Efficiency Analysis of Some Cryptographic Primitives

Egyetemi doktori (PhD) értekezés

A szerző neve: Major Sándor Roland

A témavezető neve: Dr. Herendi Tamás

Debreceni Egyetem
Természettudományi és Informatikai Doktori Tanács
Informatikai Tudományok Doktori Iskola
Debrecen, 2024

Ezen értekezést a Debreceni Egyetem Természettudományi és Informatikai Doktori Tanács Informatikai Tudományok Doktori Iskola *Elméleti számítástudomány, adatvédelem és kriptográfia* programja keretében készítettem a Debreceni Egyetem műszaki tudományokban doktori (PhD) fokozatának elnyerése céljából.
Nyilatkozom arról, hogy a tézisekben leírt eredmények nem képezik más PhD disszertáció részét.
Debrecen, 2024

.....
a jelölt aláírása

Tanúsítom, hogy Major Sándor Roland doktorjelölt 2011-2014 között a fent megnevezett Doktori Iskola *Elméleti számítástudomány, adatvédelem és kriptográfia* programjának keretében irányításommal végezte munkáját. Az értekezésben foglalt eredményekhez a jelölt önálló alkotó tevékenységével meghatározóan hozzájárult. Nyilatkozom továbbá arról, hogy a tézisekben leírt eredmények nem képezik más PhD disszertáció részét.
Az értekezés elfogadását javasolom.
Debrecen, 2024

.....
a témavezető aláírása

Efficiency Analysis of Some Cryptographic Primitives

Értekezés a doktori (Ph.D.) fokozat megszerzése érdekében
az informatika tudományágban

Írta: Major Sándor Roland

Készült a Debreceni Egyetem Informatikai Tudományok doktori iskolája
(*Elméleti számítástudomány, adatvédelem és kriptográfia* programja) keretében

Témavezető: Dr. Herendi Tamás

Az értekezés bírálói:

Dr. .....
Dr. .....

A bírálóbizottság:

elnök: Dr. .....
tagok: Dr. .....
Dr. .....
Dr. .....
Dr. .....

Az értekezés védésének időpontja: 20 .....

Contents

Chapter 1

Introduction

Cryptographic primitives are the low-level concepts and algorithms that are used as the basic building blocks of more complex, higher level cryptographic algorithms and protocols. Because of the wide variety of ideas used in cryptography, the primitives show great diversity as well: hash functions, symmetric and asymmetric keys, digital signatures, block and stream ciphers, random number generators, linear codes, and many more are used in modern cryptographic systems.

This thesis focuses on two of these primitives: random number generators, and linear codes.

Pseudorandom numbers have many practical applications. In cryptography specifically, they are used for purposes such as key generation, stream ciphers and asymmetric cryptosystems [1]. More broadly, they are also used for simulations, Monte-Carlo methods [2], and many others. Depending on the application, pseudorandom number sequences need to fulfill different requirements, such as the distribution of its elements, the period length of the sequence, computation speed of the next element, and unpredictability. A detailed examination of generating random sequences and using statistical tests to check their properties can be found in [3].

Part I of the thesis is based on an algorithm used to construct pseudorandom number generators described in [4], developed by Tamás Herendi. The algorithm can be used to create linear recurrence sequences with uniform distribution modulo powers of 2, with theoretically arbitrarily large period lengths. The elements of the sequences can likewise be arbitrarily large. Computing new elements is simple, using linear feedback shift registers. Although unpre-

dictability does not hold for the base sequences created, the elements can be transformed in ways to make this requirement hold.

Chapter 2 introduces the basic concepts used throughout this part of the thesis, such as finite fields, polynomials over finite fields, and irreducibility.

Chapter 3 describes the algorithm in detail, and identifies the most computationally expensive steps. As input, the algorithm requires an irreducible polynomial of high degree and order over $GF(2)$. The chapter also discusses methods of creating such a polynomial, including test results of the methods in practice.

Chapter 4 presents multiple implementations of the previously identified computationally expensive steps, using both hardware and software platforms. The chapter compares the performance of these implementations to find the most efficient solutions to be used in practice.

Chapter 5 details tests carried out to measure the statistical properties of the pseudorandom number sequences created using the algorithm and tools determined in the previous chapters.

The theoretical design of the presented results was developed during joint research between Tamás Herendi and the author of the thesis. The application development, implementation and analysis are the results of the author of the thesis. The relevant publications by the author for this part of the thesis are [5], [6] and [7].

Linear codes were developed as a way to encode information in a way that is capable of identifying and correcting some number of errors that can happen during transmission. In cryptography, they can be used for constructing algebraic manipulation detection codes [8], or in algorithms such as the McEliece [9] and Niederreiter [10] cryptosystems.

Part II of the thesis is concerned with the problem of constructing linear codes with given, close to arbitrary parameters. A software package named Torch, created by the author, is presented that is extensible and reconfigurable to allow for a wide variety of search conditions and support experimentation by incorporating new research results. The software can be used for tasks such as classification, finding specific existing linear codes, or searching for currently unknown lienar codes.

Chapter 6 introduces the concepts relevant to this part of thesis, such as linear codes, generator matrices, permutation equivalence, and self-dual codes.

Chapter 7 describes the practical usage, basic architecture and dependencies of the Torch package.

Chapter 8 presents the basic search algorithms available in the package, along with the implementation details necessary to make the solutions viable

in practice. The algorithms included in Torch are a basic depth-first search, a search based on finding cliques in a graph, and a hill-climbing and best-first heuristic searches. The chapter also gives practical results of using the presented algorithms.

Chapter 9 details elements of the Torch framework that can be used to fine-tune and extend the behaviour of the searches. These elements include a large number of option variables, and condition objects that can be added to the previously described algorithms.

Chapter 10 presents a number of configurations that are currently available in Torch. Each configuration is a collection of modules and options built to provide efficient solutions to specific search tasks. Configuration are created to be extensible and to allow combinations of multiple configurations to be used together in one search.

The theoretical design of the presented results was developed during joint research between Carolin Hannusch and the author of the thesis. The software development, algorithmic and theoretical solutions used in the presented configurations, implementation and analysis are the results of the author of the thesis. The relevant publications by the author for this part of the thesis are [11] and [12].

The thesis ends with an Appendix that contains tables of test results, computational results and pseudocodes of presented algorithms that would be too long to include in the main text.

Part I

Random Number Generation

Chapter 2

Basic concepts

In this chapter we will present the basic concepts and notations necessary to understand the topic of Part I of the thesis. These concepts are mostly related to the topic of finite fields, polynomials over finite fields, and the irreducibility of polynomials.

2.1 Finite Fields

The origin of the theory of finite fields dates back to at least the 17th century with related results from Pierre de Fermat, with later contributions from mathematicians such as Leonhard Euler, Joseph-Louis Lagrange, Adrien-Marie Legendre and Carl Friedrich Gauss. The modern concept of finite fields as understood today was introduced in the works of Evariste Galois. In the 20th century, interest was renewed in this area because of its applications in combinatorics, coding theory and the study of circuits. The following definitions, which will be used throughout the thesis, are found in [13]:

- Algebraic structures such as *groups*, *rings*, *fields* and *finite fields*.
- The *order* and *characteristic* of a finite field.
- *Subfields, extension fields, proper subfields*, and *algebraic extensions* of a field.

Of special interest to us is the finite field $GF(2)$, which contains only two elements, 0 and 1. Addition in $GF(2)$ is equivalent to the logical XOR operation,

or addition in $\mathbb{N}$, mod 2. Multiplication in $GF(2)$ is equivalent to the logical AND operation, or multiplication in $\mathbb{N}$. Note that in $GF(2)$, $a - b = a + b$ for all $a, b \in \{0, 1\}$.

2.2 Polynomials over Finite Fields

The results presented in Part I of the thesis use the following definitions related to polynomials over finite fields, found in [13]:

- *Univariate polynomials* over a field, and *monic* polynomials.
- The *degree, leading coefficient* and *constant term* of a polynomial.
- The *polynomial ring* over a field.
- The *splitting field* of a polynomial.
- *Divisibility, divisibility with remainder*, and *greatest common divisor (GCD)* of polynomials.
- *Order* of polynomials. The order of the polynomial used as input will play an important role in determining the period length of the pseudorandom number sequence generated by the algorithm presented in chapter 3.

Definition 2.2.1 (Order of polynomials)**.** Let $F \in \mathbb{F}[x]$ be a non-zero polynomial. If the constant term of F is not 0, then the *order* of F, denoted as $ord(F)$, is the smallest positive integer e for which $F|x^e - 1$.

If the constant term of f is 0, then F can be written as $F(x) = x^h G(x)$, with uniquely determined $h \in \mathbb{N}$ and $G \in \mathbb{F}[x]$ such that the constant term of G is not 0. Then, the order of F is defined to be $ord(F) = ord(G)$.

If we can calculate the order of irreducible polynomials, then we can find the order of any polynomial that we can factorize. In practice however, finding the order of an irreducible polynomial with a very large degree is a difficult computational task.

2.3 Irreducible Polynomials

Irreducible polynomials will play a central role in the methods discussed in later chapters. Aside from defining irreducible polynomials and some of their properties, we need to be able to test polynomials for irreducibility, and be able to generate such polynomials.

Definition 2.3.1 (Irreducible polynomial)**.** A polynomial $P \in \mathbb{F}[x]$ is *irreducible over* $\mathbb{F}$, or *irreducible in* $\mathbb{F}[x]$ if $deg(P) > 0$ and $P = BC$, $B, C \in \mathbb{F}[x]$ implies B or C is a constant polynomial.

If $F \in \mathbb{F}_q[x]$ is an irreducible polynomial over $\mathbb{F}_q$, and $deg(F) = n$, then $ord(F)|q^n - 1$. This fact will be significant when deciding the degree of the polynomial we want to use as input for the algorithm to generate pseudorandom number sequences.

Chapter 3

Creating pseudorandom number sequences

In this chapter, an algorithm developed by Tamás Herendi is described that is capable of creating pseudorandom number sequences with uniform distribution and arbitrarily large period length. Two approaches are discussed: one where the most computationally expensive step is matrix exponentiation, and one where it is computing polynomial modulo.

The algorithm requires an irreducible polynomial over $GF(2)$ with high degree and order as an input. Tests for deciding the irreducibility of a polynomial are described in section 3.2. Two methods of constructing such polynomials are presented in section 3.3. Empirical test results of their comparison can be found in section 3.4.

3.1 Algorithm

The algorithm described in [4] constructs linear recurring sequences to generate pseudo-random numbers. The main advantage of the constructed sequences is uniform distribution of elements with an extremely large (theoretically arbitrary) period length. The elements of the sequence can also be arbitrarily large, not just single bits. Because of the basic properties of linear recurring sequences, unpredictability of forthcoming elements does not hold. The sequences are suitable for any practical application where unpredictability is not required.

The sequences are based on the following theorem.

Theorem 3.1.1. Let $Q \in \mathbb{Z}[x]$ be a monic polynomial, $deg(Q) = k$, such that its reduction modulo 2 is irreducible. Let $P \in \mathbb{Z}[x]$ be monic satisfying

$$P(x) \equiv (x^2 - 1)Q(x) \ mod\ 2 \tag{3.1}$$

Define the following four polynomials:

$$\begin{aligned} P_1(x) &= P(x), \\ P_2(x) &= P(x) - 2, \\ P_3(x) &= P(x) - 2x, \\ P_4(x) &= P(x) - 2x - 2 \end{aligned}$$

For each polynomial P_i, let $u^{(i)}$ be a linear recurring sequence such that the minimal period length of $u^{(i)}$ modulo 2 is $2ord(Q)$, where $ord(Q)$ is the order of Q in $F_2[x]$. Then at least one $u^{(i)}$ is uniformly distributed modulo 2^s with a period length $2^s ord(Q)$ for any $s \in \mathbb{N}$.

Two approaches are discussed for finding the desired linear recurring sequence.

3.1.1 Matrix exponentiation approach

The most computationally expensive step of this approach is matrix exponentiation. An older, much smaller implementation of this approach can be found in [5]. Let

$$x^d - a_{d-1}x^{d-1} - \ldots - a_0 = P_i(x)$$

Then the integer d is the order of $u^{(i)}$, and the companion matrix is

$$M_{(i)} = \begin{pmatrix} 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \\ a_0 & a_1 & \cdots & a_{d-2} & a_{d-1} \end{pmatrix}$$

To find the sequence with uniform distribution, we need to compute the matrix $M(i)^{2^{d+1}-2}\ mod\ 4$. If $M(i)^{2^{d+1}-2}\ mod\ 4$ equals the identity matrix, which means the period length of $u^{(i)}$ is $2^{d+1} - 2$, then it is not desired sequence.

The computational cost of this approach grows very quickly with d. Using fast exponentiation, we need to perform $d + 1$ matrix multiplications on $d \times d$ sized matrices.

3.1.2 Polynomial modulo approach

The most computationally expensive step of this approach is polynomial modulo. Let P be a polynomial over a finite field and $P(0) \neq 0$. Then, $ord(P) = e$ is the order of P, where e is the smallest integer for which $P(x)|x^e - 1$. If P is irreducible over a finite field with q elements, then $ord(P)|q^k - 1$.

Let ρ_i be $ord(P_i)$ over $F_2[x]$, $i \in \{1, 2, 3, 4\}$. For this approach, to find the sequence with uniform distribution, we need to find P_i for which

$$P_i(x) \not| x^{\rho_i} - 1 \ mod\ 4$$

In practice, this requires computing

$$R_i(x) = x^{\rho_i/2} \ mod\ P_i(x) \ (mod\ 2)$$

Then, we need to check

$$1 \not\equiv R_i(x)^2 \ mod\ P_i(x) \ (mod\ 4)$$

Except for the last step, which needs to performed over $GF(4)$, the computation can be performed over $GF(2)$. This requires $\rho_i/2$ squaring and polynomial modulo operations.

In practice, the polynomial modulo approach is significantly more efficient than the matrix exponentiation approach. Note that the space requirement of the companion matrix $M_{(i)}$ used in the matrix exponentiation approach is quadratic in the size of the polynomial P_i, which can become impractical when $deg(P_i)$ is high. Another advantage is that the polynomial modulo operations required in the second approach can be implemented with $O(n^2)$ time compexity, which is lower than even the best known algorithms for matrix multiplication.

The following chapters of the thesis present in detail the implementation of the computationally expensive steps of the above algorithm: matrix multiplication over $GF(4)$, and polynomial modulo over $GF(2)$.

3.2 Irreducibility Testing

Multiple methods of testing a given polynomial's irreducibility are known. As previously mentioned, a polynomial $P \in F[x]$ of degree n is irreducible if it has no nontrivial factors over $\mathbb{F}$. That is, $P(x) = P_1(x)P_2(x)$ can not hold if

$deg(P_1), deg(P_2) > 0$. The natural way to prove a polynomial's irreducibility is therefore to factor it and show that no such factors can be found.

An algorithm for factoring a polynomial over a finite field was by Berkelamp [14]. It is a deterministic algorithm that requires a square-free polynomial, and is well suited for cases where the cardinality of the finite field is small. Later, the Cantor-Zassenhaus algorithm [15] provided a practical solution even for polynomials over large finite fields. This algorithm is probabilistic in nature. A detailed description of both methods can be found in [16].

Rabin's test [17] provides an algorithm based on two conditions. A polynomial F of degree n over $GF(2)$ is irreducible if and only if:

1. $F(x)|x^{2^n} - x$
2. $\forall\ p_i\ GCD(x^{2^{n/p_i}}, F(x)) = 1$

Where p_i are the prime factors of n. The test computes all $x^{2^{n/p_i}}\ mod\ F(x)$ polynomials using repeated squaring and polynomial modulo operations, then uses polynomial GCD to check condition 2. A commonly used variant of Rabin's test was proposed in [18].

Ben-Or's test [19] modifies this approach by computing $gcd(x^{2^i}\ mod\ F, F)$ for every $i \in \{1..\frac{n}{2}\}$. In practice, this improves average performance when testing random polynomials. A randomly selected polynomial is much more likely to have factors of small degrees than be the product of only large-degree factors. Since Ben-Or's test checks for factors of small degrees first, these polynomials are very quickly eliminated. A comparison between the performance of Rabin's test and Ben-Or's test can be found in [18].

Victor Shoup, the author of the NTL software library used in the performance test in a later chapter, also published a deterministic irreduciblity test in [20] and a probabilistic algorithm is [21].

3.3 Generating Irreducible Polynomials

In this section, two methods for creating irreducible polynomials are presented. These methods are used for creating the polynomials required for the pseudo-random number generators discussed earlier in this chapter.

The first method, developed by the author and Tamás Herendi, is based on making sure that a randomly generated polynomial does not have any small factors. The second method uses a known tranformation named *Q-transform* to create infinite families of irreducible polynomials. The suitability of the

polynomials created by these algorithms to be used in pseudorandom number generation can be found in [7] and is discussed in section 3.4.

The algorithm are specifically used to create irreducible polynomials over $GF(2)$, since the number generators described previously require polynomials over this field.

3.3.1 Small factor excluding method

The number of monic irreducible polynomials of degree n over a finite field is given by Gauss's formula. We are interested in irreducibles over $\mathbb{F}_2$:

$$G(n) = \frac{1}{n} \sum_{d|n} \mu(n/d) 2^d$$

where $\mu(x)$ is the Möbius function [13]:

$$\mu(n) = \begin{cases} 1 & if\ n = 1 \\ (-1)^k & if\ n\ is\ the\ product\ of\ k\ distinct\ primes \\ 0 & if\ n\ is\ divisible\ by\ the\ square\ of\ a\ prime \end{cases}$$

As a trivial upper limit, we can use

$$G(n) < \frac{1}{n} \sum_{i=1}^{n} 2^i = \frac{2^{n+1} - 1}{n}.$$

The probability that a randomly generated degree n polynomial is irreducible, supposing that our pick has uniform distribution, can be estimated as

$$P(random\ deg(n)\ polynomial\ is\ irreducible) = \frac{G(n)}{2^n} < \frac{2^{n+1} - 1}{n 2^n} < \frac{2}{n}$$

Let $I(x)$ be an irreducible polynomial over $\mathbb{F}_2$, $deg(I) = n$. If $Q(x)$, $deg(Q) = m$, is divisible by $I(x)$, then it can be written as $Q(x) = I(x)R(x)$ where $deg(R) = m - n$. This means that the number of degree m polynomials that can be divided by a given degree n irreducible polynomial is equal to the number of degree $m - n$ polynomials, i. e. 2^{m-n}. If $S(x)$ is a randomly generated degree m polynomial, picked with uniform distribution, then

$$P(I(x)|S(x)) = \frac{2^{m-n}}{2^m} = \frac{1}{2^n},\ and\ P(I(x)\ \not|S(x)) = 1 - \frac{1}{2^n}$$

Note that this probability is independent of m.

Let $I_1(x), ..., I_n(x)$ be irreducible polynomials. Let $S(x)$ be a randomly generated polynomial, picked with uniform distribution, and $\forall i \ deg(S) \geq deg(I_i)$. Then

$$P(\forall i I_i(x) \not| S(x)) = \prod_{i=1}^{n} 1 - \frac{1}{2^{deg(I_i)}}$$

We use this observation when searching for a large, irreducible polynomial over $\mathbb{F}_2$. The general idea is that we first generate a candidate polynomial that is guaranteed to not have any small factors, then check for irreducibility using one of the testing methods described in subsection 3.2. Making sure that the candidate polynomial doesn't have small factors multiplies the chance of it being irreducible by a constant factor over randomly choosing a polynomial with uniform distribution. For very large degrees, this method is not practical, but it does provide a significant upgrade for the range of degrees we are interested in for the purposes of this test, which is between $2^{16} - 2^{20}$.

Table 3.1 in the Appendix shows what proportion of randomly chosen polynomials S do not have a factor of degree i or lower.

Let n be the degree of the irreducible polynomial over $GF(2)$ we wish to generate. We will create candidate polynomials $R(x)$, $deg(R(x)) = n$, which will be tested for irreducibility. The steps of creating the candidate polynomials are as follows:

1. Let M_i be the set of all irreducible polynomials over $GF(2)$ with degree at most i. Let the polynomials $T_i(x)$ be the following:

   $$T_i(x) = \prod_{P \in M_i} P$$

   In other words, $T_i(x)$ is the product of all irreducible polynomials over $GF(2)$ with degree at most i.

   In practice, these polynomials can be precomputed. Table 3.1 in the Appendix shows the degrees of $T_i(x)$ for $i \leq 20$.

2. Let $j \in \mathbb{N}$ be the largest integer such that $deg(T_j) < n$.

3. Let $k = n - deg(T_j)$.

4. Let:

   $$l = \begin{cases} \frac{k}{j+1} & if \ (j+1)|k \\ \lfloor \frac{k}{j+1} \rfloor - 1 & otherwise \end{cases}$$

5. Let $m = k - l(j+1)$. Note that m is either 0 or $j+1 < m < 2(j+1)$.

6. Let $I_1(x), I_2(x), \ldots, I_l(x)$ be randomly chosen irreducible polynomials of degree $j+1$. Let $J(x)$ be a randomly chosen irreducible polynomial of degree m if $m \neq 0$, or $J(x) = 1$ otherwise.

7. Let:
$$R'(x) = T_j(x)J(x)\prod_{i=1}^{l} I_i(x)$$
Note that because the polynomial $P(x) = x$ is irreducible, and therefore a factor of any $T_i(x)$, the constant term of $R'(x)$ is 0.

8. Let $R(x) = R'(x) + 1$.

The polynomial $R(x)$ created this way is clearly not divisible by any of the factors of $T_j(x)$, any of $I_i(x)$, $1 \leq i \leq l$, or $J(x)$. In other words, it is guaranteed to not have any factors with degree j or smaller.

In practice, this method requires the precomputed $T_i(x)$ polynomials, and a collection of irreducible polynomials of small degrees, to randomly choose the $I_i(x)$ and $J(x)$ polynomials from. In the current implementation, the largest $T_i(x)$ stored is $T_{23}(x)$, which has a degree of 16772836. The collection contains all irreducible polynomials of degree 23 or smaller, and 50000 polynomials each of degrees $24 \leq n \leq 47$.

Example 3.3.1. We wish to create an irreducible polynomial of degree $n = 216091$. The parameters defined by the method are the following:

- $j = 16$, $deg(T_j) = 130486$
- $k = 85605$
- $l = 5034$
- $m = 27$

The candidate polynomial $R(x)$ is created by multiplying $T_{16}(x)$, 5034 randomly chosen irreducibles of degree 17, and one degree 27 irreducible polynomial, and finally setting the constant term of the result to 1.

The polynomial $R(x)$ obtained this way can be tested for irreducibility using the methods described in subsection 3.2. Because of the way it was constructed, it is guaranteed to be in the $\sim 3.4\%$ of degree 216091 polynomials that have no factors of degree 16 or smaller. In other words, it is roughly 30 times more likely to be irreducible than a randomly chosen degree 216091 polynomial.

3.3.2 Q-Transform

A promising concept for constructing uniformly distributed high order linear recurring sequences is the application of the theory of Q-transform. In this subsection, some definitions and results that allow us to formulate infinite series of irreducible polynomials are introduced. The Q-transform and its properties are presented in [22].

Based on the algorithms described earlier in the chapter, we can use such polynomials for creating uniformly distributed pseudorandom sequences with large period lengths.

During the subsection, q is a prime power, $\mathbb{F}$ denotes a field, $\mathbb{F}_q$ is a finite field of q elements, and $\mathbb{K}$ is an algebraic extension field of $\mathbb{F}$ or $\mathbb{F}_q$, depending on the context. The proofs of the propositions presented here can be found in [7], [22] and [23].

Definition 3.3.1 (Reciprocal of a polynomial, self-reciprocal polynomials)**.** Let $P \in \mathbb{F}[x]$ be a polynomial of degree d. We say that the *reciprocal* polynomial of P is $P^*(x) = x^d P(x^{-1})$. We call a polynomial P *self-reciprocal*, if $P = P^*$.

Remark 1. a) If $P \in \mathbb{F}[x]$, then $P^* \in \mathbb{F}[x]$ and $(P^*)^* = P$.

b) Let $P, R, S \in \mathbb{F}[x]$ be such that $S = P \cdot R$. Then $S^* = P^* \cdot R^*$.

c) For any $P \in \mathbb{F}[x]$, P is irreducible if and only if P^* is irreducible.

d) Let $P, R, S \in \mathbb{F}[x]$ be such that $S = P \cdot R$. If P and R are self-reciprocal, then S is self-reciprocal, as well.

Proposition 1. Let $P \in \mathbb{F}[x]$, and $\mathbb{K}$ be the splitting field of P. Then the following statements are equivalent.

a) P is self-reciprocal;

b) $\forall \alpha \in \mathbb{K} \setminus \{0\}$: $P(\alpha) = 0$ implies $P\left(\alpha^{-1}\right) = 0$.

Corollary 1. If $P \in \mathbb{F}[x]$ is self-reciprocal and irreducible of odd degree, then $P(x) = ax + a$, with some $a \in \mathbb{F}$.

Corollary 2. Let $P \in \mathbb{F}[x]$ be a self reciprocal polynomial, and $P_1, \ldots, P_k \in \mathbb{F}[x]$ be distinct irreducible polynomials such that $P = P_1^{n_1} \cdot \cdots \cdot P_k^{n_k}$. Then for each $1 \leq i \leq k$ there exists $1 \leq j \leq k$ such that $P_i = P_j^*$ and $n_i = n_j$.

Remark 2. In the previous corollary, $i = j$ if and only if P_i is self-reciprocal.

Definition 3.3.2 (Q-transform)**.** Let $P \in \mathbb{F}[x]$ be a polynomial of degree d. The Q-transform of P is $\tilde{P}(x) = x^d P(x + x^{-1})$.

In practice, the Q-transform of $P(x) = \sum_{i=0}^{n} a_i x^i$ is computed as the following:

$$\tilde{P}(x) = \sum_{i=0}^{n} a_i (x^2 + 1)^i x^{n-i}$$

Remark 3. If $P \in \mathbb{F}[x]$, then $\tilde{P} \in \mathbb{F}[x]$, and $\deg(\tilde{P}) = 2\deg(P)$.

Proposition 2. Let $P, R, S \in \mathbb{F}[x]$ be such that $S = P \cdot R$. Then $\tilde{S} = \tilde{P} \cdot \tilde{R}$.

Let $P \in \mathbb{F}[x]$, and $\alpha \in K \setminus \{0\}$. Then $\tilde{P}(\alpha) = 0$ if and only if $\tilde{P}(\alpha^{-1}) = 0$. By Proposition 1, we may state the following.

Proposition 3. If $P \in \mathbb{F}[x]$, then $\tilde{P}$ is self-reciprocal.

Proposition 4. The Q-transform is an injection.

Let

$$\mathcal{P}_q(d) = \{P \mid P \in \mathbb{F}_q[x], \deg(P) = d\} \ ,$$
$$\mathcal{Q}_q(d) = \{P \mid P \in \mathbb{F}_q[x], \deg(P) = 2d, P = P^*\} \ .$$

Since $|\mathcal{P}_q(d)| = |\mathcal{Q}_q(d)|$, Proposition 4 implies the following.

Corollary 3. Let $P \in \mathbb{F}_q[x]$ be a self-reciprocal polynomial. Then there exists a unique $R \in \mathbb{F}_q[x]$ such that $P = \tilde{R}$.

Notation 1. Let $P \in \mathbb{F}[x]$ and $k \in \mathbb{N}$. We denote by $\tilde{P}^{(k)}$ the following iterated Q-transform:

$$\text{if } k = 0, \text{ then } \tilde{P}^{(k)} = P \ ;$$
$$\text{if } k > 0, \text{ then } \tilde{P}^{(k)} = \tilde{R} \text{ , where } R = \tilde{P}^{(k-1)} \ .$$

Corollary 4. Let $P \in \mathbb{F}_q[x]$ be a self-reciprocal polynomial. Then there exists a unique $R \in \mathbb{F}_q[x]$, not a self-reciprocal polynomial, and $k \in \mathbb{N}$ such that $P = \tilde{R}^{(k)}$.

Corollary 5. Let $P \in \mathbb{F}_q[x]$ be irreducible. Then $\tilde{P}$ is either irreducible or there exist $P_1, P_2 \in \mathbb{F}_q[x]$ irreducible polynomials such that $\tilde{P} = P_1 \cdot P_2$, and $P_1 = P_2^*$.

Proposition 5. Let $P \in \mathbb{F}_2[x]$ be an irreducible polynomial in the form $P(x) = x^d + a_{d-1}x^{d-1} + \cdots + a_1 x + 1$. Then $\tilde{P}$ is irreducible if and only if $a_{d-1} = a_1 = 1$. Furthermore, the coefficient of the linear term of $\tilde{P}$ is 1.

Corollary 6. Let $P \in \mathbb{F}_2[x]$ be an irreducible polynomial, and $P(x) = x^d + x^{d-1} + a_{d-2}x^{d-2} + \cdots + a_2 x^2 + x + 1$. Then $\tilde{P}^{(k)}$ is irreducible for all $k \in \mathbb{N}$.

This result implies that any irreducible polynomial in the form as in Proposition 5 determines an infinite sequence of irreducible Q-iterated polynomials. Every self-reciprocal polynomial of even degree is contained in exactly one of such sequences.

Proposition 6. Let $P \in \mathbb{F}_q$ be an irreducible polynomial, accomplishing $\deg(P) = 2d$. Then P is self-reciprocal if and only if $\mathrm{ord}(P) | q^d + 1$.

For the construction of pseudorandom number sequences with high period length, we need irreducible polynomials of high order. Actually, the period length is proportional to the order. Based on our experience, we have the following conjecture.

Conjecture 1. Let $P \in \mathbb{F}_q[x]$ be an irreducible self-reciprocal polynomial of degree $\deg(P) = 4d$. Then $q^d + 1 < \mathrm{ord}(P)$.

Furthermore, we have encountered Q-iterated polynomials having maximal order in many cases.

3.4 Statistical testing

In this section, we describe a test carried out to examine the statistical properties of the pseudorandom number sequences generated using the previously detailed method. Two irreducible polynomials of large degree were created, one using the small factor excluding method presented in subsection 3.3.1, and one using Q-transformations, as described in subsection 3.3.2. The pseudorandom sequences generated using these polynomials were tested using the NIST statistical test suite.

The software and documentation of the NIST test suite are available at [24]. The suite includes 15 tests designed to examine the properties of pseudorandom bit sequences, such as:

- *Frequency test*: a simple check to determine the proportion of ones and zeroes in a binary sequence.

- *Runs test*: checking the number of runs (uninterrupted sequence of identical bits) of various lengths to see how closely matches the expected value in a truly random sequence.
- *DFT (Spectral) test*: determining the peak heights in the Discrete Fourier Transform of the sequence, with the purpose of finding periodic features.
- *Template matching test*: finding occurrences of predetermined target strings, to detect generators producing too many such patterns. Both overlapping and non-overlapping tests are included.
- *Maurer's "Universal Statistical" test*: checking whether or not the sequence can be significantly compressed without loss of information.
- *Linear complexity test*: attempting to determine the length of the LRS that characterizes the sequence.

The first irreducible polynomial tested, denoted by T_1, was generated using the small factor excluding method. The implementation uses the NTL (Number Theory Library) available at [25].

The degree of T_1 was chosen to be 216091. The reason for this choice is that $2^{216091} - 1$ is a Mersenne prime. Choosing a value this way simplifies the computation of the algorithm described in section 3.1. Note that the algorithm requires the computation of the order of the irreducible polynomial, which is a divisor of $2^d - 1$, where d is the degree of the polynomial. If d is large, this step becomes computationally impractical, but choosing $2^d - 1$ to be a prime gives a simple solution to the problem.

The second irreducible polynomial tested, denoted T_2, was created using iterated Q-transform, using the following method:

1. Let Q be a self-reciprocal irreducible monic polynomial, with $\deg(Q) = d$.
2. Run the algorithm described in the previous section, using Q as input. Let P be the candidate polynomial that remains after Step 4. Determine $S, R \in \mathbb{Z}[x]$ such that $P = SQ + R$, and $\deg(R) < \deg(Q)$.
3. Compute $T = S\tilde{Q}^{(n)} + r$, where $\tilde{Q}^{(n)}$ is the iterated Q-transform, described in Notation 1 of Section 3.3.2. Use T to construct the linear recurrence sequence.

Based on practical observation, if the sequence produced by P has uniform distribution, then the sequence produced by T will also have uniform distribution. However, the proof of this conjecture is currently an open question.

To create T_2, the following polynomial was used as a starting point:

$$\begin{aligned} Q_2 =& x^{14} + x^{13} + x^{12} + x^{11} + x^{10} + x^9 \\ &+ x^7 + x^5 + x^4 + x^3 + x^2 + x + 1 \ . \end{aligned}$$

As it is stated in Proposition 6, the order of a self-reciprocal irreducible monic polynomial of degree d is at most $2^{\frac{d}{2}} + 1$. The above polynomial was chosen because $\mathrm{ord}(Q), \mathrm{ord}(\tilde{Q})$, and $\mathrm{ord}(\tilde{Q}^{(2)})$ all reach this maximum value. Theoretically, it does not guarantee that this maximality property will hold after further Q-transformations, but practical observations suggest that the order of $\tilde{Q}^{(n)}$ will grow at a rate that is sufficient for use in the applications described in this paper. We stated our related experience in Conjecture 1.

Using the above method, $P_2 = P_2Q_2 + R_2$ was determined, and the polynomial to be used was set as $T_2 = S_2\tilde{Q_2}^{(14)} + R_2$. Note that $\deg(\tilde{Q_2}^{(14)}) = 229376$, and $\deg(T_2) = 229378$.

Using T_1 and T_2, two LRSs were created to generate the pseudorandom sequences to be tested, denoted $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively. Both LRSs generate 64-bit words. Following the recommendations in the documentation of the NIST test suite, 16MB (2^{21} words) of test data were generated using $\mathcal{L}_1$ and $\mathcal{L}_2$ each.

For each of these two streams, the NIST suite split the data into 100 bitstreams. The testing software provides a detailed output of the tests, as well as a summary showing the number of bitstreams that passed each test. The minimum pass rate for a test is considered to be 96 out of a sample size of 100.

Table 3.2 in the Appendix shows some of the result obtained from the tests. The full report can be found at
https://arato.inf.unideb.hu/major.sandor/statistical_results/.

The results show the two generators to have very similar statistical properties, with $\mathcal{L}_2$ being only slightly weaker in some tests. Since the order of T_2 is significantly lower than the order of T_1, this result is to be expected. Also of note is that both generators produced very high quality pseudorandom sequences, passing all relevant benchmarks set by the test suite.

This shows that using the Q-transformation described above to generate irreducible polynomials of very large degree is completely suitable for use in generating uniformly distributed pseudorandom linear recurrence sequences.

