Lattice Sieving With G6K

Katrine Moksheim

Supervisors: Håvard Raddum and Morten Øygarden

Master's Thesis in Informatics
Secure and Reliable Communication
Department of Informatics
University of Bergen

June 1, 2023

Acknowledgements

First I would like to thank my two supervisors, Håvard Raddum and Morten Øygarden for their support, guidance, and encouragement in experimenting with G6K and writing of the thesis.

I would also like to give a huge thanks to Mads and Wendy for giving me feedback on my thesis and taking time out of their day to help me out.

At last, I want to thank my family and friends for keeping me motivated and inspired during my studies. Especially thanks to my friends at the study hall for making the year great and having some good crossword sessions.

Katrine Moksheim
June 1, 2023

Abstract

Recent advances in quantum computing threaten the cryptography we use today. This has led to a need for new cryptographic algorithms that are safe against quantum computers. The American standardization organization NIST has now chosen four quantum-safe algorithms in their process of finding new cryptographic standards. Three out of the four algorithms are based on the hardness of finding a shortest vector in a lattice. The biggest threat to such schemes is lattice reduction. One of the best tools used for lattice reduction is the G6K framework. In this thesis, we study sieving algorithms and lattice reduction strategies implemented in G6K.

After an introduction to cryptography, we go over the necessary preliminary lattice theory, important concepts, and related problems. Further, we look at lattice reduction where we study different approaches with a main focus on lattice sieving. We then explore the G6K framework, before finally performing some experiments using G6K.

The results we get often depend on what type of lattice we are working on. Our experiments show that it is still possible to improve G6K for solving the shortest vector problem for some lattice types.

Contents

List of Figures

Chapter 1

Introduction

1.1 Cryptography Through History

To be able to make messages secret, and not easily available to unwanted adversaries, has been a necessity for a very long time. We use cryptography to accomplish this, and methods used for it have evolved a lot over the years. We will now see how this evolution has been throughout our history.

In ancient Greece, they used a method where they wrote a message on a board and then added a wax layer on top of it. This made the message hidden and it can be seen as an example of what we call steganography. One of its weaknesses is that if people were aware of the method used to hide the message, the meaning of the message could easily be revealed. Since the message is only made secret due to hiding it, it means that anybody who finds it will be able to understand it. However, this method was not very secure, and therefore new methods evolved.

Moving forward to the Romans in Julius Caesar's time (100BC) a new strategy started to be used. Instead of hiding the message physically, they hid the actual meaning of the message. People who found the message would most likely not be able to read the true meaning behind the message since it all looked like gibberish. This is what cryptography is about.

Julius Caesar used a cipher, which we today call the Caesar cipher, on the messages he wanted to send. His idea was to shift the letters in the alphabet a certain amount of times to the right, and then write the messages with the new alphabet, see Figure 1.1. This technique is what we call a substitution cipher.

Alphabet:	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Cipher alphabet:	X Y Z A B C D E F G H I J K L M N O P Q R S T U V W
Message:	Hello world
Ciphertext:	Ebiil tloia

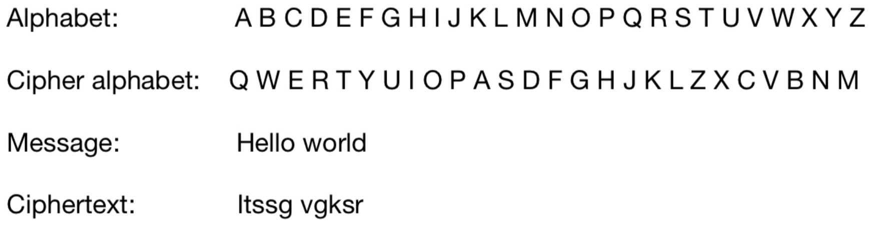
Figure 1.1: Caesar Cipher with three shifts

The message written with the cipher alphabet is called ciphertext. When the receiver deciphers the ciphertext, they need to know the number of shifts the sender has used to encrypt the message. This is called the cipher key. Additionally, we must assume that the receiver knows which cipher is used for them to easily decipher it. In other words, the receiver must know that Caesar cipher has been used to encrypt the message.

If we assume that the message is encrypted with Caesar cipher, would a person without any knowledge of the key be able to decipher the message? If we look at the Caesar cipher we will actually see that there are only 25 possible shifts of the alphabet, since there are only 26 letters (using the English alphabet as an example). As 25 different combinations is not a lot, one could brute force the ciphertext. This means trying all shifts of the alphabet until we get a plaintext that makes sense. Since the Caesar cipher is easily broken, it has not been considered safe.

Additionally, there was also an improvement made to the Caesar cipher, where one did not just do a cyclic shift of the alphabet but used an arbitrary permutation. This means taking a random order of the letters, and then pairing each letter with another letter in the alphabet, see Figure 1.2. As a result, there was was no longer only 25 different permutations of the alphabet, but so many that brute force by trying every combination of letters becomes very time-consuming and actually impossible to do in practice. Since all letters can be placed at every place it gives us 26! different shifts of the alphabet, hence it can take more than a lifetime to figure out the correct one by trial and error.

However, this method also has flaws that became more apparent as time passed. It turned out that this cipher could be broken by a new important discovery in cryptanalysis years later.



Alphabet:	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Cipher alphabet:	Q W E R T Y U I O P A S D F G H J K L Z X C V B N M
Message:	Hello world
Ciphertext:	Itssg vgksr

Figure 1.2: Improvement of Caesar cipher

The breakthrough in cryptanalysis happened however in the Islamic golden age. Cryptanalysis is about how we can break ciphers, or in other words how to find weaknesses in cryptography. Some theologians were doing work on analyzing the Quran and the Hadiths to find out if the texts were correct. They used frequency analysis on words and letters in the two texts to compare them. Al-Kindí, a mathematician and philosopher, then realized that this type of analysis also could be used to break ciphers. As each letter has a frequency of how often they usually appear, he found out that we can use this to check the frequency of letters in a ciphertext, and then decipher it by using the frequency analysis. It also turned out that all cryptography in use at this point was vulnerable to this type of analysis, so new methods needed to be found.

A consequence of having a weak cipher can be observed during the reign of Queen Mary and Queen Elizabeth 1 in the 1500s. Elizabeth was the reigning Queen of England, and her cousin Mary was the queen of Scotland. At that time England was considered a protestant country supported by Queen Elizabeth. However, there were some groups of people that meant England should be catholic, and this was also Queen Mary's wish. Queen Mary, together with a rebel group started to plot her way to the throne so that England could become catholic. All communications between Queen Mary and the rebels were encrypted, to avoid the content of the messages would get into the hands of their enemy. Sir Francis Walsingham, who was Queen Elizabeth's principal secretary, got his hands on the messages and was able, with the help of the best codebreaker in the country, to decipher the message. They could therefore expose Queen Mary and the plotters. As a consequence of the solid proof, they executed the plotters and had Queen Mary up for a trial, which concluded in her death. This means that Queen Mary lost her life due to a weak cipher [31, pp. 32-44].

In the late 1800s, a man named Auguste Kerckhoffs wrote down 6 principles on what he meant should be rules for any cryptosystem. He published the principles in two articles called *La Cryptographie Militaire* and was highly focused on how to make telegraphic systems secure. Even though these principles were written many years ago, some of them are still applicable to the cryptosystems we use today. The principles are ([25])

1. The system must be substantially, if not mathematically, undecipherable.
2. The system must not require secrecy and can be stolen by the enemy without causing trouble
3. It must be easy to communicate and remember the keys without requiring written notes, and it must also be easy to change or modify the keys with different participants.
4. The system ought to be compatible with telegraph communication.
5. The system must be portable, and its use must not require more than one person.
6. Finally, regarding the circumstances in which such system is applied, it must be easy to use and must neither require stress of mind nor the knowledge of a long series of rules.

Today it is the second principle that is the most important. This principle is often referred to as *Kerckhoffs principle*. The principle describes that the ciphers in use should not be held secret, but rather be in the public domain. Today this is the standard, and the only thing that is kept a secret is the keys that are used by the ciphers. This principle is also the earliest example of the term "key", and an understanding of the fact that we could make ciphers with only a small part of the scheme being secret (the key). Additionally, another important point from the mentioned principles is that a key could easily be changed. Today this is also regarded as important for a cryptographic scheme.

During the first world war we can see examples of how cryptography was an important aspect during this time. The year is 1914 and the radio has been taken into use. Now the military was not depending on the electric telegraph to send messages anymore. The benefit of the radio compared to the telegraph was that it was not dependent on cables, making it harder to sabotage. However, since the messages in radio travel through the air people could more easily intercept the signals, meaning that the information was not protected. Therefore they had to make the message into codes or ciphers to make it indecipherable. At this point, they still used substitution ciphers. Some of these were the Playfair cipher and the ADFGX cipher.

There were many ciphers that got deciphered during the first war, and one of them was the Zimmermann telegram in 1917. The Zimmermann telegram was a message for the German embassy in Mexico where the Germans tried to convince Mexico to attack the US and join forces with Germany. However, this message got decrypted and published all over the world. The US was not part of the war yet, but after this message got exposed they decided to join in as well [32]. Now Germany had more enemies at a time when they were already struggling. This is an example of a time when crucial and secret information often got decrypted by the other parties, and we can say that the codebreakers made more progress than the code makers.

When World War 2 began there was also some new technology that was widely used by the military when it came to cryptography. The new technology included machines that were used for creating advanced ciphers. These were called Enigma machines. Before the war began the Germans were already using an Enigma rotor to make a complex substitution cipher. At this time they had the most secure communication at this point in time. However, the Polish mathematician Marian Rejewski was able to break this enigma before the war. This was a huge breakthrough for cryptanalysis. Poland later decided to share this knowledge with France and Britain as the Germans began developing an improved enigma.

The Germans would continue developing the Enigma throughout World War 2, and the cryptanalysts were working hard on breaking it. Both the French and the British had considered the Enigma to be unbreakable but were now proved wrong by the Polish invention. They also started to see the benefits of having mathematicians as codebreakers, which had usually only consisted of linguistics and classicists in Britain. Bletchley Park was a top-secret place for the British codebreakers, and here the famous Alan Turing was a member. He continued the work of Rejewski and was able to come up with an idea that would be able to crack the enigma. Turing and his team were able to break the Enigma with the concept of a machine that turned out to be the first programmable computer. Now the Allies had an advantage as they could decipher almost everything the Germans were communicating. This is considered by many to have shortened the war drastically, as the allies kept it secret that they could read many of their rival's plans. It was in fact not revealed before decades after the war that they had been able to crack the enigma.

From the second world war and up to today there has been a massive evolution in technology. The computer as well as the internet has become a part of our daily life. We use it for almost everything, such as banking information, personal information, and emails. Since we have so much important and private information online it is important that this is protected. Up until World War 2 cryptography had been something that was developed in secret and mainly used by the military and governments. However, today it is a part of the public domain and is available to everyone. Hence we should have standards and requirements for cryptography.

Claude Shannon, who by many is considered to be the father of information theory, published his ideas of perfect secrecy in 1949 [29]. This was an important idea that the standardizations that were developed later took into strong consideration. These modern standardizations are described in Section 1.2, and include how we started to use mathematical problems that are hard to solve for a computer as a part of the encryption standards.

Today good encryption is regarded as important for our privacy and safety in our everyday use. We have standardizations that work well for today, but will these be sustainable for the future? The answer is most likely no. As demonstrated in Section 1.3, there are potential threats against today's standards if quantum computers get evolved on a big scale. So today we have already started to develop new and quantum-safe standardizations. However, these will be more described in Section 1.4.

1.2 Modern Cryptography

As mentioned before, cryptography is about how we are able to protect information using cryptographic primitives. A reason we want to protect information might be to prevent others to be able to read private messages. One of the methods used to do this is by using ciphers, which provide confidentiality. Ciphers consist of three different operations: key generation, encryption, and decryption. We can define ciphers, or encryption schemes, as follows [13, pp. 26-27].

Definition 1 *Ciphers consist of three operations and three spaces. Key generation is an algorithm based on probability, which will output a key based on a distribution. All possible keys the key generation algorithm can output are called the key-space and are denoted $\mathcal{K}$. A message m is part of the space $\mathcal{M}$, which contains all possible messages. The encryption algorithm takes a $k \in \mathcal{K}, m \in \mathcal{M}$ as input, and outputs a ciphertext. We denote the encryption as $c = Enc_k(m)$. All of the possible outputs of the encryption algorithm make up the ciphertext space $\mathcal{C}$. The decryption algorithm takes as input $c \in \mathcal{C}, k' \in \mathcal{K}$, where k' is determined by k, and outputs $m \in \mathcal{M}$. The notation of decryption is $m = dec_{k'}(c)$. We also require that $dec_{k'}(enc_k(m)) = m$ for all $m \in \mathcal{M}$ and $k \in \mathcal{K}$.*

Encryption and decryption behave a little differently. This is because, in the encryption algorithm, we base the output on probability. So encrypting the same message multiple times can generate different ciphertexts. The decryption on the other hand should always return the same plaintext as the original message, hence providing perfect correctness.

We can also see from the definition that both encryption and decryption use a key from the key-space. If they both use the same key we call it symmetric cryptography (see Figure 1.3). With this type of cryptography, we obtain confidentiality.

However, using only symmetric cryptography has its limits. Integrity, which we consider to be a basic need for good cryptography, is not given. By integrity, we mean that we should be certain that the message we receive has not been tampered with after it was sent. A solution to this is using message authentication codes (MAC). The goal of MAC is to prevent a third party from altering the messages without them going undetected. Today we have begun to use MAC and symmetric cryptography together

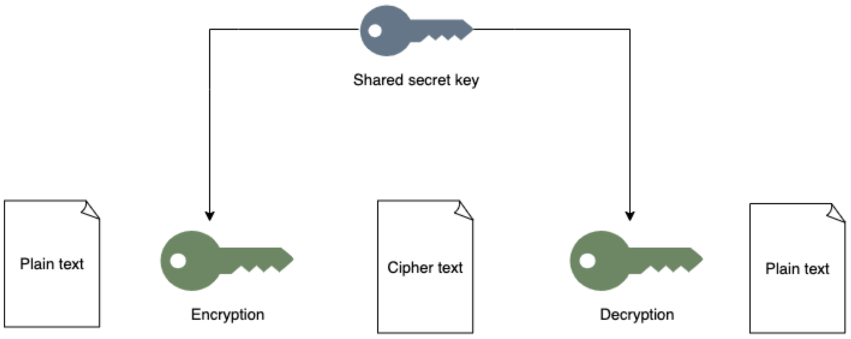


Figure 1.3: Symmetric cryptography

in one algorithm. This is called authenticated encryption and provides confidentiality and integrity in one cryptographic primitive.

Another difficulty for symmetric cryptography is signing. For signing to work one would need both sender and receiver to share a secret key. This can lead to problems, so therefore we want to use a method where we can verify the sender without having to do a key exchange first. We want anybody to be able to encrypt messages for a specific receiver, and one should also be able to do verification of the sender.

Hence we started using asymmetric cryptography, or public-key cryptography as it is also called (see Figure 1.4). This is based on the fact that we have a public key for encryption and a private key for decryption. So here we use two different keys from the key-space. By using public-key cryptography we also obtain the possibility to sign messages, and then again get authenticity.

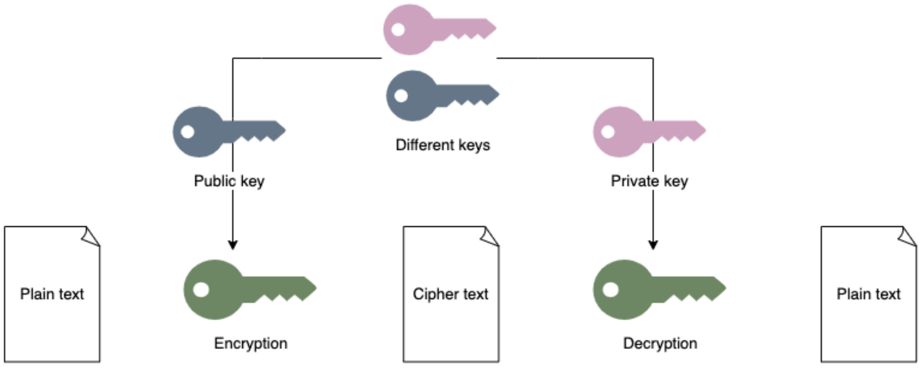


Figure 1.4: Asymmetric cryptography

Hard problems in cryptography

In this thesis, we will only focus on asymmetric cryptography. The concept of public-key cryptography was introduced in 1976 in a paper by Diffie and Hellman [4]. They presented a new protocol now called the Diffie-Hellman (DH) key exchange protocol. DH allows two parties to share a secret key over a channel that is not secure and does this without sharing any secret information in advance. The protocol consists of three main steps. First, the parties agree upon the public values g and p. The g is a generator of the group $\mathbb{Z}_p^*$, where p is a prime. Then each of the parties adds their secret value x to the public ones to create their public key as $A = g^x \mod p$ and $B = g^y \mod p$. The next step is transmitting the public values A and B to each other and then creating a shared secret by raising the received value to the power of their private value.

$$A^y = B^x = g^{xy} \mod p$$

Now both parties will share a secret value without any knowledge of the other person's secret value.

For a third party to be able to find the secret key they must solve the equation g^{xy} mod p given the values g^x and g^y, and this is known as the Discrete Logarithm Problem (DLP). This problem turns out to be quite difficult to solve when p is large. So Diffie-Hellman is dependent on that for the group that is chosen ($\mathbb{Z}_p^*$), there exists no efficient algorithm to solve DLP.

One important thing to mention with the first version of the DH-protocol is that it was vulnerable to a man-in-the-middle-attack. As it does not provide any authentication an eavesdropper could just stand in the middle of the conversation and send messages back and forth without the communicating parties being able to notice. The eavesdropper will pretend to be Bob with Alice, and pretend to be Alice with Bob. The attacker does this by creating its own x and y which they share with the other parties (Alice and Bob) to make the shared secret key. Now the eavesdropper can read all their messages, so this is really not secure. If we add some sort of signature to the scheme, it should not be vulnerable to this attack as we now also have authentication.

One of the earliest public-key cryptosystems is called RSA. The cryptosystem was developed by R.L Rivest, A. Shamir, and L. Adleman and presented in a paper in 1978 [27], it is still used a lot today. The motivation behind the paper was the introduction of email and there was a need for an electronic form of mail with similar qualities as the paper mail. These qualities were privacy and signatures. Therefore in RSA, we introduce encryption and decryption for privacy and signing and verification as digital signatures. We define these methods with RSA as follows

Definition 2 *Let p and q be two different large primes, and $N = pq$ be an RSA modulus. Then we let e be a positive integer such that $gcd(e, \phi(N)) = 1$, and let d be a positive integer satisfying $e \cdot d \equiv 1 \mod \phi(N)$. The public key is N,e, and the private key is p, q, d. Then encryption is done as $c \equiv m^e \mod N$ and decryption as $m \equiv c^d \mod N$.*

For signatures, we can make a signature as $s \equiv H(m)^d \mod N$, where H is a cryptographic hash function. To verify the signature one check if $s^e \equiv H(m) \mod N$.

As we can see from the definition both the encryption/decryption and signing/verification are dependent on public and private keys. In RSA the public key is e,N, and the private key is d. The public value N is a composite number that is constructed by the primes p and q by multiplying these together. Both p and q are secret. The d and e needs to follow the property $d \cdot e \equiv 1 \mod \phi(N)$, where $\phi(N) = (p-1)(q-1)$. So if we are able to obtain p and q, we will also be able to construct the private key d as d will be the only unknown part of the equation. If we can obtain d, we say that one has broken the system.

