SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

A Concrete Evaluation of Information Set Decoding Techniques

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Emanuele Lunardi**

Student ID: 945330
Advisor: Prof. Alessandro Barenghi
Academic Year: 2020-21

Abstract

Nowadays, the most used public-key cryptosystems are based on the hardness of factoring very large numbers or on the intractability of the discrete logarithm problem. We already know that in the future, when it will be possible to build large quantum computers, these problems will be solvable easily thanks to the Shor's algorithm.
Therefore, there is a need to study alternative cryptosystems based on different hard mathematical problems that will be resistant in the era of the quantum computers. One of this alternative is code-based cryptography on which the McEliece and Niederreiter cryptosystems are based on. The functioning of these cryptosystems is founded on the hardness of the syndrome decoding problem, or equivalently, the decoding of a random linear code.
In this work we are going to analyze and implement the algorithms with the best complexities in the current state of the art that are called Information Set Decoding algorithms: they try to break the code-based cryptosystems solving the syndrome decoding problem. After analyzing subroutines that will be used in the implementation of the ISD algorithms, evaluating different procedures to understand the most efficient one, we are going to study how the Information Set Decoding algorithms work with their complexities and then, we will present the results obtained by a concrete evaluation of them to understand their behaviours in practice.

Keywords: information set decoding, post-quantum cryptography, McEliece cryptosystem, code-based cryptosystems, asymmetric cryptosystems

Abstract in lingua italiana

Attualmente, i crittosistemi a chiave pubblica maggiormente utilizzati, sono basati sulla difficoltà di fattorizzare numeri molto grandi oppure sull'intrattabilità del problema del logaritmo discreto. Sappiamo già che nel futuro, quando saranno disponibili computer quantistici sufficientemente potenti, sarà possibile risolvere questi problemi facilmente grazie al noto algoritmo di Shor.
Dunque, c'è bisogno di studiare crittosistemi alternativi, basati su diversi problemi matematici difficili, che saranno resistenti nell'era dei computer quantistici. Una di queste alternative è la crittografia basata su codici lineari che è alla base dei crittosistemi di McEliece e di Niederreiter. Il funzionamento di questi crittosistemi è fondato sulla difficoltà di risolvere il problema di decodifica di una sindrome, o equivalentemente, la decodifica casuale di un codice lineare.
In questo lavoro andremo ad analizzare e ad implementare gli algoritmi con la miglior complessità attualmente conosciuti nello stato dell'arte chiamati algoritmi di Information Set Decoding: essi cercano di rompere i crittosistemi basati su codici lineari risolvendo il problema di decodifica di una sindrome. Dopo aver analizzato diversi sottoprogrammi che saranno utili nell'implementazione degli algoritmi ISD, valutando diverse procedure per capire la più efficiente da utilizzare, andremo a studiare come funzionano i vari algoritmi Information Set Decoding con le relative complessità, e infine, saranno presentati i risultati ottenuti da una concreta valutazione di questi algoritmi per capire il loro comportamento nella pratica.

Parole chiave: information set decoding, crittografia post-quantum , crittosistema di McEliece, crittosistemi basati su codici lineari, crittosistemi asimmetrici

Contents

Abstract i

Abstract in lingua italiana iii

Contents v

Introduction 1

1 Preliminaries 5

1.1 Coding theory basics . . . 5

1.2 Code-based cryptosystems . . . 8

1.2.1 McEliece cryptosystem . . . 8

1.2.2 Niederreiter cryptosystem . . . 9

2 Useful algorithms used in the ISD 11

2.1 Reduced Row Echelon Form . . . 11

2.1.1 Standard RREF . . . 12

2.1.2 RREF with reusing existing pivots optimization . . . 14

2.1.3 Partial Reduced Row Echelon Form . . . 16

2.1.4 Optimized Partial RREF . . . 17

2.1.5 Method of four russians for inversion . . . 22

2.2 Binary Search variants . . . 25

2.2.1 Binary Search . . . 25

2.2.2 Boundless Binary Search . . . 26

2.2.3 Doubletapped Binary Search . . . 27

2.2.4 Monobound Binary Search . . . 27

2.2.5 Tripletapped Binary Search . . . 29

2.2.6 Monobound Quaternary Search . . . 29

2.2.7 Monobound Interpolated Binary Search . . . 31

2.2.8 Adaptive Binary Search . . . 32

2.2.9 Boundless Binary Range Search 32
2.3 NextComb and NextColSum algorithms 34
2.3.1 NextComb 34
2.3.2 NextColSum 36
2.4 Sorting algorithms 39
2.4.1 Quicksort 39
2.4.2 Djbsort 40

3 Information Set Decoding algorithms 43
3.1 Basic of Information Set Decoding algorithm 43
3.1.1 Basic structure of an ISD algorithm 44
3.1.2 Complexity analysis of the basic structure of an ISD algorithm . . . 47
3.2 Analysis of Information Set Decoding algorithms 49
3.2.1 Prange 49
3.2.2 Lee-Brickell 51
3.2.3 Leon 54
3.2.4 Stern 57
3.2.5 Ball-Collision Decoding 62
3.2.6 Finiasz-Sendrier 65
3.2.7 May-Meurer-Thomae 68
3.2.8 Becker-Joux-May-Meurer 74
3.2.9 Both-May 80
3.2.10 Esser-Bellini 85
3.3 Implementation techniques 89
3.3.1 Representation of the bit matrices and vectors 89
3.3.2 Advanced Vector Extensions 2 instructions 91
3.3.3 Hamming Weight computation 91
3.3.4 Multithreading 92
3.3.5 Hashtables 93

4 Experimental Evaluation 95
4.1 RREF analysis 95
4.2 Binary range search variants analysis 97
4.3 Sorting algorithms analysis 99
4.4 Estimators to find the optimal ISD parameters 100
4.5 Information Set Decoding algorithms testing 102
4.5.1 Syndrome tests 103
4.5.2 McEliece Tests 113

5 Conclusions and future developments **123**

Bibliography **125**

A Appendix A **129**

List of Figures **133**

List of Tables **135**

List of Algorithms **137**

List of Symbols **139**

Introduction

In every day of our life we use cryptography for securing our communications over the internet thanks to the public key cryptosystems.
These cryptosystems are called even asymmetric cryposystems since pairs of keys are used: each pair consists of a public key which can be known to everyone and a private key (the secret) known only to the owner.
If Bob wants to communicate secretly with Alice he must send an encrypted message to her using the Alice's public key for encrypting it: now only the owner of the private key can decrypt the message for reading its content and since the private key is the secret hold only by Alice, only her can read the message.
This scenario is possible thanks to the mathematical procedures called one way functions: they are easy to compute having a special parameter (the private key in the previous example) but hard to compute without the knowledge of the secret parameter (very hard to decrpyt the message without the knowledge of the private key).
Nowadays the one way function used in the cryptosystems are the hardness of factoring very large numbers or the intractability of the discrete logarithm problem.
In the future, when it will be possible to build large quantum computers, we already know that many algorithms considered hard will be solved very efficiently. For example, the Shor's algorithm [27] finds the prime factors of an integer in polynomial time with a quantum computer: this means that the RSA cryptosystem based on integer factorization can no longer be used in the post-quantum era and this will be a problem for our communications.
Knowing that, many researches began to find alternatives for the current hardness mathematical problems that are the building blocks of our actual public key algorithms and one of these alternative is code-based cryptography.
The McEliece cryptosystem [21] is the first public key cryptosystem based on coding theory: it was presented in 1978 and it has never gain popularity since the research starts to find post-quantum cryposystems.
It has begun to be interesting to study because it is immune against attacks using Shor's algorithm and so there is a possibility that it can be used in post-quantum cryptography.

The McEliece cryptosystem is based on the hardness of decoding a random linear code which is known to be NP-hard and can be used as a public key scheme: a variant of it exists with the same security level called Niederreiter cryptosystem. Niederreiter cryptosystem is based on the hardness of the syndrome decoding problem, the dual problem of the decoding random linear code, and it can be used as a public key scheme and even as a digital signature scheme.
We need to understand better if the McEliece cryptosystem or its variant, the Niederreiter cryptosystem , can be valid public asymmetric encryption algorithms in the future era of quantum computers.
To do that, it is important to apply cryptanalysis techniques for analyzing these systems and discovering possible flaws. So, cryptanalysis can be used for trying to break the cryptosystems with techniques faster than bruteforce and discover the contents of the encrypted messages even if the private key is unknown.
In the current state of the art the algorithms with the best complexities for decoding a random linear code or for solving the syndrome decoding problem are called Information Set Decoding (ISD) algorithms. They all have exponential running time in the code length n of the form $T(n) = 2^{\tau n}$ where τ is a constant used as a metric for comparing different information set decoding algorithms.
The main goal of this thesis is a concrete evaluation of the complexity of the ISD algorithms used to break code-based cryposystems: all the ISD algorithms known in the current state of the art has been implemented for studying their behaviours and to help the design of the secure parameters for the code-based cryptosystems with the intent of making unfeasible these attacks against these schemes with proper parameters.
In a nutshell, all the ISD implemented solve the syndrome decoding problem for breaking the Niederreiter cryptosystem that as we will see, is equivalent to the decoding random linear code. In the syndrome decoding problem we have a parity-check matrix $H \in \mathbb{F}_q^{(n-k)\times n}$, a syndrome vector $s \in \mathbb{F}_q^{n-k}$ and an integer w called weight. The duty of an information set decoding algorithm is to recover an error vector $e \in \mathbb{F}_q^n$ with Hamming weight $\leq w$ such that $He^T = s$.
The study and the implementation of the algorithms starts from the first ISD designed by Prange [24] up to the variant of Both-May described by Esser and Bellini in [13].
This thesis is organized as follows. The first chapter is dedicated to the background knowledge of coding theory and code-based cryptosystems necessary to understand the information set decoding algorithms. The second chapter contains helper routines studying their optimizations and variants that will be used in many ISDs later, like the reduced row echelon form algorithm (RREF), binary range searches, sorting algorithms and algorithm to produce sequentially all the possible combinations of a range of numbers. The

third chapter explains the working principles and the complexities of all the ISD algorithms that have been implemented: Prange, Lee-Brickell, Leon, Stern, Ball-Collision Decoding, Finiasz-Sendrier, May-Meurer-Thomae, Becker-Joux-May-Meurer, Both-May and Esser-Bellini and some implementation choices. The fourth chapter is dedicated to the experimental evaluation where different procedures described in the second chapter are tested to understand the efficient variant to use inside the ISD algorithms. Then, the testing results relative to the ISD techniques are presented. For each ISD two family of tests are used: the Syndrome Decoding tests and the Mc-Eliece tests. Both of these tests ask to solve the syndrome decoding problem and they can be found on the webpage dedicated https://decodingchallenge.org/ [4] . The last section summarizes the conclusions.

1 | Preliminaries

In this chapter we are going to present the basic of coding theory for understanding the code-based cryptosystems. We will see the process of encoding and decoding with the error correcting capability of a linear code. Then, the McEliece and the Niederreiter cryptosystems, based on the linear codes theory will be described.

1.1. Coding theory basics

One of the application of coding theory is the error detection and correction over unreliabile communication channels. Communication channels introduce noise in the messages sent over them, so a procedure to remove the noise must be taken into account. Claude Shannon in [26] tells us that for any communication channels it is possible to communicate discrete data nearly error free up to a maximum rate (the channel capacity).
For solving the problem of noisy channels an error correcting code can be used. The sender encodes the original message adding redundancy information to it, then, the encoded message called codeword goes through the noisy channel and arrives to the receiver perturbated by a certain error e that depends on the current noise of the channel. At the end, the receiver recovers the original message removing the perturbation error thanks to the decoding phase. The scenario described here can be seen in Figure 1.1:

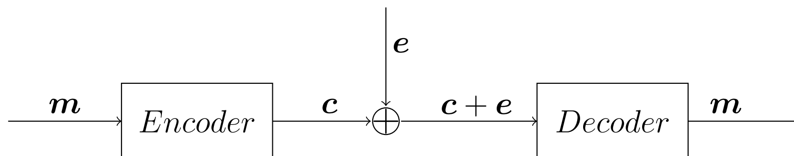


Figure 1.1: Error correcting code over a noisy channel

There are many types of different codes but we will focus our attention on binary linear codes. We are going to explain formally what is a linear code for understanding how the encoding and the decoding phases are carried out.

Definition 1.1.1 (Linear code)**.** A linear code of length n and dimension k is a linear

subspace C with dimension k of the vector space $\mathbb{F}_q^n$, where $\mathbb{F}_q$ is the finite field with q elements. If $q = 2$ the code is called binary code.

The vectors $c \in C$ are called codewords and the size of a binary linear code is the number of codewords in it and it is equal to 2^k.

Definition 1.1.2 (Generator matrix)**.** A generator matrix $G \in \mathbb{F}_q^{k \times n}$ is a matrix whose rows form a basis for a linear code, this means that the matrix G has maximum rank equal to k.

The encoding phase consists in multiplying a vector message $m \in \mathbb{F}_q^k$ with a matrix $G \in \mathbb{F}_q^{k \times n}$ for obtaining a codeword $c \in \mathbb{F}_q^n$. If the matrix G is a generator matrix it has rank k and so there is a one to one correspondence between the message space and the code space. The generator matrix is not unique, a code can have many generator matrices but they all have rank equal to k.

Definition 1.1.3 (Dual code)**.** Given a linear code $[n, k]$ called C we define as the dual code : $C^\perp = \{x \in \mathbb{F}_q^n \quad | \quad x \cdot c = 0, \quad \forall c \in C\}$ where $x \cdot c = \sum_{i=1}^n x_i c_i$.

The dual code $C^\perp$ is a linear code with parameters $[n, n-k]$.

Definition 1.1.4 (Parity-check matrix)**.** A parity-check matrix $H \in \mathbb{F}_q^{(n-k) \times n}$ of a linear code C is a generator matrix of the dual code $C^\perp$. This means that $c \in C$ if and only if $Hc^T = 0$.

From this we can derive that a generator matrix for the dual code is a parity-check matrix for the original code and vice versa. The parity-check matrix plays an important role in the decoding phase and we can derive it from the generator matrix. For example, if we have the generator matrix G in standard form $G = [I_k \quad A]$, where $I_k \in \mathbb{F}_q^{k \times k}$ is the identity matrix of size k, we can compute the parity check matrix as $H = [-A^T \quad I_{n-k}]$.

Definition 1.1.5 (Syndrome)**.** A syndrome of a vector $x \in \mathbb{F}_q^n$ is $s \in \mathbb{F}_q^{n-k}$ with $s = Hx^T$ where H is a parity-check matrix.

Having two vectors $x, y \in \mathbb{F}_q^n$ the Hamming distance between x and y is defined as $d_h(x, y) = |\{i \mid x_i \neq y_i\}|$ while the Hamming weight of a vector x is $\texttt{HW}(x) = \{i \mid x_i \neq 0\}$.

Definition 1.1.6 (Minimum distance of a linear code)**.** The minimum distance of a linear code C is:

$$d(C) = min\{d_h(c_1, c_2) \mid c_1, c_2 \in C \wedge c_1 \neq c_2\}$$

.

A linear code of length n, dimension k, and distance d is called an $[n, k, d]$ code.

The minimum distance of a linear code is a fundamental parameter: higher the minimum distance higher is the number of errors that the code can correct. It is relevant to note that a code C has $d(C) = d$ if and only if every set of $(d-1)$ columns of the parity check matrix H are linearly indipendent. For example if $d(C) = 2$, H has two columns that are linearly dependent. In the binary case this means that two columns of H are equals.
The singleton bound tells us that given a $[n, k]$ code the following relation holds $d(C) \leq n - k + 1$ since any $n - k + 1$ columns of H are linearly dependent because H has $n - k$ rows and n columns.
The next bound is an important one and tell us the condition of existance of a $[n, k, d]$ code setting a limit on the parameters.

Definition 1.1.7 (Gilbert-Varshamov bound)**.**

$$\sum_{i=0}^{d-2} (q-1)^i \binom{n-1}{i} < q^{n-k} \implies \text{Exists an } [n, k, d] \text{ code.}$$

The capability of a code for detecting an error and correcting an error is different. As we said before the sender sends the encoded message, a codeword $c \in C$. During the transimission, due to the noise, there can be different error patterns $e \in \mathbb{F}_q^n$ and so, the receivers receives $y = c + e \in \mathbb{F}_q^n$.

Definition 1.1.8 (Detectable errors)**.** Let C be an $[n, k]$ code with minimum distance d: any error pattern of size at most $d - 1$ can be detected. Moreover, the number of detectable errors are $q^n - q^k$.

If $\mathtt{HW}(e) \leq d - 1$ we can conclude that $y = c + e \notin C$ so we are able to detect an error if the initial condition is verified.
For correcting an error the situation is different. The minimum distance decoding after receiving $y = c + e$ looks for $x \in C$ such that $d_h(y, x)$ is minimized.

Theorem 1.1. *Let C be an $[n, k]$ code with minimum distance d then:*

$$C \textit{ can correct } t \textit{ errors} \iff t \leq \lfloor \tfrac{d-1}{2} \rfloor.$$

From this theorem we can see how the minimum distance plays a very important role since it determines the correcting error capabilities of a code.
Let us now focus on the decoding phase: we receive a vector composed by the encoded message with the generator matrix and the error pattern of the channel $y = Gm^T + e$ and we want to retrieve the message m that minimizes $d_h(y, Gm^T)$. The bruteforce way to do it is enumerating all the codewords of C, compute all the Hamming distances between y and the codewords and return the codeword that minimizes the Hamming distance, but this would cost $\mathcal{O}(nq^k)$.

If the receiver knows the code that has been used and an efficient decoding algorithm, he can directly retrieve the codeword. On the other hand, instead of retrieving the codeword directly, we can retrieve the error using the parity-check matrix and the syndrome definition previously introduced: if y is the received vector we know that $Hy^T = H(c+e)^T = Hc^T + He^T = He^T$. Therefore, having the syndrome $s = He^T$ we are interested in retrieving an error e. This problem will be fundamental with the Niederreiter code-based cryptosystem that we will see on the next section.

1.2. Code-based cryptosystems

In this section we introduce the two cryptosystems based on coding theory, the McEliece and the Niederreiter ones.

1.2.1. McEliece cryptosystem

The McEliece cryptosystem is an asymmetric encryption algorithm developed by Robert McEliece in 1978 [21] based on the hardness of decoding a random linear code. It has the advantage of having very fast encryption and decryption methods and it gained popularity later since it is a candidate as a post-quantum cryptosystem. One of the main disadvantage is the large size of the matrices used as public and private keys.
Suppose Bob wants to communicate with Alice, the working principle is the following: Alice chooses a linear code C from a family of codes having an efficient decoding algorithms capable of correcting w errors and being indistinguishable from random codes. The original algorithm uses binary Goppa codes family and it is still the best option so far. After have chosen the linear code, Alice has the generator matrix G that will be one part of the private key and only her knows it. Then, she selects a random $k \times k$ non-singular binary matrix S, a random $n \times n$ permutation matrix P and computes the $k \times n$ matrix $\widetilde{G} = SGP$. The matrix S and P are known only by Alice while the matrix $\widetilde{G}$ is known to everyone who wants to communicate with her.

The public key of Alice is $\langle \widetilde{G}, w \rangle$. The private key of Alice is $\langle S, P, G \rangle$

Let's see how the encryption works: Bob wants to send a message m to Alice whose public key is $\langle \widetilde{G}, w \rangle$.
First, Bob encodes the message m as a binary string of length k. Then, he computes $c' = \widetilde{G}m^T$ and he generates a random n-bit vector e containing exactly w errors. Finally he computes the ciphertext as $c = c' + e$.
Alice needs to decrypt the ciphertext c just received: she computes the inverse of the

matrix P, P^{-1}, and uses this inverse for computing $\widehat{c} = cP^{-1} = SGm^T + eP^{-1}$.
Now with the efficient decoding algorithm of the code known only by Alice she decodes $\widehat{c}$ obtaining $\widehat{m} = Sm^T$. Since S is invertible she can recover the original message m.
This scheme works thanks to the hardness of decoding random linear code: the original code with generator matrix G is hidden by the code generated with the matrix $\widetilde{G}$ obtained perturbating randomly G with the matrices S and P.

Definition 1.2.1 (Decoding random linear code)**.** Let $\widetilde{G} \in \mathbb{F}_q^{k\times n}$ be a random-looking generator matrix and $c = \widetilde{G}m^T + e$ the ciphertext with $e \in \mathbb{F}_q^n$ and $\mathtt{HW}(e) = w$. The decoding random linear code problem asks to recover the original message m.

1.2.2. Niederreiter cryptosystem

The Niederreiter cryptosystem was designed by Harald Niederreiter in 1986 [23] and it is a variant of the previously described McEliece cryptosystem based on the hardness of the syndrome decoding problem. The encryption of Niederreiter is ten time faster than the one of McEliece and a great advantage is the possibility to construct a digital signature scheme unlike in the McEliece scheme.
The first design used generalized Reed-Solomon codes but it was proven to be broke by Sidel'nikov and Shestakov in [28]. Replacing the Reed-Solomon codes with the Goppa codes yields a cryptosystem that is currently unbroken. As before, we describe the key generation, the encryption and the decryption assuming Alice and Bob want to exchange messages.
First, Alice chooses a binary $[n, k]$ linear Goppa code capable of correcting w errors and with an efficient syndrome decoding algorithm. Alice computes the parity-check matrix H of the code from G, chooses a random $(n - k) \times (n - k)$ non-singular binary matrix S and a random $n \times n$ permutation matrix P holding all of these matrices secret. She now computes $\widetilde{H} = SHP$ that will be known to everyone who wants to communicate with her.

The public key of Alice is $\langle \widetilde{H}, w\rangle$. The private key of Alice is $\langle S, P, H\rangle$

Since now the public key is a parity-check matrix we can see that Niederreiter scheme reduces the key size compared to the McEliece one.
The encryption works as follows: Bob knowing the public key of Alice $\langle \widetilde{H}, w\rangle$ wants to send a message m with weight $\leq w$. He computes the ciphertext as the syndrome of m: $c = \widetilde{H}m^T \in \mathbb{F}_q^{n-k}$.
Alice, after receiving c from Bob, computes $S^{-1}c = HPm^T$. Then, she applies the efficient

syndrome decoding algorithm for the code she chose, known only by her, to recover Pm^T. Now she can retrieve the message m by multiplying the inverse of the matrix P.
Here the original code C is hidden by $\widehat{C}$ using the parity-check matrix instead of the generator matrix as done in McEliece. Without knowing the private key the problem of finding the original message is known to be NP-hard and is formulated in the following definition.

Definition 1.2.2 (Syndrome decoding problem)**.** Let $\widetilde{H} \in \mathbb{F}_q^{(n-k)\times n}$ be a parity check matrix, $s = \widetilde{H}e^T \in \mathbb{F}_q^{n-k}$ and w an integer. The Syndrome decoding problem asks to find an error $e \in \mathbb{F}_q^n$ with $\mathtt{HW}(e) \leq w$ such that $\widetilde{H}e^T = s$.

To try to break these cryptosystems to understand their robustness, we need to analyze attacks against them that solves the decoding random linear code or the syndrome decoding problem. These problems are equivalent since one is the dual of the other and viceversa: solving one of these solves the other too. Knowing this, from know on we consider only the syndrome decoding problem for analyzing and breaking these cryptosystem because the parity check matrix is smaller than the generator matrix and so the computational effort for working with the H will be less. The only case in which the generator matrix G and the parity-check matrix H have the same dimension is when the code rate $R = \frac{k}{n} = 0.5$: in all the other case H is smaller than G since $R > 0.5$.
The attacks taken in consideration are the ones with the best complexities for solving the syndrome decoding problem and they are called Information Set Decoding (ISD) algorithms. After analyzing useful subroutines that will be used in the implementation of the ISDs, in chapter 3 we are going to explain the working principles of the ISD algorithms analyzing all the known ISD from the state of the art in the syndrome decoding problem perspective with the goal of breaking these cryptosystems.

2 | Useful algorithms used in the ISD

In this chapter we are going to explain and analyze different algorithms used as subroutines in the implementation of the Information Set Decoding techniques considering many variants and optimizations. In the first section we will study the reduced row echeleon form since, as we will see, the first step of every ISD is to compute the reduced row echelon form of a binary matrix H: this means bringing the matrix H in systematic form H_{rref} having a square identity sub-matrix on the left resulting in $H_{rref} = [I_r \quad V]$. Some authors specify the square identity matrix on the right part but in this thesis we adopt the convention of the identity on the left following the convention reported on the decoding challenge site [4] used for the generation of the test cases.
Next, we will analyze many variants of binary searches algorithms for using them subsequently in the ISD in the form of binary range searches. In the third section we will see an algorithm for generating all the possible combinations of p integers from 1 to k and an algorithm for optimizing the columns sum of a matrix. Finally, a small section on the sorting algorithms taken into consideration.

2.1. Reduced Row Echelon Form

Having a binary parity-check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ of a binary linear code C we are interested in computing its systematic form $H_{rref} = [I_r \quad V]$ where $I_r \in \mathbb{F}_2^{r\times r}$ and $V \in \mathbb{F}_2^{r\times k}$. From now on, we denote $r = n - k$: r represents the number of rows of the parity-check matrix H. We will analyze the standard RREF algorithm with no optimization, the RREF algorithm with the reusing pivots optimization, the standard partial RREF algorithm, the partial RREF with reusing existing pivots and adaptive information sets optimizations and last, the four russians matrix inversion algorithm for computing the systematic form of a matrix based on the work by Gregory V. Bard in [6].

2.1.1. Standard RREF

To obtain H_{rref} from the parity-check matrix H there are two main steps:

- First, we randomly permute the columns of the matrix H saving the permuted matrix in H_p. The permutations are held by an array χ of size n: initially this array contains the column's indices of H from 0 to $n-1$ and then it is shuffled randomly for producing a random permutation to apply to the matrix H.
- Then, we try to transform the left $r \times r$ part of the permuted matrix H_p in a full identity matrix of size $r \times r$. We do that applying row operations on the matrix H_p and we save all the operations done in a matrix U of size $r \times r$ initially equal to an identity matrix. If we end this step obtaining $H_p = [I_r \quad V]$, we have successfully found a correct systematic form and the matrix U contains the transformations applied during the algorithm, otherwise, we need to pick another random permutation and retry to compute a correct RREF until we find one.

For describing the algorithms we use the following notation: having a matrix H we indicate as $H_{(i,j)}$ the element of the matrix in the i-th row and in the j-th column. $H_{(i,:)}$ represents the full i-th row of the matrix while $H_{(:,j)}$ the full j-th column.
After we have been permuted the matrix H obtaining H_p, the row transformations for obtaining an identity matrix on the left works as follows.
We indicate with j the index traversing the columns of the matrix H_p while with z the index traversing the rows. The index j goes from 0, the first column, to $r-1$, the last column of the identity matrix that will be place on the left of H_p, while z goes from 0, the first row, to $r-1$, the last row.

1. We want that all the elements in the diagonal of the left sub-matrix of H_p are equal to one : $\forall 0 \leq j < r : H_{p(j,j)} = 1$ (the pivot). So, we check this diagonal elements and if we encounter one element $H_{p(j,j)} = 0$ we search a row $z > j$ such that $H_{p(z,j)} = 1$: if we can find one of this row we can swap the rows z and j for placing the pivot in the right position, otherwise we need to try another premutation since all the column elements are equal to zero.
2. After obtaining $H_{p(j,j)} = 1$, we need to set to zero all the elements in the column j except this one for having a column of the identity matrix: this is simply done saving on all the rows z such that $H_{p(z,j)} = 1$ the result obtained by xoring the row z with the row j containing the pivot.

In the next page we can see the pseudocode of the standard RREF algorithm just described.

Algorithm 2.1.1: Reduced Row Echelon Form

Input: $H \in \mathbb{Z}_2^{r \times n}$: binary parity-check matrix

Output: $H_p \in \mathbb{Z}_2^{r \times n}$: binary parity check matrix storing the current permutation of the matrix H, at the end of a successful RREF is in systematic form $H_p = [I_r \quad V]$
$U \in \mathbb{Z}_2^{r \times r}$: matrix saving the elementary row operations, initially is an identity matrix
$V \in \mathbb{Z}_2^{r \times k}$: right part of H_p at the end of the RREF
χ: array of size n storing the indices of the matrix H that represents the current permutation of H columns
$rref_error$: boolean variable representing if the computation of the RREF was successful or not with the current permutation

```
1  CLEAR_MATRIX(U)
2  CLEAR_MATRIX(H_p)
3  SET_IDENTITY_MATRIX(U)
4  for i ← 0 to n do
5  |   χ_i ← i
6  SHUFFLE(χ)
7  for i ← 0 to r do
8  |   for j ← 0 to n do
9  |   |   if H_(i,χ_j) = 1 then
10 |   |   |   H_p(i,j) ← 1
11 for j ← 0 to r do
12 |   if H_p(j,j) = 0 then
13 |   |   for z ← j + 1 to r do
14 |   |   |   if H_p(z,j) = 1 then
15 |   |   |   |   SWAP_ROWS(H_p, j, z)
16 |   |   |   |   SWAP_ROWS(U, j, z)
17 |   if H_p(j,j) = 0 then
18 |   |   return true;
19 |   for z ← 0 to r do
20 |   |   if z ≠ j then
21 |   |   |   if H_p(z,j) = 1 then
22 |   |   |   |   H_p(z,:) ← H_p(j,:) ⊕ H_p(z,:)
23 |   |   |   |   U_(z,:) ← U_(j,:) ⊕ U_(z,:)
24 for i ← 0 to r do
25 |   for j ← r to n do
26 |   |   V_(i,j) ← H_p(i,j)
27 return false;
```

The computational complexity of the standard RREF Algorithm 2.1.1 is:

$$C_{RREF}(n,r) = \mathcal{O}\left(\frac{1}{2}\sum_{i=0}^{r-1}(r-i)(n+r) + \frac{1}{2}r(r-1)(n+r) + r(n-r)\right)$$

This cost is derived doing the following reasoning: the algorithm starts checking if the current row has the pivot in the right position and if it is not, we have to search in the rows below it if there is a 1 in the wanted position. For the first row we need to search in the $r-1$ rows below it, for the second row in the $r-2$ rows below it and so on until the last row. The probability of swapping a row is $\frac{1}{2}$ since the value of the pivot can be 0 or 1. We need to swap also the rows of the matrix U: therefore, we have a swapping cost for placing the pivot in the right place of $\mathcal{O}(\frac{1}{2}\sum_{i=0}^{r-1}(r-i)(n+r))$ (H has n columns while U has r columns). Then, we have to set to zero all the elements in the column with the new pivot: the $r-1$ elements in the same column of the pivot have a probability equal to $\frac{1}{2}$ to be equal to 1. Every row with these elements needs to be xored with the row holding the pivot and the same xor operation needs to be done on the matrix U. Having r rows, we obtain a cost of $\mathcal{O}(\frac{1}{2}r(r-1)(n+r))$ that takes into account the xoring of the rows during the RREF. The last term is for saving the right part of the systematic form inside the matrix V.

2.1.2. RREF with reusing existing pivots optimization

As we will see in the next chapter, the ISD algorithm after computing a correct systematic form can fail to find a target error, so, they need to compute a different systematic form that could be a useful one. Instead of calling repeatedly the previous algorithm to find each time a new RREF with a new permutation starting from the original matrix H, we can exploit the fact that after the first successfull call of the previous algorithm, the returned matrix is in systematic form, and if we want to compute a different form, we can use this matrix as a starting point for optimizing the process. Therefore this time, we assume to have in input a matrix H already in systematic form, $H = [I_r \quad V]$. As before, initially, we permute randomly the matrix H obtaining a permuted matrix $H_p = [X \quad T]$ using the array χ. Here in the left sub-matrix X we can have columns of the previously identity matrix: probabilistically X will contain about $\frac{r^2}{n}$ columns of I_r because the probability of a single column of H_p to be part of the previous I_r is $\frac{r}{n}$ (I_r has only r columns while H has a total of n columns). The advantage here is that we can just swap the rows for placing the pivots in the right position saving the cycles of the previous algorithm for the index that respects this constraint. We can ignore in average $\frac{r^2}{n}$ columns of H_p during the computation of the RREF and this can speed up the algorithm a lot.

Let's see an example for understanding how it works before seeing the pseudocode of the algorithm. Consider an input matrix H with $r = 3$ and $n = 6$ already in systematic form:

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Now we permute the original matrix with a random χ obtaining H_p:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

We can notice that the first column and the third column contain only one 1 and so we can just swap the rows for placing the pivots in the correct place. We save the index of the columns that are already correct in an array called *pivots*. We place the pivot of the first row in the correct position swapping the first and the third rows resulting in:

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix}$$

And finally we swap the second and the third rows:

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Now we apply the same transformations of the standard RREF for obtaining a new systematic form taking into account that $pivots = [0, 2]$. Knowing that, we only need to transform the second column and not three columns as before for having an identity matrix on the left.

The computational complexity of the RREF with reusing existing pivots described in the Algorithm 2.1.2 is:

$$C_{RREF}(n,r) = \mathcal{O}\left(r + \frac{r^2}{n}(n+r) + \frac{1}{2}\sum_{i=\frac{r^2}{n}}^{r-1}(r-i)(n+r) + \frac{1}{2}(r - \frac{r^2}{n})(r-1)(n+r) + r(n-r)\right)$$

First, we have to initialize the *pivots* array doing a check for every row of the matrix resulting in a cost equal to r. Then, with the already discussed probability equal to $\frac{r^2}{n}$,

we swap the rows for placing the pivots in the correct positions based on the indices saved in the array, swapping either the rows in the matrix U resulting in a cost equal to $\frac{r^2}{n}(n+r)$. The other terms are derived following the same reasoning used in the standard RREF except that here, we avoid doing any operations for the columns indexed in the array *pivots*.

2.1.3. Partial Reduced Row Echelon Form

Some ISD algorithms will need to work not with a full systematic form as the one returned by the previosuly described algorithms, but with a partial systematic form. These algorithms, starting from the Finiasz-Sendrier one, introduce a new parameter $0 \leq \ell < r-1$ for computing the so called partial systematic form. Having in input a binary parity-check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ we want to compute its partial systematic form $H_{prref} \in \mathbb{F}_2^{(n-k)\times n}$ structured like in the following figure:

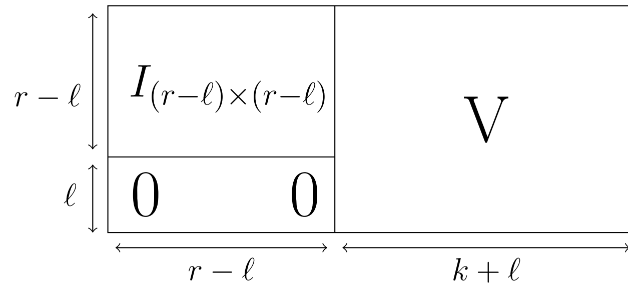


Figure 2.1: Partial systematic form of H

As we can see we need only an identity matrix of size $(r-\ell)\times(r-\ell)$ with a null matrix on the bottom of size $\ell \times (r-\ell)$. It follows the same steps done in the standard RREF, the only thing that changes is that we consider the index of the column j not anymore traversing from column 0 to the $r-1$-th column but from the 0 to the $(r-\ell-1)$-th column.

The computational complexity of the partial RREF described in the Algorithm 2.1.3 is:

$$C_{P-RREF}(n,r,\ell) = \mathcal{O}\Big(\frac{1}{2}\sum_{i=0}^{r-\ell-1}(r-i)(n+r) + \frac{1}{2}(r-\ell)(r-1)(n+r) + r(n-r-\ell)\Big)$$

This is derived using the same reasoning applied to the standard RREF complexity cost: the only thing that changes is that the columns of the identity matrix are $r-\ell$ and not r like in the RREF algorithm.

2.1.4. Optimized Partial RREF

The optimized partial RREF algorithm takes into account two optimizations: the already discussed resuing existing pivots and the adaptive information set optimization.
The reusing existing pivots procedure is exactly the same as the one explained before in the case of the full systematic form, we need only to change the range traversed by the columns index j.
The adaptive information set optimization is explained in the following. At the end of the partial RREF we check if the identity matrix has size $(r-\ell) \times (r-\ell)$: if it is not the case we need to restart the algorithm with a new columns permutation loosing all the work done. This seems inefficient, in fact we can adapt our choice of the $r-\ell$ columns swapping those columns that introduce the linear dependency between the $r-\ell$ columns with the $k+\ell$ columns placed on the right. If we do this we need to update the permuatation array χ accordingly. Let's see an example for understanding the method with parameters $n = 5$, $k = 2$ and $\ell = 1$. Consider the matrix H_p at the end of the partial RREF with the following permutation array $\chi = [3, 4, 1, 0, 2]$:

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

We can see that the second column is not correct since we expect an identity matrix with dimension 2×2 on the upper-left $(r - \ell = 2)$. The standard RREF starts again taking another permutation but thanks to the adaptive information set we can find a column on the right with a 1 in the second row. From the example we can take the third column and swapping it with the second updating the permutation array $\chi = [3, 1, 4, 0, 2]$:

$$\begin{bmatrix} 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

Now we are able to compute the partial RREF saving in the third row the xor between the second row and itself obtaining the desired form:

$$\begin{bmatrix} 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

The computational complexity of the partial RREF optimized described in the Algorithm 2.1.4 is:

$$C_{P-RREF}(n,r,\ell) = \mathcal{O}\Bigg((r-\ell)+\frac{r^2}{n}(n+r)+\frac{1}{2}\sum_{i=\frac{r^2}{n}}^{r-\ell-1}(r-i)(n+r)+\frac{1}{2}(r-\ell-\frac{r^2}{n})(r-1)(n+r)+r(n-r-\ell)\Bigg)$$

The same reasoning of the RREF with reusing pivots has been followed, the only thing that changes is the size of the identity matrix to produce that is $(r-\ell)\times(r-\ell)$ instead of $r \times r$. The adaptive information set optimization doesn't change the cost complexity since it decreases the number of iterations for finding the identity matrix compared to the standard partial RREF.

Algorithm 2.1.2: RedRowEchelonForm: Reusing Existing Pivot

Input: $H \in \mathbb{Z}_2^{r\times n}$: binary parity-check matrix already in red row echelon form

Output: $H_p \in \mathbb{Z}_2^{r\times n}$: binary parity check matrix storing the current permutation of the matrix H, at the end of the RREF is in systematic form $H_p = [I_r \quad V]$

$U \in \mathbb{Z}_2^{r\times r}$: matrix representing elementary row operations, originally is an identity matrix

$V \in \mathbb{Z}_2^{r\times k}$: right part of H_p at the end of the RREF

χ: an array of size n storing the indices of the matrix H that represents the current permutation of H columns

$rref_error$: boolean variable representing if the computation of the RREF was succesful or not

Data: $pivots$: an array of size r containing the indices of the columns with only one 1 among the first r columns of H_p

```
1  CLEAR_MATRIX(U)
2  CLEAR_MATRIX(H_p)
3  SET_IDENTITY_MATRIX(U)
4  for i ← 0 to n do
5  |  χ_i ← i
6  SHUFFLE(χ)
7  for i ← 0 to r do
8  |  for j ← 0 to n do
9  |  |  if H_(i,χ_j) = 1 then
10 |  |  |  H_p(i,j) ← 1
11 for i ← 0 to r do
12 |  if χ_i < r then
13 |  |  pivots ← pivots ∪ i
14 for z ← 0 to SIZE(pivots) do
15 |  j ← pivots_z
16 |  for i ← 0 to r do
17 |  |  if H_(i,j) = 1 then
18 |  |  |  SWAP_ROWS(H_p, i, j)
19 |  |  |  SWAP_ROWS(U, i, j)
20 m ← 0
21 m_swap ← 0
22 for j ← 0 to r do
23 |  if j ≠ pivots_m then
24 |  |  if H_p(j,j) ≠ 1 then
25 |  |  |  m_swap ← m
26 |  |  |  for z ← j + 1 to r do
27 |  |  |  |  if z ≠ pivots_m_swap then
28 |  |  |  |  |  if H_p(z,j) = 1 then
29 |  |  |  |  |  |  SWAP_ROWS(H_p, j, z)
30 |  |  |  |  |  |  SWAP_ROWS(U, j, z)
31 |  |  |  |  |  |  break
32 |  |  |  |  else
33 |  |  |  |  |  m_swap ← m_swap + 1
34 |  |  |  if H_p(j,j) = 0 then
35 |  |  |  |  return true;
36 |  |  for z ← 0 to r do
37 |  |  |  if (z ≠ j) ∧ (H_p(z,j) = 1) then
38 |  |  |  |  H_p(z,:) ← H_p(j,:) ⊕ H_p(z,:)
39 |  |  |  |  U_(z,:) ← U_(j,:) ⊕ U_(z,:)
40 |  else
41 |  |  m ← m + 1
42 for i ← 0 to r do
43 |  for j ← r to n do
44 |  |  V_(i,j) ← H_p(i,j)
45 return false;
```

Algorithm 2.1.3: Partial Reduced Row Echelon Form

Input: $H \in \mathbb{Z}_2^{r \times n}$: binary parity check matrix

Output: $H_p \in \mathbb{Z}_2^{r \times n}$: binary parity check matrix, at the end of the RREF is in partial systematic form $H_p = [I_{r-\ell} \quad V]$ where V is an $r \times (k+\ell)$ binary matrix and $I_{r-\ell}$ is composed in the upper part by an $(r-\ell) \times (r-\ell)$ binary identity matrix and in the lower part by a $\ell \times (r-\ell)$ matrix of zeroes

$U \in \mathbb{Z}_2^{r \times r}$: an $r \times r$ binary matrix representing elementary row operations, originally is an identity matrix

χ: an array of size n storing the indices of the matrix H that represents the current permutation of H columns

$rref_error$: boolean variable representing if the computation of the RREF was succesful or not

```
1  CLEAR_MATRIX(U)
2  CLEAR_MATRIX(H_p)
3  SET_IDENTITY_MATRIX(U)
4  for i ← 0 to n do
5  |   χ_i ← i
6  SHUFFLE(χ)
7  for i ← 0 to r do
8  |   for j ← 0 to n do
9  |   |   if H_(i,χ_j) = 1 then
10 |   |   |   H_p(i,j) ← 1
11 for j ← 0 to r − ℓ do
12 |   if H_p(j,j) = 0 then
13 |   |   for z ← j + 1 to r do
14 |   |   |   if H_p(z,j) = 1 then
15 |   |   |   |   SWAP_ROWS(H_p, j, z)
16 |   |   |   |   SWAP_ROWS(U, j, z)
17 |   if H_p(j,j) = 0 then
18 |   |   return true;
19 |   for z ← 0 to r do
20 |   |   if z ≠ j then
21 |   |   |   if H_p(z,j) = 1 then
22 |   |   |   |   H_p(z,:) ← H_p(j,:) ⊕ H_p(z,:)
23 |   |   |   |   U_(z,:) ← U_(j,:) ⊕ U_(z,:)
24 for i ← 0 to r do
25 |   for j ← r to n do
26 |   |   V_(i,j) ← H_p(i,j)
27 return false;
```

Algorithm 2.1.4: PartialRedRowEchelonForm: Optimized

Input: $H \in \mathbb{Z}_2^{r \times n}$: binary parity check matrix already in partial red row echelon form

Output: $H_p \in \mathbb{Z}_2^{r \times n}$: binary parity check matrix, at the end of the RREF is in partial systematic form $H_p = [I_{r-\ell} \quad V]$ where V is an $r \times (k+\ell)$ binary matrix and $I_{r-\ell}$ is composed in the upper part by an $(r-\ell) \times (r-\ell)$ binary identity matrix and in the lower part by a $\ell \times (r-\ell)$ matrix of zeroes
$U \in \mathbb{Z}_2^{r \times r}$: an $r \times r$ binary matrix representing elementary row operations, originally is an identity matrix
χ: an array of size n storing the indices of the matrix H that represents the current permutation of H columns
$rref_error$: boolean variable representing if the computation of the RREF was succesful or not

Data: $pivots$: an array of maximum size $r-\ell$ containing the indices of the columns with only one 1 among the first $r-\ell$ columns of H_p

```
1  CLEAR_MATRIX(U)
2  CLEAR_MATRIX(H_p)
3  SET_IDENTITY_MATRIX(U)
4  for i ← 0 to n do
5  |   χ_i ← i
6  SHUFFLE(χ)
7  for i ← 0 to n do
8  |   H_p(:,i) = H_(:,χ_i)
9  for i ← 0 to r − ℓ do
10 |   if χ_i < r − ℓ then
11 |   |   pivots ← pivots ∪ i
12 for z ← 0 to SIZE(pivots) do
13 |   j ← pivots_z
14 |   for i ← 0 to r do
15 |   |   if H_(i,j) = 1 then
16 |   |   |   SWAP_ROWS(H_p, i, j)
17 |   |   |   SWAP_ROWS(U, i, j)
18 m ← 0
19 m_swap ← 0
20 for j ← 0 to r − ℓ do
21 |   if j ≠ pivots_m then
22 |   |   if H_p(j,j) ≠ 1 then
23 |   |   |   found ← false
24 |   |   |   m_swap ← m
25 |   |   |   for z ← j to r do
26 |   |   |   |   skip_pivot ← false
27 |   |   |   |   for i ← m_swap to SIZE(pivots) do
28 |   |   |   |   |   if pivots_i > z then
29 |   |   |   |   |   |   break
30 |   |   |   |   |   if z = pivots_i then
31 |   |   |   |   |   |   skip_pivot ← true
32 |   |   |   |   |   |   m_swap ← i
33 |   |   |   |   |   |   break
34 |   |   |   |   if !skip_pivot then
35 |   |   |   |   |   if H_p(z,j) = 1 then
36 |   |   |   |   |   |   SWAP_ROWS(H_p(j,:), H_p(z,:))
37 |   |   |   |   |   |   SWAP_ROWS(U_(j,:), U_(z,:))
38 |   |   |   |   |   |   found ← True
39 |   |   |   |   |   |   break
40 |   |   |   |   |   else
41 |   |   |   |   |   |   m_swap ← m_swap + 1
42 |   |   |   if !found then
43 |   |   |   |   for z ← r − ℓ to n do
44 |   |   |   |   |   if H_(j,z) = 1 then
45 |   |   |   |   |   |   SWAP_COLUMNS(H_p(:,j), H_p(:,z))
46 |   |   |   |   |   |   SWAP_POSITIONS(α_j, α_z)
47 |   |   |   |   |   |   break
48 |   |   |   |   |   if z = n then
49 |   |   |   |   |   |   return true;
50 |   |   for z ← 0 to r do
51 |   |   |   if (z ≠ j) ∧ (H_p(z,j) = 1) then
52 |   |   |   |   H_p(z,:) ← H_p(j,:) ⊕ H_p(z,:)
53 |   |   |   |   U_(z,:) ← U_(j,:) ⊕ U_(z,:)
54 |   else
55 |   |   m ← m + 1
56 for i ← 0 to r do
57 |   for j ← r − ℓ to n do
58 |   |   V_(i,j) ← H_p(i,j)
59 return false;
```

The explicit time complexities, derived from the formulas seen for each RREF method, and the space complexities of the RREF algorithms just described are reported in the following tables.