Chapter 4

Performance tests

In this chapter, the question of how to efficiently implement the algorithm to create pseudorandom number sequences introduced in Section 3.1 is investigated in detail.

Section 4.1 describes the various solutions that were used to implement the computation heavy steps of the algorithm.

Section 4.2 details a hardware module for matrix multiplication over $GF(4)$. The hardware module described in Section 4.3 carries out polynomial modulo over $GF(2)$. These modules were created by the author using a RIVYERA FPGA platform as highly parallelized implementations of the required computation heavy steps.

As mentioned in Section 3.1, in practice, the approach based on polynomial modulo has proven to be significantly more efficient than the approach based on matrix exponentiation. Section 4.4 presents the results of comparing the different hardware and software implementations of polynomial modulo and polynomial GCD.

4.1 Software libraries and hardware platform

The following software libraries and hardware platform were used in the test comparing polynomial modulo and polynomial GCD operations found in a later section.

4.1.1 NTL

NTL is a number theory library authored by Victor Shoup, which started development in 1990. It is distributed online under LGPLv2.1+ licence at *https://www.shoup.net/ntl*. The performance test was done using version 11.3.2. The library is written in C++ and uses GMP (GNU Multi-Precision library).

NTL implements base classes for arbitrary length integers, finite fields and rings of integers modulo n and classes for vectors, matrices and polynomials over these bases. It provides efficient algorithms for common operations such as arbitrary precision integer arithmetic and floating point arithmetic, polynomial arithmetic and factorization, irreducibility testing, lattice basis reduction, and linear algebra.

Notably for our goal, an optional *gf2x* package is also available, which implements highly optimized algorithms specifically for polynomials over $GF(2)$, significantly improving the performance of multiplication, division and GCD. The performance test makes use of the *gf2x* package.

4.1.2 FLINT

FLINT (Fast Library for Number Theory) started development in 2007 by William Hart and David Harvey. It is likewise distributed under LGPLv2.1+, available at *http://www.flintlib.org/index.html*. Version 2.5.2 was used in the test. It is written in ANSI C and depends on either GMP or MPIR (Multiple Precision Integers and Rationals) libraries, and the MPFR (Multiple Precision Floating-Point Reliably) library.

Similarly, FLINT supports arithmetic operations with bases such as multi-precision integers and rationals, rings of integers modulo n, finite fields, real and complex numbers and p-adic numbers. It also supports polynomials, power series and matrices over these bases. It offers a large number of algorithms for these data types, the full documentation of which can be found online at *http://www.flintlib.org/flint-2.5.pdf*.

For polynomial arithmetic, the version used offers five suitable representations for $GF(2)$, depending on the base class used:

- Polynomials over $\mathbb{Z}/n\mathbb{Z}$
- Polynomials over $\mathbb{Z}/n\mathbb{Z}$ when n is small (a separate implementation from the previous one for word-sized moduli)
- Polynomials over finite fields

- Polynomials over finite fields using small representation (i.e. the characteristic is word-sized)
- Polynomials over finite fields using Zech representation, where the elements of the field are represented as a power of a generator

FLINT and NTL have been the subject of performance comparisons before, the common conclusion being that the two systems excel in different areas. General benchmarks can be found on the websites of both libraries.

4.1.3 SciEngines RIVYERA

The third implementation to be compared in the perfomance test is a hardware module created by the author on a SciEngines RIVYERA S6-LX150 FPGA-computer.

The RIVYERA is the successor to the COPACOBANA that was used to break the DES cipher in 2006 [26]. It is used in fields such as cryptography, digital signal processing, data mining, high performance computing and ASIC prototyping.

It enables very large scale physical parallelism with the use of FPGAs (field-programmable gate arrays). These chips are reprogrammable hardware that can be configured to create custom circuits using look-up tables, registers, block RAM, dedicated multipliers and other simple computing elements.

The machine used in the test contains 16 Spartan-6 LX150 FPGAs. SciEngines provides an API for the architecture that consists two major parts: a host-side API in C and Java, and an FPGA-side API in VHDL hardware definition language.

4.2 Matrix multiplication implementation

The hardware module implements a massively parallel version of the classic $O(n^3)$ matrix multiplication algorithm. The algorithm is very well suited for parallelization, as each element of the result can be computed independently. In fact, all n^3 basic multiplications could be done in parallel, given enough processing hardware.

The implemented matrix multiplication operates on two 1024×1024 matrices over $GF(4)$, every element being represented as two bits. On the FPGAs, this size is considered fixed. For smaller $n \times n$ sized matrices, the data is padded with $1024 - n$ rows and columns of zeroes on the host PC, before being sent to

the FPGAs. Let A and B be the two input matrices, and $C = AB$ the output matrix we want to compute.

Logically, the matrices are divided into blocks. The size of the blocks is determined by the multiplier hardware created on the FPGAs. Each block contains 16×16 elements (512 bits). Using these blocks, the matrices are handled as 64×64 sized block matrices:

$$M = \begin{pmatrix} b_{(0,0)} & b_{(0,1)} & \cdots & b_{(0,62)} & b_{(0,63)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ b_{(61,0)} & b_{(61,1)} & \cdots & b_{(61,62)} & b_{(61,63)} \\ b_{(62,0)} & b_{(62,1)} & \cdots & b_{(62,62)} & b_{(62,63)} \\ b_{(63,0)} & b_{(63,1)} & \cdots & b_{(63,62)} & b_{(63,63)} \end{pmatrix}$$

The RIVYERA used to implement the module contains 16 FPGAs. All 16 are used in parallel to compute the result. Physically, the FPGAs are located on two cards with 8 FPGAs each. Every chip has a slot address $s_a \in \{0, 1\}$ and an FPGA address $f_a \in \{0, 1, \ldots, 7\}$. Each FPGA computes a 16×16 block area of the result, divided as shown in figure 4.1.

Slot 0, FPGA 0	Slot 0, FPGA 1	Slot 0, FPGA 2	Slot 0, FPGA 3
Slot 0, FPGA 4	Slot 0, FPGA 5	Slot 0, FPGA 6	Slot 0, FPGA 7
Slot 1, FPGA 0	Slot 1, FPGA 1	Slot 1, FPGA 2	Slot 1, FPGA 3
Slot 1, FPGA 4	Slot 1, FPGA 5	Slot 1, FPGA 6	Slot 1, FPGA 7

Figure 4.1: Division of matrix between FPGAs

For $M \in \{A, B, C\}$, $i, j \in \{0, 1, 2, 3\}$, let M_{i*} and M_{*j} denote one quarter of the matrix, and M_ij denote one 16×16 block area of the matrix, arranged in the following way:

$$M = \begin{pmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{23} \\ M_{30} & M_{31} & M_{32} & M_{33} \end{pmatrix} \quad M_{i*} = \begin{pmatrix} M_{i0} & M_{i1} & M_{i2} & M_{i3} \end{pmatrix}$$

$$M_{*j} = \begin{pmatrix} M_{0j} \\ M_{1j} \\ M_{2j} \\ M_{3j} \end{pmatrix}$$

F_{ij} will denote the FPGA computing C_{ij}. Each FPGA F_{ij} requires A_{i*} and B_{*j} as input. These segments consist of 16×64 blocks.

Block RAM was used to store the data on the FPGA chips. The Spartan-6 LX150 chips are equipped with 268 units of 18Kb block RAM, which can be cascaded together to form larger memory arrays. To store A_{i*} and B_{*j}, the two pieces of RAM created can store 1024 blocks, configured with a 512 bit wide data port and a 10 bit address. Since one block is 512 bits, this way accessing any address of the RAM can return one entire block. The RAM created to store C_{ij}, which is 16×16 blocks, likewise has a 512 bit wide data port, but only 8 bits of address, capable of storing 256 blocks. In total, the design uses 1152Kb of block RAM on each FPGA chip.

The multiplier unit on each FPGA is created using look-up tables (LUTs). The look-up tables are organized on the chip into 23038 slices, each slice containing four 6-input LUTs and eight flip-flops. Functionally, the 6-LUTs can be interpreted as small 64 bit pieces of memory with 6 bits of address and a 1 bit data output. These 64 bits can then be configured with an appropriate value to create logic gates computing arbitrary 6-input bool functions.

The multiplier unit is created from a network of such logic gates, and is purely combinational logic. The multiplier can perform matrix multiplication on two blocks of data in one clock cycle. The structure of this unit is hierarchical.

The smallest unit of the multiplier, denoted m, performs a multiply-accumulate operation:

$$c \equiv ab + s \ mod \ 4$$

where a, b, c, s are two bit values, $a = \overline{a_1 a_0},\ b = \overline{b_1 b_0},\ c = \overline{c_1 c_0}, s = \overline{s_1 s_0}$. Two LUTs are used to create each m, connected as shown in figure 4.2. The LUT computing c_1 requires all six bits of input, while the LUT computing c_0 requires only $a_0,\ b_0,\ s_0$.

16 multiply-accumulate units are serially connected together to form a unit, denoted $\hat{m}$, that computes the dot product of vectors. Let $\hat{m} = (m_0, m_1, \ldots, m_{15})$ denote the multiply-accumulate units in $\hat{m}$. Two units m_i and m_{i+1} are connected such that $c_i = s_{i+1}$. The output of $\hat{m}$ is c_{15}.

Let $u = (u_0, u_1, \ldots, u_{15})$ and $v = (v_0, v_1, \ldots, v_{15}$ be two vectors with 16 elements over $GF(4)$. Connected in the way shown in figure 4.3, the output of $\hat{m}$ is $u \cdot v$.

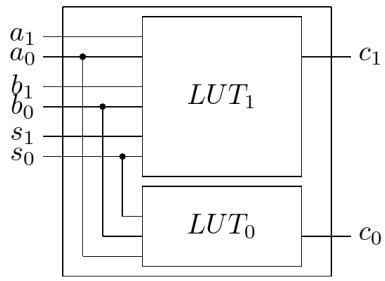

Figure 4.2: Structure of basic multiply-accumulate unit

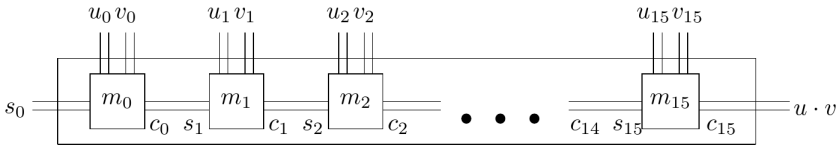

Figure 4.3: Structure of dot product unit

One $\hat{m}$ unit requires 32 LUTs to create. The dot product of vectors with more than 16 elements can be computed iteratively. If the output of $\hat{m}$ is connected to s_0 and the vectors are broken up into 16-element segments, the dot product can be computed by pushing one segment of the vectors onto the input of $\hat{m}$ every clock cycle.

On each FPGA, 256 dot product units are arranged into one matrix multiplier unit, denoted $\hat{M}$, using a total of 8192 LUTs. More precisely, it takes as input three blocks of data, A^b, B^b, S^b, and performs $C^b = A^b B^b + S^b$. As with the dot product unit, matrices $A' \in M_{n\times 16},\ B' \in M_{16\times n},\ n > 16$ can be multiplied iteratively if the output of $\hat{M}$ is connected to its S-input and the matrices are broken up into 16×16 element blocks. Such a multiplication will take $\lceil \frac{n}{16} \rceil$ clock cycles.

Using $\hat{M}$, FPGA F_{ij} can compute C_{ij} using A_{i*} and B_{*j}. Let $c_{kl}, k, l = 0, 1, \ldots, 15$ denote the blocks in C_{ij}, $a_{kl}, k = 0, 1, \ldots, 15, l = 0, 1, \ldots, 63$ denote the blocks in A_{i*}, $b_{kl}, k = 0, 1, \ldots, 63, l = 0, 1, \ldots, 15$ denote the blocks in B_{*j} and s denote a single block which will be used for iterative multiplication as mentioned above. The classic matrix multiplication algorithm is performed, as seen in listing 4.1 in the Appendix. This algorithm takes 16384 clock cycles to perform.

In the following subsections, we compare the running time of the created hardware module to implementations found in widely used number theory software libraries NTL and FLINT.

4.2.1 Performance test on the RIVYERA

The modules created on the RIVYERA run at a 100MHz clock speed. At this frequency, the multiplication itself takes $\sim 163\,\mu s$. Other operations performed on the FPGAs include reading the input data and storing it in block RAM before the computation, and sending the output from block RAM to the host PC afterward. Since the module consists of purpose-built circuits that perform no other function, the time between the FPGA receiving input and producing output is always constant.

The host side software needs to perform operations such as starting and stopping threads, data formatting operations, dividing the matrices A and B between the FPGAs before sending the input, and piecing together the matrix C from the outputs received. The speed of the host side software is nondeterministic, as it is influenced by the scheduling of the operating system.

Table 4.1 in the Appendix shows the average running time of a matrix multiplication, including the above mentioned host side operations. The first column shows the number of repetitions used in calculating the average. The second column shows the size of the matrices, $A, B \in M_{n\times n}$. The matrices were generated randomly, using standard library functions, with uniformly distributed elements over $GF(4)$. The third column shows the measured running times. Since matrices smaller than 1024×1024 are padded with zeroes, the small differences in running time are only due to the nondeterministic speed of the host side. Values of n other than 1024 are included for the sake of comparison with NTL and FLINT.

4.2.2 Performance test of NTL

The computer the test was carried out on has an i7-7700HQ CPU (4 physical cores at 2.8GHz clock speed) and 16GB of RAM at 2400MHz.

For matrix multiplication over $GF(4)$, NTL provides two suitable classes:

- *mat_ZZ_p* for matrices over integers modulo p, where p is an arbitrary precision integer.
- *mat_zz_p* for matrices over integers modulo p, where p is a single-precision integer.

Table 4.2 in the Appendix shows the measured average running times, averaged over a number of repetitions shown in the first column, for $n \times n$ sized matrices. The matrices were generated randomly using functions provided by the NTL library.

4.2.3 Performance test of FLINT

The test was carried out on the same computer as the NTL performance test.

For matrix multiplication over $GF(4)$, FLINT provides four suitable representations:

- Matrices over $\mathbb{Z}/n\mathbb{Z}$, where n is a word-sized, fixed modulus
- Matrices over finite fields with arbitrary characteristic (denoted FQ)
- Matrices over finite fields with word-sized characteristic (denoted FQ NMOD)
- Matrices over finite fields using Zech logarithm representation, where the elements of the field are represented as powers of a generator for the multiplicative group of the field (denoted FQ ZECH)

Tables 4.3 and 4.4 in the Appendix shows the measured average running times, averaged over a number of repetitions shown in the first column, for $n \times n$ sized matrices as shown in the second column. The matrices were generated randomly using functions provided by the FLINT library.

4.3 Polynomial modulo implementation

This module implements polynomial modulo over $GF(2)$: given polynomials $P, Q \in F_2[x],\ deg(P) < 2^{20},\ deg(P) \leq deg(Q) < 2^{21}$, we will compute $R(x) = Q(x)\ mod\ P(x)$. Because the polynomials are over $GF(2)$, each term can be represented by a single bit, and addition and subtraction are both equivalent to the bitwise XOR operation. To compute $R(x)$, only bit shifting and XOR operations will be required. Since the XOR operation is done independently for each term, the operation can be efficiently parallelized.

Let

$$P(x) = \sum_{i=0}^{n} p_i x^i,\ Q(x) = \sum_{j=0}^{m} q_j x^j,\ R(x) = \sum_{k=0}^{o} r_k x^k,\ p_i, q_j, r_k \in \{0, 1\}, o < n \leq m$$

The algorithm used to compute $Q(x)\ mod\ P(x)$ is shown in listing 4.2 in the Appendix.

Like the module described in the previous section, all 16 FPGAs of the RIVYERA are used in parallel when computing the polynomial modulo. $P(x)$ is stored in shift registers distributed among the FPGAs, with a maximum physical size of 1Mb when $deg(P) = 2^{20} - 1$. $Q(x)$ is stored in both shift registers and block RAM, with a maximum physical size of 2Mb when $deg(Q) = 2^{21} - 1$. The coefficients of the polynomials are aligned such that performing $P'(x) = P(x)x^s$ is not necessary on the FPGAs, in practice $P'(x) = P(x)$. Before sending the data to the chips, the host aligns the polynomials such that the leading coefficient will be at the most significant bit of the container. If $deg(P) < 2^{20}-1$, then it is shifted $2^{20} - 1 - deg(P)$ to the left. Likewise, if $deg(Q) < 2^{21} - 1$, then it is shifted $2^{21} - 1 - deg(Q)$ to the left.

The FPGAs are connected serially as shown in figure 4.4. At the start of the computation, F_i receives as input P'_i and Q'_i, two continuous 64Kb segments of $P(x)$ and $Q(x)$. These segments are stored in two shift registers created from LUTs, denoted S_P and S_Q. One quarter of LUTs on a Spartan-6 FPGA can be configured as 32 bit shift registers. The required containers are created using 2048 LUTs each. F_{15} receives the most significant bits of the polynomials, while F_0 receives the least significant bits. The shift registers store only the upper half of $Q(x)$, the lower half, denoted Q_L, is stored on F_0 in a 1Mb block RAM, denoted B_Q. This block RAM was created with a 64 bit wide data port and 14 bit address. After the computation finishes, the output data will be in S_Q. F_i computes R'_i, a 64Kb continuous segment of $R(x)$.

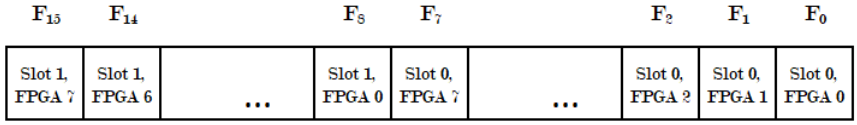

Figure 4.4: Division of polynomials between FPGAs

An important difference between this hardware module and the matrix multiplication one described previously is in the communication between the FPGAs. In the matrix multiplication module, each FPGA only communicated with the host CPU and were completely independent of each other. To compute the polynomial modulo, data must be sent between the FPGAs as well, requiring synchronization between the individual chips. In the algorithm above, the decision between the next step being an XOR or a bit shift operation depends

on the most significant bit of the container of $Q(x)$: if the bit is 1, an XOR operation is performed, otherwise $Q(x)$ is bit shifted to the left. Because this decision effects the entire polynomial, every FPGA needs to be aware of the value of this bit at every step of the computation.

For this purpose, the chips are configured with one of three variants of the hardware module, depending on their place in figure 4. All FPGAs perform XOR and bit shifting operations on their polynomial segments as described above, as well as receiving input from and sending output to the host CPU. FPGAs $F_i,\ i \in \{1, 2, \ldots 14\}$ receive data from F_{i-1} and F_{15}, and sent data to F_{i+1}. F_0 manages the block RAM B_Q, sends data to F_1 and receives data from F_{15}. F_{15} receives data from F_{14} and sends the status of the most significant bits to every other FPGA. The RIVYERA architecture provides a 64 bit data bus for communication between FPGAs. In the following, one word of data will refer to 64 bits.

In the following section, the differences between the three variants of the hardware are described in detail.

4.3.1 Variant F_i

This variant of the hardware module is used to configure all FPGAs with the exception of F_{15} and F_0. At the start of the computation, aside from the polynomial segments as described above, FPGA F_i also receives the following inputs:

- $s = deg(Q) - deg(P)$. This parameter will determine the total number of bit shift operations to perform.
- MSW_P: the most significant word of $P(x)$. This value does not change during the computation.
- MSW_Q: the most significant word of $Q(x)$. MSW_Q and MSW_P determine the steps of the computation.
- wb_{Qi}: overlapping data word of $Q(x)$ between F_i and F_{i-1}
- wb_{Pi}: overlapping data word of $P(x)$ between F_i and F_{i-1}
- wf_{Pi}: overlapping data word of $P(x)$ between F_i and F_{i+1}

Parameters wb_{Qi}, wb_{Pi} and wf_{Pi} represent overlapping data between the neighbouring FPGAs, as shown in the following figure. Note that $wb_{Qi} =$

wf_{Qi-1}, $wb_{Pi} = wf_{Pi-1}$ and $wf_{Pi} = wb_{Pi+1}$. These overlaps are required so the FPGAs can work in parallel without having to wait for every bit of new data from other chips. Instead, FPGAs only need to wait for new data words after every 64 bit shift operations. The word wf_{Qi} is not an input parameter, its value is calculated during the computation.

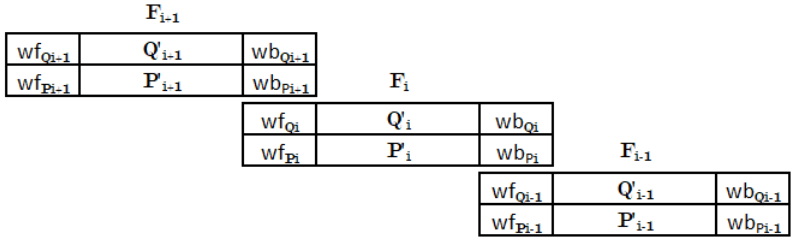

Figure 4.5: Overlapping data between FPGAs

After receiving all inputs as described above, the module begins the computation, which is broken up into a number of iterations equal to $\lceil \frac{s}{64} \rceil$.

The algorithm followed during an iteration is shown in listing 4.3 in the Appendix. The module depends on the values of MSW_P and MSW_Q to determine the next step. One iteration is completed after 64 bit shift operations have been performed. At this point, the value of MSW_Q is considered invalid, as it no longer holds any data about the most significant bits of $Q(x)$. In each iteration, the module produces one word of data, wf_{Qi}, that will be sent to F_{i+1}.

The entire computation is finished when the total number of bit shift operations, denoted b_T, is equal to s. At this point, if the most significant bit of $Q(x)$ is 1, then one last XOR operation is performed on all necessary values. Afterward, the completed output data is sent to the host CPU.

Because S_P and S_Q that store P'_i and Q'_i are created from LUTs configured as 32 bit shift registers, $Q'_i = Q'_i \ XOR \ P'_i$ takes 32 clock cycles. All other basic operations take only a single clock cycle.

Parallel to the above algorithm, the module also waits for incoming data from other FPGAs. Before the next iteration begins, the FPGA needs to receive one word from F_{15} that will be the the new MSW_Q for that iteration and one word from F_{i-1} that will be the new wb_{Qi}. The FPGA can receive and store these words while an iteration is in progress, or in-between iterations. At the end of an iteration, wf_{Qi} is sent to F_{i+1} to be wb_{Qi+1}. After the required data words are received and the output is sent, the next iteration begins.

4.3.2 Variant F_{15}

This variant of the hardware module is used to configure F_{15}, the FPGA which stores the most significant bits of the polynomials. The module functions similarly to the previous variant, with a few differences.

The input parameters received at the beginning of the computation are the same as in the previous variant, with the exception of the absence of wf_{P15}. F_{15} does not calculate wf_{Q15}, since it is the highest indexed FPGA. Note that F_{15} does not strictly require MSW_P and MSW_Q, as it could access the most significant bit of $Q(x)$ directly from the polynomial, but the physical implementation does keep them for the sake of minimal divergence from other module variants.

The iteration algorithm of F_{15} is very similar to the F_i variant, with the exception of some missing operations, as shown in listing 4.4 in the Appendix.

At the end of an iteration, the most significant word of $Q(x)$ becomes the new value of MSW_Q. This value is also broadcast to all other FPGAs to be used in the next iteration.

Like the previous variant, during or after each iteration, F_{15} needs to receive a data word from F_{14} that will be the new value of wb_{Q15} for the next iteration.

After all necessary data are exchanged, the next iteration begins.

4.3.3 Variant F_0

This variant of the hardware module is used to configure F_0, which stores the least significant bits of the polynomials. This FPGA has the 1Mb block RAM B_Q, which is loaded with the lower half of $Q(x)$ before the computation begins.

The input parameters received by F_0 are the same as variant F_i, with the exception of the absence of wb_{P0}. Since it is the lowest indexed FPGA, F_0 has no overlapping data with F_{i-1}.

The iteration algorithm for F_0 can be seen in listing 4.5 in the Appendix.

After each iteration, the new value of wb_{Q0} will be read from B_Q. Like the F_i variant, F_0 needs to receive one word from F_{15} during or after each iteration that will be the new value of MSW_Q for the next iteration. F_0 sends wf_{Q0} to F_1, which will use it as the value of wb_{Q1} in the next iteration.

After all necessary data are exchanged, the next iteration begins.

In the following section, we compare the performance of the polynomial modulo hardware implementation with the software implementation found in NTL and FLINT.

4.4 Testing polynomial modulo and GCD

The performance test focused on two polynomial operations that are commonly used in irreducibility testing: polynomial modulo and polynomial GCD. The following tables compare the running times of these operations for different input sizes.

The column n shows the number of repetitions: n pairs of polynomials were randomly generated for each test. The second and third columns, marked $deg(P)$ and $deg(Q)$ show the upper limit of the degree of the polynomials generated. Some variance in the degrees was allowed when randomly generating polynomials, the lower limit was set to be 98% of the upper limit. The fourth column shows the average running time. The random polynomial generation was always done using the random function provided by the tested library.

Two tables are presented for each implementation, the first one where $deg(P)$ and $deg(Q)$ are close, and the second one where $deg(Q) \approx 2deg(P)$. For the polynomial modulo operation, in practical situations, the second case is the one encountered in irreducibility tests like Rabin's or Ben-Or's. The running time of these of these operations is dependent on the difference between the degrees of the polynomials. The first case is present to demonstrate the running time when this value is low.

The computer the test was carried out on has an i7-7700HQ CPU (4 physical cores at 2.8GHz clock speed) and 16GB of RAM at 2400MHz.

4.4.1 Polynomial modulo

The tables referenced in this section show the average running times for the operation $Q(x) mod\ P(x), P, Q \in \mathbb{F}_2[x], deg(Q) \geq deg(P)$.

- **RIVYERA**: A detailed description of the hardware module used in the test can be found in Section 4.3. The created module uses all 16 available FPGAs of the RIVYERA in parallel. The tests were run at a 100MHz clock speed on the FPGAs. The FPGAs use a PCI Express connection to carry out I/O (i.e. reading the polynomials and sending back the result) with the host CPU.

  It implements shift registers and memory blocks to store the polynomials, divided evenly among the FPGAs. The maximum size of the polynomials that can be stored are 2^{20} bits ($deg(P) = 1048575$) and 2^{21} bits ($deg(Q) = 2097151$). The average running time for polynomials of maximum size is 265 ms ($n = 10000$).

Because of the parallel structure of the module, the running time is linear in $deg(Q) - deg(P)$, with a constant $c \approx 23.8\,\text{ms}$ overhead for I/O, independent of $deg(P)$ and $deg(Q)$.

Table 4.5 in the Appendix shows the results of the tests.

- **NTL**: The $gf2x$ optional package was used to optimize the operation. Table 4.6 in the Appendix shows the results of the tests.
- **FLINT $\mathbb{Z}/n\mathbb{Z}$ representation**: In this representation, the modulus can be arbitrary sized. Table 4.7 in the Appendix shows the results of the tests.
- **FLINT $\mathbb{Z}/n\mathbb{Z}$ representation when n is small**: In this representation, the modulus is word-sized. Table 4.8 in the Appendix shows the results of the tests.
- **FLINT finite fields representation**: In this representation the polynomials are over finite fields with arbitrary characteristic. Table 4.9 in the Appendix shows the results of the tests.
- **FLINT finite fields representation with small characteristic**: In this representation the polynomials are over finite fields with word-sized characteristic. Table 4.10 in the Appendix shows the results of the tests.
- **FLINT finite fields using Zech logarithm representation**: In this representation the elements of the finite fields are powers of a generator for the multiplicative group of the field. Table 4.11 in the Appendix shows the results of the tests.

4.4.2 Polynomial GCD

The tables referenced in this section show the average running times for the operation $gcd(P(x), Q(x)), P, Q \in \mathbb{F}_2[x]$.

- **NTL**: The $gf2x$ optional package was used to optimize the operation. Table 4.12 in the Appendix shows the results of the tests.
- **FLINT $\mathbb{Z}/n\mathbb{Z}$ representation**: In this representation, the modulus can be arbitrary sized. Table 4.13 in the Appendix shows the results of the tests.

- **FLINT $\mathbb{Z}/n\mathbb{Z}$ representation when n is small**: In this representation, the modulus is word-sized. Table 4.14 in the Appendix shows the results of the tests.
- **FLINT finite fields representation**: In this representation the polynomials are over finite fields with arbitrary characteristic. Table 4.15 in the Appendix shows the results of the tests.
- **FLINT finite fields representation with small characteristic**: In this representation the polynomials are over finite fields with word-sized characteristic. Table 4.16 in the Appendix shows the results of the tests.
- **FLINT finite fields using Zech logarithm representation**: In this representation the elements of the finite fields are powers of a generator for the multiplicative group of the field. Table 4.17 in the Appendix shows the results of the tests.

Conclusions

For the specific operations of polynomial modulo and polynomial GCD over $GF(2)$ using dense polynomials, the NTL library provides the fastest solutions.

For the polynomial modulo operation, this is followed by the RIVYERA hardware implementation. The time complexity of the hardware module is linear in $deg(Q) - deg(P)$, while the software libraries show a non-linear growth rate, meaning that for large enough polynomials the hardware implementation would overtake the NTL library in performance, but in its current status this would require impractically large polynomials.

Of the five representations FLINT provides for polynomials over $GF(2)$, the best performance can be obtained from using $\mathbb{Z}/n\mathbb{Z}$ representation with word-sized n. The finite field representations proved significantly slower than those previously mentioned.

Chapter 5

Comparison of pseudorandom number sequences

In Section 3.1 an algorithm was described to create uniformly distributed pseudorandom linear recurrence sequences with arbitrarily large period lengths. This algorithm requires an irreducible polynomial over $GF(2)$ as input, and outputs the corresponding recurrence relation (LRS).

Section 3.3 details two methods of creating irreducible polynomials with large degree and order that are suitable for the algorithm.

Chapter 4 investigated how to efficiently implement the algorithm in practice, testing the performance of multiple hardware and software solutions for the computationally expensive steps of the algorithm.

Based on the knowledge gathered this way, the polynomials modulo approach of the algorithm was implemented using the NTL software library.

In order to actually generate pseudorandom numbers using the recurrence relations created by the algorithm, the recurrence relations are implemented as *linear feedback shift registers (LFSR)*.

As described in [27], a shift register is a sequential logic circuit where the storage units are connected in a linear way, such that data is shifted down the line when the circuit is activated. An LFSR is a shift register where after each activation, the element that is shifted out of the storage unit with the highest index is taken as the output, and the storage unit with the lowest index receives

an input that is a linear function of some of the other values stored in the registers.

LFSRs are well suited for generating pseudorandom number sequences. Examples of these applications can be found in [28]. In practice, the function used to calculate the new element is usually either bitwise XOR, or a linear combination modulo a power of two.

In the implementation of the algorithm, the elements of the LFSRs created are l-bit words, $l = 64$ by default. The function to calculate the new element is a linear combination defined by the coefficients of the linear recurrence sequence, modulo 2^l. Note that the coefficients of the sequence are over $GF(4)$.

To test the sequences in practice, the NIST test suite was used to compare their statistical properties. The test suite has been described previously in Section 3.4.

Section 3.4 also compares sequences created using irreducible polynomials that were generated by different methods.

The sequences produced by the presented algorithm are also compared to other pseudorandom sequences that were defined in [29], [30] and [31]. These sequences have provably good measures of pseudorandomness, as defined in [32]. The test results of these sequences and the practical conclusions drawn from the comparisons are found in Section 5.2.

5.1 Experimental Results

Several LRSs were created with different attributes, to test the effects of the attributes on the statistical properties of the resulting pseudorandom sequences. The attributes of interest are the following:

- The degree of the irreducible polynomial used to create the LRS.
- The irreducible polynomial being *dense*, that is, having a large number of non-zero coefficients, or *sparse*.
- The initial values of the sequence, before it begins generating new pseudorandom values.
- Having a function added to the LRS that transforms each pseudorandom value generated.

Unless stated otherwise, the test data generated by each LRS consists of 2^{21} 64-bit words. The NIST test suite splits the data into 100 bitstreams. In

the following tables, the Proportion column will show the number of bitstreams that passed each test. Following the guidelines of the NIST documentation, a pass rate of 96 for a sample size of 100 is considered acceptable for a test.

5.1.1 Degree and density of irreducible polynomial

Let d be the degree of the polynomial $P(x) \in \mathbb{F}_2[x]$ that is used to create the LRS $\mathcal{L}$. Recall from Section 3.1 that the period length of $\mathcal{L}$ is directly proportional to the order of $P(x)$, which is a divisor of $2^d - 1$. To simplify the calculation of $ord(P)$, d will usually be chosen such that $2^d - 1$ is a Mersenne prime.

For the tests, the values of $d_1 = 1279$ and $d_2 = 19937$ were chosen. Section 3.3 describes LRSs created using polynomials with much larger degrees: 216091 and 229376.

Degrees d_1 and d_2 provide period lengths suitable for practical use. To compare, the standard implementation of the widely used pseudorandom number generator named the Mersenne Twister [33] also creates a sequence with period length $2^{19937} - 1$.

Another attribute of interest of the polynomial used is its density. In hardware, the number of gates required to implement a LFSR depends on the number of non-zero coefficients in the polynomial that defines the LRS. To minimize the hardware cost of implementation, the polynomial needs to be a *trinomial*, that is, a polynomial with exactly three terms.

Four LRSs were created for comparison:

- $\mathcal{L}_1$ using a dense polynomial of degree d_1
- $\mathcal{L}_1^t$ using a trinomial of degree d_1
- $\mathcal{L}_2$ using a dense polynomial of degree d_2
- $\mathcal{L}_2^t$ using a trinomial of degree d_2

Tables 5.1 and 5.2 in the Appendix show some of the result obtained from the tests. The full report can be found at
https://arato.inf.unideb.hu/major.sandor/comparisons_degree_density/.

In general, the tests show each of these LRSs create statistically suitable uniformly distributed values. Although it depends on the specific tests, LRSs using dense polynomials typically show better properties than ones using trinomials.

5.1.2 Initial values

Let $l_0, l_1, \ldots, l_{d-1}$ be the initial values of an LRS $\mathcal{L}$. The initial values of an LRS can also influence its statistical properties. In [4], a method of creating suitable initial values is given for LRSs made using the algorithm.

For the tests, an LRS $\mathcal{L}_3$, where the initial values were set using the recommended method is compared to an LRS $\mathcal{L}_4$, which was created using the same polynomial but the initial values are set to $l_0 = l_1 = \ldots = l_{d-2} = 0$, $l_{d-1} = 1$. An LRS with initial values set the same way as $\mathcal{L}_4$ is named an *impulse response sequence*.