To acquire p and q one must factorize the RSA modulus N, but this is rather difficult when p and q are large primes. This is a well-known problem called the Integer Factorization Problem (IFP). Today there exists no efficient algorithm that can solve this problem for large numbers, but there do exist algorithms such as Pollard's rho method for factorizing [13, pp. 344-345] and quadratic sieve [13, pp. 345-346] that are able to factorize numbers up to some bit length. However, the primes RSA uses are too big for any of these to be able to solve on today's computers, and we can say that RSA is dependent on IFP being a hard problem for it to be considered secure.

However, as described in Section 1.3 it has been shown that it is possible to solve both IFP and DLP with quantum computers. Today RSA is one of the schemes standardized by NIST to use for digital signatures. As this depends on IFP being a hard problem to solve we are going to look closer at the consequences this quantum algorithm has for RSA and other standards in use today in the next section.

1.3 Post-Quantum Cryptography

As we now have an understanding of what cryptography is about, we are going to look at post-quantum cryptography. Today most of the cryptographic schemes used in public-key cryptography depend on the hardness of problems such as prime factorization and discrete logarithms. These are problems that we believe a normal computer cannot solve in a reasonable time with the standards today. However, in 1994 a mathematician named Peter Shor described an algorithm for finding the prime factorization of an integer [30]. The algorithm is called "Shor's algorithm" and it is a quantum computer algorithm. This means that this algorithm only works for quantum computers, which we will look closer at in Section 1.3.1. However, a consequence that follows from this is that the public-key cryptography schemes in use today will be vulnerable and broken by Shor's algorithm if we are able to produce such quantum computers on a big scale.

Also, today's symmetric cryptography standards will be threatened by big-scale quantum computers. In 1996 Grover described a search algorithm for quantum computers, which will be able to break symmetric schemes if the key-space is too small [8]. Also, hash-functions will be vulnerable to Grover's algorithm. However, the solution to this threat is actually much simpler than for public-key cryptography, as we just need to double the key size to restore the security level.

We have now seen that the cryptography schemes in use today are not safe against large-scale quantum machines, hence they are not post-quantum secure. Post-quantum cryptography is schemes that have no known methods of solving its hard problem in polynomial time on a large-scale quantum computer.

Today the development of quantum computers has begun and big companies such as IBM and Google are actively working on it. They spend a lot of resources on developing them, but why do they do this when we have just looked at how dangerous this can be for today's cryptographic standards? Quantum computers can be used for so many things that are good for our society as well, hence there is a lot going on in this field at the moment. As of now, we do not have any big-scale quantum computers, and there is still a long way to go until they actually pose a threat to the cryptographic standards in use today. However, the standards today should be changed as soon as possible due to many reasons and NIST has come a long way in its standardization process for post-quantum cryptography. We will look closer at this process in Section 1.4.

One of the reasons we have to start a process of finding post-quantum secure standards for cryptographic schemes is because of time. It takes a lot of time to change from one standard to another. Protocols need to be changed, and old protocols need to be replaced. This has shown itself to take a lot of time in the past, hence it is likely to be a slow process now as well. Another reason would be that the process of actually finding good replacements that are post-quantum safe also takes a lot of time. The time for both designing algorithms and analyzing them using cryptanalysis should be long. It is important that these are carefully tested if they are to become new standards. The last reason is the long-term threat of adversaries. Even though it is likely that it will take decades until we have large-scale quantum computers an adversary could save the encrypted sensitive data now. They could then decrypt it later if they get access to a large-scale quantum computer. Hence sensitive information that should stay secret for maybe 20-30 years from now should ideally be encrypted with post-quantum safe encryption now.

We have seen why we need to start with the standardizations today, but what are the options for replacing DLP and IFP? There are several different approaches we could take. These could be

- Multivariate problems: the difficulty of solving systems of multivariate polynomial equations over finite fields.

- Isogeny-based problems: the difficulty of finding an isogeny between two elliptic curves.

- Code-based problems: the difficulty of decoding large general linear codes.

- Hash-based problems: the difficulty of finding collisions or pre-images in hash functions.

- Lattice-based problems: the difficulty of finding the shortest vector in a lattice.

The lattice-based problems are what we are going to focus on in this thesis, and here we are going to study the shortest vector problem (SVP), which we are going to look closer at in Chapter 2. Since there are different approaches we need to figure out which direction is the best one to choose. As mentioned NIST is running a process with new standardizations and the government, industries, and academia are involved in the work of it all.

1.3.1 Quantum Computers

A quantum computer is a computer that uses elements from quantum physics. As opposed to normal computers which use binary bits to encode information, the quantum computer uses what we call a quantum bit (qubit). What is special about qubits contra binary bits, is that binary bits are either 1 or 0 or in other words, can only represent one value at a time depending on how many bits are available. Qubits can be everything those bits can represent at the same time, meaning that they can be both 1 and 0 simultaneously. Since they can be multiple values at the same time, this also gives room for massive parallelization.

It is known that it is possible to solve IFP for a quantum computer due to its parallelization. However, this is easier said than done, as the qubits themselves are not easy to work with. In physical implementations, qubits are sensitive and can easily fail due to for instance noise.

Today's most advanced quantum computers are developed by IBM. Their quantum computers have 433 qubits to use [12], but how much is this? To give a point of view, Shor's algorithm was implemented on a quantum computer around 20 years ago by some researchers at IBM [11]. They implemented it specifically to factorize 15. The quantum machine needed 7 qubits to be able to solve this. However, this implementation will only be able to solve the factorization of that number and not any other. Having a scalable implementation turns out to be very hard, and something that will be needing a lot of work to be able to operate correctly. Shor's algorithm for larger numbers is still not solved, and it seems like it will be difficult to achieve.

We still have a long way to go and probably will need quantum computers with over a million qubits until we are able to factorize big enough numbers to threaten RSA parameters used today. In 2021 Gidney and Ekerå presented a theoretical quantum algorithm to solve 2048-bit RSA using 20 million noisy qubits [7]. It is also worth to note us that the number of qubits for solving these problems can change over time as we learn more about quantum computing.

In recent years and now quantum computer development has been an emerging field so there is a lot of money used to evolve them further. Nobody knows how fast we will be able to get these machines on a big scale yet.

1.4 Standardization of Quantum-Safe Algorithms

NIST stands for National Institute of Standards and Technology, and they work with, among other things, cyber security standards for the U.S. [19]. They standardize cryptographic protocols and algorithms that U.S. companies have to follow. Even though it is a governmental organization for the U.S., it is also useful to the rest of the world, as they spend a lot of time on research and figuring out the most secure way to go. NIST has for instance made standards such as DES and AES, and currently, they are at the end of a process of making standards for post-quantum cryptography as well. As large-scale quantum computers can be a potential threat against the standard cryptographic methods in use today, NIST has decided that they need to make new standards that are post-quantum safe. These standards are for encryption/decryption and digital signatures, and they are planning on releasing the standards in 2024. NIST wants to be precautionary for this type of threat, as they do not expect large-scale quantum machines to be available for several years/decades, but it is better to be prepared in case it does happen. As mentioned an argument as to why we need post-quantum cryptography now is that some data that is encrypted today needs to be secret for at least 20-30 years. If big-scale quantum machines should be just 20 years away, we need to change our encryption algorithms today. NIST has decided to spend a lot of time selecting the new standards, and they chose to use a competition for finding the new standards [21].

The competition started in 2016 with round one where anybody could submit a suggestion with an algorithm they had made. One would submit the code, a description of the algorithm, and who had been a part of the project. On the 30th of January 2019, 26 candidates were chosen to join the next round of the competition, where 17 of them were candidates for encryption/key-encapsulation mechanism algorithms, and 9 of them for digital signatures. For the second round, the participants were given a few months to update their implementation and specifications if they wanted, and then NIST would begin the evaluation of the algorithms. The evaluation part was expected to last for a year to a year and a half.

In 2020 on the 22nd of July, NIST announced the candidates for the third round [22]. There were 7 finalists chosen and 8 alternates. For the public-key encryption and key-establishment algorithms, the following algorithms were chosen as finalists: Classic McEliece, CRYSTALS-KYBER, NTRU, and SABER. The alternate candidates were BIKE, FrodoKEM, HQC, NTRU Prime, and SIKE. There were also chosen finalists for the digital signature algorithms, and these were: CRYSTALS-DILITHIUM, FALCON, and Rainbow. The alternate candidates chosen for the signatures were GeMSS, Picnic, and SPHINCS+. Again the candidates had some time to update comments and make some tweaks to the algorithms before NIST decided on who should advance to the next round.

On the 5th of July 2022, NIST presented the selected algorithms for the public-key encryption and key-establishment algorithms and digital signature algorithms [20]. For Encryption/key-establishment NIST chose to go for CRYSTALS-KYBER, which is a lattice-based algorithm. NIST also decided on finding some other options that are not lattice-based, so the candidates for round four are some extra algorithms. The reasoning for this is in case an effective quantum algorithm to solve the lattice-based problem is found. We would then need some other options that do not depend on the same problem to use instead. The algorithms that are in round 4 are BIKE, Classic McEliece, HQC, and SIKE [23]. For Digital signatures there were selected three algorithms, which were CRYSTALS-DILITHIUM, FALCON, and SPHINCS+. They have also made a proposal for other digital signature algorithms, and these have a submission deadline of June 1st, 2023 [24].

1.5 Task Description

In this master thesis we are going to look closer at the shortest vector problem for lattices, and at which methods used today work best for solving it. In general, it is the sieving method that is considered best for bigger lattices, and a Python library called G6K stands out as the current best tool for doing sieving.

We begin by introducing the concept of lattices, and the relevant basic properties a lattice can have. The lattice theory also consists of important theorems, notations, and definitions that will be used to describe problems such as the shortest vector problem. As the shortest vector problem is the main problem we will be looking at in this thesis we are going to explore the problem and approaches people take to solve it today.

Lattice reduction is introduced as a technique for trying to solve the shortest vector problem. Both the LLL algorithm and BKZ algorithm will be demonstrated as these are important lattice reduction algorithms. Sieving and enumeration are different methods of lattice reduction, and we will mainly focus on the sieving method. We are going to look at how sieving works and what the main idea behind sieving algorithms is.

After sieving is introduced we are going to look at G6K. For the G6K library, we will be going through how it is built up before we explain briefly which sieving algorithms they have implemented, and some of the lattice types they offer. Then some of the strategies that are presented in the G6K paper are gone through. We will also see some examples of how G6K is used today in the SVP challenge.

After presenting enough background knowledge we will do some experiments using G6K. The experiments are on how to use sieving in practice, and we will utilize the G6K library in the experiments. The experiments will try to improve a sieving strategy proposed in G6K and look at how sieving algorithms behave based on lattice type and dimension.

Chapter 2

Background

2.1 Notation

Before we begin with the actual theory, we will go through the notation we use in this thesis. For vectors, we denote them in bold text like this: $\mathbf{v}$. The vectors can also have coefficients in front of them, which we denoted with lowercase letters, an example is $a\mathbf{v}$. Matrices we denote with capitalized letters, so a matrix can look like this: M. Matrix M consists of n vectors that have length m, where the vectors are the rows of M. We can further also denote a matrix with its size $M_{n\times m}$.

A lattice will also be denoted in a capital letter, but this will always be a capital L, and we denote its generator vectors as $(\mathbf{b}_1,\dots,\mathbf{b}_n)$. Also for these lattice vectors, we denote their coefficients in lowercase letters.

For the volume of a lattice and a matrix, we denote it using the abbreviation *vol*, and we write it as $\mathrm{vol}(M)$. For the determinant of a matrix or a lattice we also use the abbreviation *det*, and denote it as $\det(M)$.

2.2 Lattice Theory

A lattice is a discrete additive subgroup of $\mathbb{R}^m$. It is generated by a set of n linearly independent vectors that are in $\mathbb{R}^m$. As we have n independent vectors in $\mathbb{R}^m$, we need $n \leq m$. We define a lattice as follows

Definition 3 *A lattice L is given as $L = \{a_1\mathbf{b}_1 + a_2\mathbf{b}_2 + \ldots + a_n\mathbf{b}_n | a_i \in \mathbb{Z}\}$,where $\mathbf{b}_1,\dots,\mathbf{b}_n$ are n linearly independent vectors in $\mathbb{R}^m$. A set of vectors that generate the lattice is called a basis.*

A basis of a lattice is not unique, hence we can have different sets of linearly independent vectors that generate the same lattice.

We can map between different bases by using a change of basis matrix, which we will denote as U. This matrix needs to have two properties for it to be a valid change of basis. The first property is that U only contains integer values, and the second one is that the determinant of U is ± 1. We can denote the mapping as $X = UY$, where X and Y are different bases of the same lattice. In this notation, we assume that the basis vectors are the rows of the matrix. From here we will also assume that $m = n$ in the rest of the thesis for simplicity purposes.

So now that we know what a lattice is, we are going to look closer at some properties of it. We begin with the fundamental domain, which we define as follows [10, p. 390]

Definition 4 *Let L be a lattice of dimension n and let $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$ be a basis for L. The fundamental domain for L corresponding to this basis, is the set $F(\mathbf{b}_1, \ldots, \mathbf{b}_n) = \{t_1\mathbf{b}_1 + t_2\mathbf{b}_2 + \ldots + t_n\mathbf{b}_n : 0 \leq t_i < 1\}$*

Every point in $\mathbb{R}^n$ can be uniquely written as $\mathbf{v} + \mathbf{r}$, where $\mathbf{v}$ is in the lattice and $\mathbf{r}$ in the fundamental domain. This means that $\{\mathbf{v} + F(\mathbf{b}_1, ..., \mathbf{b}_n) | \mathbf{v} \in L\} = \mathbb{R}^n$.

A usage of the fundamental domain of a lattice is its volume, which we define as follows

Definition 5 *Let L be a lattice of dimension n and let F be a fundamental domain for L. The volume of L is then the n-dimensional volume of F.*

We can calculate the volume of L by computing the determinant of the matrix given by the basis of L. As stated earlier, a lattice can have different bases, but we will see that the volume will still be the same as it is not dependent on the choice of basis. We can see this by having a basis B and another basis A for the same lattice, which we know can be written as $B = UA$. The volume then becomes as follows

$$|\det(B)| = |\det(U)\det(A)| = 1|\det(A)| \Rightarrow |\det(B)| = |\det(A)|$$

and we can see that any choice of bases A and B have the same volume.

We are now going to look at some applications of the fundamental domain and volume of a lattice. Hadamard's inequality is a way of measuring how close to being orthogonal your basis is. The inequality is written as [10, p. 393]

$$\text{vol}(L) \leq ||\mathbf{b}_1|| \cdot ||\mathbf{b}_2|| \ldots ||\mathbf{b}_n||$$

Here we see that on the right side of the inequality we want to look at the length of the lattice vectors of the basis. To calculate the length we use the Euclidean norm, which is defined as follows

Definition 6 *Let $\mathbf{b} = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Then the Euclidean norm is $||\mathbf{b}|| = \sqrt{a_1^2 + \cdots + a_n^2}$*

We use the $||$ symbol to show that we are looking at the length of the vector.

Hadamard's ratio is quite similar to Hadamard's inequality, but here we divide the volume of the lattice with the product of the length of the vectors in the basis. Now we will get a value between 0 and 1. The closer the value is to 1, the closer the basis is to be orthogonal. We see the same for the inequality, the closer it is to equality the closer to being orthogonal the basis is. We will go into detail on why we want to measure the orthogonality later.

Another parameter of lattices is the Euclidean ball. We denote the ball centered at the origin as follows

$$B_r = \{\mathbf{b} \in \mathbb{R}^n : ||\mathbf{b}|| \leq r\} \tag{2.1}$$

We generalize this definition to allow the Euclidean ball to be centered at any given point.

Definition 7 *Given any* $\mathbf{v} \in \mathbb{R}^n$ *and a radius* $r > 0$*, then the ball of radius* r *centered at* $\mathbf{v}$ *is the following set* $B_r(\mathbf{v}) = \{\mathbf{b} \in \mathbb{R}^n : ||\mathbf{b} - \mathbf{v}|| \leq r\}$

We are now going to look at two important lattice theorems/definitions. These are called the Minkowski Theorem and the Gaussian Heuristic. They can both be used for themselves and also to prove Hermite's theorem. Hermite's theorem tells us that a lattice of dimension n always will contain a nonzero vector whose length is upper bounded. We will have a closer look at Hermite's theorem later.

Since we are working with short vectors we have a notation to symbolize the length of a shortest vector. We denote this by using the Greek letter λ, so for instance $\lambda_1(L)$ is the shortest length a vector can have. If we instead of 1 use another number i, this means there are lattice vectors of length smaller than λ_i that span a subspace of dimension at least i. For instance $\lambda_2(L)$ will be the second shortest vector that is not a multiple of the shortest vector in L. We can define $\lambda_i(L)$ as

$$\lambda_i(L) = inf_{r \in \mathbb{R}} dim(Span(L \cap B_r(0))) \geq i \tag{2.2}$$

Now we are going to look at Minkowski's Theorem, but here we first need to know some definitions for bounded, symmetric, convex, and closed sets. These are defined as follows [10, p. 398]

Definition 8 *Let* $S \subseteq \mathbb{R}^n$

- *S is bounded if the vectors that are in S have bounded lengths.*
- *S is symmetric if for every point* $\mathbf{x}$ *in S, the negation* $-\mathbf{x}$ *is also in S.*
- *S is convex if whenever we have two points* $\mathbf{x}$ *and* $\mathbf{y}$ *that are in S, then the line segment that is between these points also lies in S.*
- *S is closed if it fulfills the property: if* $\mathbf{x} \in \mathbb{R}^n$ *is a point such that every ball* $B_r(\mathbf{x})$ *contains a point of S, then* $\mathbf{x}$ *is in S.*

Minkowski's Theorem states that if we have a lattice $L \subset \mathbb{R}^n$ of dimension n and a subset $S \subseteq \mathbb{R}^n$ that is bounded, symmetric, and convex as defined in Definition 8, we can use its volume to check for nonzero lattice vector. If $\text{vol}(S) > 2^n \det(L)$, we know there exists a non-zero lattice vector in S. This is helpful as we are able to guarantee that we have some valid lattice vector in our chosen set. If the subset S is also closed, then the same statement holds with equality.

Theorem 1 (Minkowski's Theorem) *Let L be a lattice* $L \subset \mathbb{R}^n$*, where n is the dimension of L, and let* $S \subset \mathbb{R}^n$ *be a bounded symmetric convex set such that its volume satisfies*

$$vol(S) > 2^n \det(L) \tag{2.3}$$

There will then exist a nonzero lattice vector in S.

So now we have seen that Minkowski's theorem guarantees us that there will exist a nonzero vector upper bounded by some length. Another theorem that tells us the bound of the Euclidean norm of the shortest vector in a lattice is Hermite's Theorem, which can be stated as follows

Theorem 2 (Hermite's theorem) *Every lattice L of dimension n contains a nonzero lattice vector $\mathbf{b}$ such that $||\mathbf{b}|| \leq \sqrt{n}(\det(L))^{1/n}$*

From Theorem 2 we can say that $\lambda_1(L) \leq \sqrt{n}(\det(L))^{1/n}$. By applying the boud n times we get

$$\det(L) \leq \prod_{i=1}^{n} \lambda_i(L) \leq n^{n/2} \det(L)$$

From this bound we can see that the length of the shortest vector is dependent on the determinant of the lattice, meaning that if it has a low determinant there exist shorter vectors to find.