Table 2.1: Time complexities of the RREF procedures

	Time complexity of one iteration: C_{RREF}
Standard RREF	$\frac{3nr^2}{4} + \frac{nr}{4} - \frac{n}{2} + \frac{3r^3}{4} - \frac{3r^2}{4} - \frac{r}{2}$
RREF reusing pivots	$-\frac{r^5}{4n^2} - \frac{3r^4}{4n} + \frac{3r^3}{4n} + \frac{3nr^2}{4} + \frac{nr}{4} - \frac{n}{2} + \frac{r^3}{4} + \frac{r}{2}$
Partial RREF	$-\frac{\ell^2 n}{4} - \frac{\ell^2 r}{4} - \frac{lnr}{2} - \frac{ln}{4} - \frac{lr^2}{2} - \frac{lr}{4} + \frac{3nr^2}{4} + \frac{nr}{4} - \frac{n}{2} + \frac{3r^3}{4} - \frac{3r^2}{4} - \frac{r}{2}$
Partial RREF optimized	$-\frac{\ell^4 r}{4n^2} - \frac{\ell^4}{4n} + \frac{\ell^3 r^2}{2n} - \frac{\ell^3}{2} - \frac{3\ell^2 r^3}{2n^2} - \frac{\ell^2 r^2}{n} + \frac{3\ell^2 r}{4n} - \frac{\ell^2 n}{4} + \frac{\ell^2 r}{4} + \frac{3\ell^2}{4} + \frac{\ell r^4}{n^2} + \frac{3\ell r^3}{2n} - \frac{3\ell r^2}{2n} - \frac{\ell nr}{2} - \frac{\ell n}{4} - \frac{7\ell r}{4} - \ell - \frac{r^5}{4n^2} \frac{3r^4}{4n} + \frac{3r^3}{4n} + \frac{3nr^2}{4} + \frac{nr}{4} - \frac{n}{2} + \frac{r^3}{4} + \frac{r}{2}$

Table 2.2: Space complexities of the RREF procedures

	Space complexity: S_{RREF}
Standard RREF	$\mathcal{O}(r^2 + rn + n)$
RREF reusing pivots	$\mathcal{O}(r^2 + rn + n + r)$
Partial RREF	$\mathcal{O}(r^2 + rn + n)$
Partial RREF optimized	$\mathcal{O}(r^2 + rn + n + r)$

2.1.5. Method of four russians for inversion

The previously described procedures for computing a systematic form of a matrix have been implemented in this thesis, but a method based on the method of four russians for

computing the systematic form is available publicly in the M4RI library [2]. In the testing section the procedure based on M4RI used for producing the results is inside the M4RI light project available in the implementation of Dumer's algorithm by Valentin Vasseur in [30]: as Vasseur said "This is a lighter version of the M4RI library. In order to save a few CPU cycles, it has been simplified and stripped of many functions and tests that are not required for our usage. The echelonize function has been modified to allow a partial gaussian elimination".
The computation of the systematic form using the M4RI method is based on the work by Gregory V. Bard in [6]. In this work Bard presents the Four Russians inversion algorithm and it can be used for computing the gaussian elimination and even the reduced row echelon form. The method described by G.Bard in his paper for computing an unit upper triangular form is explained in the following and as we will see, with a small modification, we can compute even the RREF following the same guide lines.
Having an $r \times n$ matrix H in input, in the M4RI algorithm q columns are processed at once, producing a $q \times q$ identity matrix in the correct spot $(h_{i,i} \dots h_{(i+q-1),(i+q-1)})$ with all zeroes below it, leaving the region above the submatrix untouched.

1. Let h_i be the first column to be processed in a given iteration. Then, a gaussian elimination is performed on the first $3q$ rows after including the i-th row to produce an identity matrix in $h_{i,i} \dots h_{(i+q-1),(i+q-1)}$ and zeroes in $h_{(i+q),i} \dots h_{(i+3q-1),(i+q-1)}$

2. Build a table consisting of the 2^q binary strings of length q in a Gray Code. A Gray code is an ordering of the binary numeral system such that two successive values differ in only one bit: having a binary value we can produce its consecutive doing only one bit change. For example, the value 1 in Gray Code is 001, the value 2 is 011, the value 3 is 010 and the value 4 is 110: as we can see two successive values differ only by one bit. Thus with only 2^q vector additions, all possible linear combinations of the q rows have been precomputed since only one vector addition is needed for calculating each line of this table.

3. Then, it is possible to process the remaining rows from $i + 3q$ until the last row r by using the table built at the previous step. As an example, suppose the j-th row has entries $h_{j,i} \dots h_{j,(i+q-1)}$ in the columns being processed before. Now we select the row of the table associated with this q-bit string, and adding it to row j will force the q columns to zero, and then adjust the remaining columns from $i + q$ to n in the appropriate way, as if Gaussian Elimination had been performed. The process is repetead $\frac{min(r,n)}{q}$ times. Each iteration resolves q columns instead of one but the running time is not q times faster because there is a trade off on the value of q and the cost of building the table on the second step.

Thus, the choice of the parameter q must be optimized taking into account the size of the table to build. For computing the RREF of a matrix we need to run the third step on rows $0 \dots i-1$ as well as on rows $i+3q \dots r$. This procedure can be used for computing even the partial RREF algorithm.
On Algorithm 2.1.5 it's reported the procedure for computing the reduced row echelon form with the M4RI method. Its analysis can be found even in [3].

Algorithm 2.1.5: Method of the Four Russians Inversion (M4RI)

Input: $H \in \mathbb{Z}_2^{r \times n}$: binary parity check matrix with r rows and n columns
q: number of columns that are processed at once, it needs to be optimized

Data: j, c: indices traversing respectively the rows and the columns of H
T: table containing 2^q binary vectors of length q

Output: $H \in \mathbb{Z}_2^{r \times n}$: binary parity check matrix in reduced row echelon form

```
1  j ← 0
2  c ← 0
3  while c < n do
4      if c + q > n then
5          q ← n − c
6      q̂ ← GaussSubMatrix(H, j, c, q, r)
7      if q̂ > 0 then
8          T ← MakeTable(H, j, c, q̂)
9          AddRowsFromTable(H, 0, j, c, q̂, T)
10         AddRowsFromTable(H, j + q̂, r, c, q̂, T)
11     j ← j + q̂
12     c ← c + q̂
13     if q ≠ q̂ then
14         c ← c + 1
```

The GaussSubMatrix routine performs a gaussian elimination on a $q \times n$ submatrix of H starting at position (j, c) and searches for pivot rows up to r. If a matrix of full rank q is not found, it returns the rank $\widehat{q}$ found so far. The MakeTable builds the table T discussed before holding all the 2^q linear combinations of the q rows starting from (j, c). Finally, the AddRowsFromTable procedure adds the appropriate linear combination in T onto a row i in order to clear q columns.

2.2. Binary Search variants

Another fundamental algorithm that will be used in the ISD implementation is the binary search one. Generally speaking, having a sorted array we want to find the position of the array where a target value resides. It is possible that the target value is not present in the array, in this case we return a non valid index. In the ISD algorithms we will have a sorted list that, most of the time, will contain duplicates. So, having a target value we would like to find the range in the sorted list such that the target value is equal to the values indexed by this range in the list: this means finding the extreme left index and the extreme right index such that all the indices between the left and the right (included them) have the same value of the target. For simplicity, in this section we are going to analyze the binary searches that given a target value returns only one index: the extreme right index if in the array there are multiples values equal to the target. It is easy to generalize the binary search in a binary range search: when we find a position where the target resides in the array, we start a search considering this position as the upper bound for finding the extreme left index and another one considering this position as the lower bound for finding the extreme right index. In the last subsection of this part an example of binary range search using the boundless binary search is reported.
Many searches have been analyzed for understanding the most efficient one to use in the implementation. All the variants of the binary search explained in this section are taken by the work "A collection of improved binary search algorithms" [25].

2.2.1. Binary Search

The binary search algorithm is the standard one: having a sorted array A we want to find the index where the target value key is placed in A. We start comparing the target with the middle element of the array and if they are not equal, the half in which the target cannot lie is deleted and the search continues on the remaining half. We repeat this procedure in which the half becomes smaller and smaller until the target value is found or concluding that it is not present in the array. In the worst case the running time of the binary search is logarithmic, $\mathcal{O}(\log n)$, where n is the size of the input array, since it makes $\mathcal{O}(\log n)$ comparisons in the worst case. The worst case is when the element is not present in the array or it is in the last remained position to analyze, where the last position is reached when all the possible halves of the array has been considered that are $\log(n)$. In Algorithm 2.2.1 we can see the pseudocode of the standard binary search.

Algorithm 2.2.1: Standard Binary Search

Input: A: sorted array
n: size of the array A
key: target value to find in A

Data: bot: bottom index of the current half
top: top index of the current half
mid: middle index of the current half

Output: the index where key is present in A, return -1 if it is not present.

```
1  if n = 0 then
2  |  return −1
3  bot ← 0
4  top ← n − 1
5  while bot < top do
6  |  mid ← top − (top−bot)/2
7  |  if key < A[mid] then
8  |  |  top ← mid − 1
9  |  else
10 |  |  bot ← mid
11 if key = A[top] then
12 |  return top
13 return −1
```

2.2.2. Boundless Binary Search

The boundless binary search uses only two indices, *bot* and *mid*: the *mid* index starts from outside the array (the position after the last valid element) and at each iteration is halved. Inside the cycle if the target value is greater or equal than the value at position $bot + \frac{mid}{2}$ the new bottom is set to $bot + \frac{mid}{2}$ eliminating the part before it since the element cannot be there and the *mid* is incremented by one. Otherwise the *bot* index remains at the current position and the *mid* one is halved: this is done to eliminate the part after $bot + \frac{mid}{2}$ since the target is less than the element in that index and so it is useless. This search performs better than the standard binary one since the loop contains 1 key check, 1 integer check, and (on average) 1.5 integer assignments.

Algorithm 2.2.2: Boundless Binary Search

Input: A: sorted array
n: size of the array A
key: target value to find in A

Data: bot: index starting at the bottom of A
mid: index starting outside A

Output: the index where key is present in A, return -1 if it is not present.

```
1  if n = 0 then
2  |  return −1
3  bot ← 0
4  mid ← n
5  while mid > 1 do
6  |  if key ≥ A[bot + mid/2] then
7  |  |  bot ← bot + mid/2
8  |  |  mid ← mid + 1
9  |  mid ← mid/2
10 if key = A[bot] then
11 |  return bot
12 return −1
```

2.2.3. Doubletapped Binary Search

The doubletapped binary search uses the same reasoning of the previous searches highlighting that when they are left with 2 elements, they take exactly 2 key checks to finish the procedure. The doubletapped binary search performs two equality checks at the end, so, when there are only two elements left, it can finish with 1 key check or 2: this in average results in performing less key checks than the previous algorithms. The doubletapped binary search is described in Algorithm 2.2.3.

2.2.4. Monobound Binary Search

The monobound binary search is exactly the same as the boundless binary search but uses an extra variable to simplify some calculations and performs slightly more keychecks as we can see in Algorithm 2.2.4.

Algorithm 2.2.3: Doubletapped Binary Search

Input: A: sorted array
n: size of the array A
key: target value to find in A
Data: bot: index starting at the bottom of A
mid: index starting outside A
Output: the index where key is present in A, return -1 if it is not present.

```
1  if n = 0 then
2  |  return −1
3  bot ← 0
4  mid ← n
5  while mid > 2 do
6  |  if key ≥ A[bot + mid/2] then
7  |  |  bot ← bot + mid/2
8  |  |  mid ← mid + 1
9  |  mid ← mid/2
10 while mid > 0 do
11 |  mid ← mid − 1
12 |  if key = A[bot+mid] then
13 |  |  return bot + mid
14 return −1
```

Algorithm 2.2.4: Monobound Binary Search

Input: A: sorted array
n: size of the array A
key: target value to find in A
Data: bot: index starting at the bottom of A
top: index starting outside A
mid index containing the half of top
Output: the index where key is present in A, return -1 if it is not present.

```
1  if n = 0 then
2  |  return −1
3  bot ← 0
4  top ← n
5  while top > 1 do
6  |  mid ← top/2
7  |  if key ≥ A[bot+mid] then
8  |  |  bot ← bot + mid
9  |  top ← top − mid
10 if key = A[bot] then
11 |  return bot
12 return −1
```

2.2.5. Tripletapped Binary Search

The Tripletapped binary search follows the same line of reasoning of the double tapped but this time improving the monobound binary search and not the boundless binary search. When we arrive at the end of a binary search and there are 3 elements left to analyze it takes 2.5 checks to finish, however, the monobound binary search takes 3 checks. So, the tripletapped variant performs 3 equality checks at the end resulting in slighly fewer key check in average.

Algorithm 2.2.5: Tripletapped Binary Search

Input: A: sorted array
n: size of the array A
key: target value to find in A

Data: bot: index starting at the bottom of A
top: index starting outside A
mid index containing the half of top

Output: the index where key is present in A, return -1 if it is not present.

```
1  if n = 0 then
2  |   return −1
3  bot ← 0
4  top ← n
5  while top > 3 do
6  |   mid ← top/2
7  |   if key ≥ A[bot+mid] then
8  |   |   bot ← bot + mid
9  |   top ← top − mid
10 while top > 0 do
11 |   top ← top − 1
12 |   if key = A[bot+top] then
13 |   |   return bot + top
14 return −1
```

2.2.6. Monobound Quaternary Search

The Monobound quaternary search performs more key checks than the monobound binary search but in some cases when the input array is very large can run faster. If the size of the array is greater than a certain value, the algorithm computes the mid index as the

quarter of the *top* index instead of its half. This is done for trying to be faster on the searching of the target value eliminating each time a bigger portion of the array where we know the target cannot lie. Probabilistically, this makes sense if we are working with a very large array. The author chooses to consider an array to be large if its size is greater than 2^{16}: if this happens we can eliminate quarters of the array instead of halves until the *top* index is greater than 2^{16}. If *top* becomes smaller than 2^{16} the same procedure seen in the tripletapped binary search is used for finding the target value in the range of the array that has been left. The procedure is reported in Algorithm 2.2.6.

Algorithm 2.2.6: Monobound Quaternary Search

Input: A: sorted array
n: size of the array A
key: target value to find in A

Data: bot: index starting at the bottom of A
top: index starting outside A
mid index containing the half or the quarter of top depending on the iteration and on the size of A

Output: the index where key is present in A, return -1 if it is not present.

```
1  if n = 0 then
2  |  return −1
3  bot ← 0
4  top ← n
5  while top ≥ 65536 do
6  |  mid ← top/4
7  |  top ← top − mid ∗ 3
8  |  if key < A[bot+mid×2] then
9  |  |  if key ≥ A[bot+mid] then
10 |  |  |  bot ← bot + mid
11 |  else
12 |  |  bot ← bot + mid × 2
13 |  |  if key ≥ A[bot+mid] then
14 |  |  |  bot ← bot + mid
15 while top > 3 do
16 |  mid ← top/2
17 |  if key ≥ A[bot+mid] then
18 |  |  bot ← bot + mid
19 |  top ← top − mid
20 while top > 0 do
21 |  top ← top − 1
22 |  if key = A[bot+top] then
23 |  |  return bot + top
24 return −1
```

2.2.7. Monobound Interpolated Binary Search

In the Monobound interpolated binary search, for calculating the remaining search space in the array in which the target value might be, the target value is used as in all the previously described algorithm but even the minimum and the maximum value present in the array. Instead of comparing the target value with the value in the half of the array as done in the standard binary search, we take in consideration the actual values present in it for computing the estimated position via a linear interpolation. The target is compared against it for eliminating the left part or the right part of the estimated position depending on the output of the key check. If we know the distribution of the elements inside the array we can tune the interpolation resulting in a very efficient search. In the Algorithm 2.2.7 the interpolation is done assuming an uniform distribution between the elements of the array: when the distribution is uneven the performance will drop, but not significantly.

Algorithm 2.2.7: Monobound Interpolated Binary Search

Input: A: sorted array
n: size of the array A
key: target value to find in A
Data: bot: index starting with the interpolated value
top: index starting outside A
mid: index containing the half top
min: integer containing the minimum value present in A
max: integer containing the maximum value present in A
Output: the index where key is present in A, return -1 if it is not present.

```
1  if n = 0 ∨ key < A[0] then
2  |   return −1
3  bot ← n − 1
4  if key ≥ A[bot] then
5  |   return (A[bot] = key ? bot : −1)
6  min ← A[0]
7  max ← A[bot]
8  top ← 64
9  if key ≥ A[bot] then
10 |   while 1 do
11 |   |   if bot + top ≥ n then
12 |   |   |   top ← n − bot
13 |   |   bot ← bot + top
14 |   |   if key < A[bot] then
15 |   |   |   bot ← bot − top
16 |   |   |   BREAK
17 |   |   top ← top × 2
18 else
19 |   while 1 do
20 |   |   if bot < top then
21 |   |   |   top ← bot
22 |   |   |   bot ← 0
23 |   |   |   BREAK
24 |   |   bot ← bot − top
25 |   |   if key ≥ A[bot] then
26 |   |   |   BREAK
27 |   |   top ← top × 2
28 while top > 3 do
29 |   mid ← top/2
30 |   if key ≥ A[bot+mid] then
31 |   |   bot ← bot + mid
32 |   top ← top − mid
33 while top > 0 do
34 |   top ← top − 1
35 |   if key = A[bot+top] then
36 |   |   return bot + top
37 return −1
```

2.2.8. Adaptive Binary Search

The Adaptive binary search is optimized if we need to do repeated binary searches on the same array. When it observes a pattern it switches from a binary search to a monobound search. This works well on uneven distribution unlike the interpolated one. Since this procedure adapts on multiple runs some variables must be shared upon all the runs.

Algorithm 2.2.8: Adaptive Binary Search

Input: A: sorted array
n: size of the array A
key: target value to find in A
Data: bot: index starting with the interpolated value
top: index starting outside A
mid: index containing the half top
$balance$: shared variable upon multiple runs, on its value depends the choice of the search
i: shared index upon the multiples runs
Output: the index where key is present in A, return -1 if it is not present.

```
1  if balance ≥ 32 ∨ n ≤ 64 then
2  |   bot ← 0
3  |   top ← n
4  |   GOTO monobound
5  bot ← i
6  top ← 32
7  if key ≥ A[bot] then
8  |   while 1 do
9  |   |   if bot + top ≥ n then
10 |   |   |   top ← n − bot
11 |   |   bot ← bot + top
12 |   |   if key < A[bot] then
13 |   |   |   bot ← bot − top
14 |   |   |   BREAK
15 |   |   top ← top × 2
16 else
17 |   while 1 do
18 |   |   if bot < top then
19 |   |   |   top ← bot
20 |   |   |   bot ← 0
21 |   |   |   BREAK
22 |   |   bot ← bot − top
23 |   |   if key ≥ A[bot] then
24 |   |   |   BREAK
25 |   |   top ← top × 2
26 monobound :
27 while top > 3 do
28 |   mid ← top/2
29 |   if key ≥ A[bot+mid] then
30 |   |   bot ← bot + mid
31 |   top ← top − mid
32 balance ← (i > bot) ? i − bot : bot − i
33 i ← bot
34 while top > 0 do
35 |   top ← top − 1
36 |   if key = A[bot+top] then
37 |   |   return bot + top
38 return −1
```

2.2.9. Boundless Binary Range Search

As we said in the introduction in the ISD algorithm we will use the binary range searches. As an example here is reported the algorithm that finds the left index and the right index in which the target values lies in a sorted array that can contains multiple times the same

value using the boundless binary search. For all the other searches the reasoning is exactly the same and their procedures don't change, so for simplicity, they are not reported.

Algorithm 2.2.9: Boundless Binary Range Search

Input: A: sorted array that can contains multiple times the same elements
n: size of the array A
key: target value to find in A
Data: bot: index starting at the bottom of A
top: index starting at the top of A
mid: index starting outside A
Output: $left_index$: first index of A in which key is present
$right_index$: last index of A in which key is present

```
1  if n = 0 then
2  |  return ⟨−1, −1⟩
3  bot ← 0
4  mid ← n
5  while mid > 1 do
6  |  if key ≥ A[bot + mid/2] then
7  |  |  bot ← bot + mid/2
8  |  |  mid ← mid + 1
9  |  mid ← mid/2
10 if key = A[bot] then
11 |  right_index ← bot
12 else
13 |  return ⟨−1, −1⟩
14 top ← n − 1
15 mid ← n
16 while mid > 1 do
17 |  if key ≤ A[top − mid/2] then
18 |  |  top ← top − mid/2
19 |  |  mid ← mid + 1
20 |  mid ← mid/2
21 if key = A[top] then
22 |  left_index ← top
23 return ⟨left_index, right_index⟩
```

2.3. NextComb and NextColSum algorithms

In this section we are going to see two important algorithms that will be used in the implementation of the ISDs. The first algorithm, NextComb, given two integer k and p computes all the possible combination of size p from 0 to $k-1$. The other, NextColSum, given an array in input and a matrix V, return the total sum of the columns of V indexed by the array. For reusing additions already done in the past an optimization is discussed for this algorithm to speed up the sums.

2.3.1. NextComb

The NextComb algorithm generates all the possible combinations of p integers from 0 to $k-1$. For reaching this we use an array that we call from now on `next_comb`. The `next_comb` has size p and contains the p integers between $first$ and $last$, where usually $first = 0$ and $last = k-1$. An extra variable called li is needed for producing all the combinations: li stores the leftmost index of `next_comb` that has been modified in the last call of the NextComb function. The idea of the procedure is explained in the following.

1. The `next_comb` array is initialized with values from $first$ to $first+p$ where p is the size of the `next_comb` and the variable li is initialized with the value $p-1$ that is the last index of the array. Therefore `next_comb` initially contains the first possible combination allowed.

2. For producing the next combination we add 1 in the position indexed by li of the `next_comb` until this value reaches $last$. li remains the same since the leftmost index modified is the same as the previous call.

3. When the value in the li position reaches $last$, we need to select another position to update on its left since we have started modifying the last value. Therefore, we search for the rightmost index in which the value can be updated by 1 for producing the next combination: we decrement li by 1 and check if the value saved at li can be incremented. The situation in which this value can not be incremented happens when it is the immediate predecessor of the value saved at $li+1$. If we increment it, it will have the same value of the element at $li+1$ that has already have the maximum possible value for that position, resulting in an error since the combination will have two positions with the same value. If it is not the immediate predecessor, we can increment it by 1 and we update all the positions at its right with the values of the integer in the previous position plus 1.

4. Now, we can restart updating the rightmost value in the array for producing the next combination. When we reach the situation where we can not update any integer it means that we have generated all the possible combination of p integers from $first$ to $last$.

Let's see an example for understanding better how it works. Let $p = 3$, $\texttt{next_comb} = [0, 3, 5]$, $first = 0$ and $last = 6$. Therefore initially we have:

$$\texttt{next_comb} = [0, 3, 5] \qquad li = 2$$

For computing the next iteration we see that the element at $\texttt{next_comb}_{[li]}$ can be updated by 1 since it is not equal to $last$ (the while is never entered and the line 9 of Algorithm 2.3.1 is executed) and so we obtain:

$$\texttt{next_comb} = [0, 3, 6] \qquad li = 2$$

Now, since $\texttt{next_comb}_{[li]} = last$, we need to find the new rightmost index that can be updated. We decrement li that becomes $li = 1$ and we check if $\texttt{next_comb}_{[li]}$ is different from the immediate predecessor of $\texttt{next_comb}_{[li+1]}$ (this is done in the checking condition of the while loop of 2.3.1). This is the case since $\texttt{next_comb}_{[1]} \neq 5$ and so, we can update this position and the position at its right (line 9-11 of Algorithm 2.3.1) obtaining the new combination with a new value of li:

$$\texttt{next_comb} = [0, 4, 5] \qquad li = 1$$

Next, we restart updating the value at the position $p - 1 = 2$ for retrieving the next combination. An extra check after computed the correct li index in the while loop is done for understanding if there are no more combinations to produce: this is true when $li = 0$ and in the first position of the `next_comb` the element is equal to the maximum admissible in that position, that is $last - p + 1$ (line 7-8 of 2.3.1).
Thanks to this algorithm, we can compute all the possible combinations in a sequential procedure without using a recursive one.
The complexity of the NextComb procedure described in the Algorithm 2.3.1 having in input an array `next_comb` of size p and $k = last - first$ is:

$$C_{NextComb}(k, p) = \mathcal{O}\left(1 + \frac{p}{k - p + 1}\right)$$

Algorithm 2.3.1: NextComb

Input: `next_comb`: array of integer holding the current combination of size p
$first, last$: minimum and maximum value that can be present in the array

Output: `next_comb`: array holding the new combination of integers
li: integer representing the leftmost index of `next_comb` that has changed
$last_comb$: boolean that is true if the last combination is produced, otherwise false if other combinations are available

```
1  li ← p − 1
2  while li ≠ first ∧ next_comb[li] = last − p + 1 + li do
3  |   li ← li − 1
4  |   if li = −1 then
5  |   |   li ← 0
6  |   |   BREAK
7  if next_comb[0] = last − p + 1 + li then
8  |   return TRUE                        // No more combination to do
9  next_comb[li] ← next_comb[li] + 1
10 for i ← li + 1 to p − 1 do
11 |   next_comb[i] ← next_comb[i−1] + 1
12 return FALSE
```

2.3.2. NextColSum

Now that we have the `next_comb` array that contains all the possible combinations of p integers from $first$ to $last$, we are interested in computing the sum of the columns of a binary matrix V indexed by the `next_comb`. The basic algorithm is trivial but we will see an optimization for speeding up this sum saving a lot of additions. We start with an initial binary vector sum that will be summed to all the columns of V indexed by `next_comb`. For example if we have:

$$\mathbf{sum} = 101 \qquad \texttt{next_comb} = [0, 1, 5] \qquad V = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix}$$

we want to compute $\mathbf{sum} = \mathbf{sum} \oplus \sum_{i \in \texttt{next_comb}} \mathbf{v}_i$ where $\mathbf{v}_i$ is the i-th column of the matrix V. We simply obtain:

$$\mathbf{sum} = 101 \oplus 100 \oplus 101 \oplus 111 = 011$$

This trivial method has a computation complexity of $\mathcal{O}(pr)$ where r is the size of the binary vectors and p the size of `next_comb` since we need to do $p+1$ sums of vector with size r. In the ISDs we will need to compute the columns sum indexed by `next_comb` each time the `next_comb` computes a new combination: we can exploit the fact that when a new combination is produced in the `next_comb`, some indices remain the same and li contains the leftmost index that has changed during the last call for reusing some additions already done. For doing this we use an array of binary vectors, this means we need a matrix S of size $(p-1) \times r$ since a binary vector is an array itself: this matrix holds $p-1$ binary vectors of size r representing intermediate sums between the columns of V indexed by the `next_comb`. We call this matrix the partial sums matrix.
The first binary vector saved in the matrix S is the sum between the inital **sum** and the column of V indexed by the first index of `next_comb`. The generic i-th binary vector in position $S_{[i]}$ is obtained as the sum between the previous vector $S_{[i-1]}$ and the column of V indexed by i in `next_comb`. The matrix is populated using i that goes from 0 to $p-2$ since the size of S is $p-1$ and we start counting from 0. The first iteration for populating the matrix S is the same as the trivial method but the big advantage comes when the `next_comb` update its combination: we know that li is the leftmost index that changed during the NextComb and therefore, we can reuse the intermediate sums saved in S for computing the new one avoiding to restart the procedure. Let's see an example having the following data with the combinations between $first = 0$ and $last = 4$ produced by `next_comb` of size $p = 4$.

$$V = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

$$\textbf{sum} = 101 \quad \texttt{next_comb} = [0,1,2,3] \quad li = 3$$

First, we initialize the partial sums matrix S holding $p-1=3$ binary vectors with the procedure described before. Each row of the matrix S contains a binary vector of size r, in this example $r = 3$. The first binary vector is computed as the sum between the initial **sum** and the first column of V since $\texttt{next_comb}_{[0]} = 0$. The second binary vector is the sum between the first and the second column of V since $\texttt{next_comb}_{[1]} = 1$ and the third follows the same reasoning obtaining:

$$S = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

For computing the wanted sum for this combination we sum the last binary vector $S_{[p-2]}$ with the column of the matrix V indexed by $\texttt{next_comb}_{[p-1]}$. This is the first iteration and we haven't gain any advantage but this is true only for the initialization. From now on we will perform the sums very efficiently: suppose that a new combination is computed obtaining $\texttt{next_comb} = [0, 1, 2, 4]$. The index li hasn't changed since we have modified the last position. Therefore, the partial sums in the matrix S are all still valid: this permits us to compute the new sum with the new combination with only one step summing $S_{[p-2]}$ with the column of V indexed by $\texttt{next_comb}_{[li]}$ that is the new index that changed without redoing the sums for the other indices. Let's see the next iteration that will change the index li. The next call of NextComb will return $\texttt{next_comb} = [0, 1, 3, 4]$ with $li = 2$. Since the index in a position different that the last has changed, the binary vectors starting from li in the matrix S need to be updated since they are invalid. In the example only the binary vector at position $S_{[2]}$ is invalid while the first two are still valid and they will save some computations for computing the final sum. The third binary vector is updated computing the sum between $S_{[li-1]}$ and the column indexed by $\texttt{next_comb}_{[li]}$ obtaining the new partial sums matrix:

$$S = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

The new sum with the new combination can be calculated as before. Using extra memory for storing S we have seen that many sums can be avoided saving a lot of time. The descriptions of the initialization of the partial sums matrix S and the optimized NextColSum procedure are reported in the following algorithms.

Algorithm 2.3.2: Init Partial Sums

Input: V: binary matrix of size $r \times k$, $\boldsymbol{v}_i$ is the i-th column of V
`next_comb`: array of integer holding the current combination of size p
S: partial sums matrix of size $(p-1) \times r$
$\boldsymbol{sum}$: initial binary vector of size r

Output: S: partial sums matrix populated

1 $S_{[0]} \leftarrow \boldsymbol{sum} \oplus \boldsymbol{v}_{\texttt{next_comb}_{[0]}}$
2 $j \leftarrow 1$
3 **while** $j < p - 1$ **do**
4 | $S_{[j]} \leftarrow S_{[j-1]} \oplus \boldsymbol{v}_{\texttt{next_comb}_{[j]}}$

Algorithm 2.3.3: NextColSum Optimized

Input: V: binary matrix of size $r \times k$
`next_comb`: array of integer holding the current combination of size p
li: index holding the leftmost index changed in the last call of NextComb function
S: partial sums matrix of size $(p-1) \times r$
$\boldsymbol{sum}$: initial binary vector of size r

Output: $\boldsymbol{final_sum}$: r size vector resulting from the sum between the initial $\boldsymbol{sum}$ and the columns of V indexed by `next_comb`

```
1 if li ≠ p − 1 then
2 |   if li = 0 then
3 |   |   S[0] ← sum ⊕ v_next_comb[li]
4 |   |   li ← li + 1
5 |   while li ≠ p − 1 do
6 |   |   S[li] ← S[li−1] ⊕ v_next_comb[li]
7 |   |   li ← li + 1
8 final_sum ← S[p−2] ⊕ v_next_comb[li]
9 return final_sum
```

2.4. Sorting algorithms

In the last section of this chapter we are going to analyze two sorting algorithms that have been taken into consideration in the implementation of the ISD algorithms: the Quicksort algorithm designed by Charles Antony Richard Hoare in [16] and the Djbsort designed by Daniel J. Bernstein upon the results in [10]. In the ISD algorithms, we will have lists of pairs made by a binary vector and an array of integers. For applying the binary range searches we first need to sort this list in ascending order based on the value of the binary vectors. Here we present the sorting algorithms applied to an array of integers but they can be easily generalized for working with the binary vectors: instead of having comparison operations between scalars we need to compare binary vectors.

2.4.1. Quicksort

Having an array of integers A we want to sort it in ascending order. The Quicksort algorithm is a a divide-and-conquer algorithm: it starts selecting a pivot element from A and then partition the other elements into two sub-arrays depending if they are greater or less than the pivot chosen. The sub-arrays are then sorted recursively (line 3-4 of Algorithm 2.4.2). Different partition schemes can be used in the algorithm, the one analyzed in this thesis is the Hoare partition. The Hoare partition scheme uses two

indices that start at the ends of the array being partitioned. The indices are moved toward each other until an inversion is detected: this happens when we find a pair of elements, one greater than the pivot and one smaller, that are in the wrong order relative to each other (this work is done in the repeat blocks inside the while in Algorithm 2.4.1). When an inversion is found the two elements pointed by the indices are swapped (line 13 of 2.4.1). The inversion check starts again until the two indices met: when the index that started from the bottom of the array becomes equal or greater than the index that started on the top. In this case the algorithm stops and returns the index that started from the top (line 12 of 2.4.1). The formal description of the Hoare partition scheme and the Quicksort algorithm are reported in 2.4.1 and 2.4.2. In the worst case the complexity of the Quicksort algorithm is $\mathcal{O}(n^2)$ where n is the size of the array to be sorted. This happens when there is a total imbalance: the partition has returned a partition long $n-1$ and one long 0. In the average case the algorithm behaves well with complexity:

$$C_{sort} = \mathcal{O}(n \log(n))$$

2.4.2. Djbsort

The djbosort is a sorting algorithm designed by Daniel J. Bernstein based on the work in [10]. The pseudocode is reported in Algorithm 2.4.3 but further detailes can be found at https://sorting.cr.yp.to/index.html.

The functioning of this sorting algorithm is based on sorting networks. An n-input sorting network is a sequence of instructions that correctly sorts every possible n-entry array $(A_0, A_1, \ldots, A_{n-1})$. Each instruction is a pair (i, j), specifying that the array entries A_i and A_j are replaced by, respectively, $\min\{A_i, A_j\}$ and $\max\{A_i, A_j\}$. Whenever we compare A_i with A_j the subsequent comparisons for the case $A_i < A_j$ are exactly the same as for the case $A_i > A_j$, but with i and j interchanged. This fact in which the comparison is forgotten after it is used to sort $\{A_i, A_j\}$, and that the indices (i, j) are then independent of the input, allows a particularly cheap transformation into a constant-time sorting algorithm: one simply has to perform each min-max computation in constant time. The classic “odd-even merging network” introduced by Batcher [7] in 1968 is used for the Djbsort algorithm.

Algorithm 2.4.1: Hoare Partition Scheme

Input: A: array of integers to be partitioned
$low, high$: indices specifying the range in which the array needs to be partitioned

Data: $pivot$: integer element picked as the first available in the array that guides the partition procedure
i: index starting at the end bottom of the range to be partitioned
j: index starting at the end top of the range to be partitioned

Output: A: array partitioned correctly

```
1  pivot ← A[low]
2  i ← low − 1
3  j ← high + 1
4  while 1 do
5      repeat
6          i ← i + 1
7      until A[i] < pivot
8      repeat
9          j ← j − 1
10     until A[j] > pivot
11     if i ≥ j then
12         return j
13     SWAP(A[i], A[j])
```

Algorithm 2.4.2: Quicksort

Input: A: array of integers to be sorted of size n
$low, high$: indices specifying the range in which the array needs to be sorted, on the first call $low = 0$, $high = n - 1$.

Data: $pivot$: index returned by the partition scheme indicating that the element in this position is in the correct one

Output: A: array between low and high sorted correctly

```
1  if low < high then
2      pivot ← HoarePartitionScheme(A, low, high)
3      Quicksort(A, low, pivot)
4      Quicksort(A, pivot + 1, high)
```

Algorithm 2.4.3: Djbsort

Input: A: array of integers to be sorted size n

Output: A: array sorted in ascending order

```
1  if n < 2 then
2  |   return
3  top ← 1
4  while top < n − top do
5  |   top ← top + 1
6  p ← top
7  while p > 0 do
8  |   i ← 0
9  |   while i < n − p do
10 |   |   if ¬(i ∧ p) then
11 |   |   |   MINMAX(A[i], A[i+p])
12 |   |   i ← i + 1
13 |   i ← 0
14 |   q ← top
15 |   while q > p do
16 |   |   while i < n − q do
17 |   |   |   if ¬(i ∧ p) then
18 |   |   |   |   pi ← A[i+p]
19 |   |   |   |   r ← q
20 |   |   |   |   while r > p do
21 |   |   |   |   |   MINMAX(A[INDEX_OF(A,pi)], A[i+r])
22 |   |   |   |   |   A[i+p] ← pi
23 |   |   |   |   |   r ← r >> 1
24 |   |   |   i ← i + 1
25 |   |   q ← q >> 1
26 |   p ← p >> 1
```

3 | Information Set Decoding algorithms

In Chapter 1 we have seen that the security of the code-based cryptosystems is based on the hardness of solving the Syndrome Decoding Problem or Decoding Random Linear Code: they are equivalent and we will analyze the Syndrome Decoding problem without loss of generality. In this chapter we will study the best known attacks for solving these problems called Information Set Decoding algorithms to understand their working principles and their complexities. All the ISD known in the state of the art have been implemented and analyzed: Prange[24], Lee-Brickell[19], Leon[20], Stern[29], Ball-Collision Decoding[9], Finiasz-Sendrier[15], May-Meurer-Thomae[1], Becker-Joux-May-Meurer[8], Both-May[12] and the Esser-Bellini[13].

3.1. Basic of Information Set Decoding algorithm

Since we consider the Syndrome Decoding problem, the goal of each ISD algorithm is finding a target error $e \in \mathbb{F}_2^n$ having a binary parity-check matrix $H \in \mathbb{F}_2^{r \times n}$ and a syndrome $s \in \mathbb{F}_2^r$, such that $He^T = s$ in less time than an exhaustive search. First, we define two important theorems and then, we define formally what is an information set for explaining after the basic structure of an ISD.

Theorem 3.1. *Let $C(n,k,d)$ be a binary linear code having $H \in \mathbb{F}_2^{r \times n}$ as a binary parity check matrix and let be $\boldsymbol{s} \in \mathbb{F}_2^r$ and $\boldsymbol{e} \in \mathbb{F}_2^n$ a syndrome vector and an error vector, respectively, such that $H\boldsymbol{e}^T = \boldsymbol{s}$. There exists a unique binary vector error $\boldsymbol{e}$ satisfying $\mathtt{HW}(\boldsymbol{e}) = w$ as long as $w \leq \lfloor \frac{d-1}{2} \rfloor$ holds.*

Proof. Let's assume $\exists \boldsymbol{e_1} : \quad H\boldsymbol{e_1}^T = s \wedge \mathtt{HW}(\boldsymbol{e_1}) = w \leq \lfloor \frac{d-1}{2} \rfloor \wedge \boldsymbol{e_1} \neq \boldsymbol{e}$. Then we can write $H\boldsymbol{e_1}^T = \boldsymbol{s} = H\boldsymbol{y}^T = H\boldsymbol{c}^T + H\boldsymbol{e}^T$ obtaining $H\boldsymbol{c}^T = H(\boldsymbol{e_1} - \boldsymbol{e})^T$. So $\boldsymbol{e_1} - \boldsymbol{e}$ is a codeword but it has weight $\leq \lfloor \frac{d-1}{2} \rfloor + \lfloor \frac{d-1}{2} \rfloor \leq d-1$ and this is a contradiction since it is less than the minimum distance d. □

When we solve a syndrome decoding problem having the parity check matrix H of a code C with minimum distance $d > 2w$, this theorem guarantees that the target error found corresponds to the one that was used to encrypt the message. In the case in which $d \leq 2w$ the adversary cannot know if the output of the resolved syndrome decoding problem corresponds to the error vector that was actually used in encryption since there are many error vectors with the same weight satisfying the equation.
The next theorem will help us in formalizing the equivalence between two problems.

Theorem 3.2. *Let be $H\boldsymbol{e}^T = \boldsymbol{s}$ a syndrome decoding problem where $H \in \mathbb{F}_2^{r\times n}$ is a parity check matrix, $\boldsymbol{s} \in \mathbb{F}_2^r$ a syndrome vector and $\boldsymbol{e} \in \mathbb{F}_2^n$ the target error. For any non-singular matrix $U \in \mathbb{F}_2^{r\times r}$ and any permutation matrix $P \in \mathbb{F}_2^{n\times n}$ the two following syndrome decoding problems are equivalent to solve:*

$$H\boldsymbol{e}^T = \boldsymbol{s} \iff \widehat{H}\widehat{\boldsymbol{e}}^T = \widehat{\boldsymbol{s}}$$

where:

$$\widehat{H} = UHP, \quad \widehat{\boldsymbol{s}} = U\boldsymbol{s}^T, \quad \widehat{\boldsymbol{e}} = \boldsymbol{e}P$$

Proof. The proof is simple, it is enough to substitute the values:

$$\widehat{H}\widehat{\boldsymbol{e}}^T = UHPP^T\boldsymbol{e}^T = UH\boldsymbol{e}^T = U\boldsymbol{s}^T = \widehat{\boldsymbol{s}} \iff H\boldsymbol{e}^T = \boldsymbol{s}$$

□

3.1.1. Basic structure of an ISD algorithm

Definition 3.1.1. Given a binary linear code C with a parity check matrix $H \in \mathbb{F}_2^{r\times n}$ we define **IS** as an information set with size k if and only if rank $H_{\mathbf{IS}^*} = |\mathbf{IS}^*| = n - k = r$ where $\mathbf{IS}^* = \{0, \dots, n-1\} \setminus \mathbf{IS}$.

All ISDs share a common structure: they choose an information set **IS** with size k of H that divides the error $\mathbf{e}$ into two parts, $\boldsymbol{e}_{\mathbf{S}^*}$ and $\boldsymbol{e}_{\mathbf{S}}$, where $\boldsymbol{e}_{\mathbf{S}^*}$ are the bits in $\boldsymbol{e}$ indexed by the information set $\mathbf{IS}^*$ while $\boldsymbol{e}_{\mathbf{S}}$ are the ones indexed by **IS**. Then, all the ISD try to guess in the $\boldsymbol{e}_{\mathbf{S}}$ part a certain weight p with different methods depending on the ISD chosen and they try to reconstruct the original error starting from this assumption. Thanks to the Theorem 3.1 we know that the target error is the correct one.
The phase for choosing the information set can be done with the RREF methods described in the Section 2.1 where a binary parity check matrix H is transformed in systematic form obtaining $\widehat{H} = [I_r \quad V]$. From this form we can see that the information set is the part of

the V matrix since on the left an identity matrix of size r is present having that the rank of I_r is equal to r. We can use this transformation for solving the syndrome decoding problem since the problem with the original H and the problem with the systematic form of H, $\widehat{H}$, are equivalent as we have proved in Theorem 3.2. We can see in Figure 3.1 the matrix $\widehat{H}$ with the permuted syndrome, the transformed error and how the error vector is splitted considering the information set after a successfull call to the RREF.

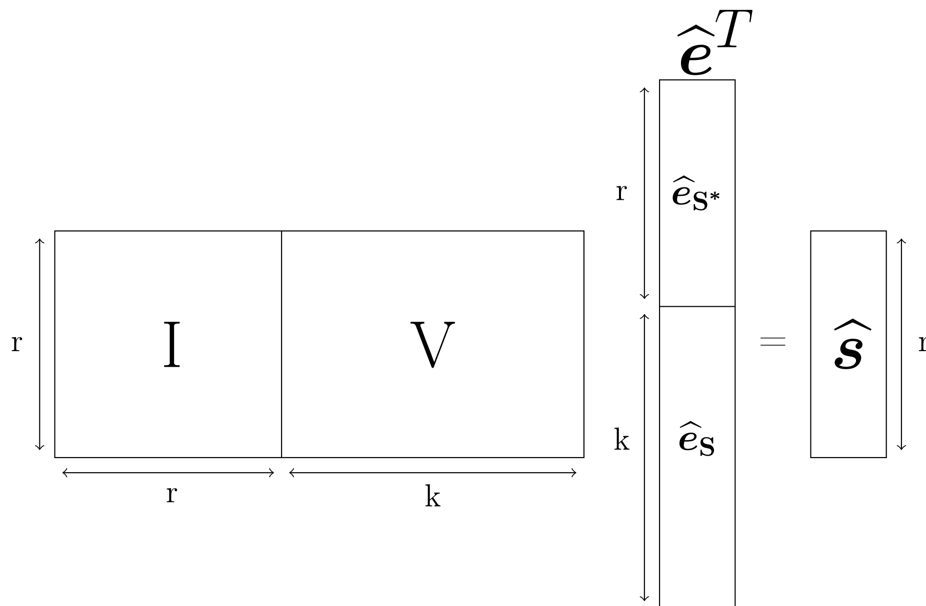


Figure 3.1: Situation with the transformed matrix and vectors after calling the RREF

The last k columns of the matrix V are indexed by the information set $\boldsymbol{IS}$ as also the last k bits of the permuted error $\widehat{\boldsymbol{e}}$. To find the permuted target error we can guess that the last k bits of it, the ones indexed by $\boldsymbol{IS}$, have a certain weight p and then checking if the part indexed by $\boldsymbol{IS}^*$ has weight $w - p$. The guessing part and the consecutive check on the first part of the permuted error depends on which ISD we are using. Therefore, an ISD algorithm is divided in two main step: the computation of a correct systematic form (we call the RREF function until a correct form is found) and the search error part that works with the results returned by the RREF.
From now on, we are going to use the following conventions to indicate sub-parts of matrices and vectors.