The polynomial used to create these LRSs is the same one that was denoted T_1 in Section 3.4, and has degree 216091.

The 2^{21} words generated by $\mathcal{L}_4$ was broken into 8 segments of 2^{18} words each. These segments will be denoted $\mathcal{L}_4^{(0)}, \mathcal{L}_4^{(1)}, \ldots, \mathcal{L}_4^{(7)}$, where $\mathcal{L}_4^{(0)}$ contains the first 2^{18} words generated by $\mathcal{L}_4$, $\mathcal{L}_4^{(1)}$ contains the next 2^{18} words, and so on for each $\mathcal{L}_4^{(i)}$.

Tables 5.3, 5.4 and 5.5 in the Appendix show some of the result obtained from the tests. The full report can be found at
https://arato.inf.unideb.hu/major.sandor/comparison_initial_values/.

The LRSs pass the overwhelming majority of statistical tests, demonstrating suitable randomness and uniform distribution for practical purposes.

$\mathcal{L}_3$ passes all statistical tests in the test suite.

The $\mathcal{L}_4^{(i)}$ sequences generally show similar statistical properties. Some of the differences include $\mathcal{L}_4^{(1)}$ and $\mathcal{L}_4^{(4)}$ failing more tests than other segments. The failed tests include non-overlapping template matching tests, which try to find occurrences of predetermined target strings, and random excursion tests, which test cumulative sum random walks. Also of note is that because of the size of the test data, *Maurer's "Universal Statistical" test* was not applicable to the $\mathcal{L}_4^{(i)}$ segments. $\mathcal{L}_4^{(5)}$, $\mathcal{L}_4^{(6)}$ and $\mathcal{L}_4^{(7)}$ are close to indistinguishable from $\mathcal{L}_3$, which is why they were not included in the tables.

5.1.3 Output transformations

A test was carried to examine the effects of applying a function to each word generated by an LRS on its statistical properties. Let $f(\mathcal{L})$ denote the pseudo-random number sequence obtained by applying a function $f(w)$ to the sequence generated by the LRS $\mathcal{L}$.

As a base sequence, the $\mathcal{L}_3$ generator from subsection 5.1.2 was used.

The following transformation functions were used in the test:

- $f_l(w)$: For each word w in $\mathcal{L}$, $f_l(w)$ is the least significant bit of w.
- $f_m(w)$: For each word w in $\mathcal{L}$, $f_m(w)$ is the most significant bit of w.
- $f_p(w)$: For each word w in $\mathcal{L}$, $f_p(w)$ is the even parity bit of w.
- $f_s(w)$: For each word w in $\mathcal{L}$, $f_s(w) = w^2 \ mod \ 2^l$, where l is the length of words generated by $\mathcal{L}$.
- $f_n(w)$: Let w' be the word generated by $\mathcal{L}$ directly before w, or $w' = 0$ if w is the first word generated by $\mathcal{L}$. Then, $f_n(w) = w + w' \ mod \ 2^l$, where l is the length of words generated by $\mathcal{L}$.
- $f_c(w)$: Let $len(w)$ be the length of w. Let $l = len(w)$ if $len(w)$ is odd, and $l = len(w) - 1$ otherwise. Let $w[a : b]$, $a \geq b$, denote the bits of word w with indices in range $[a, b]$, $a, b \in \{0, 1, \ldots, l - 1\}$.

  For each word w in $\mathcal{L}$:

$$f_c(w) = \begin{cases} w[\lfloor \frac{l}{2} \rfloor : 1] & if \ w \ mod \ 2 \ = 0 \\ w[l - 1 : \lfloor \frac{l}{2} \rfloor + 1] & otherwise \end{cases}$$

- $f_t(w)$: Let l be the following:

$$l = \begin{cases} len(w) & if \ len(w) \ mod \ 3 \ = 1 \\ len(w) - 1 & if \ len(w) \ mod \ 3 \ = 2 \\ len(w) - 2 & if \ len(w) \ mod \ 3 \ = 0 \end{cases}$$

  For each word w in $\mathcal{L}$:

$$f_t(w) = \begin{cases} w[2\lfloor \frac{l}{3} \rfloor : \lfloor \frac{l}{3} \rfloor + 1] & if \ w \ mod \ 2 \ = 0 \\ w[l - 1 : 2\lfloor \frac{l}{3} \rfloor + 1] & otherwise \end{cases}$$

Tables 5.6 to 5.9 in the Appendix show some of the result obtained from the tests. The full report can be found at
https://arato.inf.unideb.hu/major.sandor/comparison_transform/.

The transformed generators generally show high quality randomness and uniform distribution. The one exception is $f_s(\mathcal{L}_3)$, which fails almost all statistical tests, and is completely unsuitable for practical purposes.

Due to the size of the transformed test data, some tests were not applicable to all of the generators.

Notably, $f_l(\mathcal{L}_3)$ and $f_p(\mathcal{L}_3)$ failed a number of non-overlapping template matching tests, while $f_m(\mathcal{L}_3)$ passed almost all of them.

The most promising transformed generators were $f_n(\mathcal{L}_3)$ and $f_c(\mathcal{L}_3)$, both of which passed every statistical test, and $f_t(\mathcal{L}_3)$, which only failed the FFT test, which examines the peak heights in the Discrete Fourier Transform of the sequence.

5.2 Comparison to other types of sequences

In this section, we compare the statistical properties of the sequence previously obtained to sequences created using other, previously known methods. These sequences are of interest because they have provably favorable pseudorandom properties, using the measures of pseudorandomness introduced in [32].

The elements of these sequences were originally defined to have $\{-1, +1\}$ values, but this can be trivially transformed into a $\{0, 1\}$ binary sequence.

Four sequences have been implemented and tested using the same NIST test suite used in the previous section. The sequences are defined as follows:

- $\mathcal{L}_{Leg}$ [32]: Let p be a prime. Then, the sequence $\mathcal{L}_{Leg}(p)$ is

$$\mathcal{L}_{Leg}(p) = \left(\left(\frac{1}{p}\right), \left(\frac{2}{p}\right), \ldots, \left(\frac{p-1}{p}\right) \right),$$

  where $(\frac{n}{p})$ is the Legendre symbol.

- $\mathcal{L}_{Gy}$ [29]: Let p be a prime and g a primitive root modulo p. If $(a, p) = 1$, then let $ind\ a$ denote the modulo p index of a to the base g, such that

$$g^{ind\ a} \equiv a\ (mod\ p),\ 1 \leq ind\ a \leq p-1.$$

  Let $N = p - 1$. Then, the sequence $\mathcal{L}_{Gy}(N) = \{e_1, e_2, \ldots, e_N\}$ is

$$e_n = \begin{cases} +1 & if\ 1 \leq ind\ n \leq (p-1)/2 \\ -1 & if\ (p+1)/2 \leq ind\ n \leq p-1. \end{cases}$$

- $\mathcal{L}_{GMS}$ [30]: Let p be a prime and $f \in \mathbb{F}_p[x]$ a degree $k > 0$ polynomial that has no multiple zeroes in $\overline{\mathbb{F}_p}$, where $\overline{\mathbb{F}_p}$ is the algebraic closure of $\mathbb{F}_p$.

Then, the sequence $\mathcal{L}_{GMS}(p) = \{e_1, e_2, \ldots, e_p\}$ is

$$e_n = \begin{cases} \left(\frac{f(n)}{p}\right) & if\ (f(n), p) = 1 \\ +1 & if\ p|f(n). \end{cases}$$

- $\mathcal{L}_{MS}$ [31]: Let p be a prime and $f \in \mathbb{F}_p[x]$ a degree $k > 0$ polynomial that has no multiple zeroes in $\overline{\mathbb{F}_p}$. For $(a, p) = 1$, let a^{-1} denote the multiplicative inverse of a, such that $aa^{-1} \equiv 1\ (mod\ p)$. Let $r_p(n)$ denote the smallest non-negative residue of n modulo p.

  Then, the sequence $\mathcal{L}_{MS}(p) = \{e_1, e_2, \ldots, e_p\}$ is

$$e_n = \begin{cases} +1 & if\ (f(n), p) = 1\ and\ r_p(f(n)^{-1}) < p/2 \\ -1 & if\ either\ (f(n), p) = 1\ and\ r_p(f(n)^{-1}) > p/2,\ or\ p|f(n). \end{cases}$$

To construct these sequence, a prime p is required. The sequences $\mathcal{L}_{GMS}$ and $\mathcal{L}_{MS}$ also require a polynomial. In [30], an algorithm is given to create suitable polynomials to be used in these sequences.

For the tests, a random 2048 bit prime number p_1 was generated, along with a polynomial $f \in \mathbb{F}_{p_1}[x],\ deg(f) = 126$, using the algorithm defined in [30]. The prime p_1 was used to construct the sequences $\mathcal{L}_{Leg}$, $\mathcal{L}_{GMS}$ and $\mathcal{L}_{MS}$. The polynomial f was used in both $\mathcal{L}_{GMS}$ and $\mathcal{L}_{MS}$.

A 32 bit prime number p_2 was generated and used in $\mathcal{L}_{Gy}$. Larger prime numbers made the computation required to calculate the elements of $\mathcal{L}_{Gy}$ impractically slow. Note that computing an element of $\mathcal{L}_{Gy}$ requires solving a discrete logarithm problem.

For each of these four sequences, 2^{27} bits of test data were generated, equal to the amount created for the sequences examined in the previous section.

Some of the results of the NIST tests for these four sequences can be found in Tables 5.10 and 5.11 in the Appendix. The full test reports can be found at *https://arato.inf.unideb.hu/major.sandor/comparison_other_types/*.

The sequences showed statistical properties very similar to the sequences presented in the previous section, passing all relevant tests and producing high-quality random bitstreams.

For practical purposes, these sequences have shown both advantages and disadvantages compared to the sequences created by the algorithm analyzed in this thesis.

One big practical advantage of the sequences presented in this section is the ease with which they can be constructed. As mentioned previously, they only

require a prime number, and for $\mathcal{L}_{GMS}$ and $\mathcal{L}_{MS}$, a polynomial that fulfills a condition that is easily satisfied. As the previous chapters of the thesis have shown, the sequences created by the presented algorithm are significantly more difficult to construct for degrees that are useful in practice.

However, computing new elements of the sequences $\mathcal{L}_{Gy}$, $\mathcal{L}_{GMS}$ and $\mathcal{L}_{MS}$ can become costly if the prime p or the field $\mathbb{F}_p$ becomes large enough. This could become a potential disadvantage in situations where a large amount of random data is required quickly. This problem is made worse by the fact that the sequences create data a single bit at a time. The sequences created by the previously presented algorithm avoid this issue by having arbitrarily sized elements and by being easily computable using LFSRs.

Also of note is that the four sequences shown in this section have a period length equal to p or $p - 1$. If the prime number p is chosen to be small, for example, to avoid the previously mentioned issues, the resulting small period length may become a problem in some practical applications. One of the main motivations behind the algorithm analyzed in the previous chapters of the thesis was the extremely large period length of the sequences that can be created.

Part II

Linear Code Generation

Chapter 6

Basic concepts

In this chapter, we introduce the concepts of binary linear codes. Most of the definitions and naming conventions are taken from [34]. In practice, these codes are most commonly used for their ability to detect and correct errors. The idea of error correcting codes first arose in the 1940s after Shannon's channel capacity theorem and noisy-channel coding theorem, which showed the theoretical limits of sending data through a noisy medium [35] [36]. This was followed by the invention of block codes such as the Hamming-code [37], Reed-Solomon code [38], Bose–Chaudhuri–Hocquenghem (BCH) codes [39] [40] in the 1950s, and convolutional codes and the Viterbi algorithm in the 1960s [41]. Until the 1970s, the most common use of error correcting codes was in military communication and other systems that required high reliability in the messages sent arriving at their destination without alterations. Later, they became widely used commercially with the spread of telecommunication and digital communication.

6.1 Codes

Figure 6.1 shows the basic model of a communication system. An information source sends a message u through a noisy channel. Before transmission, the message is encoded into a codeword c. At the destination, an encoded message v is received, which is potentially different than c. The received encoded message v is then decoded into the message u'.

Definition 6.1.1 (Source alphabet, message, codeword, code alphabet)**.** Let the messages $u = (u_1, u_2, \ldots, u_k)$ and $u' = (u'_1, u'_2, \ldots, u'_k)$ be length k vectors,

Figure 6.1: Abstract model of a communication system

$u_i \in F,\ i \in \{1, \ldots, k\}$. Then, F is named the *source alphabet*. The message u is encoded into a length n vector, $c = (c_1, c_2, \ldots, c_n)$, called a *codeword*, where $c_i \in Q,\ i \in \{1, \ldots, n\}$. Then Q is named the *code alphabet*.

The vector $v = (v_1, v_2, \ldots, v_n),\ v_i \in Q,\ i \in \{1, \ldots, n\}$ received at the destination is also a vector of length n of symbols from the code alphabet. Crucially, since the channel is noisy, $c \neq v$ may occur.

Definition 6.1.2 (Hamming distance)**.** Let $d(c, v)$ denote the number of coordinates i where $c_i \neq v_i$. Then, $d(c, v)$ is named the *Hamming distance* of vectors c and v.

Definition 6.1.3 (Minimum distance)**.** Given a code C, let d_{min} be:

$$d_{min} = \min_{\substack{c \neq c' \\ c, c' \in C}} d(c, c').$$

Then, d_{min} is the *minimum distance* of code C.

Definition 6.1.4 (Block code, block length)**.** $C \subseteq Q^n$ is called a *block code*. The elements of C are codewords, with *block length* n.

For the rest of the thesis, if there is no need for disambiguation, *code* will refer to block codes.

Definition 6.1.5 (Hamming weight)**.** The *Hamming weight* of a binary codeword $c \in \{0, 1\}^n$, denoted $w(c)$, is the number of coordinates $i \in \{1, 2, \ldots, n\}$ for which $c_i = 1$.

The Hamming weight of a code C is the smallest non-zero Hamming weight of the codewords of C.

Throughout the thesis, if not stated otherwise, the *weight* of a codeword will refer to Hamming weight.

Theorem 6.1.1 (Singleton bound)**.** [42] Let $C \subseteq Q^n$ be a code with minimum distance $d_{min},\ |C| = M$, and let $|Q| = q$. Then the following bound holds:

$$M \leq q^{n - d_{min} + 1}.$$

Definition 6.1.6 ((n,k) code)**.** In practice, it is often the case that $|F| = |Q| = q$ and $M = q^k$. This means that the code C can encode any length k message into a length n codeword. Such codes are named *(n,k) codes.*

For an (n, k) code, the Singleton bound is simply $d_{min} \leq n - k + 1$.

Definition 6.1.7 (Binary code)**.** A code $C \subseteq Q^n$ where $Q = \{0, 1\}$ is called a binary code.

6.2 Linear Codes

Definition 6.2.1 (Linear code)**.** A code $C \subseteq Q^n$ is linear if, for every codeword $c, c' \in C,\ c + c' \in C$ also holds. In other words, C is a subspace of Q^n.

Throughout the thesis, if there is no need for disambiguation, *linear code* will refer to binary linear codes.

For binary linear codes, the addition of vectors is done over $GF(2)$, that is, if $c = a + b,\ a, b, c \in \{0, 1\}^n$, then $c_i = a_i\ XOR\ b_i$.

It follows from the definition, that the codeword with all-0 bits, denoted **0** is a codeword of all linear codes.

Linear codes are used as error-correcting codes, as they are capable of detecting and correcting a certain number of errors that can happen during the transmission of a message.

Definition 6.2.2 ((n,k,d) code)**.** A binary (n, k) linear code with minimum distance d is named an *(n,k,d) code.*

Note that the minimum distance d of an (n, k, d) linear code is equal to the smallest non-zero weight of a codeword in the code. Minimum distance and minimum weight are used interchangeably for linear codes.

Definition 6.2.3 (Subcode)**.** Let C_1 be an (n, k_1, d_1) linear code and C_2 be an (n, k_2, d_2) linear code. C_2 is a *subcode* of C_1, denoted $C_2 \subset C_1$, if all codewords of C_2 are also codewords of C_1.

If $C_2 \subset C_1$, then $k_2 < k_1$ and $d_2 \geq d_1$.

Definition 6.2.4 (Weight distribution, weight enumerator)**.** The *weight distribution* of a linear code C is the sequence of integers $A_{Ci}, i \in \{0, 1, \ldots, n\}$, defined the following way:

$$A_{Ci} = |\{w(c) = i | c \in C\}|.$$

That is, A_{Ci} denotes the number of codewords in C with weight i.

The *weight enumerator* of a linear code C is the polynomial $W_C(x, y)$ defined using the weight distribution in the following way:

$$W_C(x, y) = \sum_{i=0}^{n} A_{Ci} x^{n-i} y^i.$$

An (n, k, d) linear code is always a k-dimensional subspace of the n-dimensional vector space of binary vectors.

Definition 6.2.5 (Linear independence)**.** Let C be an (n, k, d) linear code. Codewords $g_1, g_2, \ldots, g_j \in C$ are *linearly independent* if

$$\sum_{i=1}^{j} \alpha_i g_i = \mathbf{0},\ \alpha_i \in \{0, 1\}$$

only holds for $\alpha_i = 0$ for all $i \in \{1, 2, \ldots, j\}$.

Definition 6.2.6 (Basis)**.** Codewords $g_1, g_2, \ldots, g_k \in C$ form a *basis* of linear code C if they are linearly independent, and all $c \in C$ can be written in the form:

$$c = \sum_{i=1}^{k} u_i g_i,\ u_i \in \{0, 1\}.$$

In other words, all codewords can be written as a linear combination of the base codewords.

Note that the basis of a linear code is not unique. Any k linearly independent codewords from C form a basis of C.

Definition 6.2.7 (Generator matrix)**.** Let C be an (n, k, d) linear code, and $g_1, g_2, \ldots, g_k$ a basis of C. Because the base vectors of C are linearly independent, the way to write codewords as a linear combination of base codewords is always unique. Equivalently, it can also be written as:

$$c = uG$$

where $u = (u_1, u_2, \ldots, u_k) \in \{0, 1\}^k$ and G is the $k \times n$ matrix whose rows are the base vectors g_i:

$$G = \begin{pmatrix} g_1 \\ g_2 \\ \vdots \\ g_{k-1} \\ g_k \end{pmatrix}.$$

Matrix G is then called a *generator matrix* of C.

An (n, k, d) linear code encodes length k message into length n codewords. In practice, it often does not matter which message is encoded into which specific codeword, that is, the code is only regarded as the set of codewords. In this case, two generator matrices can be regarded as generating the same linear code if they generate the same set of vectors.

Definition 6.2.8 (Systematic code, standard generator matrix)**.** A generator matrix of the form

$$G = (I_k|A)$$

is said to be in *standard form*, where I_k is the $k \times k$ identity matrix, and A is a $k \times (n - k)$ matrix.

A linear code generated by a generator matrix that is in standard form is called *systematic*.

Let C be a systematic linear code generated by a standard generator matrix G. Then, the codewords $c \in C$ all have the following form:

$$c = (u_1, u_2, \ldots, u_k, c_{k+1}, c_{k+2}, \ldots, c_n).$$

In other words, the first k bits of the codeword are the message, followed by $n - k$ other bits.

Definition 6.2.9 (Permutation equivalence)**.** Let C_1 and C_2 be (n, k, d) linear codes. Let $p_\sigma(c)$ denote permuting the elements of a codeword c according to a permutation σ of coordinates $\{1, 2, \ldots, n\}$. C_1 is *permutation equivalent* to C_2 if there exists a σ permutation of coordinates $\{1, 2, \ldots, n\}$ such that:

$$C_2 = \{p_\sigma(c)|c \in C_1\}.$$

In other words, two linear codes are permutation equivalent if a permutation exists that transforms the codewords of one code into the codewords of the other. It is clear that permutation equivalence is reflexive, symmetric and transitive, that is, it is an equivalence relation.

In practice, permutation-equivalent linear codes are often considered essentially the same since permuting the coordinates of a code does not change its most interesting properties, such as its minimum distance or its weight distribution.

Given a generator matrix G that generates a linear code C, we can obtain codes that are permutation equivalent to C by permuting the columns of G.

For any generator matrix G, generating a linear code C, it is possible to use column permutations and Gaussian elimination to create a generator matrix G' that is in standard form and generates a systematic linear code C' that is permutation-equivalent to C. In other words, all linear codes are permutation-equivalent to at least one systematic linear code.

Definition 6.2.10 (Automorphism group)**.** The *automorphism group* of an (n, k, d) linear code C, denoted $Aut(C)$ is the set of permutations of coordinates $\{1, 2, \ldots, n\}$ that do not change C:

$$Aut(C) = \{\sigma | \sigma \in S_n,\ \sigma(C) = C\}.$$

Definition 6.2.11 (Dot product)**.** For binary codewords $a, b \in \{0, 1\}^n$, the *dot product* is the following:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i$$

where addition and multiplication is done over $GF(2)$.

Definition 6.2.12 (Orthogonal codewords)**.** Binary codewords $a, b \in \{0, 1\}^n$ are *orthogonal*, denoted as $a \perp b$, if $a \cdot b = 0$.

Equivalently, a and b are orthogonal if the number of coordinates $i \in \{1, 2, \ldots n\}$ for which $a_i = b_i = 1$ is even.

Definition 6.2.13 (Dual code)**.** Let C be an (n, k, d) linear code. Then, the *dual code* of C, denoted $C^\perp$, is the linear code of all codewords that are orthogonal to every codeword in C:

$$C^\perp = \{d | d \in \{0, 1\}^n,\ \forall c \in C : d \perp c\}.$$

The dimension of the dual code is always $n - k$.

If $G = (I_k | A)$ is a standard generator matrix, generating a linear code C, then $G' = (A^T | I_{n-k})$ generates $C^\perp$.

Definition 6.2.14 (Self-orthogonal code)**.** Let C be a linear code such that the following holds:

$$\forall c_1, c_2 \in C : c_1 \perp c_2.$$

Then, C is called *self-orthogonal*.

Definition 6.2.15 (Self-dual code)**.** Let C be an (n, k, d) linear code. C is called a *self-dual code* if $C = C^\perp$.

Self-dual codes always have the following properties:

- The block length n of a self-dual code is always even, and the dimension is always $k = \frac{n}{2}$.
- The codeword with all-1 bits, denoted **1** is a codeword of all self-dual codes.
- It follows from the previous item that the weight distribution of a self-dual code is symmetrical. In other words, if C is an (n, k, d) self-dual code, then $A_{Ci} = A_{Cn-i}$ for all $i \in \{0, 1, \ldots, \frac{n}{2}\}$.
- All subcodes of a self-dual code are self-orthogonal.

Definition 6.2.16 (Singly even codeword, doubly even codeword)**.** A codeword $c \in \{0,1\}^n$ is called *singly even* if $2|w(c)$ and $4 \nmid w(c)$. A codeword $c \in \{0,1\}^n$ is called *doubly even* if $4|w(c)$.

Definition 6.2.17 (Type I code, singly even code)**.** [43] An (n, k, d) linear code C is called a *Type I* code if it is self-dual, all codewords $c \in C$ have even weight, and at least one codeword has a singly even weight. Type I codes are also called *singly even codes*.

The weight distribution of a Type I code is such that the number of singly even codewords and doubly even codewords is always equal. In other words, a Type I (n, k, d) code always contains 2^{k-1} singly even codewords and 2^{k-1} doubly even codewords.

Type I codes exist for all even block lengths.

Definition 6.2.18 (Type II code, doubly even code)**.** [43] An (n, k, d) linear code C is called a *Type II* code if it is self-dual and all codewords $c \in C$ have doubly even weight. Type II codes are also called *doubly even codes*.

Type II codes exist only if the block length is divisible by 8.

For the rest of the thesis, subcodes of Type I and Type II codes will also be referred to as Type I and Type II, or singly even and doubly even, to emphasize the possible weights of such codes.

6.3 The problem of finding linear codes

Since the beginning of Coding Theory, the practical question of searching for linear binary codes has been given considerable attention by researchers [44].

Nowadays, one of the most important areas where such codes are used is Code-based Cryptography. Certain cryptographic schemes using these codes are promising candidates in Post-Quantum Cryptography, that is, cryptography believed to be secure against attack using quantum computers [45].

Due to their specific properties as described in the previous section, self-orthogonal and self-dual codes enjoy special attention in Coding Theory.

A central problem of finding linear codes is that for a given code length n, the question of whether or not codes with certain (n, k, d) parameters exist cannot be answered in general. No known universal algorithm exists that can construct any possible linear code with reasonable efficiency, nor are there any theoretical results that can answer if a linear code exists with given arbitrary parameters.

Clearly, a naive approach to generating linear codes would start becoming computationally infeasible even for very small codes. As an example, the generator matrix of a linear code with block length $n = 64$ and dimension $k = 32$ contains 2048 bits. If we constrain the generator matrix to be in standard form, then half of these bits would form the identity matrix I_{32}, still leaving 1024 free bits. Searching for a code with specific properties in the space of 2^{1024} possible binary matrices cannot be accomplished by brute force alone.

In practice, a number of different approaches and methods exist for constructing linear codes, but all of them are only efficient or capable of constructing certain specific families of codes. A description of commonly used methods can be found in [46].

One approach to creating new linear codes is to modify already known linear codes, constraining the space of possibilities by providing a starting point to our search. If the code we begin with has some favorable properties, we can hope to propagate these properties to other codes. Methods for combining codes to obtain new ones can be found in [47] and [48].

The following six basic methods are typically used when modifying linear codes:

- Augmenting: Given an (n, k, d) linear code, we construct a $(n, k+l, d')$ linear code, $l \geq 1$. The dimension of the code is increased, but the minimum weight can at best stay equal, that is, $d' \leq d$.

- Expurgating: Given an (n, k, d) linear code, we construct a $(n, k-l, d')$ linear code, $l \geq 1$. This operation can be seen as the reverse of augmentation, as it decreases the dimension of the code. The minimum weight of the subcode can be greater than the original code, $d' \geq d$.

- Extending: Given an (n, k, d) linear code, we construct a $(n+l, k, d')$ linear code, $l \geq 1$. The block length of the code is increased, and the minimum weight can also increase, $d' \geq d$.

- Puncturing: Given an (n, k, d) linear code, we construct a $(n-l, k, d')$ linear code, $l \geq 1$. As the reverse of extending, puncturing erases certain coordinates from a code. The minimum weight can decrease, $d' \leq d$.

- Lengthening: Given an (n, k, d) linear code, we construct a $(n+l, k+l, d')$ linear code, $l \geq 1$. Both the block length and the dimension of the code is increased by the same amount, the value of $n-k$ remains the same. The minimum weight of a lengthened code can be greater, equal or smaller than the original code.

- Shortening: Given an (n, k, d) linear code, we construct a $(n-l, k-l, d')$ linear code, $l \geq 1$. The reverse operation of lengthening, n and k decrease such that $n-k$ remains the same. The minimum weight can change in either direction.

The specific goal of finding linear codes can take multiple forms.

One common goal is the *classification* of linear codes. By classifying linear codes of certain (n, k, d) parameters, we want to find codes $C_1, C_2, \ldots, C_m$ such that for all $i, j \in \{1, 2, \ldots, m\}, i \neq j$, C_i and C_j are not permutation equivalent, and any (n, k, d) linear code is permutation equivalent to exactly one code on the list. In other words, we wish to find exactly one "representative" code from each permutation equivalent set of codes that fulfill the given criteria.

The challenges of classification include obtaining an exhaustive list of representative codes, and also showing that we have truly obtained an exhaustive list.

In the case of self-dual codes with block length n, the following mass formula can be used to determine if a list of $C_1, C_2, \ldots, C_m$ representatives is exhaustive [49]:

$$\sum_{i=1}^{m} \frac{n!}{|Aut(C_i)|} = \prod_{i=1}^{\frac{n}{2}-1} (2^i + 1).$$

The complete classification of self-dual codes is known up to at least $n = 40$ [50], with partial results known for higher values of n.

Another possible goal can be finding linear codes with previously unknown (n, k, d) parameters. Surprisingly, even for relatively small values of these parameters, there are codes whose existence or non-existence is unknown.

Remark 4 (Upper bounds on d)**.** For self-dual codes, some bounds on the possible values of d are known [43]:

- If $n \not\equiv 22 \mod 24$, then $d \leq 4\lfloor \frac{n}{24} \rfloor + 4$.
- If $n \equiv 22 \mod 24$, then $d \leq 4\lfloor \frac{n}{24} \rfloor + 6$.
- For Type I codes, if $n \equiv 0 \mod 24$, then $d \leq 4\lfloor \frac{n}{24} \rfloor + 2$.

Definition 6.3.1 (Extremal code, optimal code)**.** If C is an (n, k, d) code, where d reaches the upper bound shown in Remark 4, then C is called *extremal*.

If d does not reach this upper bound, but there does not exist a code with the same length and dimension which would have a higher minimum distance, then C is called *optimal*.

There are several open research questions regarding the existence of certain extremal and optimal self-dual codes that have not been found for decades, including:

- The existence of $(56, 28, 12)$ Type I codes [51].
- The existence of $(72, 36, 16)$ Type II codes [52].

The continued search for these codes have resulted in multiple other families of codes being found [53], [54], but the original questions remain unanswered to this day.

The minimum distance is the most common criterion when searching for codes, along with type, orthogonality, or being self-dual, but other conditions can be important as well. Any number of other properties can be of interest, both for practical or theoretical purposes, including weight distribution, the size and composition of the code's automorphism group, the number of codewords of a certain weight, the inclusion or exclusion of certain codewords or subcodes, and so on.

Codes fulfilling these special criteria can then be used as starting points for other methods of creating linear codes, as described previously. As an example, [55] gives a way to construct a $(72, 36, 16)$ Type II code, given a $(56, 21, 16)$ self-orthogonal doubly even code that contains the $\mathbf{1}$ codeword, which to date has also not been found. These searches require specific knowledge related to the properties of the code in order to be efficient.

The following chapters present a framework and software implementation developed by the author, designed to be capable of all of the above types of searches for binary linear codes, with emphasis placed on flexibility to adapt the search to arbitrary conditions.

Chapter 7

Torch software package

7.1 Introduction

As described previously, finding linear codes with certain parameters and properties can be challenging. The work presented in this and the following chapters was motivated by the need to handle the challenges that arise during research related to the questions described in the previous chapter. Whether it needs codes with specific properties for certain experiments, trying to find examples or counter-examples for theorems or hypotheses, or classifying certain classes of codes, different approaches are required to find the needed linear codes. A practical need arises for a framework that is adaptable to almost arbitrary conditions, has an architecture that is easily extensible, can incorporate new research results, and enables rapid development and experimentation.

Searching for $(56, 28, 12)$ Type I and $(72, 36, 16)$ Type II codes were some of the often recurring research goals during the continual development of the framework. These searches motivated many of the incorporated solutions, but at the time of writing, the codes remain to be seen only in dreams.

The framework, design and software implementation presented is the original work of the author of the thesis. Many of the theoretical results incorporated are the result of joint research with Carolin Hannusch.

The framework is named Torch.

7.2 Implementation details

7.2.1 Dependencies

Torch uses multiple mathematical software to make use of efficient implementations of basic operations. The current implementation has the following software as dependencies:

- **SageMath** [56]: SageMath, often referred to as Sage, is an open-source, free software system licensed under the GNU General Public License (GPL). The biggest advantage of SageMath is that its library incorporates a large and ever-growing number of other free, open-source mathematical software packages. At the time of writing, the number of packages included is nearly 100, according to Sage's website. In this way, it aims to provide a single, integrated access point to all the freely available most efficient implementations related to the covered fields of mathematics.

  The unified interface SageMath provides to its included packages uses syntax that is built on the Python programming language. This allows for a simple way to develop software that can make use of multiple very specialized, purpose-build packages that would otherwise require a significant amount of effort to use through their disparate interfaces.

  Version 9.x is used in the current implementation of Torch.

- **The GAP Guava package** [57]: Groups, Algorithms and Programming (GAP) is a free software system for computational discrete algebra, also licensed under GPL. GUAVA is a GAP package for algorithms related to coding theory. Although GUAVA is not included in SageMath by default, it can be added as an optional package.

  Torch makes use of GUAVA for certain operations that are more efficiently implemented than the ones included in the default packages of SageMath, such as calculating the minimum weight of linear codes.

- **Magma** [58]: The Magma Computational Algebra System is distributed under a proprietary license. Magma features its own programming language, which provides a very rigorous environment for working with mathematical objects.

  Magma implementations of algorithms are very highly optimized, and typically display the best runtime performance between the software described in this section.

Because of its proprietary license, Magma is not part of SageMath, but an interface is available in Sage that can connect to a Magma installation if it is present, allowing it to be used through the Sage environment.

7.2.2 Using Torch

Torch is implemented as a Python package using Python 3.x. It can be used from inside the SageMath interactive environment or SageMath scripts. For testing, portability, and isolation purposes, the experiments presented in the rest of the thesis were run inside a Linux virtual machine.

The top-level function of the linear code search algorithm is `get_code()`. Other top-level functions in the package include wrappers and utilities for various convenience purposes, such as type conversion, data formatting, calculating various properties of linear codes, and so on.

The `get_code()` function is implemented as a Python generator. Generator functions are iterators that can be used to create a sequence of values one at a time without having to create and store the entire collection of values at the same time. Each function call to a generator returns one value. The internal state of the generator is stored, and the next function call to the same generator will continue the execution from that state, generating the next value. In this way, a generator can be used to exhaustively iterate through an entire sequence or stop the execution after the desired number of return values have been obtained.

The return values of the `get_code()` function are objects that belong to the class `sage.coding.linear_code.LinearCode`.

By default, `get_code()` both logs extensive information about the execution of the search and persistently saves all returns values.

The most commonly used arguments of `get_code()` are the following:

- Block length n and dimension k are required arguments. These are arbitrary integer values.
- The minimum distance d is likewise an integer value. If no d is given, then it defaults to a value dependent on the type of code we are searching for: $d = 1$ for non-orthogonal codes, $d = 2$ for singly even codes, $d = 4$ for doubly even codes.
- The *type* parameter denotes the type of code being search for, $type = 1$ denoting singly even codes, $type = 2$ denoting doubly even codes. By default, the value of this parameter is *None*, denoting non-orthogonal codes.

- The optional parameters *verbose* and *save* control the way the search algorithm logs its execution and saves the codes it finds. Some more detail about them can be found in the chapter detailing options.

- The *config* parameter provides the names of the configuration files used to determine the objects necessary for the execution of the search. These enable the search algorithm to be adapted to the various conditions required by the task or experiment. More details can be found in subsection 7.2.3. The current default value of this parameter is *config="torch4"*, pointing to the configuration file containing the dependency information of the default search algorithm.