However, there usually exist vectors that are shorter than this bound, and here the Gaussian Heuristic comes in. The expected shortest length of a non-zero lattice vector in L is given by the following formula

$$\sigma(L) = \sqrt{\frac{n}{2\pi e}}(\det(L))^{1/n} \tag{2.4}$$

The Gaussian Heuristic says that we expect the shortest vector to be approximately this length. We define the Gaussian Heuristic as

Definition 9 (Gaussian Heuristic) *The shortest nonzero vector in L will satisfy the approximation $\lambda_1(L) \approx \sigma(L)$.*

Later in Subsection 2.3 we will look at lattice problems where a goal is to look for short vectors in a lattice. Here these theorems are useful as the Gaussian Heuristic gives us an estimation of how long the lattice vector is.

At last, we define sublattices

Definition 10 *Let L and L' be lattices such that $L' \subseteq L$. We say that L' is a sublattice of L.*

2.2.1 Gram-Schmidt Orthogonalization

Gram-Schmidt orthogonalization is a well-known concept from linear algebra, and we will see that it is important for some of the lattice reduction algorithms later. We define a Gram-Schmidt basis as

Definition 11 *Let $\mathbf{b}_1, \ldots, \mathbf{b}_n$ be a basis of independent vectors in $\mathbb{R}^n$. A Gram-Schmidt basis consists of orthogonal vectors, such that the space spanned by $< \mathbf{b}_1, \ldots, \mathbf{b}_i >$ and $< \mathbf{b}_1^*, \ldots, \mathbf{b}_i^* >$ are the same for all i, where $i = 1, \ldots, n$.*

One can calculate a Gram-Schmidt basis from a lattice basis by using Algorithm 1. We let $\mathbf{b}_1, \ldots, \mathbf{b}_n$ be a basis for a vector space $V \subset \mathbb{R}^m$. Then we use the algorithm to create an orthogonal basis $\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*$ for V.

So what happens when we put a lattice basis through the Gram-Schmidt orthogonalization algorithm is that we get a basis back that is orthogonal. The basis we get back is usually not a basis for the lattice, but it does however have the same determinant as the original lattice basis. We will use this fact later when we in Section 3.1.1 describe the LLL algorithm.

Algorithm 1 Gram-Schmidt Algorithm

```
Require: Basis $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$
Ensure: Gram basis $\{\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*\}$
  $\mathbf{b}_1^* \leftarrow \mathbf{b}_1$
  for $i = 2, 3, \ldots, n$ do
      for $j = 1, 2, \ldots, j < i$ do
          Compute $\mu_{i,j} = \frac{<\mathbf{b}_i, \mathbf{b}_j^*>}{||\mathbf{b}_j^*||^2}$
      end for
      $\mathbf{b}_i^* \leftarrow \mathbf{b}_i - \sum_{j=1}^{i-1}(\mu_{i,j}\mathbf{b}_j^*)$
  end for
```

Projection of a lattice vector

As seen above we can denote the projection of $\mathbf{b}_i$ onto $Span(\mathbf{b}_1, \ldots, \mathbf{b}_{i-1})^{\perp}$ as $\mathbf{b}_i^*$. Now we are going to look closer at what this means. First we introduce another notation for projection which is $proj_S(\mathbf{x})$, where S is the space where we want to project vector $\mathbf{x}$ onto. We let $S = span(\mathbf{b}_1, \ldots, \mathbf{b}_{i-1})$, which also is the same as $span(\mathbf{b}_1^*, \ldots, \mathbf{b}_{i-1}^*)$. If we want to do a projection of $\mathbf{x}$ onto S we do as follows

$$proj_S(\mathbf{x}) = \frac{<\mathbf{x}, \mathbf{b}_1^*>}{||\mathbf{b}_1^*||^2}\mathbf{b}_1^* + \cdots + \frac{<\mathbf{x}, \mathbf{b}_{i-1}^*>}{||\mathbf{b}_{i-1}^*||^2}\mathbf{b}_{i-1}^*$$

However, what we use more in this thesis is projection onto the orthogonal space $S^{\perp}$. This is quite easy when we have projection onto the space S. To project $\mathbf{x}$ onto $S^{\perp}$ we do the following

$$proj_{S^{\perp}}(\mathbf{x}) = \mathbf{x} - proj_S(\mathbf{x})$$

This is exactly what is happening when we do a Gram-Schmidt orthogonalization, and the process can be seen inside the for-loop of the pseudo-code above. A projection can be visualized as seen in Figure 2.1. The figure shows both projection onto a space and its orthogonal space.

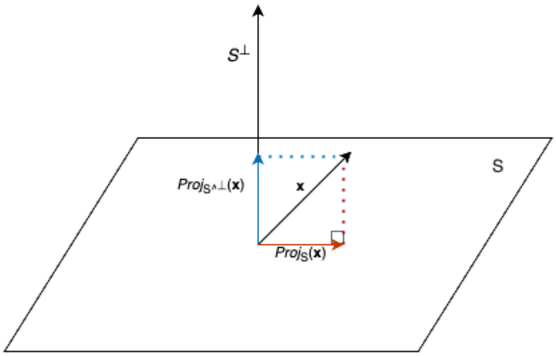


Figure 2.1: Projection of $\mathbf{x}$ *onto S and* $S^{\perp}$

The context of a lattice

Later in the thesis, we will refer to what we call a *context* of a lattice. The context can be denoted as $L_{[l:r]}$, where $[l:r]$ is the context itself.

In the contexts, l and r represent indices from which lattice vectors we are looking at. We can look at our basis as column vectors where the first column (leftmost) will be at index 1, while the last column (rightmost) will be index n, where n is the dimension of the lattice. We also note that l and r in the context have the property $1 \leq l < r \leq n$.

The lattice $L_{[l:r]}$ is generated by the basis $B_{[l:r]}$. We let the basis $B = \{\mathbf{b}_1, \dots, \mathbf{b}_n\}$ and we say that $S = span(\mathbf{b}_1, \dots, \mathbf{b}_{l-1})$. Then we can write out $B_{[l:r]}$ as

$$B_{[l:r]} = \{proj_{S^\perp}(\mathbf{b}_l), \dots, proj_{S^\perp}(\mathbf{b}_{r-1})\}$$

So what we see from this notation is that all of the vectors in the context are projected onto the space orthogonal to the space spanned by the lattice vectors that are to the left of the context.

2.3 Shortest Vector Problem

When working with lattices one of the main problems is finding a shortest nonzero lattice vector. This problem is known as the shortest vector problem (SVP), and it turns out to be a rather hard problem. When trying to solve SVP in a lattice we sometimes know how short the Euclidean norm of a shortest vector is, but this is not always the case. However, the issue is not finding this length but finding a lattice vector that has the desired length. We say we want to find a shortest vector and not the shortest vector as there usually exist multiple vectors that have the same norm. An example is if we work with a two-dimensional lattice that has the basis $\{(0,1),(1,0)\}$ (which is in $\mathbb{Z}^2$), then both of the basis vectors will have the same norm and be a shortest vector. We also know that if lattice vector $\mathbf{b}$ is a shortest vector, then also $-\mathbf{b}$ will be a shortest vector since these will always have the same norm.

There also exists an approximation version of SVP, and this is called apprSVP. Now we do not need to find a shortest vector, but we want to find a vector that is at most $f(n)\lambda_1(L)$ in length. Here $f(n)$ is a function of n (the dimension), and the hardness of apprSVP depends on which function you use as $f(n)$. ApprSVP is a problem that we are able to solve within some factor using for instance LLL reduction which we will look at in Chapter 3. If the function chosen is small, the problem gets very difficult as well.

2.3.1 Other Lattice Problems

Another similar problem to SVP is the closest vector problem (CVP). The goal is now to find the closest lattice vector $\mathbf{b}$ to a given vector $\mathbf{a}$, where $\mathbf{a}$ is not in the lattice. Vector $\mathbf{b}$ needs to be a lattice vector that makes the smallest Euclidean norm

$$min_{\mathbf{b} \in L} ||\mathbf{a} - \mathbf{b}||$$

where $\mathbf{b}$ runs over all lattice vectors.

CVP is also considered to be a very difficult problem to solve, and in practice, it is even a little harder than SVP [10]. In similarity with SVP, also CVP has an approximate version of the problem. We will not look any closer at this, but it is the same concept as with apprSVP.

The shortest basis problem (SBP) is another variant of the problems above, and there also exist many versions of this problem. The problem is to obtain a lattice basis that is short according to some measures. An example of a measure could be the biggest norm of the basis vectors being minimized. The reason it exists variations of this problem is the fact that you can decide on the measurement factor.

As already stated SVP is a very hard problem, which makes it interesting to use in cryptography. SVP will be the focus problem of this thesis. In the next chapter, we are going to look at reduction algorithms, where the goal is to find short vectors.

Chapter 3

Lattice Reduction

A method for finding short vectors in a lattice is using lattice reduction algorithms. With lattice reduction, we try to reduce the length of the basis vectors we begin with, to something shorter. The goal of lattice reduction is to obtain a better basis (see below) for the lattice. The most well-known lattice reduction algorithm is called the LLL algorithm, which will be further described in Section 3.1.1. It is also possible to split lattice reduction into two different categories when thinking about types of methods for the reduction. These are called *enumeration and sieving*. This thesis will be focusing on the sieving method.

What is a good basis?

As mentioned in Section 2.2 a basis of a lattice is not unique, so there are many different lattice bases that generate the same lattice. One of the goals of lattice reduction is to obtain a better basis, but what does a "good basis" mean? There is no definite answer, but we usually mean that the basis contains lattice vectors that are reasonably short and nearly orthogonal. A way to measure how good a basis is can be done by using Hadamard's ratio (see Section 2.2). As stated the Hadamard's ratio gives a value between 0 and 1 and the closer the value is to 1, the better the basis is.

3.1 Lattice Reduction Algorithms

There are two different methods used for lattice reduction: enumeration and sieving, and we are now going to give a brief description of the difference between them. In enumeration, we do an exhaustive search to find combinations of the basis vectors that have a norm that is shorter than some predefined bound. We can look at the search as a depth-first search tree.

The idea of sieving however is based on having a database with lattice vectors, then looking at the length of the difference or sum between the vector pairs $(||\mathbf{b}_1 \pm \mathbf{b}_2||)$. Like this, we obtain shorter vectors by taking this difference or sum. So the main idea is to start with many vectors, and then gradually obtain shorter and shorter lattice vectors, where we continuously replace the longer vectors with the new shorter ones. Like this, the shorter vectors always become a part of the next iteration of the sieve. This process is continued until we reach a desired lattice vector with a length within some given bound.

In summary, there are two main methods for lattice reduction, but which one works the best? In theory, it is the sieving method that should work the best, however, this is not always the case in practice. If we are working on big lattices it seems that sieving is the better method, while for smaller lattices enumeration is the best. However, it has been debated where the crossing point of dimension for where sieving starts to work better. In 2014 Daniele Micciancio and Michael Walter estimated lattices of up to dimension 120 to be where enumeration works the best [17]. However, as the years have passed this estimation has become lower and lower. Léo Ducas showed in [5] just 4 years later that this estimation could be set down to 90. In 2019, by using G6K with optimization it is stated in [2] that the crossing point is at dimension 70. Today the records for solving the SVP challenges (see Section 3.4) are done using a sieving method using G6K [35].

Continuing on we are going to look at some of the most well-known lattice reduction algorithms: the LLL algorithm and the BKZ algorithm.

3.1.1 The LLL Algorithm

The LLL algorithm was published in 1982 by Lenstra, Lenstra, and Lovász [15], and the algorithm returns a lattice basis that is said to be LLL-reduced. The LLL-reduced basis is considered a good basis, here meaning that we get an upper bound for both the norm of the vectors and their perpendicularity. These bounds can be seen in the Equations 3.1-3.3. For a basis to be LLL-reduced, it must satisfy two conditions called size condition and Lovász condition which are defined as follows:

Definition 12 *Let $B = \{\mathbf{b}_1, \dots, \mathbf{b}_n\}$ be a basis for the lattice L, and let the assosiated Gram-Schmidt basis be $B^* = \{\mathbf{b}_1^*, \dots, \mathbf{b}_n^*\}$. The basis B is said to be LLL-reduced if it fulfills the following conditions:*

$$\textit{Size condition: } |\mu_{i,j}| = \frac{| < \mathbf{b}_i, \mathbf{b}_j^* > |}{||\mathbf{b}_j^*||^2} \leq \frac{1}{2}, \textit{for all } 1 \leq j < i \leq n$$

$$\textit{Lovász condition: } ||\mathbf{b}_i^*||^2 \geq (\frac{3}{4} - \mu_{i,i-1}^2)||\mathbf{b}_{i-1}^*||^2\textit{, for all } 1 < i \leq n$$

Now we know what an LLL-reduced basis is, but how can we find them? It turns out that we can find LLL-reduced bases for any lattice, meaning that there always exists an LLL-reduced basis for a given lattice. The way of obtaining an LLL-reduced basis is by using the LLL algorithm, which is given in a simplified version in Algorithm 2.

The algorithm consists of three main events: size reduction, checking the Lovász condition, and swapping the lattice vectors. First, we loop over the basis and perform a size reduction on each of the lattice vectors. This is quite an easy process to do, and it is done in stages rather than all at once. The LLL algorithm also has to update its Gram-Schmidt basis (see Section 2.2.1), as this is an important part of the algorithm. It is also important to note that the order of the lattice vectors matters, and here the Lovász condition comes in. If the Lovász condition is satisfied we continue on to the next lattice vector and repeat the while loop, if it is not met we swap the order of the lattice vectors and go back one step in the basis. When the basis is both size-reduced and all

Algorithm 2 LLL Algorithm

```
Require: basis $B = \{\mathbf{b}_1, \dots, \mathbf{b}_n\}$ for lattice $L$
Ensure: LLL-reduced basis for $L$
  $k = 2$
  $B^* = computeGramSchmidt(B)$
  while $k \leq n$ do
      for $j = k-1, \dots, 1$ do
          $\mathbf{b}_k = \mathbf{b}_k - \lceil \mu_{k,j} \rfloor \mathbf{b}_j$
          $updateGramSchmidt(\mathbf{b}_1, \dots, \mathbf{b}_n)$
      end for
      if LovászCondititon($k$) then
          $k = k+1$
      else
          $swap(\mathbf{b}_k, \mathbf{b}_{k-1})$
          $updateGramSchmidt(\mathbf{b}_1, \dots, \mathbf{b}_n)$
          $k = max(k-1, 2)$
      end if
  end while
```

of the lattice vectors satisfy the Lovász condition we have an LLL-reduced basis, and the algorithm terminates. It is also important to mention that an LLL-reduced basis will have the lattice vectors in an order where they are sorted on lengths, so the first one is always the shortest, and so on.

The LLL algorithm will terminate in polynomial time [10, p. 443], and returns a basis that satisfies the following bounds [10, p. 441]

$$\prod_{i=1}^{n} ||\mathbf{b}_i|| \leq 2^{n(n-1)/4} \det(L) \tag{3.1}$$

$$||\mathbf{b}_j|| \leq 2^{(i-1)/2} ||\mathbf{b}_i^*|| \text{ for all } 1 \leq j \leq i \leq n \tag{3.2}$$

As mentioned, the order of the basis vectors is in ascending order, meaning that the first basis vector should be the shortest one. Hence the first lattice vector in the LLL-reduced basis will meet the following bounds

$$||\mathbf{b}_1|| \leq 2^{(n-1)/4} |\det(L)|^{1/n} \text{ and } ||\mathbf{b}_1|| \leq 2^{(n-1)/2} \lambda_1(L) \tag{3.3}$$

The reason that the bounds in 3.3 hold from the bounds 3.1 and 3.2 is:

First we set $j = 1$ in (3.2) and multiply over n times $||\mathbf{b}_1||^n \leq \prod_{i=1}^{n} 2^{(i-1)/2} ||\mathbf{b}_i^*||$. From [10, Prop.7.68] we know that $\prod_{i=1}^{n} ||\mathbf{b}_i^*|| = |\det(L)|$, and multiplying out the $2^{(i-1)/2}$ factors we get

$$||\mathbf{b}_1||^n \leq 2^{n(n-1)/4} \prod_{i=1}^{n} ||\mathbf{b}_i^*|| = 2^{n(n-1)/4} \det(L)$$

We then take the n-th root of the equation and obtain $||\mathbf{b}_1|| \leq 2^{(n-1)/4} |\det(L)|^{1/n}$, which is the first bound in equation 3.3. The second part of bound 3.3 can be found proved in [10].

The bounds from (3.3) express that an LLL-reduced basis will be able to solve apprSVP within a factor of $2^{(n-1)/2}$ in polynomial time.

3.1.2 The BKZ Algorithm

The BKZ algorithm is a block reduction algorithm and it was introduced in the late 1980s by Schnorr [28]. The algorithm is a generalization of the well-known LLL algorithm we saw above. In BKZ we look at one block of the basis at a time, where the size of this block is decided by a parameter β. A block is a projected sublattice, meaning that the current block consists of basis vectors that are projected onto the orthogonal space that is spanned by the basis vectors that are previous to the block. We denote it as $span(\mathbf{b}_1,\ldots,\mathbf{b}_{i-1})^{\perp}$ when the block starts at index i. The block can be compared to what we called context in Section 2.2, and by "previous to the block", we refer to the order of lattice vectors in the basis.

The block moves up as $B_{[1,\beta]}, B_{[2,\beta+1]}$ until we have reached the end of the basis. We call this process a BKZ tour, and at each step of the tour the current block is reduced. It is important to notice that we try to solve SVP in each block, either using enumeration or sieving. The short vector that we find in each (projected) block is then lifted and put into the lattice basis. The vector will be put into the leftmost position in the block. As we continue to solve SVP on the various blocks, the results in these might change as vectors are moved around the block during its tour. The BKZ-tour is therefore repeated until there are no more changes to be done to the basis or some other stop condition is met, and it will then return a BKZ-reduced basis. The basis that the BKZ algorithm returns will be of good quality and will contain short vectors.

We mentioned that BKZ is a generalization of LLL, and BKZ with $\beta = 2$ is similar to LLL. We know that the algorithm will terminate in polynomial time when $\beta = 2$, but is this true for all other β? BKZ will solve apprSVP within a factor of approximately $\beta^{n/\beta}$[10, p. 427]. So the bigger we have β, the shorter lattice vectors we should be able to find. However, increasing β comes at a cost of running time. This means that we have a trade-off between the quality of the basis and running time.

3.2 Lattice Sieving

We begin this section by introducing the concept of sieving before we go into more detail on the different approaches one can take.

In 2001 Ajtai, Kumar, and Sivakumar introduced in a paper the first sieving algorithm with the concept of a sieving approach for lattice reduction [1]. In the paper, they demonstrate how this new technique is able to solve SVP for dimensions that earlier had been considered difficult to do.

The whole idea of lattice sieving is in short based on looking at pairs of lattice points and then taking the sum or difference between these. If the sum of the lengths reveals a shorter norm we have found a shorter vector, which also will belong to the lattice. It belongs to the lattice as it is a linear combination of two other lattice vectors. In many sieving algorithms, we choose the lattice vectors we want to take pairs from in a database. The databases can be created differently, some will sample many vectors in the beginning and then use this as a base, while others might build the database as they go. We will see later in the section below which algorithms implemented in G6K use which approaches.