Definition 3.1.2. Let $x \in \mathbb{F}_2^n$ a binary vector indexed from 1 to n and a and b two integers respecting $1 \leq a \leq b \leq n$. We indicate as $x_{[a:b]}$ the sub-vector of x holding the bits between a and b.
$x_{[:b]}$ takes the bits between the very first of x and the one indexed by b while $x_{[a:]}$ between

a and the very last bit of x.

The same reasoning applied for matrices.
Using this notation we have that $\widehat{\boldsymbol{e}}_{\mathbf{S}^*} = \widehat{\boldsymbol{e}}_{[1:r]}$, $\widehat{\boldsymbol{e}}_{\mathbf{S}} = \widehat{\boldsymbol{e}}_{[r+1:n]}$, $I = \widehat{H}_{[1:r][1:r]}$ and $V = \widehat{H}_{[1:r][r+1:n]}$. The general reasoning of the guessing part works as follows. After the RREF we can observe that:

$$\widehat{H}\widehat{\boldsymbol{e}}^T = [I \quad V] \cdot [\widehat{\boldsymbol{e}}_{\mathbf{S}^*} \quad \widehat{\boldsymbol{e}}_{\mathbf{S}}]^T = \widehat{\boldsymbol{s}} \Leftrightarrow \widehat{\boldsymbol{e}}_{\mathbf{S}^*}^T + V\widehat{\boldsymbol{e}}_{\mathbf{S}}^T = \widehat{\boldsymbol{s}} \tag{3.1}$$

resulting in:

$$\widehat{\boldsymbol{e}}_{\mathbf{S}^*}^T = \widehat{\boldsymbol{s}} - V\widehat{\boldsymbol{e}}_{\mathbf{S}}^T$$

We can follow a bruteforce technique for guessing p positions set to 1 in $\widehat{\boldsymbol{e}}_{\mathbf{S}}$: the guess is correct if and only if $\mathtt{HW}(\widehat{\boldsymbol{e}}_{\mathbf{S}^*}) = w - \mathtt{HW}(\widehat{\boldsymbol{e}}_{\mathbf{S}}) = w - p$.
When a right guess is found we obtain the permuted error as the concatenation between $\widehat{\boldsymbol{e}}_{\mathbf{S}^*}$ and $\widehat{\boldsymbol{e}}_{\mathbf{S}}$: now the target error of the original problem is computed applying the inverse permutation to the permuted error found $\boldsymbol{e} = \widehat{\boldsymbol{e}}P^{-1}$. Otherwise, if we have tried all the possible guesses without find a correct error, we need to choose another information set calling another time the RREF procedure and restart the guessing procedure. In Algorithm 3.1.1 there is a formalization of the basic structure of an ISD just discussed.

Algorithm 3.1.1: Basic Structure of an ISD algorithm

Input: $\boldsymbol{s} \in \mathbb{Z}_2^r$: syndrome vector
$H \in \mathbb{Z}_2^{r\times n}$:binary parity check matrix
w: the weight of the error vector to be recovered

Output: $\boldsymbol{e} \in \mathbb{Z}_2^n$: error vector to be recovered such that $H\boldsymbol{e}^T = \boldsymbol{s}$, with $\mathtt{wt}(\boldsymbol{e}) = w$

Data: χ: an array of size n containing the indices of the columns of H after being permuted in the RREF
$[I_r \quad V]$: matrix H in systematic form after applying the RREF
$U \in \mathbb{Z}_2^{r\times r}$: matrix representing elementary row operations in the RREF, at first is an identity matrix
$\widehat{\boldsymbol{s}} \in \mathbb{Z}_2^r$: permuted syndrome obtained multiplying the matrix U with $\boldsymbol{s}$

```
1 repeat
2     repeat
3         ⟨[I_r  V], U, χ, rref_error⟩ ← FIND_RREF(H)
4     until rref_error = true
5     ŝ ← PRODUCT_BIT_MATRIX_VECTOR(U, s)
6     ê ← SEARCH_ERROR(V, ŝ)
7 until HAMMING_WEIGHT(ê) = w
8 e ← ERROR_RECONSTRUCTION(ê, χ)
9 return e
```

An interesting thing to notice is that for permuting the matrix H has been used an array containing its indices and not the matrix P previously described. This is the same method we have seen in Section 2.1 discussing the RREF procedures. At the end of the ISD the original error is retrieved applying the permutation saved in the array χ to the permuted error $\widehat{\boldsymbol{e}}$.

3.1.2. Complexity analysis of the basic structure of an ISD algorithm

All the ISD algorithms try to retrieve the error vector repeating a certain number of times an attempt whose average value depends on the success probability of the single attempt itself. Therefore, the complexity of each ISD algorithm can be expressed as the product between the complexity of each attempt, that we denote from now on as c_{iter}, and the average number of attempts. The average number of attempts can be computed as the reciprocal of the success probability of each attempt that we denote as Pr_{succ}. Hence, having the binary code properties $[n, k]$ and the value of the target error's weight w to be found, we can write the complexity of each ISD algorithm as:

$$C_{ISD}(n, r, w) = \frac{c_{iter}}{Pr_{succ}} \tag{3.2}$$

If the conditions of the Theorem 3.1 are not satisfied the complexity of each ISD changes since there are many error vectors with the same weight satysfing the syndrome decoding problem's equation. In the work [5] at pages 5-6 we can find the actual reasoning in this situation ending having the following complexity for doing a message recovery attack:

$$WF_{MRA} = \alpha C_{ISD}(n, k, w) \tag{3.3}$$

where:

$$\alpha = \frac{NPr_{succ}}{1 - (1 - Pr_{succ})^N} \geq 1 \tag{3.4}$$

During this thesis the complexities of the ISD algorithms are studied assuming that the Theorem 3.1 always holds.
Now let's study the complexity of the Algorithm 3.1.1.

Theorem 3.3. *Given the basic structure of the ISD Algorithm 3.1.1 with* $H \in \mathbb{Z}_2^{r \times n}$ *binary parity check matrix,* $\boldsymbol{s} \in \mathbb{Z}_2^r$ *the syndrome vector and* w *the target weight, the*

computational complexity of this algorithm can be computed as:

$$C_{ISD}(n,r,w) = \frac{1}{Pr_{succ}} c_{iter} = \frac{1}{Pr_{succ}}(C_{IS}(n,r) + C_{SEARCH}(n,r,w))$$

where

$$C_{IS}(n,r) = \frac{1}{Pr_{rref}} C_{RREF}(n,r) + r^2 = \frac{1}{\prod_{i=1}^{r}(1-2^{-i})} C_{RREF}(n,r) + r^2$$

Proof. A single iteration cost is composed by two parts: the computation of the RREF and the searching of the error. The computation of a correct systematic form ends when there is a full identity matrix on the left or equivalently when we find an information set. The complexity of this part is indicated as $C_{IS}(n,r)$: since some calls to the RREF procedure can fail to produce a suitable systematic form, we need to consider the success probabilty of finding a correct form, and this is taken into account by the first addend. In a $r \times r$ binary matrix the first row has a probability of $\frac{1}{2^r}$ of being linearly dependent from itself (i.e. row equal to zero); the second row has a probability of $\frac{2}{2^r}$ of being linearly dependent(i.e. zero or equal to the first). Generalizing this procedure we obtain that the r-th row has a probability of $\frac{2^{r-1}}{2^r}$ of being linearly dependent from the previous ones. We want an identity matrix with full rank so all the rows need to be linearly independent from each other and this is happens with probability $Pr_{rref} = \prod_{i=1}^{r}(1 - \frac{2^{i-1}}{2^r}) = \prod_{i=1}^{r}(1 - \frac{1}{2^i})$. This probability is multiplied by the cost $C_{RREF}(n,r)$ that depends on the procedure we want to use for computing the RREF; these complexities can be seen in the Table 2.1. The second addend considers the transformation of the syndrome vector using the matrix U: a matrix vector multiplication took place and since $\boldsymbol{s}$ has size r the total cost of it is r^2. Pr_{succ} is obtained as the number of permuted error vectors with the error-affected positions in line with the hypotheses made by the ISD in analysis, divided by the number of all the possible error vectors, while C_{search} depends on the ISD we choose to use. □

For clarity we introduce these definitions that will be used in all the ISD descriptions.

Definition 3.1.3. We define FIND_RREF as the general algorithm for computing the full systematic form of a matrix H and it can be the standard RREF algorithm 2.1.1, the RREF with reusing pivots optimization 2.1.2 or the M4RI method explained in Section 2.4. The computational complexity of this algorithm is denoted as $C_{IS}(n,r)$ and is the one descripted in Theorem 3.3.

Definition 3.1.4. We define FIND_PARTIAL_RREF as the general algorithm for computing the partial systematic form of a matrix H having in input the extra parameter

ℓ and it can be the standard partial RREF algorithm 2.1.3, the partial RREF optimized 2.1.4 or the M4RI method explained in Section 2.4. The computational complexity of this algorithm is:

$$C_{IS-P}(n,r,\ell) = \frac{1}{Pr_{p-rref}} C_{P-RREF}(n,r,\ell) + r^2 = \frac{1}{\prod_{i=1}^{r-\ell}(1-2^{-i})} C_{P-RREF}(n,r,\ell) + r^2$$

where $C_{P-RREF}(n,r,\ell)$ is reported in Table 2.1 and depends on the method we use.

3.2. Analysis of Information Set Decoding algorithms

In this section we are going to study in details all the ISD algorithm variants in their syndrome decoding formulation reporting the actual computational and space complexities.

3.2.1. Prange

Prange's algorithm [24] was the first variant of ISD designed. Prange is based on the idea of guessing a set k of error-free positions in the target vector $\widehat{\boldsymbol{e}}$ to be found. After it has found a correct RREF computed from the binary parity check matrix in input obtaining $\widehat{H}$, $\widehat{\boldsymbol{s}}$ and the permutation array χ, it guesses that the second part of $\widehat{\boldsymbol{e}}$ long k bits has weight equal to 0. This means that the part of the error indexed by the information set is composed by only zeroes and all the weight of the error is in the first part. In Figure 3.2 we can see the weight distribution for the Prange algorithm.

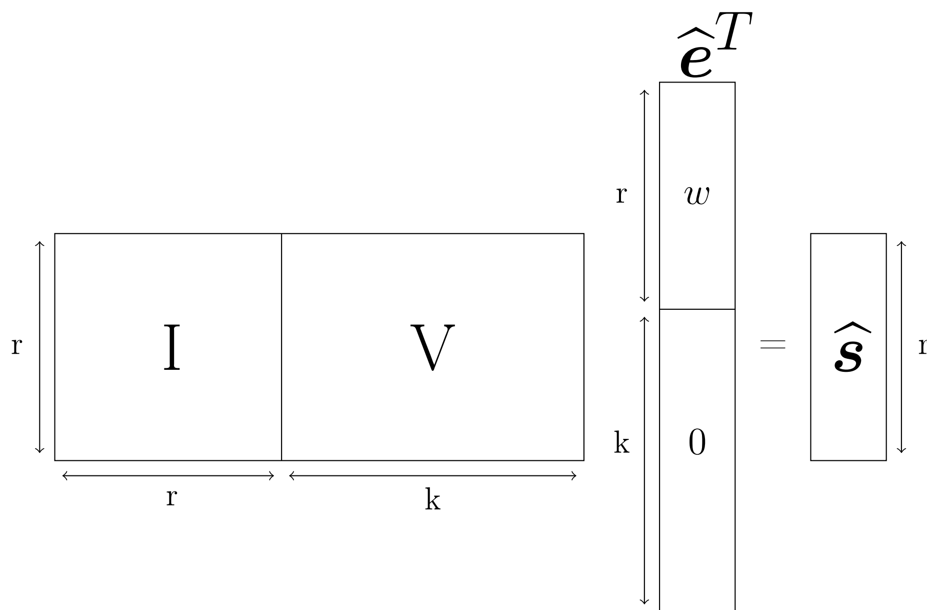


Figure 3.2: Weight distribution in Prange algorithm

From Equation 3.1 we can derive:

$$\widehat{\boldsymbol{e}}_{\mathbf{S}^*}^T = \widehat{\boldsymbol{s}} - V\widehat{\boldsymbol{e}}_{\mathbf{S}}^T = \widehat{\boldsymbol{s}} - V\mathbf{0}^T \Leftrightarrow \widehat{\boldsymbol{e}}_{\mathbf{S}^*}^T = \widehat{\boldsymbol{s}} \tag{3.5}$$

Therefore, Prange simply checks if the permuted syndrome $\widehat{\boldsymbol{s}}$ obtained at the end of the RREF has weight equal to w for building the target error. Until the weight of $\widehat{\boldsymbol{s}}$ is not equal to w, new permutations are picked and new computations of RREF are done. When we find the desired weight on the syndrome we can reconstruct the permuted error as $\widehat{\boldsymbol{e}} = [\widehat{\boldsymbol{s}} \quad \mathbf{0}_{1\times k}]$.
In Algorithm 3.2.1 we can see the detailed description of the Prange algorithm.

Theorem 3.4. *Given an instance of the syndrome decoding problem with $H \in \mathbb{Z}_2^{r\times n}$, $\boldsymbol{s} \in \mathbb{Z}_2^r$ and the target weight w, the time complexity of Prange algorithm 3.2.1 for finding a target error $\boldsymbol{e} \in \mathbb{Z}_2^n$ such that $H\boldsymbol{e}^T = \boldsymbol{s}$ and $\texttt{HW}(\boldsymbol{e}) = w$ is:*

$$C_{ISD}(n, r, w) = \frac{1}{Pr_{succ}} c_{iter} = \frac{\binom{n}{w}}{\binom{r}{w}} C_{IS}(n, r) + n$$

While the spatial complexity is:

$$S_{ISD}(n, r, w) = S_{\text{RREF}}(n, r) + r$$

Proof. The term that dominates the complexity of Prange's algorithm is the one relative to the computation of the RREF $C_{IS}(n, r)$ that is exactly the same complexity we saw in 3.3. The success probability Pr_{succ} in the case of Prange is the division between all the permuted error vectors admissible by the ISD that are $\binom{r}{w}$ (since we want all the weight w in the first r part of the permuted error) and all the possible error vectors that are $\binom{n}{w}$ (all the vectors long n with weight w). The last term is the cost of building the permuted error and it is n since its length is equal to n.
Even the spatial complexity depends only on the computation of the RREF, no extra memory is used for searching the error, therefore we have the parity check matrix in systematic form with size $r \times n$, the matrix U with size $r \times r$, the permutation array with size n included in the term $S_{\text{RREF}}(n, r)$ referring to the complexity in Table 2.2 and then the permuted syndrome of size r reported as the last term. □

Algorithm 3.2.1: Prange algorithm

Input: $\boldsymbol{s} \in \mathbb{Z}_2^r$: syndrome vector
$H \in \mathbb{Z}_2^{r \times n}$:binary parity check matrix
w: the weight of the error vector to be recovered

Output: $\boldsymbol{e} \in \mathbb{Z}_2^n$: error vector to be recovered such that $H\boldsymbol{e}^T = \boldsymbol{s}$, with $\mathtt{wt}(\boldsymbol{e}) = w$

Data: χ: an array of size n containing the indices of the columns of H after being permuted in the RREF
$[I_r \quad V]$: matrix H in systematic form after applying the RREF
$U \in \mathbb{Z}_2^{r \times r}$: matrix representing elementary row operations in the RREF, at first is an identity matrix
$\widehat{\boldsymbol{s}} \in \mathbb{Z}_2^r$: permuted syndrome obtained multiplying the matrix U with $\boldsymbol{s}$

```
1 repeat
2   repeat
3     ⟨[I_r  V], U, χ, rref_error⟩ ← FIND_RREF(H)
4   until rref_error = true
5   ŝ ← PRODUCT_BIT_MATRIX_VECTOR(U, s)
6 until HAMMING_WEIGHT(ŝ) = w
7 ê ← [ŝ  0_{1×k}]
8 e ← ERROR_RECONSTRUCTION(ê, χ)
9 return e
```

3.2.2. Lee-Brickell

The Lee-Brickell algorithm [19] starts with the same computation of Prange retrieving the systematic form of the parity check matrix H, but then improve the Prange's guessing allowing p positions set to 1 in the second part of the permuted error long k. From Equation 3.1 we can see that $\widehat{\boldsymbol{e}}_{\mathbf{S}^*}^T + V\widehat{\boldsymbol{e}}_{\mathbf{S}}^T = \widehat{\boldsymbol{s}}$ where $\widehat{\boldsymbol{e}}_{\mathbf{S}}^T$ is the second part of the permuted errors long k bits with weight p, from which follows that $\widehat{\boldsymbol{e}}_{\mathbf{S}^*}^T = \widehat{\boldsymbol{s}} + V\widehat{\boldsymbol{e}}_{\mathbf{S}}^T$ must have weight equal to $w - p$. In Figure 3.3 we can visualize the situation just described.
Now, since we need to guess p error-affected positions in $\widehat{\boldsymbol{e}}_{\mathbf{S}}^T$, we have to consider all the possible combinations of k size vectors with weight equal to p. Each time we try a new combination of $\widehat{\boldsymbol{e}}_{\mathbf{S}}^T$ with $\mathtt{HW}(\widehat{\boldsymbol{e}}_{\mathbf{S}}^T) = p$ we need to control if $\mathtt{HW}(\widehat{\boldsymbol{s}} + V\widehat{\boldsymbol{e}}_{\mathbf{S}}^T) = w - p$: we try all the possible combinations until the last equality holds. If all the combinations has been tested and no target error is found a new permutation is picked for computing a new RREF and the guessing part is restarted.
In practice, for considering all the possible combinations $\binom{k}{p}$, we use the `next_comb` array described in Section 2.3. Each time we call the NextComb algorithm 2.3.1 the `next_comb` array will contain a new combination. When a new combination is produced we have to compute the vector resulting from $\widehat{\boldsymbol{s}} + V\widehat{\boldsymbol{e}}_{\mathbf{S}}^T$ for checking its weight. For doing this we exploit the procedure described in Section 2.3. using the NextColSumOptimized algorithm

2.3.3. We use the matrix of partial sums S and we initialize it with the Algorithm 2.3.2 using as the initial *sum* the permuted syndrome $\widehat{\boldsymbol{s}}$. Then, we are able to compute the different terms $\widehat{\boldsymbol{s}}+V\widehat{\boldsymbol{e}}_{\mathbf{S}}^T$ efficiently each time the combination in `next_comb` array changes thanks to the NextColSum optimized Algorithm 2.3.3.

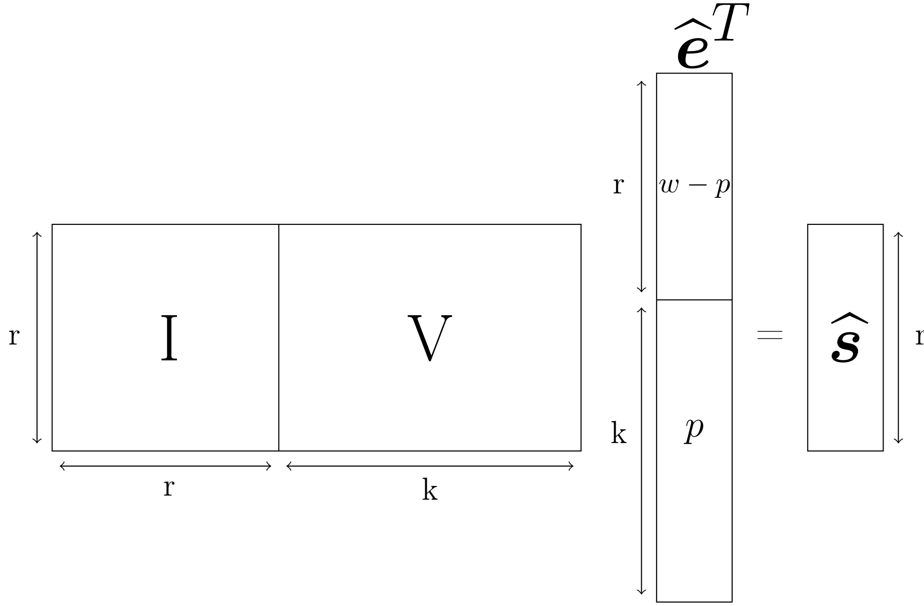


Figure 3.3: Weight distribution in Lee-brickell algorithm

Looking at the Algorithm 3.2.2 we can see that after producing a correct $\widehat{H}$ and $\widehat{\boldsymbol{s}}$ we initialize our `next_comb` array, that we will call α, and our partial sums matrix S. Then we span all the possible combinations $\binom{k}{p}$: at each iteration α holds the index of the p error-affected positions of $\widehat{\boldsymbol{e}}_{\mathbf{S}}^T$ and the vector $\boldsymbol{\sigma} = \widehat{\boldsymbol{s}} + V\widehat{\boldsymbol{e}}_{\mathbf{S}}^T$ is computed. If $\boldsymbol{\sigma}$ has the desired weight a target error is found and it is reconstructed setting the first part of the permuted error $\widehat{\boldsymbol{e}}_{\mathbf{S}*}^T$ equal to $\boldsymbol{\sigma}$ and the second part $\widehat{\boldsymbol{e}}_{\mathbf{S}}^T$ with the bits indexed by α set to 1 and all the others set to 0. We now have a permuted error with the desired weight and we can return the original error applying the permutation.

Theorem 3.5. *Given an instance of the syndrome decoding problem with* $H \in \mathbb{Z}_2^{r\times n}$, $\boldsymbol{s} \in \mathbb{Z}_2^r$ *and the target weight* w*, the time complexity of Lee-Brickell algorithm 3.2.2 for finding a target error* $\boldsymbol{e} \in \mathbb{Z}_2^n$ *such that* $H\boldsymbol{e}^T = \boldsymbol{s}$ *and* $\mathtt{HW}(\boldsymbol{e}) = w$ *is:*

$$C_{ISD}(n,r,w,p) = \frac{1}{Pr_{succ}} c_{iter} =$$

$$= \frac{\binom{n}{w}}{\binom{k}{p}\binom{r}{w-p}} \left(C_{IS}(n,r) + \binom{k}{p} (C_{NextComb}(k,p) + C_{NextColSum}(k,p,r)) \right) + p$$

While the spatial complexity is:

$$S_{ISD}(n, r, w, p) = S_{\text{RREF}}(n, r) + \mathcal{O}\left((p-1)r + p + r\right)$$

Algorithm 3.2.2: Lee-Brickell algorithm

Input: $\boldsymbol{s} \in \mathbb{Z}_2^r$: syndrome vector
$H \in \mathbb{Z}_2^{r \times n}$: binary parity check matrix
w: the weight of the error vector to be recovered

Output: $\boldsymbol{e} \in \mathbb{Z}_2^n$: error vector to be recovered such that $H\boldsymbol{e}^T = \boldsymbol{s}$, with $\texttt{wt}(\boldsymbol{e}) = w$

Data: χ: an array of size n containing the indices of the columns of H after being permuted in the RREF
p: the weight of the last k bits of $\widehat{\boldsymbol{e}}$, $0 \leq p \leq w$
$[I_r \quad V]$: matrix H in systematic form after applying the RREF
$U \in \mathbb{Z}_2^{r \times r}$: matrix representing elementary row operations in the RREF, at first is an identity matrix
$\widehat{\boldsymbol{s}} \in \mathbb{Z}_2^r$: permuted syndrome obtained multiplying the matrix U with $\boldsymbol{s}$
α: an array of size p containing indices in $\{0, \ldots, k-1\}$ holding the possible combinations $\binom{k}{p}$
$S \in \mathbb{Z}_2^{p-1 \times r}$: partial sums matrix that contains $p-1$ vectors obtained by the sum between $\widehat{\boldsymbol{s}}$ and the columns of the matrix V
$\boldsymbol{\sigma} \in \mathbb{Z}_2^r$: vector containing the sum between $\widehat{\boldsymbol{s}}$ and the columns of V indexed by the α array.

```
1  repeat
2      repeat
3          ⟨[I_r  V], U, χ, rref_error⟩ ← FIND_RREF(H)
4      until rref_error = true
5      ŝ ← PRODUCT_BIT_MATRIX_VECTOR(U, s)
6      α ← INIT_COMBINATION_ARRAY(p, 0)
7      INIT_PARTIAL_SUMS(S, V, ŝ, α)
8      for j ← 0 to (k choose p) do
9          σ ← NEXT_COL_SUM_OPTIMIZED(S, V, ŝ, α)
10         if HAMMING_WEIGHT(σ) = w − p then
11             ê ← [σ  0_{1×k}]
12             foreach i ∈ α do
13                 ê_{i+r} ← 1
14             e ← ERROR_RECONSTRUCTION(ê, χ)
15             return e
16         NEXT_COMB(α, 0, k − 1)
17 until HW(e) = w
```

Proof. The success probability in the Lee-Brickell algorithm is the division between the admissibile errors satisfying the hypothesis of the ISD that are $\binom{k}{p}\binom{r}{w-p}$ (since the first r part of the error needs to have weight $w-p$ while the remaining k part p) and all the

possible error vectors with weight w $\binom{n}{w}$. An iteration spans all the combinations $\binom{k}{p}$ and for each of them the NextColSum and the NextComb procedures are invoked. The last term is for composing the k part of the error setting p bits resulting in a p term.
The spatial complexity accounts for H, U and χ in the term refering to the RREF as seen in Prange 3.4. The others accounts for the matrix of the partial sums S of size $(p-1) \times r$, the `next_comb` array α with size p and the vector σ of size r. □

3.2.3. Leon

The Leon algorithm [20] improves the Lee-Brickell algorithm assuming that the contribution to the value of the first ℓ bits of the syndrome $\widehat{\boldsymbol{s}}$, $\widehat{\boldsymbol{s}}_{\mathbf{up}}$, comes only from columns in V: this means that there is a run long ℓ bits of zeroes in the first part of $\widehat{\boldsymbol{e}}_{\mathbf{S}^*}$. The situation is showed in Figure 3.4.

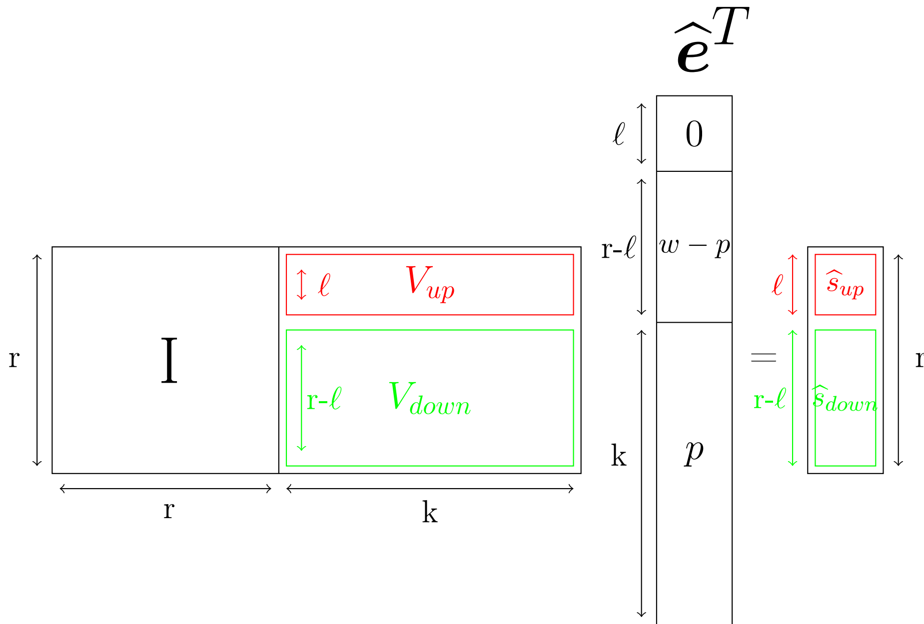


Figure 3.4: Weight distribution in Leon algorithm

Leon algorithm lowers down the success probability of an iteration since it discards all the possible target errors that do not have a starting run of ℓ zeroes. Nevertheless, we can perform a preliminary check computing the sum between the columns of the sub-matrix V_{up} indexed by the `next_comb` array α (holding the p positions of the second part of the error) and the first part of the permuted syndrome composed by ℓ bits, $\widehat{\boldsymbol{s}}_{\mathbf{up}}$. If the resulting vector has weight 0 we can proceed to check if the remained part has weight $w-p$, otherwise, we try another combination and retest only the ℓ part. So, initially, instead of working with columns long r we can work with columns long ℓ: we perform

more iterations than before but one iteration is more faster thanks to the preliminary check that work with smaller dimensions. The choice of the parameter ℓ needs to be optimized following this trade-off. Formally, from the Equation 3.1, we derive:

$$V_{\text{up}}\widehat{\boldsymbol{e}}_{\mathbf{S}} + \widehat{\boldsymbol{s}}_{\mathbf{up}} = \widehat{\boldsymbol{e}}_{\mathbf{S}^*[1:\ell]} = \mathbf{0}_{1\times\ell} \qquad V_{\text{down}}\widehat{\boldsymbol{e}}_{\mathbf{S}} + \widehat{\boldsymbol{s}}_{\mathbf{down}} = \widehat{\boldsymbol{e}}_{\mathbf{S}^*[\ell+1:r]}$$

Hence, we can do the folllowing preliminary check:

$$V_{\text{up}}\widehat{\boldsymbol{e}}_{\mathbf{S}} = \widehat{\boldsymbol{s}}_{\mathbf{up}} \quad \Leftrightarrow \quad \mathtt{HW}(V_{\text{up}}\widehat{\boldsymbol{e}}_{\mathbf{S}} + \widehat{\boldsymbol{s}}_{\mathbf{up}}) = 0 \tag{3.6}$$

If the test is not passed we pick another combination otherwise we can compute:

$$\widehat{\boldsymbol{e}}_{\mathbf{S}^*[\ell+1:r]} = V_{\text{down}}\widehat{\boldsymbol{e}}_{\mathbf{S}} + \widehat{\boldsymbol{s}}_{\mathbf{down}}$$

checking if it has weight equal to $w - p$.

Theorem 3.6. *Given an instance of the syndrome decoding problem with $H \in \mathbb{Z}_2^{r\times n}$, $\boldsymbol{s} \in \mathbb{Z}_2^r$ and the target weight w, the time complexity of Leon algorithm 3.2.3 for finding a target error $\boldsymbol{e} \in \mathbb{Z}_2^n$ such that $H\boldsymbol{e}^T = \boldsymbol{s}$ and $\mathtt{HW}(\boldsymbol{e}) = w$ is:*

$$C_{ISD}(n, r, w, p, \ell) = \frac{1}{Pr_{succ}} c_{iter} =$$

$$= \frac{\binom{n}{w}}{\binom{k}{p}\binom{r-\ell}{w-p}} \left(C_{IS}(n, r) + \binom{k}{p} (C_{NextComb}(k, p) + C_{NextColSum}(k, p, \ell) + \frac{\binom{k}{p}}{2^\ell} p(r - \ell) \right) + p$$

While the spatial complexity is:

$$S_{ISD}(n, r, w, p, \ell) = S_{\text{RREF}}(n, r) + \mathcal{O}\left((p - 1)\ell + p + \ell + (r - \ell)\right)$$

Proof. The success probability in the Leon algorithm is the division between the admissibile errors satisfying the hypothesis of the ISD that are $\binom{k}{p}\binom{r-\ell}{w-p}$ (since the first part of the error needs to have the first ℓ bits set to 0 and the remained $r - \ell$ with weight $w - p$) and all the possible error vectors with weight w $\binom{n}{w}$. An iteration spans all the combinations $\binom{k}{p}$ and for each of them the NextColSum and the NextComb procedures are invoked. If we find a vector obtained by NextColSum with weight 0 we need to sum $p + 1$ vectors of size $r - \ell$ obtaining the complexity $p(r - \ell)$. These sums are done with probability of $\frac{\binom{k}{p}}{2^\ell}$ since all the possible vectors $\widehat{\boldsymbol{s}}_{\mathbf{up}}$ are 2^ℓ and only $\binom{k}{p}$ attempts hitting the correct one are made. The last p term is for reconstructing the error from α. In the spatial complexity the terms different from the one used for the RREF accounts for the

Algorithm 3.2.3: Leon algorithm

Input: $\boldsymbol{s} \in \mathbb{Z}_2^r$: syndrome vector
$H \in \mathbb{Z}_2^{r\times n}$: binary parity check matrix
w: the weight of the error vector to be recovered

Output: $\boldsymbol{e} \in \mathbb{Z}_2^n$: error vector to be recovered such that $H\boldsymbol{e}^T = \boldsymbol{s}$, with $\mathtt{wt}(\boldsymbol{e}) = w$

Data: χ: an array of size n containing the indices of the columns of H after being permuted in the RREF
p: the weight of the last k bits of $\widehat{\boldsymbol{e}}$, $0 \le p \le w$
ℓ: length of the run of zeroes at the starting bits of the error $\widehat{\boldsymbol{e}}$
$U \in \mathbb{Z}_2^{r\times r}$: matrix representing elementary row operations in the RREF, at first is an identity matrix
$\widehat{\boldsymbol{s}} \in \mathbb{Z}_2^r$: permuted syndrome equal to the syndrome of $\boldsymbol{e}$ through $[I_r \quad V]$, $\widehat{\boldsymbol{s}} = \begin{bmatrix} \widehat{\boldsymbol{s}}_{\mathbf{up}} \\ \widehat{\boldsymbol{s}}_{\mathbf{down}} \end{bmatrix}$, with $\widehat{\boldsymbol{s}}_{\mathbf{up}} = \widehat{\boldsymbol{s}}_{[1:\ell]}$ and $\widehat{\boldsymbol{s}}_{\mathbf{down}} = \widehat{\boldsymbol{s}}_{[\ell+1:r]}$
$V \in \mathbb{Z}_2^{r\times k}$: matrix $V = \begin{bmatrix} V_{\text{up}} \\ V_{\text{down}} \end{bmatrix}$ with $V_{\text{up}} = V_{[1:\ell][:]}$ and $V_{\text{down}} = V_{[\ell+1:r][:]}$, where $\boldsymbol{v}_{\mathbf{up}\ \mathbf{i}}$ and $\boldsymbol{v}_{\mathbf{down}\ \mathbf{i}}$ are the columns of matrix V_{up} and V_{down} indexed by i
α: an array of size p containing indices in $\{0, \dots, k-1\}$ used to compute the sum over the columns of V
$S_{up} \in \mathbb{Z}_2^{p-1\times\ell}$: partial sums up matrix that contains $p-1$ vectors obtained by the sum between $\widehat{\boldsymbol{s}}_{\mathbf{up}}$ and the columns of the matrix V_{up}
$\boldsymbol{\sigma}_{\mathbf{up}} \in \mathbb{Z}_2^{\ell}$: vector containing the sum between $\widehat{\boldsymbol{s}}_{\mathbf{up}}$ and the columns of V_{up} indexed by the α array
$\boldsymbol{\sigma}_{\mathbf{down}} \in \mathbb{Z}_2^{r-\ell}$: vector containing the sum between $\widehat{\boldsymbol{s}}_{\mathbf{down}}$ and the columns of V_{down} indexed by the α array

```
1  repeat
2  |  repeat
3  |  |  ⟨[I_r  V], U, χ, rref_error⟩ ← FIND_RREF(H)
4  |  until rref_error = true
5  |  ŝ ← PRODUCT_BIT_MATRIX_VECTOR(U, s)
6  |  α ← INIT_COMBINATION_ARRAY(p, 0)
7  |  INIT_PARTIAL_SUMS(S_up, V_up, ŝ_up, α)
8  |  for j ← 0 to (k choose p) do
9  |  |  σ_up ← NEXT_COL_SUM_OPTIMIZED(S_up, V_up, ŝ_up, α)
10 |  |  if HAMMING_WEIGHT(σ_up) = 0 then
11 |  |  |  foreach i ∈ α do
12 |  |  |  |  σ_down ← σ_down ⊕ v_down i
13 |  |  |  σ_down ← σ_down ⊕ ŝ_down
14 |  |  |  if HAMMING_WEIGHT(σ_down) = w − p then
15 |  |  |  |  ê ← [0_{1×ℓ}  σ_down  0_{1×k}]
16 |  |  |  |  foreach i ∈ α do
17 |  |  |  |  |  ê_{i+r} ← 1
18 |  |  |  |  e ← ERROR_RECONSTRUCTION(ê, χ)
19 |  |  |  |  return e
20 |  |  NEXT_COMB(α, 0, k − 1)
21 until HW(e) = w
```

matrix of the partial sums S of size $(p-1) \times \ell$, the `next_comb` array α with size p, the vector $\boldsymbol{\sigma}_{\mathbf{up}}$ of size ℓ and the vector $\boldsymbol{\sigma}_{\mathbf{down}}$ of size $r-\ell$. □

3.2.4. Stern

Stern's algorithm [29] improves Leon's ISD by employing a meet-in-the-middle strategy for finding which set of size p, containing ℓ bit portions of columns of V, adds up to the first ℓ bits of the syndrome, $\widehat{\boldsymbol{s}}_{\mathbf{up}}$. The part of the permuted error with weight p, $\widehat{\boldsymbol{e}}_{\mathbf{S}}$, is splitted into two binary vectors with weight $\frac{p}{2}$:

$$\widehat{\boldsymbol{e}}_{\mathbf{1}} = \widehat{\boldsymbol{e}}_{\mathbf{S}[1:\frac{k}{2}]} \qquad \mathtt{HW}(\widehat{\boldsymbol{e}}_{\mathbf{1}}) = \lceil \frac{p}{2} \rceil \qquad \widehat{\boldsymbol{e}}_{\mathbf{2}} = \widehat{\boldsymbol{e}}_{\mathbf{S}[\frac{k}{2}+1:k]} \qquad \mathtt{HW}(\widehat{\boldsymbol{e}}_{\mathbf{2}}) = \lfloor \frac{p}{2} \rfloor \tag{3.7}$$

In this case we will use two `next_comb` arrays: α_1 of size $\lceil \frac{p}{2} \rceil$ holds the error-affected positions of $\widehat{\boldsymbol{e}}_{\mathbf{1}}$ and α_2 of size $\lfloor \frac{p}{2} \rfloor$ holds the error-affected positions of $\widehat{\boldsymbol{e}}_{\mathbf{2}}$. Starting from the Equation 3.6 of Leon we can write:

$$V_{\text{up}} \widehat{\boldsymbol{e}}_{\mathbf{S}}^T = \widehat{\boldsymbol{s}}_{\mathbf{up}} \quad \Leftrightarrow \quad \widehat{\boldsymbol{s}}_{\mathbf{up}} = V_{\text{up}1} \widehat{\boldsymbol{e}}_{\mathbf{1}}^T + V_{\text{up}2} \widehat{\boldsymbol{e}}_{\mathbf{2}}^T = \sum_{i \in \alpha_1} \boldsymbol{v}_{\mathbf{up\ i}} + \sum_{j \in \alpha_2} \boldsymbol{v}_{\mathbf{up\ j}} \tag{3.8}$$

where $\boldsymbol{v}_{\mathbf{up\ i}}$ is the i-th column of V_{up}. From this, we can precompute the value of $\widehat{\boldsymbol{s}}_{\mathbf{up}} + \sum_{i \in \alpha_1} \boldsymbol{v}_{\mathbf{up\ i}}$ for all the possible $\binom{k/2}{p/2}$ choices of α_1 and store them into a list or into an hash table θ, saving for each resulting vector the corresponding indices of α_1 used for computing it. Then, we enumerate all the possible combinations $\binom{k/2}{p/2}$ of α_2 computing for each one of these the vector $\boldsymbol{\sigma}_{\mathbf{up2}} = \sum_{j \in \alpha_2} \boldsymbol{v}_{\mathbf{up\ j}}$ and checking if this vector is present in θ. If $\boldsymbol{\sigma}_{\mathbf{up2}}$ is inside θ we have found a candidate pair (α_1, α_2) for which $\widehat{\boldsymbol{s}}_{\mathbf{up}} = \sum_{i \in \alpha_1 \cup \alpha_2} \boldsymbol{v}_{\mathbf{up\ i}}$ holds and so, we can proceed to check if $\widehat{\boldsymbol{s}}_{\mathbf{down}} = \sum_{i \in \alpha_1 \cup \alpha_2} \boldsymbol{v}_{\mathbf{down\ i}}$.

This strategy reduces the cost of computing an iteration quadratically at the price of increasing the number of iterations. Moreover, here we need a significant amount of memory to store the $\binom{k/2}{p/2}$ precomputed values. The data structures taken in consideration in the implementation of the algorithm are two: a list of pairs made by a binary vector long ℓ and an array of indices long $\lceil \frac{p}{2} \rceil$ and an hashtable holding a binary vector long ℓ, an array of indices long $\lceil \frac{p}{2} \rceil$ and the pointer to the next element of the hash table having the same hash code. If the list is used, when we have to find if a certain target vector is present in the list, we can apply a binary range search based on the methods seen in Section 2.2. Before doing the search the list must be sorted with one of the algorithms seen in Section 2.4.

Definition 3.2.1. We define as FIND_COLLISION the procedure that given the list θ,

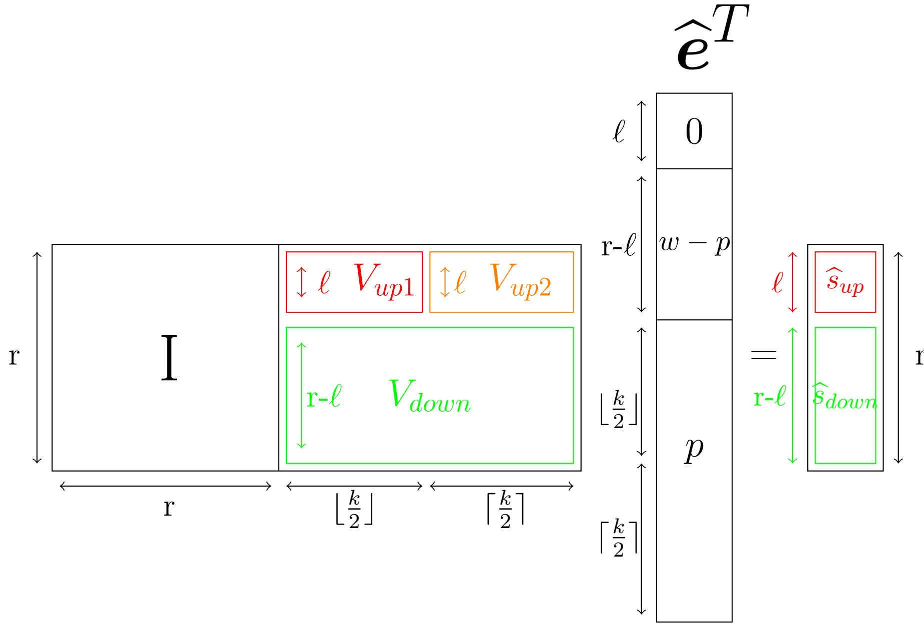


Figure 3.5: Weight distribution in Stern algorithm

composed of pairs made by a binary vector $\boldsymbol{\sigma}_{\mathbf{up1}}$ long ℓ and an array of indices α_1, and a target vector $\boldsymbol{\sigma}_{\mathbf{up2}}$ to find in θ, returns the left index and the right index such that all the binary vectors saved in the list between the left and the right index are equal to $\boldsymbol{\sigma}_{\mathbf{up2}}$. If the target vector is not present, -1 for both the indices is returned. The complexity of this procedure considering n the size of the list θ is $C_{\text{FindColl}}(n) = \mathcal{O}(2log(n))$ since we need to apply two times the binary searches, one time for finding the right index and the other for the left.

Definition 3.2.2. We define SORT as the procedure that given the list θ, composed of pairs made by a binary vector $\boldsymbol{\sigma}_{\mathbf{up1}}$ long ℓ and an array of indices α_1, sorts the list based on the value of the binary vector $\boldsymbol{\sigma}_{\mathbf{up1}}$ in ascending order. This can be done using the Quicksort or the Djbsort seen in Section 2.4.

We can see in Algorithm 3.2.4 the procedure of the Stern ISD using the list as data structure. Otherwise, if the hash table is used, the sorting and the binary range searches are not necessary. For finding a target vector inside the hashtable it has been used a lookup procedure that returns the index of the hash table in which the target vector has been found. The chaining technique has been implemented for managing duplicates and collisions, therefore, when an index is returned from the lookup we span all the chained list starting at this index. If multiple vectors that are equal are inserted in the hash table, with different array of indices, we can find them at the same index of the hashtable following the chaining list. It may happen that different vectors could be placed in the

same index of the hash table: in this case while we are scanning the chained list we can find invalid elements, the ones with the binary vector different from the target. So, we need to manage these collisions checking always if the current element in analysis has the correct binary vector. In Algorithm 3.2.5 the procedure of the Stern ISD using the hashtable as data structure is reported. At line 19 we can see the check just described for handling the collisions.