- Any number of additional keyword arguments can be added to the function call. These additional arguments typically configure the lower level components of the search algorithm, as defined in the config files being used.

The following are a few examples of using the `get_code()` function to generate binary linear codes:

- Creating the Hamming(7,4) code. The Python `next()` function can be used to stop the execution after the first (and only) solution has been found.

  ```
  h = next(get_code(7, 4, 3))
  ```

  The generator matrix of the `LinearCode` object stored in `h` can be printed in Sage the following way:

  ```
  h.generator_matrix()
  [1 0 0 0 1 1 1]
  [0 1 0 0 1 1 0]
  [0 0 1 0 1 0 1]
  [0 0 0 1 0 1 1]
  ```

- Create a list of $(16, 8, 4)$ Type I codes, using Python's list comprehension:

  ```
  my_list = [code for code in get_code(16, 8, 4, 1)]
  ```

- Create the extended binary Golay code, without logging information about the execution:

```
g = next(get_code(24, 12, 8, 2, verbose=False))
```

- Create a list of $(22, 11, 6)$ Type I codes, with only one code from each permutation equivalent set of codes present on the list:

```
classification = [code for code in
    get_code(22,11,6,1,config=["torch4","perm_equiv"])]
```

- Create a $(56, 27, 12)$ Type II code, using a heuristic search, with a given heuristic function. For more details, see the chapter describing configurations.

```
c = next(get_code(56,27,12,2,config=["torch4","heuristic"],
    heuristics_list=["extremal_word_count_heuristic"]))
```

7.2.3 Software architecture

As mentioned previously, one important goal of Torch was to provide a solution to constructing search algorithms in a way that is flexible and easily extensible with new research results. In order to accomplish this, the framework uses a custom dependency injection system to assemble the objects needed for the search.

Dependency injection (DI) is an implementation of the software architectural pattern known as *inversion of control*. The goal is to create loosely coupled code, that is, code where the behavior of modules can be easily changed by switching out objects for other implementations that provide the same functionality. Architectures built this way are defined by the interfaces of the different components, which regulate how objects can communicate with each other.

Martin Fowler has talked about dependency injection in 2004 [59]. The dependencies of an object are the other objects it requires to carry out its actions, for example, the objects needed to execute a method call. An important question regarding an object's dependencies is the responsibility of instantiating them. Letting an object create its own dependencies leads to tightly coupled code that is difficult to reuse for new purposes.

Using dependency injection, this responsibility is assigned to a separate component of the software. This component is called the *DI container*, although "pure DI" solutions that do not use a container also exist. The DI container determines the appropriate dependencies an object needs, and injects them into it after instantiating them. The injection itself can be done in multiple ways,

such as through the constructor, through a setter method, or by using a method defined in an interface. Changing the behavior of a component using DI is accomplished by switching out the dependencies of the component to another implementation that fits the required interface. More information about the practices of dependency injection can be found in [60].

While many DI solutions already exist, during development the need arose for a custom system where configuration handling and object creation can be precisely controlled. The DI container implemented for Torch is a general purpose container, capable of instantiating and assembling any object described in Torch configuration files. The config files use the *json* format. Each json file contains an associative array, that is, a list of key-value pairs. The keys of the config files are strings which serve as the identifiers of the objects we wish to assemble. The value assigned to a key is also an associative array, which describes all the information needed to instantiate the object.

Each config file can also contain a key called `defaults`. The value assigned to this key is an associative array that contains the default values of the important parameters of the configuration.

As an example, listing 7.1 shows the config information of the default top-level searcher object.

```
"searcher": {
  "module": "torch.torchsearcher",
  "class": "TorchSearcher",
  "known_arguments": {
    "codeclass": "codeclass",
    "starters": "starters",
    "descendant_generator": "descendants",
    "saver": "saver",
    "logger": "logger"
  }
}
```

Listing 7.1: Example object description in Torch config file

The `module` and `class` keys in the above example determine the exact place of the class of the object in the package hierarchy of Torch. These keys are mandatory for an object description.

The `known_arguments` key is optional. The value assigned to this key is an associative array containing the explicitly given dependencies of the object.

Another optional key in the description of an object is `unknown_arguments`.

These arguments are the dependencies of the object that are not explicitly defined by the configuration.

The DI container tries to deduce the value of these arguments when it is called to instantiate the object. The final value of an unknown argument can be taken from two collections: the `defaults` provided by the configuration file, and the keyword arguments given to the DI container during the initialization of the container, which happens at every `get_code()` call. Priority is given to the keyword arguments of the DI container. This is the mechanism by which the keyword arguments given to `get_code()` find their way to the module that uses it.

All `unknown_arguments` fall into one of two groups of values: `requireds` and `optionals`. These groups are given in the description of an object as lists. Empty lists can be omitted. Arguments in the `requireds` list are necessary for the object to function. If the DI container cannot deduce a value for a required unknown argument, an error is thrown and the object fails to instantiate. Arguments in the `optionals` list are ones the object can function without. If the DI container does not find a value for an optional unknown argument, the object can still be created without error.

Listing 7.2 shows an object description with all optional keys present.

```
"object_identifier": {
  "module": "module.name",
  "class": "ClassName",
  "known_arguments": {
    "known_arg_a": "a",
    "known_arg_b": "b"
  },
  "unknown_arguments": {
    "requireds": ["required_arg_c","required_arg_d"],
    "optionals": ["optional_arg_e","optional_arg_f"]
  }
}
```

Listing 7.2: Example object description with all optional keys present

The value of an argument (known or unknown) can be either a literal of a primitive type, an expression that evaluates to a primitive type, an object defined in the configuration, or a list containing any combination of the previous.

If the value of an argument is a string, the DI container checks the following possibilities:

- The string matches an object identifier found in the configuration. For example, in listing 7.1, each value in the `known_arguments` points to another object also defined in the same config file. When evaluating such an argument, the DI container either creates and stores that object if it does not yet exist, or retrieves the object from its storage if it already does, and returns the object as the value of the argument.

- The string matches a variable name in the config `defaults` or the keyword arguments given to the DI container. In this case, the argument evaluates to the value found this way.

- If a string matches neither of the above, it is treated as a string literal.

In the case of list valued arguments, the above evaluation is carried out for each string on the list.

A config file contains the descriptions of all the objects needed to implement a certain algorithm, or incorporate a condition or research result into another algorithm. This gives the framework the flexibility to adapt to different search requirements.

To enable even further flexibility, and reduce duplication of common object descriptions in the configurations, the DI container allows multiple config files to be used together. If the DI container is initialized with a list of more than one config files, it first creates an *effective configuration* by starting out with the contents of the first file, and successively updating it with the contents of the subsequent files, in the order that they appear on the list. Object instantiation can only begin after the effective configuration has been determined.

If the same object identifier appears in multiple config files, then the descriptions of the object are merged together, with priority given to the information appearing in the file later in the list. As an example, listings 7.3 and 7.4 show two config files we wish to use together. If the files in this example are used in the order `config=["config1","config2"]`, then the effective configuration the DI container determines is seen in listing 7.5.

Listing 7.3: config1.json

```
"object_id": {
  "module": "old.module",
  "class": "OldClass",
  "known_arguments": {
    "dependency_a": "a",
    "dependency_b": "b"
  },
  "unknown_arguments": {
    "optionals": ["c","d"]
  }
}
```

Listing 7.4: config2.json

```
"object_id": {
  "class": "NewClass",
  "known_arguments": {
    "dependency_a": "x",
    "dependency_c": "y",
    "c": 496
  },
  "unknown_arguments": {
    "requireds": ["e"]
  }
}
```

Listing 7.5: Effective configuration of config1.json and config2.json

```
"object_id": {
  "module": "old.module",
  "class": "NewClass",
  "known_arguments": {
    "dependency_a": "x",
    "dependency_b": "b",
    "dependency_c": "y",
    "c": 496
  },
  "unknown_arguments": {
    "requireds": ["e"],
    "optionals": ["d"]
  }
}
```

The rules for merging object descriptions are as follows:

- The `module` and `class` keys are only mandatory to be defined in the final effective configuration. In other words, they can be omitted from an object description in a config file, but at least one config file containing that object must define them.
- The associative arrays for `known_arguments` and `unknown_arguments` are merged, with the arguments in the latter file overwriting the arguments

in the former file in the case of matches.

- In the case of an unknown argument in the former file matching a known argument in the latter file, the argument is promoted to being a known argument.
- An exception to the rule of the latter file taking precedence over the former file is the case of a known argument in the former file being defined as an unknown argument in the latter file. In this case, the argument remains a known argument, with the value defined in the former file. In other words, once the value of an argument is known, it can not become unknown again.
- In some cases, instead of merging two argument collections together, one collection needs to replace the other. One such case is removing unwanted arguments. The DI container recognizes special syntax to accomplish this.
- Similarly, in the case of adding new items to list valued arguments at a specific position in the list, special syntax is used.

Aside from assembling objects to initiate a search, another important use of the DI container is testing. The lower level components of a search can all be instantiated on their own, with any desired parameters and dependencies, making it convenient both for unit testing, to see how the objects behave in isolation, and for integration testing, where larger parts of the search algorithm are assembled together to test if the components work together as expected.

Because of the way multiple configurations can build on top of one another, one config file only needs to contains the objects and changes that apply to the specific algorithm that configuration is implementing. In practice, this typically leads to short, easy to understand configurations. In the current implementation of Torch, the base objects of the default search algorithm are all described in one default configuration, which all others add modifications to.

The combination of multiple configurations also naturally lends itself to experimentation, when the goal is to find the most efficient way to construct or classify linear codes. One note of caution though is that while configurations are created with modularity in mind, naturally not all algorithms can be mixed together in a meaningful way.

The algorithms and results presented in the following chapters of the thesis are all built using the tools and principles detailed in this chapter.

These tools allow a researcher or developer to easily create modules for experimenting or making use of their own research results which they wish to add to the framework of Torch.

Chapter 8

Search algorithms

In this chapter, the basic search algorithms used in the Torch package are described. Other search algorithms, implemented for more specific purposes, using configurations described in later chapters, are modifications or extensions of the ones presented here.

All of the algorithms build linear codes by augmentation, that is, the block length n of the code is fixed from the start, and new codewords with favorable properties are added to it until the code reaches the dimension k that is required.

Although the basic ideas of the search algorithms are very straight-forward, the nature of linear codes makes it necessary to weigh each practical decision carefully to find effective solutions to even minute sub-problems.

An in-depth discussion of the topic of generating combinatorial objects and the challanges of applying these algorithms to objects like linear codes can be found in [61]. The highlighted chapter focuses on isomorph-free generation, which is a very desirable property for classification, but for targeted searches, it can even be a disadvantage in certain cases. Configurations implementing isomorph-free generation algorithms are discussed in a later chapter. For now, the relevant insights of [61] are the following:

Some objects, like permutations, have a straightforward inductive structure, making it easy to create an exhaustive generation algorithm for them. Some objects, like linear codes of a given minimal distance, have nontrivial regularity properties. For these objects, search algorithms are implemented that have to search through a set of partial objects to find the ones that can be augmented step-by-step into the complete objects, and reject the partial objects that cannot lead to the complete objects.

The set of partial objects can be much larger than the set of complete objects, making naive approaches impractical and requiring significant insight into the structure of the objects to carry out the search efficiently. Consider that if the goal is to prove the non-existence of a certain object using these methods, then all possible ways of constructing such an object need to be exhausted.

To generate these types of objects, search trees are constructed, where nodes represent partial objects, connected by edges if one node can be created from the other using one augmentation step. The search can be thought of as the traversal of this tree.

In order to constrain the set of partial objects that need to be considered, the following principles are employed:

- Knowledge about the structure of the objects need to be incorporated. This knowledge can help define conditions about which partial objects are useful, and direct the order of the traversal.
- The number of child nodes created from one node should be minimized. Since the size of the search trees can be exceptionally gigantic, minimizing the number of branches that can emerge from a node is crucial.
- The search should be aborted on branches that cannot lead to a new solution as early as possible. Recognising which branches can be pruned in such a way again requires knowledge specific to the problem being considered.

The methods implemented in Torch were designed with the above principles in mind.

This chapter presents the following search algorithms, along with experimental results of their performance:

- The default search defined in Torch is a depth-first search algorithm. An overview is given of the most important components of the search, and the critical steps are described in detail.
- A clique-based search algorithm is described. In some scenarios, it is efficient to change a tree-based search into a different approach, based on finding cliques in a graph. In practice, this method is not typically used by itself, but as a hybrid with another tree-based algorithm.
- Heuristic searches are described. For some targeted searches, the use of heuristic algorithms to guide the traversal of the search tree can greatly improve performance. Specific heuristic functions, used to sort the fitness of linear codes for these searches, are also described.

8.1 Default algorithm

The default configuration defines a depth-first search. When searching for an (n, k, d) code, the starting node is a generator matrix for an $(n, 1, d')$ code, $d' \geq d$. The search tries to augment this matrix one row at a time, such that at a depth of $k' \leq k$, the current node will be a generator matrix for an (n, k', d'') code, $d' \geq d'' \geq d$. When a depth of k is reached, and all conditions set for the linear code are fulfilled, the code object is returned as a solution and the search continues.

To carry out the search, the algorithm needs to generate candidate codewords that could be appended to the current generator matrix. Each time a new candidate codeword is found, a list of conditions are checked, which are specific to the type and parameters of the linear code being searched. If all conditions are fulfilled, a new node in the search is created where the parent node's generator matrix has been augmented by the candidate codeword. The efficiency of the search greatly depends on how fast the candidate codewords can be created, and how many branches of the search tree can be pruned by the list of conditions.

To ensure that the default search is exhaustive, the conditions included are such that at least one linear code should be returned from each permutation equivalence class.

The following modules are important for instantiating the search algorithm:

- *weights*: Generator object that generates the permitted Hamming weight values for the codewords of a given code. This module is used in multiple other modules to determine possible weights.

- *mu_table*: This module stores tables of precomputed values that help generate candidate codewords. These values determine how the bits of two codewords are allowed to be arranged such that their sum is still a valid codeword for the type of code being searched for. More details about this module are found in subsection 8.1.1.

- *starters*: Generator object that generates the possible root nodes of the search. It yields $(n, 1, d')$ generator matrices. The number of possible starting matrices depend on the permitted Hamming weights as determined by the *weights* module, and other given search properties, such as if we wish the final generator matrices to be in standard form.

- *solutions*: Given an (n, k', d') generator matrix G, the *solutions* module generates the possible candidate codewords that can be appended to G.

The algorithm also checks a number of conditions during the intermediate steps of generating a codeword, in order to abandon incorrect prefixes.

In practice, the default configuration uses a *solutions* module that dynamically switches between two different implementations during the search, trying to determine which implementation leads to a better performance at any given node. These implementations, and the conditions used during the generation of codewords, are described later in this chapter.

- *children*: Given a node containing an (n, k', d') generator matrix, the *children* module generates the possible $(n, k' + 1, d'')$, $d \leq d'' \leq d'$, generator matrices that can be created by augmenting the input matrix with one new codeword. This module uses the *solutions* module to find the possible candidate codewords.

  The *children* module is also configured with a list of conditions that must be fulfilled in order to accept an augmented matrix as a new node. These conditions depend on the specific properties of the code being searched for. New conditions can be easily created by extending an abstract base class and adding the new class to the appropriate configuration file. Adding new conditions like this is a typical way of incorporating new research results in the search in order to speed up the algorithm.

  The conditions used by the default search and other algorithms in Torch are described in a later chapter.

- *descendants*: Given a node containing an (n, k', d') generator matrix G, this module generates the possible (n, k, d) matrices that can be created by augmenting G with $k - k'$ new codewords. This module uses the *children* module for the intermediate steps of the search. The default search is done by calling the *descendants* module using the root nodes returned by the *starters* module as inputs.

  Like the *solutions* module, the *descendants* module can also switch between implementations during the search when certain conditions are met, the two default implementations being the depth first search and the clique-based search that uses a graph that can improve on the performance of the tree-based search in certain scenarios.

As mentioned in the *children* module, another important component of searches are the condition objects, which check if a given code has a certain property. These conditions can be injected into modules in the same manner as any other dependency.

The Torch framework includes a base class for condition objects, which can be extended to create custom conditions. Torch condition objects also include the following features:

- Detecting if a condition is not mathematically relevant to a search and should not be included. For example, some implemented conditions that are part of the default configuration only apply to self-dual codes. If the search is for a code that is not self-dual, these condition objects are automatically omitted.
- Providing additional information about the result of checking a code.
- Appending multiple conditions after each other to form a list of conditions. In a condition list, all conditions need to pass in order to consider the list passed.

Using the modules described above, injected as dependencies as described in section 7.2.3, the generator method implementing the search algorithm can be simplified in the way shown in listing 8.1 in the Appendix.

In practice, the algorithm also includes modules that perform other tasks, like logging, which are not relevant to the algorithmic description of the search.

The *get_descendants()* method of the *descendants* module implementing the depth first search can be simplified in the way shown in listing 8.2 in the Appendix.

The *yield conditions* are the search specific properties we wish the code to fulfill in order to consider it complete. These conditions can be arbitrarily complex, or as simple as checking that the dimension of the code has reached the desired value of k.

As mentioned in the description of the *descendants* module, the implementation of this module used in the default configuration also uses the clique-based search described later in this chapter.

The generator matrices created by the default search algorithm have the following properties, which together will be referred to as *default form*.

Definition 8.1.1 (Default form, ordered form, weight sorted form)**.** A generator matrix G, generating an (n, k, d) linear code C, is in *default form*, if all three of the following hold:

- G is in standard form: $G = (I_k|A)$.

- G is in *ordered form*: Let $\hat{a}_i$ denote the ith column of A. For all $i, j \in \{1, 2, \ldots, n-k\}$, if $i < j$, then $\hat{a}_i \geq_l \hat{a}_j$, where $\geq_l$ denotes lexicographical order.

- G is in *weight sorted form*: Let g_i denote the ith row of G. For all $i, j \in \{1, 2, \ldots, k\}$, if $i < j$, then $w(g_i) \geq w(g_j)$.

Optionally, all of the above properties can be turned on or off in a search. The lexicographical order between the columns, and the numerical order between the row weights can also be changed. For more details, see the chapter describing options.

In practice, if the search is for generator matrices in standard form $G = (I_k|A)$, then the algorithm only builds and stores A, and extends the rows with the rows of I_k only when necessary.

All linear codes are permutation equivalent to at least one linear code that is generated by a generator matrix in default form. Therefore, an algorithm generating all default form generator matrices will create at least one linear code from all permutation equivalent sets of codes. In this and following chapters, we will see some methods built on the default algorithm whose aim is to reduce the number of permutation equivalent codes created.

8.1.1 Mu tables

The *mu_table* module creates a table of precomputed values before a search begins using the following function:

Definition 8.1.2 (μ function)**.** Let C be an (n, k, d) linear code, and $a, b \in C$. Let $\mu(a, b)$ be defined the following way:

$$\mu(a, b) = \left| \left\{ i | a_i = b_i = 1, i \in \{1, 2, \ldots, n\} \right\} \right|$$

That is, $\mu(a, b)$ is the number of coordinates i such that $a_i = b_i = 1$.

For any codewords $a, b \in C$, $w(a + b) = w(a) + w(b) - 2\mu(a, b)$.

Since the possible weights of a codeword $a + b$ are limited, it is possible to precompute the allowed values of $\mu(a, b)$ for any pair of codewords weights $w(a)$ and $w(b)$ such that the above equality holds.

For any linear code C, if $\mathbf{1} \in C$, which is true for all self-dual codes, then for all $c \in C$, except for $\mathbf{0}$ and $\mathbf{1}$, $d \leq w(c) \leq n - d$. This follows from the observation that for all $c \in C, w(c + \mathbf{1}) = n - w(c)$. Another consequence of

this is that if $\mathbf{1} \in C$, then the weight distribution of C is symmetrical, that is, $A_i = A_{n-i}$ for all $i \in \{0, 1, \ldots, \lfloor \frac{n}{2} \rfloor\}$.

The Mu table is used to help determine candidate codewords to augment a generator matrix with during a search. As an example, table 8.1 shows the precomputed table of μ-values built when searching for a $(56, 28, 12)$ Type I code using the default algorithm. Note that because the generator matrix G being built is in default form and only A in $G = (I_k|A)$ is stored, the maximum weight a row in A can have is 27. For each row of A, denoted a_i, and corresponding row of G, denoted g_i, $w(g_i) = w(a_i) + 1$.

If the length of a_i is r, then the value of $\mu(a, b)$ has to satisfy the following inequality:

$$w(a) + w(b) - r \leq \mu(a, b) \leq min\{w(a), w(b)\}$$

If C is a self-orthogonal linear code, then $\mu(a, b)$ is even for all $a, b \in C$.

Table 8.1 in the Appendix shows the allowed μ-values where $w(b)$ is the weight of the candidate codeword we wish to augment A with, and $w(a)$ is the weight of a row in A. When building a generator matrix this way, in order to accept b as a new row, it is a necessary, but not sufficient condition that for all a_i, $\mu(a_i, b)$ needs to be a value found in the appropriate cell of the Mu table.

In practice, the algorithm also needs to know the allowed μ-values between a new candidate codeword b, and a codeword a that is the sum of multiple rows of A. The reason for this is described in the following subsection. In this case, the values of the above table need to be adjusted. For this purpose, the *mu_table* module actually constructs multiple Mu tables to cover all such needs.

The following subsection describes using the Mu table to build a set of conditions that is used by the *solutions* module to generate candidate codewords.

8.1.2 μ-condition set

We wish to create a default form generator matrix $G = (I_k|A)$ generating an (n, k, d) linear code. During the execution of the algorithm, only the $k' \times n - k$ matrix A' is stored at each node. If $k' < k$, then the algorithm tries to find candidate codewords that can be used to augment A'. A' is assigned to be A when $k' = k$.

As defined previously, if G is in default form, then it is also ordered, meaning the columns of A are in lexicographic descending order. During the execution of the algorithm, the columns of A' must also keep this property. To achieve this, at the start of the algorithm and after each augmentation step, a set of

conditions are created such that candidate codewords that fail any of the conditions are rejected.

Let a_i denote the ith row of $A', 1 \leq i \leq k'$, r denote the number of columns of A', and $\hat{a}_i$ denote the ith column of $A', 1 \leq i \leq r$. We want to augment A' with a codeword b.

Define $L_1, L_2, \ldots, L_m$, $m \leq r$, to be sets such that

$$\bigcup_{i=1}^{m} L_i = \{1, 2, \ldots, r\}$$

and there exist border values $l_1, \ldots, l_m$ such that $1 \leq l_1 < l_2 < \ldots < l_{m-1} < l_m = r$, which separate the L_i sets the following way:

$$L_1 = \{1, \ldots, l_1\}$$

$$L_2 = \{l_1 + 1, \ldots, l_2\}$$

$$\vdots$$

$$L_i = \{l_{i-1} + 1, \ldots, l_i\}$$

for all $1 < i \leq m$.

These sets are chosen such that for any L_i, if $s, t \in L_i$ and $v \notin L_i$, then $\hat{a}_s = \hat{a}_t$, and $\hat{a}_v \neq \hat{a}_s$. In other words, each set L_i contains the indices of columns of A' that are identical to each other, and no columns with indices in different sets are identical. Note that these sets partition the set of indices in the way described above because the matrix A' is ordered.

Let $a_j^{L_i}$ denote row a_j being punctured to only the coordinates in L_i. Because of the way the L_i sets are defined, for all rows a_j and sets L_i, $a_j^{L_i}$ contains either all 1 bits or all 0 bits. Let $V(a_j) \subseteq \{1, 2, \ldots, m\}$ denote the set of indices i such that $a_j^{L_i}$ contains all 1 bits.

The process of finding a candidate codeword b with a given weight w_b means distributing w_b number of 1 bits among the available r coordinates. Let v_i denote the number of 1 bits assigned to the coordinates in L_i, $1 \leq i \leq m$. Let $Mu[w(a), w(b)]$ denote the allowed values of $\mu(a, b)$ as described by the Mu table. Then, the sum of all $v_i, i \in V(a_j)$ is going to equal $\mu(a_j, b)$. This value must be in $Mu[w(a), w(b)]$ for all rows a_j.

In summary, a valid candidate codeword b needs to fulfill all of the following conditions:

$$\sum_{i=1}^{m} v_i = w_b$$

$$0 \leq v_i \leq |L_i|, \quad 1 \leq i \leq m$$

$$\sum_{i \in V(a_j)} v_i \in Mu[a_j, b], \quad 1 \leq j \leq k'$$

Example 8.1.1. As an example, consider being in the process of building an $(20, 10, 4)$ Type I code. Let A' be the following matrix:

$$A' = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Here, $r = 10$ and $k' = 3$. The L_i sets are the following:

$$L_1 = \{1, 2, 3, 4\}, L_2 = \{5, 6\}, L_3 = \{7\}, L_4 = \{8\}, L_5 = \{9, 10\}$$

If we want to augment A' with a codeword b that has weight $w_b = 5$, then first we find a solution to variables $v_1, v_2, v_3, v_4, v_5 \in \mathbb{N}$ that meets the following conditions:

$$v_1 + v_2 + v_3 + v_4 + v_5 = 5$$

$$0 \leq v_1 \leq 4, \quad 0 \leq v_2 \leq 2, \quad 0 \leq v_3 \leq 1, \quad 0 \leq v_4 \leq 1, \quad 0 \leq v_5 \leq 2$$

$$v_1 + v_2 + v_3 \in \{2, 4\} \quad // \quad \mu(a_1, b) \in Mu[7, 5]$$

$$v_1 + v_2 + v_4 \in \{2, 4\} \quad // \quad \mu(a_2, b) \in Mu[7, 5]$$

$$v_1 + v_4 \in \{0, 2, 4\} \quad // \quad \mu(a_3, b) \in Mu[5, 5]$$

If a given $Mu[w(a), w(b)]$ cell of the Mu table is empty, that means the generator matrix can not have two lines with weights $w(a)$ and $w(b)$. A μ-condition that has an empty cell on the right side can not be satisfied. If such a condition is found, the algorithm constructing candidate codewords skips trying to create codewords with weight $w(b)$.

In this subsection so far, including the example above, we focused only on making sure that the candidate codeword b fits together with the rows of A' in such a way as to not violate the desired minimum distance d of the code we wish

to build. The μ-condition set described above is a necessary, but not sufficient set for this goal.

In practice, we can extend the set of conditions significantly. We can create analogous conditions between b and any codeword generated by A'.

Suppose we want to add an extra condition to the above set based on the relationship between b and $a = \sum_{i \in I} a_i, |I| > 1$. The L_i sets are valid for all codewords of A' and need no adjustment. The set $V(a)$ needed to know which variables v_j need to be added together to get $\mu(a, b)$ can be computed based on I: let the $\mathscr{L}$ be a multiset defined by the union $\mathscr{L} = \bigcup_{i \in I} L_i$. Then, $V(a)$ is the set of all elements of $\mathscr{L}$ that appear in $\mathscr{L}$ an odd number of times.

The Mu table also needs adjustment based on $|I|$. Let $Mu^{|I|}[w(a), w(b)]$ denote the Mu table adjusted to the μ-values allowed when comparing the candidate codeword b and a codeword a that is the sum of $|I|$ rows of A'.

Using these steps, we can create the condition $\mu(a, b) \in Mu^{|I|}[w(a), w(b)]$ for any codeword a generated by A'.

Example 8.1.2. We wish to extend example 8.1.1 with a new condition for the relationship between b and $a = a_1 + a_2$. Then, $V(a) = \{3, 4\}$ and $w(a) = 2$. The new condition is:

$$v_3 + v_4 \in \{0, 2\} \quad // \quad \mu(a, b) \in Mu^2[2, 5]$$

The number of μ-conditions created this way impact the performance of the search significantly. Having a significant number of μ-conditions helps the algorithm detailed in the following subsection to not waste time creating invalid candidate codewords that need to be rejected at a later step. On the other hand, if the number of conditions is too great, checking them becomes computationally heavy and can slow the search down. The Torch framework provides an option to set a limit to the number of μ-conditions the algorithm should create for each matrix it needs to augment.

The following subsection presents the algorithms used to generate the candidate codewords for augmenting a generator matrix with, using the the μ-condition set described in this subsection.

8.1.3 Candidate codeword generation

The default *solutions* module uses two separate algorithms for finding all codewords that satisfy a given set of μ-conditions. The first is a general algorithm that can be used for any linear code search, the second is an algorithm used only when searching for self-orthogonal codes. These algorithms are necessary since a

naive approach to generating candidate codewords would be impractically slow even for relatively small linear codes.

The first goal of the general algorithm is to find all possible evaluations of variables $v_1, v_2, \ldots, v_m \in \mathbb{N}$ given a set of μ-conditions as described in subsection 8.1.2, then construct the codewords using these variables and the L_i sets of indices.

The basic idea of the general algorithm is as follows: at the start, all variables v_i are unassigned. Given a partial evaluation of the variables, an iteration repeats the following steps: the v_i with the smallest index i that is currently unassigned is given a value between a predetermined upper and lower bound. The relevant μ-conditions are checked. If the new partial evaluation fails a given condition, the iteration moves on to the next possible value. Otherwise, a recursive function call repeats the algorithm with the new partial evaluation. If all v_i variables are assigned, passing all conditions, the evaluation is returned.

In this way, an incorrect prefix, that fails a necessary μ-condition, is immediately rejected before the algorithm wastes more time trying to finish it, effectively rejecting all codewords that begin with that prefix at once.

A simplified pseudocode of the algorithm can be seen in listing 8.3 in the Appendix. Note the following points about the algorithm:

- *Li* is a list, *Li[i]* containing the set L_i, $1 \leq i \leq m$.
- *wb* is the weight of the candidate codeword b we want to construct.
- *mu_conds* is an object containing the set of μ-conditions described in subsection 8.1.2.
- *ps* is a list of partial sums of $|L_i|$ that is precomputed before the recursion begins, calculated by the *get_partial_sums()* function. These values are needed to determine the correct lower limit a variable v_i can take. Specifically, $ps[i] = \sum_{j=i}^{m} |L_j|, L_i$, $1 \leq i \leq m$, that is, the ith partial sum is the sum of $|L_j|$ where $j \geq i$.
- *vi* starts as an empty list. The evaluation of variable v_i will be stored in *vi[i]*, $1 \leq i \leq m$.
- The *rem_wb* parameter of the recursive function *solve_rec()* is the weight that remains to be distributed among the variables v_i that have not been assigned a value yet. At the start of the algorithm, *rem_wb=wb*, and after all variables are evaluated, *rem_wb=0*. For any call of *solve_rec()*, the sum of *rem_wb* and the total weight of *vi* is *wb*.

- The *check_mu_conditions()* function check the current evaluation of vi against the μ-conditions. For a given μ-condition $M_{a,b} = \sum_{i \in V(a)} v_i \in Mu[a,b]$, let $I(M_{a,b})$ be the biggest index in $V(a)$. Note that not all variables v_i need to have a value assigned to check if a given μ-condition $M_{a,b}$ holds: the condition can be checked as soon as $v_{I(M_{a,b})}$ has a value. If the condition does not hold, the recursive function call immediately returns. In this way, the algorithm immediately abandons prefixes that can not lead to valid candidate codewords.

- The *upper_limit* and *lower_limit* variables determine the maximum and minimum values a variable v_i can take. The upper limit is $|L_i|$ or the remaining weight that needs to distributed, whichever is lower. The lower limit is 0, or the remaining weight minus the sum of all $|L_j|$ where $j > i$, whichever is higher. If the lower limit is a positive w_p, that means assigning any weight less than w_p to the variable currently being evaluated would leave too much remaining weight to fit into the remaining variables.

Once an evaluation is obtained for $v_1, v_2, \ldots, v_m$ that fulfills all μ-conditions, a candidate codeword b can be created. To preserve the generator matrix being in ordered form, b is constructed the following way:

For $1 \leq i \leq m$, let

$$b_i = (\underbrace{1\ 1\ \ldots\ 1}_{v_i}\underbrace{0\ 0\ \ldots\ 0}_{|L_i| - v_i})$$

Note that v_i or $|L_i| - v_i$ may be 0.

Then, b is the concatenation of all $b_i, 1 \leq i \leq m$.

In order to generate all possible candidate codewords that fulfill a given set of μ-conditions, the generator described in listing 8.3 is called for all permissible weights wb. If the matrix being constructed is in weight sorted form, this constrains the weight wb to be at most the weight of the last row of A'.

The Torch framework optionally allows the ordered form of the generator matrix to be reversed, that is, create the matrix A such that its columns are in lexicographical ascending order. In this case, the 0 bits would precede the 1 bits in each b_i in the above construction.

If the search algorithm is configured to create generator matrices that are not in ordered form, then the algorithm simply treats all L_i sets as having $|L_i| = 1, 1 \leq i \leq m = n$.

The method described in this subsection so far can be used to create candidate codewords for augmenting any kind of linear code. In the case of constructing self-orthogonal codes, the framework also employs a second method.

Again let A' be the $k' \times n-k$ matrix that is stored by the algorithm at each node when trying to construct a $G = (I_k|A)$ generator matrix that generates a self-orthogonal (n, k, d) linear code C. Let g_j denote the jth row of G, i_j denote the jth row of I_k and a_j denote the jth row of A. The following holds for any $j, l \in \{1, 2, \ldots, k\}, j \neq l$:

$$\mu(g_j, g_l) = \mu(i_j, i_l) + \mu(a_j, a_l) = \mu(a_j, a_l)$$

Since the μ-value between two rows of I_k is always 0, two row of G can only be orthogonal if the corresponding rows of A are orthogonal. That is, if G generates a self-orthogonal code, then the rows of A must be pairwise orthogonal. Any candidate codeword b for augmenting the matrix A' therefore needs to be a codeword of the dual code $A'^{\perp}$.

The codewords of $A'^{\perp}$ are typically filtered by two criteria. The first is the weight of the codeword, such that only those with permissible weights are admitted. This criteria is used in all search scenarios, not just when creating a weight sorted matrix, since the dual code can contain codewords with weight that falls outside the permissible range for the code C. As a second criteria, if the matrix being constructed is in ordered form, then the codewords that would break this requirement are also filtered out.

An important aspect to the performance of this method is the dimension of $A'^{\perp}$. Recall that if the dimension of A' is k', then the dimension of $A'^{\perp}$ is $n-k'$. For all but very small values of n, this means that at the start of the search, when k' is small, the dimension of the dual code is so large that the number of codewords the algorithm would need to filter through is impractically high. If $n-k'$ is small however, then generating candidate codewords using this method becomes significantly more efficient than the general method.

The default search algorithm keeps track of the dimension of $A'^{\perp}$ whenever it is creating candidate codewords to augment A' when searching for a self-orthogonal code. If the dimension is above an adjustable threshold, then the general method is used, otherwise it switches to the method using the dual code.