Another important aspect of sieving is the fact that we want to keep the shorter vectors we obtain along the way. We want this as these can contribute to discovering new and shorter lattice vectors. In some methods, we will keep on updating the same database, and in others, we change out the old database with a new one that contains the shorter lattice vectors we have found. As we continue on with iterations of taking pairs and updating the database we will get shorter and shorter lattice vectors until we obtain a shortest nonzero lattice vector.

Since Ajtai et al. released their paper on the sieving method there have been multiple attempts by different people to utilize this, and also make it more effective. We will look at some different approaches in Section 3.3.1, where we look at the sieving algorithms implemented in G6K.

3.3 General Sieve Kernel

General Sieve Kernel (G6K) is a framework that was presented in an article in 2019 [2]. Following this framework, we also have a Python library with the same name. This library is developed by some of the authors of the article and other contributors. G6K provides a set of lattice reduction algorithms that focus on the sieving method. In their sieving strategies, they usually split the lattice they are working on into blocks of smaller sublattices where they reduce each block. We will have a closer look at this later in the section.

G6K uses FPYLLL, which is a Python interface for FPLLL (a package for lattice reduction). FPLLL provides a wide range of lattice types and lattice reduction algorithms such as LLL. We look at the different lattice types it provides below. G6K follows the structure of FPYLLL so these can be integrated together easily.

Both G6K and FPYLLL are open-source using the GNU General Public License v2.0, which means that it is free software that anybody can download and use with disclosure of source. The functions that G6K provides are also designed to be easy to tweak. Like this people are able to experiment and test out things as they use the library. It is also designed in a scalable way, meaning that it should be easy to parallelize the code and also use modern architectures such as multi-core processors and GPUs for optimizing efficiency.

3.3.1 Sieving Algorithms in G6K

In this section, we are going to look closer at the different sieving algorithms that G6K provides. These are used in the experiments we are going to do in Chapter 4.

Nguyen-Vidick Sieve (`nv`)

Nguyen-Vidick is a heuristic sieving algorithm that was proposed in 2008 to show that sieving algorithms were practical [18]. It works by first sampling many random and possibly long lattice vectors in a list/database, and then it will apply a sieve on this list. The sieving part consists of going through all the pairwise combinations of the lattice vectors by summing them together. If the sum is smaller than the original vectors, this new vector is added to a new list. These new vectors will also be a part of the lattice as any such combination of lattice vectors will create a vector that is in the lattice. Then the sieving process is repeated on the new list, and it will continue doing so. Like this, we will work with smaller and smaller lists after each sieve and continuously discard old lists. During this process, there is a possibility of throwing away short vectors that could be of value for the reduction. Hence, one could look at this as wasteful, and today the algorithm is not used a lot in practice[14].

BeckerGamaJoux Sieve (`bgj1`)

The main idea behind the `bgj1` algorithm was introduced by Becker, Gama, and Joux in 2015 [3]. The idea was to implement an algorithm for finding pairs of neighbors in a database. What we mean by neighbors are lattice vectors that are close to being parallel to each other. When the pairs are close to being parallel there is a bigger chance of getting a result that is what we seek, namely shorter vectors. The algorithm fills buckets according to a filtering rule, where the center of the bucket is a random vector. In G6K `bgj1` only has one level of filtering, meaning it is a simpler version of what it has the potential to be. In the buckets, it performs pairwise tests to find shorter vectors (the sieving part as described in Section 3.2), so each bucket represents a database. The bucket size is based on the cost of both filling the bucket and operations done inside the bucket. The algorithm terminates heuristically when the sieving has finished in a given amount of buckets.

Gauss Sieve (`gauss`)

The Gauss sieve was introduced in 2010 by Daniele Micciancio and Panagiotis Voulgaris [16]. The main idea behind the Gauss sieve is to have a list/database D that obtains shorter and shorter lattice vectors. An important difference to `nv` is that we begin with an empty D, and then fill it up as we go. To obtain shorter vectors the algorithm picks a vector $\mathbf{x}$, and by using the vectors in the database tries to reduce it by taking $||\mathbf{x} \pm \mathbf{y}||(\mathbf{y} \in D)$, before this vector is also added to the database. While trying to reduce $\mathbf{x}$, Gauss also tries to reduce the vectors already in the database. It does this by replacing $\mathbf{y}$ or $\mathbf{x}$ with $\mathbf{x} \pm \mathbf{y}$ if $(||\mathbf{x} \pm \mathbf{y}||) \leq max(||\mathbf{x}||, ||\mathbf{y}||)$. Like this, the database gets updated as much as we can in each round with shorter vectors. This process continues until we have a lattice vector that is of a desired length.

Triple Sieve (`hk3`)

Gottfried Herold and Elena Kirshanova introduced an idea in the paper "Improved Algorithms for the Approximate k-List Problem in Euclidean Norm" in 2017 based on doing sieving on k lattice vectors [9]. The paper explains how to incorporate the k-list problem into sieving. The goal of doing this is to reduce memory, and the technique is not a sieving method in itself and it can be combined with other existing sieving algorithms.

In G6K this idea is implemented and is called `hk3` or triple sieve. The sieving algorithm uses Gauss sieve as a base and uses a 2-sieve (the normal sieving technique) and a 3-sieve. What we mean by 3-sieve is that if the vectors $\mathbf{x}, \mathbf{y}, \mathbf{z}$ are in the database D and $||\mathbf{x} \pm \mathbf{y} \pm \mathbf{z}||$ is short, we replace the longest vector with the found vector in the database D. It is also shown that you are able to find shorter vectors with 3-sieve than a 2-sieve when the same database is used. Due to this we can use a smaller database and reduce memory usage. The implementation in G6K also uses some filtering techniques as well, and here they use the one from their `bgj1` algorithm.

BDGL Sieve (`bdgl, bdgl1, bdgl2, bdgl3`)

The bdgl algorithm is very similar to the `bgj1` algorithm, and the difference is only in how the center of a bucket is chosen. In bdgl $\mathbf{x}$ is chosen from a spherical code, and not a lattice vector. What is important to note here is that the decoding is expensive and will therefore give some overhead.

In G6K we are provided with `bdgl, bdgl1, bdgl2, bdgl3` where the difference is what its `blocks` parameter is set as. The `blocks` parameter can be either $1,2$, or 3. If you use `bdgl` the `blocks` parameter is set to be *none*, where in the code it will give it a value that is valid, which we have found to be 2. So in other words `bdgl` is `bdgl2`. However, `blocks` can also differ based on which dimension you are trying to sieve on, hence even though you are using `bdgl3` the `blocks` parameter could be set to 1 instead of 3 if the dimension allows it to be. This is in G6K decided by dividing the dimension with 28 (rounded down), and if this is less than the `blocks` parameter given, it is set to be this value. Why the dividing number is 28 is not specified. Hence the bdgl-algorithms should behave similarly when the dimension is low, and the difference is more apparent when the dimension increases. As mentioned this algorithm uses `buckets` as well, and the number of buckets is also decided based on the `blocks` and some other parameters.

3.3.2 Lattice Types in G6K

The G6K library uses the FPYLLL library for creating the lattices given as matrices of basis vectors. To create a random matrix/lattice we use the command

```
A = IntegerMatrix.random(dimension, type, bits)
```

Some of the matrices also have other parameters, but these are the basic ones that all of them use. The parameter `dimension` is an integer that decides which dimension the lattice will have. The `type` parameter is which type the lattice should be, and these are described below. At last, the `bits` parameter is an integer telling how big the numbers in the basis vectors should be (at maximum).

In this section, we are going to look closer at some of the types of lattices that FPYLLL provides which we have used in the experiments later in Chapter 4. FPYLLL also provides two other types called lower triangular matrices and simdioph matrices, but we are not going to look at these in this thesis.

q-ary (`qary`)

The q-ary lattice type uses either `bits` or `q`, and a `k` as extra parameters in the command when you create the lattice. The `type` parameter needs to be set as `qary`. In the experiments we have used `bits` instead of `q`, so here it will sample an integer q that is smaller or of the same size as the `bits` we chose. Now a lattice is generated whose determinant is equal to q^k. The lattice basis is of the following form

$$\left[\begin{array}{c|c} qI_{k\times k} & 0 \\ \hline H & I_{k\times(n-k)} \end{array}\right] \tag{3.4}$$

Here I is an identity matrix, which means that we just have zeros except for the diagonal which consists of ones. It has two blocks that are I, where one is multiplied with q as well. Then we have H which is a $k \times (n-k)$ matrix that contains uniformly random integers modulo q (recall that n is the dimension). At last, we have a block that only contains zeros.

NTRULike (`ntrulike`)

This type of matrix comes from the NTRU cryptosystem, and the NTRU matrix is built up to follow a certain pattern. We split the lattice into four blocks that all are of the same size ($N \times N$). First, we have a block that contains the identity matrix I. The rows of the block to the right of I contain all cyclic rotations of the coefficients of a polynomial h. Here h is an element of $\mathbb{Z}_q[x]/(x^N - 1)$ sampled uniformly at random. The value of q is either given as a parameter or is generated based on the `bits` parameter. The lower part of the lattice begins with a block containing a zero matrix, and to the right for this we have another identity matrix block, but this one is also multiplied by an integer q. The following matrix is a representation of the NTRU lattice

$$\left[\begin{array}{c|c} I & rotation(h) \\ \hline 0 & qI \end{array}\right] \tag{3.5}$$

G6K provides a very similar type which is called NTRULike. To generate the NTRULike lattice we write `ntrulike` as the `type` parameter. In our experiments, we also do not use the extra q parameter and only use `bits`.

A comment/warning from the documentation of FPYLLL states that "the NTRULike lattice does not produce genuine NTRU lattice with an unusually short dense sublattice" [33].

Knapsack (`intrel`)

In the knapsack lattice, the first column of the matrix contains different random numbers, while the rest of the lattice basis is an identity matrix (see Example 3.6). When making the lattice we write `intrel` as the `type` parameter to get the knapsack lattice. The size of the random numbers is given by the `bits` parameter.

$$\begin{bmatrix} a & 0 & \dots & 0 \\ b & 1 & \dots & 0 \\ \vdots & 0 & \ddots & 0 \\ z & 0 & 0 & 1 \end{bmatrix} \tag{3.6}$$

Uniform lattices (`uniform`)

This is a lattice where all the entries are integers sampled uniformly at random. The length of the integers is less than or equal to the `bits` that is provided by the user. To get the uniform lattice we write `uniform` as the `type` parameter. An example is as follows

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \tag{3.7}$$

3.3.3 Lattice Reduction Strategies in G6K

The G6K framework introduces some advanced strategies and new operations for lattice reduction. There are five primary operations that the strategies use, and we will first have a look at these. Before we go into detail we first need to state how we visualize our lattice basis. The basis consists of column vectors where $\mathbf{b}_1$ is the left-most vector, and $\mathbf{b}_n$ is the vector the right-most.

The different operations work on the context of the lattice (see Section 2.2), which we repeat is a part of the basis that is projected. We denote it as $L_{[l:r]}$, where $0 \leq l < r \leq n$, which will be the lattice generated from the projected basis $B_{[l:r]}$.

Basic Lattice Operations

All of the following lattice operations take place within a context of a lattice, and the operations also are able to somewhat keep the shortness of the vectors.

Before we look at the operations it is important to note that we now are looking at the coefficients of the lattice vectors, and not the lattice vectors themselves. This is due to how the database in G6K works. The coefficients of a lattice vector are denoted as $(v_1, \ldots, v_n)$, and is what is actually stored in the database. In the operations below we will use $d = r - l$ in the notation of the coefficients.

- (ER) Extend Right, which is also called inclusion by G6K. The goal is to increase the context by one step to the right by adding a 0 to the rightmost position in the context. It can be written as follows $L_{[l:r]} \to L_{[l:r+1]}$. Where the coefficients of the lattice vectors in the projected lattice are $(v_1, \ldots, v_d) \to (v_1, \ldots, v_d, 0)$.

- (SL) Shrink Left, where we decrease the context by removing the leftmost coefficient in the context. This operation is also called projection by G6K, and it can be shown as $L_{[l:r]} \to L_{[l+1:r]}$. Now the lattice basis shrinks by simply removing the first vector coefficient as $(v_1, \ldots, v_d) \to (v_2, \ldots, v_d)$

- (EL) Extend left, consists of increasing the context, but this time we do it by adding a coefficient at the leftmost position in the context. G6K here adds a constant c, which is determined by the formula $c = \sum_{j=0}^{d} \mu_{l-1,l+j} v_j$. This operation is also known as Babai-lift and is expressed as $L_{[l:r]} \to L_{[l-1:r]}$, where the coefficients are $(v_1, \ldots, v_d) \to (-\lceil c \rfloor, v_1 \ldots, v_d)$.

- (I) Insert function, which turns out to be a bit complicated. The insert operation lets us insert a vector $\mathbf{b}$ at position i using a local change of basis operation. To do this type of operation a unimodular matrix is needed, and the way this is done in G6K is a complex operation.

- (S) Sieve, which performs a sieving algorithm on a given lattice/sublattice.

Lattice Reduction Strategies

The main concept of the G6K framework is the Pump algorithm, which uses a progressive sieving strategy and "dimensions for free". The "dimension for free" is a trick that was introduced by Léo Ducas in [5] and it is making sieving faster, and G6K uses it in its strategies.

The progressive sieve has usually been done from left to right, but as G6K now provides the possibility for the opposite direction they tested this out and it turned out to be performing better. Due to this the Pump algorithm takes advantage of that and performs the operations in the other direction. Pump is described as

$$\texttt{Pump}_{\kappa,f,\beta,s} : \texttt{Reset}_{\kappa,\kappa+\beta,\kappa+\beta}, (\texttt{EL},\texttt{S})^{\beta-f}, (\texttt{I},\texttt{S}^{s})^{\beta-f} \tag{3.8}$$

The parameters Pump needs are κ, f, β and s. The first parameter κ is an integer value that indicates the start of the lifting context. The second parameter of Pump is f and it represents the "dimensions for free". The blocksize we want Pump to work on is given by the β parameter. At last, the s parameter is either 0 or 1 to indicate whether sieving takes place or not in the last half of the procedure (pump-down phase).

The first part of Pump is the Reset operation, which simply is emptying the database and setting $l = \kappa + \beta$ and $r = \kappa + \beta$. Next up is the extending left and then sieve with one of the algorithms from their sieving algorithm $\beta - f$ times. This first part we call "pump-up". After that we begin the "pump-down" phase. It begins by inserting vectors and then it is optional if we want to sieve the current block or not. In our experiments we will not be using the $s = 1$, meaning we will never sieve on the "pump-down". Hence we can simply look at Pump as the following

$$\texttt{Pump}_{\kappa,f,\beta} : \texttt{Reset}_{\kappa,\kappa+\beta,\kappa+\beta}, (\texttt{EL},\texttt{S})^{\beta-f}, (\texttt{I})^{\beta-f} \tag{3.9}$$

So now we have the Pump algorithm, let us look at some applications of Pump that G6K provides. The first strategy that is presented is the WorkOut routine. The WorkOut contains Pumps that follow the sequence

$$\begin{aligned} \texttt{WorkOut}_{\kappa,\beta,f,f^+,s} :& \texttt{Pump}_{\kappa,\beta-f^+,\beta,s}, \texttt{Pump}_{\kappa,\beta-2f^+,\beta,s}, \\ & \texttt{Pump}_{\kappa,\beta-3f^+,\beta,s} \dots \texttt{Pump}_{\kappa,f,\beta,s} \end{aligned} \tag{3.10}$$

This has been used in their experiments for solving SVP in the SVP challenges. Continuing, the WorkOut routine has also become incorporated into a version of the BKZ algorithm.

Another strategy that also consists of a sequence of Pumps is the PumpNJumpTour. In similarity to WorkOut, also this can be made into a version of the BKZ algorithm. In the PumpNJumpTour we use Pump on blocks of the lattice that we jump between. The PumpNJumpTour can be described using this sequence of Pump:

$$\begin{aligned} \texttt{PumpNJumpTour}_{\beta,f',j} :& \texttt{Pump}_{0,0,\beta-f'}, \texttt{Pump}_{0,j,\beta-f'+j}, \dots, \texttt{Pump}_{0,f',\beta}, \\ & \texttt{Pump}_{j,f',\beta}, \texttt{Pump}_{2j,f',\beta}, \dots \end{aligned} \tag{3.11}$$

We are going to look closer at this strategy in Section 4.2 where we do some experiments with it.

3.4 Darmstadt's SVP Challenges

The SVP challenge is a challenge by Technische Universität Darmstadt to see how big dimensions we are able to solve SVP in [35]. Since we can not always know for sure that we have found a shortest vector, the challenge is to find a vector that is shorter than $1.05\sigma(L)$, where $\sigma(L)$ is the Gaussian heuristic (see Equation 2.4). Today the record for the highest dimension where it has been found a short vector within the approximation factor is 180. The record was set by using a sieving algorithm from G6K using a GPU implementation and can be found in the SVP challenge's Hall of Fame.

There are two ways to get a place in the Hall of Fame. One is finding a shorter vector than the shortest one found for the same dimension. The other way is finding a vector with a Euclidean norm less than the approximation 1.05 of the shortest vector in a higher dimension. Looking at the Hall of Fame we can see that sieving is the dominant method used. In the top ten we also see that it is mostly versions of G6K that are used for solving SVP.

The record for the shortest vector found in the highest dimension was set in 2021. There has not been any higher dimension than 180 where SVP has been solved as we know of today. However, there have been entries to the Hall of Fame in 2022 as well, but these have not been for higher dimensions, but shorter vectors for existing dimensions.

The motivation for the SVP challenge is to better understand the algorithms used to solve SVP, and also for being able to compare the methods to each other. As future cryptographic standards (see Section 1.4) will be based on the hardness of solving SVP, it is important that we understand potential threats against it.

Chapter 4

Experiments and Results

In this section, we are going to report on experiments we have done. All experiments were performed using the G6K library [34]. The experiments have been run on a server with 96 cores of type AMD EPYC 7451, a CPU with 2.5 GHz clock frequency, and up to 192GB RAM available. Each experiment used a single core. Our goal for the following experiments is to test how different sieving algorithms compare to each other and investigate different sieving strategies.

We are going to look at different types of lattices, and the performance of different sieving algorithms while we change the lattice dimension. Will there be a sieving algorithm that is best independent of both lattice type or dimension, or will it vary?

In the second section, we are going to study one of the strategies introduced by G6K [2]. We will try to improve upon a version of the `PumpNJumpTour` presented in [2]. We will then compare the versions on different lattices with different parameters.

For the last experiment, we want to use what we have learned from the two experiments above, tweak the `Pump` algorithm according to the results we got in Section 4.1, and use it in `PumpNJumpTour`. We will then test the different versions of `PumpNJumpTour` on the lattices that gave more different results in the experiment from Section 4.2. Will the observations from the other experiments give us better results than what the default `Pump` does?

REMARK: In the following experiments we will present norms that have been given in the results. These values will be squared as this is how G6K returns the norms as well. This also makes it so we have cleaner numbers to work with.