Theorem 3.7. *Given an instance of the syndrome decoding problem with $H \in \mathbb{Z}_2^{r \times n}$, $\boldsymbol{s} \in \mathbb{Z}_2^r$ and the target weight w, the computational complexity of Stern algorithm 3.2.4 for finding a target error $\boldsymbol{e} \in \mathbb{Z}_2^n$ such that $H\boldsymbol{e}^T = \boldsymbol{s}$ and $\mathtt{HW}(\boldsymbol{e}) = w$ is:*

$$C_{\text{ISD}}(n, r, w, p, \ell) = \frac{1}{\text{Pr}_{\text{succ}}} c_{\text{iter}} = \frac{\binom{n}{w}}{\binom{k/2}{p/2}^2 \binom{r-\ell}{w-p}} \Bigg(C_{\text{IS}}(n, r) + \binom{k/2}{p/2} \Big(2C_{\text{NextComb}}(\frac{k}{2}, \frac{p}{2}) + 2C_{\text{NextColSum}}(\frac{k}{2}, \frac{p}{2}, \ell) + C_{\text{FindColl}}(\binom{k/2}{p/2}) + \frac{\binom{k/2}{p/2}}{2^\ell} p(r - \ell) \Big) \Bigg) + C_{\text{sort}}(\binom{k/2}{p/2}, \ell) + p$$

While the spatial complexity is:

$$S_{ISD}(n, r, w, p, \ell) = S_{\text{RREF}}(r, n) + \mathcal{O}\Bigg(\binom{k/2}{p/2} (\frac{p}{2} \log_2 (\frac{k}{2}) + \ell) \Bigg)$$

Proof. The success probability in the Stern algorithm is the division between the admissibile errors satisfying the hypothesis of the ISD that are $\binom{k/2}{p/2}^2 \binom{r-\ell}{w-p}$ and all the possible error vectors with weight w $\binom{n}{w}$. An iteration spans two times all the combinations $\binom{k/2}{p/2}$ and for each of them the NextColSum and the NextComb procedures are invoked: the first time we build the list θ and the second time we compute the target vectors for finding collisions. In one iteration we call the find collision procedure $\binom{k/2}{p/2}$ times, one for each target vector computed from α_2. When we find a collision we need to sum $p+1$ vectors of size $r - \ell$ obtaining the complexity $p(r - \ell)$. These sums are done with probability of $\frac{\binom{k/2}{p/2}}{2^\ell}$ since all the possible vectors $\widehat{\boldsymbol{s}}_{\mathbf{up}}$ are 2^ℓ and only $\binom{k/2}{p/2}$ attempts hitting the correct one are made. The sorting algorithm for ordering the list θ is called only once during an iteration and the last p term is for reconstructing the error from α. In the spatial complexity we have reported only the significant terms. The first is the one relative to the computation of the RREF. Then, we have to store a list θ made by $\binom{k/2}{p/2}$ elements. Each element takes ℓ bits for storing the binary vector and $\frac{p}{2} \log_2 (\frac{k}{2})$ bits for storing the array of indices. In conclusion, the space required by the list is $\binom{k/2}{p/2}(\frac{p}{2} \log_2 (\frac{k}{2}) + \ell)$ bits. □

Algorithm 3.2.4: Stern algorithm using lists for finding collisions

Input: $\boldsymbol{s} \in \mathbb{Z}_2^r$: syndrome vector
$H \in \mathbb{Z}_2^{r\times n}$: binary parity check matrix
w: the weight of the error vector to be recovered

Output: $\boldsymbol{e} \in \mathbb{Z}_2^n$: error vector to be recovered such that $H\boldsymbol{e}^T = \boldsymbol{s}$, with $\mathtt{wt}(\boldsymbol{e}) = w$

Data: χ: an array of size n holding the indices of the columns of H being permuted in the RREF
p: the weight of the last k bits of $\widehat{\boldsymbol{e}}$, $0 \leq p \leq w$
ℓ: length of the run of zeroes at the starting bits of the error $\widehat{\boldsymbol{e}}$
$U \in \mathbb{Z}_2^{r\times r}$: matrix representing elementary row operations done in the RREF
$\widehat{\boldsymbol{s}} \in \mathbb{Z}_2^r$: permuted syndrome equal to the syndrome of $\boldsymbol{e}$ through $[I_r \quad V]$,$\widehat{\boldsymbol{s}} = \begin{bmatrix}\widehat{\boldsymbol{s}}_{\mathbf{up}}\\ \widehat{\boldsymbol{s}}_{\mathbf{down}}\end{bmatrix}$, with $\widehat{\boldsymbol{s}}_{\mathbf{up}} = \widehat{\boldsymbol{s}}_{[\mathbf{1}:\boldsymbol{\ell}]}$ and $\widehat{\boldsymbol{s}}_{\mathbf{down}} = \widehat{\boldsymbol{s}}_{[\boldsymbol{\ell}+\mathbf{1}:\boldsymbol{r}]}$
$V \in \mathbb{Z}_2^{r\times k}$: matrix $V = \begin{bmatrix}V_{\text{up}}\\ V_{\text{down}}\end{bmatrix}$ with $V_{\text{up}} = V_{[1:\ell][:]}$ and $V_{\text{down}} = V_{[\ell+1:r][:]}$, where $\boldsymbol{v}_{\mathbf{up\ i}}$ and $\boldsymbol{v}_{\mathbf{down\ i}}$ are the columns of matrix V_{up} and V_{down} indexed by i
α_1, α_2: arrays of sizes respectively $\lceil \frac{p}{2} \rceil$ and $\lfloor \frac{p}{2} \rfloor$ containing indices in $\{0, \ldots, \lfloor \frac{k}{2} \rfloor - 1\}$ and in $\{\lfloor \frac{k}{2} \rfloor, \ldots, k-1\}$
$S_{up1} \in \mathbb{Z}_2^{\lceil \frac{p}{2} \rceil -1\times \ell}, S_{up2} \in \mathbb{Z}_2^{\lfloor \frac{p}{2} \rfloor -1\times \ell}$: partial sums matrices containing vectors obtained as the sum between $\widehat{\boldsymbol{s}}_{\mathbf{up}}$ and the columns of the matrix V_{up} indexed by α_1 and α_2
$\boldsymbol{\sigma}_{\mathbf{up1}}, \boldsymbol{\sigma}_{\mathbf{up2}} \in \mathbb{Z}_2^{\ell}$
$\boldsymbol{\sigma}_{\mathbf{down}} \in \mathbb{Z}_2^{r-\ell}$: vector containing the sum between $\widehat{\boldsymbol{s}}_{\mathbf{down}}$ and the columns of V_{down} indexed by $\alpha_1 \cup \alpha_2$
θ: list containing pairs made by indices taken from α_1 and by $\boldsymbol{\sigma}_{\boldsymbol{up1}}$ vector,it has size equal to $\binom{k/2}{p/2}$ and initially it is empty

```
1  repeat
2    repeat
3      ⟨[I_r  V], U, χ, rref_error⟩ ← FIND_RREF(H)
4    until rref_error = true
5    ŝ ← PRODUCT_BIT_MATRIX_VECTOR(U, s)
6    α_1 ← INIT_COMBINATION_ARRAY(⌈p/2⌉, 0)
7    INIT_PARTIAL_SUMS(S_up1, V_up, ŝ_up, α_1)
8    for j ← 0 to binom(k/2, p/2) do
9      σ_up1 ← NEXT_COL_SUM_OPTIMIZED(S_up1, V_up, ŝ_up, α_1)
10     θ ← θ ∪ ⟨α_1, σ_up1⟩
11     NEXT_COMB(α_1, 0, ⌊k/2⌋ − 1)
12   SORT(θ)
13   α_2 ← INIT_COMBINATION_ARRAY(⌊p/2⌋, ⌊k/2⌋)
14   INIT_PARTIAL_SUMS(S_up2, V_up, 0_{1×ℓ}, α_2)
15   for j ← 0 to binom(k/2, p/2) do
16     σ_up2 ← NEXT_COL_SUM_OPTIMIZED(S_up2, V_up, 0_{1×ℓ}, α_2)
17     ⟨left, right⟩ ← FIND_COLLISION(σ_up2, θ)
18     if left ≠ −1 ∧ right ≠ −1 then
19       foreach ⟨α_1, σ_up1⟩ in θ_[left:right] do
20         σ_down ← σ_down ⊕ ŝ_down
21         foreach i in α_1 ∪ α_2 do
22           σ_down ← σ_down ⊕ v_down i
23         if HAMMING_WEIGHT(σ_down) = w − p then
24           ê ← [0_{1×ℓ}  σ_down  0_{1×k}]
25           foreach i ∈ α_1 ∪ α_2 do
26             ê_{i+r} ← 1
27           e ← ERROR_RECONSTRUCTION(ê, χ)
28           return e
29      NEXT_COMB(α_2, ⌊k/2⌋, k − 1)
30 until HW(e) = w
```

Algorithm 3.2.5: Stern algorithm using hash table for finding collisions

Input: $\boldsymbol{s} \in \mathbb{Z}_2^r$: syndrome vector
$H \in \mathbb{Z}_2^{r\times n}$:binary parity check matrix
w: the weight of the error vector to be recovered

Output: $\boldsymbol{e} \in \mathbb{Z}_2^n$: error vector to be recovered such that $H\boldsymbol{e}^T = \boldsymbol{s}$, with $\mathtt{wt}(\boldsymbol{e}) = w$

Data: χ: an array of size n holding the indices of the columns of H being permuted in the RREF
p: the weight of the last k bits of $\widehat{\boldsymbol{e}}$, $0 \le p \le w$
ℓ: length of the run of zeroes at the starting bits of the error $\widehat{\boldsymbol{e}}$
$U \in \mathbb{Z}_2^{r\times r}$: matrix representing elementary row operations done in the RREF
$\widehat{\boldsymbol{s}} \in \mathbb{Z}_2^r$: permuted syndrome equal to the syndrome of $\boldsymbol{e}$ through $[I_r \quad V]$,$\widehat{\boldsymbol{s}} = \begin{bmatrix}\widehat{\boldsymbol{s}}_{\mathbf{up}}\\ \widehat{\boldsymbol{s}}_{\mathbf{down}}\end{bmatrix}$, with $\widehat{\boldsymbol{s}}_{\mathbf{up}} = \widehat{\boldsymbol{s}}_{[\mathbf{1}:\boldsymbol{\ell}]}$ and $\widehat{\boldsymbol{s}}_{\mathbf{down}} = \widehat{\boldsymbol{s}}_{[\boldsymbol{\ell}+\mathbf{1}:\boldsymbol{r}]}$
$V \in \mathbb{Z}_2^{r\times k}$: matrix $V = \begin{bmatrix}V_{\text{up}}\\ V_{\text{down}}\end{bmatrix}$ with $V_{\text{up}} = V_{[1:\ell][:]}$ and $V_{\text{down}} = V_{[\ell+1:r][:]}$, where $\boldsymbol{v}_{\mathbf{up\ i}}$ and $\boldsymbol{v}_{\mathbf{down\ i}}$ are the columns of matrix V_{up} and V_{down} indexed by i
α_1, α_2: arrays of sizes respectively $\lceil \frac{p}{2} \rceil$ and $\lfloor \frac{p}{2} \rfloor$ containing indices in $\{0, \ldots, \lfloor \frac{k}{2} \rfloor - 1\}$ and in $\{\lfloor \frac{k}{2} \rfloor, \ldots, k-1\}$
$S_{up1} \in \mathbb{Z}_2^{\lceil \frac{p}{2} \rceil - 1\times\ell}, S_{up2} \in \mathbb{Z}_2^{\lfloor \frac{p}{2} \rfloor - 1\times\ell}$: partial sums matrices containing vectors obtained as the sum between $\widehat{\boldsymbol{s}}_{\mathbf{up}}$ and the columns of the matrix V_{up} indexed by α_1 and α_2
$\boldsymbol{\sigma}_{\mathbf{up1}}, \boldsymbol{\sigma}_{\mathbf{up2}} \in \mathbb{Z}_2^{\ell}$
$\boldsymbol{\sigma}_{\mathbf{down}} \in \mathbb{Z}_2^{r-\ell}$: vector containing the sum between $\widehat{\boldsymbol{s}}_{\mathbf{down}}$ and the columns of V_{down} indexed by $\alpha_1 \cup \alpha_2$
θ: hash table containing pairs made by indices taken from α_1 and $\boldsymbol{\sigma}_{\boldsymbol{up1}}$ vector, initially is empty.

```
1  repeat
2      repeat
3          ⟨[I_r  V], U, χ, rref_error⟩ ← FIND_RREF(H)
4      until rref_error = true
5      ŝ ← PRODUCT_BIT_MATRIX_VECTOR(U, s)
6      α_1 ← INIT_COMBINATION_ARRAY(⌈p/2⌉, 0)
7      INIT_PARTIAL_SUMS(S_up1, V_up, ŝ_up, α_1)
8      for j ← 0 to binom(k/2, p/2) do
9          σ_up1 ← NEXT_COL_SUM_OPTIMIZED(S_up1, V_up, ŝ_up, α_1)
10         HASH_TABLE_INSERT(θ, α_1, σ_up1)
11         NEXT_COMB(α_1, 0, ⌊k/2⌋ − 1)
12     α_2 ← INIT_COMBINATION_ARRAY(⌊p/2⌋, ⌊k/2⌋)
13     INIT_PARTIAL_SUMS(S_up2, V_up, 0_{1×ℓ}, α_2)
14     for j ← 0 to binom(k/2, p/2) do
15         σ_up2 ← NEXT_COL_SUM_OPTIMIZED(S_up2, V_up, 0_{1×ℓ}, α_2)
16         index ← HASH_TABLE_LOOKUP(θ, σ_up2)
17         if index ≠ −1 then
18             foreach ⟨α_1, σ_up1⟩ in θ_index do
19                 if σ_up1 = σ_up2 then
20                     σ_down ← σ_down ⊕ ŝ_down
21                     foreach i in α_1 ∪ α_2 do
22                         σ_down ← σ_down ⊕ v_down i
23                     if HAMMING_WEIGHT(σ_down) = w − p then
24                         ê ← [0_{1×ℓ}  σ_down  0_{1×k}]
25                         foreach i ∈ α_1 ∪ α_2 do
26                             ê_{i+r} ← 1
27                         e ← ERROR_RECONSTRUCTION(ê, χ)
28                         return e
29         NEXT_COMB(α_2, ⌊k/2⌋, k − 1)
30 until HW(e) = w
```

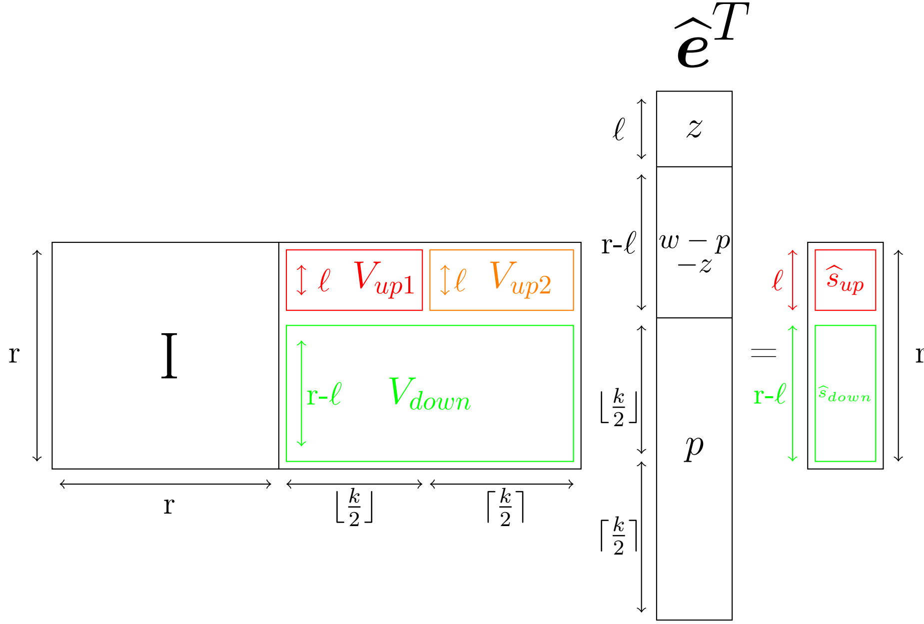


Figure 3.6: Weight distribution in Ball-Collision Decoding algorithm

3.2.5. Ball-Collision Decoding

The Ball-Collision Decoding [9] algorithm is a generalitazion of the Stern algorithm presented in [9]. The authors notice that is unlikely to have the ℓ starting part of the permuted error with weight equal to 0, therefore, they decide to allow even in that position, a weight z. Like in the Stern's algorithm we split the p weight binary error vector in two parts with weights $\frac{p}{2}$ and apply a meet-in-the-middle strategy. Moreover, in the Ball-Collision Decoding algorithm, we also divide the z part of the error into two binary vectors with weights $\frac{z}{2}$. The p part of the error satisfies the already seen relation 3.7 in Stern but here, we need to consider even the z part having:

$$\widehat{\boldsymbol{e}}_{\boldsymbol{z_1}} = \widehat{\boldsymbol{e}}_{\mathbf{S^*}[1:\frac{\ell}{2}]} \qquad \mathtt{HW}(\widehat{\boldsymbol{e}}_{\boldsymbol{z_1}}) = \lceil \frac{z}{2} \rceil \qquad \widehat{\boldsymbol{e}}_{\boldsymbol{z_2}} = \widehat{\boldsymbol{e}}_{\mathbf{S^*}[\frac{\ell}{2}+1:\ell]} \qquad \mathtt{HW}(\widehat{\boldsymbol{e}}_{\boldsymbol{z_2}}) = \lfloor \frac{z}{2} \rfloor$$

We have the same α_1 and α_2 arrays of the Stern algorithm that work on the p part but here we need also other two arrays that work on the z part of the error. We call β_1 and β_2 the `next_comb` arrays of sizes respectively $\lceil \frac{z}{2} \rceil$ and $\lfloor \frac{z}{2} \rfloor$ containing the indices of $\widehat{\boldsymbol{e}}_{\boldsymbol{z_1}}$ and $\widehat{\boldsymbol{e}}_{\boldsymbol{z_2}}$. Formally, the algorithm exploits the following equation:

$$V_{\text{up}} \widehat{\boldsymbol{e}}_{\mathbf{S}}^T + \widehat{\boldsymbol{s}}_{\mathbf{up}} = \widehat{\boldsymbol{e}}_{\mathbf{S^*[1:\boldsymbol{\ell}]}}^T$$

$$\widehat{\boldsymbol{s}}_{\mathbf{up}} + V_{\text{up1}} \widehat{\boldsymbol{e}}_{\mathbf{1}}^T + V_{\text{up2}} \widehat{\boldsymbol{e}}_{\mathbf{2}}^T = [\widehat{\boldsymbol{e}}_{\boldsymbol{z_1}} \quad \widehat{\boldsymbol{e}}_{\boldsymbol{z_2}}]^T$$

$$\widehat{\boldsymbol{s}}_{\mathbf{up}} + V_{\text{up1}} \widehat{\boldsymbol{e}}_{\mathbf{1}}^T + [\widehat{\boldsymbol{e}}_{\boldsymbol{z_1}} \quad \mathbf{0}_{1\times\ell/2}]^T = V_{\text{up2}} \widehat{\boldsymbol{e}}_{\mathbf{2}}^T + [\mathbf{0}_{1\times\ell/2} \quad \widehat{\boldsymbol{e}}_{\boldsymbol{z_2}}]^T$$

From the latter we can precompute the following value:

$$\boldsymbol{\sigma}_{\mathbf{up1}} = \widehat{\boldsymbol{s}}_{\mathbf{up}} + V_{\text{up1}}\widehat{\boldsymbol{e}}_{\mathbf{1}}^{T} + [\widehat{\boldsymbol{e}}_{\boldsymbol{z_1}} \quad \mathbf{0}_{1\times\ell/2}]^{T}$$

for all the possible $\binom{k/2}{p/2}$ choices of α_1 and all the possible $\binom{l/2}{z/2}$ choices of β_1 and store the resulting vector $\boldsymbol{\sigma}_{\mathbf{up1}}$ together with the corresponding indices of α_1 and β_1 in a list θ. Then enumerating all the possible $\binom{k/2}{p/2}$ choices of α_2 and all the possible $\binom{l/2}{z/2}$ choices of β_2 we compute:

$$\boldsymbol{\sigma}_{\mathbf{up2}} = V_{\text{up2}}\widehat{\boldsymbol{e}}_{\mathbf{2}}^{T} + [\mathbf{0}_{1\times\ell/2} \quad \widehat{\boldsymbol{e}}_{\boldsymbol{z_2}}]^{T}$$

and we try to find this vector in the saved list θ. If a collision is found we have found a candidate quadruple $(\alpha_1, \beta_1, \alpha_2, \beta_2)$ for which we can control if $\mathtt{HW}(V_{\text{down}}\widehat{\boldsymbol{e}}_{\mathbf{S}}^{T} + \widehat{\boldsymbol{s}}_{\mathbf{down}}) = w - p - z$, otherwise we try other combinations. Here the data structure is composed by triples holding a binary vector $\boldsymbol{\sigma}_{\mathbf{up1}}$ long ℓ, an array of indices of size $\lceil \frac{p}{2} \rceil$ and an array of indices of size $\lceil \frac{z}{2} \rceil$. For simplicity we present the algorithm with the list as data structure but it has implemented even with the hash table: the procedure described in 3.2.6 can be transformed in an algorithm using the hash table like we have done in the Stern. The sorting function is not present and the find collision routine is substituted by the lookup on the hash table with the check on the collisions: the rest is exactly the same.

Theorem 3.8. *Given an instance of the syndrome decoding problem with $H \in \mathbb{Z}_2^{r\times n}$, $\boldsymbol{s} \in \mathbb{Z}_2^r$ and the target weight w, the time complexity of Ball-Collision Decoding algorithm 3.2.6 for finding a target error $\boldsymbol{e} \in \mathbb{Z}_2^n$ such that $H\boldsymbol{e}^T = \boldsymbol{s}$ and $\mathtt{HW}(\boldsymbol{e}) = w$ is:*

$$\begin{aligned} C_{\text{ISD}}(n,r,w,p,\ell,z) &= \frac{1}{\Pr_{\text{succ}}} c_{\text{iter}} = \frac{\binom{n}{w}}{\binom{k/2}{p/2}^2 \binom{\ell/2}{z/2}^2 \binom{r-\ell}{w-p-z}} \Bigg(C_{\text{IS}}(n,r) + \\ &\binom{k/2}{p/2} \Bigg(2C_{\text{NextComb}}(\frac{k}{2}, \frac{p}{2}) + 2C_{\text{NextColSum}}(\frac{k}{2}, \frac{p}{2}, \ell) + \\ &\binom{l/2}{z/2} \big(2C_{\text{NextComb}}(\frac{\ell}{2}, \frac{z}{2}) + z + C_{\text{FindColl}}(\binom{k/2}{p/2}\binom{\ell/2}{z/2}) + \\ &\frac{\binom{k/2}{p/2}\binom{\ell/2}{z/2}}{2^\ell} p(r-\ell)\big)\Bigg)\Bigg) + C_{\text{sort}}(\binom{k/2}{p/2}\binom{\ell/2}{z/2}, \ell) + p + z \end{aligned}$$

While the spatial complexity is:

$$S_{ISD}(n,r,w,p,\ell,z) = S_{\text{RREF}}(r,n) + \mathcal{O}\Bigg(\binom{k/2}{p/2}\binom{\ell/2}{z/2}(\frac{p}{2}\log_2(\frac{k}{2}) + \frac{z}{2}\log_2(\frac{\ell}{2}) + \ell)\Bigg)$$

Algorithm 3.2.6: Ball-Collision algorithm using lists for finding collisions

Input: $\boldsymbol{s} \in \mathbb{Z}_2^r$: syndrome vector
$H \in \mathbb{Z}_2^{r\times n}$: binary parity check matrix
w: the weight of the error vector to be recovered

Output: $\boldsymbol{e} \in \mathbb{Z}_2^n$: error vector to be recovered such that $H\boldsymbol{e}^T = \boldsymbol{s}$, with $\mathtt{wt}(\boldsymbol{e}) = w$

Data: χ: an array of size n containing the indices of the columns of H after being permuted in the RREF
p: the weight of the last k bits of $\widehat{\boldsymbol{e}}$, $0 \le p \le w$
ℓ: length of the run of the starting bits of the error $\widehat{\boldsymbol{e}}$
z: the weight of ℓ run of bits
$U \in \mathbb{Z}_2^{r\times r}$: matrix representing elementary row operations in the RREF, at first is an identity matrix
$\widehat{\boldsymbol{s}} \in \mathbb{Z}_2^r$: permuted syndrome equal to the syndrome of $\boldsymbol{e}$ through $[I_r \quad V]$, $\widehat{\boldsymbol{s}} = \begin{bmatrix} \widehat{\boldsymbol{s}}_{\mathbf{up}} \\ \widehat{\boldsymbol{s}}_{\mathbf{down}} \end{bmatrix}$, with $\widehat{\boldsymbol{s}}_{\mathbf{up}} = \widehat{\boldsymbol{s}}_{[\mathbf{1:\ell}]}$ and $\widehat{\boldsymbol{s}}_{\mathbf{down}} = \widehat{\boldsymbol{s}}_{[\boldsymbol{\ell}+\mathbf{1:r}]}$
$V \in \mathbb{Z}_2^{r\times k}$: matrix $V = \begin{bmatrix} V_{\text{up}} \\ V_{\text{down}} \end{bmatrix}$ with $V_{\text{up}} = V_{[1:\ell][:]}$ and $V_{\text{down}} = V_{[\ell+1:r][:]}$, where $\boldsymbol{v}_{\mathbf{up}\ \mathbf{i}}$ and $\boldsymbol{v}_{\mathbf{down}\ \mathbf{i}}$ are the columns of matrix V_{up} and V_{down} indexed by i
α_1, α_2: arrays of sizes respectively $\lceil \frac{p}{2} \rceil$ and $\lfloor \frac{p}{2} \rfloor$ containing indices in $\{0, \dots, \lfloor \frac{k}{2} \rfloor - 1\}$ and in $\{\lfloor \frac{k}{2} \rfloor, \dots, k - 1\}$
β_1, β_2: arrays of sizes respectively $\lceil \frac{z}{2} \rceil$ and $\lfloor \frac{z}{2} \rfloor$ containing indices in $\{0, \dots, \lfloor \frac{\ell}{2} \rfloor - 1\}$ and in $\{\lfloor \frac{\ell}{2} \rfloor, \dots, \ell - 1\}$
$S_{up1} \in \mathbb{Z}_2^{\lceil \frac{p}{2} \rceil - 1 \times \ell}$, $S_{up2} \in \mathbb{Z}_2^{\lfloor \frac{p}{2} \rfloor - 1 \times \ell}$: partial sums matrices containing vectors obtained as the sum between $\widehat{\boldsymbol{s}}_{\mathbf{up}}$ and the columns of the matrix V_{up} indexed by α_1 and α_2
$\boldsymbol{\sigma}_{\mathbf{up1}}, \boldsymbol{\sigma}_{\mathbf{up2}} \in \mathbb{Z}_2^{\ell}$
$\boldsymbol{\sigma}_{\mathbf{down}} \in \mathbb{Z}_2^{r-\ell}$: vector containing the sum between $\widehat{\boldsymbol{s}}_{\mathbf{down}}$ and the columns of V_{down} indexed by $\alpha_1 \cup \alpha_2$
θ: list containing triples made by indices taken from α_1, indices taken from β_1 and $\boldsymbol{\sigma}_{\boldsymbol{up1}}$ vector, initially is empty.

```
1  repeat
2      repeat
3          ⟨[I_r  V], U, χ, rref_error⟩ ← FIND_RREF(H)
4      until rref_error = true
5      ŝ ← PRODUCT_BIT_MATRIX_VECTOR(U, s)
6      α_1 ← INIT_COMBINATION_ARRAY(⌈p/2⌉, 0)
7      INIT_PARTIAL_SUMS(S_up1, V_up, ŝ_up, α_1)
8      for j ← 0 to binom(k/2, p/2) do
9          σ_up1 ← NEXT_COL_SUM_OPTIMIZED(S_up1, V_up, ŝ_up, α_1)
10         β_1 ← INIT_COMBINATION_ARRAY(⌈z/2⌉, 0)
11         for h ← 0 to binom(l/2, z/2) do
12             foreach index in β_1 do
13                 FLIP_BIT(σ_up1, index)
14             θ ← θ ∪ ⟨α_1, β_1, σ_up1⟩
15             NEXT_COMB(β_1, 0, ⌊ℓ/2⌋ − 1)
16         NEXT_COMB(α_1, 0, ⌊k/2⌋ − 1)
17     SORT(θ)
18     α_2 ← INIT_COMBINATION_ARRAY(⌊p/2⌋, ⌊k/2⌋)
19     INIT_PARTIAL_SUMS(S_up2, V_up, 0_{1×ℓ}, α_2)
20     for j ← 0 to binom(k/2, p/2) do
21         σ_up2 ← NEXT_COL_SUM_OPTIMIZED(S_up2, V_up, 0_{1×ℓ}, α_2)
22         β_2 ← INIT_COMBINATION_ARRAY(⌊z/2⌋, ⌊ℓ/2⌋)
23         for h ← 0 to binom(ℓ/2, z/2) do
24             foreach index in β_2 do
25                 FLIP_BIT(σ_up2, index)
26             ⟨left, right⟩ ← FIND_COLLISION(σ_up2, θ)
27             if left ≠ −1 ∧ right ≠ −1 then
28                 foreach ⟨α_1, β_1, σ_up1⟩ in θ_[left:right] do
29                     σ_down ← ŝ_down
30                     foreach i in α_1 ∪ α_2 do
31                         σ_down ← σ_down ⊕ v_down i
32                     if HAMMING_WEIGHT(σ_down) = w − p − z then
33                         ê ← [0_{1×ℓ}  σ_down  0_{1×k}]
34                         foreach i ∈ β_1 ∪ β_2 do
35                             ê_i ← 1
36                         foreach i ∈ α_1 ∪ α_2 do
37                             ê_{i+r} ← 1
38                         e ← ERROR_RECONSTRUCTION(ê, χ)
39                         return e
40             NEXT_COMB(β_2, ⌊ℓ/2⌋, ℓ − 1)
41         NEXT_COMB(α_2, ⌊k/2⌋, k − 1)
42 until HW(e) = w
```

Proof. The success probability in the BCD algorithm is the division between the admissibile errors satisfying the hypothesis of the ISD that are $\binom{k/2}{p/2}^2\binom{\ell/2}{z/2}^2\binom{r-\ell}{w-p}$ and all the

possible error vectors with weight w $\binom{n}{w}$. The loop at lines 8-16 builds the list θ spanning all the possible combinations $\binom{k/2}{p/2}$: for each combination all $\binom{\ell/2}{z/2}$ possible ones are considered. Hence, for one combination in $\binom{k/2}{p/2}$ we call the NextComb and the NextColSum and then we call $\binom{\ell/2}{z/2}$ times the NextComb and we flip $z/2$ bits on the vector. Then, for computing the vectors for finding collisions in θ, we have the same passages already described: one call to NextComb and NextColSum, $\binom{\ell/2}{z/2}$ calls to NextComb and $z/2$ bit flip that adds to the previous one resulting in the z linear term we can see in the formula. In one iteration we call the find collision procedure $\binom{k/2}{p/2}\binom{\ell/2}{z/2}$ times, one for each target vector computed from α_2 and β_2. When we find a collision we need to sum $p+1$ vectors of size $r-\ell$ obtaining the complexity $p(r-\ell)$. These sums are done with probability of $\frac{\binom{k/2}{p/2}\binom{\ell/2}{z/2}}{2^\ell}$ since all the possible vectors $\widehat{\boldsymbol{s}}_{\mathbf{up}}$ are 2^ℓ and only $\binom{k/2}{p/2}\binom{\ell/2}{z/2}$ attempts hitting the correct one are made. The sorting algorithm for ordering the list θ is called only once during an iteration and the last p term is for reconstructing the error from α while the z term from β. In the spatial complexity we report only the significant terms. The first is the one relative to the computation of the RREF. Then we have to store a list θ made by $\binom{k/2}{p/2}\binom{\ell/2}{z/2}$ elements. Each element takes ℓ bits for storing the binary vector, $\frac{p}{2}\log_2\left(\frac{k}{2}\right)$ bits for storing the array of indices of size $p/2$ and $\frac{z}{2}\log_2\left(\frac{\ell}{2}\right)$ bits for storing the array of indices of size $z/2$. In conclusion, the space required by the list is $\binom{k/2}{p/2}\binom{\ell/2}{z/2}\left(\frac{p}{2}\log_2\left(\frac{k}{2}\right)+\frac{z}{2}\log_2\left(\frac{\ell}{2}\right)+\ell\right)$ bits. □

3.2.6. Finiasz-Sendrier

The Finiasz-Sendrier algorithm [15] improves the Stern's algorithm removing the ℓ window of zeroes in the permuted error and moving this ℓ region in the part of the error where we need to guess p error bits. Since the p positions to be guessed are picked among the last $k+\ell$ position of the error vector and not among the last k as in the previous ISDs, we need to have only an identity matrix of size $(r-\ell)\times(r-\ell)$ on the upper leftmost portion of $\widehat{H}$ obtaining $\widehat{H}=\begin{bmatrix} I_{r-\ell} & V_{\text{up}} \\ \mathbf{0}_{(r-\ell)\times\ell} & V_{\text{down}} \end{bmatrix}$. The first part of the algorithm is dedicated not to produce a full systematic form but a partial systematic form: this matrix can be computed with the standard partial RREF seen in Section 2.1.3, the partial RREF optimized seen in Section 2.1.4 or the M4RI method seen in Section 2.1.5. After obtaining the partial systematic form with V of size $r\times(k+\ell)$ we split the p weight binary error vector into two vectors of weight $\frac{p}{2}$ applying the same meet-in-the-middle strategy seen before. Formally we start from these relations where we indicate $\widehat{\boldsymbol{e}}_{\mathbf{S}}=\widehat{\boldsymbol{e}}_{[r-\ell+1:n]}$:

$$\widehat{\boldsymbol{e}}_{\mathbf{1}}=\widehat{\boldsymbol{e}}_{\mathbf{S}[1:\frac{k+\ell}{2}]} \qquad \texttt{HW}(\widehat{\boldsymbol{e}}_{\mathbf{1}})=\lceil\frac{p}{2}\rceil \qquad \widehat{\boldsymbol{e}}_{\mathbf{2}}=\widehat{\boldsymbol{e}}_{\mathbf{S}[\frac{k+\ell}{2}+1:k]} \qquad \texttt{HW}(\widehat{\boldsymbol{e}}_{\mathbf{2}})=\lfloor\frac{p}{2}\rfloor$$

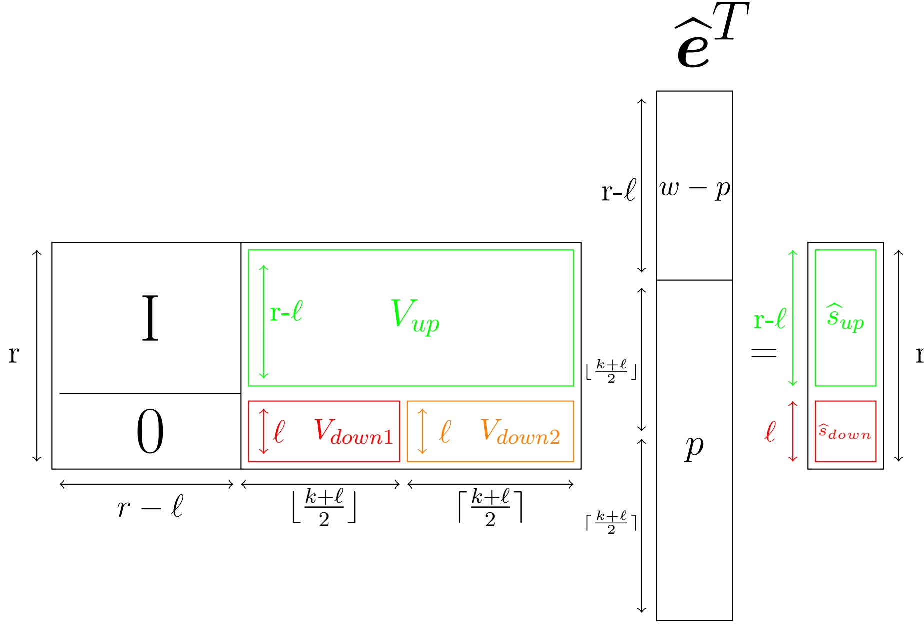


Figure 3.7: Weight distribution in Finiasz-Sendrier algorithm

and we can derive:

$$V_{\text{down}}\widehat{\boldsymbol{e}}_{\mathbf{S}}^T = \widehat{\boldsymbol{s}}_{\mathbf{down}} \quad \Leftrightarrow \quad \widehat{\boldsymbol{s}}_{\mathbf{down}} = V_{\text{down1}}\widehat{\boldsymbol{e}}_{\mathbf{1}}^T + V_{\text{down2}}\widehat{\boldsymbol{e}}_{\mathbf{2}}^T = \sum_{i\in\alpha_1} \boldsymbol{v}_{\mathbf{down\ i}} + \sum_{j\in\alpha_2} \boldsymbol{v}_{\mathbf{down\ j}} \tag{3.9}$$

where α_1 and α_2 are the `next_comb` arrays of sizes respectively $\lceil\frac{p}{2}\rceil$ and $\lfloor\frac{p}{2}\rfloor$ holding the error-affected positions in $\widehat{\boldsymbol{e}}_{\mathbf{1}}$ and $\widehat{\boldsymbol{e}}_{\mathbf{2}}$. For all the possible combinations $\binom{(k+\ell)/2}{p/2}$ in α_1 we compute:

$$\boldsymbol{\sigma}_{\mathbf{down1}} = \widehat{\boldsymbol{s}}_{\mathbf{down}} + V_{\text{down1}}\widehat{\boldsymbol{e}}_{\mathbf{1}}^T$$

saving in the list θ the binary vector $\boldsymbol{\sigma}_{\mathbf{down1}}$ with the corresponding indices of α_1. Then for all the possible combinations $\binom{(k+\ell)/2}{p/2}$ in α_2 we compute:

$$\boldsymbol{\sigma}_{\mathbf{down2}} = V_{\text{down2}}\widehat{\boldsymbol{e}}_{\mathbf{2}}^T$$

trying to find collisions in θ. If we find a collision we can control if $\texttt{HW}(\boldsymbol{\sigma}_{\mathbf{up}} = \widehat{\boldsymbol{s}}_{\mathbf{up}} + V_{\text{up}}\widehat{\boldsymbol{e}}_{\mathbf{S}}^T) = w-p$. If this holds we can reconstruct the target error otherwise we try other combinations. As before, the ISD is analyzed using a list as collision structure but the algorithm has been implemented even with the hash table and it can be derived with the same reasoning followed in the Ball-Collision Decoding algorithm.

Theorem 3.9. *Given an instance of the syndrome decoding problem with* $H \in \mathbb{Z}_2^{r\times n}$,

Algorithm 3.2.7: Finiasz-Sendrier algorithm using lists for finding collisions

Input: $\boldsymbol{s} \in \mathbb{Z}_2^r$: syndrome vector
$H \in \mathbb{Z}_2^{r\times n}$: binary parity check matrix
w: the weight of the error vector to be recovered
ℓ: algorithmic parameter such as $0 \le \ell \le r - w + p$

Output: $\boldsymbol{e} \in \mathbb{Z}_2^n$: error vector to be recovered such that $H\boldsymbol{e}^T = \boldsymbol{s}$, with $\mathtt{wt}(\boldsymbol{e}) = w$

Data: χ: an array of size n containing the indices of the columns of H after being permuted in the RREF
p: the weight of the last $k + \ell$ bits of $\widehat{\boldsymbol{e}}$, $0 \le p \le w$
$U \in \mathbb{Z}_2^{r\times r}$: matrix representing elementary row operations in the RREF, at first is an identity matrix
$\widehat{\boldsymbol{s}} \in \mathbb{Z}_2^r$: permuted syndrome equal to the syndrome of $\boldsymbol{e}$ through $[I_r \quad V]$, $\widehat{\boldsymbol{s}} = \begin{bmatrix}\widehat{\boldsymbol{s}}_{\mathbf{up}}\\ \widehat{\boldsymbol{s}}_{\mathbf{down}}\end{bmatrix}$, with $\widehat{\boldsymbol{s}}_{\mathbf{up}} = \widehat{\boldsymbol{s}}_{[\mathbf{1}:\boldsymbol{r}-\boldsymbol{\ell}]}$ and $\widehat{\boldsymbol{s}}_{\mathbf{down}} = \widehat{\boldsymbol{s}}_{[\boldsymbol{r}-\boldsymbol{\ell}+\mathbf{1}:\boldsymbol{r}]}$
$V \in \mathbb{Z}_2^{r\times(k+\ell)}$: matrix $V = \begin{bmatrix}V_{\text{up}}\\ V_{\text{down}}\end{bmatrix}$ with $V_{\text{up}} = V_{[1:r-\ell][:]}$ and $V_{\text{down}} = V_{[r-\ell+1:r][:]}$, where $\boldsymbol{v}_{\mathbf{up}\ \mathbf{i}}$ and $\boldsymbol{v}_{\mathbf{down}\ \mathbf{i}}$ are the columns of matrix V_{up} and V_{down} indexed by i
α_1, α_2: arrays of sizes respectively $\lceil \frac{p}{2} \rceil$ and $\lfloor \frac{p}{2} \rfloor$ containing indices in $\{0, \dots, \lfloor \frac{k+\ell}{2} \rfloor - 1\}$ and in $\{\lfloor \frac{k+\ell}{2} \rfloor, \dots, k + \ell - 1\}$
$S_{down1} \in \mathbb{Z}_2^{\lceil \frac{p}{2} \rceil - 1\times\ell}, S_{down2} \in \mathbb{Z}_2^{\lfloor \frac{p}{2} \rfloor - 1\times\ell}$: partial sums matrices containing vectors obtained as the sum between $\widehat{\boldsymbol{s}}_{\mathbf{down}}$ and the columns of the matrix V_{down} indexed by α_1 and α_2
$\boldsymbol{\sigma}_{\mathbf{down1}}, \boldsymbol{\sigma}_{\mathbf{down2}} \in \mathbb{Z}_2^{\ell}$
$\boldsymbol{\sigma}_{\mathbf{up}} \in \mathbb{Z}_2^{r-\ell}$: vector containing the sum between $\widehat{\boldsymbol{s}}_{\mathbf{up}}$ and the columns of V_{up} indexed by $\alpha_1 \cup \alpha_2$
θ: list containing pairs made by indices taken from α_1 and $\boldsymbol{\sigma}_{\boldsymbol{down1}}$ vector, initially is empty.

```
1  repeat
2    repeat
3      ⟨[I_r  V], U, χ, rref_error⟩ ← FIND_PARTIAL_RREF(H)
4    until rref_error = true
5    ŝ ← PRODUCT_BIT_MATRIX_VECTOR(U, s)
6    α_1 ← INIT_COMBINATION_ARRAY(⌈p/2⌉, 0)
7    INIT_PARTIAL_SUMS(S_down1, V_down, ŝ_down, α_1)
8    for j ← 0 to binom((k+ℓ)/2, p/2) do
9      σ_down1 ← NEXT_COL_SUM_OPTIMIZED(S_down1, V_down, ŝ_down, α_1)
10     θ ← θ ∪ ⟨α_1, σ_down1⟩
11     NEXT_COMB(α_1, 0, ⌊(k+ℓ)/2⌋ − 1)
12   SORT(θ)
13   α_2 ← INIT_COMBINATION_ARRAY(⌊p/2⌋, ⌊(k+ℓ)/2⌋)
14   INIT_PARTIAL_SUMS(S_down2, V_down, 0_{1×ℓ}, α_2)
15   for j ← 0 to binom((k+ℓ)/2, p/2) do
16     σ_down2 ← NEXT_COL_SUM_OPTIMIZED(S_down2, V_down, 0_{1×ℓ}, α_2)
17     ⟨left, right⟩ ← FIND_COLLISION(σ_down2, θ)
18     if left ≠ −1 ∧ right ≠ −1 then
19       foreach ⟨α_1, σ_down1⟩ in θ_[left:right] do
20         σ_up ← σ_up ⊕ ŝ_up
21         foreach i in α_1 ∪ α_2 do
22           σ_up ← σ_up ⊕ v_up i
23         if HAMMING_WEIGHT(σ_up) = w − p then
24           ê ← [σ_up  0_{1×(k+ℓ)}]
25           foreach i ∈ α_1 ∪ α_2 do
26             ê_{i+r−ℓ} ← 1
27           e ← ERROR_RECONSTRUCTION(ê, χ)
28           return e
29     NEXT_COMB(α_2, ⌊(k+ℓ)/2⌋, k + ℓ − 1)
30 until HW(e) = w
```

$\boldsymbol{s} \in \mathbb{Z}_2^r$ and the target weight w, the time complexity of Finiasz-Sendrier algorithm 3.2.7 for finding a target error $\boldsymbol{e} \in \mathbb{Z}_2^n$ such that $H\boldsymbol{e}^T = \boldsymbol{s}$ and $\mathtt{HW}(\boldsymbol{e}) = w$ is:

$$C_{\text{ISD}}(n, r, w, p, \ell) = \frac{1}{\text{Pr}_{\text{succ}}} c_{\text{iter}} = \frac{\binom{n}{w}}{\binom{(k+\ell)/2}{p/2}^2 \binom{r-\ell}{w-p}} \Big(C_{\text{IS-P}}(n, r, \ell) +$$

$$\binom{(k+\ell)/2}{p/2} \Big(2C_{\text{NextComb}}(\frac{(k+\ell)}{2}, \frac{p}{2}) + 2C_{\text{NextColSum}}(\frac{(k+\ell)}{2}, \frac{p}{2}, \ell) +$$

$$C_{\text{FindColl}}(\binom{(k+\ell)/2}{p/2}) + \frac{\binom{(k+\ell)/2}{p/2}}{2^\ell} p(r-\ell) \Big) \Big) + C_{\text{sort}}(\binom{(k+\ell)/2}{p/2}, \ell) + p$$

While the spatial complexity is:

$$S_{ISD}(n, r, w, p, \ell) = S_{\text{RREF}}(r, n) + \mathcal{O}\Big(\binom{(k+\ell)/2}{p/2} (\frac{p}{2} \log_2 (\frac{k+\ell}{2}) + \ell) \Big)$$

Proof. The success probability in the Finiasz-Sendrier algorithm is the division between the admissibile errors satisfying the hypothesis of the ISD that are $\binom{(k+\ell)/2}{p/2}^2 \binom{r-\ell}{w-p}$ and all the possible error vectors with weight w $\binom{n}{w}$. The loop at lines 8-11 builds the list θ spanning all the possible combinations $\binom{(k+\ell)/2}{p/2}$: for each of these we call the NextComb and the NextColSum. Then, we consider again all the possible combinations $\binom{(k+\ell)/2}{p/2}$ for computing the vectors for finding collisions in θ: we have one call to NextComb, one call to NextColSum and then we call the find collision procedure. When we find a collision we need to sum $p+1$ vectors of size $r-\ell$ obtaining the complexity $p(r-\ell)$. These sums are done with probability of $\frac{\binom{(k+\ell)/2}{p/2}}{2^\ell}$ since all the possible vectors $\widehat{\boldsymbol{s}}_{\mathbf{down}}$ are 2^ℓ and only $\binom{(k+\ell)/2}{p/2}$ attempts hitting the correct one are made. The sorting algorithm for ordering the list θ is called only once during an iteration and the last p term is for reconstructing the error from α. In the spatial complexity the first term is the one relative to the computation of the RREF. Then, we have to store a list θ made by $\binom{(k+\ell)/2}{p/2}$ elements. Each element takes ℓ bits for storing the binary vector and $\frac{p}{2} \log_2 (\frac{k+\ell}{2})$ bits for storing the array of indices. In conclusion, the space required by the list is $\binom{(k+\ell)/2}{p/2}(\frac{p}{2} \log_2 (\frac{k+\ell}{2}) + \ell)$ bits. □