As mentioned previously, a codeword b satisfying the conditions described in this section is necessary but not suffucient to ensure that augmenting A' with b will result in a node in the search that can lead to successfully finding a given linear code. To further filter the candidate codewords and thus reduce the

number of branches in the search tree starting from a given node, the *children* module checks a list of conditions before an augmented matrix can become a new node. Some of these conditions are detailed in a later chapter.

Using the tools and methods described in this section, the default search algorithm can implement a depth-first search for an (n, k, d) linear code of a given type that generates at least one linear code from each permutation equivalent set of codes. In practice, while the algorithm has good performance at finding many different linear codes that can be used for practical purposes, the list of results for a given search will usually contain many permutation equivalent codes, which makes classification tasks difficult. Minimizing and eliminating these equivalent codes is the motivation behind multiple other configurations discussed in later chapters.

The default configuration combines the depth-first search with another algorithm based on finding cliques in a graph to reduce the number of permutation equivalent codes returned.

8.2 Clique-based search algorithm

In practice, during a typical search for an (n, k, d) linear code using the depth-first search described in the previous section, the L_i sets described in subsection 8.1.2 calculated for a matrix become smaller and smaller the deeper the node containing the matrix is in the search tree. In fact, since in most practically interesting cases a generator matrix for a linear code does not contain two identical columns, the search tree will contain nodes where $|L_i| = 1, 1 \leq i \leq m = n$. If the generator matrix being created is in ordered form, and all sets L_i for the matrix contain only one index, then the ordered property of the matrix can no longer be broken: no matter what codeword is used to further augment the matrix, it will always remain ordered.

Suppose we wish to create a generator matrix $G = (I_k|A)$ in default form, generating an (n, k, d) linear code C. As previously, let A' be the $k' \times n - k$ matrix being stored at a node during the search, $k' < k$. Let $\chi = k - k'$ denote the number of codewords A' needs to be augmented by to obtain a desired A. For the rest of this section, let A' be such that all columns of A' are different, but A'' obtained by removing the last row of A' does contain identical columns.

The rows of A will be denoted as $< a_1, a_2, \ldots, a_{k'}, b_1, b_2, \ldots, b_\chi >$.

If a row permutation is applied to the b_i rows in such a way that the matrix remains weight sorted, then the resulting matrix A^* will also be a default form

matrix that the depth-first search can consider a new result for the search. Since A^* is permutation equivalent to A, this should be avoided in tasks where we do not wish to find equivalent codes, such as classification.

Note that any codeword $b_i, 1 \leq i \leq \chi$, would satisfy the μ-condition set of A', and be a valid candidate codeword to augment A' with. Therefore, the algorithms described in subsection 8.1.3 can already find all of these codewords when creating candidate codewords for A'. In other words, once A' is obtained, we can create a list of all possible codewords that A' can be augmented with that can possibly lead to a result for the search.

In order to avoid generating matrices containing a given set of $\{b_1, b_2, \ldots, b_\chi\}$ rows more than once, the following algorithm is employed:

- All codewords that can lead to a search result when used to augment A' are determined.
- An undirected graph H is constructed where the vertices are the above codewords. The edges are determined based on conditions described in this subsection. The graph is such that a set of codewords that create a valid search result when augmenting A' appear as a clique in the graph.
- To optimize searching for such cliques, the graph is pruned of vertices and edges that are not part of a clique of correct size.
- A recursive algorithm is used to search the graph for cliques that satisfy the conditions of the search.
- For each unique clique found, one generator matrix is constructed as a search result.

Definition 8.2.1 (Clique, χ-clique)**.** A *clique* in a graph is a subgraph such that all vertices in the clique are adjacent to all other vertices in the clique. In other words, a clique in a graph is a subgraph that forms a complete graph.

If the clique has χ vertices, then it is called a χ-*clique*.

Let $c_1, c_2, \ldots, c_{l'}$ denote the candidate codewords that satisfy the μ-conditions of A', determined using the algorithms described in subsection 8.1.3. If the search needs to create matrices in weight sorted form, then the codewords are ordered such that $w(c_i) \geq w(c_i + 1), 1 \leq i \leq l - 1$. Let $\begin{pmatrix} A' \\ c_i \end{pmatrix}$ denote the matrix A' augmented by the codeword c_i.

The graph $H =< V, E >$ is constructed as follows. The vertices $V = \{c_1, c_2, \ldots, c_l\}$ are the codewords c_i such that $\begin{pmatrix} A' \\ c_i \end{pmatrix}$ satisfies all search conditions. Note that $l \leq l'$: as mentioned previously, the μ-conditions are not sufficient and the default search algorithm checks other conditions on an augmented code as well. The conditions checked by the default *children* module are such that if the conditions hold for a matrix A, then they also hold for any subset of rows of A.

An edge $e_{i,j} = (c_i, c_j), i < j$ exists if the matrix $\begin{pmatrix} A' \\ c_i \\ c_j \end{pmatrix}$ satisfies all search conditions.

If $\{b_1, b_2, \ldots, b_\chi\}$ is a set of codewords that lead to a completed search result when augmenting A', then $\begin{pmatrix} A' \\ b_i \\ b_j \end{pmatrix}$ also satisfies all search conditions for any $1 \leq i < j \leq \chi$, therefore (b_i, b_j) is always an edge in graph H. In other words, the codewords $\{b_1, b_2, \ldots, b_\chi\}$ form the vertices of a χ-clique in H.

However, it is not enough to simply search for all cliques of sufficient size in the graph, since not all χ-cliques in H necessarily lead to a search result.

Before the search for χ-cliques begins, some vertices and edges can be pruned from the graph. Clearly, if a vertex has less than $\chi - 1$ adjacent vertices, then it cannot be a part of a χ-clique. The algorithm takes the following steps:

1. Search for all vertices in V that have less than $\chi - 1$ adjacent vertices, denoted by V'.
2. Delete all vertices in V' and all edges in E connecting to a vertex in V'.
3. If V' was not empty, then new vertices may fall under the adjacency threshold after V' was deleted. Repeat from step 1.
4. If V' was empty, then all vertices remaining in the graph have a sufficient number of adjacent vertices.

After graph H is determined and pruned, the algorithm starts searching H for χ-cliques that satisfy the search conditions. For the rest of this subsection, assume that $\chi \geq 2$, since if $\chi = 1$, then we can construct the search results after the set of vertices V is determined.

A clique will be identified by its list of vertices, ordered by their indexes: $\mathscr{C}(c_i, c_j, c_k, \ldots), 1 \leq i < j < k < \ldots \leq l$. In order to ensure that a given clique is constructed only once, they are generated recursively:

- If we need to build 2-cliques, yield $\mathscr{C}(c_i, c_j)$ for each edge $e_{i,j} \in E$.
- If we need to build χ-cliques, $\chi > 2$, then for each $(\chi-1)$-clique $\mathscr{C}(\ldots, c_i)$, the algorithm tries to append c_j to the clique for each c_j that is adjacent to all vertices of the clique and $j > i$.
- A vertex c_j can be appended to a clique if the matrix obtained by augmenting A' with all the vertices of the cliques and c_j satisfies all search conditions.

Listing 8.4 in the Appendix shows the simplified pseudocode of the algorithm used for generating χ-cliques from graph H. In the implementation, the module keeps track of the mutual neighborhood of each clique. The mutual neighborhood of a clique is the set of vertices of H that are not in the clique but are adjacent to all vertices of the clique. If the algorithm needs to build a χ-clique, then a χ'-clique, $\chi' < \chi$, can not be a part of such a larger clique if the number of mutual neighbors it has is less than $\chi - \chi'$.

Note the following about listing 8.4:

- The *chi* and *chi_* parameters represent χ, the total size of the cliques needed to find search results, and χ', the size of the cliques the method is currently called to generate, respectively. χ is needed in order to calculate the number of mutual neighbours a clique is required to have.
- Y is the search conditions the matrix need to satisfy in order to be accepted. It is a Torch condition object, injected as a dependency into the module.
- The *mut_neighbours()* method of the clique object returns a list of vertices of H that are adjacent to all vertices in the clique, and have an index greater than the largest index in the clique.

The performance of the clique-based search depends greatly on the size of the graph H. In most practical cases, the number of candidate codewords that can augment A' is small enough to make the construction and search of the graph efficient.

The search algorithm implemented in the default configuration of Torch uses the depth-first search algorithm when searching from a node that contains a matrix with at least two identical columns. If a search node contains a matrix where

all columns are different, it switches to the clique-based algorithm described in this subsection.

The clique-based search algorithm is not always compatible with other configurations. For some specialized search conditions, the assumption that all valid search results appear as a clique in graph H does not hold. In such cases, the framework can be forced to use the depth-first search only.

Subsection 8.2.1 contains some experimental results comparing the search algorithms detailed in this chapter so far.

8.2.1 Experimental results

The following tables contain some computational results when using the default and depth-first search only search algorithms. The implementation was run on a typical laptop with AMD Ryzen 5 7640HS CPU (4.30GHz). The Sagemath environment was run inside a Linux virtual machine.

The tables show the (n, k, d) parameters and the type of code being searched, the number of codes found, the execution time, and the number of codes that are not permutation equivalent among the codes found.

Table 8.2 in the Appendix shows the results of using the default configuration, using depth-first search and cliqued-based search, generating default form matrices.

Table 8.3 in the Appendix shows the results of using only the depth-first search to generate default form matrices.

In general, the default configuration shows better performance for classification problems, while the depth-first search will sometimes yield faster results when the goal is to generate only a given number of codes that fit the search parameters.

In some cases, the algorithms can generate a lot of equivalent codes. As shown in the tables, when generating $(24, 12, 8)$ Type II codes, the algorithms will quickly create a hundred generator matrices, and will continue to create more if left to execute, even though all codes of this type are equivalent. This is one of the main issues that are addressed by other configurations.

Also note that since the search algorithm can not know which search result will be the last, in practice the algorithms may continue to execute for a long time after the last search result has been found, as it completes traversing the search tree. As seen in the tables, when searching for $(63, 57, 3)$ codes, the first and only search result is found quickly, but the search continues, trying to find more.

8.3 Heuristic search algorithms

For classification problems, when the goal is to construct an exhaustive list of linear codes that are not permutation equivalent that satisfy a given set of parameters, the entire search tree needs to be traversed by the search algorithm before the task can be considered finished. In this case, the order of traversal of the child nodes of a given node in the tree does not impact the overall runtime of traversing the entire tree.

If the goal of the search is not classification, but to find a number of linear codes that fit the search criteria, then the order in which the nodes of tree is traversed becomes important.

The search algorithms described in the previous sections of this chapter have been uninformed searches, meaning that the algorithms did not use any knowledge about the linear codes themselves to determine the direction of the search in the tree. The order of traversal is determined by the order in which the candidate codewords are created at each node to augment the given generator matrix. The default *solutions* module offers some options to control this order, such as ascending or descending lexicographical order, or sorting the codewords by weight. However, in general, these options do not determine which augmented code would lead the search to a result the fastest. These uninformed searches rely on using a variety of conditions to check each node that is constructed to see if it can lead to a search result, abandoning branches that are determined to be dead ends. In this way, the algorithms try to prune the search tree down to a size where finding search results becomes more likely.

Informed searches are a classical topic of artificial intelligence, used in a wide variety of problems to improve the performance of finding states with favorable properties in a state space. In these algorithms, special knowledge about the properties of the objects being searched are incorporated to guide the direction of the search.

The Torch framework provides the possibility of using informed search algorithms for constructing linear codes. These algorithms rely on using a heuristic function to calculate a value for each generator matrix, representing how favourable a given matrix is to the search. One challenge of using such algorithms for searching for linear codes is that the nature of linear codes makes it difficult to determine functions that are both helpful in guiding the search and are efficient to compute. Practical experience has shown these algorithms to be most useful when searching for codes with very specific properties, instead of more general search scenarios.

The current implementation of the framework provides two basic informed

search algorithms: a hill climbing search, and a best-first search. These are found in two separate configuration files in the Torch package. The framework also provides a number of built-in heuristic functions to choose from, and the ability to define custom heuristic functions to enable experimentation. The provided heuristic functions and the way to combine and evaluate them are described in subsection 8.3.1.

The hill climbing search is a greedy algorithm. Given the same parameters and search criteria, the search tree constructed by the hill climbing search contains the same nodes as the tree constructed by the depth-first search. The sole difference is the order in which the nodes are constructed.

The hill climbing algorithm is implemented by replacing the *solutions* module of the configuration it is added to. The new *solutions* module wraps the previous one. When creating candidate codewords to augment a given generator matrix, the codewords yielded remain unchanged, but the order in which they are yielded is determined by the heuristic function. The candidate codewords can be sorted in ascending or descending order of their heuristic value.

Determining the candidate codeword with the best heuristic value for a given generator matrix requires computing and sorting all possible candidate codewords for that matrix. In the previous uninformed depth-first search, the generator methods creating these codewords did so one at a time whenever they were called to construct candidates. The next candidate codeword for a matrix would only be created when the search backtracked to the node after being finished with the branch created by the previous candidate. Depending on the parameters of the search and the generator matrix at the current node, it can be impractical to compute and store all candidate codewords at once. To address this, the *solutions* module of the hill climbing search creates and sorts candidate codewords in batches of a given size. The size of a batch can a finite value, or unlimited if we wish to effectively turn off the batching instead. If the size of the batches is set to a finite value s, then the *solutions* module will create a list of at most s candidate codewords at a time. These batches are then sorted using the heuristic function, and yielded one by one.

Listing 8.5 in the Appendix shows the simplified pseudocode of the candidate codeword generator method used by the *solutions* module of the hill climbing search algorithm. Note the following about the method:

- The module wraps another *solutions* module. In practice, the *solutions* module the hill climbing algorithm takes an existing candidate codeword generating algorithm, like the one detailed in subsection 8.1.3, and sorts the codewords generated by it according to the heuristic function.

- The heuristic function *h()* determines the fitness of a given generator matrix. The function $h()$ can be single or multivalued. This function is injected by the DI container as a dependency, allowing the heuristic being used to be determined by the user before the search starts.

- The *batch_size_limit* parameter determines the upper limit to the number of candidate codewords that are generated and sorted for a given generator matrix each time the algorithm requires new candidate codewords. This parameter is injected by the DI container as a dependency when the module is being created.

- The batches of candidate codewords created by the *get_batches()* method are lists of $(h(G'), b)$ pairs where G' is the matrix augmented by the candidate codeword b. These batches are then sorted by the heuristic value before the *get_candidates()* generator method yields the candidate codewords b in the batch one by one. After all codewords in a batch have been yielded, a new batch is created.

The best-first search (BFS) algorithm, defined in a separate config file, replaces the *descendants* module of the configuration preceding it.

Given the same set of parameters and search criteria, the search tree traversed by the best-first search is a subtree of the tree traversed by the depth-first search algorithm. Depending on the parameters of the best-first search and the size of the tree, this subtree might be a proper subtree, or it might be the same tree as built by the depth-first search. As with the hill climbing algorithm, it uses a heuristic function to determine the fitness of each created node, using it to guess which node is more likely to lead to search results.

The algorithm keeps track of a list of *open nodes*, that is, nodes that can have children nodes that have not been traversed yet. At the start of the search, the list of open nodes has one element, the root node of the search tree. This node corresponds to the generator matrix given as a parameter to the *get_descendants()* method of the *descendants* module. See section 8.1 for details. The list of open nodes is always kept sorted by the heuristic value of the nodes.

The algorithm is a loop that checks the list of open nodes at the start of each iteration. The search finishes when the list becomes empty, which means all nodes of the search tree have been traversed. If the list is not empty, then the node with the greatest heuristic value, located at the beginning on the list, is expanded.

When the algorithm *expands* an open node, that is, creates child nodes for it, it does so in batches as described in the hill climbing algorithm, for the same practical reasons. Any child nodes that are valid search results are yielded by the search algorithm instead of added to the batch. The batch of child nodes created is added to the list of open nodes, while keeping the list sorted by heuristic value. As long as the node that was expanded can have more child nodes, it remains on the list of open nodes. If all child nodes of the expanded nodes have been created, then it is removed from the list.

The main practical challenge of the algorithm is the extreme size of the search trees that can be build for linear codes of even relatively small sizes. Since a single node in the tree can have a large number of child nodes, the list of open nodes grows very quickly at the start of the search. If the list is allowed to grow without limit, then the space required by the algorithm can quickly become impractically large to run on a typical desktop computing environment. To address this, the Torch framework provides an option to limit the size of the list of open nodes. This option can be set to be unlimited, if we wish to allow the list to grow to arbitrary size. If the option is set to a finite value, then at the end of each iteration of the algorithm, if the size of the list has exceeded the limit, the nodes with the worst heuristic values are deleted from the list until its size is reduced to the limit. Branches starting at nodes that are deleted this way will never be traversed by the search algorithm. If this happens during the execution of a search, then the tree traversed by the best-first search is a proper subtree of the tree traversed by the corresponding depth-first search. Such a search can not be used for classification problems, as generally there is no guarantee that the branches deleted by the search did not contain linear codes that are not found in the tree traversed by the search.

Listing 8.6 in the Appendix shows the simplified pseudocode of the best-first search algorithm used by Torch. Note the following about the algorithm:

- The input G and dependencies C and Y of the method are the same as those of the default descendant generator method described in listing 8.2.
- The heuristic function $h()$ has the same properties as described in the hill climbing method.
- The open nodes of the search are stored as a pair of the heuristic value $h(G)$ and the generator matrix G.
- The list of open nodes begin with one element: the node representing the starting matrix.

- The *batch_size_limit* parameter determines the upper limit of how many child nodes are created at once when an open node u is expanded. The parameter is treated as a dependency that is injected by the framework before the search begins.
- The *limit* parameter determines the maximum number of open nodes the algorithm can store at any given time. Excess nodes are deleted using the *cut()* method. The *limit* is treated as a dependency by the framework.
- The *add_and_sort()* method adds the child nodes obtained from expanding the current open node to the list of open nodes. The list is then sorted by the heuristic value of each node.

Another challenge of the best-first search algorithm being applied to linear codes is the choice of heuristic function. If the heuristic function h is such that $h(G_1) > h(G_2)$ always holds if the number of rows of G_1 is greater then the number of rows of G_2, then the best-first search algorithm is effectively the same as the hill climbing algorithm. However in practice, if a generator matrix with a smaller number of rows can have a better heuristic value than a generator matrix with a greater number of rows, then the best-first search can easily get stuck at a certain depth level of the search tree, rarely moving on to greater depths to actually produce search results.

8.3.1 Heuristic functions

This subsection presents the predefined heuristic function available for the hill climbing and best-first search algorithms in the Torch framework at the time of writing. As mentioned previously, the user can also define custom heuristic functions to enable further experimentation.

The heuristic search algorithms described in the previous subsection can sort the nodes of the search in descending or ascending order of heuristic value, determined by an option during the initialization of the search.

The heuristic functions take a generator matrix of any size as input. The return value of a heuristic function can be any data type that can be ordered.

The functions can be combined together to form multi-valued functions. If $h_1, h_2, \ldots, h_l$ are heuristic functions, then

$$h'(G) = (h_1(G), h_2(G), \ldots, h_l(G))$$

is also a valid heuristic function that can be used by the search algorithms. If a heuristic function h' is multi-valued, then $h'(G_1)$ and $h'(G_2)$ are compared

using their lexicographic order: let h_i be the component of h' with the smallest index i such that $h_i(G_1) \neq h_i(G_2)$. If $h_i(G_1) > h_i(G_2)$, then $h'(G_1) > h'(G_2)$, otherwise $h'(G_1) < h'(G_2)$.

Different generator matrices can have equal heuristic values. In this case, the heuristic search algorithms will sort these generator matrices in the order that they were created. In other words, sorting matrices by heuristic value is stable.

The predefined heuristic functions of the Torch framework are the following:

- *Weight heuristic*: This function is not meant to be used on its own, but as a component of a multi-valued heuristic function. Let G be a $k \times n$ generator matrix, and let g_i denote the ith row of G. Then, the weight heuristic h_1 is the weight of the last row of the generator matrix:

$$h_1(G) = w(g_k)$$

- μ^2 *heuristic*: Let G be a $k \times n$ generator matrix, and let g_i denote the ith row of G. Then, the μ^2 heuristic h_2 is the following sum:

$$h_2(G) = \sum_{i=1}^{k-1} \mu(g_i, g_k)^2$$

- μ*-list heuristic*: This heuristic fulfills a very similar function to the μ^2 heuristic. The μ-list heuristic is multi-valued, returning a list of integers. These lists are compared in a lexicographical way. Using the same notation as before, let the μ-list heuristic h_3 of a generator matrix G be the following:

$$h_3(G) = sorted((\mu(g_1, g_k), \mu(g_2, g_k), \ldots, \mu(g_{k-1}, g_k)))$$

  Even though this function is multi-valued, it can still be assigned to be a component another multi-valued heuristic function.

- *Extremal word count heuristic*: This heuristic is typically used in searches where the linear code is expected to have a symmetrical weight distribution, such as self-dual codes. Let C be the linear code generated by G. Recall that A_{Ci} denotes the number of codewords in C with weight i. Then, the extremal word count heuristic h_4 of G is:

$$h_4(G) = A_{Cd} + A_{Cn-d}$$

In a code with symmetrical weight distribution, d and $n-d$ are the smallest and biggest weights a codeword can have, aside from codewords **0** and **1**.

- $|L|^2$ *heuristic*: Recall the L_i sets defined in subsection 8.1.2. For any generator matrix G, the $L_1, L_2, \ldots, L_m$ sets define the sets of indices belonging to identical columns of G. Then, the $|L|^2$ heuristic h_5 is defined as the following:

$$h_5(G) = \sum_{i=1}^{m} |L_i|^2$$

  With h_5, the search algorithm will prefer candidate codewords that keep the most (or least) number of columns identical when augmenting the generator matrix.

- *Base vector sum heuristic*: This heuristic is mostly used with the *paperclip* configuration discussed in a later chapter. The function returns a codeword as a heuristic value. These codewords are compared in a lexicographical way. The base vector sum heuristic h_6 of G is:

$$h_6(G) = \sum_{i=1}^{k} g_i$$

  When using the paperclip algorithm, the $h_6(G)$ codeword is the lexicographically largest codeword in the code.

- *Automorphism group size heuristic*: Again let C be the linear code generated by G. Recall from section 6.2 that $Aut(C)$ denotes the automorphism group of the linear code C. The automorphism group size heuristic h_7 is the following:

$$h_7(G) = |Aut(C)|$$

  Note that augmenting a generator matrix can both increase or decrease the size of its automorphism group, depending on the specific matrix and codeword used to augment it.

8.3.2 Experimental results

The tables in this subsection compare the computational results of using the hill climbing and best-first search algorithms described in section 8.3. Like the experimental results in previous sections, the implementation was run on a

laptop with AMD Ryzen 5 7640HS CPU (4.30GHz), in a SageMath environment executed inside a Linux virtual machine.

The tables show the (n, k, d) parameters and the type of code being searched, the heuristic function and batch size limit used in the search, the number of codes found, the execution time, and the number of codes that are not permutation equivalent among the codes found. The batches created by the algorithms were sorted in ascending order of heuristic value. The heuristic functions used were those defined in subsection 8.3.1.

Table 8.4 in the Appendix shows the results of using the hill climbing search to generate default form matrices.

Table 8.5 in the Appendix shows the results of using the best-first search to generate default form matrices.

Since the heuristic search algorithms only alter the order in which the nodes of the search tree are visited, the runtime of classification problems, which necessarily traverse the entire tree, do not significantly differ from the depth-first search algorithm. As an example, compare the first two rows of table 8.5 with table 8.3 in the Appendix. For non-classification problems, unless shown otherwise in the *codes found* column of the tables, the goal was to generate 100 generator matrices using the specified algorithms.

Table 8.4 demonstrates the effect of the choice of heuristic function and batch size on search results. All codes shown in this table are types that the default algorithm does not generate at all in a practical period of time. While both the h_2 and h_4 were effective in generating $(32, 16, 8)$ Type II codes, h_4 was significantly more efficient. However, changing the batch size by even a small degree, such as from 20 to 16, caused the search using h_4 to halt generating codes after 69 results, while the previous cases would continue generating well over the 100 results shown in the table. Note however, that many of these results were equivalent.

As mentioned in Section 8.3, the best-first search algorithm is rarely practical when used with the previously defined heuristic functions. Table 8.5 also demonstrates the use of multi-valued heuristic functions. When generating $(24, 11, 6)$ Type I codes, h_5 does not produce any results after 5 minutes, while h_3 produces 100 equivalent codes in a little under 4 minutes. However, combining the two heuristic functions results in more non-equivalent codes in a significantly shorter time.

Chapter 8 has presented the basic search algorithms used by Torch, that are modified by the various options and configurations provided to achieve the necessary flexibility to execute searches for linear codes with arbitrary properties.

Chapter 9

Other framework elements

The behavior of the modules implemented in the Torch framework can be modified in multiple ways. The configurations defined in the config files describe the most common ways the modules are used together in practice. This chapter describes elements that affect the execution of the search, and are usable with most configurations.

9.1 Options

To make modules flexible enough to be reusable in many different search scenarios, most of them are defined to accept a number of different options. In the config files, these options typically appear as *optional unknown arguments*. Recall from subsection 7.2.3 that these arguments are resolved by the DI container during instantiation. The values of these arguments are taken from either a set of default values defined in the config files, or the keyword arguments given to the `get_code()` function, with the latter taking precedence over the former. If the DI container finds no value, the option defaults to a value defined in the module.

Important options specific to modules that are only used by one configuration are described when discussing that configuration. The options presented in this chapter are ones that are used in multiple common search scenarios.

Common options that impact the behavior of the search algorithm are:

- *save*: By default, the search algorithm saves all linear codes that are yielded as search results in separate files, using the Python *pickle* package.

The *save* option takes a string, which determines the path of the save folder where the serialized linear code objects will be stored. If set to *None*, it turns off the saver module.

- *verbose*: Most modules log their internal activity to allow the user to follow and analyze the execution of the search. The basic logger module writes logs to console. The *verbose* option takes a boolean, its default value is *True*. If set to *False*, this option turns of the logger module.

- *tree_search*: The default search algorithm uses a combination of depth-first search and clique-based search, as described in chapter 8. The *tree_search* option takes a boolean, its default value is *False*. If set to *True*, the algorithm is forced to use only the depth-first search. In practice, this option is used when running a search scenario that includes conditions incompatible with the clique-based search.

- *standard*: This option takes a boolean, its default value is *True*. It determines if the generator matrices created by the algorithm should be in standard form or not. If set to *False*, the nodes of search tree will store $k' \times n$ matrices at depth level k', instead of $k' \times n - k$ matrices. This affects the values stored in the Mu tables.

- *ordered*: This option takes a boolean, its default value is *True*. Determines if the generator matrices should use ordered form to reduce the number of codes created that are permutation equivalent to each other. If set to *False*, the L_i sets defined in subsection 8.1.2 are set to $|L_i| = 1$ for all $1 \leq i \leq m$.

- *ones_before_zeroes*: When using ordered form, the columns of a generator matrix G yielded by the search are in lexicographical order. The *ones_before_zeroes* option takes a boolean, its default value is *True*. If this option is set to *True*, the columns are ordered to be in descending lexicographical order from left to right. If it is set to *False*, the columns are in ascending order. The option has no effect if the *ordered* option is *False*.

- *wsorted*: This option takes a boolean, its default value is *True*. Determines if the rows of the generator matrices created should be sorted by Hamming weight. If set to *False*, the weights of the rows can be in any arbitrary order.

Another option, named *relation*, defines the relation between the weights of the rows. By default, $w(g_{i+1}) \leq w(g_i)$ for $1 \leq i \leq k-1$, where g_i is the ith row of generator matrix G.

- *row_relation*: This option takes a string representing a relation, or *None*. The *row_relation* determines the lexicographical relation between the rows of the generator matrices. Its default value is *None*, allowing any relation between the rows. If the option is set to e.g. "<=", then for all g_i rows of G, $1 \leq i \leq k-1$, $g_{i+1} \leq g_i$ must hold. Note that in practice, the rows of a generator matrix will not be equal, but the option is typically set to "<=" or ">=" to allow the *solutions* module to check the condition even during candidate codeword generation, while only a prefix of the codeword is available.

- *solution_weight_sorted*: This option takes a boolean, its default value is *True*. When set to *True*, the *solutions* module generates candidate codewords sorted by weight, and secondarily sorted in lexicographical order in the case of equal weights. If set to *False*, the candidate codewords are generated in lexicographical order.

- *ascending*: This option takes a boolean, its default value is *False*. This option determines the order in which the valid weights of a linear code are listed by the *weights* module. This also affects the order of the candidate codewords being generated by the *solutions* module. If the *solution_weight_sorted* option is *True*, then the *ascending* option determines if the candidate codewords will be generated in ascending or descending order by weight.

- *descending_solutions*: This option takes a boolean, its default value is *True*. If this option is *True*, then the candidate codewords generated by the *solutions* module are created in descending lexicographical order. If set to *False*, the codewords are generated in ascending lexicographical order. This ordering may be the primary or secondary sorting priority, as determined by *solution_weight_sorted*.

- *equation_limit*: This option takes a positive integer, or *None*. If set to a positive integer, the option determines an upper limit on the number μ-conditions that can be created for a generator matrix, as described in subsection 8.1.2. If set to *None*, a μ-condition is created for every codeword generated by the matrix, with no upper limit.

- *dimension_limit*: This option takes a positive integer. As described in subsection 8.1.3, when searching for self-orthogonal codes, the *solutions* module may switch over to using a dual code based algorithm for generating candidate codewords if the dimension of the dual code is small enough. This switch occurs if this dimension is less than or equal to the value of the *dimension_limit* option.

9.2 Conditions

Another element of the Torch framework that are commonly used by configurations are condition objects. As described in section 8.1, the framework provides a base class to extend when creating classes that can check if a linear code satisfies a given condition. These classes provide features that make logging, bypassing, and combining conditions more convenient. It also allows the conditions to be injected by the DI container as a dependency, providing a way to easily change conditions or triggers in the search algorithm, enabling experimentation.

Condition objects can fulfill a variety of roles in the search algorithm. Some conditions are used as triggers to change the execution of the algorithm. Other conditions are used to prune the search tree, detecting generator matrices that can not lead to search results, so the algorithm can skip creating nodes for them.

Because of the varied purposes these types of objects fulfill, the condition classes included in the framework range from checking a very basic property of the linear code, to complex research results. Some of the simplest configurations in the framework add nothing except for a single new condition to the search algorithm.

Some modules used only in one specific configuration also use condition objects for their particular purposes. These conditions will be described when presenting those configurations in a later chapter. The common modules used in basic search algorithms use condition objects for the following:

- The default *descendants* module uses a condition object to check at each node if the search results that are the descendants of that node should be determined using the depth-first search algorithm or the clique-based search algorithm.

- All *descendants* modules have a condition object that checks if the node where the execution of the search is currently at is a search result. In other words, it checks if the current node fulfills all the conditions we were searching for.

- The default *solutions* module uses a condition object to check at each node if the candidate codewords to augment the matrix at that node should be determined using the general algorithm or the dual code based algorithm, as described in subsection 8.1.3.

- The default *children* module uses a condition object to determine at each node which of the augmented matrices, created by augmenting the matrix at the node with a candidate codeword yielded by the *solutions* module, should be allowed to be added to the search as a new node, and which of the augmented matrices can not lead to search results and should be skipped.

The condition object in the *children* module has a very significant effect on the performance of the search, as it is one of the main elements that determine the size of the search tree.

This object contains a list of conditions, and a generator matrix must satisfy all of them in order to be added as a new node. The conditions on this list need to fulfill certain requirements in order to be effective:

- They need to be necessary conditions for the specific kind of linear code that we are searching for.

- To support classification tasks, the conditions on this list should be such that at least one code from each permutation equivalent set of codes must satisfy them.

- Recall that the non-leaf nodes of a search tree represent subcodes of the kind of linear code we are searching for. Therefore the conditions must be such that for any linear code C that should be yielded as a search result, there must exist an i-dimensional subcode C_i for $1 \leq i \leq k$ that satisfies the conditions, and $C_1 \subset C_2 \subset \ldots \subset C_k = C$.

The order in which the conditions appear on this list can also impact the performance of the search. Typically, conditions that can be executed the fastest are at the beginning of the list, such that more computation heavy conditions need only be executed if the previous conditions have already passed.

If the goal of a condition is to reduce the size of the search tree, then it is the most effective if it can filter out nodes even at a low depth level. The closer a node is to the root, the bigger the subtree is that starts from it, therefore conditions that can filter out nodes close to the root save the most time during the execution of the algorithm.

Each condition object has a method that takes an object representing a linear code C generated by a generator matrix G as an input, and determines if the code satisfies the given condition. The condition objects can also determine which search scenarios they are mathematically relevant in, and are automatically bypassed in a scenario in which they are not relevant. The following is a non-exhaustive list of conditions implemented in the Torch framework that are used in basic search algorithms:

- *d check*: This condition class takes an integer d as a parameter during instantiation. This is a basic condition that checks if the given linear code C has a minimum weight that is not less than d. Optionally, other relations are also available. This condition is a part of the list of default conditions used in the *children* module.

- *k check*: This condition class takes an integer k as a parameter during instantiation. This is a basic condition that checks if the given linear code C has a dimension that is equal to k. This condition is used by the *descendants* module when determining if a code should be yielded as a search result.

- *L check*: This condition determines if the L_i sets of generator matrix G, defined in subsection 8.1.2, satisfy $|L_i| = 1$ for all $1 \leq i \leq m$. This condition is used by the *descendants* module to determine if it should switch to the clique-based search algorithm from the depth-first search.

- *$C^\perp$ dimension check*: This condition class takes an integer l as a parameter during instantiation. The condition checks if the dimension of the dual code $C^\perp$ is not more than l. The check is a necessary but not sufficient condition used by the *solutions* module to determine when it can switch to a dual code based candidate codeword generation algorithm, as described in subsection 8.1.3.

- *linear independence check*: This condition checks if the rows of the generator matrix G are linearly independent. The check is automatically bypassed if the generator matrices created by the search are in a form that include the columns of the identity matrix, such as standard form. This condition is a part of the list of default conditions used in the *children* module.

- *A^T check*: This condition is only relevant when searching for self-dual codes, creating generator matrices in standard form. It is automatically

bypassed in all other searches. Recall that if $G = (I_k|A)$ is a standard generator matrix for a linear code C, then $G' = (A^T|I_{n-k})$ generates $C^\perp$, which is the same as C if C is self-dual. In this case, it follows that $(I_k|A^T)$ generates a code C' that is permutation equivalent to C. Therefore, if A is a matrix that satisfies all necessary conditions to allow G to generate an (n, k, d) self-dual code, then A^T must also satisfy those conditions. One condition that can be applied to A^T is that the rows of $(I_k|A^T)$ must have weights that are valid codeword weights for an (n, k, d) self-dual code of the type we are searching for. This condition is a part of the list of default conditions used in the *children* module.