4.1 Lattice Type vs Sieving Algorithm

One of the first things that seemed interesting was to have a look at how the different sieving algorithms work with different types and dimensions of lattices. Will one sieving algorithm work better regardless of which lattice type or dimension we use to sieve on, or will this vary? In the G6K library, there are chosen default sieving algorithms (`gauss` and `hk3`), hence we assume the G6K team believes these are the best sieving algorithms overall. We suppose this choice has been supported by experiments as well, but these are not really shown, hence we want to have a closer look at this. Also, we wanted to see how the different sieving algorithms react to the different types of lattices in different dimensions, and if there are any patterns.

In these experiments, we will test the lattices of type q-ary, NTRULike, knapsack, and uniform (see Section 3.3.2). We then go through the sieving algorithms: `nv, bgj1, gauss, bdgl1`, and `bdgl2` (see Section 3.3.1) and keep track of how long time they use on a single sieve and their shortest vector found. We do the experiments on five different dimensions of each lattice type: $40, 50, 60, 70$, and 80. As we are not looking at higher dimensions `bdgl3` will never be `bdgl3` (see Section 3.3.1), hence we will not use this for our experiment. We also note that `bdgl2` does not start before we use dimension 60.

In the experiments, we use the G6K library, where we first generate a lattice of a specific type and dimension. Then we will run the different sieving algorithms once each on the specific lattice. For each lattice, we record the length of the shortest vector the algorithm finds, the time it takes to find it, and what dimension we sieve on.

The experiment will be done 10 times where we use different seeds for each of the lattices. After this has been done, we will look at the data we have gathered and calculate the average timing used on each dimension for each lattice type. For all of the experiments, we save the data in tables which can be found in the Appendix. The data consists of which dimension we are working with, how long time the sieving algorithm took on average, a best norm found, which sieving algorithm was used, and the time it took divided by the best time among the different sieving algorithms. Since we are doing the same experiment multiple times we can detect what the common pattern could be, and also catch if there are some special cases for some specific seed used in the experiments. We also note that the tables in the Appendix contain values for `bdgl, bdgl1, bdgl2` and `bdgl3`. This is to observe if the bdgl-algorithms behave as expected.

The lessons learned from these experiments will be further studied in the experiments in Section 4.3

q-ary

We begin with the q-ary lattice, which is described in Section 3.3.2. For the lattices we use the parameters `bits=70` and `k=n/2` on all of the q-ary lattices in the different dimensions n.

In Table 1 in the Appendix we can see the data we collected from our experiment. The data in this table is an average of the 10 different q-ary lattices. The norms in the table are taken from the first lattice we tested to show how all algorithms are able to find the same shortest vector norm for each dimension. Using this table we have visualized the results in Figure 4.1 where the dimension of the lattice is the x-axis and the y-axis is the time divided by the best time. Note that with this scaling of the y-axis, all of the algorithms will have values ≥ 1, and the fastest sieving algorithm will be equal to 1. This scaling is chosen for easier comparison between the algorithms.

We see from Figure 4.1 that the `gauss` algorithm appears to give the best results up until dimension 60. By best we mean that it is both the fastest and is able to find the same shortest vector as the other algorithms. In the figure, we clearly see that dimension 60 also seems to be a crossing point for many of the algorithms. When we reach dimension 70 we see that `bdgl2` becomes the best. We see that `bdgl1` is a little slower than `bdgl2`, but it becomes better than `gauss` in this dimension.

Also `bgj1` is getting better than `gauss` after dimension 60, but not as good as the

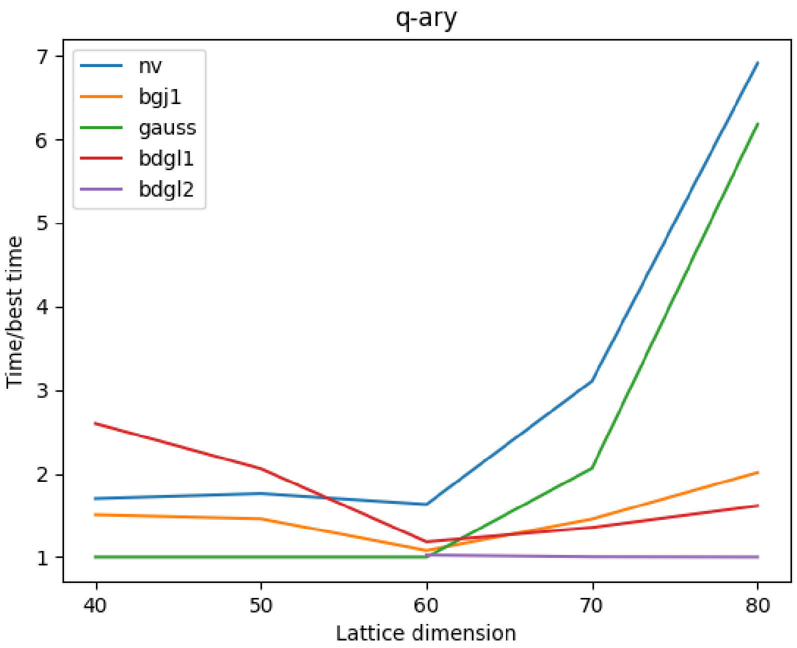


Figure 4.1: Results from the experiment on q-ary Lattice from Table 1

bdgl's. Having a look at nv we see that after dimension 60 it is the worse of the algorithms, and continues to be so. Before dimension 60 it is just middling, so not the best but not really the worst either.

NTRULike

For the NTRULike lattice, we use the parameter bits=70 for all of the dimensions, and we do not use the q parameter for this experiment. Looking at the NTRULike lattice (see Section 3.3.2) we can see some more variation in the results in Table 2 in the Appendix. Also in this table, the timings are averages of 10 runs of each. For the norms, this is the result we got from an NTRULike lattice just to show the pattern of how the different algorithms find short vectors.

At first glance, it could seem that the nv algorithm always is the fastest. However, it always finds a much longer vector than the shortest vector found by the other algorithms. It is also unclear if the nv algorithm has even tried to do any reduction at all. To verify this we check what the basis vectors norms are in the lattice before we sieve. We find that the basis vectors actually have a shorter norm, and when we run nv we are finding worse vectors than before after one sieve. Hence we are just going to ignore this algorithm, and in Figure 4.2 we are not considering the nv algorithm at all.

Another comment that needs to be made is that in one run of the experiment of the bgj1 algorithm in dimensions 70 and 80, we did encounter a "Saturation Error". This error means that we were not able to find a short enough vector, and we will look closer at this in the knapsack section. We have chosen to look away from this when taking the average of the timings as it does not happen often. However, it is important to note that this could happen.

Also in one run of bdgl1 in dimension 40, it uses a much longer time than usual. It also found a longer vector that is more similar to what `nv` finds. Since the difference in the timings is so big, it ends up not really giving us a reasonable average. Hence, we look away from this outlier when we are taking the average. Again, it is important to note that this could also happen when we are working in the NTRULike lattice.

From Table 2 in the Appendix we see that at first only `gauss` always finds the shortest vector. We do know the length of the shortest vector as this is special for the NTRU lattice. However, the other algorithms also find vector lengths that are quite short. If we examine these lengths closely, we can see that these are multiples of the shortest vector that `gauss` has found. This is also a property that the NTRU lattice has, that there will be vectors that are multiples of the shortest vector that are still shorter than $\lambda_2(L)$ (see Section 2.2). This should not be very difficult to implement a check for in G6K, as you can check the following. If the shortest vector found is $v_1\mathbf{b}_1 + \ldots + v_n\mathbf{b}_n$ where all v_i have a common factor f (which we do think happens for NTRULike), then $(v_1/f)\mathbf{b}_1 + \ldots + (v_n/f)\mathbf{b}_n$ be f times shorter.

In Figure 4.2 we have not taken into consideration that the different algorithms find different shortest vectors at first. We do this as `gauss` is both fastest and finds the shorter vectors until the rest of the algorithms catch up. When we reach dimension 70 all of them are able to find the same shortest vector, and this is usually the case. `Gauss` is also the fastest algorithm until dimension 70, so we say that `gauss` is the best algorithm until dimension 70 is reached. When we look at dimension 70 we see that the `bdgl1` algorithm is the fastest now. However, as we increase the dimension again to 80, we see that it is now `bdgl2` that is the fastest. An observation that we can make from Table 2 in the Appendix is how different the bdgl-algorithms behave even though they should be the same (see Section 3.3.1). This contrasts with the q-ary lattice where they were nearly identical to each other. So for the bdgl-algorithms, it seems that for NTRULike it behaves more unexpectedly even though it should be the same.

For `bgj1` we see that this algorithm starts to outperform `gauss` in dimensions higher than 70. So an observation is that for the NTRULike lattice the crossing point between `gauss` and the other algorithms seem to be in dimension 70. Which is different from the q-ary lattice, where it was in dimension 60.

Knapsack

Now we are going to look at the knapsack lattice, where we use the parameter `bits=70` for all the dimensions of the lattice. With a quick look at Table 3 in the Appendix, we see that all of the algorithms are able to find the same shortest vector, meaning we mostly need to look at how fast the algorithms are. Figure 4.3 shows the data plotted from this table, where the x-axis represents the dimension, and the y-axis represents time/best time. As with the other tables, the numbers are an average of the timings from 10 runs, and the norms are taken from a knapsack lattice we used.

The first thing we need to comment on is that we get saturation errors for all of the bdgl-algorithms in some of the runs in dimension 60. The error is raised when the sieve cannot find short enough vectors according to some radius. However, why we get this error is unknown, and on the G6K's GitHub page, one of the creators Léo Ducas said the following, when asked about this issue [6]:

> Saturation issues remain mysterious to us as well, and occur in specific scenario indeed. We have not found a satisfying solution to the issue.

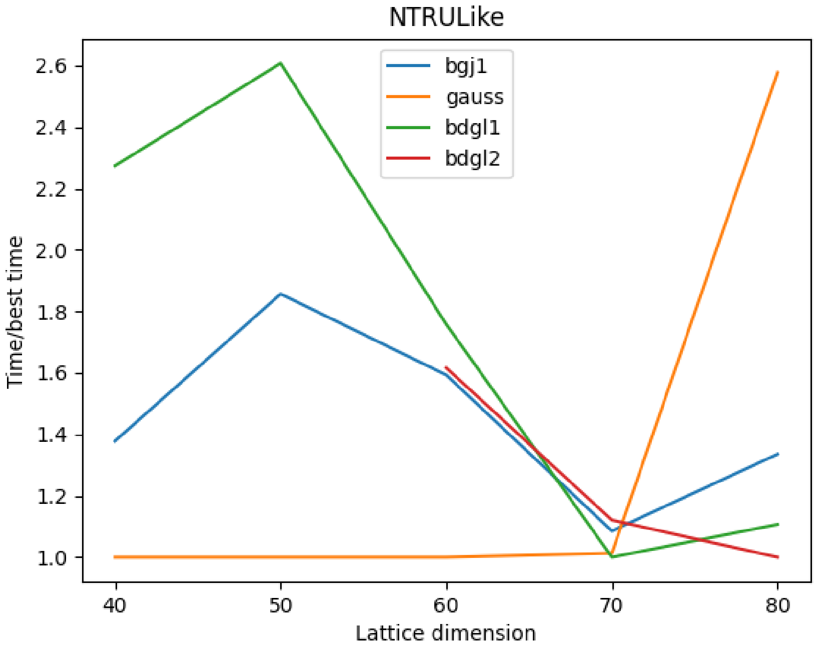


Figure 4.2: Results from the experiment on NTRULike Lattice from Table 2

So to find the average of the different bdgl versions, we have only taken the average of those who did not get a saturation error. Hence also in Table 3 in the Appendix we show a run without any saturation errors.

In similarity to the q-ary lattice, we see that `gauss` is the best algorithm up to dimension 60. When we reach dimension 70 it again changes, and now the bdgl-algorithms are faster with `bdgl2` being the fastest. Hence we see a pattern now where when the dimension grows the bdgl-algorithms become better. We also see that `bgj1` compared to `gauss` is a better option of an algorithm as the dimension increases from 60. The `nv` algorithm also in similarity to `gauss` gets worse after dimension 60, and before this, it is not really the best or worse.

An observation we can note is that the figures for both the q-ary and the knapsack lattice are very similar. We see the same patterns for the algorithms, and the crossing point is 60 for both.

Uniform Lattice

At last, we have a look at the uniform lattices. For the uniform lattice, we use the parameter `bits=20`, and we keep this smaller than the other lattice types as the basis is just an unstructured matrix consisting of randomly chosen integers. The norms of the shortest vectors the different sieving algorithms find are all the same, hence we just look at the best times. The results are given in Figure 4.4, where we again have dimension as the x-axis and the y-axis as time divided by the best time of all the algorithms. The plot is based on the data in Table 4 in the Appendix, which contains the average timings for 10 runs of each algorithm in each dimension. The norms in the table are from a uniform lattice used in the experiment.

Again the `gauss` algorithm is the fastest up to dimension 60, and then the bdgl-

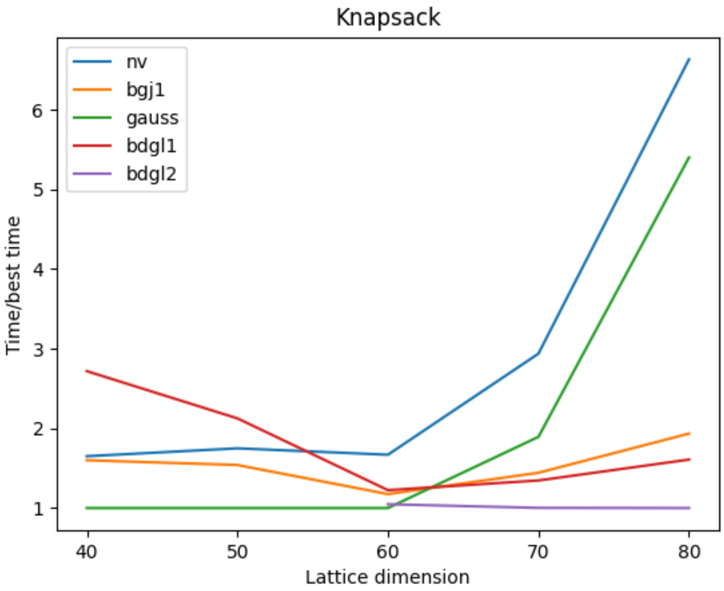


Figure 4.3: Results from the experiment on knapsack Lattice from Table 3

algorithms take the lead. The bdgl-algorithms behave a bit differently, where `bdgl1` is a little slower than `bdgl2`.

Taking a look at `bgj1`, we see it is slower than the bdgl's, but faster than `gauss` after dimension 60. Again `nv` is the worst algorithm after dimension 60, and before this point, it is not the best either. This is the same behavior as seen in both q-ary and knapsack lattices, and all of their plots follow the same patterns.

Summary

To summarize this experiment we have tested out different sieving algorithms on different types of lattices. We have looked at the different dimensions to see if we find if a sieving algorithm works best in general or if there are any patterns.

To conclude, it seems that overall `gauss` is the best sieving algorithm for all of the lattices, up until dimension 60. The authors of G6K had a similar conclusion, but their crossing point was mentioned in [2, Sec. 5] to be at dimension 50 and not 60, which we have found it to be. We also see that the NTRULike lattice behaves differently than the other lattice types and the crossing point appears to be closer to 70 than 60.

After the crossing point, it is one of the bdgl-algorithms that work better. This applies to most of the lattice types. Also `bgj1` gets ahead after this crossing point, but is not a better option than one of the bdgl's. Also, another comment for the bdgl-algorithms is that `bdgl1` is usually the worse choice, and `bdgl2` normally performs better. The only exception we saw for this is in the NTRULike lattice, where in dimension 70 it performed the best of the algorithms.

Another thing we also can remark from our experiments is that `nv` follows the same pattern as `gauss`, but always takes longer time than `gauss`. If we look at the tables in the Appendix, we see that `nv` uses the longest time after dimension 60 out of all the algorithms. This behavior was expected as it is believed to be the least efficient among the sieving algorithms. We have not seen a single lattice where it makes sense to sieve

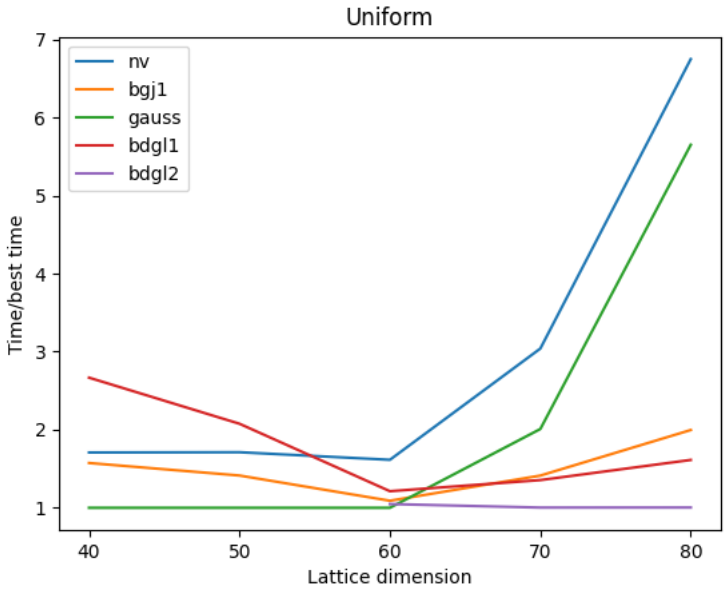


Figure 4.4: Results from the experiment on uniform Lattice from Table 4

with nv. So here we see that theory and practice agree.

In conclusion, gauss works well up until dimension 60 for all of the lattice types we have tested on. After this crossing point is reached it is a version of bdgl that works better in general. At most times we see that bdgl2 is the best choice between the bdgl's. As G6K has set the default bdgl to have blocks=2 (identical to bdgl2), it makes sense with the results we have gotten that they have chosen this.

Comment on triple sieve (hk3)

The default algorithm set for the sieving in G6K is either gauss or hk3, depending on which dimension you have (50 is set to be the cross-over point). However, when we tried to sieve using a single sieve with hk3 we only got Saturation Errors. The only exception was for lattices in dimension 40, while for all the others it did not work. We sent an issue to G6K's GitHub page and asked why it was not possible to use, but at this time of writing, we have not received an answer for this [26]. Having also looked through the code base it could seem to be some kind of rank loss bug, but we cannot know for sure. Hence, we have not used the hk3 algorithm for this experiment.

4.2 Reversing PumpNJumpTour

An advanced sieving strategy in G6K is the PumpNJumpTour. The strategy uses the Pump algorithm (3.9) on different blocks in the lattice, and we recall from Section 3.3.3 that PumpNJumpTour is following the sequence

$$\begin{aligned}\texttt{PumpNJumpTour}_{\beta,f',j} :& \texttt{Pump}_{0,0,\beta-f'}, \texttt{Pump}_{0,j,\beta-f'+j}, \dots, \texttt{Pump}_{0,f',\beta}, \\ & \texttt{Pump}_{j,f',\beta}, \texttt{Pump}_{2j,f',\beta}, \dots\end{aligned} \tag{4.1}$$

BKZ can be seen as a special case of PumpNJumpTour where $j = 1$. In PumpNJumpTour the block we are working on can move more than one step at each time, and we say that it can take a *jump* further in the lattice. The algorithm does this from left to right. What if we tweak the algorithm to change the order we are doing things to see if this could be a better approach?

