD:*



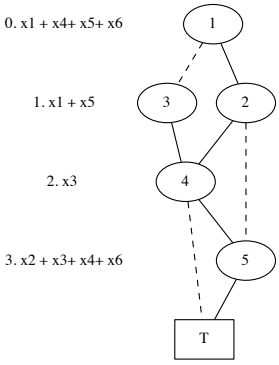
(a) After swapping l_0 and l_1.



(b) After adding l_0 to l_2.



(c) After adding $l_0 + l_2$ to l_3.



(d) After deleting 1-edges and removing 0-level, followed by reduction

Figure 2.4: Absorbing the linear dependency $l_0 + l_2 + l_3 = 0$ where $l_0 = x_1 + x_5$, $l_2 = x_1 + x_3 + x_5$ and $l_3 = x_3$. Every BDD in the figure has the same solution space and encodes the same Boolean function.

Algorithm 3 Linear Absorption $(i_1, i_2, \ldots, i_r)$

Input: BDD with $l_{i_1}, \ldots, l_{i_r}$ as linear combinations on levels $i_1, i_2, .., i_r$ with $i_1 < i_2 < \ldots < i_r$ such that $l_{i_1} + \ldots + l_{i_r} = 0$.
Output: New BDD without the linear dependency $l_{i_1} + \ldots + l_{i_r} = 0$.

for $j = 1$ to $r - 1$ **do**
  $c \leftarrow i_j$
  while $c < i_{j+1} - 1$ **do**
    Swap level (c).
    $c \leftarrow c + 1$
  end while
  Add level (c)
end for
for Every node n on level i_r **do**
  Delete 1-edge from n
  Redirect all incoming edges to n to 0-child of n
  Delete n
end for

A BDD is said to be *reduced* if, for a given arrangement of its linear combinations, it has the minimal number of nodes among all BDD encoding the same set of binary vectors. In other words, a BDD is reduced if no further nodes can be deleted or merged in it while keeping the set of encoded solutions intact.

Performing linear absorption may result in one of the following cases, in which we apply reduction:

(i) The BDD after linear absorption might contain nodes with no incoming or outgoing edges. Such nodes become dead ends for any path reaching them, and they can never be part of a complete consistent path and can therefore be removed.

(ii) If both the 0- and 1-edge of a node t in level i point to the same node t_0 in some level, then we redirect all incoming edges of t to t_0 and delete the node t.

(iii) If a node s on level i has the same 1-child and 0-child as t, then we redirect all incoming edges of s to t and delete s.

Note that deleting one dead end node may create other dead end nodes on the level above, and all such nodes should be deleted in a recursive fashion. We always start reduction on the lowest level and perform the operations level by level, going upwards. This ensures that at any particular level, the part of the BDD below will always be reduced.

In practice, reduction is a semi-local operation that only affects the nodes close to some particular level, and, in the worst case, applying it to one particular level has run time $\mathcal{O}(n^2)$. It has been shown [15] that for a fixed set of linear combinations associated with the levels, a reduced BDD is always unique (regardless of the order in which the nodes are deleted).

Algorithm 4 Reduce (B)

Input: BDD B with k levels
Output: Reduced BDD

```
for Each i = k to 1 do
    for Each node t on level i do
        if t has no incoming or outgoing edges then
            Delete t
        end if
        if 0-edge and 1-edge of t points to the same node t_0 then
            Redirect all incoming edges of t to t_0
            Delete t
        end if
        if Another node s on level i has the same 1-child and 0-child as t then
            Redirect all incoming edges of s to t
            Delete s
        end if
    end for
end for
```

2.5 Integer multiplication represented as a BDD

In this section we will show how the multiplication of two n-bit numbers can be represented via a BDD. The motivation behind this is that we want to express multiplication as as system of Boolean equations, with the unknown factors p and q of a known integer $N = pq$ as variables. By building a BDD that represents these equations and applying the linear absorption operation, we can solve the set of equations in order to factor $N = pq$. We focus on the case of RSA moduli, i.e. $N = pq$ where both $p = (p_{n-1}, \ldots, p_0)$ and $q = (q_{n-1}, \ldots, q_0)$ have n bits each, and N is a fixed and known $2n$-bit value. Nonetheless, the method can easily be extended to the multiplication of arbitrary integers. We have

$$N = \sum_{i=0}^{2n-1} N_i 2^i \qquad p = \sum_{i=0}^{n-1} p_i 2^i \qquad q = \sum_{j=0}^{n-1} q_j 2^j.$$

The product $N = pq$ can now be written as follows, in terms of bits:

$$\begin{array}{cccccccccccc}
 & (q_{n-1} & \ldots & q_2 & q_1 & q_0) & (p_{n-1} & \ldots & p_2 & p_1 & p_0) \\
\hline
 & & & & & & p_0q_{n-1} & \cdots & p_0q_2 & p_0q_1 & p_0q_0 \\
+ & & & & & p_1q_{n-1} & p_1q_{n-2} & \cdots & p_1q_1 & p_1q_0 & \\
+ & & & & p_2q_{n-1} & p_2q_{n-2} & p_2q_{n-3} & \cdots & p_2q_0 & & \\
\vdots & & & \iddots & \vdots & \vdots & \vdots & \iddots & & & \\
+ & & p_{n-1}q_{n-1} & \cdots & p_{n-1}q_2 & p_{n-1}q_1 & p_{n-1}q_0 & & & & \\
\hline
= & N_{2n-1} & N_{2n-2} & \ldots & N_{n+1} & N_n & N_{n-1} & \ldots & N_2 & N_1 & N_0
\end{array} \tag{2.1}$$

The plus signs in Table 2.1 represent ordinary integer addition. We refer to all terms of the form p_iq_j with $i + j = c$ as *column* c of the table.

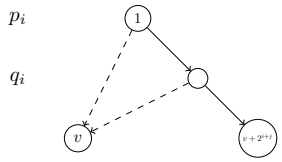


Figure 2.5: The term $p_i q_j$ adds 2^{i+j} to v when $p_i = q_j = 1$, and 0 otherwise.

Let v_k be the number given by the k least significant bits of N. We now look more closely at how the number N is built from the unknown terms $p_i q_j$. We set $v_0 = 0$. We look at column 0, which has the term $p_0 q_0$. This column will contribute the value $2^{i+j} = 2^{0+0} = 1$ to v_0, provided $p_0 = q_0 = 1$. For column 1, we will add the value $2^{i+j} = 2^{0+1} = 2$ to v_1 for each of the terms $p_0 q_1$ and $p_1 q_0$ that evaluates to 1. We compute the numbers v_k for $0 \leq k \leq 2n - 1$ in the same manner.

Adding up the first k columns of the multiplication table above will result in the number $v_k + 2^k v$, where v is the carry pattern from the additions that will affect the next columns. Note that the k least significant bits of N will be determined by the additions in the first k columns, and will not change when performing the rest of the multiplication.

2.5.1 Basic building block of the BDD

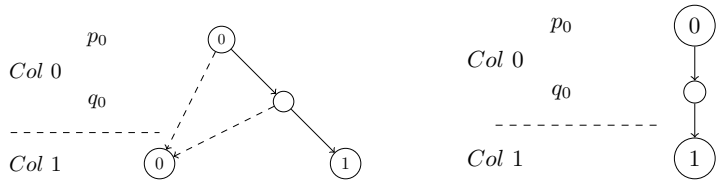
The terms that are added when computing the product are of the form $p_i q_j$, which can have values 0 or 1. The term $p_i q_j$ appears in column $i + j$, and hence will contribute either 0 or 2^{i+j} to N. We can associate the value of N computed so far to the nodes in a BDD. If the value computed before processing $p_i q_j$ is v, the value after processing $p_i q_j$ will be either v or $v + 2^{i+j}$. This is expressed in the graph structure in Figure 2.5. This small subgraph is the basic building block for constructing a BDD representing the whole multiplication $N = pq$.

2.5.2 Building the multiplication BDD

We construct a BDD representing the multiplication by going through the multiplication table (2.1) column by column, starting at column 0.

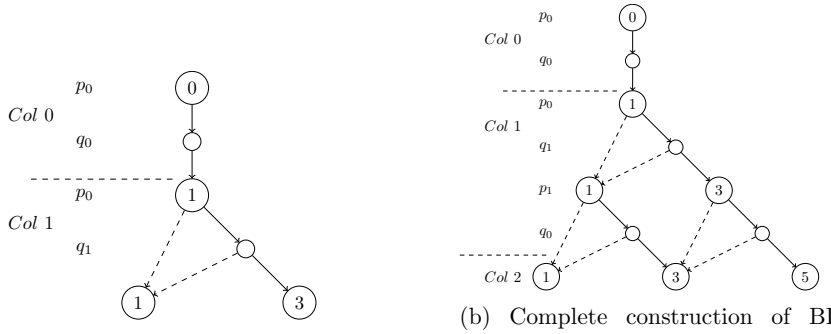
Column 0: Initially, the value computed so far, v_0, is 0. We begin constructing the BDD by building the structure for column 0, which only contains the term $p_0 q_0$. The top of the BDD will only contain the basic building block, as shown in Figure 2.6a. Since processing $p_0 q_0$ completes the contribution from column 0, the possible values for v_1 are listed in the nodes on the lowest level (so far only 0 and 1). We can now find all values that do not match the actual v_1 given by the known N, and delete the nodes corresponding to those values. In the RSA model, N being odd, the bit values p_0 and q_0 are necessarily equal to 1, and so we delete the node that corresponds to $v_1 = 0$. The result is shown in Figure 2.6b.

Column 1: We continue building the BDD from the bottom node in Figure 2.6b. The first term in column 1 is $p_0 q_1$, so the two new levels we get when adding the basic building block starting from this node will be associated with p_0 and q_1. The basic building block has two nodes on the bottom, and the values in these nodes will be 1 in the case that $p_0 q_1 = 0$, and 3 when $p_0 = q_1 = 1$. This is shown in Figure



(a) p_0q_0 adds 1 when $p_0 = q_0 = 1$. (b) After deleting impossible node for v_1.

Figure 2.6: BDD after processing column 0.



(a) Adding p_0q_1 to the value we have so far. (b) Complete construction of BDD for columns 0 and 1.

Figure 2.7: BDD after processing column 1.

2.7a.

To add the second term p_1q_0 in column 1, we extend the BDD by adding two basic building blocks from each of the bottom nodes we have up to now. The levels for the nodes will be associated with p_1 and q_0. Extending from the node containing the value 1 will produce two new bottom nodes with values 1 and 3, respectively. Extending from the node with value 3 will produce two new bottom nodes with values 3 and 5, respectively. Nodes with equal values can be merged, so we end up with the BDD in Figure 2.7b.

We have now finished the construction for column 1, and can check to see which nodes have values consistent with the known v_2, i.e. the number given by the two least significant bits of N. If $v_2 = (01)_2$, we delete the node on the lowest level with value 3, and if $v_2 = (11)_2$, we delete the nodes containing 1 and 5.

Column k: We can continue building the BDD in a recursive fashion. Assume that we have completed the construction for column $k-1$, and have a current bottom level with t nodes on it. Let these nodes be $A_0, \ldots, A_{t-1}$, where the value in each A_i is $v_k + i2^k$. We extend a basic building block corresponding to the first term in column k from each A_i.

The values in the two bottom nodes extending from A_i will be $v_k + i2^k$ and $v_k + (i+1)2^k$. The latter of these values will be the same as the value in the first bottom node extending from A_{i+1}. These nodes can be merged, so all basic building blocks will be linked together as shown in Figure 2.8. Due to this linking, the number of nodes on the new bottom level will be $t + 1$ instead of $2t$.

We continue in this fashion for each of the terms in column k. Let the number of terms in column k be a_k. Each term adds one to the number of nodes on the

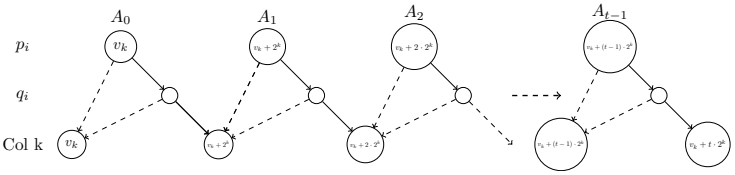


Figure 2.8: Adding the first term $p_i q_j$ in column k. Basic building blocks are linked together.

new bottom level, so after adding the basic building blocks for the last term, the number of nodes on the bottom level will be $t + a_k$. Their values will be $v_k + i2^k$ for $i = 0, \ldots, t + a_k - 1$.

At this point the addition of column k is complete, and we must check which nodes have values consistent with v_{k+1}. If $N_k = 0$, the nodes with values $v_k + i2^k$ for even i will match the given N, and if $N_k = 1$, the nodes with i odd will be consistent with N. Hence, every other node will be inconsistent with v_{k+1} and must be deleted, so that we end up starting the next column with $\lfloor (t + a_k)/2 \rfloor$ or $\lceil (t + a_k)/2 \rceil$ nodes, depending on the given value of N.

Completing the BDD: Once the construction for the final column is done, the full multiplication $pq = N$ is expressed in the BDD. At this point, only the node on the bottom level having value N should be kept, and all other nodes on that level should be deleted. The node with value N becomes the bottom node of the BDD. Finally, the BDD should be reduced to remove any remaining dead ends.

2.5.3 Size of the constructed BDD

We conclude the section by proving that the number of nodes in the initial BDD representing $N = pq$ will be $\mathcal{O}(n^3)$. Hence, the multiplication of large RSA numbers that cannot be factored in practice can still be easily represented as a BDD.

The following lemma gives a formula for the number of terms in a particular column. The correctness of the formula easily follows from the multiplication table (2.1).

Lemma 2.5.1. *Let a_k be the number of terms in column k of the multiplication table representing the product $pq = N$, where p and q are n-bit numbers. Then*

$$a_k = \begin{cases} k + 1 & 0 \leq k \leq n - 1, \\ 2n - k - 1 & n \leq k \leq 2n - 2. \end{cases}$$

The next step is to count the number of nodes on the bottom level of the BDD when starting the construction of the nodes for column k. Our aim is to get an upper bound on the number of nodes in the BDD. In the following, when we have t nodes on the bottom level before deleting nodes that contradict the known value of v_k, we always assume that the number of nodes remaining after deletion is $\lceil t/2 \rceil$.

Lemma 2.5.2. *Let t_k be the number of nodes on the first level representing the addition of terms in column k. Then*

$$t_k \leq \begin{cases} 1 & k = 0, \\ k & 1 \leq k \leq n, \\ 2n - k + 1 & n + 1 \leq k \leq 2n - 1. \end{cases}$$

Proof. The case for $k = 0$ is trivial: the BDD starts with a single top node, so clearly $t_0 = 1$.

We notice that the multiplication table (2.1) of $2n$-bit numbers p, q follows the pattern that the number of terms in each column increases by one up until column $n-1$. We prove our claim for k in the range $1 \le k \le n$ by induction. The statement is true for $k = 1$ since we always start with a single node after column 0 is complete when making nodes for column 1. Assume that $t_k = k$ for some $1 \le k \le n-1$. The addition of the first term in column k will add two new levels to the BDD; the first will contain k nodes, and the second will contain $k+1$ nodes. Each addition of a new term adds two new levels, and the number of nodes on the second of these will have one more node than the previous one. Since there are $a_k = k+1$ terms by Lemma 2.5.1 in column k, the number of nodes on the bottom level after adding the last term will be $k + (k+1) = 2k+1$. Half of the nodes are inconsistent with the given v_{k+1}; after removing them, we are left with $t_{k+1} = \lceil (2k+1)/2 \rceil = k+1$ nodes. This shows the correctness for the case $1 \le k \le n$.

The pattern in the multiplication table (2.1) from column n to column $2n-1$ is that the number of terms decreases by one for each column. We show the formula for $n+1 \le k \le 2n-1$ also by induction, but the base case is less trivial this time. We know that $t_n = n$, and need to show that $t_{n+1} = 2n - (n+1) + 1 = n$ to start the induction. The construction of nodes for column n starts with t_n nodes. In a similar fashion as explained above, two new levels get added for each term in column n, and the second of these increases by one node as compared to the previous one. The bottom level, after adding all a_n terms, is the starting level for column $n+1$, and will contain $t_n + a_n$ nodes. We then delete half of these nodes (which are in conflict with the value of v_{n+1}). With $t_n = n$ and $a_n = n-1$ from Lemma 2.5.1, we get $t_{n+1} = \lceil (2n-1)/2 \rceil = n$, which justifies the base case.

Assume now $n+1 \le k \le 2n-2$ and $t_k = 2n-k+1$. As explained above, the number of nodes on the level at which column $k+1$ starts will be $t_k + a_k$ before deleting nodes, and $\lceil (t_k + a_k)/2 \rceil$ after deletion. We then get $t_{k+1} = \lceil (2n-k+1+2n-k-1)/2 \rceil = \lceil (4n-2k)/2 \rceil = 2n-(k+1)+1$, as desired. □

Knowing a_k and t_k for $0 \le k \le 2n-2$, we can count the (maximum) number of nodes in the part of the BDD representing the additions in column k.

Lemma 2.5.3. *Let T_k be the number of nodes in the BDD representing the additions in column k. Then*

$$T_k \le \begin{cases} 2 & k = 0, \\ 3k^2 + 3k & 1 \le k \le n-1, \\ 3n^2 - 5n + 2 & k = n, \\ 3(2n-k-1)(2n-k) & n+1 \le k \le 2n-2. \end{cases}$$

Proof. By the construction of the previous columns, the level at which additions in column k begin contains t_k nodes. Each term in column k adds first a new level with the same number of nodes as the level above, and then a level with one node more. This continues for each of the a_k terms in column k, so that the last two levels added have $t_k + a_k - 1$ and $t_k + a_k$ nodes, respectively. The last level belongs to column $k+1$, and should not be counted, so we get $t_k + t_k + (t_k + 1) + (t_k + 1) +$

$\ldots + (t_k + a_k - 1) + (t_k + a_k - 1)$ nodes for column k in total. This can be written as

$$T_k = \sum_{i=0}^{a_k-1} 2(t_k + i) = 2a_k t_k + 2\sum_{i=0}^{a_k-1} i = 2a_k t_k + a_k(a_k - 1) = a_k(2t_k + a_k - 1).$$

It is now straightforward to verify the four cases stated in the lemma by inserting the expressions for a_k and t_k from Lemmas 2.5.1 and 2.5.2 into the equation. □

We are now ready to state the main result.

Theorem 2.5.4. *Let $N = pq$, where N is known and p and q are unknown n-bit integers. Let B_n be the number of nodes in the BDD representing $N = pq$. Then*

$$B_n \leq 2n^3 - 4n + 5.$$

Proof. The proof follows by summing up the values for T_k given in Lemma 2.5.3 for all columns in the multiplication table, and adding 1 for the bottom node. In the calculations we make use of the standard formulas

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

and

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}.$$

The proof then follows by straightforward calculation:

$$\begin{aligned}
B_n &= 1 + \sum_{k=0}^{2n-2} T_k = 1 + T_0 + \sum_{k=1}^{n-1} T_k + T_n + \sum_{k=n+1}^{2n-2} T_k \\
&\leq 1 + 2 + \sum_{k=1}^{n-1}(3k^2 + 3k) + (3n^2 - 5n + 2) + \sum_{k=n+1}^{2n-2} 3(2n - k - 1)(2n - k) \\
&= 3n^2 - 5n + 5 + 3\sum_{k=1}^{n-1} k^2 + 3\sum_{k=1}^{n-1} k + 3\sum_{k=1}^{n-2}(2n - (n+k) - 1)(2n - (n+k)) \\
&= n^3 + 3n^2 - 6n + 5 + 3\sum_{k=2}^{n-1}(k-1)k \\
&= 2n^3 - 4n + 5.
\end{aligned}$$

□

2.6 Factoring using BDD with linear absorption

Summarizing all our observations and results above, an algorithm for finding the factors p and q of an unknown integer $N = pq$ can be states as follows.

We have constructed many BDD for various values of N, following the description given in the previous section. For two n-bit numbers $p = (p_{n-1} \ldots p_0)$ and

Algorithm 5 BDD Factoring (N)

Input: A positive integer $N = pq$ where p and q are primes of equal bit-length n.
Output: p and q

$B \leftarrow BDD$ representing $N = pq$
while Linear dependencies exists in B **do**
    Find $i_1, i_2 \leftarrow$ levels such that $l_{i_1} + l_{i_2} = 0$
    Linearly absorb (i_1, i_2)
    Reduce(B)
end while
$\Delta \leftarrow$ path in B
Assign values of $p_0, p_1, \ldots, p_{n-1}$ and $q_0, q_1, \ldots, q_{n-1}$ from Δ
Return $p = \sum_{i=0}^{n-1} p_i 2^i$ and $q = \sum_{i=0}^{n-1} q_i 2^i$.

$q = (q_{n-1} \ldots q_0)$, there are n^2 different terms $p_i q_j$ used in the computation of the product pq. Each term will give two levels in the constructed BDD, so the total number of levels in the BDD will be $2n^2$. There are exactly $2n$ unknown variables p_i and q_j, and each variable initially occurs on exactly n different levels. This duplication of variables in different levels is what gives rise to all the linear dependencies. See Figure 2.9 for an example of what the structure of a complete multiplication BDD looks like. Note the pattern that emerges from the borders between different columns, where half the nodes have been deleted.

Each path in the BDD suggests values for the unknown p_i and q_j. Almost all of the paths are inconsistent, because a path may very well choose $p_i = 0$ on one level where p_i occurs and $p_i = 1$ on another. If we can remove all inconsistent paths, any remaining path in the BDD will give values of p_i and q_j such that $pq = N$.

Figure 2.9: BDD for $N = 471953$.

2.6.1 Observed complexity of factoring

We have performed experiments using linear absorption to remove inconsistent paths; the exact procedure is given in Algorithm 5. There are $2n$ variables in total and $2n^2$ levels in the BDD, so there will be $2n^2 - 2n$ different dependencies we need to absorb before every remaining path is consistent. Once we have reached that point there will only be two paths remaining in the BDD. Both of them give values for p and q such that $pq = N$ (there are two paths because it is undecided which factor is p and which is q, e.g. for $N = 77$ we can have $p = 7, q = 11$ or $p = 11, q = 7$).

An RSA modulus represented as a BDD can be factored using linear absorption, but we need a measure of its complexity. As proved in Theorem 2.5.4, the initial BDD will contain only polynomially many nodes in the number of bits. When doing the swap and add operations during linear absorption, the number of nodes will increase. However, the final BDD, after all dependencies have been absorbed, is very small since it only contains two paths. At some point, the BDD will therefore reach a maximum

number of nodes, and operating on this BDD will give the heaviest work, in terms of both time and space complexity. Hence, we take the maximum number of nodes during factoring as our measure of complexity.

It is very hard to predict in advance how many nodes the BDD will contain after absorbing a number of dependencies. Due to this, we do not have a closed formula $f(n)$ for the complexity of factoring an n-bit number using our method. The strategy which we use to absorb the linear dependencies is as follows. First, we compute all linear dependencies that exist in the BDD. For each dependency, we compute its cost. The cost is computed by the number of nodes that are involved in all levels that will be used in the add and swap operations during the linear absorption of this particular dependency. The strategy for choosing which dependency to absorb is to employ a greedy approach by always absorbing the dependency with the smallest cost. The observed complexity of factoring $2n$-bit numbers with the BDD approach and this strategy of linear absorption appears to be of the order 2^n. Table 2.1 shows the details and actual run times for some particular values of N.

The construction and resolution of the BDD used to find the factors p, q was first coded in C. The results are shown in Table 2.1. Later, with the development of the "Cryptapath" tool [26], new experiments were conducted for larger moduli using this tool. For each $20 \leq n \leq 28$, we generated 100 instances of RSA numbers to factor. Figure 2.10 shows the minimum, the maximum and the mean number of nodes in the BDD constructed when factoring these numbers.

N	p	q	$\lceil \log_2(N) \rceil$	peak number of nodes	runtime (s)
479069	571	839	19	$2^{12.996}$	0.316
1887239	1249	1511	21	$2^{14.070}$	0.760
8795869	2741	3209	24	$2^{14.925}$	1.246
288676361	16603	17387	29	$2^{18.347}$	18.86
9657443137	93407	103391	34	$2^{20.136}$	80.3
163580897747	405917	402991	38	$2^{22.303}$	537

Table 2.1: Some factoring experiments.

2.6.2 Further observations on the constructed BDD

Clearly, factoring RSA moduli via the method proposed in this chapter cannot compete with any of the best factoring algorithms known, such as the number field sieve [70], in terms of asymptotic complexity. The purpose of this chapter is rather to give a different approach to the factoring problem, and hopefully inspire some new ideas. Below, we present two observations that might be useful for future work.

Absorbing $2n - 2$ dependencies for free

Finding an order in which to absorb the dependencies such that the overall complexity is minimised is a challenging problem. However, it is possible to absorb $2n - 2$ dependencies from the starting BDD without any increase in the number of nodes. This can be explained as follows.

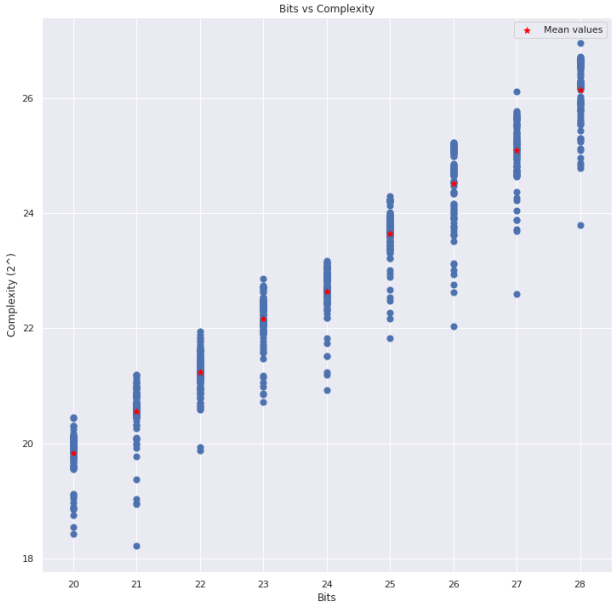


Figure 2.10: Complexities for factoring RSA numbers with p and q of bit-size 20 to 28. 100 instances were generated for each bit-size, with each recorded complexity represented as a blue dot. The red stars mark the mean values for each bit-size.

The terms in column $k < n$ are $p_0q_k, p_1q_{k-1}, \ldots, p_kq_0$, but the order of adding the terms is irrelevant to the computation of N. In the BDD, the variables p_i and q_{k-i} must appear on adjacent levels, but the pairs (p_i, q_{k-i}) can be put in any order on the levels representing column k without changing the solution set corresponding to the BDD. Similarly, the order of the components within the pair does not matter, i.e. either (p_i, q_{k-i}) or (q_{k-i}, p_i) can be considered. Permuting the variables like this within one column does not change the node structure in the BDD.

Say now that p_i appears in both column k and column $k-1$. It is then possible to put the pair (p_i, q_{k-i-1}) at the lowest possible levels within column $k-1$, and set p_i as the lowest of these two. In column k, we can set the pair (p_i, q_{k-i}) at the highest possible levels within column k, and set p_i as the highest. The two levels with p_i will now be adjacent and can be merged into one, absorbing one dependency. The number of nodes in the BDD will actually decrease since the BDD was unchanged up until the two levels containing p_i were added together, and then the lower of the two levels gets deleted.

We can let two variables "meet" like this on each of the $2n-2$ column boundaries, and absorb $2n-2$ variables without any increase in the number of nodes. Unfortunately, this process can not be readily repeated since the column boundaries now consist of three levels that are dependent on each other and can not freely move without increasing the number of nodes. To merge other variables that are equal

means that one of them has to cross a column boundary, with no guarantees on how this will affect the resulting number of nodes.