3.2.7. May-Meurer-Thomae

The May-Meurer-Thomae (MMT) algorithm [1] improves the Finiasz-Sendrier's algorithm changing the way in which the p positions of the vector are chosen. Instead of splitting them equally as $\frac{p}{2}$ in the lefmost $\frac{k+\ell}{2}$ columns and $\frac{p}{2}$ in the rightmost $\frac{k+\ell}{2}$ ones, the algorithm picks two disjoint sets $\alpha, \beta \subset \{0, \dots, k+\ell-1\}$. The MMT algorithm considers the

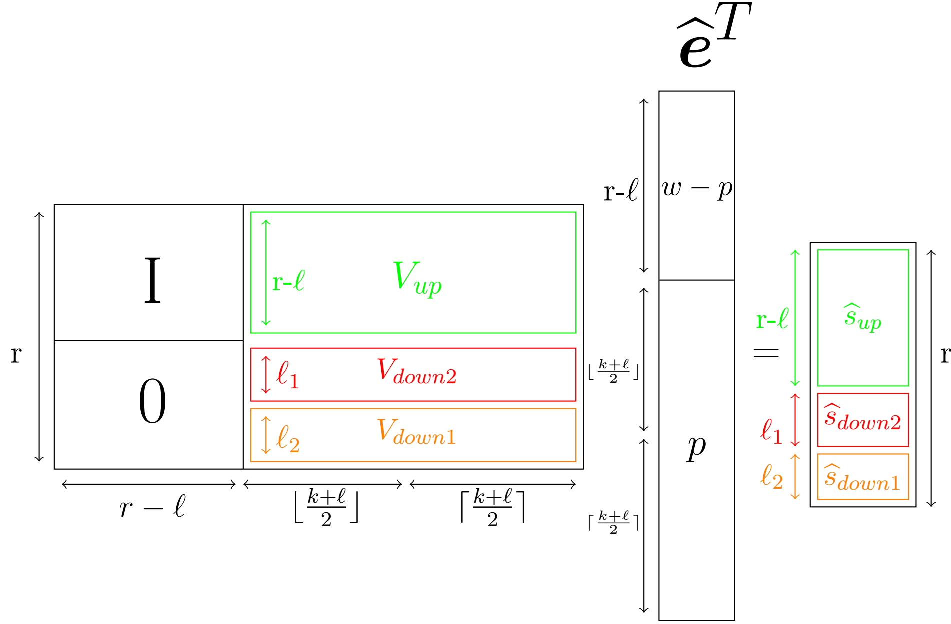


Figure 3.8: Weight distribution in May-Meurer-Thomae algorithm

V_{down} obtained after the partial RREF logically splitted row-wise into two sub-matrices: V_{down2} having ℓ_1 rows and V_{down1} having ℓ_2 rows, $V_{\text{down}} = \begin{bmatrix} V_{down2} \\ V_{down1} \end{bmatrix}$. After the call to the partial RREF the MMT works in the situation showed in Figure 3.8.
The sets α and β have size respectively equal to $p_{11} = \lceil \frac{p}{2} \rceil$ and $p_{12} = \lfloor \frac{p}{2} \rfloor$ and their union composes the p indices of the vector $\widehat{\boldsymbol{e}}_{\mathbf{S}} = \widehat{\boldsymbol{e}}_{[r-\ell+1:n]}$ set to 1. These sets are in turn obtained as the disjoint union of a pair of subsets of size $\frac{p}{4}$. To be precise: $\alpha = \alpha_1 \cup \alpha_2$ where α_1 and α_2 have size respectively equal to $p_{21} = \lceil \frac{p_{11}}{2} \rceil$ and $p_{22} = \lfloor \frac{p_{11}}{2} \rfloor$ while $\beta = \beta_1 \cup \beta_2$ where β_1 and β_2 have size respectively equal to $p_{23} = \lceil \frac{p_{12}}{2} \rceil$ and $p_{24} = \lfloor \frac{p_{12}}{2} \rfloor$.
The disjoint unions are realized picking the indices of α_1, β_1 in $\{0, \ldots, \frac{k+\ell}{2} - 1\}$ and the indices of α_2, β_2 in $\{\frac{k+\ell}{2}, \ldots, k+\ell-1\}$. Formally we have:

$$V_{\text{down}} \widehat{\boldsymbol{e}}_{\mathbf{S}}^T = \widehat{\boldsymbol{s}}_{\mathbf{down}} \quad \Leftrightarrow \quad \widehat{\boldsymbol{s}}_{\mathbf{down}} = \sum_{i \in \alpha} \boldsymbol{v}_{\mathbf{down\ i}} + \sum_{j \in \beta} \boldsymbol{v}_{\mathbf{down\ j}}$$

$$\begin{bmatrix} \widehat{\boldsymbol{s}}_{\mathbf{down2}} \\ \widehat{\boldsymbol{s}}_{\mathbf{down1}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{a}_{\mathbf{down2}} \\ \widehat{\boldsymbol{s}}_{\mathbf{down1}} \end{bmatrix} + \begin{bmatrix} \boldsymbol{b}_{\mathbf{down2}} \\ \mathbf{0}_{1 \times \ell_2} \end{bmatrix}, \begin{bmatrix} \boldsymbol{a}_{\mathbf{down2}} \\ \widehat{\boldsymbol{s}}_{\mathbf{down1}} \end{bmatrix} = \sum_{j \in \beta} \begin{bmatrix} \boldsymbol{v}_{\mathbf{down2\ j}} \\ \boldsymbol{v}_{\mathbf{down1\ j}} \end{bmatrix}, \begin{bmatrix} \boldsymbol{b}_{\mathbf{down2}} \\ \mathbf{0}_{1 \times \ell_2} \end{bmatrix} = \sum_{i \in \alpha} \begin{bmatrix} \boldsymbol{v}_{\mathbf{down2\ i}} \\ \boldsymbol{v}_{\mathbf{down1\ i}} \end{bmatrix}$$

and exploiting the same reasoning of Stern we can derive:

$$\begin{bmatrix} \boldsymbol{a}_{\mathbf{down2}} \\ \widehat{\boldsymbol{s}}_{\mathbf{down1}} \end{bmatrix} + \sum_{j \in \beta_1} \begin{bmatrix} \boldsymbol{v}_{\mathbf{down2\ j}} \\ \boldsymbol{v}_{\mathbf{down1\ j}} \end{bmatrix} = \sum_{j \in \beta_2} \begin{bmatrix} \boldsymbol{v}_{\mathbf{down2\ j}} \\ \boldsymbol{v}_{\mathbf{down1\ j}} \end{bmatrix}, \quad \begin{bmatrix} \boldsymbol{b}_{\mathbf{down2}} \\ \mathbf{0}_{1 \times \ell_2} \end{bmatrix} + \sum_{i \in \alpha_1} \begin{bmatrix} \boldsymbol{v}_{\mathbf{down2\ i}} \\ \boldsymbol{v}_{\mathbf{down1\ i}} \end{bmatrix} = \sum_{i \in \alpha_2} \begin{bmatrix} \boldsymbol{v}_{\mathbf{down2\ i}} \\ \boldsymbol{v}_{\mathbf{down1\ i}} \end{bmatrix}$$

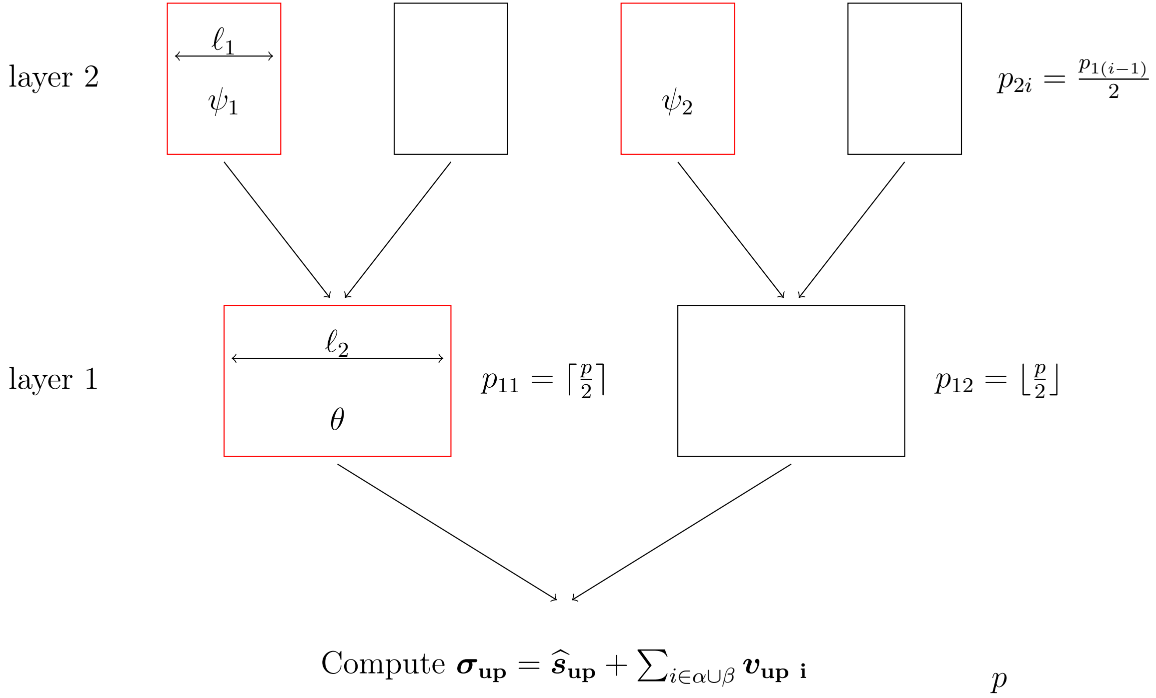


Figure 3.9: Lists of MMT algorithm

The elements of the first final list (at layer 1) holding candidate values for α, that we call from now θ, respect the following equation: $0_{1\times\ell_2} + \sum_{i\in\alpha_1} \boldsymbol{v}_{\mathbf{down1\ i}} = \sum_{i\in\alpha_2} \boldsymbol{v}_{\mathbf{down1\ i}}$ (θ in Figure 3.9). While the elements of the second final list respect: $\widehat{\boldsymbol{s}}_{\mathbf{down1}} + \sum_{j\in\beta_1} \boldsymbol{v}_{\mathbf{down1\ i}} = \sum_{j\in\beta_2} \boldsymbol{v}_{\mathbf{down1\ i}}$. Only the first final list θ needs to be materialized since all the vectors of the second final list can be computed on the fly reducing the memory used by the algorithm. For building the list θ we have to merge the two upper lists at layer 2. We will call ψ_1 the first materialized list at layer 2 holding pairs made by an array of indices α_1 long p_{21} and a binary vector long ℓ_1 resulting from $\sum_{i\in\alpha_1} \boldsymbol{v}_{\mathbf{down2\ i}}$. Then, for each combination of α_2 we compute on the fly the vectors resulting from $\sum_{i\in\alpha_2} \boldsymbol{v}_{\mathbf{down2\ i}}$: for each of these vectors we try to find collisions with the vectors saved in the list ψ_1. When we find a collision we save $\alpha = \alpha_1 \cup \alpha_2$ in the list θ at layer 1 together with the vector $\sum_{i\in\alpha} \boldsymbol{v}_{\mathbf{down1\ i}}$. The same reasoning applies for the right part of the Figure 3.9: we materialize a list ψ_2 holding pairs made by an array of indices β_1 and binary vectors $\widehat{\boldsymbol{s}}_{\mathbf{down2}} + \sum_{j\in\beta_1} \boldsymbol{v}_{\mathbf{down2\ i}}$ long ℓ_1 for all the possible combinations. Then, we compute on the fly the vectors for finding collisions in ψ_2 as done before. When a collision is found, we have a candidate pair (β_1, β_2) and we compute on the fly $\boldsymbol{\sigma}_{\mathbf{down1}} = \widehat{\boldsymbol{s}}_{\mathbf{down1}} + \sum_{j\in\beta} \boldsymbol{v}_{\mathbf{down1\ i}}$. Now, without saving the result, we immediately try to find a collision in the list θ: if a collision is found, and α and β are disjoint, we can check if $\mathtt{HW}(\widehat{\boldsymbol{s}}_{\mathbf{up}} + \sum_{i\in\alpha\cup\beta} \boldsymbol{v}_{\mathbf{up\ i}}) = w - p$. In Figure 3.9 the highlighted red lists are the materialized ones, while the others are not stored in the

implementation of the algorithm.

Algorithm 3.2.8: May-Meurer-Thomae algorithm

Input: $\boldsymbol{s} \in \mathbb{Z}_2^r$: syndrome vector
$H \in \mathbb{Z}_2^{r\times n}$: binary parity check matrix
w: the weight of the error vector to be recovered
ℓ : algorithmic parameter such as $0 \le \ell \le r - w + p$

Output: $\boldsymbol{e} \in \mathbb{Z}_2^n$: error vector to be recovered such that $H\boldsymbol{e}^T = \boldsymbol{s}$, with $\mathtt{wt}(\boldsymbol{e}) = w$

Data: χ: an array of size n containing the indices of the columns of H after being permuted in the RREF
p: the weight of the last $k+\ell$ bits of $\widehat{\boldsymbol{e}}$, $4 \le p \le w$.
$p_{11} = \lceil \frac{p}{2} \rceil, p_{12} = \lfloor \frac{p}{2} \rfloor, p_{21} = \lceil \frac{p_{11}}{2} \rceil, p_{22} = \lfloor \frac{p_{11}}{2} \rfloor, p_{23} = \lceil \frac{p_{12}}{2} \rceil, p_{24} = \lfloor \frac{p_{12}}{2} \rfloor$
ℓ_1, ℓ_2: parameters such that $\ell_1 + \ell_2 = \ell$
$U \in \mathbb{Z}_2^{r\times r}$: matrix representing elementary row operations in the RREF, at first is an identity matrix
$\widehat{\boldsymbol{s}} \in \mathbb{Z}_2^r$: permuted syndrome, $\widehat{\boldsymbol{s}} = \begin{bmatrix} \widehat{\boldsymbol{s}}_{\mathbf{up}} \\ \widehat{\boldsymbol{s}}_{\mathbf{down2}} \\ \widehat{\boldsymbol{s}}_{\mathbf{down1}} \end{bmatrix}$, with $\widehat{\boldsymbol{s}}_{\mathbf{up}} = \widehat{\boldsymbol{s}}_{[\mathbf{1}:\boldsymbol{r}-\boldsymbol{\ell}]}$, $\widehat{\boldsymbol{s}}_{\mathbf{down2}} = \widehat{\boldsymbol{s}}_{[\boldsymbol{r}-\boldsymbol{\ell}+\mathbf{1}:\boldsymbol{r}-\boldsymbol{\ell_2}]}$ and $\widehat{\boldsymbol{s}}_{\mathbf{down1}} = \widehat{\boldsymbol{s}}_{[\boldsymbol{r}-\boldsymbol{\ell_2}+\mathbf{1}:\boldsymbol{r}]}$
$V \in \mathbb{Z}_2^{r\times(k+\ell)}$: matrix $V = \begin{bmatrix} V_{\text{up}} \\ V_{\text{down2}} \\ V_{\text{down1}} \end{bmatrix}$ with $V_{\text{up}} = V_{[1:r-\ell][:]}$, $V_{\text{down2}} = V_{[r-\ell+1:r-\ell_2][:]}$ and $V_{\text{down1}} = V_{[r-\ell_2+1:r][:]}$, where $\boldsymbol{v}_{\mathbf{up}\ \mathbf{i}}$,$\boldsymbol{v}_{\mathbf{down2}\ \mathbf{i}}$ and $\boldsymbol{v}_{\mathbf{down1}\ \mathbf{i}}$ are the columns of matrix V_{up}, V_{down2} and V_{down1} indexed by i
α_1, α_2: arrays of sizes respectively p_{21} and p_{22} containing indices in $\{0, \dots, \lfloor \frac{k+\ell}{2} \rfloor - 1\}$ and in $\{\lfloor \frac{k+\ell}{2} \rfloor, \dots, k+\ell-1\}$
β_1, β_2: arrays of sizes respectively p_{23} and p_{24} containing indices in $\{0, \dots, \lfloor \frac{k+\ell}{2} \rfloor - 1\}$ and in $\{\lfloor \frac{k+\ell}{2} \rfloor, \dots, k+\ell-1\}$
$S_{\alpha_1} \in \mathbb{Z}_2^{p_{21}-1\times\ell}, S_{\alpha_2} \in \mathbb{Z}_2^{p_{22}-1\times\ell}$: partial sums matrices containing vectors obtained as the sum of the columns of the matrix V_{down2} indexed by α_1 and α_2
$S_{\beta_1} \in \mathbb{Z}_2^{p_{23}-1\times\ell}, S_{\beta_2} \in \mathbb{Z}_2^{p_{24}-1\times\ell}$: partial sums matrices containing vectors obtained as the sum of the columns of the matrix V_{down2} indexed by β_1 and β_2
$\boldsymbol{\sigma}_{\mathbf{down2}} \in \mathbb{Z}_2^{\ell_1}$
$\boldsymbol{\sigma}_{\mathbf{down1}} \in \mathbb{Z}_2^{\ell_2}$
$\boldsymbol{\sigma}_{\mathbf{up}} \in \mathbb{Z}_2^{r-\ell}$:
ψ_1: first materialized list at layer 2 containing pairs made by indices in $\{0, \dots, \frac{k+\ell}{2} - 1\}$ taken from α_1 and $\boldsymbol{\sigma_{down2}}$ vector of size ℓ_1. The length of the list is $\binom{(k+\ell)/2}{p/2}$
ψ_2: second materialized list at layer 2 list containing pairs made by indices in $\{0, \dots, \frac{k+\ell}{2} - 1\}$ taken from β_1 and $\boldsymbol{\sigma_{down2}}$ vector of size ℓ_1. The length of the list is $\binom{(k+\ell)/2}{p/2}$
θ: list at layer 1 containing pairs made by indices in $\{0, \dots, k+\ell-1\}$ taken from $\alpha = \alpha_1 \cup \alpha_2$ and $\boldsymbol{\sigma_{down1}}$ vector of size ℓ_2. The length of the list is kept at most at $\binom{(k+\ell)/2}{p/2} / \binom{p}{p/2}$

```
1  repeat
2      repeat
3          ⟨[I_r  V], U, \chi, rref_error⟩ ← FIND_PARTIAL_RREF(H)
4      until rref_error = true
5      \hat{s} ← PRODUCT_BIT_MATRIX_VECTOR(U, s)
6      \alpha_1 ← INIT_COMBINATION_ARRAY(p_{21}, 0)
7      INIT_PARTIAL_SUMS(S_{\alpha_1}, V_{down2}, 0_{1×\ell_1}, \alpha_1)
8      \psi_1 ← ∅
9      for j ← 0 to \binom{(k+\ell)/2}{p_{21}} do
10         \sigma_{down2} ← NEXT_COL_SUM_OPTIMIZED(S_{\alpha_1}, V_{down2}, 0_{1×\ell_1}, \alpha_1)
11         \psi_1 ← \psi_1 ∪ ⟨\alpha_1, \sigma_{down2}⟩
12         NEXT_COMB(\alpha_1, 0, ⌊(k+\ell)/2⌋ − 1)
13     SORT(\psi_1)
14     \alpha_2 ← INIT_COMBINATION_ARRAY(p_{22}, ⌊(k+\ell)/2⌋)
15     INIT_PARTIAL_SUMS(S_{\alpha_2}, V_{down2}, 0_{1×\ell_1}, \alpha_2)
16     \theta ← ∅
17     for j ← 0 to \binom{(k+\ell)/2}{p_{22}} do
18         \sigma_{down2} ← NEXT_COL_SUM_OPTIMIZED(S_{\alpha_2}, V_{down2}, 0_{1×\ell_1}, \alpha_2)
19         ⟨left_2, right_2⟩ ← FIND_COLLISION(\sigma_{down2}, \psi_1)
20         if left_2 ≠ −1 ∧ right_2 ≠ −1 then
21             foreach ⟨\alpha_1⟩ in \psi_{1[left_2:right_2]} do
22                 \alpha ← \alpha_1 ∪ \alpha_2
23                 foreach i in \alpha do
24                     \sigma_{down1} ← \sigma_{down1} ⊕ v_{down1 i}
25                 \theta ← \theta ∪ ⟨\alpha, \sigma_{down1}⟩
26         NEXT_COMB(\alpha_2, ⌊(k+\ell)/2⌋, k + \ell − 1)
27     SORT(\theta)
28     \beta_1 ← INIT_COMBINATION_ARRAY(p_{23}, 0)
29     INIT_PARTIAL_SUMS(S_{\beta_1}, V_{down2}, 0_{1×\ell_1}, \beta_1)
30     \psi_2 ← ∅
31     for j ← 0 to \binom{(k+\ell)/2}{p_{23}} do
32         \sigma_{down2} ← NEXT_COL_SUM_OPTIMIZED(S_{\beta_1}, V_{down2}, 0_{1×\ell_1}, \beta_1)
33         \psi_2 ← \psi_2 ∪ ⟨\beta_1, \sigma_{down2}⟩
34         NEXT_COMB(\beta_1, 0, ⌊(k+\ell)/2⌋ − 1)
35     SORT(\psi_2)
36     \beta_2 ← INIT_COMBINATION_ARRAY(p_{24}, ⌊(k+\ell)/2⌋)
37     INIT_PARTIAL_SUMS(S_{\beta_2}, V_{down2}, 0_{1×\ell_1}, \beta_2)
38     for j ← 0 to \binom{(k+\ell)/2}{p_{24}} do
39         \sigma_{down2} ← NEXT_COL_SUM_OPTIMIZED(S_{\beta_2}, V_{down2}, \hat{s}_{down2}, \beta_2)
40         ⟨left_2, right_2⟩ ← FIND_COLLISION(\sigma_{down2}, \psi_2)
41         if left_2 ≠ −1 ∧ right_2 ≠ −1 then
42             foreach ⟨\beta_1⟩ in \psi_{2[left_2:right_2]} do
43                 \sigma_{down1} ← \hat{s}_{down1}
44                 foreach i in \beta_1 ∪ \beta_2 do
45                     \sigma_{down1} ← \sigma_{down1} ⊕ v_{down1 i}
46                 ⟨left_1, right_1⟩ ← FIND_COLLISION(\sigma_{down1}, \theta)
47                 if left_1 ≠ −1 ∧ right_1 ≠ −1 then
48                     foreach ⟨\alpha⟩ in \theta_{[left_1:right_1]} do
49                         if \alpha ∩ (\beta_1 ∪ \beta_2) = ∅ then
50                             \sigma_{up} ← \hat{s}_{up}
51                             foreach i in \alpha do
52                                 \sigma_{up} ← \sigma_{up} ⊕ v_{up i}
53                             foreach i in \beta_1 ∪ \beta_2 do
54                                 \sigma_{up} ← \sigma_{up} ⊕ v_{up i}
55                             if HAMMING_WEIGHT(\sigma_{up}) = w − p then
56                                 \hat{e} ← [\sigma_{up}  0_{1×k+\ell}]
57                                 foreach i ∈ \alpha do
58                                     \hat{e}_{i+r−\ell} ← 1
59                                 foreach i ∈ \beta_1 ∪ \beta_2 do
60                                     \hat{e}_{i+r−\ell} ← 1
61                                 e ← ERROR_RECONSTRUCTION(\hat{e}, \chi)
62                                 return e
63         NEXT_COMB(\beta_2, ⌊(k+\ell)/2⌋, k + \ell − 1)
64 until HW(e) = w
```

Theorem 3.10. *Given an instance of the syndrome decoding problem with $H \in \mathbb{Z}_2^{r\times n}$, $\boldsymbol{s} \in \mathbb{Z}_2^r$ and the target weight w, the time complexity of May-Meurer-Thomae algorithm 3.2.8 for finding a target error $\boldsymbol{e} \in \mathbb{Z}_2^n$ such that $H\boldsymbol{e}^T = \boldsymbol{s}$ and $\mathtt{HW}(\boldsymbol{e}) = w$ is:*

$$C_{\text{ISD}}(n, r, w, p, \ell_1, \ell_2) = \frac{1}{\text{Pr}_{\text{succ}}} c_{\text{iter}} = \frac{\binom{n}{w}}{\binom{k+\ell}{p}\binom{r-\ell}{w-p}} \Bigg(C_{\text{IS}-\text{P}}(n, r, \ell) +$$
$$\binom{\frac{k+\ell}{2}}{\frac{p}{4}} \Bigg(4C_{\text{NextComb}}((k+\ell)/2, p/4) + 4C_{\text{NextColSum}}((k+\ell)/2, p/4, \ell_1) +$$
$$2C_{\text{FindColl}}(\binom{\frac{k+\ell}{2}}{\frac{p}{4}}) + 2\frac{\binom{(k+\ell)/2}{p/4}}{\binom{p}{p/2}}\frac{1}{2^{\ell_1}}\frac{p}{2}\ell_2 +$$
$$\frac{\binom{(k+\ell)/2}{p/4}}{\binom{p}{p/2}}\frac{1}{2^{\ell_1}}(\frac{p}{2}\ell_2 + C_{\text{FindColl}}(\binom{\frac{k+\ell}{2}}{\frac{p}{2}})) + \frac{\binom{(k+\ell)/2}{p/2}\binom{p}{p/2}}{2^{\ell_2}} p(r-\ell)) \Bigg) \Bigg) +$$
$$2C_{\text{sort}}(\binom{\frac{k+\ell}{2}}{\frac{p}{4}}, \ell_1) + C_{\text{sort}}(\binom{\frac{k+\ell}{2}}{\frac{p}{2}}, \ell_2) + p$$

While the spatial complexity is:

$$S_{ISD}(n, r, w, p, \ell_1, \ell_2) = S_{\text{RREF}}(r, n) +$$
$$\mathcal{O}\Bigg(\binom{(k+\ell)/2}{p/4}(\frac{p}{4}\log_2(\frac{k+\ell}{2}) + \ell_1) + \binom{(k+\ell)/2}{p/2}(\frac{p}{2}\log_2(k+\ell) + \ell_2)\Bigg)$$

Proof. The success probability in the May-Meurer-Thomae algorithm is the division between the admissibile errors satisfying the hypothesis of the ISD that are $\binom{k+\ell}{p}\binom{r-\ell}{w-p}$ and all the possible error vectors with weight w $\binom{n}{w}$. Then we span four times all the possible $\binom{(k+\ell)/2}{p/4}$ combinations for building the two materialized lists at layer 2 and for computing on the fly the vectors of the other two lists. Therefore, we have four calls to NextComb and NextColSum. Then, for building the materialized list at layer 1, we call the find collision procedure and when collisions are found we need to sum $\frac{p}{2}$ vectors of size ℓ_2 that are the vectors we will save in the list at layer 1. These sums are done with probability of $\frac{\binom{(k+\ell)/2}{p/4}}{\binom{p}{p/2}}\frac{1}{2^{\ell_1}}$ since all the possible vectors are 2^{ℓ_1} and only $\frac{\binom{(k+\ell)/2}{p/4}}{\binom{p}{p/2}}$ attempts hitting the correct one are made since only $\binom{p}{p/2}$ representations of the solution exists. The find collision procedure is called another time to compute the vectors on the fly of the other list at layer 1. Then, with the same probability already discussed, we need to do $(p+1)/2$ sums between vectors of size ℓ_2 and we call the find collision procedure to find collisions in the list at layer 1. If a collision is found we finally sums $p+1$ vectors of size $r-\ell$ for checking the weight. These final sums are done with probability of $\frac{\binom{(k+\ell)/2}{p/2}\binom{p}{p/2}}{2^{\ell_2}}$ since all the possible vectors are 2^{ℓ_2} and only $\binom{(k+\ell)/2}{p/2}\binom{p}{p/2}$ attempts hitting the correct one

are made. Since we have three lists to sort for applying later the find collision algorithm we have three calls to the sorting procedure. The last p term is for reconstructing the error from $\alpha \cup \beta$. In the spatial complexity we report only the significant terms. The first is the one relative to the computation of the RREF. Then, we have to store two lists made by $\binom{(k+\ell)/2}{p/4}$ elements. Each element takes ℓ_1 bits for storing the binary vector and $\frac{p}{4}\log_2\left(\frac{k+\ell}{2}\right)$ bits for storing the array of indices. The last term is the space for the list at layer 1. □

3.2.8. Becker-Joux-May-Meurer

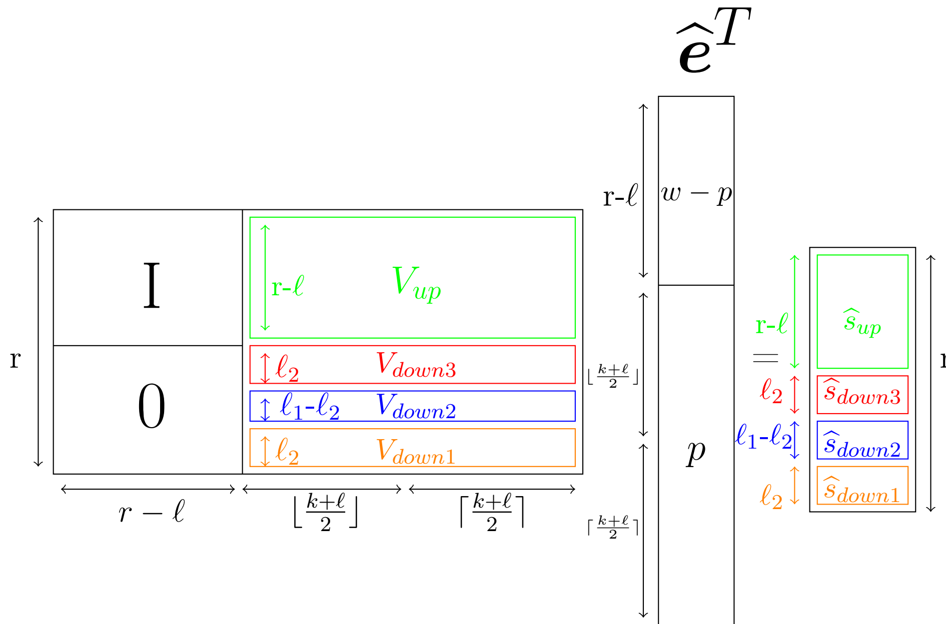


Figure 3.10: Weight distribution in BJMM algorithm

The Becker-Joux-May-Meurer algorithm [8] improves the May-Meurer-Thomae algorithm adding one layer for building the lists recursively and changing how the elements of the lists are generated. The BJMM algorithm considers that it is possible to represent the error with weight p, $\widehat{\boldsymbol{e}}_{\mathbf{S}}$, as the sum of two error vectors $\widehat{\boldsymbol{e}}_1$ and $\widehat{\boldsymbol{e}}_2$ with weight equal to $\frac{p}{2}+\Delta$ under the assumption that the extra Δ ones cancel out during the addition. Allowing this extra weights raises the number of valid pairs $\widehat{\boldsymbol{e}}_1, \widehat{\boldsymbol{e}}_2$ to represent $\widehat{\boldsymbol{e}}_{\mathbf{S}}$ by a factor equal to $\binom{k+\ell-p}{\Delta}$. This improvement is employed to reduce the size of the lists of values which need to be computed to find $\widehat{\boldsymbol{e}}_{\mathbf{S}}$ with a meet-in-the-middle strategy checking $V_{\text{down}}\widehat{\boldsymbol{e}}_1^T + V_{\text{down}}\widehat{\boldsymbol{e}}_2^T = \widehat{\boldsymbol{s}}_{\text{down}}$. The existing pairs $(\widehat{\boldsymbol{e}}_1, \widehat{\boldsymbol{e}}_2)$ for constructing $\widehat{\boldsymbol{e}}_{\mathbf{S}}$ are $\mathcal{R} = \binom{p}{p/2}\binom{k+\ell-p}{\Delta}$ and only a $\frac{1}{\mathcal{R}}$ fraction of the $\binom{k+\ell}{p/2+\Delta}^2$ possible pairs can be exhaustively searched with the expectation of finding a solution (assuming an uniform distribution over all the possible

ones). For enumerating only a $\frac{1}{\mathcal{R}}$ fraction of the pairs the BJMM, as the MMT algorithm, performs partial checks on a smaller bits than ℓ discarding the invalid pairs. The partial checks seen in MMT work only with two layers while the BJMM works in three layers considering the extra weights. Formally, for constructing $\widehat{\boldsymbol{e}}_{\mathbf{S}}$, we search for two binary vectors $\widehat{\boldsymbol{e}}_1{}^{(1)}$ and $\widehat{\boldsymbol{e}}_2{}^{(1)}$ with weights repectively equal to $p_{11} = \lceil \frac{p}{2} \rceil + \Delta_1$ and $p_{12} = \lfloor \frac{p}{2} \rfloor + \Delta_1$. In turn $\widehat{\boldsymbol{e}}_1{}^{(1)}$ and $\widehat{\boldsymbol{e}}_2{}^{(1)}$ are generated by the pairs $\widehat{\boldsymbol{e}}_1{}^{(2)}$, $\widehat{\boldsymbol{e}}_2{}^{(2)}$ and $\widehat{\boldsymbol{e}}_3{}^{(2)}$, $\widehat{\boldsymbol{e}}_4{}^{(2)}$ with weights respectively equal to $p_{21} = \lceil \frac{p_{11}}{2} \rceil + \Delta_2, p_{22} = \lfloor \frac{p_{11}}{2} \rfloor + \Delta_2, p_{23} = \lceil \frac{p_{12}}{2} \rceil + \Delta_2, p_{24} = \lfloor \frac{p_{12}}{2} \rfloor + \Delta_2$. Finally $\widehat{\boldsymbol{e}}_1{}^{(2)}$, $\widehat{\boldsymbol{e}}_2{}^{(2)}$, $\widehat{\boldsymbol{e}}_3{}^{(2)}$, $\widehat{\boldsymbol{e}}_4{}^{(2)}$ are generated by the pairs $\widehat{\boldsymbol{e}}_{2i}{}^{(3)}$,$\widehat{\boldsymbol{e}}_{2i+1}{}^{(3)}$, $i \in \{1,2,3,4\}$ with weights $p_{31} = \lceil \frac{p_{21}}{2} \rceil, p_{32} = \lfloor \frac{p_{21}}{2} \rfloor, p_{33} = \lceil \frac{p_{22}}{2} \rceil, p_{34} = \lfloor \frac{p_{22}}{2} \rfloor, p_{35} = \lceil \frac{p_{23}}{2} \rceil, p_{36} = \lfloor \frac{p_{23}}{2} \rfloor, p_{37} = \lceil \frac{p_{24}}{2} \rceil, p_{38} = \lfloor \frac{p_{24}}{2} \rfloor$. At layer 3 no extra weights are used for generating the errors at layer 2.

For adopting this approach no overlapping positions for the ones should be present between any pairs of errors at layer 3: this is solved choosing the positions of the ones of $\widehat{\boldsymbol{e}}_{2i}{}^{(3)}$ from a set that is disjoint from the set where we pick the ones of $\widehat{\boldsymbol{e}}_{2i+1}{}^{(3)}$. After the partial RREF we have the situation showed in Figure 3.10 where we have to choose two parameters, ℓ_1 and ℓ_2 such that $\ell_1 > \ell_2$. The last thing we need to consider is that all the values $V_{\text{down}_1}\widehat{\boldsymbol{e}}_i^{(1)}, V_{\text{down}_2}\widehat{\boldsymbol{e}}_i^{(2)}$ are matched against should be unrelated, so that the sampling of pairs during the merge of two lists is picking the items indipendently from another list merger on the same level. This allows the list merger at a lower level to consider the elements from above to be picked at random. Formally we need to modify the matching equations, considering $\boldsymbol{x_1}$, $\boldsymbol{x_2}$ and $\boldsymbol{x_3}$ random bit vectors long ℓ_2:

$$V_{\text{down1}}\widehat{\boldsymbol{e}}_1{}^{(1)} = V_{\text{down1}}\widehat{\boldsymbol{e}}_2{}^{(1)} + \widehat{\boldsymbol{s}}_{\text{down1}} \rightarrow \text{no change}$$

$$V_{\text{down2}}\widehat{\boldsymbol{e}}_1{}^{(2)} = V_{\text{down2}}\widehat{\boldsymbol{e}}_2{}^{(2)} + \widehat{\boldsymbol{s}}_{\text{down2}} \rightarrow V_{\text{down2}}\widehat{\boldsymbol{e}}_1{}^{(2)} = V_{\text{down2}}\widehat{\boldsymbol{e}}_2{}^{(2)} + \widehat{\boldsymbol{s}}_{\text{down2}} + \boldsymbol{x_1}$$
$$V_{\text{down2}}\widehat{\boldsymbol{e}}_3{}^{(2)} = V_{\text{down2}}\widehat{\boldsymbol{e}}_4{}^{(2)} + \mathbf{0_{1\times(\ell_1-\ell_2)}} \rightarrow V_{\text{down2}}\widehat{\boldsymbol{e}}_3{}^{(2)} = V_{\text{down2}}\widehat{\boldsymbol{e}}_4{}^{(2)} + \boldsymbol{x_1}$$

$$V_{\text{down3}}\widehat{\boldsymbol{e}}_1{}^{(3)} = V_{\text{down1}}\widehat{\boldsymbol{e}}_2{}^{(3)} + \widehat{\boldsymbol{s}}_{\text{down3}} \rightarrow V_{\text{down3}}\widehat{\boldsymbol{e}}_1{}^{(3)} = V_{\text{down3}}\widehat{\boldsymbol{e}}_2{}^{(3)} + \widehat{\boldsymbol{s}}_{\text{down3}} + \boldsymbol{x_1} + \boldsymbol{x_2}$$
$$V_{\text{down3}}\widehat{\boldsymbol{e}}_3{}^{(3)} = V_{\text{down3}}\widehat{\boldsymbol{e}}_4{}^{(3)} + \mathbf{0_{1\times\ell_2}} \rightarrow V_{\text{down3}}\widehat{\boldsymbol{e}}_3{}^{(3)} = V_{\text{down3}}\widehat{\boldsymbol{e}}_4{}^{(3)} + \boldsymbol{x_2}$$
$$V_{\text{down3}}\widehat{\boldsymbol{e}}_5{}^{(3)} = V_{\text{down1}}\widehat{\boldsymbol{e}}_6{}^{(3)} + \mathbf{0_{1\times\ell_2}} \rightarrow V_{\text{down3}}\widehat{\boldsymbol{e}}_5{}^{(3)} = V_{\text{down3}}\widehat{\boldsymbol{e}}_6{}^{(3)} + \boldsymbol{x_1} + \boldsymbol{x_3}$$
$$V_{\text{down3}}\widehat{\boldsymbol{e}}_7{}^{(3)} = V_{\text{down3}}\widehat{\boldsymbol{e}}_8{}^{(3)} + \mathbf{0_{1\times\ell_2}} \rightarrow V_{\text{down3}}\widehat{\boldsymbol{e}}_7{}^{(3)} = V_{\text{down3}}\widehat{\boldsymbol{e}}_8{}^{(3)} + \boldsymbol{x_3}$$

As the previous algorithm we will use arrays of indices α and β instead of using the vectors $\widehat{\boldsymbol{e}}_i{}^{(j)}$ directly. Initially, for building the lists at layer 3, we need to extract random sets with random indices and span all the possible combinations: since here the indices are random and the combinations to be generated are not sequential, we can't use the

previous techniques of NextComb and NextColSum, but we need to recursively compute all the combinations of the indices inside the current set. At Figure 3.11 we can see how the lists in the BJMM algorithm are organized and at Algorithm 3.2.9 its functioning is reported.