In this experiment we are going to look at a simplified version of the strategy which is presented in [2]. The simple version works as follows

$$\texttt{SimplePumpNJumpTour}_{\beta,f,j} : \texttt{Pump}_{0,f,\beta}, \texttt{Pump}_{j,f,\beta}, \texttt{Pump}_{2j,f,\beta}, \dots \tag{4.2}$$

The difference between PumpNJumpTour and SimplePumpNJumpTour is that PumpNJumpTour has a phase at the beginning where it uses multiple Pumps on the first block. It will iteratively go through the first block and extend the scope it used Pump on until we reach the full blocksize. After this phase, it will start to use Pump only on whole blocks. SimplePumpNJumpTour is not implemented in G6K, so we have slightly adapted the code for the PumpNJumpTour method (pump_n_jump_bkz_tour()) in G6K to go from 4.1 to 4.2.

For the experiment, we need to implement a reversed version of SimplePumpNJumpTour, which we can use in the comparison. In the reversed version of the simple algorithm, we reverse the order of the contexts that we execute Pump on. The reversed version of SimplePumpNJumpTour can be specified as

$$\begin{aligned}\texttt{ReversedPumpNJumpTour}_{\beta,f,j} :& \texttt{Pump}_{n-\beta,f,\beta}, \texttt{Pump}_{n-\beta-j,f,\beta},\\ & \texttt{Pump}_{n-\beta-2j,f,\beta}, \dots, \texttt{Pump}_{0,f,\beta}\end{aligned} \tag{4.3}$$

Now that we have reversed the order, we expect the algorithm to show a different behavior than SimplePumpNJumpTour as the short vectors that we find will be put to the left in the basis. This means we will reuse them as the context block moves to the left in contrast to SimplePumpNJumpTour where newly found short vectors are not used when the block moves to the right in the basis. One of the reasons this could be interesting was that Albrecht et al., reported in an experiment ([2, sec. 4.1]) that pumping in the left direction (right to left) performed better than in the opposite way. We wonder if this could also apply to the PumpNjumpTour order.

In the following experiments, we will look at the same lattice types we looked at in Section 4.1. We will then see how SimplePumpNJumpTour and ReversedPumpNJumpTour compare on different lattice types and see how their behavior differs from each other. We will report on the shortest norm the algorithms are able to find, and also look at how well each of them performs on average.

REMARK: Both the simple and reversed versions of PumpNJumpTour use about the same amount of time on each round. As there were no notable differences here, we have not chosen to focus on the timings in this experiment. However, what we do need to remember is that the bigger we set the jump parameter the shorter time a tour takes.

q-ary lattice

In the first part of the experiment, we look at the q-ary lattice. The lattice will be of dimension 120 with bits=70 and k=60. First, we keep the parameter blocksize as 30, and the jump parameter is kept as the default $\texttt{jump} = 1$. We also set a seed to make sure we are working on the same lattice for all the experiments.

We will run a single tour of SimplePumpNJumpTour and ReversedPumpNJumpTour to see how different results we get. After a single run of both algorithms on the same lattice we got the following results

- Shortest vector norm after one round of SimplePumpNJumpTour:
  42214994127042295611392

- Shortest vector norm after one round of ReversedPumpNJumpTour:
  19981260646730723958784

We see that we get different results, and $1.998 \times 10^{22} < 4.221 \times 10^{22}$, which means that the reversed version was able to find a shorter vector. This is interesting as it corresponds to our hypothesis, but does this apply to multiple rounds of the pumpNJumpTour algorithms?

We are going to run SimplePumpNJumpTour and ReversedPumpNJumpTour 10 times each to see how the behavior is affected. Will the reversed algorithm still do better?

To get a better picture of the behavior we are going to look at the average behavior. Both SimplePumpNJumpTour and ReversedPumpNJumpTour will be run on 10 different q-ary lattices. All of the lattices use the same parameters, which are dimension=120, k=60, bits=70. We will examine the algorithms for three different values of blocksize, which are $30, 40$, and 50, and we will also look at two different jump values, 1 and 5. For each blocksize we will look at how the different algorithms with different jump values compare to each other.

For the experiments, we have looked at the following for each blocksize. We have lists $A_1, \ldots, A_4$ which each represent a different version of the algorithms. So for instance A_1 represents the SimplePumpNJumpTour with jump=1. These lists again contain 10 lists $s_1, \ldots, s_{10}$ each, where each s_i is a different q-ary lattice. A list s_i contains the shortest vector norm found in each round of the algorithm, so s_i will contain 10 values as we do 10 rounds. All of $s_1, \ldots, s_{10}$ contains the norms for the different lattices. When we have done this for all of the 4 different versions of the algorithms, we start to compare them.

First we look at how $A_1[s_1] \ldots A_4[s_1]$ compare to each other. By this, we mean first find the smallest norm, and then update the values by dividing all of $s_1[j]$ with the smallest value $min(A_i(s_1[j]))$. This is done for each $A_i[s_i]$. Now all of A_i contains lists of how well they usually compare against the other algorithms.

The next step is to take the average of each A_i. To do this we look at the s_i's, which we recall each has 10 values. We take the average of $s_1[1], \ldots, s_{10}[1]$, and get one value that tells us how well this algorithm compares to the others in the first round of it. We do this for all $s_i[j]$. Now we have a list of ten values in total, which tells us how well A_i compares to the other A_i's. The same is then done for all of the A_i's. Now we have the average values for each of the algorithm versions, and these are then plotted in the figures. In the figures, we have rounds as the x-axis and the y-axis represents how well they compare against the other algorithms. As with the experiments in Section 4.1, the value 1 is the best an algorithm can get.

The experiments are divided based on the blocksize, and we begin with blocksize=30. The results are plotted in Figure 4.5.

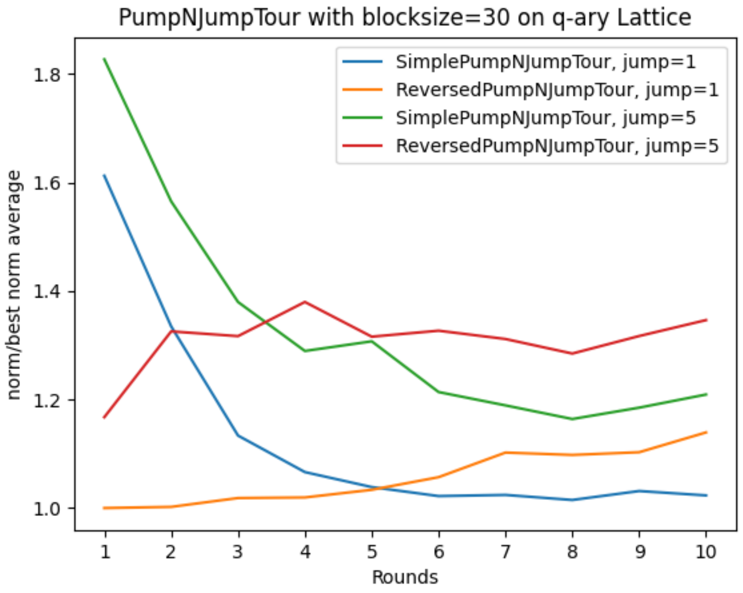


Figure 4.5: Results from the experiments on q-ary lattice with blocksize 30

From Figure 4.5 we see that in the first iteration, it is the `ReversedPumpNJumpTour` that finds the shorter vectors, but as we move on this changes. After a few iterations, it is the `SimplePumpNJumpTour` that finds the shorter ones. Now we do see an opposite behavior than what we were expecting. With the algorithms using `jump=5`, we can see for `blocksize=30` they are worse than those who have `jump=1`. However, the algorithms using `jump=5` also use shorter time on each tour, meaning this method is faster. So we have a trade-off between the timings of the tours and the shortest norm they are able to find. However, at this point it seems like `SimplePumpNJumpTour` works the best after a few rounds.

We will now explore the other `blocksize` values for our experiment, and see how this affects the algorithms. Next up is `blocksize` of 40, and the results can be seen in Figure 4.6. Again we observe that both instantiations of `ReversedPumpNJumpTour` find shorter vector norms in the beginning, but they are overtaken by `SimplePumpNJumpTour` after a few rounds.

At last we have `blocksize = 50`, where we find the results in Figure 4.7

This plot is very similar to what we already have seen. It begins with `ReversedPumpNJumpTour` being the best in both versions and then this changes to `SimplePumpNJumpTour`. In this case, it is very clear that `SimplePumpNJumpTour` with `jump=1` is the best algorithm after the first round. We also note that for this `blocksize` there is also a clearer pattern for the different versions. For `SimplePumpNJumpTour`, we see that with both `jump=5` and `jump=1` it follows the same pattern. The difference is that when `jump=5` the plot is shifted higher than `jump=1`. The same applies for `ReversedPumpNJumpTour`.

To summarize the three plots we can see that the behavior is quite similar in the three cases. `ReversedPumpNJumpTour` initially finds the shortest vector among the tested algorithms, but is outperformed by `SimplePumpNJumpTour` over multiple rounds. Another observation is that the crossing point between `SimplePumpNJumpTour` and `ReversedPumpNJumpTour` with `jump=1` comes faster the bigger we set the `blocksize`

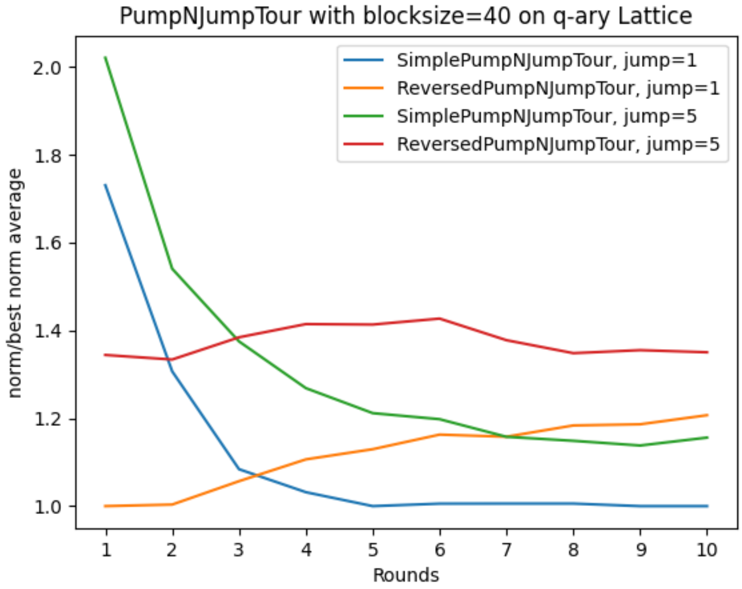


Figure 4.6: Results from the experiments on q-ary lattice with blocksize 40

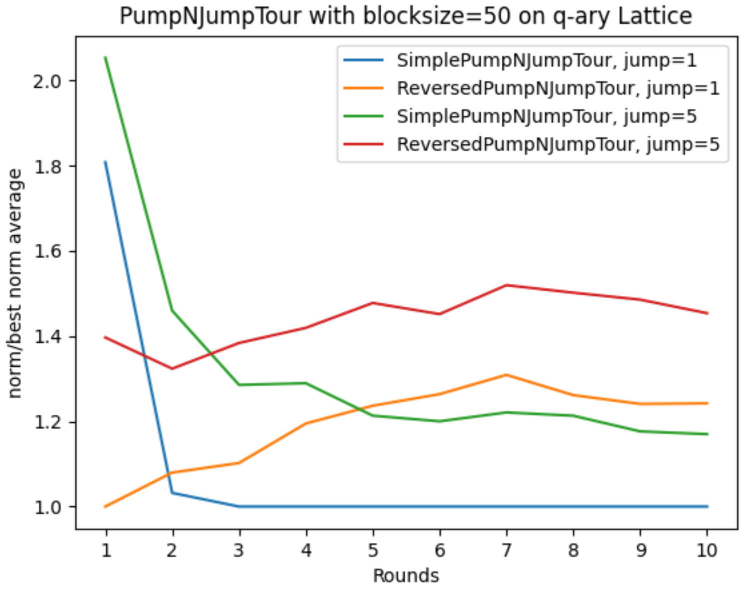


Figure 4.7: Results from the experiments on q-ary lattice with blocksize 50

as. The same applies for SimplePumpNJumpTour and ReversedPumpNJumpTour with jump=5.

As we now have seen it seems to be the original order that finds the shorter vectors after a while. However, after the first round the shorter vector is found by ReversedPumpNJumpTour, and there is also a big difference in the norm. Hence it could be interesting to see if beginning the tour with a reversed order has any impact on if we are able to find shorter vectors faster.

For this experiment, we test with the different blocksizes and jump values as we did before. As before, we keep running the algorithms for 10 rounds on 10 different q-ary lattices. We do one where we just iterate SimplePumpNJumpTour, and one where we start with one iteration of ReversedPumpNJumpTour and then iterate SimplePumpNJumpTour from there on, which we call AlternativePumpNJumpTour.

However, the results showed us that we did not get any better performance doing this, and it did not benefit from working on a better basis, which ReversedPumpNJumpTour provided.

NTRULike lattice

We look at the NTRULike lattice where we have bits as 70 and dimension as 60. We note here that the whole lattice then will be of dimension 120, as it doubles the input in the creation. A single run of ReversedPumpNJumpTour and SimplePumpNJumpTour gives the following norms

- Shortest vector norm after one round of SimplePumpNJumpTour: 60
- Shortest vector norm after one round of ReversedPumpNJumpTour: 60

The results are the same for both of the versions, hence we do not really see any difference for the NTRULike lattice. Due to this, it is not really interesting to look closer at this type of lattice.

Knapsack lattice

Next is the knapsack lattice where we keep the bits as 70 and dimension as 120. We then got the following norms from a single run of the two different PumpNJumpTour versions

- Shortest vector norm after one round of SimplePumpNJumpTour: 35
- Shortest vector norm after one round of ReversedPumpNJumpTour: 28

With the knapsack lattice, it is also the ReversedPumpNJumpTour that returns the shorter lattice vector after one run. We see that the knapsack behaves quite similarly to the q-ary lattice.

Now we are going to run 10 iterations of the algorithms on 10 different knapsack lattices. We split the experiments based on blocksize, and we again look at the values $30, 40$, and 50. For each blocksize we will consider SimplePumpNJumpTour and ReversedPumpNJumpTour with the parameter jump as 1 and 5. This means we will compare 4 versions in total per blocksize. The plots are made from data collected in the same way as described in the q-ary part of this experiment. The x-axis represents rounds, and the y-axis is the norm found by the algorithm divided by the shortest found in that round, meaning how the different algorithms compare to each other.

First we look at blocksize=30, where we can see the results in Figure 4.8. The plot shows us that ReversedPumpNJumpTour with jump=1, as expected in the first round finds the shorter norm, but this time it also works the best for almost all of the rounds (except round 2). This is a contrast to what we saw with the q-ary lattice above. The result is more of what we expected would happen, but will this be true for the other blocksizes?

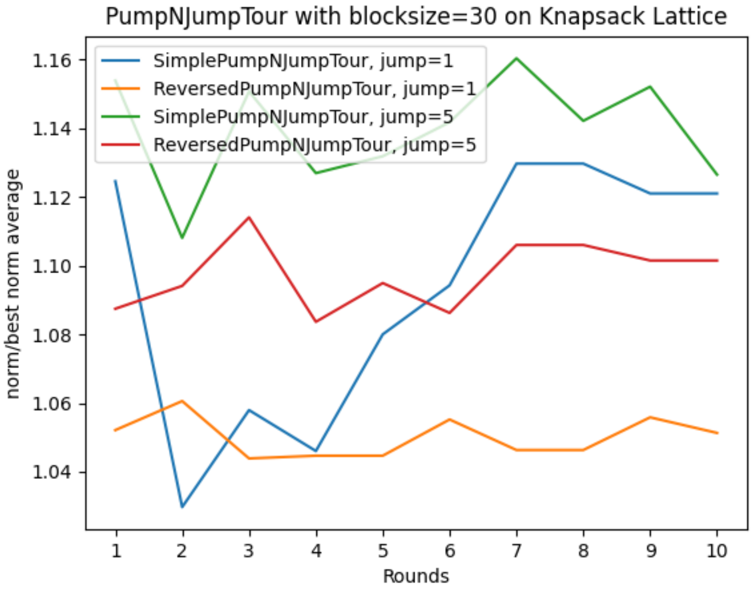


Figure 4.8: Results from experiments with PumpNJumpTour on knapsack lattice using blocksize=30

We increase the blocksize to 40, and the results har shown in Figure 4.9. In comparison to Figure 4.8, we see that this plot is very different. First, we note that between ReversedPumpNJumpTour and SimplePumpNJumpTour with jump=1, it is the reversed version that is the best. However, in this case SimplePumpNJumpTour with jump=5 does a better job than both of them and in the end is able to find the shorter vector on average. The gap between SimplePumpNJumpTour with jump=5 and ReversedPumpNJumpTour with jump=1 is quite close, thus in this case it is difficult to know exactly which of them works the best, as they are quite similar. However, what we can note is that the running time of the algorithm is shorter when jump=5 than jump=1.

When blocksize=50 we get the plot in Figure 4.10. We can again see that it is SimplePumpNJumpTour with jump=5 and ReversedPumpNJumpTour with jump=1 that works the best for blocksize=50. If we look very closely at the plot we can see that in the end it is ReversedPumpNJumpTour that usually finds the shorter vector. However, as the plots are so similar in the end, it is difficult to say for sure that one is better than the other. This is the same we saw for blocksize=40.

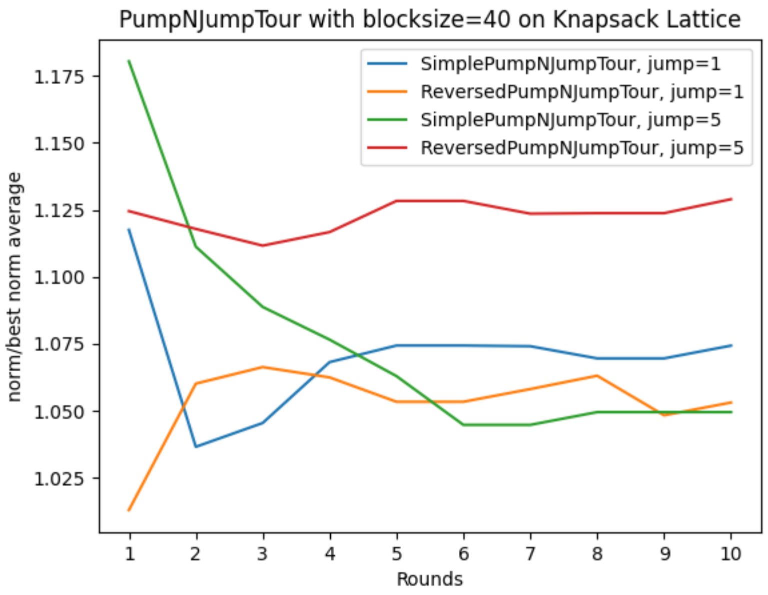


Figure 4.9: Results from experiments with PumpNJumpTour on knapsack lattice using `blocksize=40`

Uniform lattices

At last, we have the uniform lattice where the `bits` are at 20, and of dimension 120. For a uniform lattice, we found that both versions find the same shortest norm which was

- Shortest vector norm after one round of `SimplePumpNJumpTour`: 143126666654444
- Shortest vector norm after one round of `ReversedPumpNJumpTour`: 143126666654444

We again run both `SimplePumpNJumpTour` and `ReversedPumpNJumpTour` 10 rounds to see how the algorithms then behave. This was done on 10 different uniform lattices with the `blocksize` as $30, 40$, and 50 with `jump` as 1 and 5. For these most of the time, the algorithms were all able to find the same shortest vector and did not improve much. Hence there is no algorithm that is distinct from the others, and it is not that interesting to look at the `uniform` lattice any further. We will therefore not do any more experiments with the uniform lattice.