- *n-d weight limit check*: This condition is only relevant when searching for self-dual codes, otherwise it is bypassed. Recall that since a self-dual code always contains the codeword $\mathbf{1}$, the weight distribution of an (n, k, d) self-dual code can not contain any codewords with weight w_c such that $n - d < w_c < n$. Since this holds for all subcodes of such a code as well, the search can reject any generator matrix G that generates a codeword with weight that falls in the $(n - d, n)$ interval. This condition is a part of the list of default conditions used in the *children* module.

- $C^\perp$ *singly evens check*: This condition is only relevant when searching for self-dual codes of Type I, otherwise it is bypassed. Recall that a Type I (n, k, d) self-dual code C contains 2^{k-1} singly even and 2^{k-1} doubly even codewords. Because C is self-dual, $C_i \subset C \subset C_i^\perp$ for any subcode C_i of C. Let G be a matrix that is in a node at depth k' in the search tree of an (n, k, d) Type I self-dual code. Then, $G^\perp$ must contain at least 2^{k-1} codewords with singly even weights that are valid codewords weights for the kind of code we are searching for. This condition is a part of the list of default conditions used in the *children* module.

- *Symmetric* A_{Ci} *check*: This condition is only relevant when searching for self-dual codes, otherwise it is bypassed. This condition checks if the weight distribution of the linear code C is symmetrical, that is, $A_{Ci} = A_{Cn-i}$ for all $0 \leq i \leq \frac{n}{2}$.

- *General weight limit check*: This condition class takes an integer v as a parameter during instantiation. The check passes if the the linear code C does not contain any codewords with weight greater than v. This condition is added to the search is specific search scenarios, if we wish to set an upper limit to the weight of the codewords in the yielded linear codes.

Chapter 10

Configurations

In this chapter, we describe a number of configurations implemented in the Torch framework that are used in a variety of search scenarios. As previously mentioned, configurations describe a collection of modules and parameters that are commonly used together to define or modify a search algorithm to achieve a specific search goal.

Recall from subsection 7.2.3 that in practice, configurations are described in config files that are used by the DI container to instantiate the necessary objects and set the necessary parameters at the beginning of the search. An important property of configurations is that multiple of them can be combined together: given a list of config files, the DI container adds the descriptions in the files together, overwriting overlapping data in the order it is found in the list. This creates the flexibility needed to enable experimentation in a wide variety of search scenarios.

Some configurations define a complete search algorithm, such as the ones described in chapter 8. In practice, only the config file defining the default algorithm can be used on its own, all other config files use the default config as a basis to build on.

Some configurations are only meant to modify the execution of the search in a more specific way, or add a certain condition to the search. The configs presented in this chapter fall into this category. The creation of these configurations is usually motivated by the wish to constrain the search in some way, or to make the construction of a certain subset of linear codes more efficient.

The chapter presents the following non-exhaustive selection of the configurations available in the Torch framework:

- *Weight Restriction*: In practice, the name refers to two different configurations with a similar purpose. These configurations can be used to place various restrictions on the allowed weights of the rows of generator matrices created.
- *Permutation Equivalence*: A configuration used to ensure that the search algorithm does not return permutations equivalent codes, using a basic approach to isomorph rejection.
- *Paperclip*: An algorithm with the goal of almost completely eliminating permutation equivalent codes from the yielded codes, while avoiding the drawbacks of the basic *Permutation Equivalent* configuration, using a complex but efficiently computable set of conditions.

Of the configurations presented, the *Paperclip* is the most significant in terms of transforming the usual search algorithm. The set of conditions used by the Paperclip algorithm are new theoretical research results by the author.

Along with the algorithmic descriptions, experimental results are also presented for the configurations.

10.1 Weight Restriction

When constructing generator matrices with specific criteria, one of the most common requirements is the restriction of the weights of the rows of the matrix to a subset of the usually available values. This restriction can be done for a variety of reasons, such as simple experimentation to see if a generator matrix with given weights exists, or to reduce the size of the search tree by only trying weight combinations during construction that are known to exist.

As an example, one result about the weights of generator matrices can be found in [62], which states a more general form of the following:

Theorem 10.1.1. Let C be an (n, k, d) binary linear code, and let $n = t + g_2(k, d)$, where $g_2(k, d)$ is the Griesmer function:

$$g_2(k, d) = \sum_{j=0}^{k-1} \left\lceil \frac{d}{2^j} \right\rceil$$

Then, a matrix G exists where all rows have weight not exceeding $d + t$ that generates C.

Note that while this condition can be applied to any linear code search to reduce the size of the search tree, in some scenarios this can be detrimental to the search goal. For example, if the aim is to find just a limited number of codes as quickly as possible, then restricting the possible weights may result in the execution time becoming longer due to the reduced number of valid generator matrices that can be found.

The Torch framework includes two different configurations for placing restrictions on the weights used when constructing generator matrices: a *Weight Limit* configuration that defines an allowed interval for the weights, and a *Weight List* configuration that takes a list of values that can define complex conditions for the weights.

The *Weight Limit* configuration creates a new *weights* module that only generates the weights that fall into the given interval. The DI container injects this new module into all modules and conditions that depend on the possible weights of the rows of the generator matrices. This new object does not overwrite the previous *weights* module, as some modules, such as the *mu_table*, still require all possible weights for their function. The new module takes two options to determine the interval of allowed row weights: *min_line_weight* and *max_line_weight*. Both options take integers or a *None* value. The *None* value, which is the default, indicates that the minimum or maximum allowed value is the same as determined by the previous *weights* module.

The *Weight List* configuration works in similar way as the *Weight Limit* configuration, by creating a new *weights* module and injecting it into the modules where it is required. The difference is in the option that determines the accepted weight values for the rows. The option, named *weight_list*, takes a list of values $[e_1, e_2, \ldots, e_l], l \leq k$. The element e_i determines the accepted weight values of the ith row of the generator matrix. If $l < k$, then for rows g_i where $l < i \leq k$, no constraints are placed on the allowed weights. The elements e_i can be one of the following:

- an integer v: in this case, the ith row of the generator matrix must have a weight of v.
- a list of integers: in this case, the ith row of the generator matrix has to have a weight that is on the list.
- *None*: in this case, no constraint is placed on the allowed weights of the ith row of the generator matrix.

As an example, if the *weight_list* option has the following value:

$$weight\_list = [[8, 12], 4, None, 6]$$

then the configuration will create generator matrices where the first row must have a weight of either 8 or 12, the second row must have a weight of 4, the third row has no constraints on its possible weights, the fourth row must have a weight of 6, and the fifth row and all rows afterwards also have no constraints on their weights.

10.1.1 Experimental results

The tables referenced in this subsection compare the computational results of using the *Weight Limit* and *Weight List* configurations described in section 10.1. Like the experimental results in previous sections, the implementation was run on a laptop with AMD Ryzen 5 7640HS CPU (4.30GHz), in a SageMath environment executed inside a Linux virtual machine.

The tables show the (n, k, d) parameters and the type of code being searched, the weight restriction parameters used in the search, the number of codes found, the execution time, and the number of codes that are not permutation equivalent among the codes found.

The configurations are added to the default search algorithm as described in section 8.1, using both depth-first search and clique-based search algorithms.

Table 10.1 in the Appendix shows the results of using the *Weight Limit* configuration to generate default form matrices. The configuration is effective at filtering the search to a subset of possible generator matrices that the specific search task wishes to focus on. The choice of minimum and maximum row weights can impact the results greatly, as seen in the last two rows of the table.

In practice, both the *Weight Limit* and *Weight List* configurations are commonly used in addition to other configurations, to further constrain a given search.

Table 10.2 in the Appendix shows the results of using the *Weight List* configuration to generate default form matrices, with the exception that depending on the specific weight list used, the matrices may not be in weight sorted form.

Let a^n denote a sequence of a repeated n times. The weight lists used in the searches shown in the table are the following:

- $wl_1 = [8]$: The first row of the generator matrix must have a weight of 8. Other rows have no constraints.

- $wl_2 = [14, 6, 8^{12}]$: The first and second rows of the generator matrix must have weights 14 and 6 respectively. Other rows must have a weight of 8.
- $wl_3 = [[30, 26, 22, 18, 14]^{25}]$: All rows must have a singly even weight w, $14 \leq w \leq 30$.
- $w_4 = [28, 12^{25}]$: The first row must have a weight of 28, all other rows must have a weight of 12.

Note that due to algorithmic incompatibility, the *Weight List* configuration can sometimes give unsuitable results when used with the clique-based search algorithm. Given a matrix A' with k' rows, the clique-based search creates candidate codewords using the conditions constraining row $k' + 1$. Further, the algorithm assumes that the nodes present in the graph it builds can be at any available row position in the final generator matrix. These issues do not present a problem if the constraints for all rows with index $i > k'$ are the same, such as the examples shown in table 10.2.

10.2 Permutation Equivalence

Section 6.3 and chapter 8 have briefly described how in most search scenarios, especially during classification tasks, we want to avoid having linear codes among the search results that are permutation equivalent. Having permutation equivalent codes in the search tree can greatly increase its size, and thus the traversal time. Many elements of the search algorithms already presented aim at reducing the number of permutation equivalent search results.

The Torch framework also provides a configuration, named *Perm. Equiv.*, with the purpose of filtering out all permutation equivalent nodes from the search trees.

When generating combinatorial objects, there are a number of known approaches to preventing the creation of isomorphic results. These techniques are discussed in [61]. The applicability of these generation approaches depend on the specific properties of the objects being generated.

The *Perm. Equiv.* configuration uses the basic technique of recorded objects. Using this approach, during the traversal of the search tree, a global record $\mathscr{R}$ is kept of the linear codes that have been previously encountered during the search. The record contains a reference to one code from each permutation equivalent set of linear codes that have been found during execution. When a node in the tree is being traversed, the code contained in the node is compared

to the codes stored in $\mathscr{R}$. If the code is found to be permutation equivalent to a code in $\mathscr{R}$, then the node is rejected. In this way, the entire subtree starting at the rejected node is pruned from the search tree.

In practice, the *Perm. Equiv.* configuration adds one new condition to the search algorithm, that is added to the list of conditions injected into the *children* module. Thus the condition rejects permutation equivalent nodes when they are checked by the search if they need to be added to the search tree. The condition added by the configuration implements the technique of recorded objects as described above.

An important note about the approach of recorded objects is that in order to ensure that each set of isomorphic objects still has one representative returned as a result, the search tree must be such that for any pair of isomorphic nodes, the children nodes must also be pairwise isomorphic. Let $A \cong B$ denote objects A and B being isomorphic, and let $C(A)$ denote the set of children nodes of A. Then, for all nodes A and B where $A \cong B$, if $A' \in C(A)$, there must exist a $B' \in C(B)$ such that $A' \cong B'$.

When searching for linear codes using Torch, the above conditions does not hold if the generator matrices are created in ordered form. Let G_1 and G_2 be generator matrices in ordered form generating codes C_1 and C_2 respectively, such that C_1 is permutation equivalent to C_2. If G_1' is a generator matrix in ordered form generating C_1', created by augmenting G_1 with one codeword, there is no guarantee that a G_2' matrix exists, created by augmenting G_2, that generates a permutation equivalent code and is also in ordered form.

The *Perm. Equiv.* configuration includes the option *ordered=False* in order to make sure that classification tasks can be carried out correctly.

In practice, the configuration has two limits that need to considered when using. Because the generator matrices created are not limited to ordered form, the *solutions* module can take significantly more time creating candidate codewords for each matrix. This increase in potential runtime is typically exceeded by the reduction achieved by the search tree being significantly smaller. The other limitation is that the record of previously encountered codes $\mathscr{R}$ can be very large even for moderate values of (n, k, d), making the space usage of the algorithm a significant constraint in practice.

10.2.1 Experimental results

The table referenced in this subsection presents the computational results of using the *Permutation Equivalence* configuration described in section 10.2. Like the experimental results in previous sections, the implementation was run on a

laptop with AMD Ryzen 5 7640HS CPU (4.30GHz), in a SageMath environment executed inside a Linux virtual machine.

The configuration is added to the default search algorithm as described in section 8.1, using the depth-first search.

Table 10.3 in the Appendix shows the (n, k, d) parameters and the type of code being searched, the number of codes found and the execution time of the search. The generator matrices created are in default form.

The configuration is useful for classification problems, although not exclusively. The last row of the table shows a search task where the goal was to generate 100 non-equivalent $(40, 18, 8)$ Type I codes. In practice, permutation equivalence checking is often used together with other configurations to decrease the size of the search tree.

10.3 Paperclip

The ultimate motivation of the *Paperclip* configuration is to create a method that fulfills multiple criteria: it should be suitable for both classification and targeted search tasks, the search results should very rarely contain permutation equivalent codes, and the configuration should avoid the drawbacks of the *Perm. Equiv.* configuration described in section 10.2.

McKay presents an algorithm for isomorph-free generation of combinatorial objects in [63], based on labeling canonical objects in an isomorphism class. Methods for generating linear codes already exist, such as GenSelfDual by Bouyukliev et al. [50]. This method is a very efficient solution to a more constrained version of the problem considered in this paper. A comparison of this algorithm and the algorithm presented here is in a following subsection.

Given parameters (n, k, d) that we want to search for, the ideal algorithm we wish to build has the following properties:

- It is a depth-first search algorithm. The root node of the tree is an $(n, 1, d)$ code. During each step, the generator matrix is augmented by one new line, such that at depth k', all nodes are (n, k', d) codes. We find the codes we are searching for at depth k.

- Let $P(C)$ be the set of linear codes permutation equivalent to C. Let $R_{P(C)}$ be a uniquely determined linear code from $P(C)$, called the *representative* of $P(C)$. For each set $P(C)$ for an (n, k, d) code C, the tree includes $R_{P(C)}$ as a leaf node, and no other codes from $P(C)$ are present in the tree.

- Let C' and C'' be two nodes in the tree such that C' is the parent of C''. If C'' is an (n, k'', d) code, then C' is a $(n, k''-1, d)$ code and $R_{P(C')} = C'$. In other words, the parent of a representative code is the representative of its own permutation equivalence set.

Based on these properties, the presented method is classified as a Read-Faradzev algorithm [64] [65]. This type of algorithm is used for solving isomorph-free generation of cubic graphs [66], 1-factorization of complete graphs [67] and matroids [68].

As a first step, the algorithm requires a uniquely determined generator matrix to be defined for any linear code C. We will show that the *reduced row echelon* form (RRE) is suitable for the algorithm.

We also define an ordering relation between linear codes based on comparing RRE generator matrices. Using this relation, we define a representative for a set of permutation equivalent codes that satisfies the property of inheritence.

10.3.1 Reduced row echelon form

The well-known reduced row echelon (RRE) form of a generator matrix is uniquely determined. In the presented algorithm, the usual RRE form of a matrix is flipped upside-down to make certain conditions easier to understand and compute.

Definition 10.3.1 (Reduced row echelon (RRE) form, pivot)**.** Let G be a matrix that has the following properties:

- The leftmost 1 in each row is called the leading 1. For each column that contains a leading 1, all other entries are 0. The leading 1 is also called the *pivot* of the row.
- For each row after the first, the leading 1 is to the left of the leading 1 in the row above it.

The reduced row echelon form of a matrix can be obtained by Gauss-Jordan elimination. For a linear code C, any generator matrix $G(C)$ will have the same reduced row echelon form. This uniquely determined generator matrix will be denoted as $G_{RRE}(C)$.

$G_{RRE}(C)$ has the following properties:

1. Let C be an (n, k, d) binary linear code, and u and v be two codewords of C. Let $u < v$ mean that u is smaller than v in lexicographical order.

Let $c_0, c_1, ..., c_{2^k-1}$ be the codewords of C such that for all $0 \leq i \leq 2^k - 1$, $c_i < c_{i+1}$. Then, the rows of the RRE generator matrix of C are the codewords $c_{2^j}, 0 \leq j \leq k-1$:

$$G_{RRE}(C) = \begin{pmatrix} c_{2^0} \\ c_{2^1} \\ \vdots \\ c_{2^{k-2}} \\ c_{2^{k-1}} \end{pmatrix}$$

Note that $G_{RRE}(C)$ is the generator matrix of C containing the smallest possible codewords with regards to lexicographical order.

2. Let m be a k-bit long binary string (the message we want to encode), and let m_b be the integer represented by m. Then, $mG_{RRE}(C) = c_{m_b}$.

   For any integer $0 \leq i \leq 2^k - 1$, let i^R be the k-bit long, reversed binary form of i, i.e. if $k = 4$ and $i = 1$, then $i^R = 1000$.

   Then, for all $0 \leq i \leq 2^k - 1$, $i^R G_{RRE}(C) = c_i$.

   In other words, $G_{RRE}(C)$ will encode i^R to the ith largest codeword in the code in lexicographical order.

3. Define the *codeword order* of a generator matrix G be the following sequence of codewords:

   $$0^R G, 1^R G, ..., (2^k - 1)^R G$$

   The codeword order of $G_{RRE}(C)$ is strictly monotonically increasing.

4. Let g_i and g_j be two arbitrary rows of $G_{RRE}(C)$, $i \neq j$. Then $g_i < g_i + g_j$ and $g_j < g_i + g_j$.

5. Let $I_j = \{1, 2, ..., j\}$ and $S \subseteq I_j$. Let $G_{RRE}(C)$ have rows $< g_1, g_2, \ldots, g_k >$. Then, for any $j < l \leq k$, $\sum_{i \in S} g_s < g_l$.

10.3.2 Ordering linear codes using RRE

Define an ordering on generator matrices, and by extension, linear codes, the following way: let G and G' be two $k \times n$ generator matrices, $G \neq G'$, with

rows $< g_1, g_2, \ldots, g_k >$ and $< g'_1, g'_2, \ldots, g'_k >$ respectively. Let $1 \leq i \leq k$ be the smallest integer where $g_i \neq g'_i$. If $g_i < g'_i$, then $G < G'$, otherwise $G' < G$.

Let C and C' be (n, k, d) and (n, k, d') linear codes, respectively. We define $C < C'$ when $G_{RRE}(C) < G_{RRE}(C')$.

10.3.3 Superminimal generator matrix

Definition 10.3.2 (Superminimal generator)**.** Let C be a binary linear code. Recall that $P(C)$ denotes the set of codes permutation equivalent to C. The representative element of $P(C)$, denoted $R_{P(C)}$, is the minimal element of $P(C)$ with respect to the ordering defined in subsection 10.3.2.

The RRE generator matrix of $R_{P(C)}$, denoted $G_{RRE}(R_{P(C)})$, is named the *superminimal generator* of $P(C)$.

Using the paperclip configuration, the nodes of the search tree contain the superminimal generators of codes.

Superminimal generator matrices have the following necessary (but not sufficient) properties:

- Let G_C have rows $< g_1, g_2, \ldots, g_{k-1}, g_k >$. If G_C is the superminimal generator matrix of $P(C)$, then $G_{C'}$ with rows $< g_1, g_2, \ldots, g_{k-1} >$ is the superminimal generator of $P(C')$.

  Suppose $G_{C'}$ is not superminimal. Then there exists a generator matrix $G^*_{C'}$ that generates a code in $P(C')$ and $G^*_{C'} < G_{C'}$. Then we can create $G^*_C = \begin{pmatrix} G^*_{C'} \\ g^*_k \end{pmatrix}$ to be a generator matrix that generates a code in $P(C)$ and $G^*_C < G_C$, which is a contradiction.

- Let G be a superminimal generator matrix, generating a $(n, 1, d)$ code. Then, G is of the form:

$$(\underbrace{0\ 0\ \ldots\ 0}_{\text{n-d}}\underbrace{1\ 1\ \ldots\ 1}_{\text{d}})$$

  Note that the first row of any superminimal generator for an (n, k, d) code is a superminimal generator for an $(n, 1, d)$ code.

- Let G be a superminimal generator. Denote the ith column of G with $g_{(i)}$. Then, for $1 \leq i \leq n-1$, $g_{(i)} < g_{(i+1)}$.

From these properties, it follows that any superminimal generator will have a *border* of 1s such that all values in the matrix above this border are 0. An example can be seen in Table 10.4, with the *border* highlighted.

A region of this border that falls on the same row will be called a *roof*. In the given example, the first row of the generator has a *roof* of length 4, starting at column 13, the second row has a *roof* of length 2, starting at column 11, and so on. Each roof extends from the pivot of a row to the column directly before the pivot of the row above it, or to the rightmost column in case of the first row.

0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1
0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1
0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0
0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0
0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0
1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0

Table 10.4: The superminimal generator of a (16,8,4) doubly even code

The search algorithm builds generator matrices by augmenting them one row at a time. After each augmentation, we need to decide if the new matrix is superminimal or not. Because of the typical size of $P(C)$ for any interesting code C, a naive approach to this decision would be extremely impractical. In the following subsection, some conditions are given that can be efficiently evaluated to help with this decision.

10.3.4 Conditions

The following conditions are necessary, but not sufficient for deciding if an RRE generator matrix G is superminimal.

Local minimum condition

Define linear codes C and C' to be *transposition neighbors* if C can be transformed into C' with a single transposition of two columns.

When calculating the superminimal generator of a set $P(C)$, it is known that a greedy algorithm that, given a code C iteratively keeps moving toward

the smallest transposition neighbor of C is not sufficient. That is, local minima exist: codes that are smaller than all of their transposition neighbors, but whose RRE generator is not superminimal.

The naive algorithm to check if a given RRE generator is a local minimum would be to generate all transposition neighbors by transposing two columns, using Gauss-Jordan elimination to get RRE forms, and compare them to the original generator. Since the number of transposition neighbors of an (n, k, d) code is $\binom{n}{2}$, this method is impractical for all but very small values of n.

The following condition is sufficient and necessary to check if a given RRE generator matrix G of a code C, that satisfies the properties described in Section 10.3.3, is a local minimum compared to its transposition neighbors.

Suppose that we wish to create a transposition neighbor of C by transposing two columns of G with indices a and b, $1 \leq a < b \leq n$. We can assume that columns a and b are not equal and neither are all-0 columns, since these cases are trivial. The transposition neighbor of C obtained from transposing columns a and b will be denoted C_{ba}. The goal is to decide if $C < C_{ba}$ without fully computing $G_{RRE}(C_{ba})$, denoted as G'_{ba}.

Let $row_i(M)$ denote the ith row of a matrix M, and let $row_i(M)_{ba}$ denote the ith row of a matrix M with the elements at positions a and b transposed. Similarly, let $col_i(M)$ denote the ith column of a matrix M.

To decide if $C < C_{ba}$, we need to answer the following questions: what is the smallest row index i such that $row_i(G) \neq row_i(G'_{ba})$, and is $row_i(G) < row_i(G'_{ba})$?

Two cases need to be considered:

1. Columns a and b cross the border of G on different roofs. See Figure 10.1 as an example. Let r and s be the indexes of the rows where columns b and a cross the generator's border, respectively. Clearly, $1 \leq r < s \leq k$.

   The following can be shown for the rows of G and G'_{ba}:

   - For all rows with index $j < r$, $row_j(G'_{ba}) = row_j(G)$.
   - If $j = r$, then $row_j(G) \leq row_j(G'_{ba})$, since the position of the pivot in $row_r(G'_{ba})$ cannot be to the right of the position of the pivot in $row_r(G)$.
   - For all rows with index $r < j < s$, a j' row exists such that the position of the pivot in $row_j(G)$ equals the positions of the pivot in $row_{j'}(G'_{ba})$. Note that these pivot positions are all to the left of the pivot position of row r.

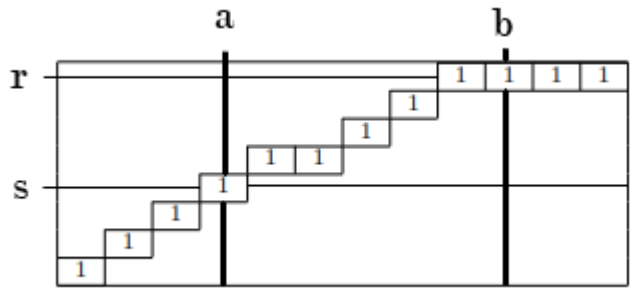

Figure 10.1: Columns a and b cross the border on different roofs

- If $j = s$, then two cases need to be considered:
  - If column a is not the position of the pivot in row s, then the position of the pivot of $row_s(G)$ equals the position of the pivot of $row_s(G'_{ba})$.
  - If column a is the position of the pivot in row s, then one step of Gaussian elimination may be required. If the value of g_{sb} is 1, then let $v = row_s(G)_{ba} + row_r(G)_{ba}$, otherwise let $v = row_s(G)_{ba}$. If $v < row_r(G)$, then $C_{ba} < C$, otherwise $C \leq C_{ba}$. Note that if the length of the roof of row s in G is greater than 1, then $v \geq row_r(G)$ always holds.
- For all rows with index $j > s$, the position of the pivot in $row_j(G)$ equals the position of the pivot in $row_j(G'_{ba})$. Note that these pivot positions are all to the left of the pivot position of row r.

It is clear that the smaller index i where $row_i(G) \neq row_i(G'_{ba})$ in this case is $i = r$. Based on the previous observations, we can also see that the only case where $C > C_{ba}$ is possible is when row s has a roof of length 1, and $v < row_r(G)$ holds for the vector v computed as described previously.

2. Columns a and b cross the border of G on the same roof. See Figure 10.2 as an example.

   If column a is not the position of the pivot in row r, then we only need to compare the binary values of columns a and b. If $col_a(G) < col_b(G)$ holds, then $C < C_{ba}$.

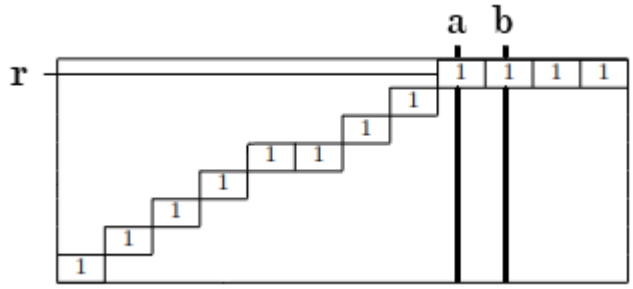

Figure 10.2: Columns a and b cross the border on the same roof

If column a is the position of the pivot in row r, then the following can be shown:

- Since by definition of the RRE form, the only 1 in column a is the pivot in row r, $col_a(G) < col_b(G)$ always holds.
- For all rows j, the position of the pivot in $row_j(G'_{ba})$ equals the position of the pivot in $row_j(G)$.
- For rows $j \leq r$, $row_j(G'_{ba}) = row_j(G)$.
- Let $t > r$ be the smallest row index such that $g_{tb} = 1$. Let $v = row_r(G_{ba}) + row_t(G_{ba})$. If $v < row_t(G)$, then $C > C_{ba}$, otherwise $C < C_{ba}$.

Here, the only case where $C > C_{ba}$ is possible is the last case described previously.

Based on these observations, we can efficiently determine if a code C is smaller than a transposition neighbor C_{ba} for any a and b, with just a few bitwise additions and comparisons, without having to compute G'_{ba}. This provides a practical method to determine if C is a local minimum compared to its transposition neighbors.

Border subcode condition 1

Let $G[a, b, c, d]$ denote a submatrix of G created by truncating G to rows from index a to b and to columns from index c to d, inclusive. See Figure 10.3 for

an example where $G' = G[4, 7, 2, 6]$. The linear code C' generated by $G' = G[a, b, c, d]$ will be called a *truncated subcode* of the linear code C generated by G.

If $G[a, b, c, d]$ is such that both the lower left and the upper right positions of the submatrix are on the border of G, we will call $G[a, b, c, d]$ a *border submatrix* of G and the code generated by G' a *border subcode* of C.

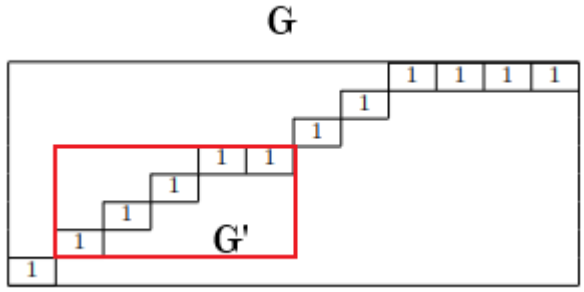

Figure 10.3: G' is a border submatrix of G

If G is a superminimal generator, generating a linear code C, then for all border submatrices G' of G, the minimum distance of the border subcode generated by G' must equal the weight of the first row of G'. Otherwise, a generator matrix G^* could be constructed such that the linear code C^* generated by G^* is permutation equivalent to C and $C^* < C$.

Suppose that a $G' = G[a, b, c, d]$ border submatrix exists, generating C', such that $d(C') < w(row_1(G'))$, where $d(C')$ denotes the minimum weight of C'. Then, by adding rows and permuting rows and columns of G', a G'' matrix can be created such that $w(row_1(G'')) = d(C')$ and $row_1(G'')$ is of the form where all positions with a value of 1 are to the right of all positions with a value of 0. The C'' code generated by G'' is permutation equivalent to C', and $C'' < C'$, because $row_1(G'') < row_1(G')$.

Clearly, by adding rows and permuting rows and columns of G, we can create a matrix G^* such that $G^*[a, b, c, d] = G''$. Then, $C^* < C$ holds for the code C^* generated by G^*, therefore G cannot be superminimal.

Border subcode condition 2

Let $G' = G[a, b, c, d]$ be a border submatrix such that $a > 1$, the lower left position of G' is the position of the leading 1 in $row_b(G)$, and the upper right position is directly to the left of the position of the leading 1 in $row_{a-1}(G)$. Then, G' will be called a *full border submatrix*, and $G'' = G[a, b, d + 1, n]$ will be called the *companion submatrix* of G', generating the *companion code* C''. See Figure 10.4 for an example. The first row of G'' will be denoted g^*.

We can assume that the condition described in the previous subsection holds, i.e., the minimum distance of the code generated by G' equals the weight of the first row of G'.

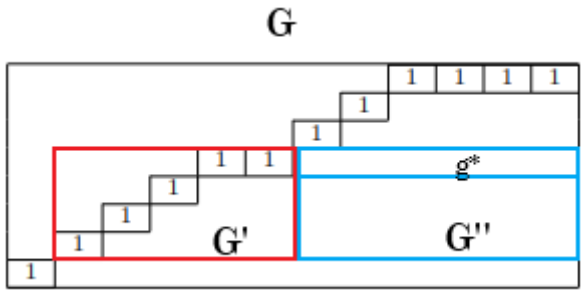

Figure 10.4: G" is the companion submatrix of G'

For a codeword $c \in C$ and a generator matrix G with rows $< g_1, g_2, \ldots, g_k >$, let $I_G(c) \subseteq \{1, 2, \ldots, k\}$ denote the set of indices such that $c = \sum_{i \in I_G(c)} g_i$.

Let G be a superminimal generator, generating a linear code C, let $G' = G[a, b, c, d]$ generating C' be a full border submatrix for which the condition described in Section 10.3.4 holds, and G'' with rows $< g''_1, g''_2, \ldots, g''_{b-a+1} >$ be the companion submatrix of G'. Then, the following condition also holds:

Let $S = \{c \in C' | w(c) = d(C')\}$ be the set of codewords with minimal weight in C'.

Then, for all $c \in S$, $\sum_{i \in I_{G'}(c)} g''_i \geq g^*$.

In other words, taking the minimal weight codewords of C' and computing $I_{G'}(c)$, the corresponding codeword in the companion code cannot have a binary form smaller than g^* if G is superminimal.

Suppose that a codeword c exists such that $\hat{g} = \sum_{i \in I_{G'}(c)} g_i''$ and $\hat{g} < g^*$. Then, similarly to the way described in Section 10.3.4, by adding rows and permuting rows and columns of G, we could create a matrix $\hat{G}$ such that $row_i(G) = row_i(\hat{G})$ for $i < a$, and $row_a(\hat{G})$ equals $row_a(G)$ except g^* is replaced by $\hat{g}$. Clearly, $\hat{C} < C$ holds for the linear code $\hat{C}$ generated by $\hat{G}$, therefore G cannot be superminimal.

Note that in practice, the number of codewords c that need to be checked for this condition will usually be low, as the minimum distance $d(C')$ is typically small, and $|I_{G'}(c)| \leq d(C')$.

Automorphism group condition

We denote the symmetric group on n elements by S_n, i.e. the group of all permutations on a set with n elements. Recall that $Aut(C) = \{p \in S_n | p(C) = C\}$ denotes the *automorphism group* of C, that is, the group of all column permutations of C that do not change the linear code.

For a generator G with rows $< g_1, g_2, \ldots, g_k >$ generating a linear code C, let C^i denote the linear code generated by G^i with rows $< g_1, g_2, \ldots, g_i >$, $1 \leq i \leq k$.

If G is a superminimal generator, then for all $1 \leq j < k$ and all $p \in Aut(C^j)$, $g_{j+1} \leq p(g_{j+1})$ must hold.

In other words, for row g_{j+1}, the permutations in the automorphism group of the code generated by the previous rows $Aut(C^j)$ can not make the binary form of g_{j+1} smaller if G is superminimal.

Suppose that $p \in Aut(C^j)$ exists such that $p(g_{j+1}) \leq g_{j+1}$. Then, $G^{j+1'}$ with rows $< g_1, g_2, \ldots, g_j, p(g_{j+1}) >$ would be a generator matrix generating a code $C^{j+1'}$ that is permutation equivalent to C^{j+1} and $C^{j+1'} < C^{j+1}$. Therefore, $p(C) < C$, which means G cannot be superminimal.

Recall that our goal is to create an algorithm that builds superminimal generators row by row, using the conditions described in Section 10.3.4 to prune the search tree of the depth-first search. During the execution of the algorithm, at depth l of the search, we have a generator matrix G_l with rows $< g_1, g_2, \ldots, g_l >$ that fulfills the four conditions described in this section, and we want to find vectors g_{l+1} such that G_{l+1} with rows $< g_1, g_2, \ldots, g_l, g_{l+1} >$ also fulfills all conditions.

In practice, this means that at depth l, we only need to check the condition described in this subsection for $Aut(C^l)$ and the vectors g_{l+1} we want to use to augment G_l.

Note that depending on the (n, k, d) parameters of the code we are searching for, $Aut(C^l)$ can be so large that exhaustive testing of all $p \in Aut(C^l)$ is not feasible. Practical application has shown that even restricting this condition to only try a limited number of permutations from $Aut(C^l)$ greatly helps in pruning the search tree from nodes that are not superminimal.