Moving one variable to the top

Each variable p_i and q_j initially occurs on n different levels. Let us start with the lowest level where, say, p_i is found, and use the swap operation to move it upwards through the BDD. Each time this instance of p_i is just below another level containing p_i, we merge the two levels using linear absorption before continuing. In the end, the variable p_i only occurs at the highest level, where the top node is located. Figure 2.11 shows an example constructed from the multiplication of two 8 bit primes $p = (p_7, \ldots, p_0) = (10010101)_2 = (149)_{10}$ and $q = (q_7, \ldots, q_0) = (11010011)_2 = (211)_{10}$.

The resulting BDD has now been split into two parts between the level where we started; one part for $p_i = 0$ and one for $p_i = 1$ (see the BDD on the left-hand side in Figure 2.11). If it were possible to somehow detect which part contains the consistent path, we would learn whether $p_i = 0$ or $p_i = 1$, delete the other part of the BDD by cutting off the 1- or 0-edge from the top node, and iterate the process with another variable.

In the BDD on the right-hand side in Figure 2.11, we show an example where we have moved q_7, initially found at the lowest level, to the top. The BDD splits completely in two, and in this example we can see that guessing $q_7 = 0$ immediately determines the values of several other variables. All paths in the part of the BDD where $q_7 = 0$ end in the same string of 1-edges, indicating that $p_5 = p_6 = p_7 = q_3 = q_4 = q_5 = q_6 = 1$ must be true for there to be any possibility for p and q to multiply to N.

2.7 Conclusion

In this chapter, we have shown how to use binary decision diagrams to factor a number N which has two prime factors p and q of the same bit size. Unfortunately, according to our experiments, the running time and space requirements for factoring RSA moduli N with $2n$ bits seems to be of order $\mathcal{O}(\sqrt{N})$. When the bit lengths of p and q are different, the complexity is expected to be of order $\mathcal{O}(\sqrt{\max(p, q)})$. As mentioned earlier, the complexity of this method is comparable only to trial division which has a similar complexity but performs much faster in practice than the method proposed in this chapter. Our results can also be compared to the SAT solving techniques used in [6, 73]. In these papers, the authors use the factorization problem to test their SAT solvers. The complexity of their method is of the same magnitude as ours, but in our work we have conducted experiments with higher bit size of N.

One interesting question arising from our work is to find a good ordering of the variables which can improve the representation of the function via BDD. Variable ordering is the process of choosing a particular order for the variables and employing a BDD to represent that system accordingly. Certain functions are sensitive to variable ordering which can induce exponential growth in the BDD. In principle, the variable ordering can be selected arbitrarily, since the BDD corresponding to any ordering will produce the correct solution. In practice, selecting an optimal ordering which leads to a compact representation of the system is critical.

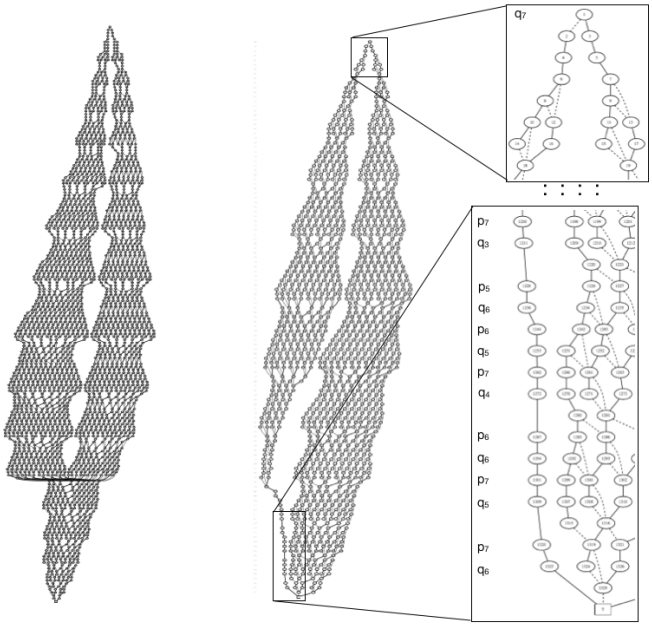


Figure 2.11: Absorbing all instances of one variable and moving it to the top splits the BDD in two.

Chapter 3

Graphs and Self-Dual Codes over $GF(4)$

3.1 Introduction

The mathematical principles underlying digital communication theory were enunciated in Shannon's seminal paper from 1949 [111]. The main theorem presented in this paper guarantees the existence of error correcting codes with desirable properties. This has motivated later researchers to look for more and better error correcting codes that could be applied in practice. Applications of such codes range from satellite communications to data compression (JPEG). Subsequently, the theory of error correcting codes has been an active area of research having surprising and deep connections with other areas of mathematics like number theory, algebraic geometry, and the geometry of numbers. In particular, error correcting codes have been constructed over Galois fields (Reed-Solomon codes, Reed-Muller codes, etc). Asymptotically good families of codes have been discovered using deep properties of class fields of imaginary quadratic extensions [128].

As mentioned above, Shannon's theorem assures the existence of good error correcting codes (that is, codes for which the probability of error after decoding can be made arbitrarily small by increasing the length of the codewords), provided certain conditions are satisfied. One of these conditions is related to the so-called *channel capacity*. The channel capacity is a numeric value that characterizes a given communication channel. For example, for the commonly encountered *binary symmetric channel* with crossover probability p, the channel capacity is given as $1 - h_2(p)$, where $h_2(p) = -p\log_2 p - (1-p)\log_2(1-p)$ is the binary entropy function. The parameters of a code are given by (n, k, d), where n is the length of the codewords, k is the number of codewords, and d is the minimum distance (that is, the smallest distance between any pair of distinct codewords). A code whose codewords form a linear subspace of the n-dimensional vector space is called a linear code; the dimension of this subspace is called the dimension k of the code, and its parameters are given by $[n, k, d]$.

A formal mathematical statement of Shannon's theorem can be given as follows. The statement refers to the so-called rate of the code. The *rate* of a q-ary code, i.e. a code over an alphabet of q symbols, with codeword length n and number of codewords k is defined as $R = n/\log_q(k)$.

Theorem 3.1.1. *[111] Consider a communication channel with channel capacity c.*

Let $\epsilon > 0$, and let $R < c$ be an integer. Then there is a sufficiently large integer n such that there exists a binary code of length n, with rate R and with error probability $P_{err} < \epsilon$.

Unfortunately, the proof of Shannon's theorem is not constructive. This has motivated a lot of research on the topic. For some applications and communication channels, it is known how to construct codes with excellent performance that are very close to the Shannon limit.

However, many important applications may impose strict limitations on the code parameters and structure, for example on the code length. For such applications, it may not be possible to find a code that operates close to the Shannon limit. Instead, a practical goal of code design is to construct a code with large minimum distance d, which will work well on a channel without too much noise. For example, on a binary symmetric channel with crossover probability p, the error probability after decoding is upper bounded (although not tightly) by $1 - \sum_{w=0}^{t} \binom{n}{w} p^w (1-p)^{n-w}$, where $t = \lceil (d-1)/2 \rceil$ is the number of errors that the code is guaranteed to correct. In practice, the value of p is assumed to be small, and therefore there is a very high probability that only a few errors will occur. Thus, it is important to find codes with a large minimum distance d, which can correct a large number t of errors.

One of the main problems in coding theory is to find codes with a given length n and a large number of codewords k having minimum distance d as large as possible. The greater the rate R of the code, the better its efficiency; on the other hand, the greater the values of k and n, the longer the decoding delay. Furthermore, the larger the minimum distance d, the more errors the code is guaranteed to correct. Clearly, these aspects are in conflict, and hence the goal of coding theory is to find codes achieving a good trade off between these values.

Other important problems concerning error correcting codes are:

(i) determining the minimum distance of a given code,

(ii) computing the weight enumeration of a given code (that is, the number of codewords of weight i for each possible value of i), and

(iii) classification of codes.

These are instances of difficult problems in computer science in the context of coding theory.

In this chapter, we focus on a set of codes called *self-dual additive codes* over the finite field of 4 elements, $GF(4)$, and address some of the issues mentioned above.

Self-dual, or self-orthogonal, codes are important for a number of practical and theoretical reasons [11, 75, 76, 79, 80, 93]. After the publication of [20], additive self-orthogonal codes over $GF(4)$ under the trace inner product (which will be defined later) became a subject of intense study due to their correspondence to additive quantum error-correcting codes known as binary stabilizer codes. Several papers [7, 20, 27, 35, 52] were devoted to constructing and classifying self-dual additive codes over $GF(4)$.

All additive self-dual codes over $GF(4)$ of length $n \leq 5$ have previously been classified up to equivalence by Calderbank et al. [20]. For $n \leq 7$, the classification was done by Höhn [52] and Hein et al. [46], and for $n \leq 9$, it was done by Glynn et al. [64]. Danielsen and Parker [27] classified all self-dual additive codes over $GF(4)$ of length $n \leq 12$ by using a graph based algorithm.

In this chapter, we follow the classification of Danielsen and Parker [27]. In the former, the classification is done for general graphs. In our work, we restrict ourselves to a sub-class of graphs: every graph has a property called *rankwidth* (which we will define later in this chapter); we only consider graphs of rankwidth 1. We study self-dual additive codes over $GF(4)$ and investigate the possibility of classifying them using graphs of rankwidth 1. By restricting ourselves to this particular sub-class of graphs, the classification becomes significantly faster. Even though this method is applicable only to a subset of codes, our approach provides a significant speed up as compared to the method of Danielsen and Parker: while the branching factor of the latter is $2^{n-1} - 1$, for graphs of rankwidth 1 it is only $3(n-1)$.

Another important step in the classification of codes (as seen in [27]) is to compute the weight enumerator W_C of the code C. Below, we formally introduce two important notions related to this concept.

Definition 3.1.1. Let C be an (n, k, d) code. For $i \in \{0, 1, 2, \ldots, n\}$ let A_i denote the number of codewords in C of weight i. The collection A_i is called the *weight distribution* of the code C.

Definition 3.1.2. Assume the same notation as in the previous definition. The *weight enumerator* of C is defined to be the homogeneous polynomial $W_C(x, y) = A_0x^n + A_1x^{n-1}y + A_2x^{n-2}y^2 + \ldots + A_ny^n$.

In general, for a linear $[n, k]$ code, the best known method of computing its weight enumerator essentially amounts to brute-force and has complexity $n2^k$. One of the most important results in this area is the MacWilliams identity which relates the weight enumerator of a linear code W_C to the weight enumerator of its dual $W_{C^\perp}$. A formal statement is given below.

Theorem 3.1.2. *(The MacWilliams theorem for binary linear codes) [77, p. 127]. If C is an $[n, k]$ binary linear code with dual code $C^\perp$ then*

$$W_{C^\perp}(x, y) = \frac{1}{|C|} W_C(x + y, x - y),$$

where $|C| = 2^k$ is the number of codewords in C. Equivalently,

$$\sum_{u \in C^\perp} x^{n-wt(u)} y^{wt(u)} = \frac{1}{|C|} \sum_{u \in C} (x + y)^{n-wt(u)} (x - y)^{wt(u)}.$$

One can determine the weight enumerator (and hence the weight distribution) of a code C given the weight enumerator of its dual code $C^\perp$. This is useful in the case when the dimension k of C satisfies $k > n - k$, because then the dimension of the dual $C^\perp$ is smaller than $n - k$, and thus its weight enumerator can be computed more efficiently, with the weight enumerator of the prime code being then obtained via the MacWilliams identity.

We propose a different way of computing the weight enumerator using Binary Decision Diagrams (BDD) (which we already introduced in Chapter 2). The complexity of computing the weight enumerator using BDD remains the same as in the case of brute force, with the advantage being that we always attain complexity $\mathcal{O}(2^{\min(k,n-k)})$ without having to explicitly compare the dimensions of the primary and dual codes or to apply the MacWilliams identity.

We also address the problem of the minimum distance of self-dual additive codes that represent graphs having rankwidth 1.

We now introduce some basic notation and definitions that are necessary for the further exposition.

Let $GF(4) = \{0, 1, \omega, \omega^2\}$ be the finite field of 4 elements, with $\omega^2 = \omega + 1$. An *additive code* $\mathcal{C}$ over $GF(4)$ of length n is an additive subgroup of $GF(4)^n$. The code C is a free $GF(2)$-module having size 2^k for some $0 \leq k \leq 2n$. In this case, C is called an $(n, 2^k)$ *code*. Being a $GF(2)$-module, it has a basis consisting of k basis vectors. A *generator matrix* of C is simply a $k \times n$ matrix with entries in $GF(4)$ whose rows form a basis of C.

In other words, let C be a code over $GF(4)$ with generator matrix G. If all codewords in C are $GF(2)$ linear combinations of the rows of G, then we say that C is an *additive code*.

There is a natural inner product arising form the trace map. The trace map $\mathrm{Tr} : GF(4) \to GF(2)$ is given by $\mathrm{Tr}(x) = x + x^2$. In particular, $\mathrm{Tr}(0) = \mathrm{Tr}(1) = 0$ and $\mathrm{Tr}(\omega) = \mathrm{Tr}(\omega^2) = 1$. The *conjugate* of an element $x \in GF(4)$, denoted as $\overline{x}$, is the image of x under the Frobenius automorphism $x \mapsto x^2$; in other words, $\overline{0} = 0$, $\overline{1} = 1$, $\overline{\omega} = \omega + 1 = \omega^2$, and $\overline{\omega^2} = \omega$.

We now define the *Hermitian trace inner product* of two vectors $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$ in $GF(4)^n$ as

$$x * y = \sum_{i=1}^{n} \mathrm{Tr}(x_i \overline{y_i}) = \sum_{i=1}^{n} (x_i {y_i}^2 + x_i^2 y_i).$$

If C is an additive code, its *dual*, denoted as $C^\perp$, is the additive code $\{x \in GF(4)^n | x * c = 0 \ , \forall c \in C\}$. If C is an $(n, 2^k)$ code, then $C^\perp$ is an $(n, 2^{2n-k})$ code. The code C is called *(trace) self-orthogonal* if $C \subseteq C^\perp$, and it is called *(trace) self-dual* if $C = C^\perp$. In particular, if C is self-dual, then C is an $(n, 2^n)$ code. We remark that additive self-dual codes exist for any length n since the identity matrix I_n generates a self-dual $(n, 2^n, 1)$ code. Any $GF(4)$-linear code is self-orthogonal under the Hermitian trace inner product if and only if it is a self-orthogonal additive code under the trace inner product [20]. Any linear Hermitian self-dual $[n, k, d]$ code is an additive self-dual $(n, 2^n, d)$ code. For example, the $[6, 3, 4]$ Hexacode [25] is an additive self-dual $(6, 2^6, 4)$ code.

The *weight* of a codeword $c \in C$, is the number of nonzero components of c. The minimum weight d of a code C is the smallest weight of any non-zero codeword of C. If C is an additive $(n, 2^k)$ code with minimum weight d, then C is called an $(n, 2^k, d)$ code. We say that C is a *Type II code* if C is self-dual and all its codewords have even weight. Type II codes of length n exist only if n is even [35]. If C is self-dual but some codewords have odd weight (in which case the code cannot be $GF(4)$-linear), the code is called a *Type I code*. Bounds on the minimum weight of additive self-dual codes of Type I and Type II are given in [126].

Given the weight distribution of a code, it is easy to determine its *minimum distance*. Nonetheless, it is hard to approximate the minimum distance of a given code, and hence the computation of the weight enumerator is also a hard problem [127].

Two self-dual additive codes C_1 and C_2 over $GF(4)$, are said to be *equivalent* if and only if the codewords of C_1 can be mapped onto the codewords of C_2 by a map that preserves self-duality. Such a map consists of a permutation of coordinates,

followed by a scaling of coordinates by elements of $GF(4)$, and, finally, by a conjugation of some of the coordinates. Two equivalent self-dual codes have the same weight enumerator [27].

3.2 Self-dual additive codes over $GF(4)$ and their connection to graphs

In this section, we see how self-dual additive code over $GF(4)$ correspond to structures known as *isotropic systems*, which in turn can be represented as graphs [12]. We recall some basic definitions and concepts from graph theory, define *graph codes* and an operation called local complementation of a graph. The main point of this section is that classifying self-dual additive codes over $GF(4)$ up to equivalence corresponds to classifying the orbits of graphs under local complementation. This simplifies the classification process since graphs are, in general, simpler to handle than codes.

3.2.1 Isotropic systems

Isotropic systems are combinatorial and algebraic structures which unify some properties shared by 4-regular graphs and pairs of dual binary matroids [13].

We define the mapping $\phi : GF(4) \to GF(2)^2$ by $\phi(x) = (\mathrm{Tr}(x\omega^2),\mathrm{Tr}(x))$. Explicitly, we have $0 \to (0,0), 1 \to (1,0), \omega \to (0,1)$ and $\omega^2 \to (1,1)$.

The inverse mapping $\phi^{-1} : GF(2)^2 \to GF(4)$ is defined as $\phi^{-1}(a,b) = a+\omega b$. Let $\boldsymbol{u} \in GF(2)^{2n}$ be written as $\boldsymbol{u} = (\boldsymbol{a}|\boldsymbol{b}) = (a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n)$. The previously defined mapping can be extended to $\phi : GF(4)^n \to GF(2)^{2n}$ by letting $\phi(\boldsymbol{v}) = (\boldsymbol{a}|\boldsymbol{b})$ where $\phi(v_i) = (a_i, b_i)$. Likewise, the inverse mapping $\phi^{-1} : GF(2)^{2n} \to GF(4)^n$ is defined as $\phi^{-1}(\boldsymbol{a}|\boldsymbol{b}) = \boldsymbol{a} + \omega\boldsymbol{b}$.

The *symplectic inner product* of $(\boldsymbol{a}|\boldsymbol{b}), (\boldsymbol{a'}|\boldsymbol{b'}) \in GF(2)^{2n}$ is defined simply as $\langle(\boldsymbol{a}|\boldsymbol{b}), (\boldsymbol{a'}|\boldsymbol{b'})\rangle = \boldsymbol{a}\cdot\boldsymbol{b'} + \boldsymbol{b}\cdot\boldsymbol{a'}$. A subset $\mathcal{I} \subset GF(2)^{2n}$ is called *totally isotropic* if $\langle \boldsymbol{u}, \boldsymbol{v}\rangle = 0$ for all $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{I}$.

Isotropic systems can be represented by simple graphs and can be defined by the row space of a full rank $n \times 2n$ binary matrix $(A|B)$, where $AB^T + BA^T = 0$. Surprisingly, self-dual additive codes over $GF(4)$ are related to isotropic systems by the following theorem.

Theorem 3.2.1. *[27] Every self-dual additive code over $GF(4)$ can be uniquely represented as an isotropic system, and every isotropic system can be uniquely represented as a self-dual additive code over $GF(4)$.*

Example 1. *[27] The row-space of the matrix $(A|B)$ given below defines an isotropic system, while the matrix $C = A+\omega B$ is a generator matrix of the $(6, 2^6, 4)$ Hexacode.*

$$(A \mid B) = \left(\begin{array}{cccccc|cccccc} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right) \quad C = \left(\begin{array}{cccccc} 1 & 0 & 0 & 1 & \omega & \omega \\ \omega & 0 & 0 & \omega & \omega^2 & \omega^2 \\ 0 & 1 & 0 & \omega & 1 & \omega \\ 0 & \omega & 0 & \omega^2 & \omega & \omega^2 \\ 0 & 0 & 1 & \omega & \omega & 1 \\ 0 & 0 & \omega & \omega^2 & \omega^2 & \omega \end{array}\right)$$

3.2.2 Graph representation

Definition 3.2.1. A graph is a pair $G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of *vertices*, or *nodes*, and $E \subseteq V \times V$ is a set of distinct pairs of elements (v_i, v_j) called *edges*.

A graph with n vertices can be represented by an $n \times n$ *adjacency matrix* Γ, where the non-diagonal elements γ_{ij} of the matrix satisfy $\gamma_{ij} = 1$ if $(i, j) \in E$, and $\gamma_{ij} = 0$ otherwise. We only consider *simple undirected* graphs whose adjacency matrices are symmetric and whose diagonal elements are all equal to 0.

The *open neighborhood* of $v \in V$ denoted by $N(v)$, is the set of vertices connected to v by an edge. The *closed neighborhood* $N[v]$, is the set of all vertices connected to v by an edge along with v itself, i.e. $N[v] = N(v) \cup \{v\}$.

The number of edges incident on v is called the *degree* of the vertex v. A vertex of degree 1 is called a *pendant*.

Definition 3.2.2. A pair of vertices u, v are called *true twins* if $N[u] = N[v]$ and are called *false twins* if $N(u) = N(v)$. We call a pair of vertices a *twin-pair* if they are true twins or false twins.

A *subgraph* H of a graph G is obtained by deleting vertices and edges from G. A subgraph H is called *induced* if H can be obtained from G by only deleting vertices from G together with all edges incident on the deleted vertices.

Definition 3.2.3. Two graphs $G = (V, E)$ and $G' = (V', E')$ are *isomorphic* if there exists a bijection $f : V \to V'$ between the vertex set of G and the vertex set of G' such that two vertices u, v are adjacent in G if and only if $f(u), f(v)$ are adjacent in G'.

The *complement*, or *inverse*, of G is a graph H on the same vertex set as G with the property that two distinct vertices are adjacent in H if and only if they are not adjacent in G.

Having recalled the basics of graph theory, we are now ready to introduce graph codes.

Definition 3.2.4. A *graph code* is an additive code over $GF(4)$ that has a generator matrix of the form $C = \Gamma + \omega I$, where I is the identity matrix, Γ is the adjacency matrix of a simple undirected graph, and ω is a primitive element of $GF(4)$.

A graph code is always self-dual, since its generator matrix has full rank over $GF(2)$ and $C\overline{C}^T$ only contains entries from $GF(2)$ whose trace must be zero. This construction for self-dual additive codes over $GF(4)$ has also been used in [123].

In the case of additive codes, the converse implication holds as well. The following theorem was first proved by Bouchet [12] and later by Van den Nest, Dehane and Moor [74] in the context of isotropic systems.

Theorem 3.2.2. *[12, 74] Every self-dual additive code over $GF(4)$ is equivalent to a graph code.*

The generator matrix of a self-dual additive code over $GF(4)$ corresponds to an $n \times 2n$ binary matrix $(A|B)$, such that $C = A + \omega B$. The row space of $(A|B)$, denoted $\mathcal{I}$, defines an isotropic system. The outline of the proof of Theorem 3.2.2 is to show that $\mathcal{I}$ is generated by $(\Gamma|I)$, where I is the identity matrix and Γ is the adjacency matrix of a simple undirected graph.

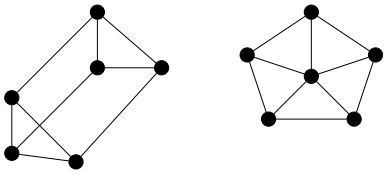


Figure 3.1: Two graph representation of the Hexacode

Example 2. *[27] Let $C = A + \omega B$ be the generator matrix of the $(6, 2^6, 4)$ Hexacode given in Example 1. Using the proof of Theorem 3.2.2, one can find $C' = \Gamma + \omega I$, which generates an equivalent graph code.*

$$(A \mid B) = \left(\begin{array}{cccccc|cccccc} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right)$$

$$(\Gamma | I) = \left(\begin{array}{cccccc|cccccc} 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right)$$

$$C = \begin{pmatrix} \omega & 0 & 1 & 0 & 1 & 1 \\ 0 & \omega & 1 & 1 & 0 & 1 \\ 1 & 1 & \omega & 0 & 0 & 1 \\ 0 & 1 & 0 & \omega & 1 & 1 \\ 1 & 0 & 0 & 1 & \omega & 1 \\ 1 & 1 & 1 & 1 & 1 & \omega \end{pmatrix}$$

Consider a graph $G = (V, E)$ and its corresponding adjacency matrix Γ. The matrix $C = \Gamma + \omega I$ is then the generator matrix for a graph code. Swapping vertex i and vertex j of the graph G can be accomplished by exchanging column i and column j of Γ and then exchanging row i and row j of Γ. We denote the resulting matrix by Γ'. Exactly the same column and row operations map $C = \Gamma + \omega I$ to $C' = \Gamma' + \omega I$. The matrices C and C' generate equivalent codes. It follows that two codes are equivalent if their corresponding graphs are isomorphic.

Figure 3.1 shows two graph representation of the $(6, 2^6, 4)$ Hexacode. From Theorem 3.2.2 and Example 2, one can see that every graph represents a self-dual additive code over $GF(4)$, and every self-dual additive code over $GF(4)$ can be represented by a graph.

3.2.3 Local complementation

We are now ready to define the operation referred to as local complementation.

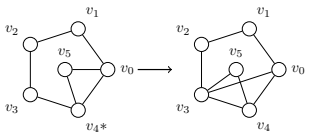


Figure 3.2: Local complementation at v_4

Definition 3.2.5. Given a graph $G = (V, E)$ and a vertex $v \in V$, the *local complementation (LC)* of G on v transforms G into G^v by replacing the induced subgraph of G with vertex set $N(v)$ by its complement.

Example 3. *Consider the graph shown in Figure 3.2. Local complementation is performed on the vertex v_4. We see that the open neighbourhood of v_4 is $N(v_4) = \{v_0, v_3, v_5\}$ and there are no edges between the neighbours $\{v_0, v_3\}, \{v_3, v_5\}$, while there is an edge between $\{v_0, v_5\}$. The effect of local complementation on v_4 results in the introduction of an edge between $\{v_0, v_3\}$ and between $\{v_3, v_5\}$, as well as the removal of the edge between $\{v_0, v_5\}$.*

The application of local complementation to graphs naturally leads to a notion of equivalence.

Definition 3.2.6. If a graph G' can be obtained from a graph G by a series of local complementation operations, we say that G and G' are *LC-equivalent*.

The codes corresponding to a given graph and to any graph obtained from the first by local complementation are equivalent, as shown in the following theorem. This leads to a useful characterization of equivalent codes in terms of their corresponding matrices.

Theorem 3.2.3. *[27] Let Γ be the adjacency matrix of the graph $G = (V, E)$ and Γ^v be the adjacency matrix of G^v after applying LC on $v \in V$. The codes generated by $C = \Gamma + \omega I$ and $C' = \Gamma^v + \omega I$ are equivalent.*

Theorem 3.2.4. *[27] Two self-dual additive codes C and C' over $GF(4)$, with corresponding graphs G and G', respectively, are equivalent if and only if there is a finite sequence of not necessarily distinct vertices $(v_1, v_2, \ldots, v_i)$ such that $(((G^{v_1})^{v_2})^{\cdots})^{v_i}$ is isomorphic to G'.*

3.2.4 Algorithm for the classification of self-dual additive codes of general graphs

In light of Theorem 3.2.4, finding the graphs that can be obtained from a given graph by means of local complementation is instrumental to classifying self-dual additive codes. This naturally leads to the following definition.

Definition 3.2.7. The *LC orbit* of a graph is the set of all unlabelled graphs that can be obtained by performing any sequence of *LC operations* on G.

It follows from Theorem 3.2.4 that two self-dual additive codes over $GF(4)$ are equivalent if and only if their graph representations are in the same LC orbit. The LC orbit of a graph can be generated by a recursive algorithm.

An algorithm for classifying self-dual additive codes corresponding to general graphs is given in [27]. It is a recursive algorithm that uses the result of classification on $n-1$ vertices to obtain a classification on n vertices. Let G_n be the set of all unlabelled simple undirected connected graphs on n vertices. Connected graphs correspond to *indecomposable* codes, i.e codes that cannot be written as the *direct sum* of two codes of smaller length.

The set of all distinct LC orbits of connected graphs on n vertices is a partitioning of G_n into i_n disjoint sets. The number i_n is also the number of indecomposable self-dual additive codes over $GF(4)$ of length n up to equivalence.