Summary

In this experiment, we have taken a closer look at the order of sieving contexts in the `PumpNJumpTour` strategy. We have looked at what happens if we go from right to left instead of left to right, as this will reuse the inserted short vectors when doing `Pump` in the next context. Our hypothesis was that this could be a more efficient way to obtain shorter lattice vectors. To experiment we first did a single round of the tour on the q-ary lattice. This seemed to support our initial hypothesis. We also tested this on the other lattice types, and it also showed that the reversed order found a shorter or the same length as the original order after a single round.

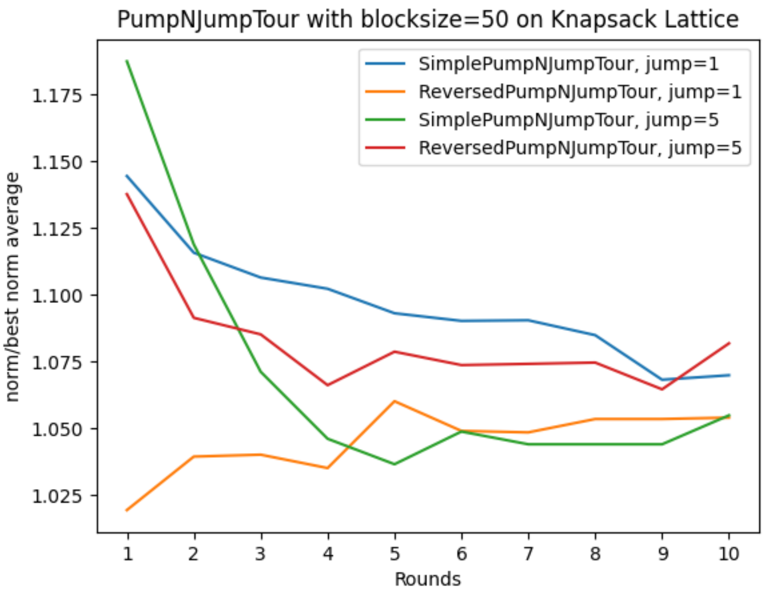


Figure 4.10: Results from experiments with PumpNJumpTour on knapsack lattice using `blocksize=50`

However, as we ran the different `PumpNJumpTour` versions multiple times we see another behavior. We run both `SimplePumpNJumpTour` and `ReversedPumpNJumpTour` for 10 rounds each. In this part of the experiment we mainly focused on knapsack and q-ary lattices. We tested with different values of the parameters `blocksize` and `jump`.

For the q-ary lattice we see that `SimplePumpNJumpTour` is the better choice, and the bigger blocksize we have, the quicker `SimplePumpNJumpTour` becomes the better option. As for the `jump` parameter, we can see that it does not benefit to increase this from 1, as it is always `SimplePumpNJumpTour` with `jump=1` that is the best option according to our plots (4.5, 4.6, 4.7).

On the other hand, we see different results for the knapsack lattice. The `ReversedPumpNJumpTour` will usually be able to find the shorter vector of the different versions with `jump=1`. However, in this case, we also see that `SimplePumpNJumpTour` is benefitting from an increased jump value, and is also a good choice of algorithm version for the knapsack lattice. When we have the `blocksize=30`, it is however clear that `ReversedPumpNJumpTour` works the best.

So to conclude we see that the choice of strategy to use depends on which lattice type we want to do lattice reduction on. Further, we can decide on the `jump` parameter depending on what we set the `blocksize` as.

4.3 Changing the `Pump` Algorithm for the `PumpNJumpTour`

In the experiments above we have studied which sieving method works the best according to lattice type and dimension. We have also experimented with one of the strategies shown in [2], the `PumpNJumpTour`. For this experiment, we are using the results we have obtained from the two previous ones. To do this we are going to do some changes to the `Pump` algorithm, as this is used in `PumpNJumpTour`, and it is in `Pump` we set the sieving method. To check which sieving method `Pump` is going to use it will first check if the block it is going to sieve is bigger or smaller than the crossing point value. The crossing point is called the gauss_crossover in G6K, and it is set as 50. The sieving algorithm used will then either be gauss or hk3 depending on the gauss_crossover, or if it is an initial sieve. Pump will always use gauss as inital sieve. As mentioned in Section 4.1, if we use hk3 as sieve first, we get a saturation error.

For this experiment, we will change the gauss_crossover to 60, rather than 50. The reason for this was that we saw in the experiments in Section 4.1 that gauss was the fastest algorithm until this dimension. What we do need to have in mind, is that we have not tested with the hk3 algorithm, so this could potentially be faster before this crossing point. The other change we will be doing is what sieving algorithm Pump will use after the gauss_crossover, which will be bdgl2 instead of hk3.

Having changed Pump we are going to use it in the SimplePumpNJumpTour (Equation 4.2) and ReversedPumpNJumpTour (Equation 4.3). Now we are going to compare how well the SimplePumpNJumpTour works with the modified Pump against the default Pump. The same is going to be done for ReversedPumpNJumpTour. We will then compare these two versions of SimplePumpNJumpTour and ReversedPumpNJumpTour against each other.

The experiment will focus on q-ary and knapsack lattices as these were the only two types of lattices that gave different results between reverse and simple PumpNJumpTour in the experiment in Section 4.2.

Knapsack lattice

We begin with the knapsack lattice as with this lattice we got better results using the ReversedPumpNJumpTour in the past experiment. The knapsack lattice is now set to have bits=70, and dimension=150. As mentioned in the last experiment the PumpNJumpTour uses two parameters: blocksize and jump. The first thing we are going to comment on is the blocksize, as the dimension of the block is what decides on which sieving algorithm Pump is choosing. Therefore we have to look at a bigger blocksize than in the last experiment, so the sieving algorithm will do significant sieving on both sides of the crossover point.

We focus on how the different strategy versions behave when we set blocksize=70 and set jump=1. First, we test it out with SimplePumpNJumpTour, and we will report on how long time the strategy uses, and the length of the shortest vector found after 10 rounds. Recall that the default values are gauss_crossover=50, sieve_after_crossover=hk3, and the modified values are gauss_crossover=60, sieve_after_crossover=bdgl2.

- SimplePumpNJumpTour with default values: Norm = 18, time = 486.9418 sec
- SimplePumpNJumpTour with modified values: Norm = 16, time = 319.0167 sec

What we can gather from this is that the SimplePumpNJumpTour using our modified values takes less amount of time than what the default values do, and it is also able to find a shorter vector than the default values. Let us see if it will be any different when we use ReversedPumpNJumpTour.

- ReversedPumpNJumpTour with default values: Norm = 16, time = 486.1472 sec

- ReversedPumpNJumpTour with modified values: Norm = 16, time = 339.2943 sec

ReversedPumpNJumpTour is able to find a shorter/ same norm as SimplePumpNJumpTour, but we also see the same pattern in timings. So with blocksize=70 it seems bdgl2 is faster than hk3.

Now we have only looked at one knapsack lattice, so to get a better understanding of the common behavior of the different versions, we test on 10 different knapsack lattices. We will then report on how well they perform against each other based on both timings and norm length found. For the timings, it is simply to take the average time each algorithm uses on the 10 different lattices, and then divide it by the shortest time of the algorithms. The norms, on the other hand, are done the same way as described in Section 4.2 for q-ary lattices. The results can be found in Table 4.1.

Algorithm	(avg) time/best time	(avg) norm/best norm
SimplePumpNJumpTour default values	1.51903	1.07046
SimplePumpNJumpTour modified values	1.0	1.06583
ReveresedPumpNJumpTour default values	1.50916	1.0776
ReversedPumpNJumpTour modified values	1.02747	1.06542

Table 4.1: Table showing how the different PumpNJumpTour versions compare to each other in knapsack lattices based on time and norm found in the last round. Data is based on Table 5 in the Appendix

In Table 5 in the Appendix, we see the data Table 4.1 is based on. What we can gather from Table 4.1 is that it is the modified versions of the algorithms that are faster. What we also can see is that for the last round of the tours, each algorithm is usually able to find the same norm on average. The modified values on average do more often also find the shorter vector, meaning that for the knapsack lattice, it seems the modified versions are the better choice.

q-ary lattice

For the q-ary lattice we know that SimplePumpNJumpTour finds shorter vectors than ReversedPumpNJumpTour from previous experiments. However, we do still want to compare how hk3 and bdgl2 behave.

The q-ary lattice has dimension=150, bits=70 and k=75 as parameters. In the following experiments we will for ReversedPumpNJumpTour and SimplePumpNJumpTour use the parameters blocksize=70 while we keep jump=1.

At first we look at the difference between hk3 and bdgl2 using SimplePumpNJumpTour, and we will report the norm found after 10 rounds and the time these 10 rounds took.

- SimplePumpNJumpTour with default values: Norm = 1.3655×10^{22}, time = 988.8556 sec

- SimplePumpNJumpTour with modified values: Norm = 1.2538×10^{22}, time = 920.2901 sec

We see that SimplePumpNJumpTour using the modified values for gauss_crossover and sieve_after_crossover, finds the shorter vector. It does also spend less time than the default version to find it. Will this also be the case for `ReversedPumpNJumpTour`? We get the following results from `ReversedPumpNJumpTour`

- `ReversedPumpNJumpTour` with default values: Norm = 1.8498×10^{22}, time = 968.0108 sec
- `ReversedPumpNJumpTour` with modified values: Norm = 2.0493×10^{22}, time = 952.5273 sec

This time we see the opposite behavior of what we saw with `SimplePumpNJumpTour`, as with `ReversedPumpNJumpTour` it is `hk3` that finds the shorter vector. The timings are also not as different in this case compared to knapsack, but we see that our modified values also for q-ary are faster.

However, we need to check if this is the common behavior of the four versions. We do the same as we did with the knapsack lattice, run them on 10 different q-ary lattices, and take the average on times and norms found compared to each other. The results are reported in Table 4.2, where the data comes from Table 6 in the Appendix.

Algorithm	(avg) time/best time	(avg) norm/best norm
`SimplePumpNJumpTour` default values	1.06974	1.02526
`SimplePumpNJumpTour` modified values	1.0	1.47986
`ReveresedPumpNJumpTour` default values	1.04187	1.03187
`ReversedPumpNJumpTour` modified values	1.02841	1.56588

Table 4.2: Table showing how the different PumpNJumpTour versions compare to each other on q-ary lattice based on time and norm found in the last round. Data is based on Tabel 6 in the Appendix.

From Table 4.2 we see that the time each algorithm uses is very similar on average. However, when we look at the norms found it shows that `hk3` usually finds the better norm with either `SimplePumpNJumpTour` or `ReversedPumpNJumpTour`. Hence when using `PumpNJumpTour` on a q-ary lattice it would be best to use what G6K already provides: `PumpNJumpTour` with default values.

Summary

We have looked closer at `PumpNJumpTour`, and specifically the `Pump` algorithm it utilizes. G6K has set default values for which sieving algorithm `Pump` uses, and when to change the sieving algorithm. From an earlier experiment in Section 4.1, we concluded with `bdgl2` being the best sieving algorithm from dimension 60, and in this experiment, we have compared this to the default values in G6K. We look at both `SimplePumpNJumpTour` and `ReversedPumpNJumpTour` on q-ary and knapsack lattices. The reason we have only looked at these two lattice types was that from the experiment in Section 4.2 these were the types where simple and reversed `PumpNJumpTour` give different results.

For the q-ary lattice, there was very little difference in the time it took for each algorithm to finish. However, the norm found had a bigger variance, and we could see that `bdgl2` was not able to find as short norms as `hk3` did. There is not much difference between `ReversedPumpNJumpTour` or `SimplePumpNJumpTour`, but as we saw in 4.2, the `SimplePumpNJumpTour` is ultimately the best strategy for q-ary.

On the knapsack lattice, the strategies behave more differently than for the q-ary lattice. The norms they find in the end are approximately the same, but the timings differ. We see that our modified values with `gauss_crossover=60` and `sieve_after_crossover= bdgl2` are the faster of the strategies. As the norms do not differ that much, it could be better for knapsack lattices to use `PumpNJumpTour` with the modified values of `gauss_crossover=60` and `sieve_after_crossover=bdgl2`. The difference between `ReversedPumpNJumpTour` and `SimpePumpNJumpTour` are not that notable for this experiment, hence one could use either.

Chapter 5

Discussion and Conclusion

5.1 Discussion

One of the first things we had to do before starting the experiments was to download and be able to use the G6K library. Getting G6K to work after downloading it turned out to be rather difficult. As G6K requires that you have FPLLL/FPYLLL, which then again requires other extensions, meaning a lot had to be installed. To download all of these extra features was rather difficult starting from scratch on a normal laptop. However, when we were using the server we were using to conduct the experiments on, it worked without any trouble because the machine already had all of the required libraries installed. So it was rather difficult and frustrating to get G6K to work on a machine without any of the required libraries.

When we first had G6K up and running we were able to test out the library, which was a nice experience. G6K provides examples of how to create lattices and do sieving on them. They also have easy versions of the `Pump` algorithm and examples of how to use it. This was helpful, as one got to play around with G6K without many of the extra elements it requires for using the other algorithms they have implemented.

After getting the hang of how the library is built up, it was easy to set up experiments so that we were able to do what we wanted to do. Adding algorithms, or editing those that were in the library was easy, and changing the parameters as we did in the last experiment in Chapter 4, was pleasant.

The documentation, on the other hand, could be improved. We struggled with understanding why the triple sieve (`hk3`) did not work to sieve straight away. It seemed like this was a bug, but there was not any explanation as to why we could not do that. Hence for this part, G6K could improve. Also as reported for the experiments we experienced "Saturation errors", and as stated the G6K team was not always sure as to why they occurred sometimes. This was a frustrating error to get because we did not have any idea as to why this was the case or how to fix the problem.

Even though G6K can be improved, it is an important tool for lattice reduction. In Section 3.4 we saw that G6K is used to solve SVP for the biggest dimension (180) as of now. The largest dimension a different algorithm has been able to solve SVP for is 152. This indicates how strong the tool G6K can be for lattice reduction. G6K was quite easy to use, meaning it should be doable for many people to utilize G6K's potential. Since it also is built in a way to incorporate your own strategies, it is well suited for further experimentation.

5.2 Future Work

In our experiments, we have worked with G6K and our observations indicate some things that can be looked at further. As the G6K strategies contain many parameters in their implementation, there are many more experiments that can be done by simply changing these. However, we are going to look closer at some other work that could have been done based on the results we have got.

First, for our experiments, our results are only based on 10 different lattices of each lattice type. As we chose to focus on trying out as many different things as possible, we did not have the time to test more. Hence for some of the more interesting results, they could be redone for more lattices to see if this is indeed the expected results for the strategy. By doing this we will get a more solid statistic, which supports or contradicts the results we have obtained.

An observation we made in the first experiment was with NTRULike lattices and that all the sieving algorithms (except for `nv`) were able to find short vectors. However, some of these were just multiples of the shortest vector. In Section 4.1 we presented a way to check this by controlling if there is a common factor in the coefficients that produce the short vector that is found. Hence implementing this control check by the G6K team could be done, making it easier to obtain a shortest vector instead of a multiple of it.

Another observation we have made from the experiments is that for the lattice type knapsack, we usually get better results using a reversed version of the strategy `PumpNJumpTour`. For the q-ary lattice we did not get the same results as with knapsack, and it did not really benefit from using a reversed version. It can seem from the results that which lattice reduction strategy we use can give different results depending on which lattice type we have. We are not sure why this is the case, hence it could be interesting to have a further look at this.

As mentioned in Section 4.2, we only did our experiments on a simple version of the implemented `PumpNJumpTour`. It could be interesting to implement a reversed version of the more advanced version of the algorithm as well and see if there is any improvement. The same experiments can be done as described in Section 4.2, but substitute `SimplePumpNJumpTour` with the more optimistic version of `PumpNJumpTour` and its reversed version.

At last, we also saw in the experiments in Section 4.3 that using `bdgl2` rather than `hk3` was faster on both knapsack and q-ary lattices when doing `PumpNJumpTour`. The experiment can be expanded to also include uniform lattices and NTRULike lattices to see if this is the case as well or if these behave differently.

There are multiple things that can be worked on further, and G6K has a lot of different parameters to play with. The suggestions from this section are based on our experiences with G6K and what we regard as the most relevant.

5.3 Conclusion

Lattice-based cryptography has come to stay. NIST has in its competition selected new standards for post-quantum secure cryptography, several of which are lattice-based. This means that it is going to be more important than ever to understand the lattice problems. Lattice-based cryptography is often based on the hardness of solving SVP, which today there are no known methods to solve efficiently, it is considered both NP-hard and quantum-safe.

The biggest threat to SVP is lattice reduction. We have seen different strategies and algorithms for lattice reduction, and today the current record for solving approximate SVP is dimension 180 according to Darmstadt's challenge. We also see from this challenge that the sieving approach using G6K is the best approach as of now. NIST has as mentioned in Section 1.4 now chosen CRYSTALS-KYBER (Kyber) to be a new quantum-safe standard. For Kyber there are three different security levels 1, 3, and 5. To find a private key in Kyber it is estimated that we need to solve SVP for lattices in dimension 375 (level 1), dimension 586 (level 2), or dimension 829 (level 5). As of now, we still have a long way to go until we are able to solve SVP for such high dimensions.

However, it is important that we continue the research on lattice reduction methods to make sure that our standardization is safe. As we have seen in this thesis there are sieving strategies that can be further improved depending on which lattice structure we are working with. However, there is still more to discover about lattices and lattice reduction.

Bibliography

[1] 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*, pages 601–610, 2001.

[2] M. R. Albrecht, L. Ducas, G. Herold, E. Kirshanova, E. W. Postlethwaite, and M. Stevens. The General Sieve Kernel and New Records in Lattice Reduction. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 717–746. Springer, 2019.

[3] A. Becker, N. Gama, and A. Joux. Speeding-up lattice sieving without increasing the memory, using sub-quadratic nearest neighbor search. Cryptology ePrint Archive, Paper 2015/522, 2015. https://eprint.iacr.org/2015/522.

[4] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 22(6), 1976.

[5] L. Ducas. Shortest vector from lattice sieving: a few dimensions for free. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part I*, pages 125–145. Springer, 2018.

[6] L. Ducas. Ask for an explanation of the "saturation condition", 2022. https://github.com/fplll/g6k/issues/99, Accessed on March 20th 2023.

[7] C. Gidney and M. Ekerå. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum*, 5:433, Apr. 2021.

[8] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.

[9] G. Herold and E. Kirshanova. Improved algorithms for the approximate k-list problem in euclidean norm. In *Public-Key Cryptography–PKC 2017: 20th IACR International Conference on Practice and Theory in Public-Key Cryptography, Amsterdam, The Netherlands, March 28-31, 2017, Proceedings, Part I*, pages 16–40. Springer, 2017.

[10] J. Hoffstein, J. Pipher, and J. H. Silverman. *An Introduction to Mathematical Cryptography*, volume 1. Springer, 2004.

[11] IBM. Factoring 15 with Shor's algorithm on a quantum computer. 2021. https://research.ibm.com/blog/factor-15-shors-algorithm, Accessed on March 31st 2023.