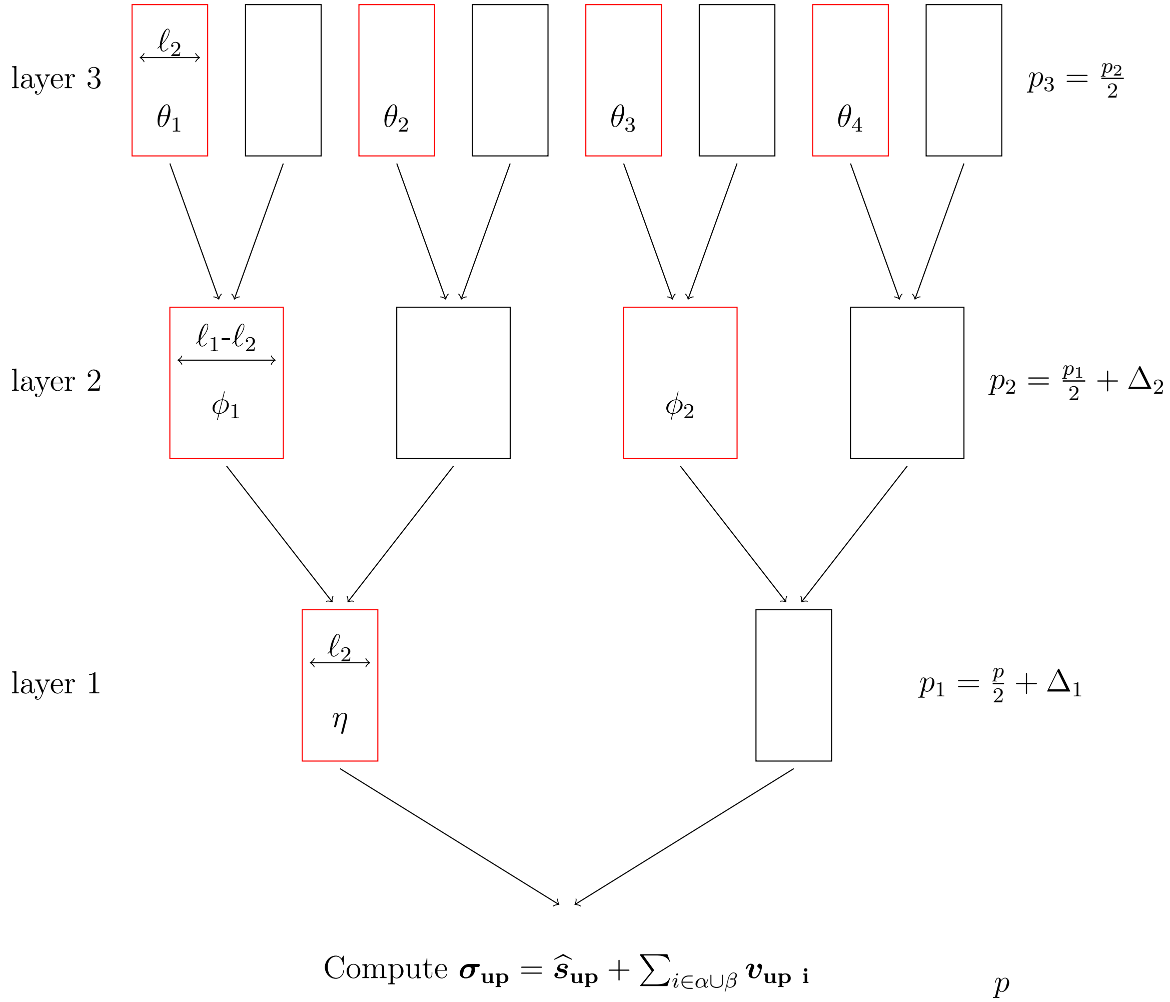


Figure 3.11: Lists of BJMM algorithm

Theorem 3.11. *Given an instance of the syndrome decoding problem with $H \in \mathbb{Z}_2^{r \times n}$, $\boldsymbol{s} \in \mathbb{Z}_2^r$ and the target weight w, the time complexity of Becker-Joux-May-Meurer algorithm 3.2.9 for finding a target error $\boldsymbol{e} \in \mathbb{Z}_2^n$ such that $H\boldsymbol{e}^T = \boldsymbol{s}$ and $\mathtt{HW}(\boldsymbol{e}) = w$ is:*

$$C_{\text{ISD}}(n, r, w, p, \ell_1, \ell_2, \Delta_1, \Delta_2) = \frac{1}{\text{Pr}_{\text{succ}}} c_{\text{iter}}$$

where the success probability is equal to the one of the MMT multiplied by the factor quantifying the fact that it is possible, picking two disjoint sets over the subsets of p_i positions from the error vector are selected, may result in a set which does not contain

Algorithm 3.2.9: Becker-Joux-May-Meurer algorithm

Input: $\boldsymbol{s} \in \mathbb{Z}_2^r$: syndrome vector
$H \in \mathbb{Z}_2^{r \times n}$: binary parity check matrix
w: the weight of the error vector to be recovered
ℓ : algorithmic parameter such as $0 \leq \ell \leq r - w + p$

Output: $\boldsymbol{e} \in \mathbb{Z}_2^n$: error vector to be recovered such that $H\boldsymbol{e}^T = \boldsymbol{s}$, with $\texttt{wt}(\boldsymbol{e}) = w$

Data: χ: an array of size n containing the indices of the columns of H after being permuted in the RREF
$U \in \mathbb{Z}_2^{r \times r}$: matrix representing elementary row operations done in the RREF
p: the weight of the last $k + \ell$ bits of $\widehat{\boldsymbol{e}}$, $0 \leq p \leq w$.
Δ_1, Δ_2: extra weights that will cancel out during the additions.
$p_{11} = \lceil \frac{p}{2} \rceil + \Delta_1, p_{12} = \lfloor \frac{p}{2} \rfloor + \Delta_1$
$p_{21} = \lceil \frac{p_{11}}{2} \rceil + \Delta_2, p_{22} = \lfloor \frac{p_{11}}{2} \rfloor + \Delta_2, p_{23} = \lceil \frac{p_{12}}{2} \rceil + \Delta_2, p_{24} = \lfloor \frac{p_{12}}{2} \rfloor + \Delta_2$
$p_{31} = \lceil \frac{p_{21}}{2} \rceil, p_{32} = \lfloor \frac{p_{21}}{2} \rfloor, p_{33} = \lceil \frac{p_{22}}{2} \rceil, p_{34} = \lfloor \frac{p_{22}}{2} \rfloor, p_{35} = \lceil \frac{p_{23}}{2} \rceil, p_{36} = \lfloor \frac{p_{23}}{2} \rfloor, p_{37} = \lceil \frac{p_{24}}{2} \rceil, p_{38} = \lfloor \frac{p_{24}}{2} \rfloor$
ℓ_1, ℓ_2: parameters respecting $0 \leq \ell_2 < \ell_1 \leq \ell$
$\widehat{\boldsymbol{s}} \in \mathbb{Z}_2^r$: permuted syndrome, $\widehat{\boldsymbol{s}} = \begin{bmatrix} \widehat{\boldsymbol{s}}_{\mathbf{up}} \\ \widehat{\boldsymbol{s}}_{\mathbf{down3}} \\ \widehat{\boldsymbol{s}}_{\mathbf{down2}} \\ \widehat{\boldsymbol{s}}_{\mathbf{down1}} \end{bmatrix}$, with $\widehat{\boldsymbol{s}}_{\mathbf{up}} = \widehat{\boldsymbol{s}}_{[\mathbf{1}:r-\ell]}$,
$\widehat{\boldsymbol{s}}_{\mathbf{down3}} = \widehat{\boldsymbol{s}}_{[r-\ell+1:r-\ell_1]}$, $\widehat{\boldsymbol{s}}_{\mathbf{down2}} = \widehat{\boldsymbol{s}}_{[r-\ell_1+1:r-\ell_2]}$ and $\widehat{\boldsymbol{s}}_{\mathbf{down1}} = \widehat{\boldsymbol{s}}_{[r-\ell_2+1:r]}$
$V \in \mathbb{Z}_2^{r \times (k+\ell)}$: matrix $V = \begin{bmatrix} V_{\text{up}} \\ V_{\text{down3}} \\ V_{\text{down2}} \\ V_{\text{down1}} \end{bmatrix}$ with $V_{\text{up}} = V_{[1:r-\ell][:]}$,
$V_{\text{down3}} = V_{[r-\ell+1:r-\ell_1][:]}$, $V_{\text{down2}} = V_{[r-\ell_1+1:r-\ell_2][:]}$ and $V_{\text{down1}} = V_{[r-\ell_2+1:r][:]}$,
where $\boldsymbol{v}_{\mathbf{up\ i}}$,$\boldsymbol{v}_{\mathbf{down3\ i}}$,$\boldsymbol{v}_{\mathbf{down2\ i}}$ and $\boldsymbol{v}_{\mathbf{down1\ i}}$ are the columns of matrix V_{up}, V_{down3},V_{down2} and V_{down1} indexed by i
α: array of size p_{3j} where $j = 1, 3, 5, 7$ containing $\lceil \frac{k+\ell}{2} \rceil$ random indices in $\{0, \ldots, k + \ell - 1\}$
β: array of size p_{3z} where $z = 2, 4, 6, 8$ containing $\lfloor \frac{k+\ell}{2} \rfloor$ random indices in $\{0, \ldots, k + \ell - 1\}$
θ: array of four lists holding the materialized lists at layer 3 , they contain pairs made by an array of indices long p_{3j} taken by α and a vector sum with length ℓ_2 where $j = 1, 3, 5, 7$ depends on the iteration of the main loop
θ_sizes: array of four integer holding the size for each list inside the array θ
θ_{aux}: auxiliary list at layer 3, it contains pairs made by an array of indices long p_{3j} taken by β and a vector sum with length ℓ_2 where $j = 2, 4, 6, 8$ depends on the iteration of the main loop
θ_{aux}_size: integer holding the size of the auxiliary list
ϕ: array of two lists holding the materialized lists at layer 2 , they contain pairs made by an array of indices with maximum length p_{2j} and a vector sum with length $\ell_1 - \ell_2$ where $j = 0, 2$ depends on the iteration of the main loop
ϕ_sizes: array of two integer holding the size for each list inside the array ϕ
η: list holding the materialized list at layer 1 , it contains pairs of an array of indices with maximum length p_{11} and a vector sum with length $\ell - \ell_1$
η_size: integer holding the size of the array η

```
1  repeat
2      repeat
3          ⟨[I_r  V], U, χ, rref_error⟩ ← FIND_PARTIAL_RREF(H)
4      until rref_error = true
5      ŝ ← PRODUCT_BIT_MATRIX_VECTOR(U, s)
6      x_1, x_2 ← GENERATE_RANDOM_BIT_STRING(ℓ_2)
7      for i ← 0 to 3 do
8          switch i do
9              case 0 do
10                 α_size ← p_31
11                 β_size ← p_32
12             case 1 do
13                 α_size ← p_33
14                 β_size ← p_34
15             case 2 do
16                 α_size ← p_35
17                 β_size ← p_36
18             case 3 do
19                 α_size ← p_37
20                 β_size ← p_38
21         if i = 1 ∨ i = 3 then
22             if φ_sizes_{i/2} > 0 then
23                 SORT(φ_{i/2})
24             else
25                 break
26         if i = 2 then
27             if η_size > 0 then
28                 SORT(η)
29             else
30                 break
31         α ← RANDOM_EXTRACT(k + ℓ)
32         β ← {0, ..., k + ℓ − 1} \ α
33         θ_sizes_i ← 0
34         foreach comb in AllComb(α) do
35             if i = 0 then
36                 σ_down3 ← ŝ_down3 ⊕ x_1 ⊕ x_2
37             if i = 1 ∨ i = 3 then
38                 σ_down3 ← x_2
39             if i = 2 then
40                 σ_down3 ← x_1 ⊕ x_2
41             σ_down3 ← σ_down3 ⊕ Σ_{z∈comb} v_down3 z
42             θ_sizes_i ← θ_sizes_i + 1
43             θ_i ← θ_i ∪ ⟨comb, σ_down3⟩
44         SORT(θ_i)
45         foreach comb in AllComb(β) do
46             σ_2 ← Σ_{z∈comb} v_down3 z
47             θ_aux ← θ_aux ∪ ⟨comb, σ_2⟩
48             θ_aux_size ← θ_aux_size + 1
49         foreach ⟨β^(3), σ_2⟩ in θ_aux do
50             ⟨left_3, right_3⟩ ← FIND_COLLISION(σ_down3, θ_i)
51             if left_3 ≠ −1 ∧ right_3 ≠ −1 then
52                 foreach ⟨α^(3), σ^(3)⟩ in θ_{i[left_3:right_3]} do
53                     δ^(2) ← α^(3) ∪ β^(3)
54                     if i = 0 then
55                         φ_0 ← φ_0 ∪ ⟨δ^(2), Σ_{z∈δ^(2)} v_down2 z ⊕ ŝ_down2⟩
56                         φ_sizes_0 ← φ_sizes_0 + 1
57                     if i = 2 then
58                         φ_1 ← φ_1 ∪ ⟨δ^(2), Σ_{z∈δ^(2)} v_down2 z⟩
59                         φ_sizes_1 ← φ_sizes_1 + 1
60                     else
61                         ⟨left_2, right_2⟩ ← FIND_COLLISION(Σ_{z∈δ^(2)} v_down2 z, φ_{i/2})
62                         if left_2 ≠ −1 ∧ right_2 ≠ −1 then
63                             foreach ⟨α^(2), σ^(2)⟩ in φ_{i/2[left_2:right_2]} do
64                                 δ^(1) ← (α^(2) ∪ δ^(2)) \ (α^(2) ∩ δ^(2))
65                                 if Size(δ^(1)) = (i = 1 ? p_11 : p_12) then
66                                     if i = 1 then
67                                         η ← η ∪ ⟨δ^(1), Σ_{z∈δ^(1)} v_down1 z ⊕ ŝ_down1⟩
68                                     else
69                                         ⟨left_1, right_1⟩ ← FIND_COLLISION(Σ_{z∈δ^(1)} v_down1 z, η)
70                                         if left_1 ≠ −1 ∧ right_1 ≠ −1 then
71                                             foreach ⟨α^(1), σ^(1)⟩ in η_[left_1:right_1] do
72                                                 δ^(0) ← (α^(1) ∪ δ^(1)) \ (α^(1) ∩ δ^(1))
73                                                 if SIZE(δ^(0)) = p then
74                                                     if HAMMING_WEIGHT(Σ_{z∈δ^(0)} v_up z ⊕ ŝ_up) = w − p then
75                                                         ê ← [Σ_{z∈δ^(0)} v_up z ⊕ ŝ_up   0_{1×k+ℓ}]
76                                                         foreach z in δ^(0) do
77                                                             ê_{z+r−ℓ} ← 1
78                                                         e ← ERROR_RECONSTRUCTION(ê, χ)
79                                                         return e
80 until HW(e) = w
```

enough positions. Such a factor is $\left(\frac{\binom{(k+\ell)/2}{p_3}^2}{\binom{k+\ell}{p_2}}\right)^4$. *The cost of an iteration is the following:*

$$\Bigg(C_{\text{IS-P}}(n,r,\ell)+4\big(k+\ell+2\binom{\frac{k+\ell}{2}}{p_3}+\binom{\frac{k+\ell}{2}}{p_3}^2(2p_3\ell_2)+C_{\text{FindColl}}(\binom{\frac{(k+\ell)}{2}}{p_3})\big)+$$

$$2\Bigg(\Big(\frac{\binom{p_1}{p_1/2}\binom{k+\ell-p_1}{\Delta_2}}{2^{\ell_2}}\binom{\frac{k+\ell}{2}}{p_3}^2\Big)^2(2p_2(\ell_1-\ell_2))\Bigg)+$$

$$\Bigg(\frac{\binom{p}{p/2}\binom{k+\ell-p}{\Delta_1}}{2^{\ell_1}}\Bigg(\frac{\binom{p_1}{p_1/2}\binom{k+\ell-p_1}{\Delta_2}}{2^{\ell_2}}\binom{\frac{k+\ell}{2}}{p_3}^2\Bigg)^2\Bigg)^2(2p_1\ell_2)+$$

$$\frac{\Bigg(\frac{\binom{p}{p/2}\binom{k+\ell-p}{\Delta_1}}{2^{\ell_1}}\Big(\frac{\binom{p_1}{p_1/2}\binom{k+\ell-p_1}{\Delta_2}}{2^{\ell_2}}\binom{(k+\ell)/2}{p_3}^2\Big)^2\Bigg)^2}{2^{\ell}}(p(r-\ell))\Bigg)+$$

$$4C_{\text{sort}}(\binom{(k+\ell)/2}{p_3},\ell_2)+2C_{\text{sort}}(\frac{\binom{(k+\ell)/2}{p_3}}{2^{\ell_2}},\ell_1)+C_{\text{sort}}(\frac{\binom{k+\ell}{p_1}\Big(\frac{\binom{(k+\ell)/2}{p_3}^2}{\binom{k+\ell}{p_1}}\Big)^2}{2^{\ell_2}},\ell_2)$$

While the spatial complexity is:

$$S_{ISD}(n,r,w,p,\ell_1,\ell_2,\Delta_1,\Delta_2)=$$

$$S_{\text{RREF}}(r,n)+\mathcal{O}\Bigg(\binom{(k+\ell)/2}{p_3}(p_3\log_2(\frac{k+\ell}{2})+\ell_2)+\frac{\binom{(k+\ell)/2}{p_3}}{2^{\ell_2}}(p_2\log_2(\frac{k+\ell}{2})+\ell_1-\ell_2)+$$

$$\frac{\binom{k+\ell}{p_1}\Big(\frac{\binom{(k+\ell)/2}{p_3}^2}{\binom{k+\ell}{p_1}}\Big)^2}{2^{\ell_2}}(p_1\log_2(\frac{k+\ell}{2})+\ell_2)\Bigg)$$

Proof. The first row of the complexity accounts for the computation of the RREF plus the creation of the lists at layer 3. $k+\ell$ refers to the computation of the disjoint sets of indices and the next terms are for computing the combinations and the vectors of length ℓ_2 taking into account that we need to build 4 materialzied lists and then 4 times to find collisions there. The second row accounts for the building of the lists at layer 2 with vectors of size $\ell_1-\ell_2$ with the same reasoning applied for the previous ISD complexities. The third row accounts for the building of the final list at layer 1 and the last considers the $p+1$ sums of vectors long $r-\ell$ done at the end of the ISD for checking if the weight of the first part of the error is equal to $w-p$. Last, we need to account the sorting for the the four lists at layer 3, the two lists at layer 2 and the one at layer 1 to apply the find collision procedure. The spatial complexity reports the memory needed for the computation of the RREF, the four lists at layer 3, the two lists at layer 2 and the final list at layer 1. □

3.2.9. Both-May

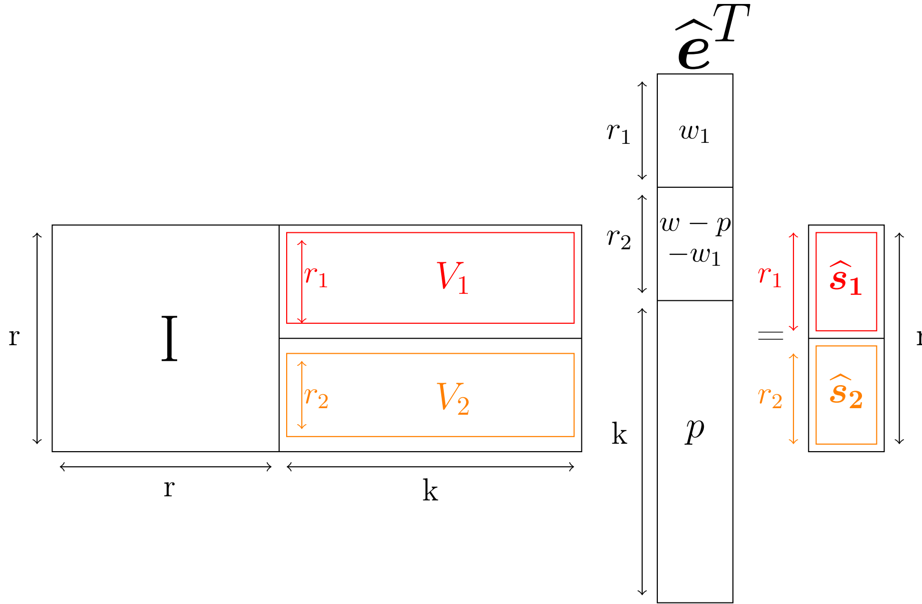


Figure 3.12: Weight distribution in Both-May algorithm

The Both-May algorithm [12] takes the idea of the BJMM algorithm but goes back computing a full systematic form and apply a nearest neighbor search technique for the construction of the lists. In the Both-May algorithm, the part of the permuted error in which we need to guess p error-affected positions is long k and not $k+\ell$ like the last three algorithms we have seen. This decreases the search space significantly. Moreover, it introduces a less restrictive weight distribution on a solution $(\widehat{e}_{S^*}, \widehat{e}_S) \in \mathbb{F}_2^r \times \mathbb{F}_2^k$, since usually $p << w$ and we only need small weight p on the last k coordinates instead of the last $k+\ell$ coordinates. This in turn means that we need less iterations to find a correct permutation that fulfills the wanted weight distribution. On the downside the routine that merges the lists is more costly than the previously seen therefore an iteration costs more but we do less iterations.

The find collision procedure used until now is replaced by the nearest neighbor search for merging the lists at the various layers. This procedure is used for solving the approximate matching problem in contrast to find the exact matching between vectors as done until now with the find collision method. Our goal is to construct $(\widehat{e}_1^{(1)}, \widehat{e}_2^{(1)})$ such that $\texttt{HW}(\widehat{e}_1^{(1)} + \widehat{e}_2^{(1)}) = p$ and the corresponding vectors $V\widehat{e}_1^{(1)}, V\widehat{e}_2^{(1)} + \widehat{s}$ approximately match on all r but $w-p$ coordinates. This immediately yields a solution $(\widehat{e}_{S^*}, \widehat{e}_S)$ with $\widehat{e}_{S^*} = V\widehat{e}_S + \widehat{s}$ and $\texttt{HW}(\widehat{s}) = w-p$. Let's suppose we have a list L_1, containing pairs made by a binary vector $x \in \mathbb{F}_2^r$ and an array of indices holding the p error-affected positions

as before, and a target binary vector $\boldsymbol{y} \in \mathbb{F}_2^r$. We are not interested anymore in finding an exact matching retrieving the range in the list such that the target vector $\boldsymbol{y}$ is equal to each vector indexed by the range, but we want to find the indices in the list such that , given a target weight w_i, satisfy $\texttt{HW}(\boldsymbol{x} + \boldsymbol{y}) = w_i$, where $\boldsymbol{x}$ is the vector saved in the list at the current index in analysis. Following this reasoning while we are constructing the k part of the error with weight p, using the same method as before with the array of indices, we can do some pre-checks even on the weight of the r-long part of the permuted error. In Algorithm 3.2.10 we can see the description of the nearest neighbor search just described. Since we need to scan all the list the complexity is linear with respect to its size: $C_{NN-search}(L, r) = \mathcal{O}(\text{SIZE}(L)(2r))$. As reported in the analysis inside the paper of the algorithm [12], even if the nearest neighbor search is more costly since we need a linear scan of all the list, it has been showed that the benefits outweigh this disadvantage, especially when the weight of our solution is large enough.

Algorithm 3.2.10: Nearest Neighbor Search algorithm

Input: L: list holding pairs made by a binary vector $\boldsymbol{x}$ long r and an array of integers α
$\boldsymbol{y}$: binary target error to find in L
w: target weight of the NN-search

Output: ρ: array containing the indices of L such that $\texttt{HW}(\boldsymbol{x} + \boldsymbol{y}) = w$

1 $\rho \leftarrow \emptyset$
2 **foreach** $\langle \boldsymbol{x}, \alpha \rangle$ **in** L **do**
3   **if** $\texttt{HW}(\boldsymbol{x} \oplus \boldsymbol{y}) = w$ **then**
4     $\rho \leftarrow \rho \cup \text{INDEX}(\eta, \langle \boldsymbol{x}, \alpha \rangle)$
5 **return** ρ

We study the algorithm with depth-2 (2 layers as in the MMT) but it can be generalized up to m layers. Let's see how the algorithm works. From Figure 3.12 we can notice that the algorithm splits the r part of the error that has weight equal to $w - p$ into two parts long r_1 and r_2 with weights respectively w_1 and w_2: doing this thanks to the nearest neighbor search we carry on through the layers not only the errors with the correct weight p_i on the k part of the permuted error but even the ones with a correct weight w_i in the r_i part of the error, where i depends on which layer we are. The p part of the error is computed as before splitting it into two errors with weights $p_1 = \frac{p}{2} + \Delta_1$. In turn, the two errors are produced from the four errors at layer 2 with weights $p_2 = \frac{p_1}{2}$. The lists at layer 2 hold the array of indices long p_2 corresponding to the error-affected positions of the k part of the permuted vector and a binary vector $\boldsymbol{\sigma_2}$ long r_1 obtained as the sum of the columns of V_1 indexed by the previous array. After we build the first list at layer 1 holding

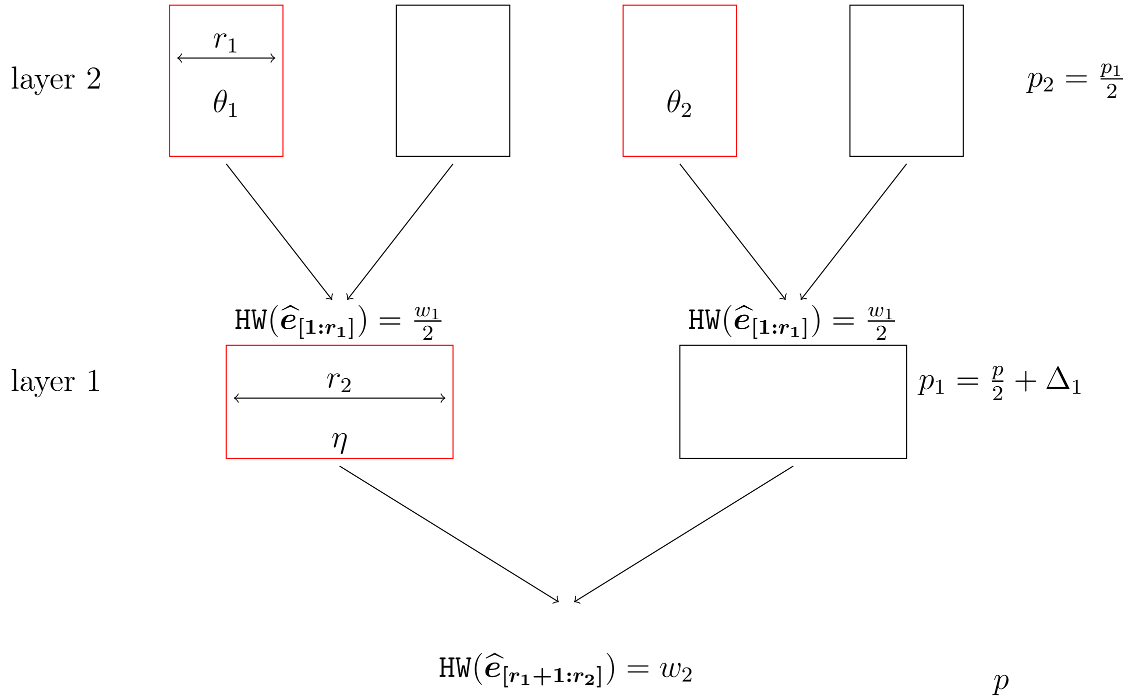


Figure 3.13: Lists of Both-May algorithm

$\widehat{\boldsymbol{e}}_{\boldsymbol{1}}^{(2)}$(lines 20-23 in Algorithm 3.2.11) , we solve the approximate matching problem using the vectors $\widehat{\boldsymbol{e}}_{\boldsymbol{2}}^{(2)}$ (computed at lines 24-29 in 3.2.11) applying the nearest neighbor search. If the sum of the vectors of the two different lists has weight equal to $\frac{w_1}{2}$, the union of the array of indices with length p_1 is saved in the first list at layer 1, together with a binary vector $\boldsymbol{\sigma_1}$ long r_2 obtained summing the columns of V_2 indexed by the union array (lines 34-40 in 3.2.11). In this situation the list at layer 1 holds the indices of p_1 positions of the k part of the error such that the first r_1 part has weight $\frac{w_1}{2}$. After doing the same steps just described for building the remained lists (the right part of the Figure 3.13), even the second list at layer 1 will contain vectors such that the r_1 part has weight $\frac{w_1}{2}$: next they will be merged using the nearest neighbor resulting on a weight $\frac{w_1}{2} + \frac{w_1}{2} = w_1$ as expected in the r_1 part (checking that the positions of the merged indices are disjoint). When we merge these two lists (line 44 in 3.2.11) the nearest neigbor search is applied again checking if the r_2 part of the permuted error has weight equal to $w_2 = w - p - w_1$: if this is the case, and the final array of indices has size equal to p, we have found a correct set for reconstructing the error. The situation just described can be seen in Figure 3.13 to understand how the lists are made.

Theorem 3.12. *Given an instance of the syndrome decoding problem with $H \in \mathbb{Z}_2^{r\times n}$, $\boldsymbol{s} \in \mathbb{Z}_2^r$ and the target weight w, the time complexity of Both-May algorithm 3.2.11 for*

Algorithm 3.2.11: Both-May algorithm

Input: $\boldsymbol{s} \in \mathbb{Z}_2^r$: syndrome vector
$H \in \mathbb{Z}_2^{r \times n}$: binary parity check matrix
w: the weight of the error vector to be recovered
Output: $\boldsymbol{e} \in \mathbb{Z}_2^n$: error vector to be recovered such that $H\boldsymbol{e}^T = \boldsymbol{s}$, with $\mathtt{wt}(\boldsymbol{e}) = w$
Data: χ: an array of size n containing the indices of the columns of H after being permuted in the RREF
$U \in \mathbb{Z}_2^{r \times r}$: matrix representing elementary row operations in the RREF, at first is an identity matrix
p: the weight of the last k bits of $\widehat{e}$, $0 \leq p \leq w$.
Δ_1: extra weight that will cancel out during the additions.
$p_{11} = \lceil \frac{p}{2} \rceil + \Delta_1, p_{12} = \lfloor \frac{p}{2} \rfloor + \Delta_1$
$p_{21} = \lceil \frac{p_{11}}{2} \rceil, p_{22} = \lfloor \frac{p_{11}}{2} \rfloor, p_{23} = \lceil \frac{p_{12}}{2} \rceil, p_{24} = \lfloor \frac{p_{12}}{2} \rfloor$
r_1, r_2: parameters that split the first part of $\widehat{e}$ respecting $r_1 + r_2 = r$
w_1, w_2: respectively the weights of the r_1 part and the r_2 part of the error. They are chosen such that $w_1 + w_2 = w - p$
$\widehat{\boldsymbol{s}} \in \mathbb{Z}_2^r$: permuted syndrome, $\widehat{\boldsymbol{s}} = \begin{bmatrix} \widehat{\boldsymbol{s}}_1 \\ \widehat{\boldsymbol{s}}_2 \end{bmatrix}$, with $\widehat{\boldsymbol{s}}_1 = \widehat{\boldsymbol{s}}_{[1:r_1]}$, $\widehat{\boldsymbol{s}}_2 = \widehat{\boldsymbol{s}}_{[r_1+1:r]}$
$V \in \mathbb{Z}_2^{r \times k}$: matrix $V = \begin{bmatrix} V_1 \\ V_2 \end{bmatrix}$ with $V_1 = V_{[1:r_1][:]}$, $V_2 = V_{[r_1+1:r][:]}$ where $\boldsymbol{v}_{1\ i}, \boldsymbol{v}_{2\ i}$ are the columns of matrix V_1 and V_2 indexed by i
α: array of size p_{2j} where $j = 1, 3$ containing $\lceil \frac{k}{2} \rceil$ random indices in $\{0, \dots, k-1\}$
β: array of size p_{2z} where $z = 2, 4$ containing $\lfloor \frac{k}{2} \rfloor$ random number indices in $\{0, \dots, k-1\}$
θ: array of two lists holding the materialized lists at layer 2 , they contain pairs made by an array of indices long p_{2j} taken by α and a vector sum with length r_1 where $j = 1, 3$ depends on the iteration of the main loop
θ_sizes: array of two integer holding the size for each list inside the array θ
θ_{aux}: auxiliary list at layer 2, it contains pairs made by an array of indices long p_{2j} taken by β and a vector sum with length r_1 where $j = 2, 4$ depends on the iteration of the main loop
θ_{aux}_size: integer holding the size of the auxiliary list θ_{aux}
η: list holding the materialized list at layer 1 , it's composed by pairs of an array of indices with maximum length p_{11} and a vector sum with length r_2. η_size is integer holding the size of the array η
$\rho^{(2)}, \rho^{(1)}$: arrays containing colliding indices returned by the nearest neighbor search at layer 2 and at layer 1

```
1  repeat
2    repeat
3      ⟨[I_r  V], U, χ, rref_error⟩ ← FIND_PARTIAL_RREF(H)
4    until rref_error = true
5    ŝ ← PRODUCT_BIT_MATRIX_VECTOR(U, s)
6    η_size ← 0
7    for i ← 0 to 1 do
8      switch i do
9        case 0 do
10         α_size ← p_21
11         β_size ← p_22
12       case 1 do
13         α_size ← p_23
14         β_size ← p_24
15         if η_size = 0 then
16           break
17     α ← RANDOM_EXTRACT(k)
18     β ← {0, ..., k − 1} \ α
19     θ_sizes_i ← 0
20     foreach comb in AllComb(α) do
21       σ_2 ← σ_2 ⊕ Σ_{z∈comb} v_1 z
22       θ_i ← θ_i ∪ ⟨comb, σ_2⟩
23       θ_sizes_i ← θ_sizes_i + 1
24     foreach comb in AllComb(β) do
25       if i = 1 then
26         σ_2 ← ŝ_1
27       σ_2 ← σ_2 ⊕ Σ_{z∈comb} v_1 z
28       θ_aux ← θ_aux ∪ ⟨comb, σ_2⟩
29       θ_aux_size ← θ_aux_size + 1
30     foreach ⟨β^(2), σ_2⟩ in θ_aux do
31       ρ^(2) ← NEAREST_NEIGHBOR_SEARCH(σ_2, θ_i)
32       if ρ^(2) ≠ ∅ then
33         foreach ⟨α^(2), μ_2⟩ in θ_{i[ρ^(2)]} do
34           δ^(1) ← α^(2) ∪ β^(2)
35           if i = 0 then
36             σ_1 ← ŝ_2
37             foreach ind in δ^(1) do
38               σ_1 ← σ_1 ⊕ v_2 ind
39             η ← η ∪ ⟨δ^(1), σ_1⟩
40             η_size ← η_size + 1
41           else
42             foreach ind in δ^(1) do
43               σ_1 ← σ_1 ⊕ v_2 ind
44             ρ^(1) ← NEAREST_NEIGHBOR_SEARCH(σ_1, η)
45             if ρ^(1) ≠ ∅ then
46               foreach ⟨α^(1), μ_1⟩ in η_[ρ^(1)] do
47                 δ^(0) ← (α^(1) ∪ δ^(1)) \ (α^(1) ∩ δ^(1))
48                 if SIZE(δ^(0)) = p then
49                   foreach ind in α^(1) do
50                     v ← v ⊕ v_1 ind
51                   v ← v ⊕ σ_2 ⊕ μ_2
52                   if HAMMING_WEIGHT(v) = w_1 then
53                     ê ← [v  μ_1 ⊕ σ_1  0_{1×k}]
54                     foreach k in δ^(0) do
55                       ê_{k+r} ← 1
56                     e ← ERROR_RECONSTRUCTION(ê, χ)
57                     return e
58 until HW(e) = w
```

finding a target error $\boldsymbol{e} \in \mathbb{Z}_2^n$ such that $H\boldsymbol{e}^T = \boldsymbol{s}$ and $\mathtt{HW}(\boldsymbol{e}) = w$ is:

$$C_{\mathrm{ISD}}(n, r, w, p, r_1, r_2, \Delta_1) = \frac{1}{\mathrm{Pr}_{\mathrm{succ}}} c_{\mathrm{iter}} = \frac{\binom{n}{w}}{\binom{k/2}{p/2}^2 \binom{r_1}{w_1}\binom{r_2}{w_2}} \Bigg(C_{\mathrm{IS}}(n, r) + 2k + \binom{\frac{k}{2}}{p_2}\Big(4 + 4p_2 r_1 + 2C_{\mathrm{NN-Search}}(\binom{\frac{k}{2}}{p_2}, r_1) + \frac{\binom{k/2}{p_2}^2\binom{r_1}{w_1/2}}{2^{r_1}} p_1 r_2 + \frac{\binom{k/2}{p_2}^2\binom{r_1}{w_1/2}}{2^{r_1}}\big(p_1 r_2 + C_{\mathrm{NN-Search}}(\frac{\binom{k/2}{p_2}^2\binom{r_1}{w_1/2}}{2^{r_1}}, r_2) + \frac{\binom{k}{p_1}\binom{p}{p/2}\binom{r_2}{w_2}}{2^{r_2}}(p + p_1 r_1)\big)\Big)\Bigg) + p$$

While the space complexity:

$$S_{\mathrm{ISD}}(n, r, w, p, r_1, r_2, \Delta_1) = S_{\mathrm{RREF}}(r, n) + \mathcal{O}\Bigg(\binom{k/2}{p_2}(p_2 \log_2(\frac{k}{2}) + r_1) + \frac{\binom{k/2}{p_2}^2\binom{r_1}{w_1/2}}{2^{r_1}}(\frac{p}{2}\log_2 k + r_2)\Bigg)$$

Proof. The success probability in the Both-May algorithm is the division between the admissibile errors satisfying the hypothesis of the ISD that are $\binom{k/2}{p/2}^2\binom{r_1}{w_1}\binom{r_2}{w_2}$ and all the possible error vectors with weight w $\binom{n}{w}$. We compute two times the random sets for an iteration resulting in the cost $2k$. Then, we span four times all the possible $\binom{k/2}{p_2}$ combinations for building the two materialized lists and the two auxiliary lists at layer 2. Therefore, we compute four times all the combinations and the sums of p_2 vectors of size r_2. Then, for the two lists at layer 1, we call the nearest neighbor procedure twice and when collisions are found we need to sum p_1 vectors of size r_2 that are the vectors we will save in the list at layer 1. These sums are done with probability of $\frac{\binom{k/2}{p_2}^2\binom{r_1}{w_1/2}}{2^{r_1}}$ since all the possible vectors are 2^{r_1} and only $\frac{\binom{k/2}{p_2}^2\binom{r_1}{w_1/2}}{2^{r_1}}$ attempts hitting the correct one are made. With the same probability already discussed we need to do $p_1 + 1$ sums between vectors of size r_2 and we call the nearest neighbor search for checking if the r_2 part has weight w_2. If a collision is found we finally compute the disjoint set of size p and we check if the vector in r_1 part has weight w_1 doing p_1 sums of vectors of size r_1. The last p term is for reconstructing the error from $\alpha \cup \beta$. In the spatial complexity we report only the significant terms. The first is the one relative to the computation of the RREF. Then we have to store two lists made by $\binom{k/2}{p_2}$ elements. Each element takes r_1 bits for storing the binary vector and $p_2 \log_2(\frac{k}{2})$ bits for storing the array of indices. The last term is the space for the list at layer 1 derived in the same manner considering that holds binary vectors long r_2. □

3.2.10. Esser-Bellini

The Esser-Bellini algorithm is a variant of the Both-May algorithm presented in [13]. The algorithm is almost equal to the Both-May algorithm, the only thing that changes is how the approximate matching problem is solved. The weight distribution is exactly the same and we can see it in Figure 3.12. In Both-May's algorithm we have used the nearest neighbor search to merge the lists present at layer 2 and at layer 1. Instead of applying the nearest neighbor search, the Esser-Bellini's algorithm uses the so called Indwik-Motwani routine based on locality sensitive hashing by Indwik and Motwani in [17]. Let's suppose we have two lists to merge L_1 and L_2 holding vectors long r: as before we want to find all the vectors $(\boldsymbol{x}, \boldsymbol{y}) \in L_1 \times L_2$ such that $\texttt{HW}(\boldsymbol{x}+\boldsymbol{y}) = w_i$. The Indwik-Motwani procedure guesses λ coordinates $I \subset [r]$ of the resulting vector $\boldsymbol{z} = \boldsymbol{x}+\boldsymbol{y}$ for which it assumes that all the bits of $\boldsymbol{z}$ indexed by I are zeroes, $\boldsymbol{z}_I = 0$. So, before checking the weight of the resulting vector, an exact matching on λ coordinate is done between $\boldsymbol{x}$ and $\boldsymbol{y}$. When we find a match we can proceed checking the weight of the resulting vector. The algorithm relies on the fact that the sum between the vectors will have small weight, therefore, for a certain λ coordinates, is more likely that on these coordinates the two vectors have the same values with respect to the situation where the sum has larger weight. The probability that $\boldsymbol{z}$ with $\texttt{HW}(\boldsymbol{z}) = w_i$ has bits equal to zero on the random choice of the λ coordinates is $P_r[\boldsymbol{z}_I = 0 \quad | \quad \boldsymbol{z} \in \mathbb{F}_2^r \wedge \texttt{HW}(\boldsymbol{z}) = w_i] = \frac{\binom{r-\lambda}{w_i}}{\binom{r}{w_i}}$. We can see the procedure just described in Algorithm 3.2.12.

Algorithm 3.2.12: Indwik-Motwani Search algorithm

Input: L_1: list holding pairs made by a binary vector $\boldsymbol{x}$ long r and an array of integers α
L_2: list holding pairs made by a binary vector $\boldsymbol{y}$ long r and an array of integers β
w_i: target weight

Data: I: set of size λ holding the random coordinates.

Output: L: list containing all the pairs $(\boldsymbol{x}, \boldsymbol{y})$ such that $\texttt{HW}(\boldsymbol{x}+\boldsymbol{y}) = w_i$

```
1 L ← ∅
2 I ← RANDOM_EXTRACT{0, ..., r − 1}
3 foreach (x, y) in L1 × L2 do
4 |   if x_I = y_I then
5 |   |   if HW(x + y) = w_i then
6 |   |   |   L ← L ∪ (x, y)
7 return L
```

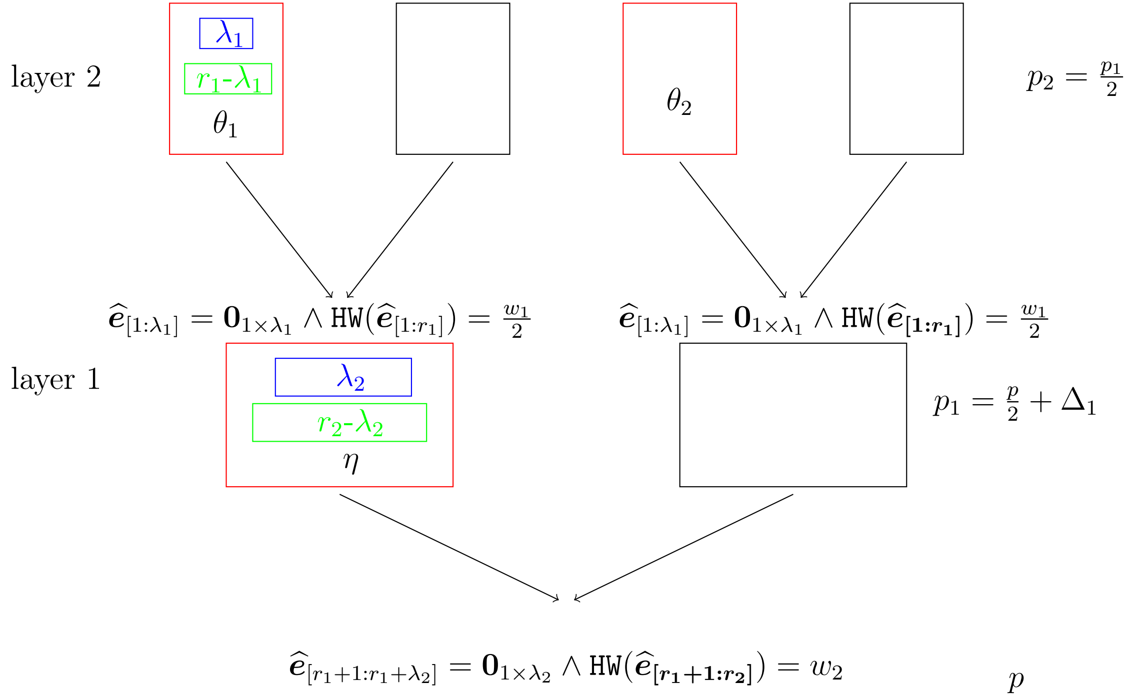


Figure 3.14: Lists of Esser-Bellini algorithm

In the implementation of this algorithm the following trick has been adopted: since the permutation applied in the RREF is random the binary vectors that will be considered will be random too. Knowing this we can consider the set I of λ coordinates always on the starting positions of the vectors. Therefore, the exact matching of the Indwik-Motwarni will be performed always on the starting λ bits of the vectors without loss of generality. In this way we consider lists containing triples made by a λ long binary vector, an $r-\lambda$ long binary vector and an array of indices used to construct the p part of the error as before. We have splitted the binary vectors long r into two parts for allowing the sorting of the lists as opposed as in Both-May where the lists couldn't be sorted since we needed to span all the elements of the list with the nearest neighbor search. Thanks to this splitting, we can sort these lists based on the λ long binary vectors: this helps us when we need to find two vectors of two lists with the same coordinates in the starting λ bits. Since the list is sorted, we can apply the binary range search to find the collision indices respecting the output of the Indiwk-Motwani and then we can proceed on checking the weight of the resulting vector obtained as the sum between the $r-\lambda$ vectors. Since we work with two layers, as in Both-May, we need to apply this procedure both in layer 2 and layer 1: we will indicate as λ_1 the starting number of coordinates used for Indwik-Motwani search at layer 2 with vector long r_1, while λ_2 the starting number of coordinates used for Indwik-Motwani search at layer 1 with vector long r_2. We can see the lists structure in the Figure 3.14.

The ISD algorithm is described in Algorithm 3.2.13, the structure is very similiar to the one of the Both-May algorithm, the only thing that changes is the splitting of the binary vectors for allowing the sorting of the lists and the subsequent call to the find collision algorithm.

Theorem 3.13. *Given an instance of the syndrome decoding problem with $H \in \mathbb{Z}_2^{r\times n}$, $\boldsymbol{s} \in \mathbb{Z}_2^r$ and the target weight w, the time complexity of Esser-Bellini algorithm 3.2.13 for finding a target error $\boldsymbol{e} \in \mathbb{Z}_2^n$ such that $H\boldsymbol{e}^T = \boldsymbol{s}$ and $\mathtt{HW}(\boldsymbol{e}) = w$ is:*

$$C_{\text{ISD}}(n, r, w, p, r_1, r_2, \lambda_1, \lambda_2, \Delta_1) = \frac{1}{\Pr_{\text{succ}}} c_{\text{iter}} = \frac{\binom{n}{w}}{\binom{k/2}{p/2}^2\binom{r_1}{w_1}\binom{r_2}{w_2}} \Bigg(C_{\text{IS}}(n, r) + 2k +$$
$$\binom{\frac{k}{2}}{p_2}\Bigg(4 + 4p_2 r_1 + 2\big(C_{\text{FindColl}}(\binom{\frac{k}{2}}{p_2}) + r_1\big) + \frac{\binom{k/2}{p_2}^2\binom{r_1}{w_1/2}}{2^{r_1}} p_1 r_2 +$$
$$\frac{\binom{k/2}{p_2}^2\binom{r_1}{w_1/2}}{2^{r_1}}\Big(p_1 r_2 + C_{\text{FindColl}}\big(\frac{\binom{k/2}{p_2}^2\binom{r_1}{w_1/2}}{2^{r_1}}\big) + r_2 + \frac{\binom{k}{p_1}\binom{p}{p/2}\binom{r_2}{w_2}}{2^{r_2}}(p + p_1 r_1)\Big)\Bigg)\Bigg) +$$
$$2C_{\text{sort}}(\binom{k/2}{p_2}, r_1) + C_{\text{sort}}(\frac{\binom{k/2}{p_2}^2\binom{r_1}{w_1/2}}{2^{r_1}}, r_2) + p$$

While the space complexity:

$$S_{\text{ISD}}(n, r, w, p, r_1, r_2, \lambda_1, \lambda_2, \Delta_1) = S_{\text{RREF}}(r, n) + \mathcal{O}\Bigg(\binom{k/2}{p_2}(p_2 \log_2(\frac{k}{2}) + \lambda_1 + (r_1 - \lambda_1)) +$$
$$\frac{\binom{k/2}{p_2}^2\binom{r_1}{w_1/2}}{2^{r_1}}(\frac{p}{2}\log_2 k + \lambda_2 + (r_2 - \lambda_2))\Bigg)$$

Proof. The complexity is similiar to the one of the Both-May's algorithm and most of the reasoning can be found there. Here the thing that changes is the using of the find collision procedure instead of the nearest neighbor search. Plus, we have a checking on the weight after the call to the find collision: two checks with vectors long r_1 and one check with vectors long r_2. For using the find collision procedure we need to sort the two lists at layer 2 and the one lists at layer 1 producing the final terms in the time complexity. For the space complexity we store two binary vectors in the lists instead of one: in the layer 2 one binary vector long λ_1 and the other long $r_1 - \lambda_1$ while in layer 1 one binary vector long λ_2 and the other long $r_2 - \lambda_2$. □

Algorithm 3.2.13: Esser-Bellini algorithm

Input: $\boldsymbol{s} \in \mathbb{Z}_2^r$: syndrome vector
$H \in \mathbb{Z}_2^{r\times n}$: binary parity check matrix
w: the weight of the error vector to be recovered

Output: $\boldsymbol{e} \in \mathbb{Z}_2^n$: error vector to be recovered such that $H\boldsymbol{e}^T = \boldsymbol{s}$, with $\mathtt{wt}(\boldsymbol{e}) = w$

Data: χ: an array of size n containing the indices of the columns of H after being permuted in the RREF
$U \in \mathbb{Z}_2^{r\times r}$: matrix representing elementary row operations in the RREF, at first is an identity matrix
p: the weight of the last k bits of $\widehat{\boldsymbol{e}}$, $0 \le p \le w$.
Δ_1: extra weight that will cancel out during the additions.
$p_{11} = \lceil \frac{p}{2} \rceil + \Delta_1, p_{12} = \lfloor \frac{p}{2} \rfloor + \Delta_1$
$p_{21} = \lceil \frac{p_{11}}{2} \rceil, p_{22} = \lfloor \frac{p_{11}}{2} \rfloor, p_{23} = \lceil \frac{p_{12}}{2} \rceil, p_{24} = \lfloor \frac{p_{12}}{2} \rfloor$
r_1, r_2: parameters that split the first part of $\widehat{\boldsymbol{e}}$ respecting $r_1 + r_2 = r$
w_1, w_2: respectively the weights of the r_1 part and the r_2 part of the error. They are chosen such that $w_1 + w_2 = w - p$
λ_1, λ_2 : parameters used for Indwik-Motwani search, they split the vector sums in two parts
$\widehat{\boldsymbol{s}} \in \mathbb{Z}_2^r$: permuted syndrome, $\widehat{\boldsymbol{s}} = \begin{bmatrix}\widehat{\boldsymbol{s}}_1\\ \widehat{\boldsymbol{s}}_2\end{bmatrix}$, with $\widehat{\boldsymbol{s}}_1 = \widehat{\boldsymbol{s}}_{[1:r_1]}$, $\widehat{\boldsymbol{s}}_2 = \widehat{\boldsymbol{s}}_{[r_1+1:r]}$
$V \in \mathbb{Z}_2^{r\times k}$: matrix $V = \begin{bmatrix}V_1\\ V_2\end{bmatrix}$ with $V_1 = V_{[1:r_1][:]}$, $V_2 = V_{[r_1+1:r][:]}$ where $\boldsymbol{v_1}\ _{\boldsymbol{i}}, \boldsymbol{v_2}\ _{\boldsymbol{i}}$ are the columns of matrix V_1 and V_2 indexed by i
α: array of size p_{2j} where $j = 1, 3$ containing $\lceil \frac{k}{2} \rceil$ random indices in $\{0, \dots, k-1\}$
β: array of size p_{2z} where $z = 2, 4$ containing $\lfloor \frac{k}{2} \rfloor$ random number indices in $\{0, \dots, k-1\}$
θ: array of two lists holding the materialized lists at layer 2 , they contain triples composed of an array of indices long p_{2j} taken by α, a vector sum with length λ_1 and a vector sum with lenght $r_1 - \lambda_1$ where $j = 1, 3$ depends on the iteration of the main loop
θ_sizes: array of two integer holding the size for each list inside the array θ
θ_{aux}: auxiliary list at layer 2, it contains pairs made by an array of indices long p_{2j} taken by β and a vector sum with length r_1 where $j = 2, 4$ depends on the iteration of the main loop. θ_{aux}_size is an integer holding the size of the auxiliary list θ_{aux}
η: list holding the materialized list at layer 1 , it's composed by triples of an array of indices with maximum length p_{11}, a vector sum with length λ_2 and a vector sum with length $r_2 - \lambda_2$. η_size is an integer holding the size of the array η.