Let L_n be the set containing one representative from each LC orbit of connected graphs on $n-1$ vertices corresponding to equivalent additive self-dual codes. The recursive algorithm can then be described as follows.

- Compute a set of graphs E_n by adding a vertex to each graph in L_{n-1}. This can be done in $2^{n-1}-1$ ways since the resulting graph must be connected. All of the $2^{n-1}-1$ graphs obtained from one graph in L_{n-1} are added to E_n. The set E_n will contain at least one representative from each LC orbit of the connected graphs on n vertices.
- For each set of isomorphic graphs, keep only one graph in E_n.
- Use weight-enumerators to partition the set E_n, i.e. graphs corresponding to codes with the same weight-enumerator are put in the same class. It is important to note that codes with different weight distributions can never be equivalent.
- Partition each class in E_n by checking for self-dual equivalence.
- Compute L_n by taking one graph from each class in E_n.
- Output L_n.

3.3 Classification of self-dual additive codes corresponding to graphs of rankwidth 1

Classifying self-dual additive codes over $GF(4)$ by considering all possible graphs is hard. This is evident from the previously performed classification [27] where self-dual codes over $GF(4)$ have been classified only for n up to 12. Focusing on a particular sub-class of graphs naturally leads to a more tractable classification problem. In this section, we initiate the study of self-dual codes whose corresponding graphs have rankwidth 1. The rankwidth of a graph is preserved under local complementation. The class of graphs with rankwidth equal to 1 is exactly the class of distance-hereditary graphs (defined later). We show that by combining the structural properties of these graphs with the algorithm used in [27], the classification of the corresponding codes becomes significantly faster. Determining how far we can take this procedure, i.e. up to what value of n we can still apply it efficiently, remains a question for future work.

There are two computationally heavy steps in the general classification algorithm: testing whether two given graphs are isomorphic to each other, and computing the weight enumeration of a given code.

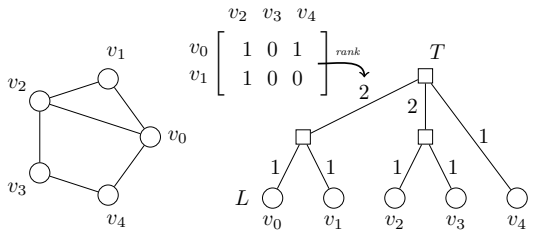


Figure 3.3: A rank decomposition (T, L) of a graph. The rankwidth of T is 2. It so happens that the rankwidth of this graph is 2 as well.

For a fixed k, testing graph isomorphism for graphs of rankwidth k is polynomial in the size of the graph [42], and it is in fact linear in n for graphs of rankwidth 1 [124]. Hence, approaching the problem of classification of such codes from the point of view of rankwidth can lead to significant computational advantages.

Computing the weight-enumerator in general is essentially a brute-force search with complexity $\mathcal{O}(2^k)$. If $k > n/2$ it is beneficial to compute the weight-enumerator of the dual code instead and to then apply the MacWilliams identity. In our work, we take a different approach, and use binary decision diagrams (BDD) to compute the weight-enumerators. The algorithm using BDD for weight enumeration has similar complexity to brute force, but has the benefit that we automatically get complexity $\mathcal{O}(2^{\min\{k,n-k\}})$ without having to explicitly examine the dimensions of the primary and dual codes, or to apply the MacWilliams identity.

The minimum distance of codes corresponding to distance-hereditary graphs is 2. We show that the minimum distance of a code is at least 4 if and only if the corresponding graph does not contain any pendant vertex or any twin-pairs.

3.3.1 Rankwidth

Let n be a positive non-zero integer. Let $\mathcal{T}_n$ be the set of all trees with the following properties:

1. every tree T in $\mathcal{T}_n$ has exactly n leaves;

2. every internal node of every tree T in $\mathcal{T}$ has degree exactly equal to 3.

Let G be a simple undirected graph with n vertices. A *rank decomposition* of G is a pair (T, L), where $T \in \mathcal{T}_n$ is such that there is a bijection between the set of leaves L of T to the set of vertices V of G. Clearly, for any given tree T, there are $n!$ bijections. See Figure 3.3 for an example.

The rankwidth of a graph G can be computed as follows. Suppose $T \in \mathcal{T}_n$ provides a rank decomposition (T, L) of G. Let e be any edge of T. Removing e from the set of edges of T yields two sub-trees, T_a and T_b, which naturally give a partition $V = A \cup B$ of the vertex set of T: more precisely, A is the set of vertices of T_a, and B is the set of vertices of T_b. We now construct a bipartite graph $((A, B), E)$ in which (e_a, e_b) is an edge for some $e_a \in A$ and $e_b \in B$ if and only if (e_a, e_b) is an edge in G. The rank of the adjacency matrix of this bipartite graph is the *weight* of the edge e. The *rankwidth of the rank decomposition* (T, L) is the maximum weight

of an edge in T. Finally, the *rankwidth of* G is the minimum rankwidth of any rank decomposition of G.

An important property of the rankwidth of a graph is that it remains invariant under local complementation.

Theorem 3.3.1 ([90]). *Given a graph G and a vertex v of G, $rw(G) = rw(G^v)$.*

The *distance* between two nodes of a graph is the length of the shortest path between them. A *distance-hereditary* graph is a graph in which the distances in any connected induced subgraph are the same as in the original graph.

The following theorem provides the connection between distance-hereditary graphs and the rankwidth.

Theorem 3.3.2 ([90]). *A graph G is distance-hereditary if and only if the rankwidth of G is at most* 1.

Recall that in the recursion step of the algorithm for computing L_n described in Section 3.2.4, we expand the graphs from L_{n-1} to graphs on n vertices by adding a new vertex and connecting it in all possible ways to to each graph in L_{n-1}. However, adding a vertex to a graph in this way does not, in general, preserve the property of being distance-hereditary. Nonetheless, as shown in the following theorem, it is possible to characterize the cases in which the property of being distance-hereditary is indeed preserved.

Theorem 3.3.3 ([8]). *Let G be a finite graph with at least two vertices. Then G is distance-hereditary if and only if G is obtained from an edge by a sequence of one of the following vertex extensions: adding the vertex as a pendant, adding the vertex as a true-twin to an existing vertex, or adding the vertex as a false-twin to an existing vertex.*

Let $\mathcal{G}_{n-1}$ be the set of all connected graphs of rankwidth 1 on $n-1$ vertices. Then $\mathcal{G}_n$ can be obtained by adding a vertex to each graph in $\mathcal{G}_{n-1}$ as a pendant or a twin to some vertex. Consider $\mathcal{C}$ to be the LC orbit of a graph $G \in \mathcal{G}_{n-1}$. Let $G_1, G_2 \in \mathcal{C}$. Then there is a sequence of local complementation operations S that can take G_1 to G_2. Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be the $3(n-1)$ extensions of G_1 and G_2 obtained via adding a node either as a pendant or a twin pair. We show that by applying S on any graph in $\mathcal{E}_1$, we end up with a graph in $\mathcal{E}_2$, implying that $\mathcal{E}_1$ and $\mathcal{E}_2$ are *LC-equivalent*.

Let u be a new vertex added to G_1 as a pendant or twin to a vertex v of G_1. The LC operations at vertices in G_1 switch the role of u relative to v as a pendant or a twin. At the same time, G_1 changes to G_2 after S has been performed. Then, u can be seen as being attached to G_2 as a pendant or twin (according to what happens after applying S to $G_1 + u$). Hence, any graph $\mathcal{E}_2$ can be seen as being obtained from a graph in $\mathcal{E}_1$ via applying S. This implies that instead of considering extensions of $\mathcal{C}$, we need only consider extensions of just one representative from $\mathcal{C}$. Let L_{n-1} be the set of representatives of all orbits in $\mathcal{G}_{n-1}$.

In the construction of E_n from L_{n-1} in the algorithm, let us add the new vertex only in one of the three different ways described in Theorem 3.3.3. There are at most $3(n-1)$ ways to do that. Therefore, instead of branching in $2^{n-1}-1$ ways, we need only branch in at most $3(n-1)$ ways. Since rankwidth is preserved by LC operations, and adding a new vertex as a pendant, true twin or false twin to

an existing vertex in L_{n-1} preserves the property of being distance hereditary, the graphs in the sets L_{n-1}, E_n and L_n can all be assumed to have rankwidth 1. Due to this, testing isomorphism can be performed in linear time.

The algorithm for classifying self-dual codes over $GF(4)$ corresponding to graphs of rankwidth 1 is similar to the algorithm defined in the previous section, with some changes highlighted below.

In the following exposition, we assume the same notation as in the description of the algorithm described in Section 3.2.4:

- Compute the set of graphs E_n by adding a vertex to each graph in L_{n-1} in $3(n-1)$ ways.
- For each set of isomorphic graphs, keep only one graph in E_n.
- Using BDD, compute the weight-enumerators of the codes corresponding to the graphs in E_n.
- Use these weight-enumerators to partition E_n into equivalence classes.
- Partition each class in E_n by checking for self-dual equivalence.
- Compute L_n by taking one graph from each equivalence class computed in the previous step.
- Output L_n.

In the next section, we shall discuss how to compute weight enumerators using Binary Decision Diagrams. Since BDD have been extensively discussed in Chapter 2 Section 2.4, we only look at the details of how to use them for computing weight enumerators.

3.4 Computing weight enumerators using BDD

One of the most important properties of any code is its weight distribution, which, among other things, allows one to compute the error probability of the code. Recall that the weight distribution of a code C is given by the bivariate polynomial

$$W_c(x,y) = \sum_{u \in C} x^{n-wt(u)} y^{wt(u)} = \sum_{i=0}^{n} A_i x^{n-i} y^i, \tag{3.1}$$

where A_i denotes the number of codewords of weight i in C and $wt(u)$ denotes the weight of u.

As described above, in our algorithm for classifying self-dual additive codes, we use BDD to compute their weight-enumerators. In this section, we take a closer look at how this approach works.

3.4.1 Construction of BDD

Let G be an $n \times n$ generator matrix of a self-dual additive code C over $GF(4)$. We compute the set of all codewords by considering all possible linear combinations over $GF(2)$ of the rows of G. This is done by first expanding the matrix into an $n \times 2n$ matrix G' over $GF(2)$ by mapping each element of $GF(4)$ to a pair of elements in $GF(2)$ as follows: $0 \mapsto (00)$, $1 \mapsto (01)$, $\omega \mapsto (10)$, $\omega^2 \mapsto (11)$. For example, consider the matrix

$$G = \begin{bmatrix} \omega & 1 \\ 1 & \omega \end{bmatrix}_{n \times n}$$

for $n = 2$. Under the mapping defined above, G becomes

$$G' = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}_{n \times 2n}$$

Now we multiply all binary strings $(c_1, c_2, c_3, \ldots, c_n)$ of length n with the matrix G' to get the set of all codewords $(x_1, x_2, x_3, \ldots, x_{2n-1}, x_{2n})$. Continuing the example above, we can obtain all the codewords by considering the products

$$(c_1, c_2) \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} = (x_1, x_2, x_3, x_4)$$

for all possible values $c_1, c_2 \in GF(2)$.

Recall that every path in a BDD corresponds to a binary vector. In order to construct a BDD that has paths corresponding to all possible codewords, we consider the parity check matrix H of the code. From coding theory, we know that the parity check matrix describes a set of linear relations that the coordinates of each codeword must satisfy. More precisely, a vector x is a codeword of a code C if and only if $xH^T = 0$, where H is the parity check matrix of C.

Let the linear equations given by $xH^T = 0$ be $l_i(x) = 0$ for $1 \leq i \leq n$. A BDD that represents all codewords of the code, i.e. vectors x satisfying $l_i(x) = 0$ for $1 \leq i \leq n$, is given on the left-hand side of Figure 3.4. In that BDD, x_{i_j} are free variables such that no l_i can be written as a sum of x_{i_j}. In other words, considering the free variables as linear combinations too, all linear combinations in the BDD are independent. The free variables can take any value, and l_i represent linear combinations of the coordinates that must be 0 in order to be a codeword.

In the next step, we apply add and swap operations (described in Algorithm 1 and Algorithm 2 of Chapter 2) to this basic BDD in order to resolve the linear combinations l_i. By resolving the linear combinations we mean that we add together some of the linear combinations and free variables in order to transform l_i into a single variable. Recall that the add operation can only be applied to two adjacent levels. If it is necessary to perform addition on two levels are not adjacent, we first apply the swap operation in order to move the corresponding variables and make them adjacent.

We keep applying these operations until each level only contains single variables. We are effectively performing Gaussian elimination with back substitution on the set of linear combinations. We sort the levels in such an order that x_1 appears on the top and x_{2n} appears on the lowest level, as shown in the BDD on the right-hand side of Figure 3.4. We give a more detailed example of how the add and swap operations

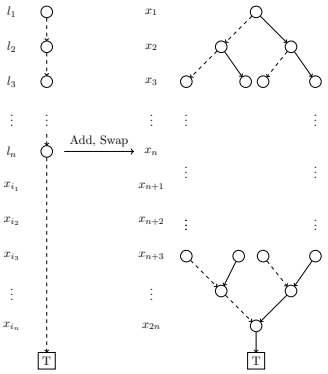


Figure 3.4: BDD after adding and swapping to resolve all linear combinations into single variables, sorted on the levels.

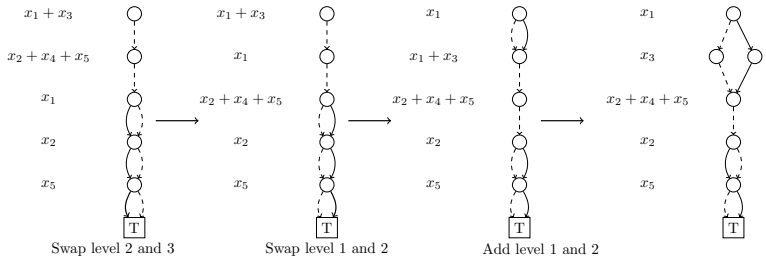


Figure 3.5: Example: Performing add and swap to a BDD to resolve $x_1 + x_3 = 0$.

can be applied to a BDD in Figure 3.5. In the end, the paths of the resulting BDD represent all codewords in C.

In this chapter, we mostly focus on the special case of codes of length $2n$ and dimension n. However, constructing the BDD representing a binary code can be done for codes of any length and dimension. Here we determine the complexity, expressed in terms of the number of nodes in the final BDD, for any $[n, k]$ linear code C over $GF(2)$.

We begin by resolving a natural question, namely, by giving an upper bound on the number of nodes in a given level in the final BDD.

Lemma 3.4.1. *The number of nodes on any level of the final BDD after resolving all linear combinations for a code C is at most 2^k.*

Proof. The number of codewords in C is 2^k, and so the total number of paths in the BDD is also 2^k. There are no edges between nodes on the same level, so all nodes on any level are part of different paths. Hence the number of nodes on any level can not be more than 2^k. □

Lemma 3.4.2. *The number of nodes on any level of the final BDD after resolving all linear combinations for a code C is at most 2^{n-k}.*

Proof. The number of nodes on any level of the basic BDD before resolving any linear combinations is 0 or 1. Applying the swap or add operation will at most double the number of nodes on the lower of the affected levels. We resolve one linear combination by adding together certain levels in the BDD. Starting with the lowest level and moving levels upwards, adding as needed, we see that each level is involved in the resolution of a linear combination only once. So the number of nodes on any level of the BDD when resolving one linear constraint will at most double. Since we are resolving $n - k$ linear combinations, the number of nodes on any level in the final BDD will be at most 2^{n-k}. □

Combining Lemmas 3.4.1 and 3.4.2, we get the following result.

Theorem 3.4.3. *The number of nodes in the final BDD representing the codewords of a binary linear $[n, k]$ code is of order $\mathcal{O}(2^{\min\{k,n-k\}})$.*

3.4.2 Algorithm for computing weight enumerators

Recall that pairs of coordinates $(x_{2i-1}, x_{2i}) \in GF(2)^2$ can be interpreted as elements in $GF(4)$. A path in the BDD with resolved and sorted levels has length $2n$, but represents a codeword of length n with elements from $GF(4)$. When computing the weight enumeration, we therefore count how many non-zero $GF(4)$ elements are on the corresponding path. Figure 3.6 shows how the different elements of $GF(4)$ are represented as paths in the BDD, and Figure 3.7 gives an example of how to count the weight of a codeword represented as a path. In both figures, a dashed line representes a 0-edge in the BDD, while a solid line represents a 1-edge.

Now we describe our algorithm for computing weight enumerators using BDD. We explain the whole process, where we start with a graph G, and want to compute the weight enumeration of its corresponding self-dual code over $GF(4)$.

Step 1: Given the adjacency matrix Γ of a graph G, we obtain the generator matrix $G = \Gamma + \omega I$ over $GF(4)$.

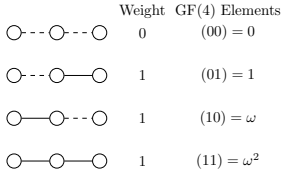


Figure 3.6: Weight of consecutive edges in a BDD

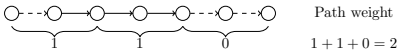


Figure 3.7: Example of computing weight of a path

Step 2: We transform the generator matrix over $GF(4)$ to a matrix over $GF(2)$ by mapping each element in $GF(4)$ to a pair of elements in $GF(2)$: $0 \mapsto (00)$, $1 \mapsto (01)$, $\omega \mapsto (10)$, $\omega^2 \mapsto (11)$.

Step 3: We obtain the parity check matrix H from the generator matrix G over $GF(2)$ and get the parity check equations $l_1 = l_2 = \ldots = l_n = 0$.

Step 4: Construct the BDD for $l_1 = l_2 = \ldots = l_n = 0$ with $x_{i_1}, x_{i_2}, \ldots, x_{i_n}$ as free variables.

Step 5: Apply add and swap operations to the BDD to resolve and sort the linear combinations (see Figures 3.4 and 3.5).

Step 6: Computing the weight enumerator: Associate a vector of length $(n+1)$ of integer values, denoted as $(p_0, p_1, p_2, \ldots, p_n)$, to each node. For a given node, p_i indicates the number of paths of weight i below this node.

1. Start by setting $(1, 0, 0, \ldots, 0)$ as the vector for the true-node at the bottom. We say there is one path of weight 0 from the true-node to itself (the empty path).
2. We compute the vectors for the other nodes recursively, from the levels at the bottom to the ones at the top. Consider a pair of edges from a node T to a node A below it. The weight distribution vector of A contributes to the computation of the weight distribution vector of T as follows. If this pair of edges contributes 0 to the path weight, the weight distribution below T along this path is the same as for A. When the pair of edges from T to A contribute 1 to the weight, the paths of weight i below A become paths of weight $i + 1$ below T. Hence the weight enumeration vector for T is obtained by shifting the vector for A by one position to the right, as shown in Figure 3.8.
3. Having computed the contribution of all edges leading from a node T (as described in the previous step), we compute the weight distribution vector of T by summing them together. This is shown in Figure 3.9.
4. Assuming all weight distribution vectors have been computed for the nodes on one level, we compute the weight distribution for the nodes two levels above by adding all the weight contributions, shifting them by one position to the right as needed. This is shown in Figure 3.9.

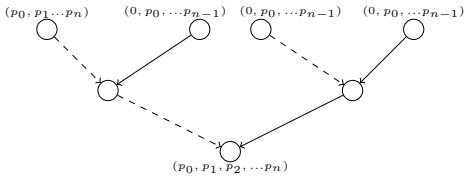


Figure 3.8: Shift weight vector by one to the right when the pair of edges indicate a non-zero $GF(4)$-element.

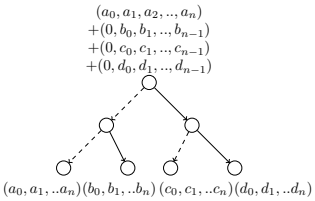


Figure 3.9: Computing weight distribution for all paths below a node.

5. Once the algorithm terminates, the weight distribution vector associated with the root node gives the weight distribution of the code.

The complexity of computing the weight enumeration of a given code represented as a BDD is $\mathcal{O}(N)$, where N is the number of nodes in the BDD, and adding two integer vectors counts as a unit operation. In terms of single integer additions, the complexity is $\mathcal{O}(nN)$.

We have described the algorithm for computing the weight enumeration for codes over $GF(4)$. Going back to the case of an $[n, k]$ linear code over $GF(2)$, we can easily modify the algorithm to compute the weight distribution for any binary linear code when it is represented as paths in a BDD.

As already mentioned, computing the weight distribution of a code is a hard problem [127] and, in general, the best known solution is by brute force which has complexity $\mathcal{O}(2^{\min(k,n-k)})$, where k is the dimension of the code. In light of this, it is not surprising that applying our BDD approach to weight enumeration yields the same complexity. Nonetheless, in order the achieve the complexity mentioned above by brute force, one has to explicitly compare the dimensions of the primary and dual code and apply the MacWilliams identity in the case than $k > n/2$. Our approach has the advantage that it automatically achieves this complexity without having to consider the dimension of the code and its dual.

3.5 Minimum distance

By far one of the most important properties of any code is its minimum distance, which determines its error-correcting capabilities. Glynn et al. [40] showed that the minimum distance of a code is equal to one plus the minimum vertex degree over all graphs in the corresponding LC orbit.

In the following, we summarize some relations between the framework that we developed in the preceding sections and the problem of computing the minimum

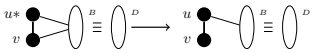


Figure 3.10: Here filled circles denote vertices and ellipses denote set of vertices. An edge from a vertex to a set denotes that vertex is adjadent to every vertex in the set. The vertices u and v are true twins. After LC at u, the degree of v reduces to 1.

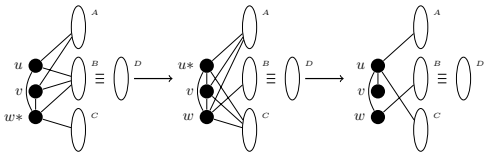


Figure 3.11: The vertices u and v are false-twins and w is a common neighbor of u and v. After LC operation at w followed by LC operation at u reduces the degree of v to 1.

distance of a given code.

Lemma 3.5.1. *If a connected graph contains a twin-pair, then the minimum distance of the corresponding code is 2.*

Proof. Let u, v be a true-twin pair. Then, after applying a LC operation to u, the degree of v in the resulting graph is 1. See Figure 3.10.

If u, v is a false-twin pair, and the degree of u and the degree of v are both 1, then the minimum distance of the corresponding code is trivially 2. In the case when u, v are not pendants, then after applying an LC operation to a common vertex w, they become true-twins in the resulting graph. Then, as in the above case, applying an LC operation at u produces a pendant. See Figure 3.11. Since LC operations preserve connectivity, this is the minimum possible degree of a vertex over the entire LC orbit of the graph. Hence, the minimum distance of the corresponding code is 2. □

Lemma 3.5.2. *Codes with corresponding graphs of rankwidth 1 have minimum distance 2.*

Proof. The graphs of rankwidth 1 are exactly the distance hereditary graphs which can be constructed recursively by adding a pendant or a twin-pair to an edge. Such graphs will have either a pendant or a twin-pair. Hence, by Lemma 3.5.1, codes with corresponding graphs of rankwidth 1 have minimum distance 2. □

Lemma 3.5.3. *If a graph contains a twin-pair, then every graph in its LC orbit will contain a twin-pair or a pendant.*

Proof. Let u and v be true-twin pairs. An LC operation at either u or v will yield a pendant. An LC operation at $w \notin N(u)$ does not effect $N(u)$, hence, u and v will remain twin pairs. Take some $w \in N(u) \setminus \{v\}$. Now, after applying an LC to w, the nodes u and v become false-twins. See Figure 3.12.

Now, suppose u and v are false-twins. They become true-twins if an LC operation is performed on w, see Figure 3.11. An LC operation at either u or v or at a vertex not in their common neighborhood does not change the neighborhood of u or v and hence, they remain false-twins. □

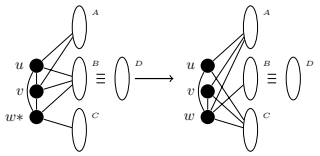


Figure 3.12: The vertex w is in the common neighborhood of true-twins u and v. LC at w makes u, v into a false-twin pair.

Lemma 3.5.4. *If G does not have a pendant or a twin-pair, then no graph in the LC orbit of G will have a twin-pair.*

Proof. We prove this by contradiction. Suppose that G does not contain a twin-pair or pendant, but applying an LC operation at a node u of G results in a graph containing a twin-pair v, w. By Lemma 3.5.3, all graphs in the LC-orbit of the resulting graph must have either a pendant or a twin-pair. Since G lies in this LC-orbit as well, it must have a pendant or twin-pair, contradicting the assumption. □

Combining Lemma 3.5.1, Lemma 3.5.3 and Lemma 3.5.4, and noting that Type II codes must have even weight, gives the following theorem.

Theorem 3.5.5. *The minimum distance of a self-dual additive code over $GF(4)$ is at least 4 if and only if the corresponding graph G has no pendants or twin-pairs.*

3.6 Conclusion

In this chapter, we introduce a new approach for classifying self-dual additive codes over $GF(4)$ whose corresponding graphs have rankwidth 1. We show that this approach provides a significant speed-up in comparison to previously applied procedures. Graphs of higher rankwidth remain an object of ongoing research. It would be interesting to study graph classes that can be constructed recursively in a manner similar to distance-hereditary graphs. It remains to experimentally determine up to what values of n this procedure can be used efficiently in practice.

We note that our use of BDD in the computation of the weight-enumerators still results in exponential complexity. Since the minimum distance can be obtained from the weight-enumerator, and it is known that the computation of the minimum distance is a hard problem, we can not hope to achieve a better complexity than the existing one. Potential optimizations to the algorithm will be of future interest. Furthermore, we have characterized the graphs having minimum degree at least 2 over their entire LC-orbit.

In light of the observed results on the the minimum distance of a code, it is interesting to discover and study other classes of graphs having high minimum degree.

Chapter 4

Reducing Lattice Enumeration Search Trees

4.1 Introduction

A lattice is a geometric object that can be described as an infinite, regular (not necessarily orthogonal) n-dimensional grid in $\mathbb{R}^n$. Despite their simplicity, lattices possess a rich combinatorial structure which has attracted mathematicians over the last two centuries. Lattices have numerous application in mathematics and computer science, ranging from number theory and Diophantine equations to cryptography.

The study of lattices in cryptography was marked by two major breakthroughs:

(i) The invention of the well-known LLL lattice reduction algorithm by Lenstra, Lenstra and Lovász in the early 80s [69], which runs in polynomial time and finds an approximate solution to many classical problems in computer science. These include solving integer programs in a fixed number of variables [71], factoring polynomials over the rationals [69], breaking knapsack based cryptosystems [88], finding solutions to many Diophantine equations [43], and the cryptanalysis of special cases of RSA and DSA [86, 81].

(ii) Ajtai's discovery of a connection between the worst-case and average-case hardness of certain lattice problems in the 90s [2]. Until that point, lattices in cryptography were primarily used as tools in cryptanalysis. In complexity theory, we say that a problem is hard if it is hard for the worst case instances; this is in contrast to cryptography, where cryptographic constructions are based on average case assumptions. Ajtai showed how to build cryptographic functions which are as hard to break in the average case as it is to solve the worst case instance of a certain lattice problem.

    This type of approach kick-started the study of lattices from the point of view of computational complexity, and, most importantly, led to the study and resolution of long standing open problems, for example, the NP-hardness of the shortest vector problem in its exact and approximate versions [87].