[12] IBM. IBM Unveils 400+ Qubit Quantum Processor and Next-Generation IBM Quantum System Two, 2022. https://newsroom.ibm.com/2022-11-09-IBM-Unveils-400-Qubit-Plus-Quantum-Processor-and-Next-Generatio
Accessed on March 31st 2023.

[13] J. Katz and Y. Lindell. *Introduction to modern cryptography*. CRC press, 2020.

[14] T. Laarhoven and A. Mariano. Progressive lattice sieving. In *Post-Quantum Cryptography: 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings*, pages 292–311. Springer, 2018.

[15] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische annalen*, 261:515–534, 1982.

[16] D. Micciancio and P. Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 1468–1480. SIAM, 2010.

[17] D. Micciancio and M. Walter. Fast lattice point enumeration with minimal overhead. In *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 276–294. SIAM, 2014.

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

[19] NIST. Nist cybersecurity. https://www.nist.gov/cybersecurity, Accessed on April 17th 2023.

[20] NIST. Post-quantum cryptography selected algorithms. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022, Accessed on April 17th 2023.

[21] NIST. Post-quantum cryptography standardization: Call for proposals. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/Call-for-Proposals, Accessed on April 17th 2023.

[22] NIST. Post-quantum cryptography standardization: Round 3 submissions. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions, Accessed on April 17th 2023.

[23] NIST. Post-quantum cryptography standardization: Round 4 submissions. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions, Accessed on April 17th 2023.

[24] NIST. Pqc standardization process: Announcing four candidates to be standardized, plus fourth round candidates. https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4, Accessed on May 24th 2023.

[25] F. A. P. Petitcolas. *Kerckhoffs' Principle*, page 675. Springer US, Boston, MA, 2011.

[26] H. Raddum. Why doesn't triple sieve work right out of the box?, 2023. https://github.com/fplll/g6k/issues/111, Accessed on May 3rd 2023.

[27] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[28] C.-P. Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical computer science*, 53(2-3):201–224, 1987.

[29] C. E. Shannon. Communication theory of secrecy systems. *The Bell system technical journal*, 28(4):656–715, 1949.

[30] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[31] S. Singh. *The code book*. Fourth Estate, 2000.

[32] The Cthaeh. Cryptography during world war i. *Probabilistic World*, 2020.

[33] The FPLLL development team. fpylll, a Python wrapper for the fplll lattice reduction library, Version: 0.5.9. Available at https://github.com/fplll/fpylll, 2023.

[34] The G6K development team. The General Sieve Kernel (G6K), Version: 0.1.2. Available at https://github.com/fplll/g6k.

[35] The SVP Challenge. Svp challenge hall of fame. https://www.latticechallenge.org/svp-challenge/halloffame.php Accessed on May 21st 2023.

Appendix

Tables from experiments in Section 4.1

The tables contain results from the experiments performed in Section 4.1. Each table shows which values we gathered for each lattice type. The data in the tables consist of the columns: dimension used, the time it took to sieve in seconds, The norm of the shortest vector found, which sieving algorithm was used, and how well it performs compared to the other algorithms based on time (time/best time).

Comment for the norm column: as the timings are averages of 10 different runnings on 10 different lattices, the norms are just from the first lattice used. As it makes no sense to take the average norm as we are looking at different lattices, hence this decision.

In the tables we also see values for all the different bdgl-algorithms, even though we are only looking at `bdgl1` and `bdgl2` in the experiments. We have them just to check that they behave the way they are supposed to. The bdgl's that are not used in the plots are marked with a "*[X]", where X is the `blocks` it should represent for the current dimension.

Dimension	Time in sec	Norm*	Algorithm	Time / Best time
40	0.135887	1305212710240913542656	nv	1.70208
40	0.120323	1305212710240913542656	bgj1	1.50713
40	**0.079836**	1305212710240913542656	gauss	1.0
40	0.204543	1305212710240913542656	bdgl*[1]	2.56204
40	0.207446	1305212710240913542656	bdgl1	2.5984
40	0.207079	1305212710240913542656	bdgl2*[1]	2.5938
40	0.205544	1305212710240913542656	bdgl3*[1]	2.57458
50	1.710862	1703685500853410666240	nv	1.76106
50	1.41706	1703685500853410666240	bgj1	1.45864
50	**0.971497**	1703685500853410666240	gauss	1.0
50	2.000753	1703685500853410666240	bdgl*[1]	2.05945
50	1.998967	1703685500853410666240	bdgl1	2.05762
50	2.022229	1703685500853410666240	bdgl2*[1]	2.08156
50	2.024907	1703685500853410666240	bdgl3*[1]	2.08432
60	31.918268	1828904046391914733952	nv	1.63138
60	21.173534	1828904046391914733952	bgj1	1.08221
60	**19.565151**	1828904046391914733952	gauss	1.0
60	19.949617	1828904046391914733952	bdgl*[2]	1.01965
60	23.176816	1828904046391914733952	bdgl1	1.1846
60	20.026027	1828904046391914733952	bdgl2	1.02356
60	20.186024	1828904046391914733952	bdgl3*[2]	1.03173
70	646.171706	2277394420634882318336	nv	3.10605
70	302.909659	2277394420634882318336	bgj1	1.45604
70	429.202474	2277394420634882318336	gauss	2.06311
70	**208.036503**	2277394420634882318336	bdgl*[2]	1.0
70	281.892827	2277394420634882318336	bdgl1	1.35502
70	208.686594	2277394420634882318336	bdgl2	1.00312
70	208.739093	2277394420634882318336	bdgl3*[2]	1.00338
80	16371.704859	2548073717437254073088	nv	6.90728
80	4768.727656	2548073717437254073088	bgj1	2.01194
80	14649.176769	2548073717437254073088	gauss	6.18054
80	2398.577419	2548073717437254073088	bdgl*[2]	1.01197
80	3827.53684	2548073717437254073088	bdgl1	1.61485
80	**2370.208476**	2548073717437254073088	bdgl2	1.0
80	2371.3622	2548073717437254073088	bdgl3*[2]	1.00049

*Table 1: q-ary lattice results, * represents that this algorithm is not in the Figure, while [X] tells us what the* `blocks` *parameter in* `bdgl` *value is.*

Dimension	Time in sec	Norm	Algorithm	Time / Best time
40	0.003335	38709266093397243744256	nv	x
40	0.112188	323208000	bgj1	1.37799
40	**0.081414**	20	gauss	1.0
40	0.185298	112500	bdgl*[1]	2.276
40	0.185121	1280180	bdgl1	2.27382
40	0.186773	119658320	bdgl2*[1]	2.29412
40	0.184663	100820	bdgl3*[1]	2.2682
50	0.017719	43358728633507495776256	nv	x
50	1.32439	2656900	bgj1	1.8567
50	**0.713302**	25	gauss	1.0
50	1.819009	2025	bdgl*[1]	2.55012
50	1.859778	6400	bdgl1	2.60728
50	1.824911	9025	bdgl2*[1]	2.5584
50	1.803038	12100	bdgl3*[1]	2.52773
60	0.088429	40900873858977346433024	nv	x
60	18.164793	270	bgj1	1.59298
60	**11.402991**	30	gauss	1.0
60	18.016136	30	bdgl*[2]	1.57995
60	20.045811	30	bdgl1	1.75794
60	18.42987	30	bdgl2	1.61623
60	18.413868	30	bdgl3*[2]	1.61483
70	0.473175	32556079570709546491904	nv	x
70	263.717643	35	bgj1	1.08524
70	246.205991	35	gauss	1.01318
70	275.025263	35	bdgl*[2]	1.13177
70	**243.003946**	35	bdgl1	1.0
70	272.339978	35	bdgl2	1.12072
70	369.153409	35	bdgl3*[2]	1.51913
80	2.563437	57429056118738805788672	nv	x
80	3799.039622	40	bgj1	1.33525
80	7333.943162	40	gauss	2.57766
80	3422.294221	40	bdgl*[2]	1.20284
80	3148.562514	40	bdgl1	1.10663
80	**2845.189109**	40	bdgl2	1.0
80	2976.25058	40	bdgl3*[2]	1.04606

*Table 2: NTRULike lattice results, * represents that this algorithm is not in the Figure, while [X] tells us what the* `blocks` *parameter in* `bdgl` *value is.*

Dimension	Time in sec	Norm	Algorithm	Time / Best time
40	0.126994	30	nv	1.65092
40	0.123092	30	bgj1	1.6002
40	**0.076923**	30	gauss	1.0
40	0.206216	30	bdgl*[1]	2.68081
40	0.20912	30	bdgl1	2.71856
40	0.208083	30	bdgl2*[1]	2.70508
40	0.206996	30	bdgl3*[1]	2.69095
50	1.696395	25	nv	1.74982
50	1.49367	25	bgj1	1.54071
50	**0.969469**	25	gauss	1.0
50	2.041456	25	bdgl*[1]	2.10575
50	2.060941	25	bdgl1	2.12585
50	2.068436	25	bdgl2*[1]	2.13358
50	2.068436	25	bdgl3*[1]	2.13358
60	32.477311	21	nv	1.6696
60	22.863862	21	bgj1	1.17539
60	**19.452196**	21	gauss	1.0
60	20.71266	21	bdgl*[2]	1.0648
60	23.822738	21	bdgl1	1.22468
60	20.380401	21	bdgl2	1.04772
60	20.524032	21	bdgl3*[2]	1.0551
70	619.367999	18	nv	2.93865
70	304.196811	18	bgj1	1.44329
70	399.234018	18	gauss	1.89421
70	**210.765891**	18	bdgl*[2]	1.0
70	283.760023	18	bdgl1	1.34633
70	211.284208	18	bdgl2	1.00246
70	**210.766525**	18	bdgl3*[2]	1.0
80	14939.570141	16	nv	6.63313
80	4357.922003	16	bgj1	1.93491
80	12164.853384	16	gauss	5.40116
80	2270.150535	16	bdgl*[2]	1.00794
80	3622.917927	16	bdgl1	1.60857
80	**2252.265945**	16	bdgl2	1.0
80	2259.903428	16	bdgl3*[2]	1.00339

*Table 3: Knapsack lattice results, * represents that this algorithm is not in the Figure, while [X] tells us what the* `blocks` *parameter in* `bdgl` *value is.*

Dimension	Time in sec	Norm	Algorithm	Time / Best time
40	0.132172	3517333461444	nv	1.70897
40	0.121733	3517333461444	bgj1	1.574
40	**0.07734**	3517333461444	gauss	1.0
40	0.207219	3517333461444	bdgl*[1]	2.67933
40	0.206285	3517333461444	bdgl1	2.66725
40	0.205643	3550463705195	bdgl2*[1]	2.65895
40	0.205843	3517333461444	bdgl3*[1]	2.66153
50	1.701509	5130513218068	nv	1.71199
50	1.404872	5130513218068	bgj1	1.41353
50	**0.993875**	5130513218068	gauss	1.0
50	2.062319	5130513218068	bdgl*[1]	2.07503
50	2.064121	5130513218068	bdgl1	2.07684
50	2.063109	5130513218068	bdgl2*[1]	2.07582
50	2.078737	5130513218068	bdgl3*[1]	2.09155
60	32.157044	5783564359637	nv	1.61498
60	21.712963	5783564359637	bgj1	1.09046
60	**19.911691**	5783564359637	gauss	1.0
60	20.625894	5783564359637	bdgl*[2]	1.03587
60	24.142661	5783564359637	bdgl1	1.21249
60	20.870412	5783564359637	bdgl2	1.04815
60	20.783174	5783564359637	bdgl3*[2]	1.04377
70	652.230525	6394343272663	nv	3.04018
70	303.077991	6394343272663	bgj1	1.41271
70	431.111389	6394343272663	gauss	2.0095
70	**214.537143**	6394343272663	bdgl*[2]	1.0
70	290.718633	6394343272663	bdgl1	1.3551
70	215.166757	6394343272663	bdgl2	1.00293
70	215.014877	6394343272663	bdgl3*[2]	1.00223
80	16696.506639	8650650517322	nv	6.74841
80	4939.995864	8650650517322	bgj1	1.99665
80	13981.003844	8650650517322	gauss	5.65085
80	**2474.140249**	8650650517322	bdgl*[2]	1.0
80	3991.259828	8650650517322	bdgl1	1.61319
80	2483.227435	8650650517322	bdgl2	1.00367
80	2480.537546	8650650517322	bdgl3*[2]	1.00259

*Table 4: Uniform lattice results, * represents that this algorithm is not in the Figure, while [X] tells us what the* `blocks` *parameter in* `bdgl` *value is.*

Tables from experiments in Section 4.3

In this section, we find the data gathered for the tables used in Section 4.3. The table contains which algorithm we use, the norms found, time used in seconds, and which lattice this data is from. For the q-ary latticed in Table 6 we are not looking at all the norms but the last vector norm found as the numbers are so big it became very messy to show them all.

Algorithm	Norms	Time	Seed
`SimplePumpNJumpTour - default`	[18, 18, 18, 18, 18, 18, 18, 18, 18, 18]	486.9418	0x1337
	[20, 19, 19, 17, 17, 17, 17, 17, 17, 17]	495.8485	0x1228
	[19, 19, 16, 16, 16, 16, 16, 16, 16, 16]	485.9951	0x1234
	[17, 17, 17, 17, 17, 17, 17, 17, 17, 17]	496.479	0x5678
	[19, 19, 19, 19, 19, 19, 19, 17, 17, 17]	499.9388	0x9534
	[18, 18, 18, 18, 18, 18, 18, 18, 18, 18]	497.0467	0x3564
	[19, 19, 19, 19, 18, 18, 18, 18, 18, 18]	487.5777	0x1278
	[21, 19, 19, 19, 19, 19, 15, 15, 15, 15]	496.9351	0xabcd
	[18, 18, 18, 18, 18, 18, 18, 18, 18, 18]	492.6572	0xef22
	[23, 20, 20, 19, 19, 19, 15, 15, 15, 15]	499.4689	0x4444
`SimplePumpNJumpTour - modified`	[21, 18, 18, 18, 16, 16, 16, 16, 16, 16]	319.0167	0x1337
	[18, 18, 17, 17, 17, 17, 17, 17, 17, 17]	322.0889	0x1228
	[20, 18, 18, 18, 17, 17, 17, 16, 16, 16]	330.5361	0x1234
	[18, 18, 18, 18, 18, 18, 18, 18, 18, 18]	324.257	0x5678
	[18, 18, 18, 18, 18, 18, 17, 17, 17, 17]	330.2325	0x9534
	[18, 15, 15, 15, 15, 15, 15, 15, 15, 15]	325.262	0x3564
	[21, 19, 19, 17, 17, 17, 17, 17, 17, 17]	319.2819	0x1278
	[18, 18, 18, 18, 18, 18, 18, 18, 18, 18]	331.234	0xabcd
	[20, 18, 18, 18, 18, 18, 18, 18, 18, 18]	327.0199	0xef22
	[20, 20, 19, 19, 19, 19, 19, 19, 19, 18]	322.4128	0x4444
`ReversedPumpNJumpTour - default`	[18, 18, 18, 16, 16, 16, 16, 16, 16, 16]	486.1472	0x1337
	[21, 19, 18, 18, 18, 16, 16, 16, 16, 16]	491.9993	0x1228
	[19, 19, 19, 19, 18, 18, 18, 18, 18, 18]	482.4157	0x1234
	[20, 19, 17, 17, 17, 17, 17, 17, 17, 17]	495.0013	0x5678
	[19, 18, 18, 18, 17, 17, 17, 17, 17, 17]	492.2047	0x9534
	[17, 17, 17, 17, 17, 17, 17, 17, 17, 17]	490.4515	0x3564
	[18, 18, 16, 16, 16, 16, 16, 16, 16, 16]	487.0034	0x1278
	[19, 18, 18, 18, 18, 18, 18, 18, 18, 18]	492.4457	0xabcd
	[17, 17, 17, 17, 17, 17, 17, 17, 17, 17]	495.8065	0xef22
	[19, 19, 18, 16, 16, 16, 16, 16, 16, 16]	493.3296	0x4444
`ReversedPumpNJumpTour - modified`	[18, 18, 18, 18, 18, 18, 18, 16, 16, 16]	339.2943	0x1337
	[19, 18, 18, 18, 17, 17, 17, 17, 17, 16]	335.841	0x1228
	[19, 19, 19, 18, 18, 18, 18, 18, 18, 18]	331.617	0x1234
	[20, 17, 17, 17, 17, 17, 17, 17, 17, 17]	332.56	0x5678
	[19, 19, 19, 15, 15, 15, 15, 15, 15, 15]	328.2032	0x9534
	[18, 17, 17, 17, 17, 17, 17, 17, 17, 17]	338.4838	0x3564
	[19, 19, 19, 19, 19, 19, 17, 17, 17, 17]	334.4255	0x1278
	[20, 20, 20, 19, 19, 18, 17, 17, 17, 17]	336.0598	0xabcd
	[17, 17, 17, 17, 17, 17, 17, 17, 17, 17]	337.041	0xef22
	[19, 18, 18, 18, 18, 18, 18, 18, 18, 18]	327.1178	0x4444

Table 5: Data from knapsack experiment on PumpNJumpTour with tweaked Pump. Blocksize=70, jump=1

Algorithm	Best Norm	Time	Seed
`SimplePumpNJumpTour - default`	13655094479723986248704	988.8556	0x1337
	23413116640586056425472	984.3853	0x1228
	28497850362935412813824	984.6697	0x1234
	20191802170637059235840	992.2837	0x5678
	1762382690775110354048	986.1828	0x9534
	21945378440289460414464	988.1728	0x3564
	5293884471580134394368	997.8186	0x1278
	2797323984533100586752	987.5037	0xabcd
	2593666870851111853056	981.7224	0xef22
	22990580541134193518592	991.0914	0x4444
`SimplePumpNJumpTour - modified`	12538353199977033611264	920.2901	0x1337
	23961253393596297871360	924.2941	0x1228
	31510163260253967740928	921.6578	0x1234
	21278828717220612429824	920.015	0x5678
	1881462440033416643968	926.906	0x9534
	21385969068148730458112	920.6825	0x3564
	5464396169943912053248	923.8766	0x1278
	2569007559439324906752	926.8994	0xabcd
	2686955287806472258816	926.7544	0xef22
	21927429444992930564096	927.0614	0x4444
`ReversedPumpNJumpTour - default`	18498465397367546359808	968.0108	0x1337
	38877923156327789166592	961.749	0x1228
	44036214027853265133568	957.6766	0x1234
	31904141503088612784128	964.4213	0x5678
	2672531588111411647232	959.3421	0x9534
	28052617011961920509952	953.6359	0x3564
	6435448519767777082368	957.2856	0x1278
	3912991887284951251968	966.3937	0xabcd
	3441602921643626237952	970.3276	0xef22
	36039631388255109496832	966.3925	0x4444
`ReversedPumpNJumpTour - modified`	20493500782456373278720	952.5273	0x1337
	31419905992969788413952	945.4036	0x1228
	46151856793271043928064	952.3993	0x1234
	31913238063305925199872	948.4451	0x5678
	3022879437899461450496	946.7952	0x9534
	30017839825830861864960	945.6137	0x3564
	7718746423535312714752	951.1008	0x1278
	4040830631419387191808	949.7025	0xabcd
	4100424490332026724352	955.5973,	0xef22
	38409153346551253598208	953.3168	0x4444

Table 6: Data from q-ary experiment on PumpNJumpTour with tweaked Pump. Blocksize=70, jump=1