10.3.5 Experimental results

Table 10.5 in the Appendix gives some results of exhaustive searches using the paperclip algorithm. Like the experimental results in previous sections, the implementation was run on a laptop with AMD Ryzen 5 7640HS CPU (4.30GHz), in a SageMath environment executed inside a Linux virtual machine.

All codes present in the table are self-dual, with the exception of the code in the last row.

As an example, the following tables found in the Appendix present some of the superminimal generator matrices found using the paperlcip algorithm:

- Table 10.6: superminimal generator for $(16, 8, 4)$ Type I code.
- Table 10.7: superminimal generators for $(16, 8, 4)$ Type II codes.
- Table 10.8: superminimal generator for $(22, 11, 6)$ Type I code.
- Table 10.9: generators for $(10, 5, 4)$ codes that pass all conditions, but are permutation equivalent. These generators are the smallest known examples where the presented algorithm generates equivalent codes.

10.4 Comparison to other algorithms

Other algorithms with similar goals as the *Paperclip* configuration already exist. An implementation, named GenSelfDual, by Bouyukliev et al. is described in [50]. The implementations are available online. These algorithms are specifically build for classification of self-dual codes, based on two theorems that can be used to create new self-dual codes given previously known self-dual codes:

- Given an $(n-2, n/2-1, d_1), d_1 \geq d-2$ self-dual binary code, Theorem 1 can be used to create $(n, n/2, d), d \geq 4$ self-dual binary codes. This theorem is used by GenSelfDualAllD.

- Given an $(n-4, n/2-2, d_1), d_1 \geq 2$ self-dual code, Theorem 2 can be used to create $(n, n/2, 4)$ self-dual binary codes. This theorem is used by GenSelfDualD4.

Compared to the Paperclip algorithm, the goal of GenSelfDual is more constrained:

- It is used to create only self-dual codes. The presented algorithm can be used to create self-orthogonal or non-orthogonal codes as well.
- The current implementation can only be used up to $n \leq 42$. The presented algorithm can be used for arbitrary n.
- GenSelfDual works by lengthening shorter codes. The common usage of the algorithms involves giving already known generator matrices as input and creating matrices by lengthening these generators. The presented algorithm starts with a 1-dimensional code as a root node. The length of the code does not change during the search. It should be noted that the presented algorithm can be modified in a straightforward manner to take larger dimensional matrices as a root.
- GenSelfDualAllD aims at constructing exhaustive lists of non-equivalent codes with $d \geq 2$. GenSelfDualD4 does the same for $d = 4$. In both cases, there is no provided way to filter the results during generation, e.g. if the goal is to only generate codes of a specific d, or other search criteria. This makes the algorithms poorly suited for targeted search for codes with specific properties or codes whose existence was previously unknown. The presented algorithm's structure and the Torch framework enables simple ways of incorporating arbitrary search filters.

GenSelfDual is supported by algorithms found in a software package named Q-Extension, also by Bouyukliev [69]. This package can be used for generating non self-dual codes, and do offer some filters for the search, although in a more limited number and in a less flexible way than those found in Torch.

GenSelfDual is exceptionally efficient at the goal it is made for. Table 10.10 in the Appendix gives some results using GenSelfDualAllD, using its default settings and using the $(4, 2, 2)$ self-dual code as its starting matrix.

Due to software dependency issues, the laptop used in the previous section could not be used to execute GenSelfDual. Instead, it is executed on a typical laptop with an Intel Core i7-7700HQ CPU (2.80GHz), on the native Windows operating system, as no Linux compatible version of GenSelfDual currently exists.

Bibliography

[1] A. Menezes, P. Van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.

[2] H. Niederreiter, *Random number generation and quasi-Monte Carlo methods*. Society for Industrial and Applied Mathematics, 1992.

[3] D. E. Knuth, *The art of computer programming, volume 2 (3rd ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[4] T. Herendi, "Construction of uniformly distributed linear recurring sequences modulo power of 2," *Uniform Distribution Theory*, vol. 13, no. 1, pp. 109–129, 2018.

[5] T. Herendi and S. R. Major, "Modular exponentiation of matrices on FPGA-s," *Acta Univ. Sapientiae, Inform.*, vol. 3, no. 2, pp. 172–191, 2011.

[6] T. Herendi and S. R. Major, "Efficient random number generation on fpgas," in *ICAI 2014: Proceedings of the 9th International Conference on Applied Informatics*, pp. 313–320, 2014.

[7] T. Herendi and S. R. Major, "Using irreducible polynomials for random number generation," *Annales Mathematicae et Informaticae*, vol. 56, p. 36–46, 2022.

[8] R. Cramer, Y. Dodis, S. Fehr, C. Padró, and D. Wichs, "Detection of algebraic manipulation with applications to robust secret sharing and fuzzy extractors," in *Advances in Cryptology – EUROCRYPT 2008*, pp. 471–488, Springer Berlin Heidelberg, 2008.

[9] R. J. McEliece, "A public-key cryptosystem based on algebraic," *Coding Thv*, vol. 4244, pp. 114–116, 1978.

[10] H. Niederreiter, "Knapsack-type cryptosystems and algebraic coding theory," *Prob. Contr. Inform. Theory*, vol. 15, no. 2, pp. 157–166, 1986.

[11] C. Hannusch and S. R. Major, "Torch: Software package for the search of linear binary codes," in *2022 IEEE 2nd Conference on Information Technology and Data Science (CITDS)*, pp. 103–106, 2022.

[12] C. Hannusch and S. R. Major, "Neighborhoods of binary self-dual codes," *Contemporary Mathematics*, vol. 4, pp. 1174–1179, 2023.

[13] R. Lidl and H. Niederreiter, *Encyclopedia of Mathematics and its Applications: Finite Fields*, vol. 20. Cambridge University Press, 1997.

[14] E. R. Berkelamp, "Factoring polynomials over finite fields," *The Bell System Technical Journal*, vol. 46, no. 8, pp. 1853–1859, 1967.

[15] D. Cantor and H. Zassenhaus, "A new algorithm for factoring polynomials over finite fields," *Mathematics of Computation*, vol. 36, no. 154, pp. 587–592, 1981.

[16] P. Naudin and Q. Claude, "Univariate polynomial factorization over finite fields," *Theoretical Computer Science*, vol. 191, pp. 1–36, 1998.

[17] M. Rabin, "Probabilistic algorithms in finite fields," *SIAM J. Comput.*, vol. 9, pp. 273–280, 1980.

[18] S. Gao and D. Panario, "Tests and constructions of irreducible polynomials over finite fields," in *Foundations of Computational Mathematics*, pp. 346–361, Springer Berlin Heidelberg, 1997.

[19] M. Ben-Or, "Probabilistic algorithms in finite fields," *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)*, pp. 394–398, 1980.

[20] V. Shoup, "New algorithm for finding irreducible polynomials over finite fields," *Mathematics of Computation*, vol. 54, no. 189, pp. 435–447, 1990.

[21] V. Shoup, "Fast construction of irreducible polynomials over finite fields," *Journal of Symbolic Computation*, vol. 17, no. 5, pp. 371–391, 1994.

[22] H. Meyn, "On the construction of irreducible self-reciprocal polynomials over finite fields," *Communication and Computing*, vol. 1, pp. 43–53, 1990.

[23] S. Cohen, "Polynomials over finite fields with large order and level," *Bull. Korean Math. Soc.*, vol. 24(2), pp. 83–96, 1987.

[24] NIST, "A statistical test suite for random and pseudorandom number generators for cryptographic applications." https://csrc.nist.gov/Projects/Random-Bit-Generation/Documentation-and-Software.

[25] V. Shoup, "Ntl: A library for doing number theory." https://libntl.org/.

[26] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, A. Rupp, and M. Schimmler, "How to break des for bc 8,980," 2009.

[27] T. W. Cusick and P. Stanica, "Chapter 2 - fourier analysis of boolean functions," in *Cryptographic Boolean Functions and Applications (Second Edition)*, pp. 7–29, Academic Press, second edition ed., 2017.

[28] W. L. Dunn and J. K. Shultis, "Chapter 3 - pseudorandom number generators," in *Exploring Monte Carlo Methods (Second Edition)*, pp. 55–110, Elsevier, second edition ed., 2023.

[29] K. Gyarmati, "On a family of pseudorandom binary sequences," *Period. Math. Hungar*, vol. 49, pp. 45–63, 2004.

[30] L. Goubin, C. Maudit, and A. Sárközy, "Construction of large families of pseudorandom binary sequences," *J. Number Theory*, vol. 106, pp. 56–69, 2004.

[31] C. Maudit and A. Sárközy, "Construction of pseudorandom binary sequences by using the multiplicative inverse," *Acta Math. Hungar.*, vol. 109, pp. 75–107, 2005.

[32] C. Maudit and A. Sárközy, "On finite pseudorandom binary sequences I: Measure of pseudorandomness, the Legendre symbol," *Acta Arith.*, vol. 82, no. 4, pp. 365–377, 1997.

[33] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, p. 3–30, jan 1998.

[34] L. Györfi, S. Győri, and I. Vajda, *Információ- és kódelmélet.* Budapest: Typotex, 2000.

[35] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[36] C. Shannon, "Communication in the presence of noise," *Proceedings of the IRE*, vol. 37, no. 1, pp. 10–21, 1949.

[37] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[38] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[39] A. Hocquenghem, "Codes correcteurs d'erreurs," *Chiffres*, pp. 147–156, 1959.

[40] R. Bose and D. Ray-Chaudhuri, "On a class of error correcting binary group codes," *Information and Control*, vol. 3, no. 1, pp. 68–79, 1960.

[41] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, 1967.

[42] R. Singleton, "Maximum distance q-nary codes," *IEEE Transactions on Information Theory*, vol. 10, no. 2, pp. 116–118, 1964.

[43] D. Joyner and J. L. Kim, *Selected unsolved problems in coding theory.* Boston, MA: Birkhäuser, 2011.

[44] V. Pless, R. A. Brualdi, and W. C. Huffman, *Handbook of coding theory.* Elsevier Science Inc., 1998.

[45] R. Overbeck and N. Sendrier, *Code-based cryptography. In: Post-Quantum Cryptography*, pp. 95–145. Springer Berlin Heidelberg, 2009.

[46] M. Tomlinson, C. J. Tjhai, M. A. Ambroze, M. Ahmed, and M. Jibril, *Good Binary Linear Codes*, pp. 101–136. Springer International Publishing, 2017.

[47] N. J. A. Sloane, M. R. Sudhakar, and C. L. Chen, "New binary codes," *IEEE Trans. Inf. Theory*, vol. 18, pp. 503–510, 1972.

[48] W. Alltop, "A method for extending binary linear codes (corresp.)," *IEEE Transactions on Information Theory*, vol. 30, no. 6, pp. 871–872, 1984.

[49] E. M. Rains and N. J. A. Sloane, *Self-dual codes. In: Handbook of coding theory*, pp. 177–294. Amsterdam: Elsevier, 1998.

[50] I. Bouyukliev, S. Bouyuklieva, M. Dzhumalieva-Stoeva, and V. Monev, "What is genselfdual?," *Serdica Journal of Computing*, vol. 10, no. 3-4, pp. 231–244, 2016.

[51] S. T. Dougherty, J. L. Kim, and P. Solé, "Open problems in coding theory," *Contemp. Math*, vol. 634, pp. 79–99, 2015.

[52] J. H. Conway and N. J. A. Sloane, "A new upper bound on the minimal distance of self-dual codes," *IEEE Transactions on Information Theory*, vol. 36, no. 6, pp. 1319–1333, 1990.

[53] M. Harada and K. Saito, "Singly even self-dual codes constructed from hadamard matrices of order 28," *Australasian Journal of Combinatorics*, vol. 70, no. 2, pp. 288–296, 2018.

[54] T. A. Gulliver and M. Harada, "On doubly circulant doubly even self-dual [72, 36, 12] codes," *Australasian Journal of Combinatorics*, vol. 40, pp. 137–144, 2008.

[55] S. Bouyuklieva, "Self-dual codes with some applications to cryptography." NATO Advanced Research Workshop, 2008.

[56] "Sagemath - open-source mathematical software system.." https://www.sagemath.org.

[57] "Gap - groups, algorithms, programming - a system for computational discrete algebra." https://www.gap-system.org/Packages/guava.html.

[58] "Magma computational algebra system." http://magma.maths.usyd.edu.au/magma/.

[59] "Inversion of control containers and the dependency injection pattern." https://martinfowler.com/articles/injection.html.

[60] S. van Deursen and M. Seemann, *Dependency Injection Principles, Practices, and Patterns*. Manning Publications, 2019.

[61] P. Kaski and P. R. Östergård, *Isomorph-Free Exhaustive Generation. In: Classification Algorithms for Codes and Designs*, pp. 105–143. Springer Berlin Heidelberg, 2006.

[62] S. M. Dodunekov, "A comment on the weight structure of generator matrices of linear codes," *Problemy Peredachi Informatsii*, vol. 26, no. 2, p. 101–104, 1990.

[63] B. D. McKay, "Isomorph-free exhaustive generation," *Journal of Algorithms*, vol. 26, no. 2, pp. 306–324, 1998.

[64] R. C. Read, "Every one a winner," *Annals Discrete Math.*, vol. 2, p. 107–120, 1978.

[65] I. A. Faradzev, "Constructive enumeration of combinatorial objects," *Problemes Combinatoires et Theorie des Graphes Colloque Internat, CNRS 260 CNRS Paris*, p. 131–135, 1978.

[66] G. Brinkmann, "Fast generation of cubic graphs," *J. Graph Theory*, vol. 23, p. 139–149, 1996.

[67] J. H. Dinitz, D. Garnick, and B. D. McKay, "There are 526,915,620 non-isomorphic one-factorizations of k 12," *J. Combinatorial Designs*, vol. 2, p. 273–285, 1994.

[68] Y. Matsumoto, S. Moriyama, H. Imai, and D. Bremner, "Matroid enumeration for incidence geometry," *Discrete Comput. Geom.*, vol. 47(1), p. 17–43, 2012.

[69] I. Bouyukliev, "What is q-extension?," *Serdica Journal of Computing*, vol. 1, no. 2, pp. 115–130, 2007.

Part III

Appendix

1.1 Chapter 3

i	$P(\forall I(x) \mid S(x) : deg(I) > i)$	$deg(T_i)$
1	0.25	2
2	0.1875	4
3	0.1436	10
4	0.1183	22
5	0.0978	52
6	0.0848	106
7	0.0737	232
8	0.0655	472
9	0.0587	976
10	0.0533	1966
11	0.0487	4012
12	0.0449	8032
13	0.0415	16222
14	0.0387	32476
15	0.0362	65206
16	0.034	130486
17	0.0321	261556
18	0.0303	523132
19	0.0288	1047418
20	0.0274	2094958

Table 3.1: Proportion of polynomials with no small factors

Statistical Test	$\mathcal{L}_1$		$\mathcal{L}_2$	
	P-value	Proportion	P-value	Proportion
Frequency	0.779188	100/100	0.955835	100/100
Runs	0.514124	100/100	0.108791	98/100
FFT	0.924076	99/100	0.678686	98/100
OverlappingTemplate	0.012650	96/100	0.035174	97/100
Universal	0.935716	97/100	0.249284	100/100
LinearComplexity	0.699313	99/100	0.719747	100/100

Table 3.2: NIST test results of $\mathcal{L}_1$ and $\mathcal{L}_2$ generators

1.2 Chapter 4

Listing 4.1: Block matrix multiplication

```
Function block_matrix_mult(A_{i*}, B_{*j}):
Input:
    matrix over GF(4): A_{i*}, B_{*j}
begin
    for p in 0 to 15:
        for q in 0 to 15:
            s = 0_{16,16}
            for r in 0 to 63:
                s = M̂(a_{pr}, b_{rq}, s)
            c_{pq} = s
    return C_{ij}
end
```

repetitions	n	RIVYERA
10000	64	18.1 ms
10000	128	17.7 ms
10000	256	18.1 ms
10000	512	17.9 ms
10000	1024	17.7 ms

Table 4.1: Performance of matrix multiplication on RIVYERA

repetitions	n	*mat_ZZ_p*
10000	64	1.01 ms
10000	128	4.25 ms
10000	256	18.2 ms
10000	512	81.9 ms
10000	1024	430 ms

repetitions	n	*mat_zz_p*
10000	64	43.2 µs
10000	128	316 µs
10000	256	2.09 ms
10000	512	14.6 ms
10000	1024	92.9 ms

Table 4.2: Performance of matrix multiplication using NTL

repetitions	n	$\mathbb{Z}/n\mathbb{Z}$ (n is small)
10000	64	39.2 µs
10000	128	299 µs
10000	256	2.29 ms
10000	512	17.3 ms
10000	1024	126 ms

repetitions	n	FQ
10000	64	3.78 ms
1000	128	16.1 ms
1000	256	70.3 ms
1000	512	329 ms
1000	1024	1.64 s

Table 4.3: Performance of matrix multiplication using FLINT

repetitions	n	FQ NMOD
10000	64	3.1 ms
1000	128	13.6 ms
1000	256	60.3 ms
1000	512	275 ms
1000	1024	1.42 s

repetitions	n	FQ ZECH
10000	64	4.73 ms
1000	128	19.3 ms
1000	256	86.2 ms
1000	512	391 ms
1000	1024	1.82 s

Table 4.4: Performance of matrix multiplication using FLINT

Listing 4.2: Polynomial modulo over GF(2)

```
Function polynomial_mod(P(x),Q(x),n,m):
Input:
    polynomial over GF(2): P(x),Q(x)
    integer: n,m
begin
    s = m - n
    P'(x) = P(x)x^s
    c = 0
    while c < s:
        if q_m = 1:
            Q(x) = Q(x) XOR P'(x)
        else:
            Q(x) = Q(x)x
            c = c + 1
    if q_m = 1:
        Q(x) = Q(x) XOR P'(x)
    return Q(x)/x^s
end
```

Listing 4.3: Iteration-F_i

```
Function iter_fi(P'_i, Q'_i, s, b_T, MSW_P, MSW_Q, wb_Qi, wb_Pi, wf_Pi):
Input:
    64Kb polynomial segments: P'_i, Q'_i
    integer: s, b_T
    64 bit word: MSW_P, MSW_Q, wb_Qi, wb_Pi, wf_Pi
begin
    v = 0
    wf_Qi = 0
    while v < 64:
        if b_T = s:
            finish the computation
        elif MSB(MSW_Q) = 1:
            Q'_i = Q'_i XOR P'_i
            MSW_Q = MSW_Q XOR MSW_P
            wb_Qi = wb_Qi XOR wb_Pi
            wf_Qi = wf_Qi XOR wf_Pi
        else:
            v = v + 1
            b_T = b_T + 1
            wf_Qi = (wf_Qi ≪ 1) + MSB(Q'_i)
            Q'_i = (Q'_i ≪ 1) + MSB(wb_Qi)
            wb_Qi = wb_Qi ≪ 1
    send wf_Qi to F_{i+1}
end
```

Listing 4.4: Iteration-F_{15}

```
Function iter_f15(P'_15, Q'_15, s, b_T, MSW_P, MSW_Q, wb_Q15, wb_P15):
Input:
    64Kb polynomial segments: P'_15, Q'_15
    integer: s, b_T
    64 bit word: MSW_P, MSW_Q, wb_Q15, wb_P15
begin
    v = 0
    while v < 64:
        if b_T = s:
            finish the computation
        elif MSB(MSW_Q) = 1:
```

```
            Q'_15 = Q'_15 XOR P'_15
            MSW_Q = MSW_Q XOR MSW_P
            wb_Q15 = wb_Q15 XOR wb_P15
        else:
            v = v + 1
            b_T = b_T + 1
            Q'_15 = (Q'_15 ≪ 1) + MSB(wb_Q15)
            wb_Q15 = wb_Q15 ≪ 1
    MSW_Q = MSW(Q'_15)
    send MSW_Q to each other FPGA
end
```

Listing 4.5: Iteration-F_0

```
Function iter_f0 (P'_0, Q'_0, s, b_T, MSW_P, MSW_Q, wb_Q0, wf_P0):
Input:
    64Kb polynomial segments: P'_0, Q'_0
    integer: s, b_T
    64 bit word: MSW_P, MSW_Q, wb_Q0, wf_P0
begin
    v = 0
    wf_Q0 = 0
    while v < 64:
        if b_T = s:
            finish the computation
        elif MSB(MSW_Q) = 1:
            Q'_0 = Q'_0 XOR P'_0
            MSW_Q = MSW_Q XOR MSW_P
            wf_Q0 = wf_Q0 XOR wf_P0
        else:
            v = v + 1
            b_T = b_T + 1
            wf_Q0 = (wf_Q0 ≪ 1) + MSB(Q'_0)
            Q'_0 = (Q'_0 ≪ 1) + MSB(wb_Q0)
            wb_Q0 = wb_Q0 ≪ 1
    send wf_Q0 to F_1
end
```

n	deg(P)	deg(Q)	avg. t	n	deg(P)	deg(Q)	avg. t
10000	100	100	23.8 ms	10000	100	200	45.5 ms
10000	1000	1000	23.8 ms	10000	1000	2000	45.9 ms
10000	10000	10000	30.6 ms	10000	10000	20000	49 ms
10000	100000	100000	42.9 ms	10000	100000	200000	66.8 ms
10000	1000000	1000000	46.8 ms	10000	1000000	2000000	254 ms

Table 4.5: Performance of polynomial modulo on RIVYERA

n	deg(P)	deg(Q)	avg. t	n	deg(P)	deg(Q)	avg. t
1000000	100	100	164 ns	1000000	100	200	1.49 µs
1000000	1000	1000	210 ns	1000000	1000	2000	3.62 µs
100000	10000	10000	682 ns	100000	10000	20000	56.3 µs
10000	100000	100000	4.97 µs	10000	100000	200000	1.26 ms
1000	1000000	1000000	60.1 µs	1000	1000000	2000000	36.8 ms

Table 4.6: Performance of polynomial modulo using NTL

n	deg(P)	deg(Q)	avg. t	n	deg(P)	deg(Q)	avg. t
1000000	100	100	10.2 µs	1000000	100	200	94.3 µs
100000	1000	1000	87 µs	100000	1000	2000	1.71 ms
10000	10000	10000	936 µs	10000	10000	20000	26.7 ms
1000	100000	100000	9.82 ms	1000	100000	200000	369 ms
100	1000000	1000000	92.7 ms	100	1000000	2000000	4.71 s

Table 4.7: Performance of polynomial modulo using FLINT $\mathbb{Z}/n\mathbb{Z}$

n	deg(P)	deg(Q)	avg. t	n	deg(P)	deg(Q)	avg. t
1000000	100	100	918 ns	1000000	100	200	10.4 µs
100000	1000	1000	8.96 µs	100000	1000	2000	225 µs
10000	10000	10000	79.3 µs	10000	10000	20000	2.37 ms
1000	100000	100000	752 µs	1000	100000	200000	43.9 ms
100	1000000	1000000	9.58 ms	100	1000000	2000000	698 ms

Table 4.8: Performance of polynomial modulo using FLINT $\mathbb{Z}/n\mathbb{Z}$ with small n

n	deg(P)	deg(Q)	avg. t	n	deg(P)	deg(Q)	avg. t
1000000	100	100	33.6 µs	1000000	100	200	324 µs
10000	1000	1000	315 µs	10000	1000	2000	5.02 ms
1000	10000	10000	2.86 ms	1000	10000	20000	69.2 ms
100	100000	100000	28.8 ms	100	100000	200000	897 ms
50	1000000	1000000	296 ms	50	1000000	2000000	11.1 s

Table 4.9: Performance of polynomial modulo using FLINT finite fields

n	deg(P)	deg(Q)	avg. t	n	deg(P)	deg(Q)	avg. t
1000000	100	100	48 µs	1000000	100	200	372 µs
10000	1000	1000	445 µs	10000	1000	2000	6.64 ms
1000	10000	10000	4.37 ms	1000	10000	20000	99.7 ms
100	100000	100000	43.5 ms	100	100000	200000	1.28 s
50	1000000	1000000	483 ms	50	1000000	2000000	15.8 s

Table 4.10: Performance of polynomial modulo using FLINT finite fields with small characteristic

n	deg(P)	deg(Q)	avg. t	n	deg(P)	deg(Q)	avg. t
1000000	100	100	54 µs	1000000	100	200	158 µs
10000	1000	1000	525 µs	10000	1000	2000	5.7 ms
1000	10000	10000	5 ms	1000	10000	20000	99.7 ms
100	100000	100000	52.4 ms	100	100000	200000	1.46 s
50	1000000	1000000	518 ms	50	1000000	2000000	18.8 s

Table 4.11: Performance of polynomial modulo using FLINT finite fields using Zech logarithm

n	deg(P)	deg(Q)	avg. t	n	deg(P)	deg(Q)	avg. t
1000000	100	100	1.95 µs	1000000	100	200	2.96 µs
1000000	1000	1000	27.5 µs	1000000	1000	2000	32.4 µs
100000	10000	10000	1.04 ms	100000	10000	20000	1.09 ms
10000	100000	100000	20.3 ms	10000	100000	200000	21.2 ms
1000	1000000	1000000	340 ms	1000	1000000	2000000	381 ms

Table 4.12: Performance of polynomial GCD using NTL

n	deg(P)	deg(Q)	avg. t	n	deg(P)	deg(Q)	avg. t
1000000	100	100	285 µs	1000000	100	200	372 µs
10000	1000	1000	8.55 ms	10000	1000	2000	9.56 ms
1000	10000	10000	159 ms	1000	10000	20000	175 ms
50	100000	100000	2.42 s	50	100000	200000	2.8 ms
10	1000000	1000000	32.4 s	10	1000000	2000000	36.5 s

Table 4.13: Performance of polynomial GCD using FLINT $\mathbb{Z}/n\mathbb{Z}$

n	deg(P)	deg(Q)	avg. t	n	deg(P)	deg(Q)	avg. t
1000000	100	100	29 µs	1000000	100	200	55 µs
100000	1000	1000	1.38 ms	100000	1000	2000	1.63 ms
1000	10000	10000	23.5 ms	1000	10000	20000	26.8 ms
100	100000	100000	394 ms	100	100000	200000	450 ms
10	1000000	1000000	7.02 s	10	1000000	2000000	7.94 s

Table 4.14: Performance of polynomial GCD using FLINT $\mathbb{Z}/n\mathbb{Z}$ with small n

n	deg(P)	deg(Q)	avg. t	n	deg(P)	deg(Q)	avg. t
1000000	100	100	1.05 ms	1000000	100	200	1.31 ms
10000	1000	1000	25.9 ms	10000	1000	2000	30.9 ms
500	10000	10000	386 ms	500	10000	20000	494 ms
50	100000	100000	5.26 s	50	100000	200000	6.85 s
10	1000000	1000000	69.8 s	10	1000000	2000000	89.1 s

Table 4.15: Performance of polynomial GCD using FLINT finite fields

n	deg(P)	deg(Q)	avg. t	n	deg(P)	deg(Q)	avg. t
1000000	100	100	1.3 ms	1000000	100	200	1.88 ms
10000	1000	1000	40.8 ms	10000	1000	2000	45.1 ms
500	10000	10000	688 ms	500	10000	20000	786 ms
50	100000	100000	9.85 s	50	100000	200000	11.8 s
10	1000000	1000000	129 s	10	1000000	2000000	137 s

Table 4.16: Performance of polynomial GCD using FLINT finite fields with small characteristic

n	deg(P)	deg(Q)	avg. t	n	deg(P)	deg(Q)	avg. t
1000000	100	100	359 µs	1000000	100	200	468 µs
10000	1000	1000	26.5 ms	10000	1000	2000	31.4 ms
500	10000	10000	640 ms	500	10000	20000	726 ms
50	100000	100000	10.9 s	50	100000	200000	11.9 s
10	1000000	1000000	154 s	10	1000000	2000000	166 s

Table 4.17: Performance of polynomial GCD using FLINT finite fields using Zech logarithm

1.3 Chapter 5

Statistical Test	$\mathcal{L}_1$		$\mathcal{L}_1^t$	
	P-value	Proportion	P-value	Proportion
Frequency	0.897763	100/100	0.010988	99/100
Runs	0.739918	99/100	0.224821	98/100
FFT	0.699313	100/100	0.040108	96/100
OverlappingTemplate	0.911413	100/100	0.474986	100/100
Universal	0.350485	99/100	0.000199	99/100
LinearComplexity	0.115387	99/100	0.275709	100/100

Table 5.1: NIST test results of $\mathcal{L}_1$ and $\mathcal{L}_1^t$ generators

Statistical Test	$\mathcal{L}_2$		$\mathcal{L}_2^t$	
	P-value	Proportion	P-value	Proportion
Frequency	0.071177	98/100	0.657933	100/100
Runs	0.911413	99/100	0.366918	100/100
FFT	0.971699	100/100	0.236810	99/100
OverlappingTemplate	0.062821	98/100	0.304126	98/100
Universal	0.637119	100/100	0.020548	99/100
LinearComplexity	0.275709	100/100	0.334538	99/100

Table 5.2: NIST test results of $\mathcal{L}_2$ and $\mathcal{L}_2^t$ generators

Statistical Test	$\mathcal{L}_3$		$\mathcal{L}_4^{(0)}$	
	P-value	Proportion	P-value	Proportion
Frequency	0.779188	100/100	0.798139	99/100
Runs	0.514124	100/100	0.534146	98/100
FFT	0.924076	99/100	0.678686	98/100
OverlappingTemplate	0.012650	96/100	0.030806	99/100
LinearComplexity	0.699313	99/100	0.062821	100/100

Table 5.3: NIST test results of $\mathcal{L}_3$ and $\mathcal{L}_4^{(0)}$ generators

Statistical Test	$\mathcal{L}_4^{(1)}$		$\mathcal{L}_4^{(2)}$	
	P-value	Proportion	P-value	Proportion
Frequency	0.153763	99/100	0.213309	100/100
Runs	0.171867	99/100	0.137282	98/100
FFT	0.006196	100/100	0.236810	100/100
OverlappingTemplate	0.003996	100/100	0.137282	99/100
LinearComplexity	0.534146	99/100	0.999777	96/100

Table 5.4: NIST test results of $\mathcal{L}_4^{(1)}$ and $\mathcal{L}_4^{(2)}$ generators

Statistical Test	$\mathcal{L}_4^{(3)}$		$\mathcal{L}_4^{(4)}$	
	P-value	Proportion	P-value	Proportion
Frequency	0.275709	99/100	0.759756	99/100
Runs	0.419021	98/100	0.554420	99/100
FFT	0.911413	95/100	0.071177	100/100
OverlappingTemplate	0.319084	100/100	0.289667	99/100
LinearComplexity	0.554420	98/100	0.171867	100/100

Table 5.5: NIST test results of $\mathcal{L}_4^{(3)}$ and $\mathcal{L}_4^{(4)}$ generators

Statistical Test	$\mathcal{L}_3$		$f_l(\mathcal{L}_3)$	
	P-value	Proportion	P-value	Proportion
Frequency	0.779188	100/100	0.366918	100/100
Runs	0.514124	100/100	0.289667	98/100
FFT	0.924076	99/100	0.055361	98/100
OverlappingTemplate	0.012650	96/100	0.739918	97/100
LinearComplexity	0.699313	99/100	0.249284	96/100

Table 5.6: NIST test results of $\mathcal{L}_3$ and $f_l(\mathcal{L}_3)$ generators

Statistical Test	$f_m(\mathcal{L}_3)$		$f_p(\mathcal{L}_3)$	
	P-value	Proportion	P-value	Proportion
Frequency	0.637119	99/100	0.798139	100/100
Runs	0.383827	99/100	0.978072	99/100
FFT	0.911413	99/100	0.096578	100/100
OverlappingTemplate	0.474986	97/100	0.964295	99/100
LinearComplexity	0.419021	97/100	0.883171	97/100

Table 5.7: NIST test results of $f_m(\mathcal{L}_3)$ and $f_p(\mathcal{L}_3)$ generators

Statistical Test	$f_s(\mathcal{L}_3)$		$f_n(\mathcal{L}_3)$	
	P-value	Proportion	P-value	Proportion
Frequency	0.000000	0/100	0.554420	99/100
Runs	0.000000	0/100	0.191687	99/100
FFT	0.000000	0/100	0.678686	99/100
OverlappingTemplate	0.000000	0/100	0.911413	98/100
LinearComplexity	0.262249	98/100	0.048716	98/100

Table 5.8: NIST test results of $f_s(\mathcal{L}_3)$ and $f_n(\mathcal{L}_3)$ generators

Statistical Test	$f_c(\mathcal{L}_3)$		$f_t(\mathcal{L}_3)$	
	P-value	Proportion	P-value	Proportion
Frequency	0.289667	99/100	0.935716	100/100
Runs	0.816537	100/100	0.759756	99/100
FFT	0.851383	99/100	0.000296	95/100
OverlappingTemplate	0.383827	99/100	0.798139	96/100
LinearComplexity	0.455937	98/100	0.739918	98/100

Table 5.9: NIST test results of $f_c(\mathcal{L}_3)$ and $f_t(\mathcal{L}_3)$ generators

Statistical Test	$\mathcal{L}_{Leg}$		$\mathcal{L}_{Gy}$	
	P-value	Proportion	P-value	Proportion
Frequency	0.213309	97/100	0.334538	100/100
Runs	0.554420	98/100	0.816537	98/100
FFT	0.779188	99/100	0.554420	99/100
OverlappingTemplate	0.978072	99/100	0.401199	98/100
Universal	0.798139	98/100	0.304126	97/100
LinearComplexity	0.129620	99/100	0.739918	100/100

Table 5.10: NIST test results of $\mathcal{L}_{Leg}$ and $\mathcal{L}_{Gy}$ generators

Statistical Test	$\mathcal{L}_{GMS}$		$\mathcal{L}_{MS}$	
	P-value	Proportion	P-value	Proportion
Frequency	0.191687	98/100	0.759756	98/100
Runs	0.419021	99/100	0.455937	98/100
FFT	0.334538	100/100	0.955835	98/100
OverlappingTemplate	0.122325	100/100	0.514124	97/100
Universal	0.171867	98/100	0.474986	99/100
LinearComplexity	0.759756	99/100	0.494392	98/100

Table 5.11: NIST test results of $\mathcal{L}_{GMS}$ and $\mathcal{L}_{MS}$ generators

1.4 Chapter 8

Listing 8.1: Simplified pseudocode of basic search generator method

```
Method get_search_results():
Dependencies:
    starters module: S,
    descendants module: D
begin
    for starter in S.get_starter_matrices():
        for descendant in D.get_descendants(starter):
            yield descendant
end
```

Listing 8.2: Simplified pseudocode of basic descendant generator method

```
Method get_descendants(G):
Input:
    generator matrix: G
Dependencies:
    children module: C
    yield conditions: Y
begin
    if Y(G) is True:
        yield G
    else:
        for child in C.get_children(G):
            for descendant in get_descendants(child):
                yield descendant
end
```