The main computational problem associated with lattices is the shortest vector problem. In this problem, we are given a lattice represented by a fixed set of basis vectors as input, and our goal is to output the shortest non-zero vector in the lattice. There are two main algorithmic techniques for solving lattice problems. The first

technique is *lattice basis reduction*, which started with the famous LLL algorithm and was further developed to block-wise reduction algorithms such as the BKZ algorithm [87]. The main goal of lattice reduction is to transform the vectors of the given lattice basis into a basis whose vectors have shorter length and the angles between them are as close to orthogonal as possible. The LLL algorithm runs in polynomial time, but its approximation factor (that is, the factor expressing the difference between the norm of the actual shortest vector and the one found by the algorithm) is asymptotically exponential. The second, and more basic, approach, which is the focus of our work, is the *enumeration technique* which started with Pohst [94], Fincke–Pohst [33, 34] and Kannan [56] and is still an active area of research.

Lattice enumeration is an exhaustive search to find the best integer combination of the basis vectors such that the resulting vector has minimal norm. This is done by searching all lattice points in a bounded region (typically an r-dimensional ball). Enumeration can be seen as searching for a short lattice vector in a depth first search tree, where the leaves correspond to the lattice vectors and the internal nodes correspond to the partial assignments of integers to the coefficients in the combination. From every node in the tree an interval, or range, of possible integer values for the coefficients is computed, and the algorithm branches from that node by trying all values from the range in order to find the shortest vector. The standard enumeration technique has complexity $2^{\mathcal{O}(n^2)}$, where n is the dimension of the lattice [109].

In this chapter, we propose two ideas to shorten the ranges of the coefficient vectors, thereby aiming to reduce the search space.

(i) *Hybrid enumeration:* in this technique, we compute all possible intervals for the remaining coefficients (the coefficients that have not yet been branched for) for a particular node, and choose the shortest one to branch for. This is a greedy approach that always attempts to minimize the number of nodes in the next level of the search tree.

(ii) *Signed enumeration:* in this technique, we estimate the sign (plus or minus) of the integer values in the computed range. This is a probabilistic approach and, depending on the confidence measure of the estimated sign, we prune away some values that have the opposite sign in the interval, thereby reducing the intervals and, consequently, the size of the search tree.

The state of the art concerning the enumeration technique is lattice enumeration using extreme pruning [37]. Since enumeration is expensive, extreme pruning attempts to avoid enumerating all the tree nodes by discarding certain branches. The performance of pruned enumeration leads asymptotically to an exponential speed up of about $2^{\frac{n}{2}}$ compared to standard enumeration, while maintaining a success probability of finding the shortest vector of at least 95%.

A natural question is then, how much does our proposed method, which we call hybrid enumeration, increase the computational efficiency of the standard enumeration technique. The complexity of the latter after LLL reduction of the basis vectors is $2^{\mathcal{O}(n^2)}$ [109]. Based on our experimental results, we can see that our technique results in a reduction of the number of nodes, thus making the full enumeration of the lattice faster. The complexity of the hybrid enumeration algorithm has the

same asymptotic complexity as standard enumeration, i.e. $2^{\mathcal{O}(n^2)}$. Unfortunately, this does not yield an exponential speed up compared to extreme pruning, which, for the time being, remains optimal.

Definition 4.1.1. Let m and n be positive integers with $m \geq n$, and let $\boldsymbol{B} = \{\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n\}$ be a set of n linearly independent vectors in $\mathbb{R}^m$. A *lattice* in $\mathbb{R}^m$ is the set $\mathcal{L}(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n) = \{\sum_{i=1}^{n} x_i \boldsymbol{b}_i : x_i \in \mathbb{Z}\}$ of all integer combinations of $\{\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n\} \in \mathbb{R}^m$.

Alternatively, lattices can be defined as discrete additive subgroups of $\mathbb{R}^m$.
Let us now consider some examples of lattices and non-lattices.

- The singleton set $\{\mathbf{0}\} \subset \mathbb{R}^m$ is a lattice.
- The set of integers $\mathbb{Z} \subset \mathbb{R}$ forms a 1-dimensional lattice, and the set $\mathbb{Z}^m = \{(x_1, x_2, \ldots, x_m) : x_i \in \mathbb{Z}\}$ forms a lattice in $\mathbb{R}^m$.
- For a lattice L, its scaling $cL = \{c\boldsymbol{x} : \boldsymbol{x} \in L\}$ by any real number c is a lattice. More generally, any linear transformation applied to a lattice results in a lattice.
- The set of all rationals $\mathbb{Q}$ is *not* a lattice in $\mathbb{R}$ as it is not discrete. In fact, every ϵ-neighborhood of $0 \in \mathbb{Q}$ contains infinitely many rationals.
- Not every discrete subset of $\mathbb{R}$ is a lattice; for example, the set of primes does not form a subgroup since 0 is not in the set.
- The subgroup $G = \mathbb{Z} \oplus \mathbb{Z}\sqrt{2} = \{(a, b\sqrt{2}) \ : a, b \in \mathbb{Z}\}$ is a lattice in $\mathbb{R}^2$ with two linearly independent generators, $(1, 0)$ and $(0, \sqrt{2})$.

The set of vectors $\{\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n\}$ is called a *lattice basis* and can be expressed using a matrix having the basis vectors as columns, which we denote by $\mathcal{L}(\boldsymbol{B}) = \{\boldsymbol{B}\boldsymbol{x} : \boldsymbol{x} \in \mathbb{Z}^n\}$, where $\boldsymbol{B}\boldsymbol{x}$ is the usual matrix-vector multiplication. Henceforth we only consider lattices in $\mathbb{R}^n$, and restrict to the case $m = n$. A given lattice has many different bases. When the basis $\boldsymbol{B}$ is clear from the context, we can denote the lattice simply by $\mathcal{L}$ instead of $\mathcal{L}(\boldsymbol{B})$.

Definition 4.1.2. Let $\boldsymbol{B} = \{\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n\} \in \mathbb{R}^n$. Then $Span(\boldsymbol{B}) = \{\boldsymbol{B}\boldsymbol{x} : \boldsymbol{x} \in \mathbb{R}^n\}$.

Note that the difference between $\mathcal{L}(\boldsymbol{B})$ and $Span(\boldsymbol{B})$ is that one can use arbitrary real coefficients to combine the basis vectors in $Span(\boldsymbol{B})$, whereas in $\mathcal{L}(\boldsymbol{B})$ only integer linear combinations are allowed. We note that $Span(\boldsymbol{B})$ does not depend on the particular basis vectors $\boldsymbol{B}$. If $\boldsymbol{B}$ and $\boldsymbol{B'}$ are two bases which generate the same lattice, then $Span(\boldsymbol{B}) = Span(\boldsymbol{B'})$.

Any set of n linearly independent lattice vectors $\boldsymbol{B'} \in \mathcal{L}(\boldsymbol{B})$ is a basis for $Span(\boldsymbol{B})$ as a vector space. However, $\boldsymbol{B'}$ is not necessarily a lattice basis for $\mathcal{L}(\boldsymbol{B})$. For example, consider the 2-dimensional basis vectors $\boldsymbol{b}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$, $\boldsymbol{b}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$.

Let $\boldsymbol{b}'_1 = \boldsymbol{b}_1 + \boldsymbol{b}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$, $\boldsymbol{b}'_2 = \boldsymbol{b}_1 - \boldsymbol{b}_2 = \begin{bmatrix} 0 \\ 3 \end{bmatrix}$. The vectors $\boldsymbol{b}'_1$ and $\boldsymbol{b}'_2$ are linearly independent and they form a basis for the plane $\mathbb{R}^2 = Span(\boldsymbol{b}_1, \boldsymbol{b}_2)$ as a vector space.

However, they are not a basis for $\mathcal{L}(\boldsymbol{b}_1, \boldsymbol{b}_2)$ because the lattice vector $\boldsymbol{b}_1$ cannot be expressed as an integer linear combination of $\boldsymbol{b}'_1$ and $\boldsymbol{b}'_2$.

To gain further insight on the topic, we define the geometric characterization of linearly independent lattice vectors that generate the whole lattice. For this purpose, we define the fundamental parallelepiped of a lattice.

Definition 4.1.3. Let $\mathcal{L}(\boldsymbol{B})$ be a lattice basis. The *fundamental parallelepiped* of the corresponding lattice is defined as $\mathcal{P}(\boldsymbol{B}) = \{\boldsymbol{Bx} : 0 \leq x_i < 1\}$.

Recall that a matrix with integer entries and determinant $\det(\boldsymbol{U}) = \pm 1$ is called a *unimodular matrix*. Two bases $\boldsymbol{B}, \boldsymbol{B'} \in \mathbb{R}^n$ generate the same lattice $\mathcal{L}$ if and only if there exists an $n \times n$ unimodular matrix $\boldsymbol{U} \in \mathbb{Z}^{n \times n}$ such that $\boldsymbol{B} = \boldsymbol{UB'}$.

Definition 4.1.4. The *determinant* of a lattice $\mathcal{L}(\boldsymbol{B})$, denoted $\det(\mathcal{L}(\boldsymbol{B}))$, is the n-dimensional volume of the fundamental parallelepiped $\mathcal{P}(\boldsymbol{B})$ spanned by the basis vectors.

If $\boldsymbol{B}$ and $\boldsymbol{B'}$ generate the same lattice, it follows immediately from the above condition that $\det(\boldsymbol{B}) = \det(\boldsymbol{U}) \det(\boldsymbol{B'})$. Geometrically, this corresponds to the intuition that the areas of the fundamental parallelepipeds $\mathcal{P}(\boldsymbol{B})$ and $\mathcal{P}(\boldsymbol{B'})$ are exactly the same because the two bases generate the same lattice. The determinant is a lattice invariant, i.e. it does not depend on the particular choice of basis of the lattice.

One way to compute the determinant of a lattice is given by the *Gram–Schmidt* orthogonalization process. Given any set $\{\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n\} \in \mathbb{R}^n$ of n linearly independent vectors , the Gram–Schmidt process produces a set of n orthogonal, linearly independent vectors $\{\boldsymbol{b}_1^*, \ldots, \boldsymbol{b}_n^*\} \in \mathbb{R}^n$ by iteratively computing

$$\boldsymbol{b}_i^* = \boldsymbol{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \boldsymbol{b}_j^*,$$

where

$$\mu_{i,j} = \frac{\langle \boldsymbol{b}_i, \boldsymbol{b}_j^* \rangle}{\langle \boldsymbol{b}_j^*, \boldsymbol{b}_j^* \rangle}.$$

In the above, $\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \sum_{i=1}^{n} x_i y_i$ is the usual inner product, or dot product, of two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ in $\mathbb{R}^n$. For every i, the vector $\boldsymbol{b}_i^*$ is the component of $\boldsymbol{b}_i$ orthogonal to $Span(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_{i-1})$. In particular, $Span(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_i) = Span(\boldsymbol{b}_1^*, \ldots, \boldsymbol{b}_i^*)$, and the vectors $\boldsymbol{b}_i^*$ are pairwise orthogonal, i.e. $\langle \boldsymbol{b}_i^*, \boldsymbol{b}_j^* \rangle = 0$ for all $i \neq j$. The determinant of the lattice can be expressed in terms of the orthogonalized vectors as

$$\det(\mathcal{L}(\boldsymbol{B})) = \prod_{i=1}^{n} \|\boldsymbol{b}_i^*\|$$

where $\|\boldsymbol{x}\| = \sqrt{\sum_i x_i^2}$ is the Euclidean norm.

It is important to note that the orthogonal vectors $\boldsymbol{b}_i^*$ depend on the order of the original basis vectors. Given a basis matrix $\boldsymbol{B} = [\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n]$, let $\boldsymbol{B}^*$ be the matrix whose columns are the orthogonalized vectors $[\boldsymbol{b}_1^*, \ldots, \boldsymbol{b}_n^*]$. The matrix $\boldsymbol{B}^*$ is a basis of $Span(\boldsymbol{B})$ as a vector space but is usually not a basis for $\mathcal{L}(\boldsymbol{B})$.

In general, a lattice does not have a basis consisting of mutually orthogonal vectors. One can compute $\det(\mathcal{L}(\boldsymbol{B}))$ as the square root of the determinant of the

Gram matrix $\boldsymbol{B}^{\boldsymbol{T}}\boldsymbol{B}$, i.e. the $n \times n$ matrix whose entry at row i and column j is the inner product $\langle \boldsymbol{b}_i, \boldsymbol{b}_j \rangle$:

$$\det(\mathcal{L}(\boldsymbol{B})) = \sqrt{\det(\boldsymbol{B}^T\boldsymbol{B})}.$$

One basic property of a lattice is the length of its non-zero vectors. Our measure of length is the Euclidean norm $\|x\| = \sqrt{\sum_i x_i^2}$, i.e. the norm corresponding to the Euclidean distance $\text{dist}(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{y}\| = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$.

The length of the shortest non-zero vector in the lattice is denoted by λ_1. It is the smallest number r such that the lattice points inside a ball of radius r span a space of dimension at least 1. This definition leads to the following generalization of λ_1 called *successive minima*.

Let $\mathcal{B}_n(\mathbf{0}, r) = \{x \in \mathbb{R}^n : \|\boldsymbol{x}\| < r\}$ be the n-dimensional open ball of radius r centered at $\mathbf{0}$. Given an n-dimensional lattice $\mathcal{L}$, we define its *successive minima* $\lambda_1, \ldots, \lambda_n$ as follows. The *i-th minimum* $\lambda_i(\mathcal{L})$ is the radius of the smallest sphere centered at the origin containing at least i linearly independent lattice vectors:

$$\lambda_i(\mathcal{L}) = \inf\{r : \dim(Span(\mathcal{L} \cap \mathcal{B}(\mathbf{0}, r))) \geq i\}.$$

Lattices can also be defined as discrete nonempty subsets $\mathcal{L}$ of $\mathbb{R}^n$ closed under addition, i.e. having the property that if $\boldsymbol{x} \in \mathcal{L}$ and $\boldsymbol{y} \in \mathcal{L}$, then $\boldsymbol{x} + \boldsymbol{y} \in \mathcal{L}$. Here "discrete" means that there exists a positive real number $\lambda > 0$ such that the distance between any two lattice vectors is at least λ. It follows from the definition of a lattice as a discrete additive subgroup of $\mathbb{R}^n$ that there always exist vectors achieving successive minima. That is, there are linearly independent vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathcal{L}$ such that $\|\boldsymbol{x}_i\| = \lambda_i$ for $i = 1, \ldots, n$. In particular, $\lambda_1(\mathcal{L}) = \min_{\boldsymbol{x} \in \mathcal{L} \setminus \{\mathbf{0}\}} \|\boldsymbol{x}\|$, the length of a shortest non-zero vector in the lattice. This is also equal to the minimum distance between any two distinct lattice points.

In the following, we state some standard results from lattice theory. For more background and a detailed treatment of the subject, we refer to [83].

The following theorem gives a useful lower bound on the length of the shortest nonzero vector in a lattice.

Theorem 4.1.1. *[83] Let $\boldsymbol{B}$ be an n-dimensional lattice basis and $\boldsymbol{B}^*$ be the corresponding Gram–Schmidt orthogonal basis. Then the first minimum of the lattice satisfies*

$$\lambda_1(\mathcal{L}) \geq min_{1 \leq i \leq n} \|\boldsymbol{b}_i^*\|.$$

We proceed by stating an important theorem on the Minkowski upper bounds on the product of successive minima of any lattice.

Theorem 4.1.2 (Blichfeld)**.** *[83] For any lattice $\mathcal{L} \subset \mathbb{R}^n$ and for any measurable set $S \subset \mathbb{R}^n$, if S has volume $vol(S) > \det(\mathcal{L})$, then there exist two distinct points $z_1, z_2 \in S$ such that $z_1 - z_2 \in \mathcal{L}$.*

As a corollary to Blichfeld's theorem, we obtain the following theorem due to Minkowski. It states that any sufficiently large, centrally symmetric convex set contains a nonzero lattice point. A set S is called centrally symmetric if for any $x \in S$ we have $-x \in S$, and it is called convex if for any $x, y \in S$ and any $\lambda \in [0, 1]$ we have $\lambda x + (1 - \lambda)y \in S$, that is, given any two points x, y in the set S, the line joining them lies entirely inside S.

Theorem 4.1.3 (Convex body theorem)**.** *[83] For any lattice $\mathcal{L} \in \mathbb{R}^n$ and any convex set $S \subset Span(\mathcal{L})$ symmetric about the origin, if $vol(S) > 2^n \det(\mathcal{L})$, then S contains a nonzero lattice point $\boldsymbol{x} \in S \cap \mathcal{L} \setminus \{\mathbf{0}\}$.*

Minkowski's convex body theorem can be used to bound the length of the shortest vector as follows. Let $S = \mathcal{B}(\mathbf{0}, \sqrt{n}\det(\mathcal{L})^{1/n})$ be the open ball of radius $\sqrt{n}\det(\mathcal{L})^{1/n}$. If the volume of S is greater than $2^n \det(\mathcal{L})$ then by the convex body theorem there exists a nonzero lattice vector in $S \cap \mathcal{L} \setminus \{\mathbf{0}\}$. Notice that S has volume strictly greater than $2^n \det(\mathcal{L})$ because it contains an n-dimensional hypercube with edges of length $2\det(\mathcal{L})^{1/n}$. This shows that for any n-dimensional lattice $\mathcal{L}$, the length of the shortest nonzero vector (in Euclidean norm) satisfies

$$\lambda_1(\mathcal{L}) < \sqrt{n}\ \det(\mathcal{L})^{1/n}.$$

The above result can be generalized to i-th minima for $i > 1$ as follows.

Theorem 4.1.4 (Minkowski's second theorem)**.** *[83] For any n-dimensional lattice $\mathcal{L}(\boldsymbol{B})$, the successive minima (in Euclidean norm) $\lambda_1, \ldots, \lambda_n$ satisfy*

$$\Big(\prod_{i=1}^{n} \lambda_i\Big)^{1/n} < \sqrt{n}\ \det(\boldsymbol{B})^{1/n}.$$

4.2 Computational problems in lattices

Two of the main computational problems involving lattices are:

(i) the closest vector problem: given a lattice in $\mathbb{R}^n$ and a target vector $\boldsymbol{x} \in \mathbb{R}^n$, the goal is to find a lattice vector closest to $\boldsymbol{x}$;

(ii) the shortest vector problem: given a lattice in $\mathbb{R}^n$, find the lattice vector with the shortest norm.

In this chapter, we focus on the shortest vector problem.

It is clear from Minkowski's convex body theorem that there is a simple way to bound the length of the shortest nonzero vector λ_1 in a lattice $\mathcal{L}(B)$. It is important to note that λ_1 can be much smaller than $\sqrt{n}\ \det(\mathcal{L})^{1/n}$. But in the worst case there exist lattices for which $\lambda_1 > c\sqrt{n}\ \det(\mathcal{L})^{1/n}$ for some constant c. Moreover, the proof of Minkowski's theorem is not constructive, meaning that although it guarantees that a short nonzero vector exists, it does not provide any way of finding such a vector. Finding such a vector constitutes the well-known shortest vector problem (SVP), which we formally define below.

Definition 4.2.1 (Shortest vector problem)**.** Given a lattice basis $\boldsymbol{B}$, the *shortest vector problem (SVP)* is to find a nonzero lattice vector $\boldsymbol{s} \in \mathcal{L}(\boldsymbol{B})$ such that $\|\boldsymbol{s}\| = \lambda_1(\mathcal{L}(\boldsymbol{B}))$.

There is an approximation variant of this problem. The goal is to output a nonzero lattice vector whose norm is greater than that of the shortest nonzero vector by at most some approximation factor γ.

Geometrically, the shortest vector problem can be seen as a generalization of the computation of the greatest common divisor (GCD). We know that, given two

integers a, b, the Euclidean algorithm actually computes the smallest nonzero linear combination of a and b. In the case of two dimensions, an algorithm for solving the shortest vector problem in polynomial time was given by Gauss (see section 4.3.2). The formulation of the problem in arbitrary dimension was given by Dirichlet and studied by Hermite [50], Korkine and Zolotarev [60]. Later, Minkowski founded the theory of geometry of numbers [21], which deals with the study of lattices and problems related to them. Algorithms for finding a shortest vector were given by Rosser [102], Knuth [59] and Dieter [29], but none of these algorithms run in polynomial time. Later, the development of the LLL basis reduction algorithm [69] was a major breakthrough in the field (see also section 4.3.3). Using the LLL algorithm, it was possible to solve the shortest vector problem in any dimension n in polynomial-time with an approximation factor of $2^{\mathcal{O}(n)}$. The LLL algorithm can be seen as a generalization of the Euclidean algorithm to n dimensions. Later, the approximation factor was improved to $2^{\mathcal{O}(n(\log\log n)^2/\log n)}$ by Schnorr [106], followed by Ajtai [3] with an approximation factor of $2^{\mathcal{O}(n\log\log n/\log n)}$ using a sieve algorithm.

Despite all these improvements, SVP resisted any attempts to devise an exact polynomial time algorithm for arbitrary dimension. Part of the difficulty of SVP comes from the fact that a lattice has many different bases and contains very long vectors. Given the above results, one might expect SVP to be NP-hard to approximate, even within very large factors. However, the best known results show that approximating SVP to within factors $2^{(\log n)^{\frac{1}{2}-\epsilon}}$ is NP-hard under quasi-polynomial time reductions [58] (an algorithm is said to run in quasi-polynomial time if its worst case time complexity is $2^{\mathcal{O}(\log(n)^c)}$ for some fixed constant c). The problem of approximating SVP to within polynomial factors n^c for $c > \frac{1}{2}$ is not believed to be NP-hard [1, 41, 63]. However, the asymptotically fastest known algorithm for SVP (namely, the AKS Sieve introduced by Ajtai, Kumar and Sivakumar [3]) runs in probabilistic exponential time $2^{\mathcal{O}(n)}$. In practice, it is not clear for which dimension n solving SVP becomes really hard or infeasible, since the problem also depends on the given lattice basis.

4.3 Lattice reduction algorithms

The goal of a lattice reduction algorithm is to transform a given lattice basis to a basis which is short and as orthogonal as possible. We describe some standard lattice reduction algorithms in the next section.

4.3.1 Gram–Schmidt orthogonalization

The Gram–Schmidt orthogonalization process may be regarded as a first step in lattice reduction. More details can be found in Chapter 4 in [36]. Given any n linearly independent vectors in $\mathbb{R}^n$, the Gram–Schmidt process produces a set of n orthogonal, linearly independent vectors in $\mathbb{R}^n$. In particular, it may be applied to a basis $\boldsymbol{B}$ of a lattice $\mathcal{L}$. The resulting orthogonal basis is, in general, not a subset of $\mathcal{L}$, as the process involves real numbers and not integers. Nevertheless, the Gram–Schmidt (GS) process gives valuable information on the minimum distance $\lambda_1(\mathcal{L})$.

For n linearly independent vectors $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n \in \mathbb{R}^n$, we compute the Gram–Schmidt orthogonalized vectors $\boldsymbol{b}_1^*, \ldots, \boldsymbol{b}_n^*$ via an iterative process. First, we define

$\boldsymbol{b}_1^* = \boldsymbol{b}_1$, and then for $i = 2, \ldots, n$, and $j = i-1, \ldots, 1$ we define $\boldsymbol{b}_j^*$ to be the component of $\boldsymbol{b}_j$ orthogonal to $Span(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_{j-1})$ which is equal to $Span(\boldsymbol{b}_1^*, \ldots, \boldsymbol{b}_{j-1}^*)$. With $\mu_{i,j}$ computed as $\mu_{i,j} = \frac{\langle \boldsymbol{b}_i, \boldsymbol{b}_j^* \rangle}{\langle \boldsymbol{b}_j^*, \boldsymbol{b}_j^* \rangle}$, the Gram–Schmidt process can be described as follows:

$$\begin{aligned}
\boldsymbol{b}_1^* &= \boldsymbol{b}_1 \\
\boldsymbol{b}_2^* &= \boldsymbol{b}_2 - \mu_{2,1}\boldsymbol{b}_1^* \\
\boldsymbol{b}_3^* &= \boldsymbol{b}_3 - \mu_{3,1}\boldsymbol{b}_1^* - \mu_{3,2}\boldsymbol{b}_2^* \\
&\vdots \\
\boldsymbol{b}_n^* &= \boldsymbol{b}_n - \sum_{1 \le j < n} \mu_{n,j}\boldsymbol{b}_j^*.
\end{aligned}$$

The set $\{\boldsymbol{b}_1^*, \boldsymbol{b}_2^*, \ldots, \boldsymbol{b}_n^*\}$ is an orthogonal basis for the same space as that spanned by $\{\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n\}$. More generally, for any $1 \le i \le n$, the subspace spanned by $\{\boldsymbol{b}_1^*, \boldsymbol{b}_2^*, \ldots, \boldsymbol{b}_i^*\}$ is the same as that spanned by $\{\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_i\}$. The procedure for producing the Gram–Schmidt vectors is given in Algorithm 6.

Theorem 4.3.1. *[36] If $\{\boldsymbol{b}_1^*, \boldsymbol{b}_2^*, \ldots, \boldsymbol{b}_n^*\}$ is the orthogonal basis obtained by the Gram–Schmidt process applied to a lattice $\mathcal{L}$, then $\lambda_1(\mathcal{L}) \ge min_{1 \le i \le n} \|\boldsymbol{b}_i^*\|$.*

Thus, applying the GS process yields a lower bound on the length of the shortest non-zero vector in a given lattice $\mathcal{L}$.

Algorithm 6 Gram–Schmidt algorithm

Input: Ordered basis $\boldsymbol{B} = (\boldsymbol{b}_1, \ldots, \boldsymbol{b_n}), \boldsymbol{b}_i \in \mathbb{R}^n$
Output: $(\boldsymbol{b}_1^*, \ldots, \boldsymbol{b}_n^*)$, the orthogonal Gram–Schmidt basis for $\boldsymbol{B}$.
$\boldsymbol{b}_1^* = \boldsymbol{b}_1$
for $i = 2$ to n **do**
  $\boldsymbol{v} = \boldsymbol{b}_i$
  for $j = i - 1$ to 1 **do**
    $\mu_{i,j} = \langle \boldsymbol{b}_i, \boldsymbol{b}_j^* \rangle / \langle \boldsymbol{b}_j^*, \boldsymbol{b}_j^* \rangle$
    $\boldsymbol{v} = \boldsymbol{v} - \mu_{i,j}\boldsymbol{b}_j^*$
  end for
  $\boldsymbol{b}_i^* = \boldsymbol{v}$
end for
return $(\boldsymbol{b}_1^*, \ldots, \boldsymbol{b}_n^*)$

4.3.2 Lattice basis reduction in two dimensions

An algorithm for lattice basis reduction in two dimensions was given by *Lagrange* and *Gauss*; see [36, p. 366]. This algorithm can be viewed as a generalization of Euclid's algorithm to a two dimensional lattice. Given a lattice basis $\boldsymbol{B} = (\boldsymbol{b}_1, \boldsymbol{b}_2), \boldsymbol{b}_i \in \mathbb{R}^2$, the goal is to output a basis for the lattice such that the lengths of the basis vectors are as short as possible (in this case, we actually obtain the shortest vector of length $\lambda_1(\mathcal{L}(\boldsymbol{B}))$. This procedure is given in Algorithm 7, where we consider an integral lattice basis, that is, we assume that the basis vectors have integer entries.

Definition 4.3.1. An ordered basis $\boldsymbol{b_1}, \boldsymbol{b_2}$ for $\mathbb{R}^2$ is *Lagrange–Gauss reduced* if $\|\boldsymbol{b_1}\| \leq \|\boldsymbol{b_2}\| \leq \|\boldsymbol{b_2} + q\boldsymbol{b_1}\|$ for all $q \in \mathbb{Z}$.