1 **repeat**
2   **repeat**
3     $\langle [I_r \quad V], U, \chi, rref_error \rangle \leftarrow$ FIND_PARTIAL_RREF(H)
4   **until** $rref_error = true$
5   $\widehat{\boldsymbol{s}} \leftarrow$ PRODUCT_BIT_MATRIX_VECTOR$(U, \boldsymbol{s})$
6   $\eta_size \leftarrow 0$
7   **for** $i \leftarrow 0$ **to** 1 **do**
8   **switch** i **do**
9   **case** 0 **do**
10         $\alpha_size \leftarrow p_{21}$
11         $\beta_size \leftarrow p_{22}$
12   **case** 1 **do**
13         $\alpha_size \leftarrow p_{23}$
14         $\beta_size \leftarrow p_{24}$
15   **if** $\eta_size > 0$ **then** SORT(η);
16   **else break**
17     $\alpha \leftarrow$ RANDOM_EXTRACT(k)
18     $\beta \leftarrow \{0, \dots, k-1\} \setminus \alpha$
19     $\theta_sizes_i \leftarrow 0$
20   **foreach** $comb$ **in** ALLCOMB(α) **do**
21       $\boldsymbol{\mu_2} \leftarrow \boldsymbol{\mu_2} \oplus \sum_{z\in comb} \boldsymbol{v_1}\ \boldsymbol{z}$
22       $\theta_i \leftarrow \theta_i \cup \langle comb, \boldsymbol{\mu_{2[1:\lambda_1]}}, \boldsymbol{\mu_{2[\lambda_1+1:r_1]}} \rangle$
23       $\theta_sizes_i \leftarrow \theta_sizes_i + 1$
24   SORT(θ_i)
25   **foreach** $comb$ **in** ALLCOMB(β) **do**
26   **if** $i = 1$ **then** $\boldsymbol{\sigma_2} \leftarrow \widehat{\boldsymbol{s}}_1$
27       $\boldsymbol{\sigma_2} \leftarrow \boldsymbol{\sigma_2} \oplus \sum_{z\in comb} \boldsymbol{v_1}\ \boldsymbol{z}$
28       $\theta_{aux} \leftarrow \theta_{aux} \cup \langle comb, \boldsymbol{\sigma_{2[1:\lambda_1]}}, \boldsymbol{\sigma_{2[\lambda_1+1:r_1]}} \rangle$
29       $\theta_{aux}_size \leftarrow \theta_{aux}_size + 1$
30   **foreach** $\langle \beta^{(2)}, \boldsymbol{\sigma_{2[1:\lambda_1]}}, \boldsymbol{\sigma_{2[\lambda_1+1:r_1]}} \rangle$ **in** θ_{aux} **do**
31       $\langle left^{(2)}, right^2 \rangle \leftarrow$ FIND_COLLISION$(\boldsymbol{\sigma_{2[1:\lambda_1]}}, \theta_i)$
32   **if** $left^{(2)} \ne -1 \wedge right^{(2)} \ne -1$ **then**
33   **foreach** $\langle \alpha^{(2)}, \boldsymbol{\mu_{2[1:\lambda_1]}}, \boldsymbol{\mu_{2[\lambda_1+1:r_1]}} \rangle$ **in** $\theta_{i[left^{(2)}, right^{(2)}]}$ **do**
34   **if** HW$(\boldsymbol{\sigma_{2[\lambda_1+1:r_1]}} \oplus \boldsymbol{\mu_{2[\lambda_1+1:r_1]}}) = w_1/2$ **then**
35             $\delta^{(1)} \leftarrow (\alpha^{(2)} \cup \beta^{(2)})$
36   **if** $i = 0$ **then**
37               $\boldsymbol{\sigma_1} \leftarrow \widehat{\boldsymbol{s}}_2$
38   **foreach** ind **in** $\delta^{(1)}$ **do**
39                 $\boldsymbol{\sigma_1} \leftarrow \boldsymbol{\sigma_1} \oplus \boldsymbol{v_2}\ \mathbf{ind}$
40               $\eta \leftarrow \eta \cup \langle \delta^{(1)}, \boldsymbol{\sigma_{1[1:\lambda_2]}}, \boldsymbol{\sigma_{1[\lambda_2+1:r_2]}} \rangle$
41               $\eta_size \leftarrow \eta_size + 1$
42   **else**
43   **foreach** ind **in** $\delta^{(1)}$ **do**
44                 $\boldsymbol{\sigma_1} \leftarrow \boldsymbol{\sigma_1} \oplus \boldsymbol{v_2}\ \mathbf{ind}$
45               $\langle left^{(1)}, right^{(1)} \rangle \leftarrow$ FIND_COLLISION$(\boldsymbol{\sigma_{1[1:\lambda_2]}}, \eta)$
46   **if** $left^{(1)} \ne -1 \wedge right^{(1)} \ne -1$ **then**
47   **foreach** $\langle \alpha^{(1)}, \boldsymbol{\mu_{1[1:\lambda_2]}}, \boldsymbol{\mu_{1[\lambda_2:r_2]}} \rangle$ **in** $\eta_{[left^{(1)}, \wedge right^{(1)}]}$ **do**
48   **if** HW$(\boldsymbol{\sigma_{1[\lambda_2:r_2]}} \oplus \boldsymbol{\mu_{1[\lambda_2:r_2]}}) = w_2$ **then**
49                     $\delta^{(0)} \leftarrow (\alpha^{(1)} \cup \delta^{(1)}) \setminus (\alpha^{(1)} \cap \delta^{(1)})$
50   **if** SIZE$(\delta^{(0)}) = p$ **then**
51   **foreach** ind **in** $\alpha^{(1)}$ **do**
52                         $\boldsymbol{v} \leftarrow \boldsymbol{v} \oplus \boldsymbol{v_1}\ \mathbf{ind}$
53                       $\boldsymbol{z} \leftarrow \boldsymbol{v}_{[\lambda_1+1:r_1]} \oplus \boldsymbol{\sigma_{2[\lambda_1+1:r_1]}} \oplus \boldsymbol{\mu_{2[\lambda_1+1:r_1]}}$
54   **if** HAMMING_WEIGHT$(\boldsymbol{z}) = w_1$ **then**
55                         $\widehat{\boldsymbol{e}} \leftarrow [\boldsymbol{0}_{1\times\lambda_1} \quad \boldsymbol{z} \quad \boldsymbol{0}_{1\times\lambda_2} \quad \boldsymbol{\mu_{1[\lambda_2+1:r_2]}} \oplus \boldsymbol{\sigma_{1[\lambda_2+1:r_2]}} \quad \boldsymbol{0}_{1\times\mathbf{k}}]$
56   **foreach** k **in** $\delta^{(0)}$ **do**
57                           $\widehat{\boldsymbol{e}}_{\boldsymbol{k+r}} \leftarrow 1$
58                         $\boldsymbol{e} \leftarrow$ ERROR_RECONSTRUCTION$(\widehat{\boldsymbol{e}}, \chi)$
59   **return** $\boldsymbol{e}$
60   NEXT_COMB$(\beta, \lfloor \frac{k}{2} \rfloor, k-1)$
61 **until** HW$(\boldsymbol{e}) = w$

3.3. Implementation techniques

In this section we are going to discuss some techniques that have been taken in consideration for implementing the ISD algorithms. The implementation of the algorithms has been done in the C programmming language for having a low level control over the data structures to be able to optimize the algorithms.

3.3.1. Representation of the bit matrices and vectors

As we have seen, in the ISD algorithms we need to work with binary matrices and binary vectors, but there is no data type available for working directly with the bits. Therefore, uint64_t, the standard type of the ISO-C99, have been used for dealing with bits: each of these uint64_t can hold an integer up to $2^{64} - 1$ but we can use them for representing 64 bits at a time. For representing a binary vector of size n we can declare an array of uint64_t of size equal to $\lfloor \frac{n-1}{64} \rfloor + 1$, since each element of the array holds one unsigned integer representing 64 bits. If the size of the vector is a multiple of 64 all the bits of the unsigned integer will be used, otherwise if $n \mod 64 \neq 0$, $64 - (n \mod 64)$ bits of the last unsigned integer in the array will be extra-bits that will not be considered.
Let's see an example using uint8_t for representing a binary vector for simplicity. Suppose we have a binary vector $\boldsymbol{x}$ of lenght $n = 12$ with the following values:

$\boldsymbol{x}$	1	1	0	1	0	0	0	1	1	1	0	1

We want to represent this binary vector using an array of uint8_t: this array has to have size equal to $\lfloor \frac{n-1}{8} \rfloor + 1 = \lfloor \frac{12-1}{8} \rfloor + 1 = 2$ hence we need two uint8_t for representing this binary vector. The array will be as follows:

$\boldsymbol{A}$	1	1	0	1	0	0	0	1	1	1	0	1	x	x	x	x
	$\boldsymbol{A}[0]$								$\boldsymbol{A}[1]$							

Figure 3.15: Array of unsigned int for representing a binary vector

From Figure 3.15 we can see that the first 8 bits of $\boldsymbol{x}$ are saved in the first element of the array while the last 4 bits of $\boldsymbol{x}$ are saved on the first 4 bits of the second element of A. Since the size of the binary vector is not a perfect multiple of the size of the unsigned integer composing the array (i.e. $n \equiv 12 \equiv 4 \not\equiv 0 \mod 8$), there are some bits

on the last element of the array that are not significant. In the example we have $8 - (n \mod 8) = 8 - (12 \mod 8) = 4$ unused bits on the last element of A, the ones represented with an x in the figure. This example can be easily generalized with the case of uint64_t that have been used in the implementation. It is important to initialize the non significant bits of the last element of the array A equal to zero if we need to compute the Hamming weight of the vector: if these unused bits are left uninitialized with random values the resulting weight of the error will be incorrect since it counts extra bits equal to one not belonging to the vector. Instead, if we clear these bits setting them equal to zero, they will not be counted in the computation of the weight.
A binary matrix H of size $r \times n$ is instead represented as a bidimensional array of uint64_t. Each row of the matrix is a binary vector of size n and so it can be represented as before. Since now we have to store r binary vectors of size n, we use simply the same technique having a bidimensional array M with r rows and $\lfloor \frac{n-1}{64} \rfloor + 1$ columns. Let's see an example using again the uint8_t for the sake of simplicity. Suppose we have a binary matrix H with $r = 3$ rows and $n = 10$ columns:

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

and we want to represent it with a bidimensional array M with 3 rows and $\lfloor \frac{n-1}{8} \rfloor + 1 = \lfloor \frac{10-1}{8} \rfloor + 1 = 2$ uint8_t for each row. In Figure 3.16 we can see the resulting bidimensional array used for representing the binary matrix H.

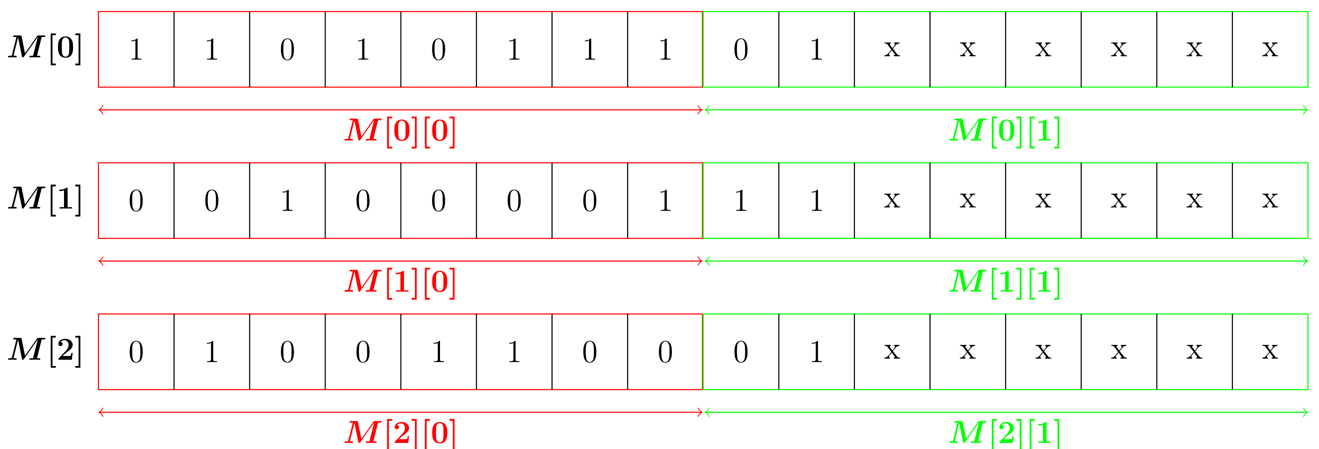


Figure 3.16: Bidimensional array of unsigned int for representing a binary matrix

As before, this example can be easily generalized for the case using the uint64_t. With this technique for representing the binary vectors and matrices we are able to optimize

as much as possible the space complexity used for storing them in the implementation of the ISD algorithms. The usual operations as reading,clearing or setting a specific bit of vector or a matrix has been implemented in a library with other useful instructions that manage the bits inside the unsigned integers.

3.3.2. Advanced Vector Extensions 2 instructions

The Advanced Vector Extensions 2 (AVX2) is a set of SIMD instructions designed by Intel that makes efficient the operations when we can work with parallel data. At [18] we can find a list of all the available AVX instructions with a description on how to use them. Having the matrices represented as discussed in the previous section, during the computation of the RREF we need to swap the rows of the matrix or computing the xor between two rows. These operations can be done with the AVX2 instructions. Since each row is composed by a certain number x of uint64_t, for swapping two rows i and j we have to swap the unsigned integer $H[i][z]$ with the unsigned integer $H[j][z]$ $\forall 0 \leq z < x$. Instead of performing one swap at a time we can use the AVX2 instructions for swapping a vector long 256 bits at a time with a single instruction. This results in doing 4 swaps with one instruction instead of a single one. To implement this swap we need to pay attention on the number x of uint64_t composing a row since we can swap a vector long 256 bits only if we have x greater or equal than 4 (4*64bits = 256 bits). Otherwise we can swap a vector long 128 bits if $x = 2$ (2*64 = 128 bits) doubling the swaps or perform a normal swap if $x = 1$. The intermediate situations can be obtained as the combinations of these: for example if $x = 7$ we have 7*64 = 448 bits on each row and so for swapping two rows we can first swap a vector long 256 bits, then a vector long 128 bits and finally a normal swap on a vector of 64 bits. Doing this we have performed 3 operations for swapping two rows composed by 7 uint64_t instead of 7 operations.
When we have to xor two rows and save the result in one of the two we can apply the same reasoning doing less instructions thanks to the AVX2.
The AVX2 instructions has been used even for the computation of the Hamming weight of a vector as explained in the next section.

3.3.3. Hamming Weight computation

For computing the Hamming weight of a binary vector two techniques has been taken in consideration: the Brian Kernighan's solution to count the set bits in an unsigned integer based on the work by Peter Wegner in [33] and the procedures using the AVX2 instructions described in [22] by Wojciech Muła, Nathan Kurz and Daniel Lemire.
The Brian Kernighan's procedure has a time complexity equal to the set bits on an

unsigned integer instead of a time equal to all the bits of the unsigned integer as the naive method. If an unsigned integer has only three bits equal to 1 the loop of the kernighan's algorithm is going to run only three times instead of checking all of its bits. In a nutshell, the bitwise-and between the input integer x and $x - 1$ is computed incrementing the weight count and assigning $x = x \wedge x - 1$ until $x > 0$. The bitwise-and operation is done for setting to zero the rightmost bit of x and proceed to count the other bit set to 1 at its left. We can see the generalization for computing the Hamming weight of a binary error vector represented as an array of uint64_t in the following algorithm.

Algorithm 3.3.1: Brian Kernighan's test for computing the Hamming weight

Input: $\boldsymbol{v}$: array of size n made by unsigned integers representing a binary vector
Output: w: number of bits set to one in $\boldsymbol{v}$

```
1 w ← 0
2 i ← 0
3 while i < n do
4 |   x ← v[i]
5 |   while x > 0 do
6 |   |   x = x ∧ (x − 1)
7 |   |   w ← w + 1
8 return w
```

The complexity of this algorithm in the worst case is $\mathcal{O}(n)$ when all the bits of the integer are set but in average, if we are considering a random binary vector, we have $\mathcal{O}(\frac{n}{2})$ since in a random vector the half of its bits are set to 1 in average.

The other technique is based on a vectorized approach using SIMD instructions and in the work [22] is showed that can be twice as fast as using the dedicated instructions on Intel processors. The technique is a vectorized version of the Harvey-Seal procedure presented in [32] and it has been implemented using the AVX instructions.

3.3.4. Multithreading

The implementation of the ISD algorithms has been done using multiple threads for trying to find a target error in the least possible time. Each ISD starts an iteration picking a random permutation for computing a systematic form of the parity-check matrix and then, it applies the specific technique for searching the error vector based on which ISD we want to use. Knowing this, we can parallelize the iterations of each ISD algorithm: we can start x threads for finding the target error. Each thread picks a permutation, compute a systematic form and try to find an error, therefore, we carry on x parallel runs for finding faster the target error. Each thread needs to have its own data structures separated from the others and one important thing is that, since each thread picks random permutations,

they need to have different initializations of the pseudo-random number generators using different initial seeds.
The PRNG used in this implementation is the xoshiro256++ based on the work in [11] by David Blackman and Sebastiano Vigna and more info about the implementation can be found at https://prng.di.unimi.it. This PNRG is initialized with four unsigned integers of 64 bits, and for each thread, we need to use different quadruples, otherwise the threads will do the same computations resulting in no advantage in finding the error in less time with parallelization.
The multithreaded implementation has been done with OpenMP(Open Multi-Processing): an application programming interface that supports shared-memory multithreading programming constisting in a set of compiler directives, library routines and environment variables. OpenMP support is available in our compiler of choice, GCC.

3.3.5. Hashtables

As we have already discussed, in Stern, Ball-Collision Decoding and Finiasz-Sendrier algorithms, we have used both lists and hashtables as collision structures for understanding which structure behaves better. Both contain pairs made by a binary error vector and an array of indices. When we need to insert in the hashtable a new pair, we have to compute the hash function having in input the binary vector represented as an array of unsigned integers. The hash function being used is an integer hash function called 64 bit mixing designed by Thomas Wang at [31]. This function takes in input an integer and return the index of the hash table where the new element needs to be placed. We need to insert the elements based on their value of the binary vector and for producing a single integer to pass to the hash function we compute the xor between all the unsigned integers composing the array that represents the binary vector. This can be done without loss of generality since the vectors taken in consideration can be considered random.
On the implementation of the ISD algorithms it is possible to choose which collision structure to use between list and hashtable by simply defining a preprocessor directive called "LIST" if we want to use the list or "HASH_TABLE" otherwise. In the next chapter we will see the testing results comparison between the implementation with the list and the implementation with the hash table as collision structure. With other define directives we can also switch the RREF methods to use: if the "CUSTOM_RREF" directive is defined the RREF with the reusing pivots optimizations 2.1.2 or the partial RREF optimized 2.1.4 depending on the algorithm we want to run is used. If no directive refering to the RREF is defined the implementation of the M4RI algorithm 2.1.5 is used.

4 | Experimental Evaluation

In this section we are going to compare the different RREF procedures presented in Section 2.1, the different binary range searches based on the algorithms in Section 2.2 and the two sorting algorithms in Section 2.4 to understand which variant is the fastest for being used in the ISD implementation. Then, we are going to see the results collected testing the different implementation of the ISD algorithms with two different types of tests for a concrete evaluation of them. The first family of tests is composed by instances of the the so called "Syndrome Decoding" that we can find at [4]. In these tests the codes taken in consideration have a rate $R = 0.5$ and the weight w of the target error to be found is close to the Gilbert-Varshamov distance. In the second family of tests the "McEliece-Goppa Syndrome Decoding" instances are used always from [4]. Here the challenges to solve are the syndrome decoding problem for a random linear code with rate $R = 0.8$ and an error weight $w = \frac{(1-R)n}{\log(n)}$ corresponding on instances of the SD problem on which the security of the McElice cryptosystem relies.
Different sizes of the codes have been used for both the cases and since the parameters of the tests change, we need to find the optimal parameters of the ISD algorithms during different runs. We have used two estimators for retrieving the optimal parameters of the ISD algorithms depending on the code properties in input: the Ledatools estimator presented in [5] by Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi and Paolo Santini and the Syndrome-Decoding estimator presented in [13] by Andre Esser and Emanuele Bellini. Then, after measuring the cost of a single iteration for each ISD algorithm, we will present the computational costs even for test cases with high parameters multiplying the cost of the single iteration with the estimated iterations for finding a target vector (the reverse of the success probability seen in the previous chapter).

4.1. RREF analysis

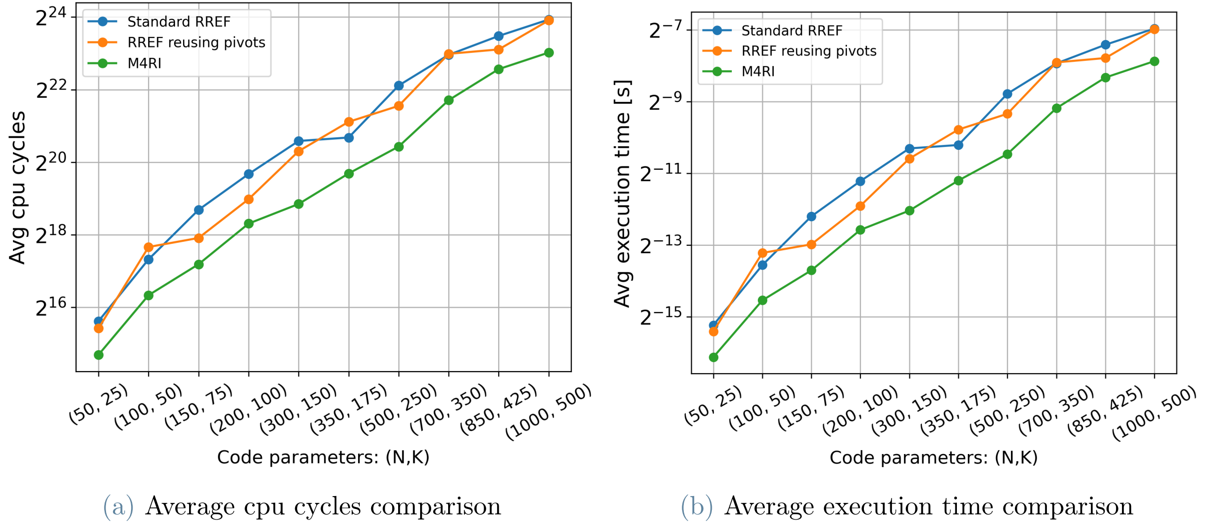
In this section we are going to compare the different procedures presented in Section 2.1 for understanding which one is the most efficient for computing a reduced row echelon form or a partial reduced row echelon form. We have taken a set of challenges having

different dimensions of the parity-check matrix to stress the procedures with increasing size of H. The matrices tested have rate $R = 0.5$ and their sizes go from the minimum $r \times n = 25 \times 50$ to the maximum 500×1000. For each code we call 30 times the RREF procedure in consideration and we save the average value of the execution time and the average value of the complexity cost measured in cpu cycles spent for computing a RREF. We need to compute the average value of a certain number of runs since the algorithm is probabilistic: each procedure starts with taking a random permutation and then try to find a correct systematic form, so, certain runs can be very fast if they pick lucky permutations, otherwise they can be very slow with other permutations (for example, the ones that don't permit to find a correct systematic form). Therefore, it is important to measure the average cost of different runs.

On the first test we have compared the implementation of the standard RREF (2.1.1), the RREF with reusing pivots optimization (2.1.2) and the M4RI algorithm (2.1.5) implemented in [30] inside the m4ri_light project based on the M4RI library [2] to produce a full systematic form. In the Figure 4.1 we can see the testing results of this test: in the first plot the average cost complexity is reported while in the second plot we can see the average execution time in seconds, both in function of different code properties (n, k) and in a logarithmic scale. We can easily notice that with increasing size of the parity-check matrix the cost of each RREF procedure increases as well. The fastest method for computing a full systematic form has been resulted in the M4RI one, available publicly in the library.

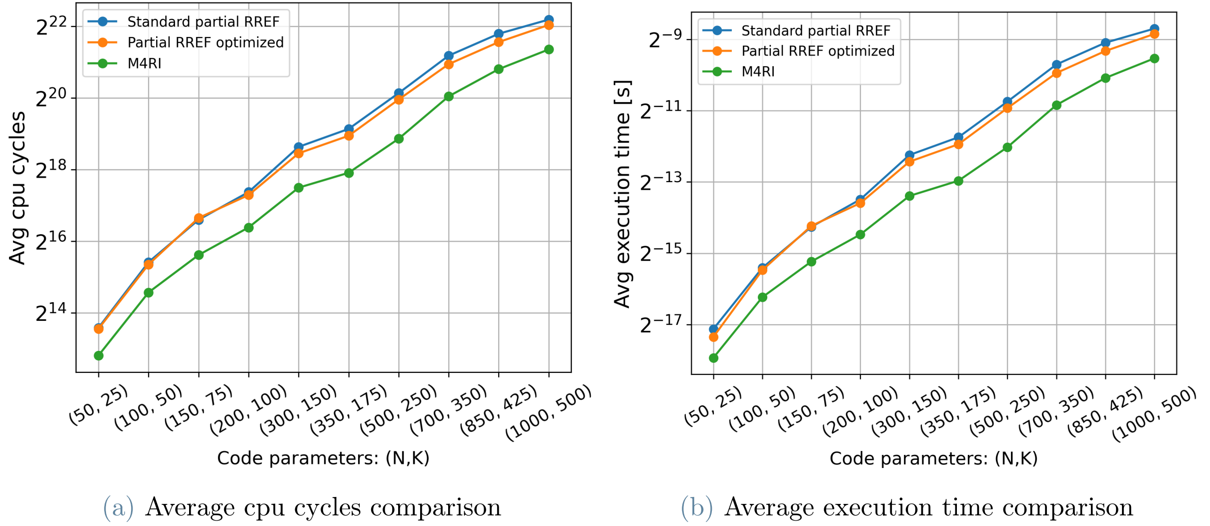
On the second test we have compared the implementation of the standard partial RREF (2.1.3), the partial RREF optimized (2.1.4) and the M4RI algorithm (2.1.5) to produce a partial systematic form. In the Figure 4.2 we can see the testing results reporting the average cpu cycles and the average execution time in seconds as in the previous test. The extra parameter ℓ, needed to compute a partial systematic form, has been considered fixed to the value 8 during the various tests without loss of generality. Taking different values of ℓ changes the nominal values of the testing but the differences between the methods are exactly the same. Since we are interested in understanding which one is the most efficient we can choose the parameter we want. Comparing the different plots, we can notice that the partial RREF is less costly than the full one for the same tests as we expect, since the identity matrix to produce has smaller dimension. We can even see that thanks to the optimizations introduced in Section 2.1.4 the partial RREF optimized is faster then the standard procedure, but even here, the fastest method has resulted in the M4RI one. From this conclusion, in the implementation of the ISD algorithms the M4RI procedure is used for computing both the full reduced row echelon form and the partial reduced row echelon form: all the tests reported in the following sections use this method. It is still

possible to use the RREF with reusing pivots and the partial RREF optimized methods inside the implementation declaring the preprocessor directive "CUSTOM_RREF": if this directive is omitted the M4RI procedure is used for both the full and the partial RREF.



(a) Average cpu cycles comparison (b) Average execution time comparison

Figure 4.1: Comparison between different RREF procedures



(a) Average cpu cycles comparison (b) Average execution time comparison

Figure 4.2: Comparison between different partial RREF procedures

4.2. Binary range search variants analysis

Here, we are going to test different find collision algorithms composed by the binary range searches based on the procedures seen in Section 2.2. For understanding which

search performs best we have tested the Stern algorithm 3.2.4 with some family of codes changing each time the method for finding the collisions in the list. To do that it is important to initialize the Stern algorithm, for each method to test, with the same initial seeds for the PNRG. With the same initial seeds we are sure that each run of the Stern algorithm will execute exactly the same operations as the other runs except for the find collision part that is the method we want to test. So, the final results will change based only to the binary range search variant currently used. Taking the final execution time for finding a target error with the Stern algorithm for each variant of the binary range search, we can conclude which is the fastest that will be used for all the other ISD algorithms. If we take different runs with different initial seeds the permutations will be different and the results will not be comparable in term of the binary range search algorithm since other factors influence the execution. Four codes (n, k, w) has been considered for this test : (230,115,30), (250,125,32), (270,135, 34) and (280,140,35) to stress the procedures with increasing size of the list. We can see the results of the test on the following plot: it reports the percentage speedup in the execution time for each variant of the binary range search with respect to the one measured for the standard binary range search.

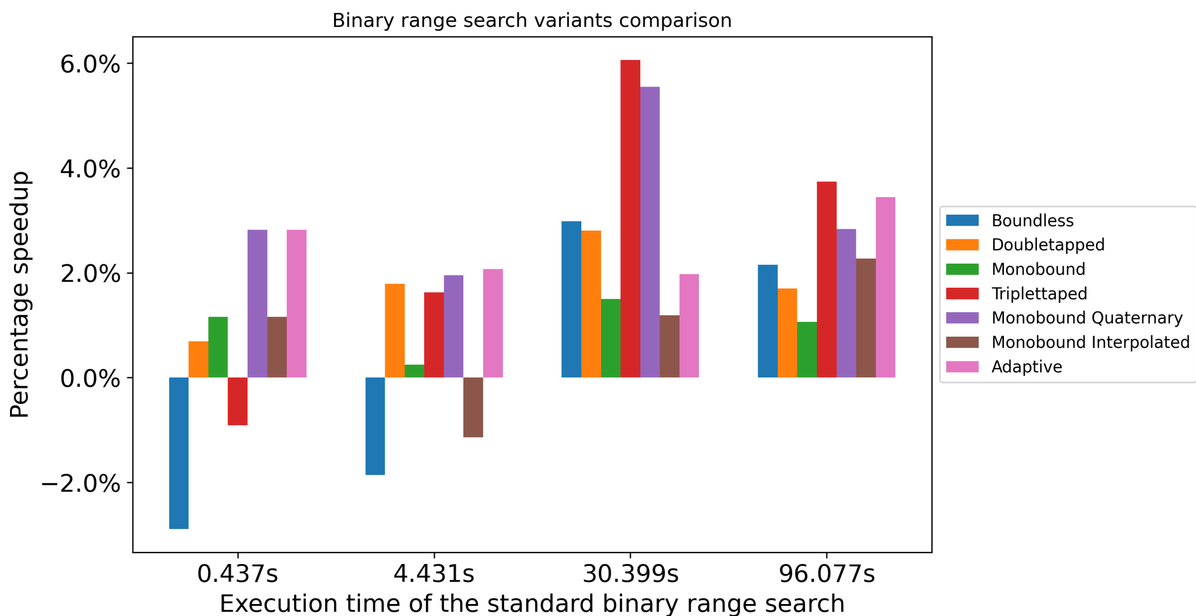


Figure 4.3: Comparison between binary range search variants: percentage speedup of the execution time with respect to the one of the standard binary range search

Since the binary range search variants are very similar to each other, the total execution times to compare are close too. To see better the differences between these times, in the Figure 4.3 we have plotted the speedup gained by each variant: a positive percentage

x% means that we have a speedup of x% compared to the execution time obtained by the standard binary range search, while a negative percentage indicates a slowdown. The execution time in seconds of the standard binary range search is reported on the x-axis. The first test is not really meaningful since we have obtained execution times less than 0.5 seconds for all the variants, therefore the time differences are negligible. The last two tests are the most significant since the execution times are big and the procedures are more stressed. We can notice that for these two tests all the variants have better performance than the standard binary range search since they all gain a speedup. The fastest variant is resulted in the tripletapped binary range search since it has the maximum speedup (minimum execution time) in the last two tests that are the most significant. The first test in the plot has been done with the code (250,125,32), the second with (230,115,30), the third with (270,135,34) and the last with (280,140,35): in the plot we have ordered the results based on the values of the execution time of the standard binary search in ascending order. The test has been done with an initial seed equal to (143,12,542,234); taking another initial seed changes the nominal values of the times but the differences between the methods remain the same. As already said, the most important thing is to use the same initial seed for each search tested otherwise the time differences obtained in the results don't depend only upon the search, as we want, but upon other factors. On the implementation of the ISD algorithms and in all the following tests the tripletapped binary range search has been used as the find collision algorithm.

4.3. Sorting algorithms analysis

Here, we are going to test the Quicksort algorithm 2.4.2 and the Djbsort algorithm 2.4.3 to see which one performs better with lists made of binary vectors. The test is done exactly with the same guidelines as the one for the binary range searches: for a family of codes we run the Stern algorithm, one time with the Quicksort algorithm and one time with the Djbsort algorithm, for sorting the list based on the values of the binary vectors inside it. Even here, the initial seeds for different runs must be the same for having a direct comparison on the sorting algorithms. The codes that have been testing are the following: (230,115,30), (250,125,32), (270,135, 34) and (300, 150, 38) to stress the sorting algorithms with different list size.
Since the difference between the execution times computed with the two different sorting algorithm is very little, in the Figure 4.4 is reported the percentage speedup of the Quicksort algorithm compared to the execution time of the Stern algorithm with the Djbsort. All the runs with the Djbsort are slower than the ones with the Quicksort: for this reason in the plot we have the Djbsort time on the x-axis while on the y-axis we report the

speedup gained by the Quicksort algorithm.

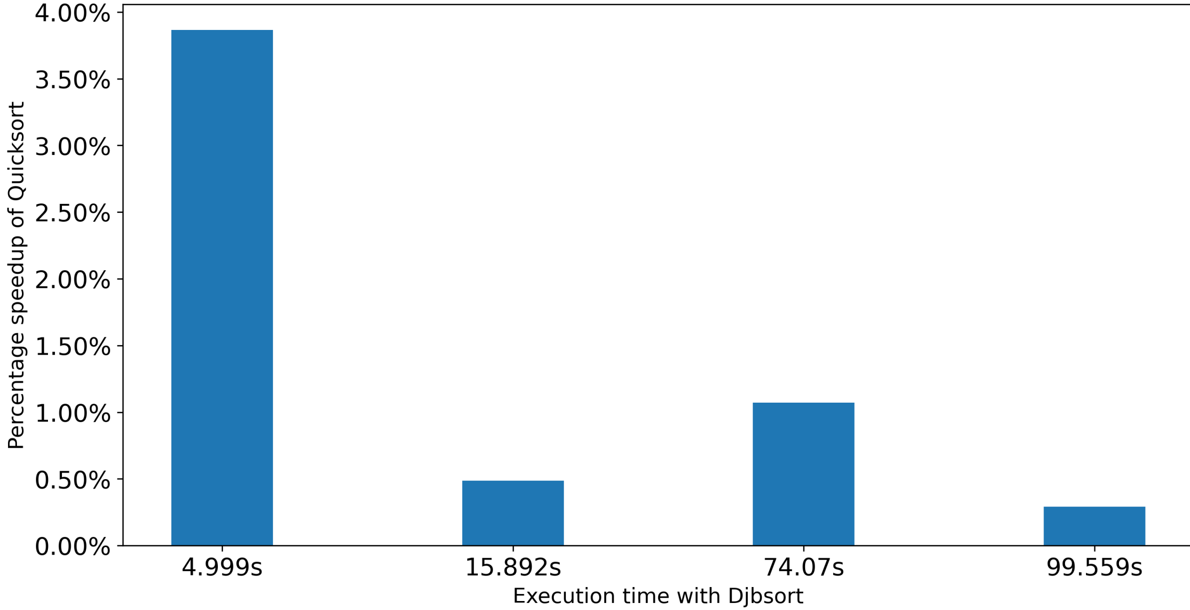


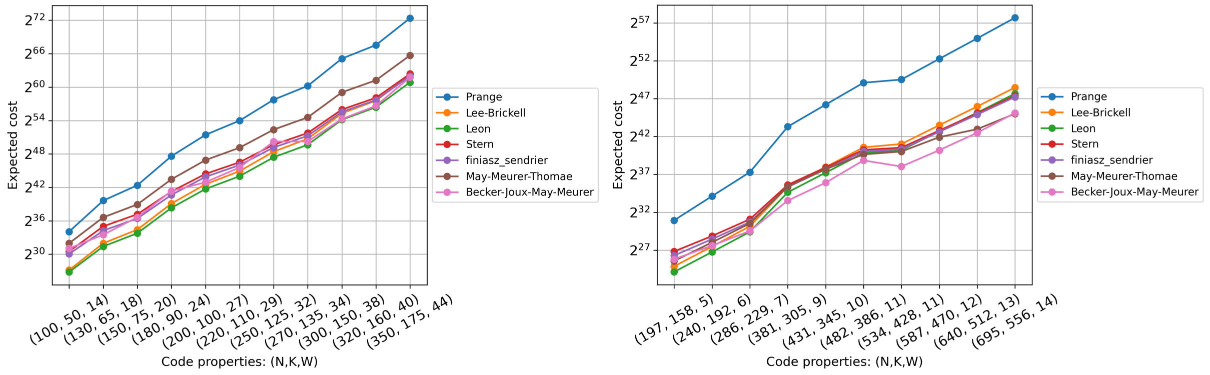
Figure 4.4: Comparison between sorting algorithms: percentage speedup of the execution time of Quicksort with respect to the one of the Djbsort

As we can see the two sorting algorithm behaves very similar, but the Quicksort algorithm performs best in each test even if the difference is very little. Therefore, on the implementation of the ISD the Quicksort algorithm has been chosen for sorting the lists in the ISD algorithms who use them.

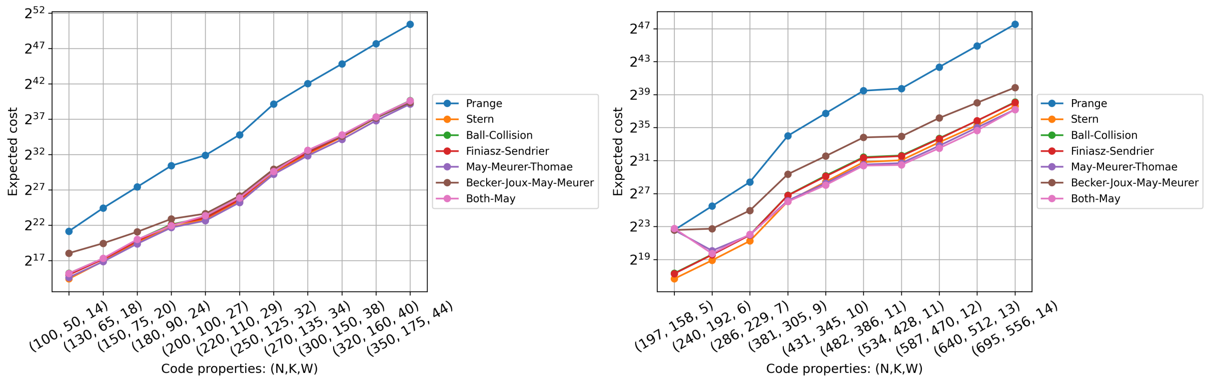
4.4. Estimators to find the optimal ISD parameters

In the next section we are going to test the ISD algorithms with tests having different code properties therefore, for each code, we need to properly tune the extra parameters used in the various ISDs, like for example p in Lee-Brickell or ℓ in Leon. The two estimators used are the Ledatools [5] and the Syndrome Decoding Estimator [13]. The estimates of the Ledatools relative to the May-Meurer-Thomae and the Becker-Joux-May-Meurer presented two flaws that have been corrected. In the May-Meurer-Thomae estimate the ℓ_2 parameter was set equal to $\frac{\binom{(k+\ell)/2}{p/4}}{p/4}$ resulting in an error. The value has been substituted with the correct one, $\log\left(\frac{\binom{(k+\ell)/2}{p/4}}{\binom{p}{p/2}}\right)$, since the correct value is taken applying the logarithm while the denominator must be $\binom{p}{p/2}$ because there exist only $\binom{p}{p/2}$ representations of the solutions as pointed out in the appendix of the work by A.Esser and E.Bellini [13]. The estimate of the Becker-Joux-May-Meurer has been rewritten from scratch following the

complexity pointed out in [5] correcting the term relative to the building of the list at layer 2 where we have binary vectors long $\ell_1 - \ell_2$ and not long ℓ_2 as reported in the paper. The Ledatools estimator returns the optimal parameters and the expected cost of solving a syndrome decoding problem for Prange, Lee-Brickell, Leon, Stern, Finiasz-Sendrier, May-Meurer-Thomae and Becker-Joux-May-Meurer algorithms while the Syndrome Decoding estimator for Prange, Stern, Ball-Collision Decoding, Finiasz-Sendrier, May-Meurer-Thomae, Becker-Joux-May-Meurer and Both-May.
In the next plots we can see the expected cost complexity for the codes that will be used in the next section for testing the ISD algorithms for both the estimators.



(a) Ledatools expected cost with syndrome tests (b) Ledatools expected cost with mceliece tests



(c) SD estimator expected cost with syndrome tests (d) SD estimator expected cost with mceliece tests

Figure 4.5: Expected computational cost returned by the Ledatools and the Syndrome Decoding estimator

We can notice from these plots that the expected cost retrieved by the Ledatools estimator is higher than the one returned by the Syndrome Decoding estimator because the Ledatools uses a logarithmic memory access cost model while the Syndrome Decoding estimator has been used with the default one. In the Appendix we can find the Table A.1

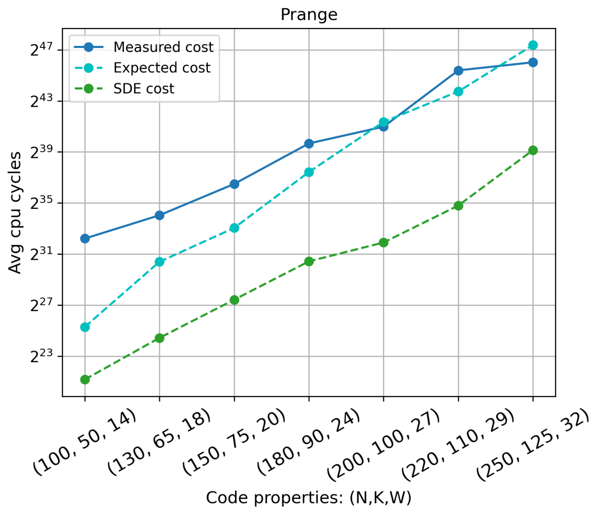
and the Table A.2 that report the optimal parameters returned by both the estimators for all the codes that have been tested: these optimal parameters will be used in the next section for the concrete evaluation of the ISD algorithms. The Lee-Brickell, Leon and Becker-Joux-May-Meurer algorithms have been tuned with the optimal parameters returned by the Ledatools, while the Stern, Ball-Collision Decoding, Finiasz-Sendrier, May-Meurer-Thomae and Both-May have been tuned with the Syndrome Decoding estimator. This choice has been done since the BJMM with the parameters returned by the Ledatools behaves better in practice than the one with the parameters returned by the Syndrome Decoding estimator, while Stern, Finiasz-Sendrier and MMT behave better with the parameters returned by this latter. For the remained algorithms there is no choice since they can be tuned only by one estimator as we can notice from the previously cited list.

4.5. Information Set Decoding algorithms testing

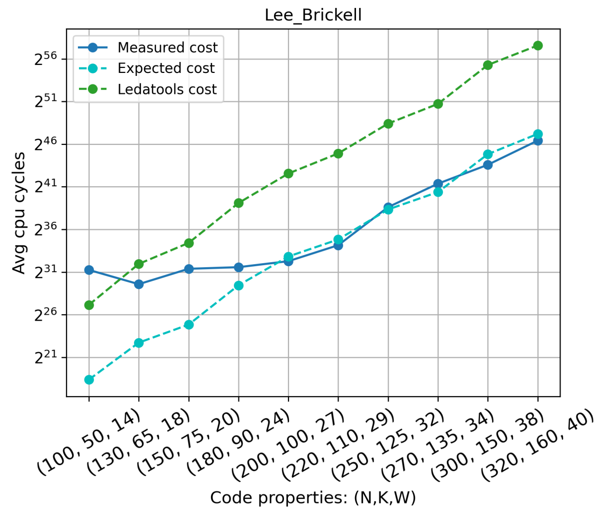
In this section we are going to test the implementation of all the ISD algorithms presented in Chapter 3. As we have already said in the introduction, we are going to use two family of tests: the first is made of random codes with rate $R = 0.5$ while the second is made of McEliece public key codes. Both tests are syndrome decoding problems where we need to find a target error with weigth w but the properties of the code used and the target weight of the error to be found are different. The procedure of finding a target error with an ISD is probabilistic since initially a random permutation is chosen to compute a systematic form, and depending on the resulted form, an error can be found or not, possibily recalling the RREF procedure and the search error function. Therefore, for measuring the cost of each ISD, we have executed 15 times each ISD for each code to test to extract a reasonable metric that takes into account the randomness part of the algorithm. For tests that take more than one minute to complete we execute them 5 times instead of 15.
The testing of the ISD algorithms has been carried out on a machine having 2 CPU sockets with 32 cores per socket and 2 threads per core resulting in a total of 64 physical cores allowing for a parallelization via 128 threads. The architecture of the CPU is $x86_64$ and the model name is AMD EPYC 7551 32-Core Processor with a clock speed equal to 2540.295 MHz. The machine is equipped with a DDR4 RAM memory of 500GB. The compiler that has been used is GCC 10.2.1 while CMAKE 3.18.4 has been used for building the project. The PNRG used is xoshiro256++ version 1.0.