Table 8.1: Mu table for a $(56, 28, 12)$ Type I default generator matrix

$w(a)$ \ $w(b)$	27	25	23	21	19
27	-	-	-	-	$\{18\}$
25	-	-	-	$\{18\}$	$\{16\}$
23	-	-	$\{18\}$	$\{16\}$	$\{14, 16\}$
21	-	$\{18\}$	$\{16\}$	$\{14, 16\}$	$\{12, 14\}$
19	$\{18\}$	$\{16\}$	$\{14, 16\}$	$\{12, 14\}$	$\{10, 12, 14\}$
17	$\{16\}$	$\{14, 16\}$	$\{12, 14\}$	$\{10, 12, 14\}$	$\{8, 10, 12\}$
15	$\{14\}$	$\{12, 14\}$	$\{10, 12, 14\}$	$\{8, 10, 12\}$	$\{6, 8, 10, 12\}$
13	$\{12\}$	$\{10, 12\}$	$\{8, 10, 12\}$	$\{6, 8, 10, 12\}$	$\{4, 6, 8, 10\}$
11	$\{10\}$	$\{8, 10\}$	$\{6, 8, 10\}$	$\{4, 6, 8, 10\}$	$\{2, 4, 6, 8, 10\}$

$w(a)$ \ $w(b)$	17	15	13	11
27	$\{16\}$	$\{14\}$	$\{12\}$	$\{10\}$
25	$\{14, 16\}$	$\{12, 14\}$	$\{10, 12\}$	$\{8, 10\}$
23	$\{12, 14\}$	$\{10, 12, 14\}$	$\{8, 10, 12\}$	$\{6, 8, 10\}$
21	$\{10, 12, 14\}$	$\{8, 10, 12\}$	$\{6, 8, 10, 12\}$	$\{4, 6, 8, 10\}$
19	$\{8, 10, 12\}$	$\{6, 8, 10, 12\}$	$\{4, 6, 8, 10\}$	$\{2, 4, 6, 8, 10\}$
17	$\{6, 8, 10, 12\}$	$\{4, 6, 8, 10\}$	$\{2, 4, 6, 8, 10\}$	$\{0, 2, 4, 6, 8\}$
15	$\{4, 6, 8, 10\}$	$\{2, 4, 6, 8, 10\}$	$\{0, 2, 4, 6, 8\}$	$\{0, 2, 4, 6, 8\}$
13	$\{2, 4, 6, 8, 10\}$	$\{0, 2, 4, 6, 8\}$	$\{0, 2, 4, 6, 8\}$	$\{0, 2, 4, 6\}$
11	$\{0, 2, 4, 6, 8\}$	$\{0, 2, 4, 6, 8\}$	$\{0, 2, 4, 6\}$	$\{0, 2, 4, 6\}$

Listing 8.3: Simplified pseudocode of general function to find solutions of v_i

```
Function solve(Li, wb, mu_conds):
Input:
    list of sets: Li
    integer: wb
    mu condition set object: mu_conds
begin
    ps = get_partial_sums(Li)
    vi = []
    return solve_rec(vi, Li, ps, wb, mu_conds)
end
```

```
Function solve_rec(vi, Li, ps, rem_wb, mu_conds):
Input:
    list of integers: vi
    list of sets: Li
    list of integer: ps
    integer: rem_wb
    mu condition set object: mu_conds
begin
    i = length(vi) + 1
    if i == 1 or check_mu_conditions(mu_conds, vi):
        if i == length(Li) + 1:
            yield vi
        else:
            upper_limit = min(rem_wb, size(Li[i]))
            lower_limit = max(rem_wb - ps[i+1], 0)
            for v in upper_limit downto lower_limit:
                vi_ = concat(vi, v)
                for s in solve_rec(vi_,Li,ps,rem_wb-vi,mu_conds):
                    yield s
end
```

Listing 8.4: Simplified pseudocode of clique generator method

```
Method get_cliques(H, chi, chi_):
Input:
    graph object: H
    integer: chi
    integer: chi_
Dependencies:
    yield conditions: Y
begin
    cdiff = chi - chi_
    if chi_ == 2:
        for edge in H.edges():
            vertex_a, vertex_b = edge.vertices()
            clq = Clique([vertex_a, vertex_b])
            if length(clq.mut_neighbours()) >= cdiff:
                yield clq
    else:
        for cli in get_cliques(H,chi,chi_-1):
```

```
            for ne in cli.mut_neighbours():
                clq = cli.add(ne)
                if length(clq.mut_neighbours()) >= cdiff:
                    if Y(clq):
                        yield clq
end
```

Table 8.2: Computational results using default configuration

(n,k,d)	Type	codes found	execution time	non-equivalent codes
(16,8,4)	Type I	43 codes (finished)	3.4 sec	1 code
(16,8,4)	Type II	27 codes (finished)	1.2 sec	2 codes
(24,12,8)	Type II	100 codes (continues)	15.7 sec	1 code
(56,26,12)	Type II	100 codes (continues)	515 sec	100 codes
(63,57,3)	None	1 code (continues)	7.4 sec	1 code

Table 8.3: Computational results using only depth-first search

(n,k,d)	Type	codes found	execution time	non-equivalent codes
(16,8,4)	Type I	59 codes (finished)	4.5 sec	1 code
(16,8,4)	Type II	37 codes (finished)	2.0 sec	2 codes
(24,12,8)	Type II	100 codes (continues)	17.4 sec	1 code
(56,26,12)	Type II	100 codes (continues)	70 sec	55 codes
(63,57,3)	None	1 code (continues)	5.2 sec	1 code

Listing 8.5: Simplified pseudocode of hill climbing generator method

```
Method get_batches(G):
Input:
    generator matrix: G
Dependencies:
    solutions module: S
    heuristic function: h
    integer: batch_size_limit
begin
    batch = []
    if batch_size_limit is None:
        for candidate in S.get_candidates(G):
            G_ = augment(G, candidate)
            batch.add((h(G_), candidate))
        yield batch
    else:
        generator = S.get_candidates(G)
        while True:
            if length(batch) == batch_size_limit:
                yield batch
                batch = []
            else:
                candidate = next(generator)
                  if candidate is None:
                          yield batch
                    break
                G_ = augment(G, candidate)
                batch.add((h(G_), candidate))
end

Method get_candidates(G):
Input:
    generator matrix: G
begin
    for batch in get_batches(G):
        for hc, c in sorted(batch):
            yield c
end
```

Listing 8.6: Simplified pseudocode of best-first search generator method

```
Method get_bfs_descendants(G):
Input:
    generator matrix: G
Dependencies:
    children module: C
    yield conditions: Y
    heuristic function: h
    integer: batch_size_limit
    integer: limit
begin
    if Y(G):
        yield G
        return
    open_nodes = [(h(G), G)]
    while length(open_nodes) != 0:
        current_h, current_matrix = open_nodes[1]
        child_batch = []
        child_gen = C.get_children(current_matrix)
        if batch_size_limit is None:
            for child in child_gen:
                if Y(child):
                    yield child
                    continue
                child_batch.add((h(child), child))
            remove(open_nodes, current_node)
        else:
            for index in range(batch_size_limit):
                child = next(child_gen)
                if child is None:
                    remove(open_nodes, current_node)
                    break
                if Y(child):
                    yield child
                    continue
                child_batch.add((h(child), child))
        add_and_sort(open_nodes, child_batch)
        if limit is not None and length(open_nodes) > limit:
```

```
            cut(open_nodes, limit)
end
```

Table 8.4: Computational results using hill climbing search

(n,k,d)	Type	codes found	heuristic & batch size	execution time	non-equivalent codes
(32,16,8)	II	100 codes (continues)	h_2 40	495 sec	2 codes
(32,16,8)	II	100 codes (continues)	h_4 20	41 sec	3 codes
(32,16,8)	II	69 codes (continues)	h_4 16	23 sec	3 codes
(40,20,8)	I	4 codes (continues)	h_5 100	456 sec	4 codes
(56,27,12)	II	100 codes (continues)	h_4 100	331 sec	100 codes

Table 8.5: Computational results using best-first search

(n,k,d)	Type	codes found	heuristic & batch size	execution time	non-equivalent codes
(16,8,4)	I	59 codes (finished)	h_4 100	4.2 sec	1 code
(16,8,4)	II	37 codes (finished)	h_4 100	1.5 sec	2 codes
(24,11,6)	I	100 codes (continues)	h_7 100	9.5 sec	2 codes
(24,11,6)	I	0 codes (continues)	h_5 100	300 sec	0 codes
(24,11,6)	I	100 codes (continues)	h_3 100	235 sec	1 code
(24,11,6)	I	100 codes (continues)	h_5, h_3 100	21 sec	6 codes

1.5 Chapter 10

Table 10.1: Computational results using the Weight Limit configuration

(n,k,d)	Type	codes found	weight limits	execution time	non-equivalent codes
(16,8,4)	Type I	37 codes (finished)	min=4 max=6	2.7 sec	1 code
(16,8,4)	Type II	25 codes (finished)	min=4 max=4	1 sec	1 code
(56,20,16)	Type II	6 codes (continues)	min=16 max=16	2409 sec	6 codes
(56,26,12)	Type II	100 codes (continues)	min=12 max=12	244 sec	6 codes
(56,26,12)	Type II	100 codes (continues)	min=12 max=16	52 sec	10 codes

Table 10.2: Computational results using the Weight List configuration

(n,k,d)	Type	codes found	weight list	execution time	non-equivalent codes
(16,8,4)	Type I	6 codes (finished)	wl_1	0.7 sec	1 code
(16,8,4)	Type II	2 codes (finished)	wl_1	0.2 sec	1 code
(28,14,6)	Type I	100 codes (continues)	wl_2	108 sec	1 code
(56,25,12)	Type I	100 codes (continues)	wl_3	102 sec	31 codes
(56,26,12)	Type II	100 codes (continues)	wl_4	219 sec	3 codes

Table 10.3: Computational results using the Perm. Equiv. configuration

(n,k,d)	Type	codes found	execution time
(16,8,4)	Type I	1 code (finished)	3 sec
(16,8,4)	Type II	2 codes (finished)	1.4 sec
(24,12,6)	Type I	1 code (finished)	559 sec
(24,12,8)	Type II	1 code (finished)	8.9 sec
(40,18,8)	Type II	100 codes (continues)	387 sec

Table 10.5: Computation results using the paperclip algorithm

(n,k,d)	Type	codes found	execution time	non-equivalent codes
(16,8,4)	Type I	1 code (finished)	2.8 sec	1 code
(16,8,4)	Type II	2 codes (finished)	1.3 sec	2 codes
(22,11,6)	Type I	1 code (finished)	9.8 sec	1 code
(24,12,8)	Type I	0 (finished)	2.3 sec	0 (this code does not exist)
(10,5,4)	None	5 codes (finished)	0.6 sec	4 codes

0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1
0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1
0 0 0 0 0 0 0 1 1 0 1 0 1 0 1 0 1
0 0 0 0 1 1 0 0 0 1 0 1 0 1 0 1
0 0 1 1 0 0 0 0 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 0 0 0 0 0 1 1
1 0 0 1 0 1 0 1 0 1 0 1 0 1 1 0

Table 10.6: (16,8,4) Type I superminimal generator

0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1	0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1
0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1	0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1
0 0 0 0 0 0 0 0 1 0 0 1 0 1 1 0	0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1
0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0	0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1
0 0 1 1 0 0 1 1 0 0 0 0 0 0 0 0	0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1
0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0	1 0 0 1 0 1 0 1 0 1 0 1 0 1 1 0

Table 10.7: (16,8,4) Type II superminimal generators

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1 0 0 1 1 0 0
0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1 1 1 1 1
0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 1 0 1 0 1 0 0
0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 1 0 1 1 1 0 1
0 0 0 0 1 0 0 1 0 0 0 1 0 1 1 1 0 0 0 1 1 0
0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 1 0 0 1 0 0 1
0 0 1 0 0 0 0 1 0 1 0 0 0 1 1 1 0 1 0 0 0 1
0 1 0 0 0 0 0 1 0 1 0 1 0 0 1 1 0 1 0 1 1 1
1 0 0 0 0 0 0 1 0 1 0 1 0 1 1 0 0 0 0 0 0 0

Table 10.8: (22,11,6) Type I superminimal generator

0 0 0 0 0 0 1 1 1 1	0 0 0 0 0 0 1 1 1 1
0 0 0 0 1 1 0 0 1 1	0 0 0 0 1 1 0 0 1 1
0 0 0 1 0 1 0 1 0 1	0 0 0 1 0 1 0 1 0 1
0 0 1 0 0 1 0 1 1 0	0 0 1 0 0 1 0 1 1 0
1 1 0 0 0 0 0 0 1 1	1 1 0 0 0 1 0 1 1 1

Table 10.9: (10,5,4) permutation equivalent generators

(n,k), $d \geq 2$	runtime	number of codes found
(16,8)	∼0.6s	7
(22,11)	∼1.9s	25
(24,12)	∼4.1s	55

Table 10.10: Computation results using GenSelfDual algorithm

Summary

Chapter 1

English summary

The thesis focuses on two cryptographic primitives: random number generators and linear codes.

Part I of the thesis is based on an algorithm developed by Tamás Herendi in [4]. The algorithm can be used to construct linear recurrence sequences (LRS) with uniform distribution modulo powers of 2, with arbitrarily large period lengths, and arbitrarily large elements. The theoretical design of the presented results was developed during joint research between Tamás Herendi and the author. The application development, along with the implementation and analysis of the tests carried out are the results of the author. The relevant publications by the author for this part of the thesis are [5], [6] and [7].

Part II of the thesis is concerned with the problem of constructing linear codes with given, close to arbitrary parameters. The author has developed a software package named Torch, that provides an extensible, reconfigurable solution that supports a wide variety of search algorithms, conditions and other options that allows researchers to experiment and incorporate new research results into linear code searches. The software can be used for classification problems, finding existing linear codes, or searching for currently unknown codes. The theoretical design of the presented results was developed during joint research between Carolin Hannusch and the author. The software development, implementation and analysis are the results of the author. The relevant publications by the author for this part of the thesis are [11] and [12].

Chapter 2 introduces the basic concepts used throughout this part of the thesis. The first important concept in this chapter is finite fields, as most results throughout both parts of the thesis require computation over $GF(2)$. Polyno-

mials over finite fields in general, and irreducible polynomials in particular, play an important role in the algorithm that Part I is based on. Certain operations introduced here, such as calculating the greatest common divisor (GCD), and the order of polynomials, are also key components of the algorithm.

Chapter 3 describes the algorithm in detail. Two approaches are discussed, one based on matrix exponentiation, and one based on polynomial modulo. These two approaches are named after their most computationally expensive steps. The second approach is more computationally efficient, and will be used for most of the rest of Part I of the thesis.

As input, the algorithm requires an irreducible polynomial of high degree and order over $GF(2)$. The output of the algorithm is a linear recurrence sequence with uniform distribution. The period length of the sequence depends on the order of polynomial used to create it.

The chapter also discusses some methods of irreducibility testing, and two ways of creating such polynomials. The first approach to generating suitable irreducible polynomials, developed by the author and Tamás Herendi, is based on making sure that a randomly generated polynomial does not have any small factors. The second method uses a known tranformation named *Q-transform* to create infinite families of irreducible polynomials. The chapter concludes with statistical test results comparing linear recurrence sequences based on polynomials created by these two methods. While the first method shows excellent test results and is more general in approach, the second method shows results very close to the first one, and is more efficient to compute.

Chapter 4 presents multiple implementations, created by the author, of the previously identified computationally expensive steps of the algorithm. The software implementations were created using the NTL and FLINT libraries. Hardware implementations designed by the author for the SciEngines RIVYERA FPGA platform are also presented. The implemented operations are matrix multiplication over $GF(4)$, polynomial GCD and polynomial modulo over $GF(2)$. NTL and FLINT provide multiple different ways to represent matrices and polynomials over finite fields. Implementations were created utilizing all available options. Performance tests were carried out to compare the efficiency of the different software and hardware solutions for these operations. The tests determined the NTL software library to be the most efficient solution to be used in the implementation of the linear recurrence sequence generating algorithm.

Chapter 5 details tests carried out to measure the statistical properties of the pseudorandom number sequences created using the algorithm and tools determined in the previous chapters. Several attributes can affect the statistical properties of the sequences created. The ones of interest for the statistical tests

are the following:

- The degree of the irreducible polynomial used to create the LRS.
- The irreducible polynomial being *dense*, that is, having a large number of non-zero coefficients, or *sparse*.
- The initial values of the sequence, before it begins generating new pseudorandom values.
- Having a function added to the LRS that transforms each pseudorandom value generated. Seven different transformations were implemented and tested.

All tests described in this chapter, as well as the statistical tests in chapter 3, use the NIST statistical test suite.

The statistical properties and the practical application of the sequences are also compared to other pseudorandom sequences that were defined in [29], [30] and [31]. These sequences have provably good measures of pseudorandomness, as defined in [32].

The test results and the practical conclusions drawn from them are included in the chapter.

Part II of the thesis begins with chapter 6. This chapter introduces the concepts relevant to this part of thesis. These concepts include binary linear codes, the (n, k, d) parameters of codes, generator matrices, systematic codes and standard generator matrices, permutation equivalence between codes, and self-dual codes. Type I and Type II self-dual codes are also introduced. The chapter then describes the problem of finding linear codes with given parameters. No known universal algorithm exists that can construct any possible linear code with reasonable efficiency, nor are there any theoretical results that can answer if a linear code exists with given arbitrary parameters. Basic methods for constructing new codes from existing ones are described. The results detailed in the following chapters of the thesis are based on algorithms that create new codes via *augmenting* existing ones.

Chapter 7 describes the practical usage, basic architecture and dependencies of the Torch package created by the author. The problem described in the previous chapter gives rise to a practical need for a framework that is adaptable to almost arbitrary conditions, has an architecture that is easily extensible, can incorporate new research results, and enables rapid development and experimentation. The chapter gives some implementation details about the package, including its software dependencies, and gives practical examples of using Torch

for common linear code search problems. These problems can include classification, generating known codes, or searching for currently unknown codes. The chapter gives a basic overview of the software architecture of Torch. In order to accomplish the required flexibility and extensibility, the framework uses a custom dependency injection system created by the author to assemble the objects needed for a given linear code search. The mechanism through which the modules of the package are assembled into configurations capable of carrying out searches is described.

Chapter 8 presents the basic search algorithms available in the package, along with the implementation details necessary to make the solutions viable in practice. The algorithms included in Torch are a basic depth-first search, a search based on finding cliques in a graph, and a hill-climbing and best-first heuristic searches. These algorithms are defined as configurations of modules that each carry out a well-defined subtask in the search. The algorithms can be modified by switching out the implementations of modules as needed. The chapter describes the default modules provided. Of note here is the Mu table. Let $\mu(a, b)$ be the number of coordinates i such that $a_i = b_i = 1$ for two binary vectors of equal length a and b. During a search, the parameters of the code being searched for determines the possible μ-values allowed between codewords. This information can be precomputed in a table, and used to create sets of conditions that help determine which codewords are suitable to augment a code. The algorithms used to generate the candidate codewords for augmentation are detailed. Besides the depth-first search, an algorithm is introduced that generates linear codes by finding cliques in a graph that represent the candidate codewords. The default search defined in Torch uses both of these algorithms, dynamically switching between them during a search. Heuristic searches are also supported, using hill-climbing and best-first search algorithms. Torch provides a number of heauritic funtions to use in these algorithms. Users can also define custom heuristic functions to enable experimentation. The chapter also gives practical results of using the presented algorithms.

Chapter 9 details elements of the Torch framework that can be used to fine-tune and extend the behaviour of the searches. These elements include a large number of option variables, and condition objects that can be added to the previously described algorithms. To make modules flexible enough to be reusable, most of them are defined to accept a number of different options. Some options can be used to set properties such as save location, or turning logging on or off. Others can affect the execution of the search, such as setting the form of the generator matrices created or changing the order in which candidate codewords are created. Another element of the Torch framework that are commonly used are

condition objects. Torch provides a base class to extend when creating classes that can check if a linear code satisfies a given condition. These classes provide features that make logging, bypassing, and combining conditions more convenient. Condition objects fulfill a variety of roles in the search algorithm, and are the most common way of incorporating new research results into a search.

Chapter 10 presents a non-exhaustive selection of configurations that are currently available in Torch. As previously mentioned, configurations describe a collection of modules and parameters that are commonly used together to define or modify a search algorithm to achieve a specific search goal. Multiple configurations can be used together in one search.

Two configuration are provided that place restrictions on the weights used when constructing generator matrices: a *Weight Limit* configuration that defines an allowed interval for the weights, and a *Weight List* configuration that takes a list of values that can define complex conditions for the weights.

The *Perm. Equiv.* configuration was created with the purpose of filtering out all permutation equivalent nodes from the search trees, using the basic technique of recorded objects, keeping a global record of non-equivalent objects encountered during the search. In practice, space usage can be a significant constraint when using this configuration.

The *Paperclip* configuration was created to fulfill mutiple criteria: it should be suitable for both classification and targeted search tasks, the search results should very rarely contain permutation equivalent codes, and the configuration should avoid the drawbacks of the *Perm. Equiv.* configuration. The goal is to construct the search tree in such a way that it contains only one code from each permutation equivalent set. *Superminimal generator matrices* are introduced, which are uniquely defined for each set of equivalent codes. Four conditions are given that are necessary, but not sufficient to determine if a generator matrix is superminimal, without the need to record or compare matrices.

The chapter also gives practical results of using the presented configurations.

Chapter 2

Magyar nyelvű összefoglaló

A disszertáció két kriptográfiai primitívhez kapcsolódó eredményeket mutat be. Ez a két primitív a véletlenszám generátorok és a lineáris kódok.

A disszertáció első része egy algoritmusra épül, amely Herendi Tamás eredménye [4]. Az algoritmussal olyan lineárisan rekurzív véletlenszám sorozatok generálhatók, amelyek egyenletes eloszlásúak, modulo 2 egy tetszőleges hatványa. A sorozat periódushossza és a sorozat elemeinek nagysága tetszőleges nagy lehet. A disszertációban bemutatott eredmények elméleti háttere Herendi Tamás és a szerző közös kutatásának eredménye. A gyakorlati alkalmazások fejlesztése, implementációja, valamint a tesztek implementációja és elemzése a szerző eredménye. A szerző ezen témakörhöz releváns publikációi [5], [6] és [7].

A disszertáció második része a lineáris kódok keresésének problémájára épül. A cél adott, gyakorlatilag tetszőleges feltételeknek megfelelő linearis kódok generálása. A szerző által fejlesztett, Torch nevű szoftver egy kiterjeszthető, újrakonfigurálható megoldást nyújt a problémára. A Torch csomag több különböző keresési algoritmust, keresési feltételeket és egyéb opciókat támogat. Ezen keretrendszer elősegíti a kísérletezést és az új kutatási eredmények beépítését lineáris kódok keresése során. A szoftver felhasználható klasszifikációs problémákhoz, már ismert lineáris kódok generálásához, vagy jelenleg ismeretlen kódok kereséséhez. A disszertációban bemutatott eredmények elméleti háttere Carolin Hannusch és a szerző közös kutatásának eredménye. A szoftver fej-lesztése, implementációja és elemzése a szerző eredménye. A szerző ezen téma-körhöz releváns publikációi [11] és [12].

A második fejezet tartalmazza a disszertáció első részében használt fogalmakat. Kiemelt fontosságú a véges test fogalma, mivel a disszertációban találha-

tó eredmények túlnyomó többsége $GF(2)$ feletti számítást igényel. A véletlenszám sorozatokat előállító algoritmusban fontos szerepet játszanak a véges testek feletti irreducibilis polinomok. Szintén megtalálhatóak a fejezetben az algoritmusban használt főbb műveletek definíciói, mint a polinomok legnagyobb közös osztója, és a polinomok rendje.

A harmadik fejezet részletesen bemutatja a véletlenszám sorozatokat előállító algoritmust. A fejezet két megközelítést ír le. Az első megközelítés legtöbb számítást igénylő művelete a mátrixhatványozás. A második megközelítésben ez a művelet a polinomális modulo. A második megközelítés jelentősebb hatékonyabb az elsőnél, a disszertáció további része ezt a megközelítést használja.

Az algoritmus inputja egy $GF(2)$ feletti, magas fokszámú és rendű irreducibilis polinom. Az algoritmus kimenete egy egyenletes eloszlású lineárisan rekurzív sorozat. A sorozat periódushossza annak a polinomnak a rendjétől függ, amely az algoritmus bemenetéül szolgált.

A fejezet ezt követően bemutat néhány irreducibilitás tesztelő módszert, valamint két eljárást az algoritmusban felhasználható polinomok előállítására. Az első eljárás a szerző és Herendi Tamás fejlesztése, amelynek célja, hogy a véletlenszerűen generált polinomoknak garantáltan ne legyen kis fokszámú osztója. A második eljárás egy ismert, *Q-transform* nevű átalakítást használ irreducibilis polinomok végtelen családjainak előállításához. A fejezet végén statisztikai tesztek eredményei találhatók, amelyek a két eljárással előállított polinomokból származó véletlenszám sorozatokat tesztelnek. Az első eljárás kiváló eredményeket mutatott fel. A második eljárás eredményei közel azonosak, és a gyakorlatban jelentősen könnyebben számítható.

A negyedik fejezet olyan a szerző által készített implementációkat mutat be, amelyek az algoritmus korábban említett nagy számításigényű műveleteit valósítják meg. A szoftver implementációk az NTL és FLINT könyvtárakat használják. A műveletekhez hardver implementációk is készültek a SciEngines RIVYERA FPGA platformon, amelyeket szintén a szerző tervezett és valósított meg. Az implementált műveletek a $GF(4)$ feletti mátrix hatványozás, $GF(2)$ feletti polinomok legnagyobb közös osztója és a polinomiális modulo. Az NTL és FLINT könyvtárak több lehetőséget biztosítanak véges testek feletti mátrixok és polinomok reprezentálására. Az összes lehetőség teszteléséhez készült önálló implementáció. A fejezet tartalmazza a különböző hardver és szoftver megvalósítások teljesítményét összehasonlító tesztek eredményeit is. A teszteredmények szerint az NTL könyvtár a leghatékonyabb megoldás az algoritmus műveleteinek megvalósításához.

Az ötödik fejezet olyan teszteket ír le, amelyek az algoritmus által előállított pszeudo-véletlenszám sorozatok statisztikai jellemzőit vizsgálják. Ezeket a jel-

lemzőket több tulajdonság befolyásolhatja. A tesztek számára releváns tulajdonságok:

- A sorozat alapjául szolgáló irreduciblis polinom fokszáma.
- Az irreducibilis polinom *sűrű* (a nem nulla együtthatók száma nagy), vagy *ritka*.
- A sorozat kezdőértékei.
- A sorozat által generált elemek transzformációja. Hét különböző transzformáló függvény lett implementálva és tesztelve.

Az ötödik és harmadik fejezetben található statisztikai tesztek a NIST tesztkészlettel lettek megvalósítva.

A sorozatok statisztikai tulajdonságai és gyakorlati felhasználhatóságuk összehasonlításra kerül más, ismert sorozatokkal [29], [30], [31]. Ezen sorozatok bizonyíthatóan jó pszeudo-véletlen tulajdonságokkal rendelkeznek [32].

A tesztek eredményei és a belőlük levont gyakorlati következtetések szintén megtalálhatóak a fejezetben.

A disszertáció második része a hatodik fejezettel kezdődik. Ez a fejezet tartalmazza a disszertáció ezen részében használt fogalmakat. Ezek közül kiemelt fontosságúak a bináris lineáris kódok, (n, k, d) paraméterű kódok, generátor mátrixok, szisztematikus kódok és standard generátor mátrixok, kódok közötti permutáció ekvivalencia, és önduális kódok fogalmai. Az Egyes és Kettes Típusú önduális kódok fogalma is bemutatásra kerül. A definíciókat követően a fejezet ismerteti a lineáris kódok keresésének problémáját. Nem ismert olyan univerzális algoritmus, ami gyakorlatban használható hatékonysággal képes tetszőleges lineáris kódot konstruálni. Szintén nem ismert olyan elméleti kutatási eredmény, amely tetszőleges adott paraméterekhez meg tudja válaszolni, hogy létezik-e ilyen paraméterekkel rendelkező lineáris kód. Bemutatásra kerül néhány alapvető módszer, amelyekkel létező lineáris kódokból új kódok állíthatók elő. A disszertáció következő fejezeteiben ismertetett eredmények *augmentáción* alapuló algoritmusokra épülnek.

A hetedik fejezet bemutatja a szerző által készített Torch csomag gyakorlati felhasználását, alapvető architektúráját és függőségeit. Az előző fejezetben ismertetett probléma egy olyan keretrendszert igényel, amely képes alkalmazkodni gyakorlatilag tetszőleges keresési feltételekhez, egyszerűen bővíthető, képes új kutatási eredmények beépítésére, és ezáltal lehetővé teszi a gyors fejlesztést és kísérletezést. A fejezet leírja azokat a külső szoftvereket, amelyektől a csomag függ, és példákon keresztül bemutatja a csomag gyakorlati felhasználását. A

Torch felhasználható klasszifikációs problémákhoz, ismert lineáris kódok generálásához, és jelenleg ismeretlen lineáris kódok kereséséhez. Arról, hogy a keretrendszer kellően flexibilis és kiterjeszthető legyen, a szerző által készített saját függőség befecskendező rendszer gondoskodik. A függőség befecskendező feladata példányosítani azokat az objektumokat, amelyek szükségesek egy adott keresési feladat elvégzéséhez. A fejezet részletesen bemutatja a mechanizmust, amelynek segítségével a csomagban definiált komponensek a keresést végző konfigurációkká állnak össze.

A nyolcadik fejezet leírja a csomagban implementált keresőalgoritmusokat, és alapvető implementációs részleteiket. A Torch csomagban megtalálható mélységi kereső, klikk-alapú kereső, hegymászó és best-first algoritmus. Ezek az algoritmusok alacsony szintű modulok konfigurációiként vannak definiálva. Minden modul egy jól definiált részfeladatot hajt végre a keresés során. Az algoritmusok viselkedése a modulok szükség szerinti lecserélésével módosítható. A fejezet bemutatja az alapértelmezetten használt modulokat. Egy kiemelt modul ezek közül a Mű-tábla. Legyen a és b két azonos hosszúságú bináris vektor, és legyen $\mu(a, b)$ azon i koordináták száma, ahol $a_i = b_i = 1$. Egy keresés során a keresett kód paraméterei meghatározzák a kódszavak között lehetséges μ-értékeket. Ezek az értékek egy táblázatban előre kiszámíthatók. Ezen táblázat segítségével feltétel halmazok definiálhatók, amelyek segítenek meghatározni azokat a kódszó jelölteket, amelyekkel megpróbálhatjuk augmentálni a kódokat. A kódszó jelölteket generáló algoritmusok is ismertetésre kerülnek. A mélységi keresőn kívül bevezetésre kerül egy algoritmus, amely a keresést gráfokban található klikkek keresésére vezeti vissza. A gráf csúcsai a kódszó jelölteket reprezentálják. A Torch alapértelmezett keresője mindkét algoritmust fehasználja, futás közben dinamikusan vált közöttük. A csomag támogat heurisztikus keresést is, hegymászó és best-first keresés formájában. A csomagban megtalálhatóak heurisztikus függvények, amelyek felhasználhatóak ezekhez az algoritmusokhoz, valamint a felhasználónak lehetősége van saját heurisztikus függvény definiálására. A fejezetben megtalálható még a bemutatott algoritmusok használatának néhány mérési eredménye.

A kilencedik fejezet a Torch keretrendszer azon elemeit mutatja be, amelyek segítségével finomhangolhatók a modulok működése. Ezek az elemek az opcionális változók, és a feltétel objektumok. A flexibilitás és újrafelhasználhatóság növelése céljából a legtöbb modul több opció fogadására is képes. Egyes opciók olyan tulajdonságokat állítanak be, mint a mentési hely, vagy a loggolás ki- vagy bekapcsolása. Más opciók a keresés menetére vannak hatással, mint az előállítani kívánt generátor mátrixok alakja, vagy a kódszó jelöltek generálási sorrendje. Szintén gyakran használt eszköz a csomagban a feltétel objektumok.

A Torch biztosít egy ősosztályt, amelynek kiterjesztésével olyan objektumok hozhatók létre, amelyek célja annak ellenőrzése, hogy egy lineáris kód megfelele egy adott feltételnek. Ez lehetővé teszi a feltételek loggolását, szükség szerinti kikerülését, vagy kombinálását. A feltétel objektumok sokféle szerepet töltenek be a keresőalgoritmusokban. Ezen objektumok használata a leggyakoribb módja az új kutatási eredmények keresésbe való beépítésének.

A tizedik fejezet a Torch csomagban elérhető konfigurációk közül mutat be néhány kiemelt fontosságút. Ahogy korábban is említve volt, a konfigurációk modulok és paraméterek olyan összességét jelentik, amelyek együttesen egy tipikus keresési feladatot képesek végrehajtani. Egy keresés során egyszerre több konfiguráció is használható.

A csomagban két konfiguráció található, amelyek súly korlátokat határoznak meg az előállított generátor mátrixok soraira. A *Weight Limit* egy egyszerű intervallummal szabja meg a megengedett súlyokat, a *Weight List* pedig komplex súlyfeltételek meghatározására képes, amelyet listával lehet megadni.

A *Perm. Equiv.* konfiguráció célja a permutáció ekvivalens kódok kiküszöbölése a keresési fából. Ezt a *feljegyzett objektumok* módszerével teszi, amely egyszerűen számon tartja a futás közben már megtalált nem ekvivalens objektumokat. A gyakorlatban a konfiguráció tárhasználata jelentős megszorítás lehet.

A *Paperclip* konfiguráció fejlesztését több cél vezérelte: legyen alkalmas klasszifikációs feladatokra és célzott keresésre egyaránt, a keresési eredmények között legyenek ritkák a permutáció ekvivalens kódok, és kerülje el a *Perm. Equiv.* konfiguráció hátrányait. A cél olyan keresőfa építése, amelyben minden permutáció ekvivalens halmazból csak egy kód található meg. Bevezetésre kerül a *szuperminimális generátor mátrix* fogalma. Ez a mátrix egyértelműen meghatározott minden permutáció ekvivalens kódhalmazhoz. Négy szükséges, de nem elégséges feltétel kerül bemutatásra, amelyek egy adott generátormátrixról segítenek eldönteni, hogy szuperminimális-e, anélkül, hogy a mátrixokat tárolni vagy összehasonlítani kellene.

A fejezetben megtalálható még a bemutatott konfigurációk használatának néhány mérési eredménye.