Algorithm 7 Lagrange–Gauss reduction

Input: Ordered basis ($\|\boldsymbol{b}_1\| \leq \|\boldsymbol{b}_2\|$) $\boldsymbol{b}_1, \boldsymbol{b}_2 \in \mathbb{Z}^2$ for a lattice $\mathcal{L}$.
Output: Basis $\boldsymbol{b}_1, \boldsymbol{b}_2$ for $\mathcal{L}$ such that $\|\boldsymbol{b}_i\| = \lambda_i(\mathcal{L})$.
$B_1 = \|\boldsymbol{b}_1\|^2$
$\mu = \langle \boldsymbol{b}_1, \boldsymbol{b}_2 \rangle / B_1$
$\boldsymbol{b}_2 = \boldsymbol{b}_2 - \lfloor \mu \rceil \boldsymbol{b}_1$ ▷ ($\lfloor \cdot \rceil$ means closest integer)
$B_2 = \|\boldsymbol{b}_2\|^2$.
while $B_2 < B_1$ **do**
  Swap $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$
  $B_1 = B_2$
  $\mu = \langle \boldsymbol{b_1}, \boldsymbol{b_2} \rangle / B_1$
  $\boldsymbol{b_2} = \boldsymbol{b_2} - \lfloor \mu \rceil \boldsymbol{b_1}$
  $B_2 = \|\boldsymbol{b_2}\|^2$
end while
return $(\boldsymbol{b_1}, \boldsymbol{b_2})$

It is interesting to study lattice reduction in two dimensions because the LLL algorithm performs a succession of steps of the Lagrange–Gauss algorithm on the local bases, and it stops when all the local bases are reduced. If $\boldsymbol{b}_1, \boldsymbol{b}_2 \in \mathbb{Z}^2$ are such that $\|\boldsymbol{b}_i\|^2 \leq X$, where $X \in \mathbb{Z}$ with $X \geq 2$, then the Lagrange-Gauss algorithm performs $\mathcal{O}(\log(X)^3)$ bit operations.

4.3.3 Lenstra-Lenstra-Lovász (LLL) algorithm

The LLL algorithm, developed by A.K. Lenstra, H.W. Lenstra and L. Lovász in 1982 [69] is an iterative algorithm that transforms a given lattice basis into a basis which generates the same lattice and which is LLL-reduced (see definition below). The LLL algorithm runs in polynomial time and finds an approximate solution $\boldsymbol{s}$ to the shortest vector problem, in the sense that the length of the solution $\boldsymbol{s}$ found by the algorithm is at most $\gamma\lambda_1(\mathcal{L})$ for some approximation factor γ. The approximation factor for dimension n is $\gamma = 2^{\mathcal{O}(n)}$.

The LLL algorithm can be regarded as a generalisation of the Gauss–Lagrange algorithm to higher dimensions. Recall that, given a set of n linearly independent vectors $\{\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n\}$ as a basis in $\mathbb{R}^n$, the Gram–Schmidt coefficients are given by

$$\mu_{i,j} = \frac{\langle \boldsymbol{b}_i, \boldsymbol{b}_j^* \rangle}{\langle \boldsymbol{b}_j^*, \boldsymbol{b}_j^* \rangle}.$$

Definition 4.3.2. Given an ordered basis for the lattice $\boldsymbol{B} = \{\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n\} \in \mathbb{R}^n$ and some $\delta \in (\frac{1}{4}, 1)$, we say that $\boldsymbol{B}$ is an *LLL-reduced basis* if the following holds:

1. For all $1 \leq j < i \leq n$, $|\mu_{i,j}| \leq \frac{1}{2}$. Such a basis is said to be size reduced.
2. For $1 \leq i < n$, $\delta\|\boldsymbol{b}_i^*\|^2 \leq \|\mu_{i+1,i}, \boldsymbol{b}_i^* + \boldsymbol{b}_{i+1}^*\|^2$. This condition is called the *Lovász condition*. It is usual to take $\delta = \frac{3}{4}$, but the algorithm works for any $\frac{1}{4} < \delta < 1$.

The LLL algorithm works as follows (see p. 375 in [36] for more details). Given an input basis $\boldsymbol{B} \in \mathbb{R}^n$, we perform the following operations:

1. Compute $\boldsymbol{B}^*$, the Gram–Schmidt orthogonalized vectors of the basis $\boldsymbol{B}$.

2. Let $\boldsymbol{B} \leftarrow \text{SizeReduce}(\boldsymbol{B})$, that is $|\mu_{i,j}| \leq 1/2$ for all i, j. The LLL algorithm ensures that the basis is size reduced, and does not change $\mathcal{L}(\boldsymbol{B})$. The idea behind the subroutine SizeReduce($\boldsymbol{B}$) is that for each $i \in \{2, \ldots, n\}$ and $j \in \{1, \ldots, i-1\}$ we set $\boldsymbol{b}_i \leftarrow \boldsymbol{b}_i - c_{i,j}\boldsymbol{b}_j$, where $c_{i,j} = \lfloor \frac{\langle \boldsymbol{b}_i, \boldsymbol{b}_j^* \rangle}{\langle \boldsymbol{b}_j^*, \boldsymbol{b}_j^* \rangle} \rceil$.

3. If there exists $1 \leq i < n$ for which the Lovász condition is violated, i.e. $\delta \|\boldsymbol{b}_i^*\|^2 > \|\mu_{i+1}\boldsymbol{b}_i^* + \boldsymbol{b}_{i+1}^*\|^2$, then swap $\boldsymbol{b}_i$ and $\boldsymbol{b}_i + 1$ and go back to Step 1. Otherwise, output $\boldsymbol{B}$.

Algorithm 8 LLL reduction

Input: Ordered basis $(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n) \in \mathbb{R}^n$, $\frac{1}{4} < \delta < 1$.
Output: basis is LLL-reduced $(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n)$**.**
Compute the Gram–Schmidt basis $(\boldsymbol{b}_1^*, \ldots, \boldsymbol{b}_n^*)$ and coefficients $\mu_{i,j}$ for $1 < j < i \leq n$.
$k = 2$
while $k \leq n$ **do**
  for $j = (k-1)$ to 1 **do** // Size reduction
    Let $c_{k,j} = \lfloor \mu_{k,j} \rceil$ and set $\boldsymbol{b}_k = \boldsymbol{b}_k - c_{k,j}\boldsymbol{b}_j$
    Update the value $\mu_{k,j}$
  end for
  if $\|\boldsymbol{b}_k^* + \mu_{k,k-1}\boldsymbol{b}_{k-1}^*\|^2 \geq \delta \|\boldsymbol{b}_{k-1}^*\|^2$ **then** // Lovász condition
    $k = k + 1$
  else
    Swap b_k with b_{k-1}
    Update the values $\boldsymbol{b}_k^*, \boldsymbol{b}_{k-1}^*$ and recompute the μ-matrix
    $k = \max\{2, k-1\}$
  end if
end while

The overall run time of the LLL algorithm is polynomial in the input size as specified in the following theorem.

Theorem 4.3.2. *[36] Let $\mathcal{L}$ be a lattice in $\mathbb{R}^n$ with basis $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n$ and let $X \in \mathbb{Z}$ with $X \geq 2$ be such that $\|\boldsymbol{b}_i\|^2 \leq X$ for $1 \leq i \leq n$. Let $1/4 < \delta < 1$. Then the LLL algorithm with factor δ terminates and performs $\mathcal{O}(n^2 \log(X))$ iterations.*

4.3.4 BKZ reduction

The Blockwise Korkine-Zolotarev (BKZ) reduction algorithm is the fastest known reduction algorithm and was first introduced by Schnorr and Euchner [107] and later improved by Nguyen [22]. The BKZ algorithm uses local blocks to achieve reduction, hence the name. The size of the block is determined by an additional input parameter to the algorithm called the block-size β. The quality of the reduction

achieved by this algorithm depends on β, with an increase in block-size leading to improved reduction.

The BKZ algorithm starts by LLL-reducing a given lattice basis. The quality of the reduction is then improved iteratively. Given an input lattice basis and the block-size β, the first block consists of the first β basis vectors. The second block is constructed similarly with a slight modification: instead of taking the second block to be $\boldsymbol{b}_2, \ldots, \boldsymbol{b}_{\beta+1}$, the projections of the aforementioned vectors onto the orthogonal complement of the first basis vector are used. The remaining blocks are constructed in the same way. For block i, the basis vectors with index j, satisfying $i \leq j < i+\beta$, are projected onto the orthogonal complement of the vector space spanned by all basis vectors $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_{i-1}$. This projection forms the local block. The index l of the last vector in a block is computed as $l = \min\{i+\beta, n\}$, where n is the dimension of the lattice. That means that the block starting at index $n-\beta+1$ is the last one of size β. From there on, the size of the last block decreases with each step. The local blocks are also called local projected blocks.

The algorithm works as follows. Each iteration takes one block at a time and performs an enumeration on the locally projected lattice to ensures that the first vector of the block is the shortest vector inside the lattice spanned by this block. If this is not the case, then a new vector is inserted in the lattice; this newly inserted vector is in the lattice, and is therefore linearly dependent on the basis vectors. This means that after inserting the new vector, the local block is not a basis for the local lattice any longer. To obtain a basis again, the algorithm runs the LLL algorithm on the expanded block. The LLL algorithm is executed for each local block, no matter whether an insertion has taken place or not.

Let π_i denote the orthogonal projection $\pi_i : \mathbb{R}^n \to Span(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_{i-1})^\perp$ such that $\boldsymbol{b} - \pi_i(\boldsymbol{b}) \in Span(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_{i-1})$. Let $\mathcal{L}_{[j,k]}$ denote the locally projected lattice defined as the lattice spanned by the orthogonal projection of $(\boldsymbol{b}_j, \ldots, \boldsymbol{b}_k)$ onto $(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_{j-1})^\perp$, where $k = \min(j+\beta-1, n)$. The local block denoted as $B_{[j,k]} = (\pi_j(\boldsymbol{b}_j), \pi_j(\boldsymbol{b}_{j+1}), \ldots, \pi_j(\boldsymbol{b}_k))$ is the result of the projection of the basis vectors. The block $B_{[j,k]}$ can be obtained using Gram–Schmidt orthogonalization. This means that all vectors of the locally projected lattice are orthogonal to all basis vectors $\boldsymbol{b}_i$ of $\mathcal{L}$ where $i < j$.

The dimension of the local lattices, i.e. the lattices corresponding to the local blocks, is determined by the block size. Let the starting index of the local blocks be j. This is shifted cyclically through 1 to $n-1$ in the algorithm, until no further improvement can be achieved using the local projected lattices for $n-1$ subsequent iteration steps. The end index k is determined as $\min\{j+\beta-1, n\}$. This means that all blocks but the last one have dimension β, whereas the blocks ending with index n have gradually decreasing dimension, starting with β and ending with 1.

The reduction process is shown in Algorithm 9. Note that the shortest vector in the local lattices is obtained using an enumeration algorithm, which is the most computationally expensive part of the process. To speed up the enumeration subroutine, one may use a pruned enumeration algorithm, which has some probability of failure, i.e. of not finding the shortest vector. No good upper bound on the complexity of the BKZ algorithm is known. The best upper bound known for the number of calls to the enumeration subroutine is exponential [44].

Algorithm 9 BKZ reduction

Input: A basis $\boldsymbol{B} = (\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n) \in \mathbb{Z}^n$, a blocksize $\beta \in \{1, \ldots, n\}$, Gram–Schmidt matrices μ_{ij} and Gram–Schmidt vectors $\|\boldsymbol{b}_1^*\|^2, \ldots, \|\boldsymbol{b}_n^*\|^2$.
Output: BKZ-β reduced basis $(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n)$.
$z \leftarrow 0$
$j \leftarrow 0$;
LLL $(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n, \mu)$ //LLL reduce the basis and update μ.
while $z < n - 1$ **do**
$j \leftarrow (j \bmod (n-1)) + 1$
$k \leftarrow \min(j + \beta - 1, n)$;
$h \leftarrow \min(k + 1, n)$; // define the local block
$\boldsymbol{v} \leftarrow$enum$(\mu_{k,j}, \|\boldsymbol{b}_j^*\|^2, \ldots, \|\boldsymbol{b}_k^*\|^2)$ // find $\boldsymbol{v} = (v_j, \ldots, v_k) \in \mathbb{Z}^{k-j+1} - \mathbf{0}$ such that $\|\pi_j(\sum_{i=j}^{k} v_i \boldsymbol{b}_i)\| = \lambda_1(\mathcal{L}_{j,k})$
if $\boldsymbol{v} \neq (1, 0, \ldots, 0)$ **then**
$z \leftarrow 0$
LLL$(\boldsymbol{b}_1, \ldots, \sum_{i=j}^{k} v_i \boldsymbol{b}_i, \boldsymbol{b}_j, \ldots, \boldsymbol{b}_h, \mu)$ at stage j; // insert the new vector in the lattice at the start of the current block, then remove the dependency in the current block, update μ.
else
$z \leftarrow z + 1$
LLL$(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_h, \mu)$ at stage $h - 1$; // LLL-reduce the next block before enumeration.
end if
end while

4.3.5 Lattice enumeration algorithms

As mentioned earlier, there are two main algorithmic techniques for solving lattice problems:

1. The *lattice basis reduction* technique works by applying successive transformations to the input basis in an attempt to make its vectors shorter and more orthogonal. The LLL algorithm runs in polynomial time but the approximation factor it provides is asymptotically exponential.

2. The *enumeration technique* is an exhaustive search aiming to find a best integer combination of the basis vectors, i.e. one which results in a vector of minimal norm. Enumeration algorithms run in exponential time but are guaranteed to find a shortest vector (as opposed to approximation techniques).

Often these two approaches are combined. In practice, enumeration algorithms are used in block-based reduction algorithms (in particular, BKZ) as a subroutine to find short vectors in low dimensional sub-lattices having the block-size as their dimension. The running time of the enumeration algorithms heavily depends on the quality of the input basis. Therefore, enumeration algorithms are almost never applied directly to the given basis. Rather, one first reduces the given lattice basis, and then runs enumeration on the reduced basis.

As mentioned earlier, given a lattice $\mathcal{L}(\boldsymbol{B}) = \{v_1\boldsymbol{b}_1 + \ldots + v_n\boldsymbol{b}_n : v_i \in \mathbb{Z}\}$, the basic enumeration algorithm is an exhaustive search whose goal is to find integer

combinations of the coefficients v_i such that the resulting combination of the basis vectors has norm below some given threshold R. The search can be seen as a depth-first search on a tree whose leaves correspond to lattice vectors, and whose internal nodes correspond to partial assignments to the coefficients of the integer combinations.

The first enumeration algorithm was developed by Pohst in the 1980s [94]. This algorithm is also known as the Fincke–Pohst enumeration algorithm and it is a deterministic algorithm based on exhaustive enumeration of lattice points within a small convex set, or hyper-sphere. The complexity of the algorithm for an n-dimensional lattice in the worst case is of order $2^{\mathcal{O}(n^2)}$. A similar, and better, version of this algorithm was developed by *Kannan*, the main difference between the two being that in Kannan's algorithm, a long pre-computation on the basis vectors is performed before starting the enumeration process. The estimated complexity is $2^{\mathcal{O}(n \log n)}$ [56]. In 1985, Helfrich [47] refined Kannan's algorithm and obtained a procedure with complexity bounded by $2^{\mathcal{O}(\frac{n}{2} \log n)}$. Even though these algorithms have low theoretical complexity, in practice they are much slower than standard enumeration.

The algorithms mentioned above obtain a substantial speedup by means of *pruning* techniques, which were first introduced by Schnorr–Euchner [109] and Schnorr–Hörner in the 90s [110]. The rough idea is to prune away subtrees of the search tree, thereby reducing the search space. Although this reduces the search space to only a subset of all possible solutions, it introduces some probability of missing the optimal solutions. The rationale is that this probability is small compared to the gain in running time.

The current state of the art is enumeration using extreme pruning, which was introduced by Gamma et al. [37]. They obtained exponential speedups using some bounding functions that significantly reduce the search space. Their analysis shows that, for a well-chosen bounding function, it is possible to obtain an exponential speedup for a dimension n lattice by a factor of $2^{\frac{n}{2}}$ as compared to basic enumeration, while maintaining a success probability of at least 95%.

4.4 The standard enumeration algorithm

Let $\mathcal{L}$ be a lattice with shortest vector $\boldsymbol{s}$ which is unique up to its sign. Assume that we are given a basis $\{\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n\}$ of $\mathcal{L}$ and an upper bound R on $\lambda_1(\mathcal{L})$ such that we need to find all vectors $\boldsymbol{w}$ in the lattice $\mathcal{L}$ that satisfy $\|\boldsymbol{w}\| \leq R$.

A shortest vector $\boldsymbol{s} \in \mathcal{L}$ can be written as $\boldsymbol{s} = v_1\boldsymbol{b}_1 + v_2\boldsymbol{b}_2 + \ldots + v_n\boldsymbol{b}_n$, where v_i for $1 \leq i \leq n$ are unknown integers. Our goal is to find $\boldsymbol{s}$.

To find $\pm\boldsymbol{s}$, the enumeration algorithm goes through an enumeration tree formed by the subspace spanned by vectors whose norm is at most R [107]. The enumeration tree is a depth first search tree of depth n. Each internal node in the tree is associated with a particular v_i, and each outgoing edge represents an assignment of an integer value (obtained from a range) to v_i. In particular, the root of the tree is the zero vector, while the leaves represent all vectors of $\mathcal{L}$ whose norm is at most R.

At any node, the enumeration algorithm is given an index k of a coefficient not yet branched for, and obtains a set of integers (an interval range) I_k for the possible values that v_k can take. For each integer $t \in I_k$, the algorithm sets $v_k = t$ and calls itself recursively to compute the interval for the next level. The length bound here

remains constant throughout the algorithm. For $1 \leq k \leq n$, the following inequality needs to be satisfied, essentially defining the interval I_k:

$$\Big(v_k + \sum_{i=k+1}^{n} \mu_{i,k} v_i\Big)^2 \|\boldsymbol{b}_k^*\|^2 + \sum_{j=k+1}^{n} \Big(v_j + \sum_{i=j+1}^{n} \mu_{i,j} v_i\Big)^2 \|\boldsymbol{b}_j^*\|^2 \leq R^2. \tag{4.1}$$

By the inequality above, for each $1 \leq k \leq n$, the interval range I_k for v_k can be obtained if the values of v_j are known for $k+1 \leq j \leq n$. Before enumeration can start, the matrix $[\mu_{i,j}]$ of Gram–Schmidt coefficients, called the μ-matrix, and the orthogonal basis vectors $\boldsymbol{b}_1^*, \ldots, \boldsymbol{b}_n^*$ must be computed. The μ-matrix depends on the particular order of the basis vectors, and once it is computed, this order remains fixed throughout the standard enumeration routine.

The actual enumeration starts by computing an interval I_n using (4.1) such that $\|\boldsymbol{s}\| \leq R$ implies $v_n \in I_n$. The algorithm then fixes an integer value in I_n for v_n, and based on this choice, computes an interval I_{n-1} such that $\|\boldsymbol{s}\| \leq R$ implies $v_{n-1} \in I_{n-1}$. Then an integer is selected from I_{n-1} and assigned to v_{n-1}, and the interval where v_{n-2} must be found is computed. This continues until a selection for v_1 can be made, in which case we find a lattice vector with length less than or equal to R, or until an empty interval I_j is computed.

Intervals are computed recursively in the order $I_n, I_{n-1}, \ldots, I_2, I_1$, and all values from all intervals must be tested to perform a complete search that guarantees that a shortest vector will be found. In the following, we denote the length of an interval I_i by $|I_i|$.

The complexity of this algorithm is $2^{\mathcal{O}(n^2)}$. A detailed description is given under Algorithms 10, 11 and 12.

Algorithm 10 Enumeration

Input: $\{\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n\}$, bound R
Output: $\{\boldsymbol{s}_1, \ldots, \boldsymbol{s}_r | \ \|\boldsymbol{s}_i\| < R\}$
Compute the Gram–Schmidt coefficients $\mu_{i,j}$ and the Gram–Schmidt vectors $\{\boldsymbol{b}_1^*, \ldots, \boldsymbol{b}_n^*\}$
$\boldsymbol{a} = \boldsymbol{0}$
enum$(n, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_n, \boldsymbol{b}_1^*, \ldots, \boldsymbol{b}_n^*, \boldsymbol{a}, \mu_{i,j}, \emptyset, R)$

Algorithm 11 enum$(k, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_n, \boldsymbol{b}_1^*, \ldots, \boldsymbol{b}_n^*, \boldsymbol{a}, \mu_{i,j}, \{v_{k+1}, \ldots, v_n\}, R)$

if $k = 0$ **then**
  return $\boldsymbol{a}$
else
  $I_k =$ ComputeInterval$(v_{k+1}, \ldots, v_n, \mu_{i,j}, \boldsymbol{b}_k^*, \ldots, \boldsymbol{b}_n^*, R)$
  for $v_k \in I_k \cap \mathbb{Z}$ **do**
    $\boldsymbol{a} = \boldsymbol{a} + v_k \boldsymbol{b_k}$
    enum$(k-1, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_n, \boldsymbol{b}_1^*, \ldots, \boldsymbol{b}_n^*, \boldsymbol{a}, \mu_{i,j}, \{v_k, \ldots, v_n\}, R)$
  end for
end if

In the rest of this chapter, we refer to the level containing the root node of the search tree as level n, the second highest level being level $n-1$, etc. That is, if a

Algorithm 12 ComputeInterval$(v_{k+1}, \ldots, v_n, \mu_{i,j}, \boldsymbol{b}_k^*, \ldots, \boldsymbol{b}_n^*, R)$

$c_k = -\sum_{i=k+1}^{n} \mu_{i,k} v_i$

$\rho_k = \sum_{j=k+1}^{n} \left(v_j + \sum_{i=j+1}^{n} \mu_{i,j} v_i \right)^2 \|\boldsymbol{b}_j^*\|^2$

Return $\left[c_k - \frac{\sqrt{R^2 - \rho_k}}{\|\boldsymbol{b}_k^*\|}, c_k + \frac{\sqrt{R^2 - \rho_k}}{\|\boldsymbol{b}_k^*\|} \right]$

node is at level k in the search tree, then only the coefficient v_k can be selected for branching at that node.

4.5 Hybrid enumeration

In this section, we study how permutations of the basis vectors of a lattice affect the running time of enumeration. Based on this we present a strategy for selecting an order of the basis vectors that results in a reduction of the number of nodes in the search tree when performing enumeration. We compare this work to standard enumeration and to extreme pruning, which is the current state of the art.

To compare the efficiency of our hybrid enumeration technique to standard enumeration, we run hybrid enumeration on top of standard enumeration, and we observe a reduction in the number of nodes even though the complexity remains asymptotically the same.

On the other hand, when comparing the efficiency of hybrid enumeration to that of extreme pruning, we first run hybrid enumeration on a given input and record its running time. We then run extreme pruning on the same input, but restrict its running time to the one measured for hybrid enumeration. We observe that extreme pruning misses several solutions as opposed to hybrid enumeration.

A detailed description of our experimental results can be found in Section 4.5.5.

One disadvantage of the standard enumeration technique is that the algorithm depends on the computed Gram–Schmidt (GS) orthogonal basis for computing the intervals of values that the coefficients v_i can take. Once the GS orthogonal basis is computed, it fixes the order in which we guess the coefficients.

In the case of hybrid enumeration, we take an approach where the basis vectors are not bound by any particular order and we are free to choose which of the unassigned coefficients v_i to branch for at any given point in the search tree. We show that dynamically changing the order of the coefficients v_i can lower the number of nodes in the search tree as compared to the standard enumeration algorithm.

The price to pay for this flexibility is an increased amount of work at each node of the search tree. The complexity of the computation at each node is $\mathcal{O}(n^4)$. Hence the actual time taken to enumerate a lattice using the new method may be longer than the time taken by the standard GS enumeration. Therefore we only propose to use the new enumeration technique at the nodes on the highest levels in the search tree, and then switch to standard GS enumeration for lower levels. This still leads to a reduction in the number of nodes in comparison to the standard enumeration method, depending on the type of lattice and the level where we switch to standard GS enumeration.

nodes in search tree	BKZ-10	BKZ-20
minimum	60.934.596	4.059.025
average	424.300.658	52.886.123
maximum	1.180.735.200	194.214.522
std. deviation	361.710.571	40.202.374

Table 4.1: Number of nodes to fully enumerate the BKZ-reduced SVP40 challenge lattice for 20 random permutations of the basis. The number of nodes in a search tree is highly dependent on the particular permutation.

4.5.1 Variations in enumeration complexity from basis permutations

As far as we know, there have been no studies on how the complexity of standard enumeration varies when the vectors in the input basis are permuted. To motivate the work that follows, we first present the results of some experiments showing that the number of nodes in the search tree during full enumeration is highly sensitive to the order of the basis vectors.

The lattice we use for the demonstration is Darmstadt's SVP40 challenge [105], generated from seed 0. All SVPn lattices from [105] have dimension n. The experiment was performed as follows: first, we ran two BKZ-reductions on the SVP40 lattice, one with block size 10, and one with block size 20. Then we performed full enumeration on each of the two BKZ-reduced lattices, counting the number of nodes in the search tree. Next, we randomly permuted the two BKZ-reduced bases 20 times each, and ran full enumeration on all of them. The average number of nodes in the search trees for the randomized bases are shown in Table 4.1, together with the maximum and minimum numbers observed, and the standard deviation.

When using extreme pruning, in case of failure to find the shortest vector, one permutes the basis vectors and runs the algorithm again. From the few trials in Table 4.1, we see that the order of the basis vectors has a big impact on the size of the enumeration search tree. The standard deviation is of similar size as the average, showing that the sizes of the search trees vary greatly with the permutation.

Another interesting observation is that the order of the reduced basis as obtained straight from BKZ is particularly good for enumeration. Enumerating the SVP40 challenge with the basis order given by BKZ-10 gives a tree with 5.968.085 nodes, and the order given by BKZ-20 gives a tree with 1.232.737 nodes, significantly smaller than the numbers observed for any of the random permutations.

4.5.2 Strategy for selecting an order for the basis vectors

Basic enumeration assumes the μ-matrix is computed once and for all before actual enumeration starts, but this is not strictly necessary. We can set every basis vector $\boldsymbol{b}_i$ in the basis as the last one, recompute the μ-matrix, and find the interval of possible coefficients for I_i. Doing this allows us to make a choice of which vector to first fix the coefficient for. For instance, we may select the basis vector giving the shortest interval as the first one to branch for.

The strategy we use for choosing the order of the basis vectors follows a greedy approach: we always choose the next index i for the coefficient v_i as the one with the

shortest interval I_i. The rationale for this strategy can be explained by the following lemma.

Lemma 4.5.1. *Let $J_1 \subseteq J_2 \subseteq (\{1, \ldots, n\} \setminus \{i\})$. Let $I_i(J_1)$ be the interval for v_i after values of v_j for $j \in J_1$ have been fixed, and let $I_i(J_2)$ be the interval for v_i after some additional v_j with $j \in J_2 \setminus J_1$ have been fixed. Then $|I_i(J_1)| \geq |I_i(J_2)|$.*

Proof. From Equation (4.1), we see that the length of $I_i(J_1)$ is determined by the sum

$$\sum_{j=k+1}^{n} \left(v_j + \sum_{i=j+1}^{n} \mu_{i,j} v_i \right)^2 \|\boldsymbol{b}_j^*\|^2, \tag{4.2}$$

while the center of the interval is determined by

$$\sum_{i=k+1}^{n} \mu_{i,k} v_i.$$

When we branch in an unspecified order, (4.2) can be written as

$$\sum_{j \in J_1} t_j^2,$$

where t_j are terms decided by the specific order in which the indices in J_1 are chosen. The larger this sum, the smaller $|I_i(J_1)|$ will be. The terms in the sum are all positive, so expanding with the extra terms to create the sum $\sum_{j \in J_2} t_j^2$ before branching for v_i can only decrease the length of I_i. Hence $|I_i(J_1)| \geq |I_i(J_2)|$. □

The point of Lemma 4.5.1 is to show that while basic enumeration tries all possible values in the computed range, there is a possibility that for some chosen value of the coefficient v_k in the range I_k, the sub tree enumerated for that particular chosen value might have either: (i) a much shorter range, or (ii) dead ends, i.e. an empty interval range for the next value. The basic enumeration algorithm tries to enumerate all possible values is some range until it encounters a dead end.

Lemma 4.5.1 shows that the longer we wait to select a particular v_i to branch for, the shorter its interval I_i will become. The idea for the branching strategy is that intervals that are long when few v_j have been selected will become short by the time the algorithm is forced to branch on them. This will lead to relatively small search trees.

One way to more easily see this is in the case when one interval I_i becomes empty after fixing some of the values v_j for $j \in J$. Suppose that the branching order has been fixed from the start, that the values of v_j have been fixed, and that I_i is empty for this choice of v_j, but v_i is only to be branched for after another 10 values v_k have been fixed. Even though it is clear that all choices of values for v_k will lead to a dead end, the traditional enumeration algorithm will try all of them before backtracking away from this sub-tree. By always selecting the next v_i to branch for as the one with the shortest interval, v_i will be selected as soon as $|I_i| = 0$ (the shortest length possible), and we will immediately backtrack to v_j with $j \in J$.

4.5.3 Cost vs effect for minimizing intervals

The drawback of checking which of the remaining indices to branch for is the extra work done in each node. If we compute an interval I_i using the μ-matrix of Gram–Schmidt coefficients, we in general have to recompute the μ-matrix as part of the process. The complexity for computing this matrix for one index is $\mathcal{O}(n^3)$ multiplications, and doing this for every remaining index not yet branched for gives an overall complexity of $\mathcal{O}(n^4)$ in each node. These complexities are quite high considering they have to be done for each node. However, they are still polynomial, and the number of nodes in a search tree is $2^{\mathcal{O}(n^2)}$, so if the reduction in the number of nodes is big enough, this can still lead to an improvement.

As we saw in Table 4.1, the number of nodes in a search tree without applying a minimizing strategy depends heavily on the order of the basis vectors. The order of the basis vectors does not matter when applying the minimizing strategy, as the vectors will be sorted as part of the enumeration routine. Hence it is hard to say anything in general about how large the effect of minimizing intervals will be, since it depends on how "lucky" the initial order of the vectors is.

4.5.4 Switch level

When many of the coefficients v_j have been assigned values (for $j \in J$), the effect of minimizing intervals for the relatively few remaining indices in $\{1, \ldots, n\} \setminus J$ is small. On the other hand, applying the minimizing strategy on the very first coefficients v_j to be fixed has a much greater effect. The number of large sub-trees rooted high up in the full tree when no ordering strategy is applied, becomes significantly smaller when minimizing intervals. In the extreme case of some interval becoming empty, the entire sub-tree is pruned away.

Thus we propose to only apply the minimizing strategy to the relatively few nodes at the highest levels of the search tree. This has the benefit of a relatively low cost and a potentially high effect. We refer to enumeration with the strategy of minimizing intervals for the first few levels of the tree as *hybrid enumeration*.

One parameter for hybrid enumeration is the level in the tree at which we switch from finding an optimal order based on minimizing intervals to classic enumeration where the basis is in some given and fixed order. We call this parameter the *switch level*.

More precisely, when we reach a node at the switch level, we do the following. We compute the interval lengths for the remaining indices one last time, and permute the remaining basis vectors according to these lengths. Indices with the shortest intervals will be branched for first. Then we perform ordinary enumeration for the sub-tree rooted at the current node, using this fixed order for the whole sub-tree. Pseudocode for hybrid enumeration is given in Algorithm 13. The intervals I_i in the experiments were computed using Algorithm 12.

For $B = \{\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n\}$, we regard the root node of the tree (the one at the top) to be at level n, and the vectors of $\mathcal{L}(B)$ to be at level 0. Note that we can run basic enumeration on the lattice by calling HybridEnumerate$(B, R, n+1, n)$. Calling HybridEnumerate(B, R, n, n) will also run basic enumeration, but the basis is first permuted according to the strategy of minimizing intervals. This makes it easy to estimate the benefit of using hybrid enumeration over basic enumeration.

Algorithm 13 HybridEnumerate(B, R, sl, l)

Input: The basis vectors $B = \{\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n\}$ of a lattice $\mathcal{L}$, a length bound R, the current level l, and the switch level sl.
Output: All vectors $\boldsymbol{s} \in \mathcal{L}$ with $\|\boldsymbol{s}\| \leq R$

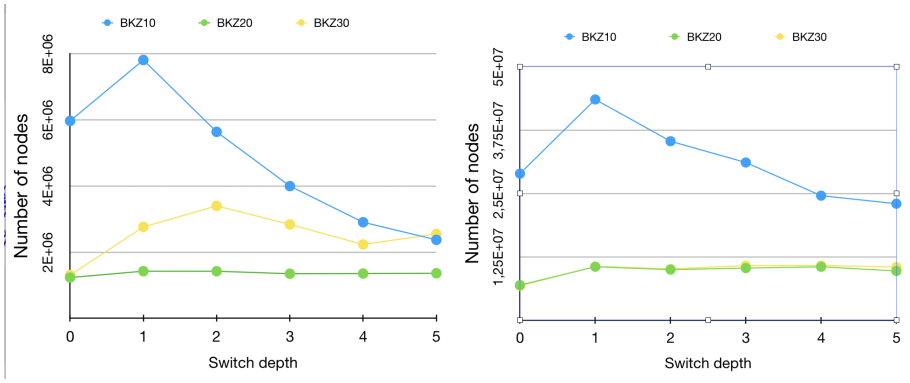
if $l > sl$ **then**
  $I_i \leftarrow$ shortest interval for all remaining $\boldsymbol{b}_i \in B$
  for $v_i \in I_i$ **do**
    $r \leftarrow$ min. length added to $\|\boldsymbol{s}\|$ due to choice of v_i
    HybridEnumerate$(B \setminus \{\boldsymbol{b}_i\}, R - r, sl, l - 1)$
  end for
end if
if $l = sl$ **then**
  Compute intervals I_j for all remaining $\boldsymbol{b}_j \in B$
  Sort B according to $|I_j|$, basis vectors on bottom of B has shortest intervals
  HybridEnumerate$(B, R, sl, l - 1)$
end if
if $l < sl$ **then**
  Run standard enumeration on B with length bound R
end if

4.5.5 Experiments

We have tested hybrid enumeration on several of the SVP challenges of [105] and counted the number of nodes that hybrid enumeration gives for different switch levels. The lattice bases were first reduced by running BKZ-β on them, for $\beta \in \{10, 20, 30\}$. For each reduced lattice, we ran hybrid enumeration with switch levels ranging from $n + 1$, equivalent to standard enumeration, to $n - 4$, counting the nodes in each search tree. The results are shown as plots in Figure 4.1, where "switch depth" refers to how many levels into the search tree we minimize intervals before switching to standard enumeration.

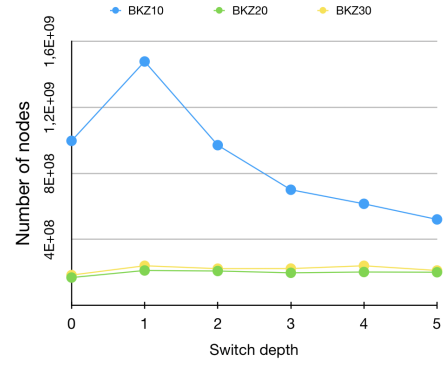
We observe a few trends in these plots. First, there is not much difference between BKZ-20 and BKZ-30 in terms of the quality of the bases. Both of them give search trees with approximately the same number of nodes after running standard enumeration, and applying the strategy of minimizing intervals does not change this by much. The order of the basis vectors given by hybrid enumeration yields search trees approximately as small as the order given by BKZ. This is in contrast to the random orders used for computing the numbers in Table 4.1, which shows a large increase in the number of nodes. Hence the strategy of sorting the basis vectors according to interval lengths is clearly a good approach for permuting the basis vectors when iteratively running extreme pruning.

For the BKZ-10 reduced bases, we see a much bigger effect. First, we see that BKZ-10 gives a significantly weaker reduction than BKZ-20 or BKZ-30, leading to larger enumeration search trees. The order as given by BKZ-10 is still good for enumeration, and doing one initial sorting of the basis according to interval lengths (switch level n) increases the search tree. However, lowering the switch level has a clear impact and significantly reduces the number of nodes in the search tree beyond the low number of nodes given by the initial BKZ-order.

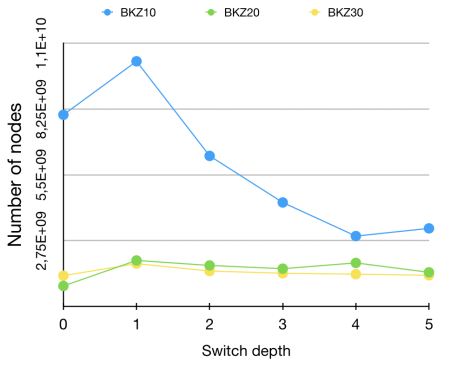


(a) SVP40

(b) SVP46



(c) SVP50



(d) SVP54

Figure 4.1: Number of nodes using hybrid enumeration on lattice bases pre-processed with BKZ-β for $\beta \in \{10, 20, 30\}$.

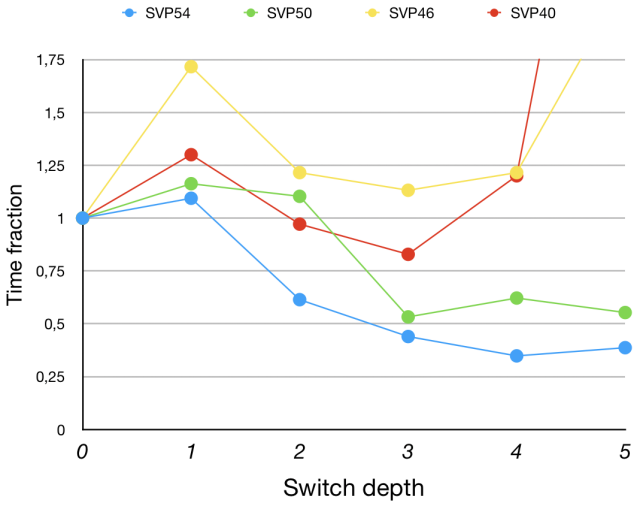


Figure 4.2: Fraction of time taken for doing full hybrid enumeration on BKZ-10 reduced lattice bases, compared to time taken for standard enumeration.

Of course, what matters in the end for a lattice enumeration algorithm is its complexity, measured in the actual time taken. We recorded the times measured in all the experiments, to see if the extra work done in the nodes at and above the switch level is worth the effort. For the enumeration of BKZ-20 and BKZ-30 reduced bases, it is clearly not worth the effort as the number of nodes slightly increases for the various switch levels. For enumerating the lattices only reduced by BKZ-10, there is a significant decrease in the number of nodes as the switch level decreases. The question is whether this is enough to make up for the $\mathcal{O}(n^4)$ operations done in each node at and above the switch level.

In Figure 4.2 we have plotted the fraction of time needed for enumerating the four lattices we have used as compared to standard enumeration, i.e. switch depth 0. The experiments were coded in C++ and ran on a DELL computer running Linux with two 2.8 GHz AMD EPYC 7451 24-Core processor and 188 GB of RAM.

We observe a few things from Figure 4.2. First, except for SVP46, the time it takes to do hybrid enumeration is less than the time for doing standard enumeration for some values of the switch level. Using switch level n leads to a longer running time due to an increase in the number of nodes. For a larger switch depth, the reduction in the number of nodes is worth the extra work done in the few nodes at the top. Second, for the bigger lattices, the time saved is largest with full hybrid enumeration, with the computation for SVP54 using switch level 51 only taking 34.8% of the time it takes to do full standard enumeration. Third, we also see that there is an optimal switch level. For SVP40 and SVP50, hybrid enumeration takes longer for switch level $n-3$ than for $n-2$, even though the number of nodes is less for switch level $n-3$. The reduction in the number of nodes is thus not worth the extra work performed for all nodes on level $n-3$.

Figure 4.2 pertains only to BKZ-10 reduced bases, and for better BKZ reductions we do not demonstrate an improvement in running time. However, the lattices we

are able to do full enumeration for in practice have dimensions in the range from 40 to 60, and a block size of 20 and 30 when running BKZ is then a large portion of that. We see in the plots that there is not much difference between BKZ-20 and BKZ-30 reduced bases, and there is hardly any improvement to be done for these cases. They appear to be quite optimal from the start.

We conjecture that for higher dimensions, like $n = 150$, BKZ-30 would not give an optimally reduced basis, and that hybrid enumeration would then show the same improvements as we see with the BKZ-10 reduced bases in our experiments. All in all, we claim that if one wants to do full enumeration on large lattices that are not optimally reduced (meaning that running BKZ with larger block sizes would improve the quality of the basis further), then hybrid enumeration will be faster than standard enumeration. However, the speedup might very well be asymptotically negligible, as the experiments from SVP50 and SVP54 indicate that the enumeration time is reduced only by a factor of 2 and 3, respectively.

4.5.6 Comparison between extreme pruning and hybrid enumeration

Lattice	Run time (sec)	# of vectors found by extreme pruning	# of vectors found by hybrid enumeration
SVP40	40	1	2
SVP42	120	5	5
SVP44	408	6	6
SVP46	1167	9	10
SVP48	2893	7	9
SVP50	9260	5	5
SVP52	30931	6	9
SVP54	65810	8	11

Table 4.2: Number of unique short vectors found by hybrid enumeration and extreme pruning.

To compare the efficiency of hybrid enumeration with that of extreme pruning, we ran experiments on the Darmstadt lattices SVPn with an even dimension for n between 40 and 54. In the experiment we searched for all vectors of length less than the bound given by [105], i.e. all vectors within 5% of the Gaussian heuristic:

$$R \leq 1.05 \cdot \frac{\Gamma(n/2+1)^{1/n}}{\sqrt{\pi}} \cdot |\det(L)|^{1/n}.$$

We first ran hybrid enumeration with the optimal switch level, and measured its running time for each instance. Then, we iterated extreme pruning on the same instance, but limited its running time to the same time that hybrid enumeration used. Finally, we compared the number of unique short vectors found by both methods.

We observe that in many cases, extreme pruning misses solutions that hybrid enumeration successfully finds. This demonstrates that given the same running time, it is possible to obtain more solutions using hybrid enumeration.

Our experimental results are summarized in Table 4.2. The particular lattices given in the first column were taken from [105], and the running time (in seconds) of hybrid enumeration applied to these lattices are given in the second column. From the table, we can see that extreme pruning misses solutions for dimensions $n \in \{40, 46, 48, 52, 54\}$.

4.6 Sign-based pruning

The second technique we provide is to estimate the signs of each v_k. The main idea behind the algorithm is to exploit the dot-product function which contains information about the length and angle between the basis vectors. Given two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, if the angle between them is less than 90 degrees, then their sum $\boldsymbol{a} + \boldsymbol{b}$ is longer (in terms of norm) than both $\boldsymbol{a}$ and $\boldsymbol{b}$, and their difference $\boldsymbol{a} - \boldsymbol{b}$ is shorter than at least one of $\boldsymbol{a}$ and $\boldsymbol{b}$. To get a short vector, we thus need to subtract one from another, which implies that the sign of these vectors are opposite with respect to each other. Similarly, when the angle between them is more than 90 degrees, then addition gives a short vector, so their relative signs should be the same.

We generalize this observation to n vectors, developing a method for estimating the signs of each coefficient v_k together with a confidence measure for each estimate. We then give a pruning strategy using which the interval computed for each v_k is cut down using the estimate of the sign and confidence factor. Unlike other pruning methods, this leads to a one-sided pruning where we only cut away a portion of possible values for v_k for which the sign is believed to be wrong.

4.6.1 Sign-estimation

Going back to the expansion of a shortest vector in terms of the basis vectors $\boldsymbol{s} = \sum_{i=1}^{n} v_i \boldsymbol{b}_i$, an enumeration algorithm computes possible values for each coefficient v_i. The equation for computing the coefficients v_i indicates that the range I_i for v_i will contain both positive and negative values when the absolute value of the center of I_i is smaller than the length given by (4.2). As both $\boldsymbol{s}$ and $-\boldsymbol{s}$ are shortest vectors, we are content in finding either of those. If we could know a priori the sign of these integers (that is, whether $v_i \leq 0$ or $v_i \geq 0$), we could discard appropriate values from I_i, making the enumeration tree smaller. This would effectively provide us with another strategy for pruning. In this subsection, we describe an algorithm for making educated guesses for the signs of these coefficients and how to use them for pruning.

First, we show how to compute the signs of the coefficients of the shortest vector when the dimension of the given lattice is only 2. Let us consider a lattice in 2 dimensions with basis vectors $\{\boldsymbol{b}_1, \boldsymbol{b}_2\}$. If $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are obtuse to each other, i.e. the angle between them is more than 90 degrees, then a shortest vector $\boldsymbol{s} = v_1\boldsymbol{b}_1 + v_2\boldsymbol{b}_2$ can only be obtained if the signs of v_1 and v_2 are the same. Similarly, if they are acute to each other, i.e. the angle between them is less than 90 degrees, a shortest vector can only be obtained if the signs of v_1 and v_2 are opposite to each other. It is easy to see this, since a (positive) sum of two vectors pointing in approximately the same direction can only increase in length.

To extend this observation to higher dimensions, we consider the inner product matrix $M = \boldsymbol{B}\boldsymbol{B}^T$, where $M_{ij} = \langle \boldsymbol{b}_i, \boldsymbol{b}_j \rangle$. Two vectors have a positive inner product

when the angle between them is less than 90 degrees, and a negative inner product when the angle between them is larger than 90 degrees. Moreover, the magnitude of $\langle \boldsymbol{b}_i, \boldsymbol{b}_j \rangle$ relative to $\langle \|\boldsymbol{b}_i\|, \|\boldsymbol{b}_j\| \rangle$ is a measure of how parallel or anti-parallel $\boldsymbol{b}_i$ and $\boldsymbol{b}_j$ are.

The algorithm for computing the estimated sign of coefficients is shown in Algorithm 14. The algorithm computes a vector $\boldsymbol{\sigma}$ of signs with entries $+1$ or -1. The signs of the coefficients v_i are computed one at a time, and the estimated sign of v_i depends on the signs of coefficients that have already been computed. Intuitively, the algorithm compares each basis vector with some reference vector to estimate the sign of the corresponding coefficient.

The sign of the first basis vector $\boldsymbol{b}_1$ is set to be positive by default, so $\sigma_1 = +1$. This can be assumed without loss of generality since both $\boldsymbol{s}$ and $-\boldsymbol{s}$ are shortest vectors and at least one of them must have non-negative v_1. The vector $\boldsymbol{b}_1$ is set as the reference vector $\boldsymbol{a}$ for the next basis vector. The first row of M contains the inner product of $\boldsymbol{b}_1 = \boldsymbol{a}$ with all the other basis vectors. The basis vector with the largest inner product in absolute value is both a relatively long vector, and makes an angle close to 0 or 180 degrees with $\boldsymbol{b}_1$. Let us denote this vector by $\boldsymbol{b}_i$. Then the sign of v_i is set to -1 if $M_{1,i} > 0$, and otherwise σ_i is set to $+1$. The reference vector is updated to $\boldsymbol{a} = \boldsymbol{a} + \sigma_i \boldsymbol{b}_i$.

Now we want to find a basis vector which is most parallel or anti-parallel to $\boldsymbol{a}$. To do this, we look at the largest entry in the vector $D = M_1 + \sigma_i M_i$, where M_1 is the top row of M and M_i is the i-th row of M. The largest entry in absolute value in D (except for the ones with index 1 and i) indicates the third vector, say $\boldsymbol{b}_j$, for which the sign is to be estimated. If $D_j > 0$ then $\sigma_j = -1$, and if $D_j \leq 0$, then $\sigma_j = +1$. The vector $\sigma_j \boldsymbol{b}_j$ is added to $\boldsymbol{a}$ and D is updated to $D = D + \sigma_j M_j$. The proceeds in the same manner until the signs for all basis vectors have been estimated.

The signs computed in Algorithm 14 are not necessarily correct for a shortest vector. For each variable v_i, we compute a number $0 \leq \gamma_i \leq 1$ to denote how confident we are that the computed σ_i is correct. When $\gamma_i = 1$, we are certain that the corresponding σ_i is correct, and $\gamma_i = 0$ means that we have no knowledge of whether the sign for v_i should be positive or negative. We compute the confidence values of the estimated signs as follows. Let $J \subset \{1, \ldots, n\}$ be the set of indices for which values have been fixed, and let the reference vector be $\boldsymbol{a} = \sum_{j \in J} \sigma_j \boldsymbol{b}_j$. Then the confidence value for the σ_i-estimate is given as $\gamma_i = |\frac{\langle \boldsymbol{a}, \boldsymbol{b}_i \rangle}{\langle \|\boldsymbol{a}\|, \|\boldsymbol{b}_i\| \rangle}|$.

The intuition behind this measure of confidence is that if two vectors are very close to being parallel, then having the same sign on the coefficients of these vectors will always lead to their sum being a longer vector that points approximately in the same direction as the other two. In order to be part of a short vector $\boldsymbol{s}$, the other basis vectors must be able to offset this long vector. If the signs of the coefficients are opposite, a sum of the two approximately parallel basis vectors would be much shorter. It is easier to sufficiently offset a short vector than a long one in order to find the shortest vector overall.

When two vectors are close to being parallel, then $\frac{\langle \boldsymbol{a}, \boldsymbol{b}_i \rangle}{\langle \|\boldsymbol{a}\|, \|\boldsymbol{b}_i\| \rangle}$ is close to being 1, and when they are close to being anti-parallel, then $\frac{\langle \boldsymbol{a}, \boldsymbol{b}_i \rangle}{\langle \|\boldsymbol{a}, \|\boldsymbol{b}_i\| \rangle}$ is close to being -1. In both cases, $\gamma_i \approx 1$.

On the other hand, when $\boldsymbol{a}$ and $\boldsymbol{b}_i$ are close to orthogonal, i.e. $\langle \boldsymbol{a}, \boldsymbol{b}_i \rangle \approx 0$, then $\boldsymbol{a} + \boldsymbol{b}_i$ and $\boldsymbol{a} - \boldsymbol{b}_i$ will be of roughly equal lengths, and it is difficult to distinguish

Algorithm 14 ComputeSign(B)

Input: The basis vectors B of the lattice $\mathcal{L}$.
Output: A vector $\boldsymbol{\sigma}$ that contains the estimated sign of each coefficient v_i in $\boldsymbol{s} = \sum_i v_i \boldsymbol{b}_i$ where $\boldsymbol{s}$ is a shortest vector, and a vector $\boldsymbol{\gamma}$ of real values indicating the confidence for each estimate.

Compute dot-product matrix M such that $M_{ij} = \langle \boldsymbol{b}_i, \boldsymbol{b}_j \rangle$.
Initialize $D := M_1$ where M_1 is the top row of M.
Set $\sigma_1 = +1$ and $\gamma_1 = 1$.
Set reference lattice vector $\boldsymbol{a} = \boldsymbol{b}_1$
Set the counter $n_s = 1$.
while $n_s \leq n$ **do**
  Let i be the index of $\max\{\|D_j\|\sigma_j$ is not already fixed$\}$.
  if $D_i > 0$ **then**
    Set $\sigma_i = -1$
  else
    Set $\sigma_i = +1$
  end if
  Set $\boldsymbol{a} = \boldsymbol{a} + \sigma_i \boldsymbol{b}_i$
  Set $D = D + \sigma_i M_i$
  Compute $\gamma_i = |\frac{\langle \boldsymbol{a}, \boldsymbol{b}_i \rangle}{\langle \|\boldsymbol{a}\|, |\boldsymbol{b}_i\| \rangle}|$
  Set $n_s = n_s + 1$
end while

which of the two cases will be most easily offset by the other basis vectors. The confidence value will therefore be close to 0 in this case.

We now explain how to use the confidence values to prune intervals in the search tree.

4.6.2 Pruning intervals based on sign estimation

We can use the sign estimations and their confidence values to reduce the intervals computed for enumeration, while still maintaining a high probability that we do not prune away all shortest vectors.

For a node in the search tree where possible values for v_i are tried, let I_i be the interval computed for v_i. Let $I_i^+ = I_i \cap [0, \infty)$ and $I_i^- = I_i \cap (-\infty, 0]$ denote the positive and negative part of the interval, respectively. For an interval $I = [l, m]$ and a positive number $\alpha \in \mathbb{R}$, let us define the interval αI to be $[\alpha l, \alpha m]$. If $\sigma_i = -1$, then I_i is pruned to $I_i = (1 - \gamma_i) I_i^+ \cup I_i^-$. If $\sigma_i = +1$, then I_i is pruned to $I_i = (1 - \gamma_i) I_i^- \cup I_i^+$. In other words, we cut away a portion of the interval where we believe a correct value for v_i will not be found. The size of portion cut away is proportional to the confidence we have in our estimate.

Sign-based pruning does not depend on how the intervals are computed. This pruning strategy reduces the search tree as long as the given intervals are non-empty, and cuts away integer values whose sign is opposite to the estimated sign.

4.6.3 Experiments on sign-based pruning

We used a few of the SVP challenge lattices to test the sign-based pruning strategy. We measured both the reduction in the number of nodes in the search tree, and whether the pruning failed to find a shortest vector. The results are summarized in Table 4.3.

Lattice	Pre-processing	node fraction	shortest vector found
SVP40	BKZ10	0.670	yes
SVP40	BKZ20	0.745	yes
SVP40	BKZ30	0.665	yes
SVP46	BKZ10	0.682	yes
SVP46	BKZ20	0.750	yes
SVP46	BKZ30	0.800	yes

Table 4.3: Measure of effect of sign-based pruning. The node fraction is the number of nodes in pruned search tree compared to the number of nodes in the full enumeration search tree.

We see from Table 4.3 that in the experiments we never failed to find the shortest vector, and that the number of nodes was reduced by a modest but still significant fraction. One explanation for the modest node reduction is that we cut away the ends of the intervals which only takes away small subtrees from the whole enumeration tree. The values for v_i found at the ends of the intervals are those that consume much of the length limit R when selected, probably quickly leading to dead ends anyway. Cutting away these values may not prune away very large parts of the search tree. Still, it is worthwhile to apply sign-based pruning as it costs practically nothing in terms of extra complexity. The actual run times compared to standard enumeration are cut down by almost the same fraction as the reduction in the number of nodes.

4.7 Conclusion

Public key encryption schemes based on lattices are one of the most promising approaches for achieving post-quantum cryptography, and it is important to understand the hardness of the SVP problem on which they are based. Lattice enumeration plays a central role in the best known methods for solving SVP, so studying how to speed up lattice enumeration is important for assessing the security of lattice-based encryption. In this chapter we have explored two different ideas for speeding up lattice enumeration.

First, we looked at how permuting the basis vectors of a lattice affects the running time of the standard enumeration algorithm. We demonstrated that the particular order of the basis vectors has a big impact on the number of nodes in the search tree and the running time. Next, we identified particular permutations that give relatively small search trees. Dynamically finding the best permutations has a high cost on its own. In order to extract the best increase in performance from this, we only applied this strategy to the first few levels in the search tree.

The state of the art which we compare our work to is the standard enumeration technique and extreme pruning. Even though pruning strategies give better results, the reason why we tried to improve the standard enumeration technique is that both

extreme pruning and the BKZ algorithm use standard enumeration in trying to solve SVP. We were curious to see if we can improve standard enumeration by applying our ideas and whether this will have a positive impact on either the extreme pruning or BKZ algorithms.

To compare the efficiency of hybrid enumeration to standard enumeration, we ran both algorithms, and we observed a reduction in the number of nodes of the search tree using hybrid enumeration. Despite this, the complexity remained asymptotically $2^{\mathcal{O}(n^2)}$.

We noticed that if extreme pruning and hybrid enumeration are run for the same amount of time on the same input instance, in many cases extreme pruning misses some of the short vectors which are found by hybrid enumeration. This gives hybrid enumeration an advantage over extreme pruning if it is necessary to reliably find all short vectors within some time bound.

Secondly, we looked at the possibility of estimating the signs of the coefficients giving a shortest vector. We can only estimate the signs with some degree of confidence, but the estimates and the confidence values lead directly to a pruning strategy. This work can be compared to the current state of the art which is extreme pruning. Unlike the extreme pruning strategy which cuts away values from both ends of the interval of possible values for a coefficient v_i, sign-based pruning only cuts off values from one side of the interval, namely, the side where the values have the sign that we believe is wrong.

We ran experiments for sign-based pruning on top of both standard enumeration and extreme pruning. In the case of running sign-based pruning on the top of standard enumeration, we observed a reduction in the number of nodes in the search tree, but this reduction was not significant. However, we never failed to find the shortest vector using sign-based pruning. This may indicate that the pruning we employed from the confidence measure is not aggressive enough, and that larger parts of the intervals could be cut away without sacrificing too much accuracy in solving the SVP.

In the case of running sign-based pruning on the top of extreme pruning, unfortunately, our method did not lead to any improvement. Further studies of sign-based pruning is a topic for future work.

Chapter 5

Estimates on Class Numbers - Elliptic and Hyperelliptic Curves

5.1 Introduction

The arithmetic of quadratic fields, i.e. degree two extensions of the rational field $\mathbb{Q}$, has interesting applications in cryptography. Many public key cryptosystems are based on intractable computational problems in number theory, such as integer factorisation, discrete logarithms, etc. A number of problems involving the structure of the class groups of these fields are believed to be intractable. The class group of a number field K is the quotient of the multiplicative group of the fractional ideals by the subgroup consisting of the principal ideals. Intuitively, it measures how much the ring of integers in K deviates from being a principal ideal domain. Consequently, corresponding cryptographic implementations built on these problems could be considered secure given large enough parameters. Instances of such problems include key exchange protocols using imaginary quadratic fields [18] and real quadratic number fields [103], the NICE (New Ideal Coset Encryption) cryptosystem based on the hidden kernel problem [92], one-way functions based on ideal arithmetic in number fields, and the Diffie-Hellman problem [16]. It is also worth mentioning the discrete logarithm problem for class groups of imaginary quadratic fields, for which no efficient algorithm is known [17].

All of the problems mentioned above involve the computation of the class numbers, i.e. the number of elements in the class group, of imaginary quadratic extensions. Factoring the discriminant would allow one to compute the order of the elements of the class group. However, factoring the discriminant appears to be a hard problem in itself.

An algebraic curve is the set of zeros of a polynomial in two variables. An *elliptic curve* E defined over a field k is given by a cubic polynomial equation of the form $Y^2 = X^3 + a_2X^2 + a_4X + a_6$. The solutions (X, Y) to the above equation are the points of the curve E. The set of points of E, along with a point at infinity, forms an Abelian group, whose group operation is given by rational functions.

The *genus* g of a curve E is an integer associated to the curve which can be computed from the degree of the polynomial equation defining the curve. Informally, the genus describes the number of "holes" when the points of the curve are considered as a topological space over complex numbers. For example, an elliptic curve corresponds to a topological space in the form of a torus, and therefore has

genus 1.

See [116] for more background on elliptic curves, including the formal definitions of e.g. the group operation and the genus of a curve, which we have described above only informally for the sake of brevity and since we do not use them in our work.

Let E be an elliptic curve given by the equation $Y^2 = X^3 + a_2X^2 + a_4X + a_6$ over a field of characteristic not equal to 2. The theory of quadratic forms over the function field $k(X)$ of discriminant $D(X) = -X^3 + a_2X^2 + a_4X + a_6$ was developed by Hellegouarch [49]. In [48], Hellegouarch et al. established a connection between the elliptic curve E and the quadratic field $k(X)(\sqrt{D(X)})$. They showed that the ideal class group of $k(X)(\sqrt{D(X)})$ is isomorphic to the Jacobian, $J(k)$, of E. A Jacobian is an algebraic variety that can be associated to an algebraic curve. The formal definition of the Jacobian is beyond the scope of this text and is omitted here; see e.g. [116] for details. Intuitively, the motivation for considering the Jacobian is that the points on a given curve do not, in general, form a group, whereas the points of the Jacobian do. This allows one to more easily extract information about the original curve through the Jacobian.

Soleng [121] adapted this theory to the polynomial ring $\mathbb{Z}[X]$. In his paper, Soleng considers the rational points of the curve E, which he calls the set of *primitive points* of E. He then defines a family of homomorphisms from a certain finite index subgroup of the primitive points to the ideal class groups of a family of suitable orders in imaginary quadratic number fields. First, he proves a conjecture of Hellegouarch [49], which essentially says that the order of the ideal classes corresponding to the points of infinite order on the curve will approach infinity as the discriminant of the order tends to infinity. Second, he constructs a family of imaginary quadratic number fields whose ideal class group contains a subgroup isomorphic to the torsion group of the elliptic curve.

Elliptic curves can be generalized to hyperelliptic curves, which can have genus $g \geq 1$. A *hyperelliptic curve* of genus $g \geq 1$ over a field k corresponds to an equation of the form $y^2 + h(x)y = f(x)$, where $f(x)$ and $h(x)$ are polynomials over k with $\deg(f) = 2g + 1$, and $\deg(h) \leq g$. The hyperelliptic curves of genus $g = 1$ are precisely the elliptic curves. See [82] for more background on hyperelliptic curves.

The generalization of Soleng's result to hyperelliptic curves was obtained by Jean Gillibert [39]. In our work, we establish a bound on an effective estimate (defined below) of these results in both the elliptic and hyperelliptic curve cases, and provide a bound on the effective estimate for the class numbers of the corresponding family of imaginary quadratic fields.

In this chapter we consider the results of Soleng [121] and Gillibert [39] regarding certain families of imaginary quadratic extensions arising out of some natural homomorphisms in the arithmetic of elliptic curves and hyperelliptic curves. These results show that the class groups will become arbitrarily big when a parameter n tends to plus or minus infinity.

The objective of this chapter is to derive a bound on the effective estimate for the orders of the class groups of a family of imaginary quadratic number fields. That is, we estimate how small or large the parameter n needs to be in order for the associated class group to have size greater than some pre-defined value M. At present we don't see any immediate application of this result to the field of cryptography; the result thus remains of independent interest.

5.2 Effective estimate in the case of elliptic curves

Definition 5.2.1. (Effective estimate) Suppose $f(n)$ is a sequence of positive numbers tending to infinity. Given any positive real number L, an *effective estimate* is to find the smallest positive integer $N = N(L)$ depending on L, such that $f(n) > L$ for all $n > N$.

Our first problem is to obtain a bound on the effective estimate from the following theorem from [121].

Theorem 5.2.1 (Soleng)**.** *[121] Let E be an elliptic curve over $\mathbb{Q}$ defined by the equation $Y^2 = X^3 + a_2X^2 + a_4X + a_6$ with coefficients in $\mathbb{Z}$. Let $P = (A/C^2, B/C^3)$ be an integral point of infinite order on the curve. For a positive integer n, let I_n be the class of the ideal*

$$(A + nC^2, -Bk + \sqrt{-n^3 + a_2n^2 - a_4n + a_6})$$

in the order $R_n = \mathbb{Z}(\sqrt{-n^3 + a_2n^2 - a_4n + a_6})$, where k is an integer such that $kC^3 \equiv 1 \pmod{A + nC^2}$. Then, as n approaches inifinity, so does the order of I_n.

In general, the order R_n in Soleng's theorem is not the full ring of integers in the field $\mathbb{Q}(\sqrt{-n^3 + a_2n^2 - a_4n + a_6})$. Nevertheless, results of Hooley on square free cubic polynomials [53] imply that R_n is indeed the full ring of integers if n belongs to a congruence class Λ of positive density. Thus, for this congruence class of integers, Soleng's theorem does guarantee that the orders of the class groups of the full ring of integers in the corresponding quadratic extension tend to infinity.

Thus, given a constant $M > 0$, we want to find N such that the order of the ideal class I_n in R_n (provided by the homomorphism) is greater than M for $n > N$. Following the proof of the main result in [121] (given in the theorem below), we derive a concrete value for N which satisfies the above conditions.

Theorem 5.2.2. *(Soleng) [121] Suppose that $P_1 = (A_1/{C_1}^2, B_1/{C_1}^3)$ and $P_2 = (A_2/{C_2}^2, B_2/{C_2}^3)$ are two rational points on the elliptic curve $Y^2 = X^3 + a_2X^2 + a_4X + a_6$ such that $P_1 \neq P_2$. Given a positive integer n, we associate to P_1 and P_2 the following quadratic forms of the discriminant $D = 4(-n^3 + a_2n^2 + a_4n - a_6)$:*

$$f_1(X, Y) = (A_1 + nC_1{}^2)X^2 + 2k_1B_1XY + \tfrac{k_1^2B_1^2 - D/4}{A_1 + nC_1^2}Y^2,$$
$$f_2(X, Y) = (A_2 + nC_2{}^2)X^2 + 2k_2B_2XY + \tfrac{k_2^2B_2^2 - D/4}{A_2 + nC_2^2}Y^2,$$

where k_1 and k_2 are integers satisfying $k_iC_i^3 \equiv 1 \mod (A_i + C_i^2n)$ for $i = 1, 2$. Then, for large enough values of n, the two forms will be inequivalent and neither of them will be equivalent to the identity.

We now provide an estimate for a value N such that all $n > N$ have the properties described in Theorem 5.2.2. We do this by following the proof of the theorem, as shown below.

Lemma 5.2.3. *Assume the same notation as in Theorem 5.2.2. For any integer n satisfying*

$$n > \max\left\{3|a_2 + C_1^2C_2^2|, \sqrt{3|A_1C_2^2 + A_2C_1^2 - a_4|}, \sqrt[3]{3|A_1A_2 + a_6|}\right\},$$

the forms f_1 and f_2 are inequivalent and neither of them is equivalent to the identity.

Proof. According to Soleng's proof of Theorem 5.2.2, the two ideals in R_n (or, equivalently, the corresponding binary quadratic forms) are in distinct equivalence classes, provided that the following inequality holds:

$$(A_1 + C_1^2 n) \times (A_2 + C_2^2 n) < \frac{-D}{4} = (n^3 - a_2 n^2 + a_4 n - a_6).$$

We shall now find a condition on n for which this holds. Rearranging the terms, the inequality above can be written as

$$n^3 - (a_2 + C_1^2 C_2^2)n^2 + (a_4 - A_1 C_2^2 - A_2 C_1^2)n - (a_6 + A_1 A_2) > 0.$$

It is clear that the term n^3 will dominate, and make the left-hand side positive for some value of n depending on the constants $a_2, a_4, a_6, A_1, A_2, C_1$ and C_2. The inequality can be rewritten as

$$\left(\frac{1}{3}n^3 - (a_2 + C_1^2 C_2^2)n^2\right) + \left(\frac{1}{3}n^3 + (a_4 - A_1 C_2^2 - A_2 C_1^2)n\right) + \left(\frac{1}{3}n^3 - (a_6 + A_1 A_2)\right) > 0.$$

If n is large enough, all the expressions inside the parentheses above will be positive, and the inequality will be satisfied. The first expression is positive for $n > 3|a_2 + C_1^2 C_2^2|$, the second one is positive for $n > \sqrt{3|A_1 C_2^2 + A_2 C_1^2 - a_4|}$, and the third one is positive for $n > \sqrt[3]{3|A_1 A_2 + a_6|}$. So for

$$n > \max\left\{3|a_2 + C_1^2 C_2^2|, \sqrt{3|A_1 C_2^2 + A_2 C_1^2 - a_4|}, \sqrt[3]{3|A_1 A_2 + a_6|}\right\},$$

and by Theorem 5.2.2, the forms f_1 and f_2 are inequivalent to each other and to the identity.

□

Now we will provide an explicit estimate for n and I_n in Theorem 5.2.1. More precisely, given a point P and a positive integer $M > 0$, we shall find N such that for all $n > N$, the ideal class I_n from Theorem 5.2.1 has order strictly greater than M.

Let $P = (A/C^2, B/C^3)$ be a point on E such that $\gcd(A, B, C) = 1$. The *height* of P is defined as $H(P) = \max\{|A|, C^2\}$.

Theorem 5.2.4. *In the setting of Theorem 5.2.1, given a positive integer M, the ideal class I_n has order at least M provided that*

$$n > c_E H(P)^{M^2/2},$$

where c_E is a constant depending only on the curve E.

Proof. The map $P \mapsto I_n$ defined in Theorem 5.2.1 is a group morphism. It follows that the image of P has order strictly larger than M if and only if the image of MP is not zero, in other words, the quadratic form in Theorem 5.2.2 associated to MP is not equivalent to the trivial form $X^2 - DY^2$. Letting $MP = (A_M/C_M^2, B_M/C_M^3)$, this holds (by adapting the proof of Theorem 5.2.2 to this situation) if

$$(A_M + C_M^2 n) < \frac{-D}{4} = (n^3 - a_2 n^2 + a_4 n - a_6),$$

which, by the definition of height, is satisfied if

$$2nH(MP) < n^3 - a_2 n^2 + a_4 n - a_6.$$

Now, if n is large enough with respect to $|a_2|$, $|a_4|$ and $|a_6|$, it suffices to ensure that

$$2nH(MP) < \frac{n^3}{2},$$

i.e.

$$2\sqrt{H(MP)} < n.$$

On the other hand, it is well-known that there exist constants c_1 and c_2 [114, 115], depending only on E, such that, for all M,

$$c_1 H(P)^{M^2} \leq H(MP) \leq c_2 H(P)^{M^2}.$$

The above inequalities show that $H(MP) = \mathcal{O}(H(P)^{M^2})$. In particular, the condition $2\sqrt{H(MP)} < n$ is satisfied if

$$2\sqrt{c_2} H(P)^{M^2/2} < n.$$

□

5.3 Effective estimates in the case of hyperelliptic curves

The generalization of Soleng's construction to the case of hyperelliptic curves was done by Gillibert [39]. In his paper, Gillibert has proved a similar qualitative analogue of Soleng's result. The following theorem presents this generalization. In our work, we simply estimate the values of n_f, n_0 and n_1 given in the paper. For more background on the topic, we refer to reader to the original paper [39].

Theorem 5.3.1. *(Gillibert) Let $C : Y^2 = f(X)$ be a hyperelliptic curve of genus g and degree $2g + 1$. Let the point L on the Picard variety of C be given (as per Mumford representation) by the quadratic form $[\frac{A}{e}, \frac{2B}{e}, \frac{C}{e}]$ as in Lemma 3.7 in [39]. For integers n sufficiently negative ($n < \min\{n_f, n_0, n_1\}$) and belonging to the index set $\Lambda \subset \mathbb{Z}$ as defined in Lemma 4.3 of [39], the orders of the specific class group elements in $\mathbb{Q}(\sqrt{f(n)})$ for $n \in \Lambda$ tend to infinity.*

We obtain the following bound on the effective estimate from Gillibert's result. The values U, V and W given in the theorem are estimates for the values of n_f, n_0 and n_1 from Theorem 5.3.1, respectively.

Theorem 5.3.2. *Let M be a positive integer, $f(x) = x^{2g+1} + \alpha_0 x^{2g} + \ldots + \alpha_{2g}$ and $h(x) = \beta_0 x^d + \ldots + \beta_d$, $d \leq 2g$ be defined as in Lemma 4.3 in [39]. Let $(f \pm h) = x^{2g+1} + \gamma_0 x^{2g} + \ldots + \gamma_{2g}$. The orders of the elements in the ideal class*

groups in $\mathbb{Q}(\sqrt{f(n)})$ *as determined by Gillibert are greater than* M *when* $n \in \Lambda$, *and* $n < \min\{U, V, W\}$, *where*

$$\begin{aligned} U &= \min\{\lfloor -|\alpha_k(2g+1)|^{\frac{1}{k+1}} \rfloor | 0 \leq k \leq 2g\}, \\ V &= \min\{\lfloor -|\tfrac{M-d\beta_k}{\beta_0}|^{\frac{1}{k}} \rfloor | 1 \leq k \leq d\}, \\ W &= \min\{\lfloor -|\gamma_k(2g+1)|^{\frac{1}{k+1}} \rfloor | 0 \leq k \leq 2g\}. \end{aligned}$$

Remark: The results of Hooley et al. [53] imply that the set Λ is a subset of positive density which can be computed effectively. This implies the existence of an infinite sequence of families of imaginary quadratic fields having class groups whose orders are greater than M.

The proof for the estimates of n_f, n_0 and n_1 is presented in the following three subsections.

5.3.1 Proof of the estimate for n_f

Let $f(x) = x^{2g+1} + \alpha_0 x^{2g} + \ldots + \alpha_{2g}$ be a hyperelliptic curve of genus g. We have $\lim_{n \to -\infty} f(n) = -\infty$. Let $n_f \in \mathbb{Z}$ be the largest integer such that $f(n) < 0$ for all $n \leq n_f$.

There are in total $2g+2$ terms in f. By distributing the highest-degree term out on all other terms, the polynomial f can be written as

$$\begin{aligned} f &= \tfrac{x^{2g+1}}{2g+1}(2g+1) + (\alpha_0 x^{2g} + \ldots + \alpha_{2g}) \\ &= \left(\tfrac{x^{2g+1}}{2g+1} + \alpha_0 x^{2g}\right) + \left(\tfrac{x^{2g+1}}{2g+1} + \alpha_1 x^{2g-1}\right) + \ldots + \left(\tfrac{x^{2g+1}}{2g+1} + \alpha_{2g}\right). \end{aligned}$$

It is sufficient to ensure that the expression inside every bracket is negative in order to guarantee that f is negative. These expressions are negative for α_k if $\frac{x^{2g+1}}{2g+1} + \alpha_k x^{2g-k} < 0$ for all $k \in \{0, 1, 2, \ldots, 2g\}$.

From the above equation for α_k, we obtain by simple algebra:

$$\begin{aligned} \alpha_k x^{2g-k} &< \frac{-x^{2g+1}}{2g+1}, \\ -(2g+1)\alpha_k x^{2g-k} &> x^{2g+1}. \end{aligned}$$

We now examine two cases:

Case 1: *k is even:* Consider the inequality $-(2g+1)\alpha_k x^{2g-k} > x^{2g+1}$. When k is even, the exponent $2g-k$ is even and x^{2g-k} is positive. We get $-(2g+1)\alpha_k > x^{k+1}$ by dividing by x^{2g-k} on both sides. If $\alpha_k < 0$, then the inequality is trivially satisfied for any $x < 0$. If $\alpha_k > 0$, the inequality is satisfied for $x < -(\alpha_k(2g+1))^{\frac{1}{k+1}}$.

Case 2: *k is odd:* when k is odd, the exponent $2g-k$ is odd and x^{2g-k} becomes negative for negative x. We get $-(2g+1)\alpha_k < x^{k+1}$ by dividing by x^{2g-k} on both sides. If $\alpha_k > 0$, then the inequality is trivially satisfied for any $x < 0$. If $\alpha_k < 0$, the inequality is satisfied for $x < -|\alpha_k(2g+1)|^{\frac{1}{k+1}}$.

Let $U = \min\{\lfloor -|\alpha_k(2g+1)|^{\frac{1}{k+1}} \rfloor | : 0 \leq k \leq 2g\}$. Then U provides a bound on the effective estimate for n_f. Now, whenever $n \leq U$, we have $f(n) < 0$.

5.3.2 Proof of the estimate for n_0

Let the polynomial $h(x)$ be defined as $h(x) = d_L A(x)$ where $A(x)$ is defined in Lemma 4.1 in [39], and let n_0 be an integer which depends on the polynomial $h(x)$.

By Lemma 4.3 in [39], $deg(h) < deg(f)$. Let $h(x) = \beta_0 x^d + \beta_1 x^{d-1} + \ldots + \beta_d$ be a polynomial of degree $d < 2g+1$ with $\beta_0 \neq 0$. According to [39], we want $|h(x)| \geq M$, or, equivalently, $|\beta_0 x^d + \beta_1 x^{d-1} + \ldots + \beta_d| \geq M$. We distribute the dominating term x^d to all other terms:

$$|h(x)| = \left| \left(\frac{\beta_0 x^d}{d} + \beta_1 x^{d-1} \right) + \left(\frac{\beta_0 x^d}{d} + \beta_2 x^{d-2} \right) + \ldots + \left(\frac{\beta_0 x^d}{d} + \beta_d \right) \right| > M.$$

Assume without loss of generality that the leading term satisfies $\beta_0 x^d > 0$ for $x < 0$. We will find values of x for which $h(x) > M$, and similarly we will find values of x for which $h(x) < -M$ if $\beta_0 x^d < 0$.

We want to make the expression inside each bracket in the inequality above greater than $\frac{M}{d}$, which will ensure that $h(x) > M$. For $1 \leq k \leq d$, we get by simple calculation

$$\begin{array}{rcl} (\frac{\beta_0 x^d}{d} + \beta_k x^{d-k}) & > & \frac{M}{d}, \\ \beta_0 x^d + d\beta_k x^{d-k} & > & M, \\ \beta_0 x^k + d\beta_k & > & \frac{M}{x^{d-k}}. \end{array}$$

For x negative, with $|x^{d-k}| \geq 1$, it is enough to find x such that $\beta_0 x^k + d\beta_k > M$. We can rewrite this inequality as

$$\begin{array}{rcl} \beta_0 x^k & > & M - d\beta_k, \\ x^k & > & \frac{M - d\beta_k}{\beta_0}, \\ x & < & -|\frac{M - d\beta_k}{\beta_0}|^{\frac{1}{k}}. \end{array}$$

Take $x < -\max\{|\frac{M-d\beta_k}{\beta_0}|^{\frac{1}{k}} : 1 \leq k \leq d\}$, and let V be $\min\{\lfloor -|\frac{M-d\beta_k}{\beta_0}|^{\frac{1}{k}} \rfloor : 1 \leq k \leq d\}$. Then V is a bound on the effective estimate for n_0, i.e. whenever $n < V$, we have $|h(n)| \geq M$.

5.3.3 Proof of the estimate for n_1

According to [39], we need to find $n_1 \in \mathbb{Z}$ such that for all $n \leq n_1$ we have $\pm h(n) + f(n) < 0$. The proof of this estimate for n_1 is analogous to that of the estimate for n_f.

Here $(f \pm h)(n)$ is a polynomial of degree $2g+1$ which can be written as $(f \pm h)(x) = x^{2g+1} + \gamma_0 x^{2g} + \ldots + \gamma_{2g}$, where all the coefficients γ_k are of the form $\gamma_k = \alpha_k \pm \beta_k$. The estimate for n_1 can be derived in exactly the same way as that for n_f. Let

$$W = \min\{-\lfloor |\gamma_k (2g+1)|^{\frac{1}{k+1}} \rfloor : 0 \leq k \leq 2g\}.$$

Then W is a bound on the effective estimate for n_1, so that whenever $n < W$, we have $(f \pm h)(n) < 0$.

This completes the proof of Theorem 5.3.2. The numbers U, V, W are effectively computable in terms of the coefficients of f and h, and if we take $n < \min\{U, V, W\}$ and $n \in \Lambda$, the ideal class in $\mathbb{Q}(\sqrt{f(n)})$ determined by Theorem 5.3.1 has order greater than M.

5.3.4 Bounds in terms of height

It is appropriate to express the estimates U, V and W as a function of the height H of the polynomials $f(x)$ and $h(x)$ which represent the equations of the hyperelliptic curve and the point on the Jacobian, respectively. The *height* of any polynomial $P(x) = p_n x^n + p_{n-1}x^{n-1} + \ldots + p_0$ is defined as $H(P) = \max\{|p_k| : 0 \leq k \leq n\}$.

First, consider $U = \min_k\{\lfloor -|\alpha_k(2g+1)|^{\frac{1}{k+1}}\rfloor\}$. Clearly,

$$|\alpha_k(2g+1)|^{\frac{1}{k+1}} \leq |\alpha_k|(2g+1)$$

and

$$\max\{|\alpha_k(2g+1)|^{\frac{1}{k+1}}| : 0 \leq k \leq 2g\} \leq H(f)(2g+1)$$

So the bound U can also be given as $U = -H(f)(2g+1)$.

We have $V = \min\{\lfloor -|\frac{M-d\beta_k}{\beta_0}|^{\frac{1}{k}}\rfloor : 1 \leq k \leq d\}$, and since $\beta_0 \in \mathbb{Z}$, we have $\frac{1}{|\beta_0|} \leq 1$. So taking $V = -(M + dH(h))$ will be a bound.

Similarly as for U, we can take $W = -H(f \pm h)(2g+1)$ as a bound. Consequently, for

$$n < \min\{-(H(f)(2g+1), -M - dH(h), -H(f \pm h)(2g+1)\},$$

we can guarantee that the orders of the elements in the ideal classgroups in $\mathbb{Q}(\sqrt{f(n)})$ will be greater than M.

5.4 Conclusion

The work of Soleng and Gillibert shows the existence of a family of elements in the class groups of a sequence of imaginary quadratic fields (which arise from the arithmetic of elliptic and hyperelliptic curves) whose orders tend to infinity. In this chapter, we have obtained bounds on the effective estimates for the orders of these elements using the theory of binary quadratic forms. Even though we have not estimated how tight our bounds are, we believe that they can be improved further. Although hard problems are used in cryptography, we have not found any direct application of our work to cryptography, and for the time being, our results remain of independent interest.

Bibliography

[1] D. Aharonov and O. Regev. "Lattice Problems in NP cap coNP". In: *45th Symposium on Foundations of Computer Science Rome, Italy, Proceedings*. IEEE Computer Society, 2004, pp. 362–371. DOI: 10.1109/FOCS.2004.35. URL: https://doi.org/10.1109/FOCS.2004.35.

[2] M. Ajtai. "Generating Hard Instances of Lattice Problems". In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. STOC '96, pp. 99–108. DOI: 10.1145/237814.237838. URL: https://doi.org/10.1145/237814.237838.

[3] M. Ajtai, R. Kumar, and D. Sivakumar. "A Sieve Algorithm for the Shortest Lattice Vector Problem". In: *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*. STOC '01. Association for Computing Machinery, pp. 601–610. DOI: 10.1145/380752.380857. URL: https://doi.org/10.1145/380752.380857.

[4] S. B. Akers. "Binary decision diagrams". In: *IEEE Transactions on Computers* C-27.6 (1978), pp. 509–516.

[5] S. Arora and B. Barak. *Complexity theory: A modern approach*. Cambridge University Press, Cambridge, 2009.

[6] J. Asketorp. *Attacking RSA moduli with SAT solvers*. KTH, School of Computer Science and Communication (CSC), https://www.diva-portal.org/smash/get/diva2:769846/FULLTEXT01.pdf. 2014.

[7] C. Bachoc and P. Gaborit. "On Extremal Additive I_4 Codes of Length 10 to 18". In: *Electronic Notes in Discrete Mathematics* 6 (2001), pp. 55–64. DOI: 10.1016/S1571-0653(04)00157-X. URL: https://doi.org/10.1016/S1571-0653(04)00157-X.

[8] H. Bandelt and H. M. Mulder. "Distance-hereditary graphs". In: *Journal of Combinatorial Theory, Series B* 41.2 (1986), pp. 182–208. URL: https://doi.org/10.1016/0095-8956(86)90043-2.

[9] B. Beate and W. Ingo. "Improving the Variable Ordering of OBDDs Is NP-Complete". In: *IEEE Trans. Computers* 45.9 (1996), pp. 993–1002. DOI: 10.1109/12.537122. URL: https://doi.org/10.1109/12.537122.

[10] A. H. Beiler. *Recreations in the Theory of Numbers*. Dovers publications, 1964.

[11] E. R. Berlekamp, J. F. MacWilliams, and N. J. A. Sloane. "Gleason's theorem on self-dual codes". In: *IEEE Trans. Information Theory* 18.3 (1972), pp. 409–414. DOI: 10.1109/TIT.1972.1054817. URL: https://doi.org/10.1109/TIT.1972.1054817.

[12] A. Bouchet. “Graphic presentations of isotropic systems”. In: *Journal of Combinatorial Theory, Series B* 45.1 (1988), pp. 58–76. ISSN: 0095-8956. DOI: https://doi.org/10.1016/0095-8956(88)90055-X. URL: http://www.sciencedirect.com/science/article/pii/009589568890055X.

[13] A. Bouchet. “Isotropic Systems”. In: *European Journal of Combinatorics* 8.3 (1987), pp. 231–244. ISSN: 0195-6698. DOI: https://doi.org/10.1016/S0195-6698(87)80027-6. URL: http://www.sciencedirect.com/science/article/pii/S0195669887800276.

[14] R. E. Bryant. “Binary Decision Diagrams”. In: *Handbook of Model Checking*. 2018, pp. 191–217. DOI: 10.1007/978-3-319-10575-8_7. URL: https://doi.org/10.1007/978-3-319-10575-8_7.

[15] R. E. Bryant. “Graph-Based Algorithms for Boolean Function Manipulation”. In: *IEEE Transactions on Computers* 35.8 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819. URL: https://doi.org/10.1109/TC.1986.1676819.

[16] J. A. Buchmann and S. Paulus. “A One Way Function Based on Ideal Arithmetic in Number Fields”. In: *Advances in Cryptology - CRYPTO ’97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA*. 1997, pp. 385–394. DOI: 10.1007/BFb0052250. URL: https://doi.org/10.1007/BFb0052250.

[17] J. A. Buchmann, T. Takagi, and U. Vollmer. “Number Field Cryptography”. In: vol. 41. Fields Institute Communications 10. American Mathematical Society, 2004, pp. 111–121.

[18] J. A. Buchmann and H. C. Williams. “A Key-Exchange System Based on Imaginary Quadratic Fields”. In: *Journal of Cryptology* 1.2 (1988), pp. 107–118. DOI: 10.1007/BF02351719. URL: https://doi.org/10.1007/BF02351719.

[19] J. P. Buhler, H. W. Lenstra, and P. Carl. “Factoring integers with the number field sieve”. In: *The development of the number field sieve*. Springer Berlin Heidelberg, 1993, pp. 50–94. ISBN: 978-3-540-47892-8.

[20] R. A. Calderbank, E. M. Rains, P. W. Shor, and N. J. A. Sloane. “Quantum Error Correction Via Codes Over GF(4)”. In: *IEEE Trans. Information Theory* 44.4 (1998), pp. 1369–1387. DOI: 10.1109/18.681315. URL: https://doi.org/10.1109/18.681315.

[21] J. Cassels. *An Introduction to the Geometry of Numbers*. Classics in Mathematics. Springer Berlin Heidelberg, 2012. ISBN: 9783642620355. URL: https://books.google.no/books?id=XyVrCQAAQBAJ.

[22] Y. Chen and P. Q. Nguyen. “BKZ 2.0: Better Lattice Security Estimates”. In: *Advances in Cryptology – ASIACRYPT 2011*. Ed. by D. H. Lee and X. Wang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–20. ISBN: 978-3-642-25385-0.

[23] S. E. Claude. “A Symbolic Analysis of Relay and Switching Circuits”. In: *Transactions of the American Institute of Electrical Engineers* 57 (12 1938), pp. 713–723. DOI: 10.1109/T-AIEE.1938.5057767.

[24] S. E. Claude. “Communication Theory of Secrecy systems”. In: *Bell Systems Technical journal* 28 (1949), pp. 656–715.

[25] J. Conway and N. J. A. Sloane. *Sphere packings, Lattices and Groups*. Springer, New York, NY, 1999. DOI: https://doi.org/10.1007/978-1-4757-6568-7_7.

[26] N. Costes, J. P. Indrøy, and H. Raddum. *CryptaPath*. https://github.com/Simula-UiB/CryptaPath.

[27] L. E. Danielsen and M. G. Parker. “On the classification of all self-dual additive codes over $GF(4)$ of length up to 12”. In: *Journal of Combinatorial Theory, Series A* 113.7 (2006), pp. 1351–1367. URL: https://doi.org/10.1016/j.jcta.2005.12.004.

[28] D. Deutsch and R. Jozsa. “Rapid Solution of Problems by Quantum Computation”. In: *Proceedings of the Royal Society of London Series A* 439 (1992), pp. 553–558. DOI: 10.1098/rspa.1992.0167.

[29] U. Dieter. “How to calculate shortest vectors in a lattice”. In: *Mathematics of Computation* 29 (1975), pp. 827–833.

[30] W. Diffie and M. E. Hellman. “New directions in cryptography”. In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654.

[31] J. D. Dixon. “Asymptotically fast factorization of integers”. In: *Mathematics of computation* 36.153 (1981).

[32] R. Feynman. “Simulating physics with computers”. In: *International Journal of Theoretical Physics* 21 (1982), pp. 467–488.

[33] U. Fincke and M. Pohst. “A procedure for determining algebraic integers of given norm”. In: *Proceedings of the European Conference on Computer Algebra - EUROCAL 1983*. Vol. 162. Lecture Notes on Computer Science. Springer, 1983, pp. 194–202.

[34] U. Fincke and M. Pohst. “Improved methods for calculating vectors of short length in a lattice, including a complexity analysis”. In: *Mathematics of Computation* (44 1985), pp. 463–471.

[35] P. Gaborit, C. W. Huffman, J. Kim, and V. Pless. “On additive $GF(4)$ codes”. In: *Codes and Association Schemes, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, November 9-12, 1999*. Ed. by A. Barg and S. Litsyn. Vol. 56. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, 1999, pp. 135–149. DOI: 10.1090/dimacs/056/12. URL: https://doi.org/10.1090/dimacs/056/12.

[36] S. D. Galbraith. *Mathematics of Public Key Cryptography*. Cambridge University Press, 2012. ISBN: 1107013925.

[37] N. Gama, P. Q. Nguyen, and O. Regev. “Lattice Enumeration Using Extreme Pruning”. In: *Advances in Cryptology - EUROCRYPT 2010*. Ed. by H. Gilbert. Springer Berlin Heidelberg, 2010, pp. 257–278.

[38] C. F. Gauss and C. A. Arthur. *Disquisitiones Arithmeticae*. Yale University Press, 1965. URL: www.jstor.org/stable/j.ctt1cc2mnd.

[39] J. Gillibert. *From Picard groups of hyperelliptic curves to class groups of quadratic fields*. arXiv:1807.02823. 2018.

[40] D. Glynn, T. A. Gulliver, J. Maks, and M. Gupta. *The geometry of additive quantum codes*. Springer-Verlag, Jan. 2004.

[41] O. Goldreich and S. Goldwasser. "On the limits of nonapproximability of lattice problems". In: *Journal of Computer and System Sciences* 60.3 (2000), pp. 540–563.

[42] M. Grohe and P. Schweitzer. "Isomorphism Testing for Graphs of Bounded Rank Width". In: *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*. 2015, pp. 1010–1029.

[43] G. Hanrot. "LLL: A Tool for Effective Diophantine Approximation". In: *The LLL Algorithm: Survey and Applications*. Ed. by P. Q. Nguyen and B. Vallée. Springer Berlin Heidelberg, 2010, pp. 215–263. ISBN: 978-3-642-02295-1. DOI: 10.1007/978-3-642-02295-1_6. URL: https://doi.org/10.1007/978-3-642-02295-1_6.

[44] G. Hanrot, X. Pujol, and D. Stehlé. "Analyzing Blockwise Lattice Algorithms Using Dynamical Systems". In: *Advances in Cryptology – CRYPTO 2011*. Ed. by P. Rogaway. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 447–464. ISBN: 978-3-642-22792-9.

[45] W. B. Hart. "A one line factoring algorithm". In: *Journal of the Australian Mathematical Society* 92.1 (2012), pp. 61–69. DOI: 10.1017/S1446788712000146.

[46] M. Hein, J. Eisert, and H. J. Briegel. "Multiparty entanglement in graph states". In: *Physical Review A* 69 (6 2004). URL: https://link.aps.org/doi/10.1103/PhysRevA.69.062311.

[47] B. Helfrich. "Algorithms to construct Minkowski reduced and Hermite reduced lattice bases". In: *Theoretical Computer Science* 41 (1985), pp. 125–139.

[48] Y. Hellegouarch, D. Mc Quillan, and R. Paysant-Le Roux. "Unités de certains sous-anneaux des corps de fonctions algébriques". In: *Acta Arithmetica* 48.1 (1987), pp. 9–47. URL: http://eudml.org/doc/206046.

[49] Y. Hellegouarch. "Positive definite binary quadratic forms over $k[X]$". In: *Lecture Notes in Mathematics* 1380 (1989), pp. 93–119.

[50] C. Hermite. "Extraits de lettres de M. Ch. Hermite à M. Jacobi sur différents objects de la théorie des nombres." In: *Journal für die reine und angewandte Mathematik* 40 (1850), pp. 261–277. URL: http://eudml.org/doc/147462.

[51] A. Hodges and D. Hofstadter. *Alan Turing: The Enigma*. Walker and Company, 2000. ISBN: 0802775802.

[52] G. Höhn. "Self-dual codes over the Kleinian four group". In: *Mathematische Annalen* 327.2 (2003), pp. 227–255. ISSN: 1432-1807. DOI: 10.1007/s00208-003-0440-y. URL: https://doi.org/10.1007/s00208-003-0440-y.

[53] C. Hooley. "On the square-free values of cubic polynomials." In: *Journal für die reine und angewandte Mathematik* 229 (1968), pp. 147–154. URL: http://eudml.org/doc/150838.

[54] S. Horie and O. Watanabe. “Hard instance generation for SAT”. In: *Algorithms and Computation*. Ed. by H. W. Leong, H. Imai, and S. Jain. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 22–31.

[55] A. Joux and C. Pierrot. “Technical history of discrete logarithms in small characteristic finite fields - The road from subexponential to quasi-polynomial complexity”. In: *Designs, Codes and Cryptography* 78.1 (2016), pp. 73–85.

[56] R. Kannan. “Improved algorithms for integer programming and related lattice problems”. In: *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing - STOC '83*. ACM Press, 1983, pp. 99–108.

[57] K. Kensuke. *BDDs Naturally Represent Boolean Functions, and ZDDs Naturally Represent Sets of Sets*. 2018. arXiv: 1806.10261 [cs.LO].

[58] S. Khot. “Inapproximability Results for Computational Problems on Lattices”. In: *The LLL Algorithm - Survey and Applications*. Ed. by P. Q. Nguyen and B. Vallée. Information Security and Cryptography. Springer, 2010, pp. 453–473. DOI: 10.1007/978-3-642-02295-1_14. URL: https://doi.org/10.1007/978-3-642-02295-1%5C_14.

[59] D. Knuth. *The Art of Computer Programming*. Vol. 4. Addison-Wesley Professional, 2009.

[60] A. Korkine and G. Zolotareff. “Sur les formes quadratiques”. In: *Mathematische Annalen* 6.3 (1873), pp. 366–389.

[61] M. Kumar, H. Raddum, and S. Varadharajan. “Reducing Lattice Enumeration Search Trees”. In: *Infocommunications* XI.4 (2019), pp. 8–16.

[62] M. Kumar, S. Varadharajan, and H. Raddum. “Graphs and Self-dual Codes over $GF(4)$”. In: *presented at WCC 2019* (2019).

[63] J. C. Lagarias, H. W. Lenstra, and C.-P. Schnorr. “Korkin-Zolotarev bases and successive minima of a lattice and its reciprocal lattice”. In: *Combinatorica* 10.4 (1990), pp. 333–348.

[64] Lagrange. *Recherches d'arithmétique*. Nouveaux mémoires de l'Académie royale des sciences et belles-lettres de Berlin, années, Jan. 2004, pp. 695–795.

[65] C. Lee. “Representation of switching circuits by binary-decision programs”. In: *Bell System Technical Journal* 38 (1959), pp. 985–999.

[66] S. Lehman. “Factoring large integers”. In: *Mathematics of computation* 28.126 (1974), pp. 637–646.

[67] D. H. Lehmer and R. E. Powers. “On factoring large numbers”. In: *Bulletin of the American Mathematical Society* 37.10 (1931), pp. 770–776.

[68] D. H. Lehmer. “The Mechanical Combination of Linear Forms”. In: *The American Mathematical Monthly* 35.3 (1928), pp. 114–121. URL: http://www.jstor.org/stable/2299504.

[69] A. Lenstra, H. W. Lenstra, and L. Lovász. “Factoring polynomials with rational coefficients”. In: *Mathematische Annalen* 261 (4 1982), pp. 515–535.

[70] A. K. Lenstra, H. W. J. Lenstra, M. S. Manasse, and J. M. Pollard. "The Number Field Sieve". In: *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, Baltimore, Maryland, USA*. 1990, pp. 564–572. DOI: 10.1145/100216.100295. URL: https://doi.org/10.1145/100216.100295.

[71] H. W. Lenstra. "Integer Programming with a Fixed Number of Variables". In: *Mathematics of Operations Research* 8.4 (1983), pp. 538–548. ISSN: 0364765X, 15265471. URL: http://www.jstor.org/stable/3689168.

[72] W. H. J. Lenstra. "Factoring integers with elliptic curves". In: *Annals of Mathematics (2)* 126.3 (1987), pp. 649–673. DOI: 10.2307/1971363. URL: https://doi.org/10.2307/1971363.

[73] D. Lundén and E. Forsblom. *Factoring integers with parallel SAT solvers*. KTH Royal IUnstitutre of Technology, CSC, https://www.semanticscholar.org/paper/Factoring-integers-with-parallel-SAT-solvers-Lunden-Forsblom/387ddf84. 2015.

[74] V. d. N. Maarten, D. Jeroen, and D. M. Bart. "Graphical description of the action of local Clifford transformations on graph states". In: *Phys. Rev. A* 69 (2 2004), p. 022316. DOI: 10.1103/PhysRevA.69.022316. URL: https://link.aps.org/doi/10.1103/PhysRevA.69.022316.

[75] F. J. MacWilliams, A. M. Odlyzko, N. J. A. Sloane, and H. N. Ward. "Self-Dual Codes over GF(4)". In: *J. Comb. Theory, Ser. A* 25.3 (1978), pp. 288–318. DOI: 10.1016/0097-3165(78)90021-3. URL: https://doi.org/10.1016/0097-3165(78)90021-3.

[76] F. J. MacWilliams, N. J. A. Sloane, and J. G. Thompson. "On the Existence of a Projective Plane of Order 10". In: *J. Comb. Theory, Ser. A* 14.1 (1973), pp. 66–78. DOI: 10.1016/0097-3165(73)90064-2. URL: https://doi.org/10.1016/0097-3165(73)90064-2.

[77] F. J. MacWilliams and N. Sloane. *The Theory of Error-Correcting Codes*. North-Holland, 1977.

[78] C. L. Mallows, A. M. Odlyzko, and N. J. A. Sloane. "Upper Bounds for Modular Forms, Lattices, and Codes". In: *Journal of Algebra* 36 (1975), pp. 68–76.

[79] C. L. Mallows, A. M. Odlyzko, and N. J. A. Sloane. "An upper bound for self-dual codes (Corresp.)" In: *IEEE Trans. Information Theory* 21.1 (1975), p. 115. DOI: 10.1109/TIT.1975.1055319. URL: https://doi.org/10.1109/TIT.1975.1055319.

[80] C. L. Mallows and N. J. A. Sloane. "An Upper Bound for Self-Dual Codes". In: *Information and Control* 22.2 (1973), pp. 188–200. DOI: 10.1016/S0019-9958(73)90273-8. URL: https://doi.org/10.1016/S0019-9958(73)90273-8.

[81] A. May. "The LLL Algorithm - Using LLL-Reduction for Solving RSA and Factorization Problems". In: ed. by P. Q. Nguyen and B. Vallée. Springer Berlin Heidelberg, 2010, pp. 315–348. ISBN: 978-3-642-02295-1. DOI: 10.1007/978-3-642-02295-1_10. URL: https://doi.org/10.1007/978-3-642-02295-1_10.

[82] A. Menezes, R. Zuccherato, and Y.-H. Wu. *An elementary introduction to hyperelliptic curves*. Faculty of Mathematics, University of Waterloo, 1996.

[83] D. Micciancio and S. Goldwasser. *Complexity of Lattice Problems: A Cryptographic Perspective*. Vol. 671. Jan. 2002. DOI: 10.1007/978-1-4615-0897-7.

[84] D. Micciancio and P. Voulgaris. "A Deterministic Single Exponential Time Algorithm for Most Lattice Problems Based on Voronoi Cell Computations". In: *SIAM J. Comput.* 42.3 (2013), pp. 1364–1391. DOI: 10.1137/100811970. URL: https://doi.org/10.1137/100811970.

[85] S. Minato. "πDD: A New Decision Diagram for Efficient Problem Solving in Permutation Space". In: *Theory and Applications of Satisfiability Testing - SAT 2011*. 2011, pp. 90–104. DOI: 10.1007/978-3-642-21581-0_9. URL: https://doi.org/10.1007/978-3-642-21581-0_9.

[86] P. Q. Nguyen. "Public-Key Cryptanalysis". In: *Recent Trends in Cryptography*. Ed. by I. Luengo. Vol. 477. Contemporary Mathematics. AMS–RSME, 2009.

[87] P. Q. Nguyen. "Lattice Reduction Algorithms: Theory and Practice". In: *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*. Ed. by K. G. Paterson. Vol. 6632. Lecture Notes in Computer Science. Springer, 2011, pp. 2–6. DOI: 10.1007/978-3-642-20465-4_2. URL: https://doi.org/10.1007/978-3-642-20465-4%5C_2.

[88] A. M. Odlyzko. "The rise and fall of knapsack cryptosystems". In: *Cryptology and Computational Number Theory*. American Mathematical Society, 1990, pp. 75–88.

[89] O. Ore. "Number Theory and its History". In: *The Mathematical Gazette* 33.304 (1948). DOI: 10.2307/3610834.

[90] S. Oum. "Rank-width and vertex-minors". In: *Journal of Combinatorial Theory, Series B* 95.1 (2005), pp. 79–100. URL: https://doi.org/10.1016/j.jctb.2005.03.003.

[91] C. Paar and J. Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. 1st. Springer Publishing Company, Incorporated, 2009. ISBN: 3642041000.

[92] S. Paulus and T. Takagi. "A New Public-Key Cryptosystem over a Quadratic Order with Quadratic Decryption Time". In: *Journal of Cryptology* 13.2 (2000), pp. 263–272. DOI: 10.1007/s001459910010. URL: https://doi.org/10.1007/s001459910010.

[93] V. Pless and N. J. A. Sloane. "On the Classification and Enumeration of Self-Dual Codes". In: *J. Comb. Theory, Ser. A* 18.3 (1975), pp. 313–335. DOI: 10.1016/0097-3165(75)90042-4. URL: https://doi.org/10.1016/0097-3165(75)90042-4.

[94] M. Pohst. "On the Computation of Lattice Vectors of Minimal Length, Successive Minima and Reduced Bases with Applications". In: *SIGSAM Bull.* 15.1 (Feb. 1981), pp. 37–44. ISSN: 0163-5824. DOI: 10.1145/1089242.1089247. URL: https://doi.org/10.1145/1089242.1089247.

[95] J. M. Pollard. “A monte carlo method for factorization”. In: *BIT Numerical Mathematics* 15.3 (1975), pp. 331–334. DOI: 10.1007/BF01933667.

[96] J. M. Pollard. “Theorems on factorization and primality testing”. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 76.3 (1974), pp. 521–528. DOI: 10.1017/S0305004100049252.

[97] C. Pomerance. “The Quadratic Sieve Factoring Algorithm”. In: *Advances in Cryptology: Proceedings of EUROCRYPT 84*. 1984, pp. 169–182. DOI: 10.1007/3-540-39757-4_17. URL: https://doi.org/10.1007/3-540-39757-4%5C_17.

[98] H. Raddum and O. Kazymyrov. “Algebraic Attacks Using Binary Decision Diagrams”. In: *Cryptography and Information Security in the Balkans - First International Conference, BalkanCryptSec, Istanbul, Turkey*. 2014, pp. 40–54. DOI: 10.1007/978-3-319-21356-9_4. URL: https://doi.org/10.1007/978-3-319-21356-9_4.

[99] H. Raddum and S. Varadharajan. “Factorization Using Binary Decision Diagrams”. In: *Cryptography and Communications* 11.3 (2018), pp. 443–460.

[100] R. L. Rivest, A. Shamir, and L. Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM* (1978).

[101] R. Rivest and B. Kaliski. “RSA Problem”. In: *Encyclopedia of Cryptography and Security, 2nd Ed.* Springer, 2011, pp. 1065–1069. DOI: 10.1007/978-1-4419-5906-5_475. URL: https://doi.org/10.1007/978-1-4419-5906-5%5C_475.

[102] B. Rosser. “A generalization of the Euclidean algorithm to several dimensions”. In: *Duke Mathematical Journal* 9.1 (Mar. 1942), pp. 59–95. DOI: 10.1215/S0012-7094-42-00907-4. URL: https://doi.org/10.1215/S0012-7094-42-00907-4.

[103] R. Scheidler, J. A. Buchmann, and H. C. Williams. “A Key-Exchange Protocol Using Real Quadratic Fields”. In: *Journal of Cryptology* 7.3 (1994), pp. 171–199. DOI: 10.1007/BF02318548. URL: https://doi.org/10.1007/BF02318548.

[104] E. T. Schilling and H. Raddum. “Solving Compressed Right Hand Side Equation Systems with Linear Absorption”. In: *Sequences and Their Applications - SETA 2012 - 7th International Conference, Waterloo, ON, Canada*. 2012, pp. 291–302. DOI: 10.1007/978-3-642-30615-0_27. URL: https://doi.org/10.1007/978-3-642-30615-0_27.

[105] N. Schneider and N. Gama. *SVP Challenge*. https://www.latticechallenge.org/svp-challenge/index.php.

[106] C. P. Schnorr. “A hierarchy of polynomial time lattice basis reduction algorithms”. In: *Theoretical Computer Science* 53 (2 1987), pp. 201–224.

[107] C. P. Schnorr and M. Euchner. “Lattice basis reduction: Improved practical algorithms and solving subset sum problems”. In: *Mathematical Programming* 66.1 (1994), pp. 181–199.

[108] C. P. Schnorr and H. H. Hörner. "Attacking the Chor-Rivest Cryptosystem by Improved Lattice Reduction". In: *Advances in Cryptology - EUROCRYPT '95*. Ed. by L. C. Guillou and J.-J. Quisquater. Springer Berlin Heidelberg, 1995, pp. 1–12.

[109] C. Schnorr and M. Euchner. "Lattice basis reduction: Improved practical algorithms and solving subset sum problems". In: *Math. Program.* 66 (1994), pp. 181–199. DOI: 10.1007/BF01581144. URL: https://doi.org/10.1007/BF01581144.

[110] C. Schnorr and H. H. Helmut. "Attacking the Chor-Rivest Cryptosystem by Improved Lattice Reduction". In: *Electronic Colloquium on Computational Complexity (ECCC)* 2.26 (1995). URL: http://eccc.hpi-web.de/eccc-reports/1995/TR95-026/index.html.

[111] C. Shannon. "A mathematical theory of communication". In: *Bell system technical journal* 27.3 (1948), pp. 379–423.

[112] M. Shi, Y. Choie, A. Sharma, and P. Solé. *Codes and Modular Forms: A Dictionary*. World Scientific, 2020. DOI: 10.1142/11628.

[113] P. W. Shor. "Algorithms for quantum computation: discrete logarithms and factoring". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science* (1994), pp. 124–134.

[114] J. Silverman. "An introduction to height functions". In: *MSRI Workshop on Rational and Integral Points on Higher-Dimensional Varieties*. 2006.

[115] J. Silverman. "Computing heights on elliptic curves". In: 51 (July 1988), pp. 339–358. DOI: 10.1090/S0025-5718-1988-0942161-4.

[116] J. H. Silverman. *The arithmetic of elliptic curves*. Vol. 106. Springer Science & Business Media, 2009.

[117] S. Singh. *The Code Book: The Evolution of Secrecy from Mary, Queen of Scots, to Quantum Cryptography*. 1st. USA: Doubleday, 1999. ISBN: 0385495315.

[118] A. Siper, R. D. Farley, and C. Lombardo. "The Rise of Steganography". In: Proceedings of Student/Faculty Research Day, New York, 2005.

[119] D. Skidmore. *Efficiently representing the integer factorization problem using binary decision diagrams*. All Graduate Plan B and other Reports, 1043. https://digitalcommons.usu.edu/gradreports/1043. 2017.

[120] N. J. A. Sloane. "Further Connections Between Codes and Lattices". In: *Sphere Packings, Lattices and Groups*. Springer New York, 1993, pp. 181–205. ISBN: 978-1-4757-2249-9. DOI: 10.1007/978-1-4757-2249-9_7. URL: https://doi.org/10.1007/978-1-4757-2249-9_7.

[121] R. Soleng. "Homomorphism from the Group of Rational points on Elliptic curve to Class groups of Quadratic Number Fields". In: *Journal of Number Theory* 46 (2 1994), pp. 214–229.

[122] C. H. Thomas, L. E. Charles, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[123] V. D. Tonchev. "Error-correcting codes from graphs". In: *Discrete Mathematics* 257.2 (2002), pp. 549–557. ISSN: 0012-365X. DOI: https://doi.org/10.1016/S0012-365X(02)00513-7. URL: http://www.sciencedirect.com/science/article/pii/S0012365X02005137.

[124] R. Uehara and T. Uno. *Canonical tree representation of distance hereditary graphs and its applications*. Tech. rep. School of Information Science, Japan Advanced Institute of Science and Technology, 2006.

[125] S. Varadharajan. "Estimates on Class Numbers - Elliptic and Hyperelliptic Curves". In: *submitted to Journal of Algebra, Number Theory and Applications* (2019).

[126] Z. Varbanov. "Some new Results for Additive Self-Dual Codes over GF(4)". In: *Serdica Journal of Computing* 1.2 (2007), pp. 213–227. URL: http://eudml.org/doc/11421.

[127] A. Vardy. "The intractability of computing the minimum distance of a code". In: *IEEE Transactions on Information Theory* 43 (1997), pp. 1757–1766.

[128] G. Venkatesan. "Constructions of codes from number fields". In: *IEEE Transactions on Information Theory* 49.3 (2003), pp. 594–603.

[129] S. Wagstaff. *The joy of factoring*. American Mathematical Society, 2013.

[130] X. Wang, M. Liu, C. Tian, and J. Bi. "Improved Nguyen-Vidick heuristic sieve algorithm for shortest vector problem". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011*. 2011, pp. 1–9. DOI: 10.1145/1966913.1966915. URL: https://doi.org/10.1145/1966913.1966915.

[131] H. C. Williams and J. O. Shallit. "Factoring integers before computers". In: *Mathematics of computation, 1943–1993, Fifty Years of Computational Mathematics (W. Gautschi, ed.), American Mathematical Society* (1994).

[132] P. Woelfel. "Bounds on the OBDD-Size of Integer Multiplication via Universal Hashing". In: *J. Comput. Syst. Sci.* 71.4 (Nov. 2005), pp. 520–534. ISSN: 0022-0000. DOI: 10.1016/j.jcss.2005.05.004. URL: https://doi.org/10.1016/j.jcss.2005.05.004.

Graphic design: Communication Division, UiB / Print: Skipnes Kommunikasjon AS

uib.no

ISBN: 9788230869253 (print)
9788230858721 (PDF)