4.5.1. Syndrome tests

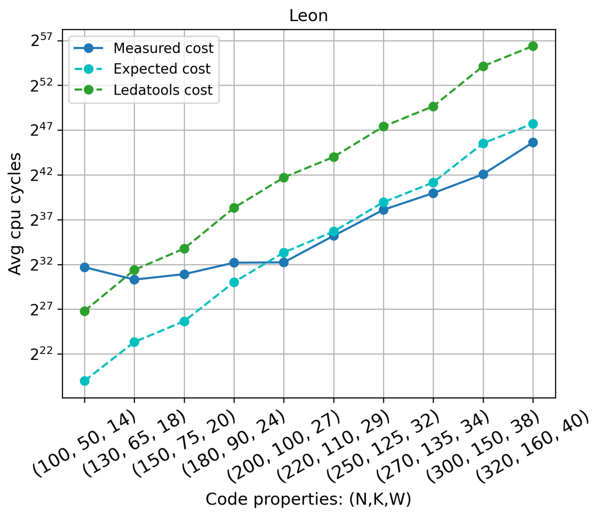
In these tests the binary parity-check matrix H has a number of rows equal to the half of the columns since the code rate R is equal to 0.5 and so, $n = 2k$. The tests in input are taken in the "Syndrome Decoding Problem" section at [4].
We have tested each ISD algorithm with a certain number of tests varying the code properties. The dimensions of the tests taken in consideration goes from $n = 100$ and $w = 14$ to $n = 350$ and $w = 44$. For each ISD algorithm is reported one plot showing the average cost complexity of the ISD measured by cpu cycles using a logarithmic scale. This plot includes even the cost returned by the estimator that has been used for tune the optimal parameters and the estimated cost complexity computed as the cost of a single iteration of the ISD algorithm times the number of expected iterations to find the target error ($\frac{1}{Pr_{succ}} c_{iter}$ as we have seen in the complexity formulas of the previous chapter). The cost of a single iteration of an ISD is measured as the average cost taken from 30 runs of the algorithm: for each test we execute the ISD algorithm only for one iteration, not worrying about finding the target error, but focusing on the cost of a single iteration (one RREF computation and one call to the search error part). Then, having the cost of a single iteration, we can estimate the total cost for finding an error vector with the desired weight multiplying it with the expected number of iterations relative to the ISD in use. Having this estimate we can see if the practical measured cost complexity is the one we expect: if yes, we can derive the total complexity of the ISD algorithms even with high dimensions tests that could be run for months before finding the target error.
As we can see in the following plots, for the tests with smaller dimensions ($n \leq 180$), the measured cost is much higher than the expected cost: this happens because the ISD algorithms find the error in less than one second for these tests, and since we are using 128 threads, the final cost takes into account the cost of each thread resulting in an overhead. In fact, with these small tests, the algorithm doesn't take any advantage using threads since the problems are easy and solvable almost instantly. For this reason, after the results with the syndrome tests, we will see the plots reporting the cost of the ISD algorithm for the small tests without using parallelization to have a proper comparison with the estimated cost. On the other hand, we can see that when the tests begin to be more complex (n $>$ 200), the measured complexity and the expected one ($\frac{1}{Pr_{succ}} c_{iter}$) are similar as we expected.
We can see the results taken from the evaluation of the ISD algorithms from Figure 4.6 to Figure 4.8.



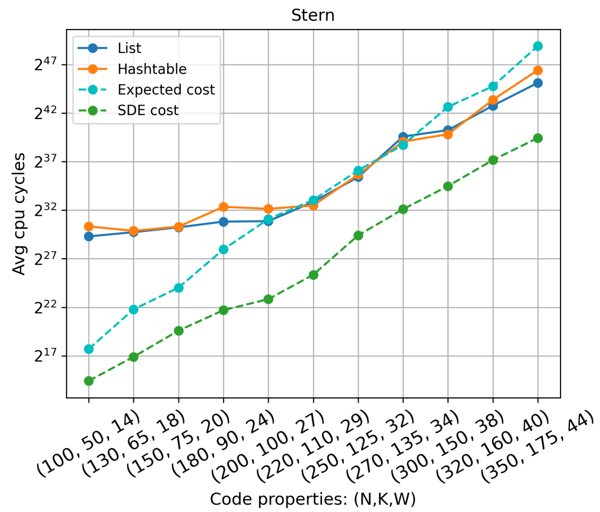
(a) Prange avg cpu cycles to find the target error



(b) Lee-Brickell avg cpu cycles to find the target error

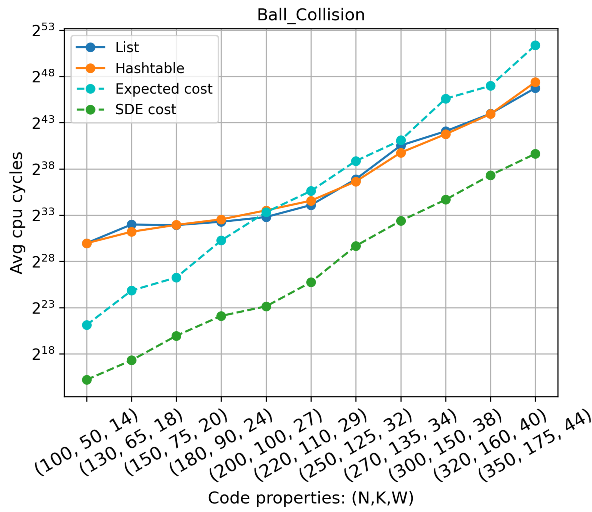


(c) Leon avg cpu cycles to find the target error

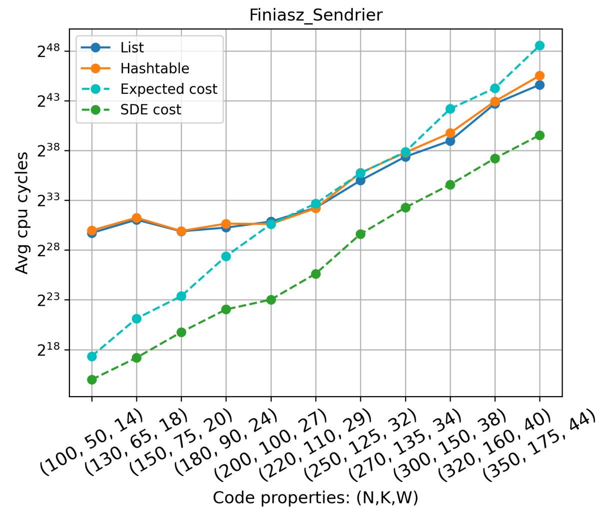


(d) Stern avg cpu cycles to find the target error

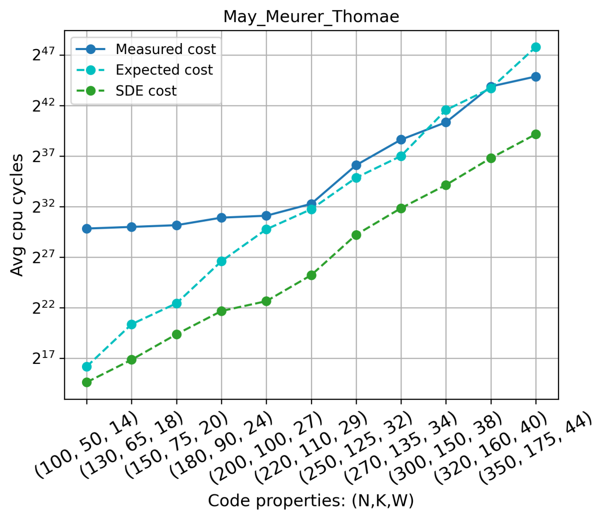
Figure 4.6: Prange, Lee-Brickell, Leon and Stern algorithms evaluation (syndrome tests)



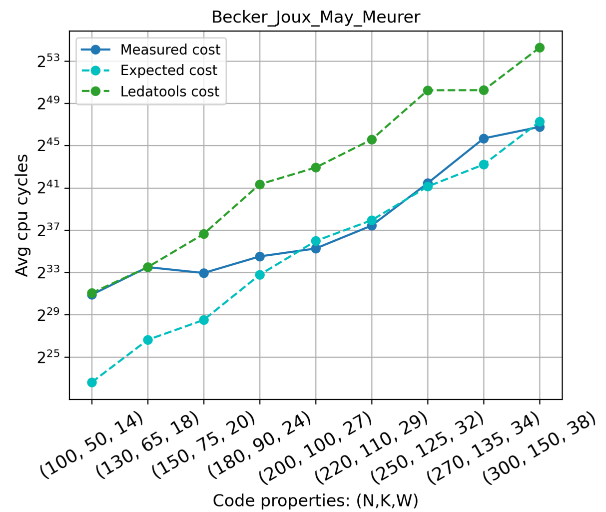
(a) Ball-Collision avg cpu cycles to find the target error



(b) Finiasz-Sendrier avg cpu cycles to find the target error

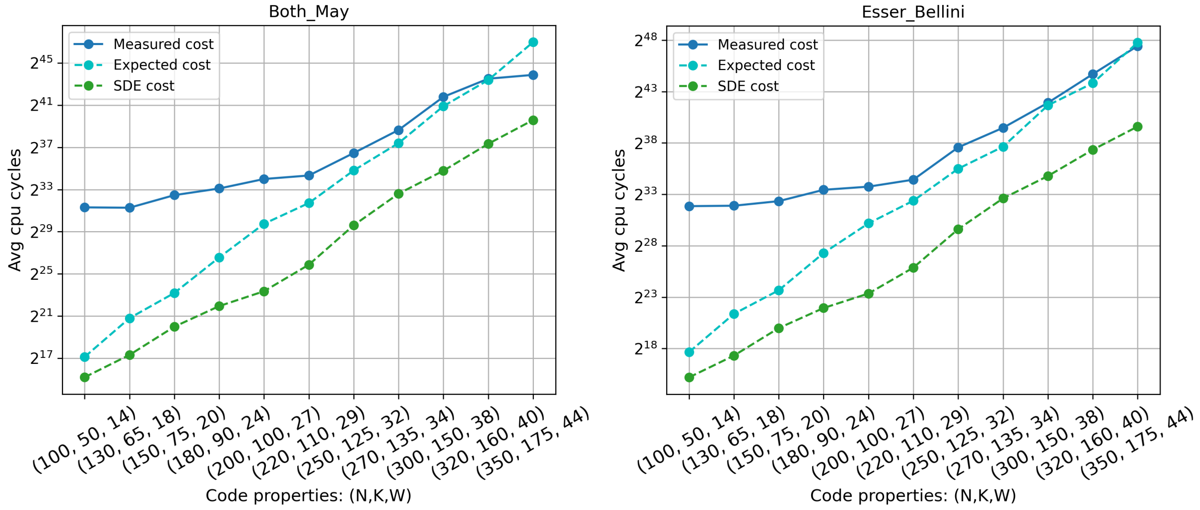


(c) May-Meurer-Thomae avg cpu cycles to find the target error



(d) BJMM avg cpu cycles to find the target error

Figure 4.7: Ball-Collision Decoding, Finiasz-Sendrier, May-Meurer-Thomae and Becker-Joux-May-Meurer algorithms evaluation (syndrome tests)



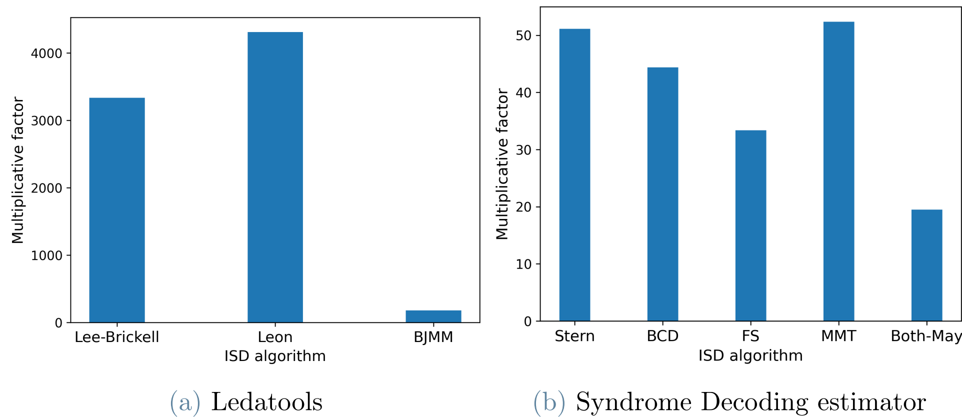
(a) Both-May avg cpu cycles to find the target error

(b) Esser-Bellini avg cpu cycles to find the target error

Figure 4.8: Both-May and Esser-Bellini algorithms evaluation (syndrome tests)

The Prange algorithm execution in the Figure 4.6 is not tested after the code with $n = 250$ since for finding an error with this code it takes more than four minutes in average. For the Stern, Ball-Collision Decoding and the Finiasz-Sendrier we can see from Figures 4.6 and 4.7, that both the execution with the list and the hashtable is reported for a direct comparison. For these algorithms there are no huge differences in the usage of the list or the hashtable, but with the last two tests, that are the complex ones, we can see that the implementation with the list behaves better even if the difference is little.
For all the syndrome tests taken in consideration, the optimal parameter z (the weight of the ℓ part of the error) of the Ball-Collision Decoding algorithm returned by the syndrome decoding estimator [13] is always zero: this means that the optimal configuration of the Ball-Collision coincides with the one of the Stern algorithm since having $z = 0$ is the same as using the Stern. As a consequence, we have tested the Ball-Collision Decoding algorithm with the minimum admissible weight z even if it's not the optimal one, $z = 2$, to obtain a different test with respect to the one of the Stern. For this reason the performance obtained by the Ball-Collision is worse than the one obtained by the Stern since it hasn't been used the optimal parameter: the optimal result for the Ball-Collision Decoding can be seen in the plot of the Stern algorithm.
As expected, between the cost returned by the proper estimator (the Ledatools for Leon, Lee-Brickell and BJMM and the Syndrome Decoding estimator for the other algorithms) and the cost measured directly testing the ISD algorithms, there is not a perfect match but there is a multiplicative factor between the two. This happens because the estimates

produced by both the estimators are in terms of abstract elementary operations while the measure done in cpu cycles is made by concrete operations. The multiplicative factor models how many elementary abstract operations the machine done for each clock cycle and it is substantially constant for each algorithm. We have quantified this factor for both the estimators considering the greatest code tested computing the ratio between the cost returned by the estimator and the measured cost, or viceversa if this latter is greater than the estimate.

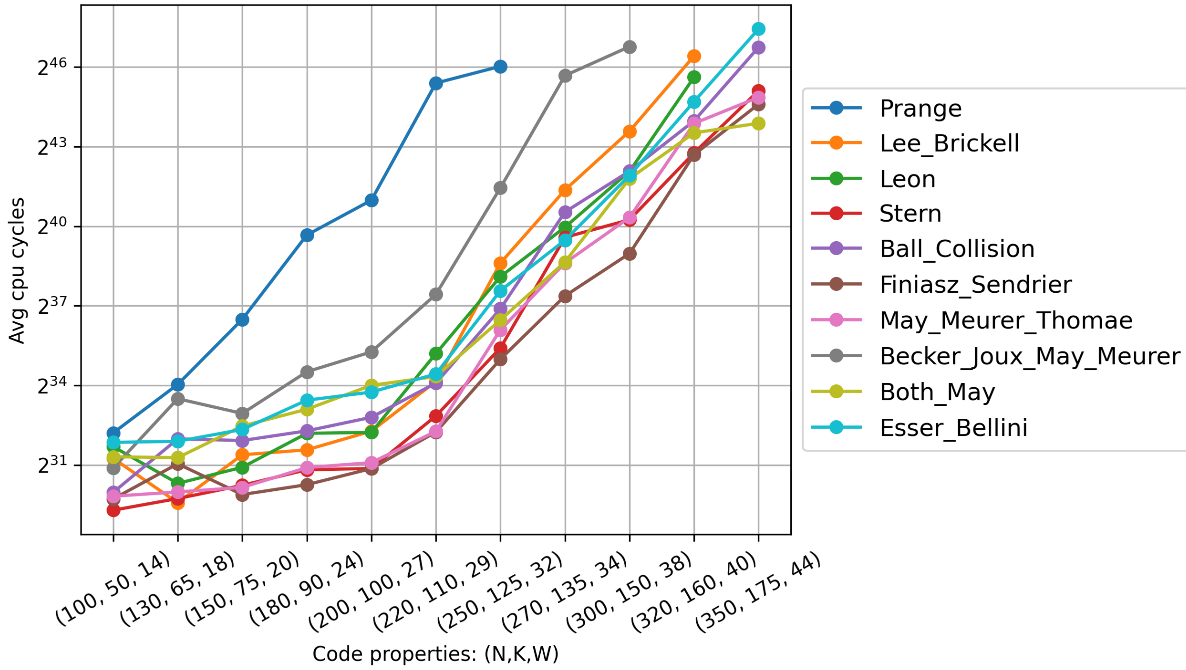


(a) Ledatools (b) Syndrome Decoding estimator

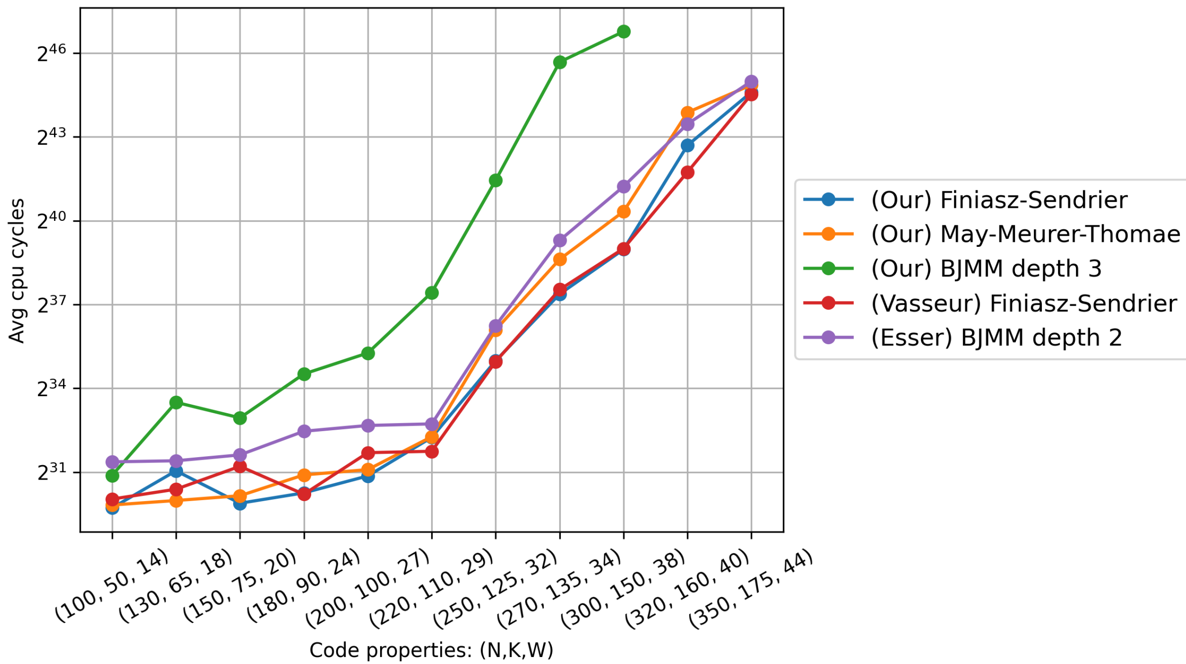
Figure 4.9: Multiplicative factor between the measured cost and the estimate returned by the estimators (syndrome tests)

For each algorithm the multiplicative factor is substantially constant between different runs but taking different algorithms it varies since the mix of operations is different between them: some algorithms have more calculations to do in one iteration or more accesses to the memory than other algorithms. Because of that the ratio between the abstract operations used in the estimators and the measured cpu cycles depends on the mix of the instructions of the current ISD in analysis. The CPU does more than one binary operation per clock cycle but it pays more a memory access with respect to the logarithmic access model if a cache miss happens. In the following page two plots are reported inside the Figure 4.10: one collects the results of all our ISD algorithms implementation for a direct comparison between them while the second compares our Finiasz-Sendrier with the one implemented by Vasseur [30] and our MMT and BJMM with the BJMM implemented by Esser,May and Zweydinger done in [14]. The tests for these two implementations are done on the same machine described before. The BJMM implemented by Esser, May and Zweydinger works with depth 2 different with our BJMM with depth 3. The testing of this implementation has been done using 128 outer threads to parallelize the permutations while the tree for searching the error is not parallelized. On the first plot for the Stern,

Ball-Collision Decoding and Finiasz-Sendrier algorithms only the results obtained using the list as collision structure is reported for not to overpopulate the plots.



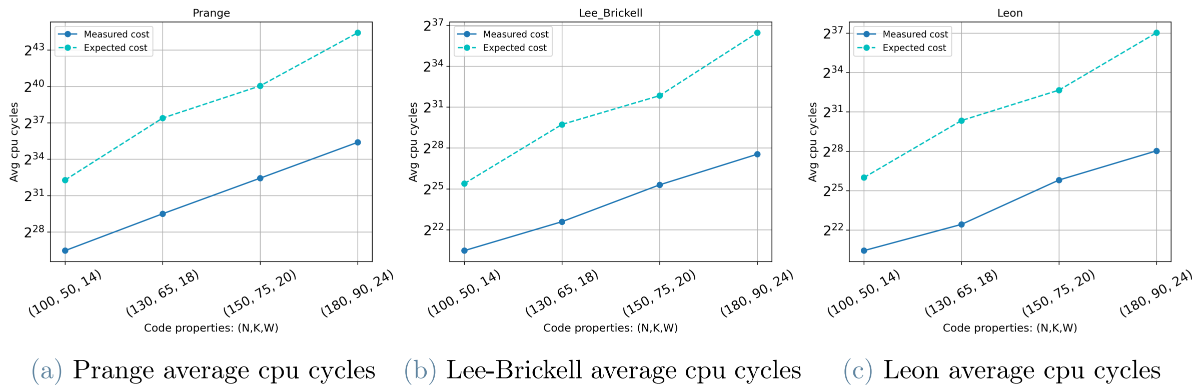
(a) Comparison between our ISD algorithms



(b) Comparison between our ISD with the Vasseur [30] and the Esser,May,Zweydinger [14] implementations

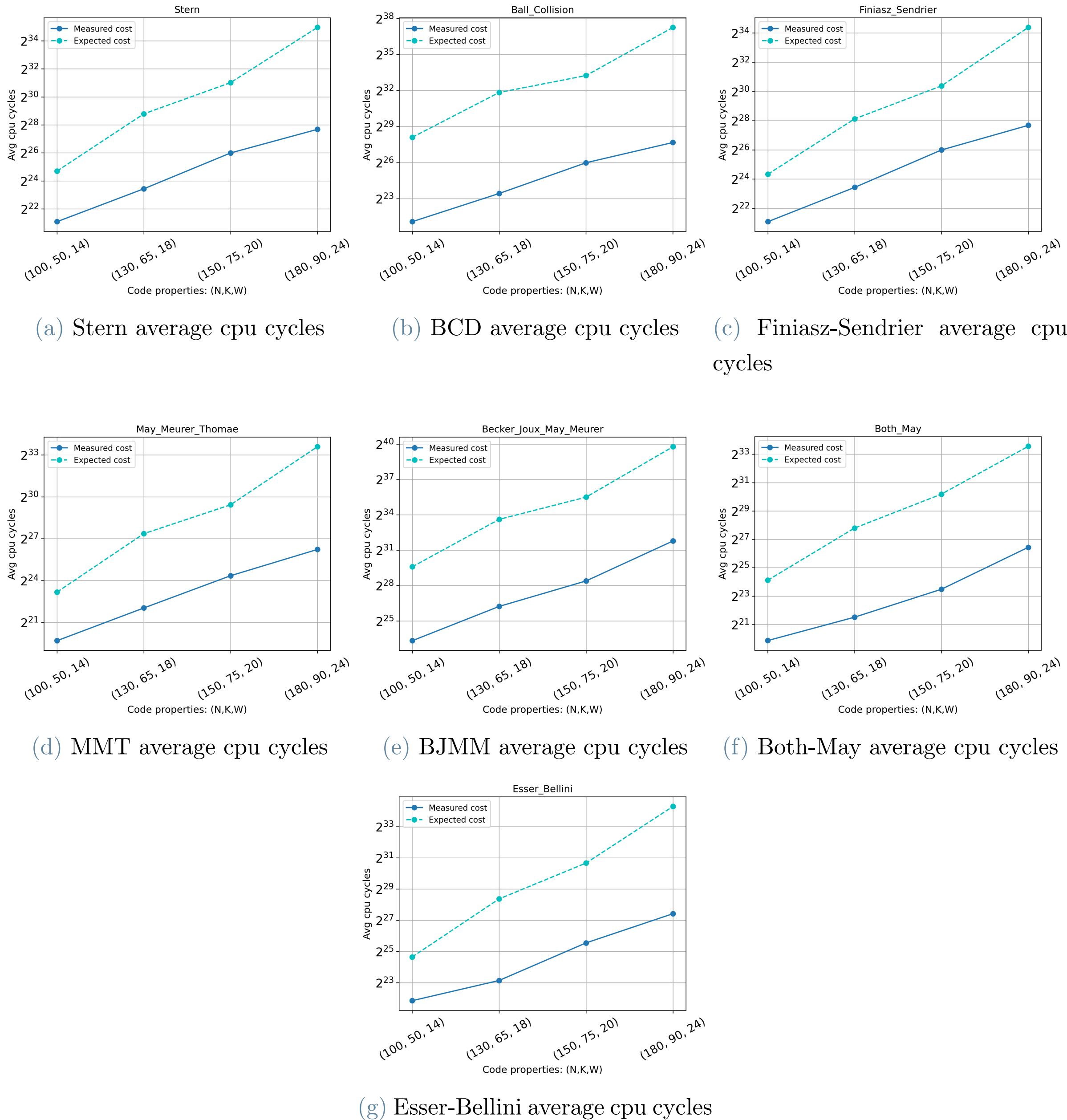
Figure 4.10: Computational cost comparison between all the ISD algorithms (syndrome tests)

From Figure 4.10 we can see that the fastest ISD algorithms for this kind of tests are resulted to be the Finiasz-Sendrier, the May-Meurer-Thomae and the Both-May algorithms. The Both-May has better performances compared to the ones of the Esser-Bellini: with the syndrome tests using the nearest neighbor search implemented in Both-May is preferable than splitting the vectors using the λ coordinates for the Indwik-Motwani search used in Esser-Bellini with the current data structures used in the implementation. The other algorithms behave as expected except the Becker-Joux-May-Meurer since the computation of the disjoint arrays holding the indices for managing the extra weights has resulted in a bottleneck with the current implementation.
On the second plot we can notice that the Finiasz-Sendrier's implementation by Vasseur is in line with our implementation of the same algorithm while the Becker-Joux-May-Meurer with depth 2 implemented by Esser,May and Zweydinger for the syndrome tests behaves a little worse than our May-Meurer-Thomae but better than our Becker-Joux-May-Meurer with depth 3.
As we have said, for the smallest tests there is an overheading introduced by the usage of threads. Therefore, in Figures 4.11 and 4.12 we can see the plots with the smaller tests (from $n = 100$ to $n = 180$) without using threads to see the comparison between the measured complexity and the expected one.



(a) Prange average cpu cycles (b) Lee-Brickell average cpu cycles (c) Leon average cpu cycles

Figure 4.11: Evaluation of the ISD algorithms without using threads to find the target error for small tests (syndrome tests) (1)



(a) Stern average cpu cycles

(b) BCD average cpu cycles

(c) Finiasz-Sendrier average cpu cycles

(d) MMT average cpu cycles

(e) BJMM average cpu cycles

(f) Both-May average cpu cycles

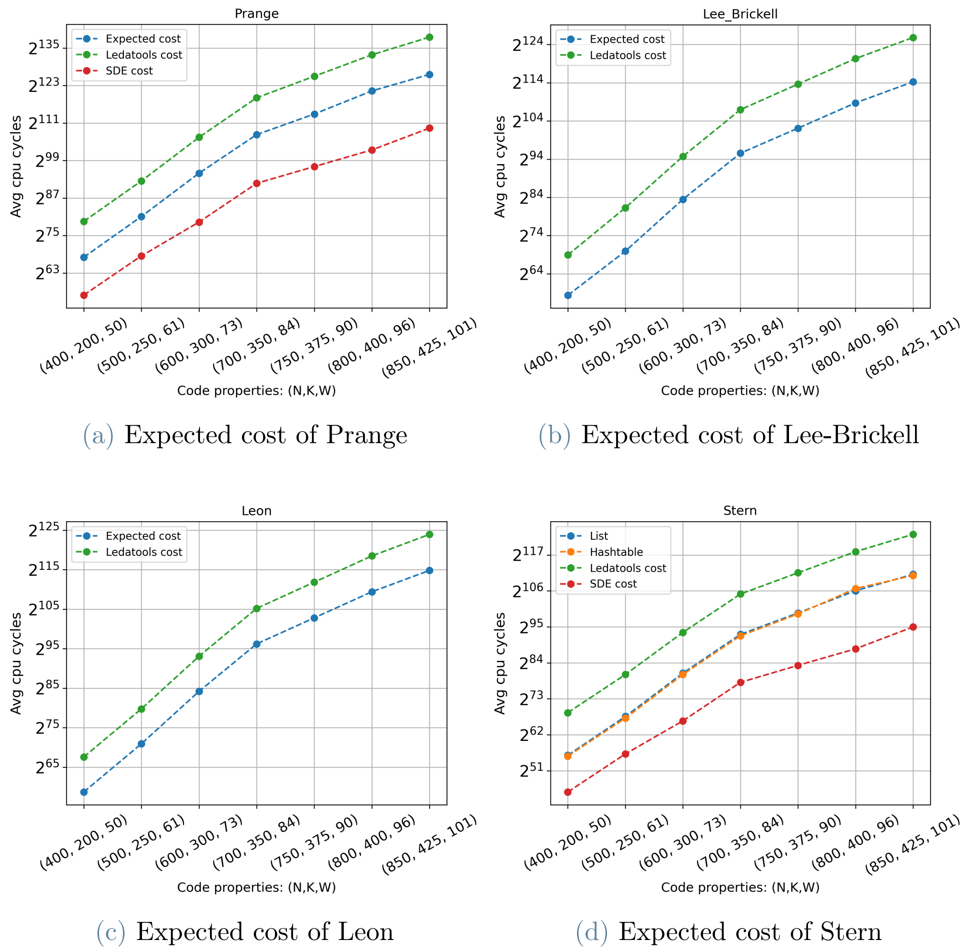
(g) Esser-Bellini average cpu cycles

Figure 4.12: Evaluation of the ISD algorithms without using threads to find the target error for small tests (syndrome tests) (2)

Surprisingly, the measured complexity and the estimated one are not quite the same but the measured is less: this means that for these small tests the current implementation finds the target error faster than the expectation. We have seen how the usage of the threads introduce an overheading for the smallest tests since in the plots with the threads the measured complexity was much bigger than the estimated one while without using the threads this behaviour is not replicated.

We have seen from Figure 4.6 to Figure 4.8 that when the tests begin to be more complex, the estimated cost of the ISD algorithms, computed multiplying the cost of a single iteration with the expected number of iterations, is in line with the cost measured concretly testing the implementation. Therefore, we can produce the plots reporting the estimated cost complexity for each of the ISD implemented using tests who can take from days to several months to complete. To do so, for each test taken in consideration, we have directly measured on the implementation the average cost of a single iteration using the optimal parameters returned by the proper estimator, and multiply it with the expected number of iterations to obtain the expected total cost of the ISD.

In the following figure we can see the results obtained for each ISD together with the cost returned by the estimators, while in Figure 4.15 all the ISD algorithm's results are plotted in one figure to see a direct comparison between them for complex tests.



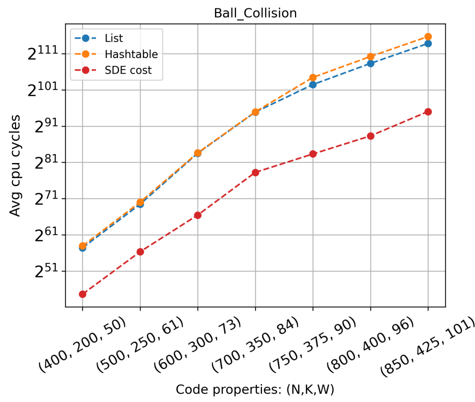
(a) Expected cost of Prange

(b) Expected cost of Lee-Brickell

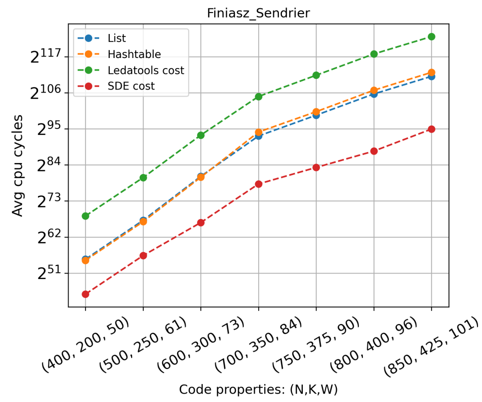
(c) Expected cost of Leon

(d) Expected cost of Stern

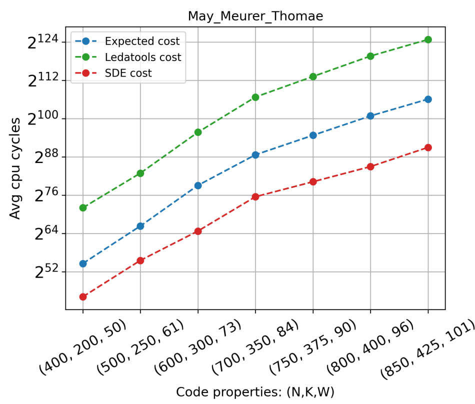
Figure 4.13: Expected total cost of the ISD algorithms to find the target error with complex tests (syndrome tests) (1)



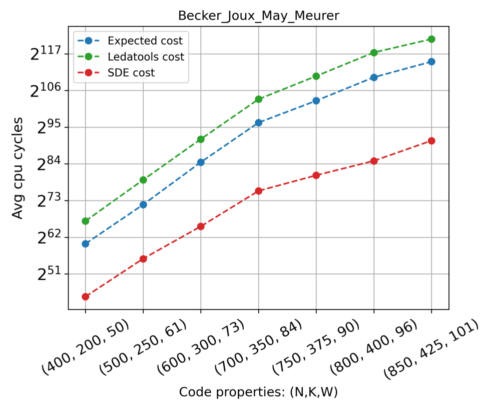
(a) Expected cost of BCD



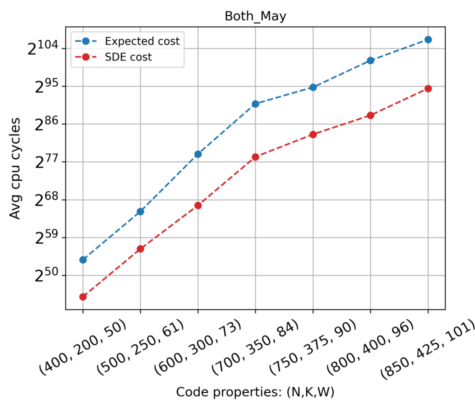
(b) Expected cost of Finiasz-Sendrier



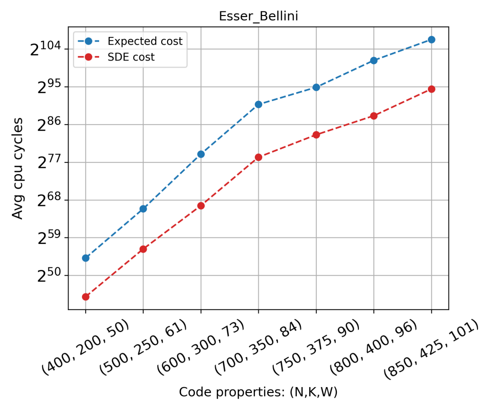
(c) Expected cost of MMT



(d) Expected cost of BJMM



(e) Expected cost of Both-May



(f) Expected cost of Esser-Bellini

Figure 4.14: Expected total cost of the ISD algorithms to find the target error with complex tests (syndrome tests) (2)

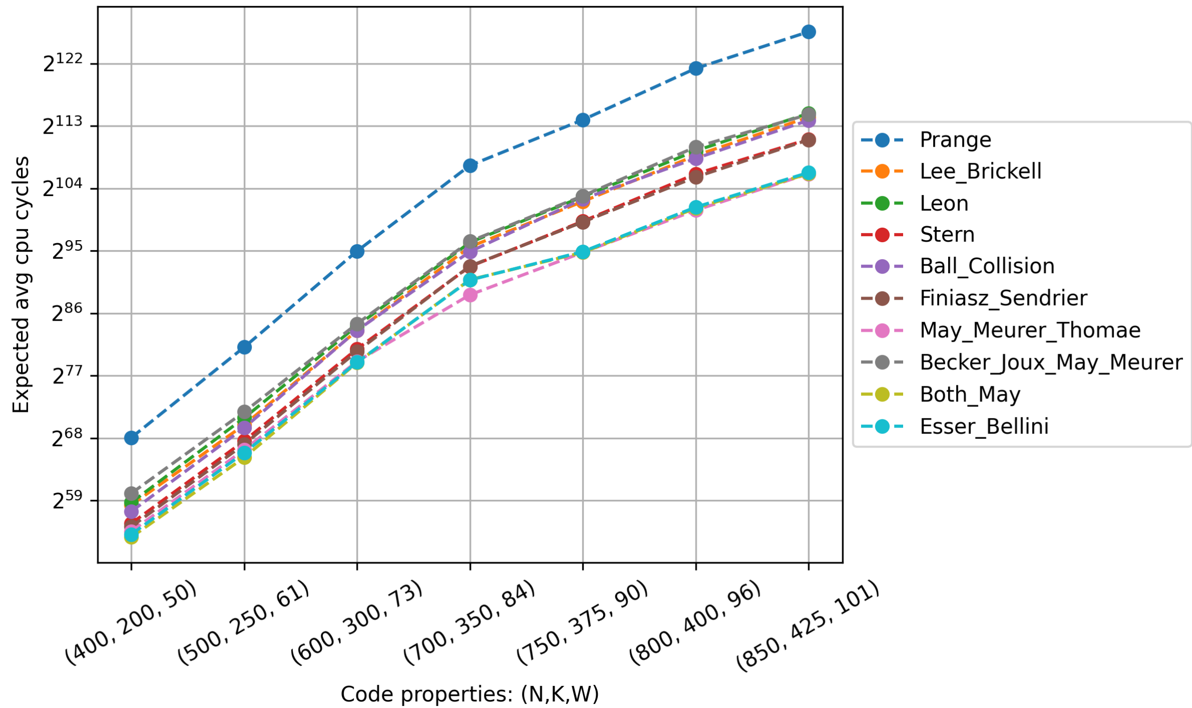


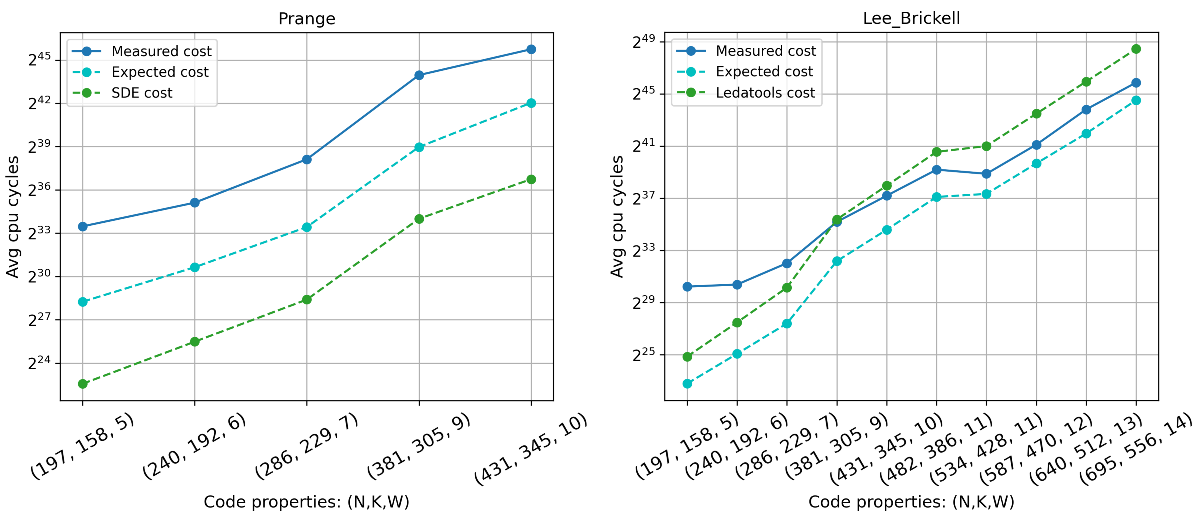
Figure 4.15: Expected total cost for all the ISD algorithms with complex tests (syndrome tests)

We can see how with the increasing size of the tests the cost for finding an error with the ISD algorithms increases exponentially: the syndrome decoding problem, as we expected, starts becoming intractable as the code properties used in the tests increase.

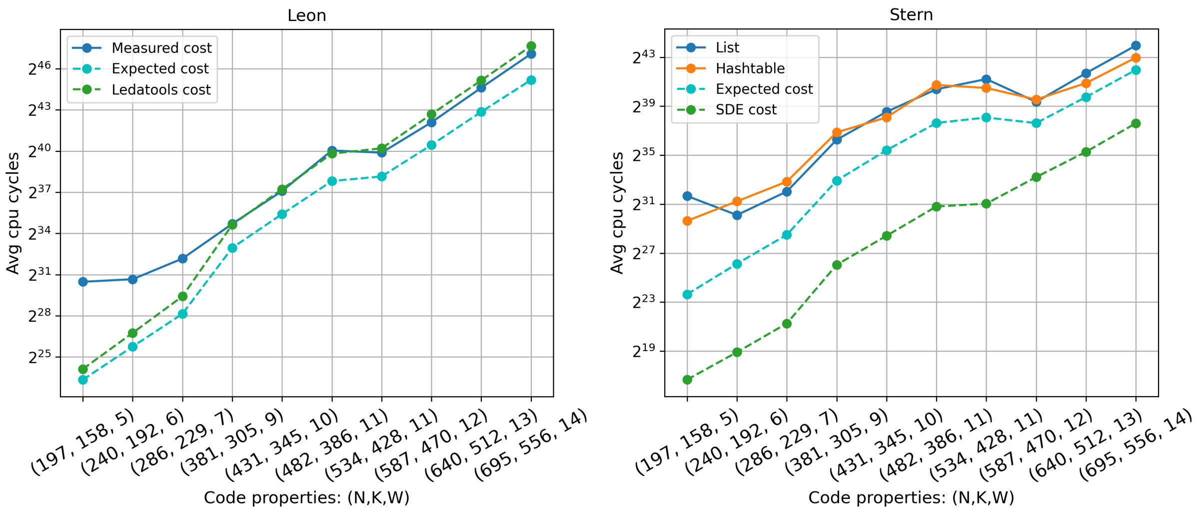
4.5.2. McEliece Tests

In these tests the code rate R is equal to 0.8 resulting in $k = 0.8n$, while the weight of the target error to find is $w = \lceil \frac{n}{5\lceil \log(n) \rceil} \rceil$. The tests in input are taken in the "Syndrome Decoding in the Goppa-McEliece Setting" section at [4] and they are intended to be as close as possible to the problem on which relies the Classic McEliece cryptosystem proposed for the NIST standardization process. Under these conditions, instances with cryptographic size are assumed to be out of reach, so we have considered instances with increasing size, as in the previous tests, to see how hard this problem is in practice. All the guidelines followed in the "Syndrome tests" are still valid. We report one plot for each ISD showing the measured average cost complexity together with the estimated cost complexity and the cost returned by the proper estimator as in the previous section.
It is important to highlight that even here, in the Ball-Collision Decoding algorithm, the optimal parameter z (the weight of the ℓ part of the error) is returned with a value

equal to 0 by the Syndrome Decoding estimator [13] for the family of tests taken in consideration. Having $z = 0$ implies that the Ball-Collision Decoding algorithm coincides with the Stern one: for this reason, we have tested the Ball-Collision Decoding with the minimum admissible z, that is equal to 2, to obtain a different test with respect to the one of the Stern. The performances obtained by the Ball-Collision Decoding algorithm are worse than the other algorithms since it hasn't been used the optimal parameter: the optimal result for the Ball-Collision Decoding can be seen in the plot of the Stern algorithm.

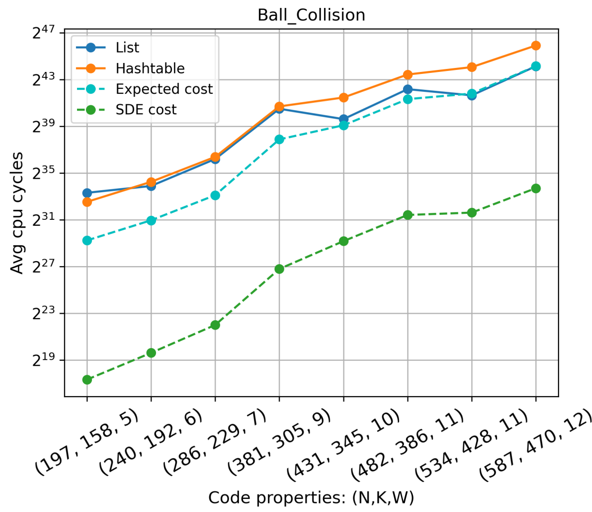


(a) Prange avg cpu cycles to find the target error (b) Lee-Brickell avg cpu cycles to find the target error

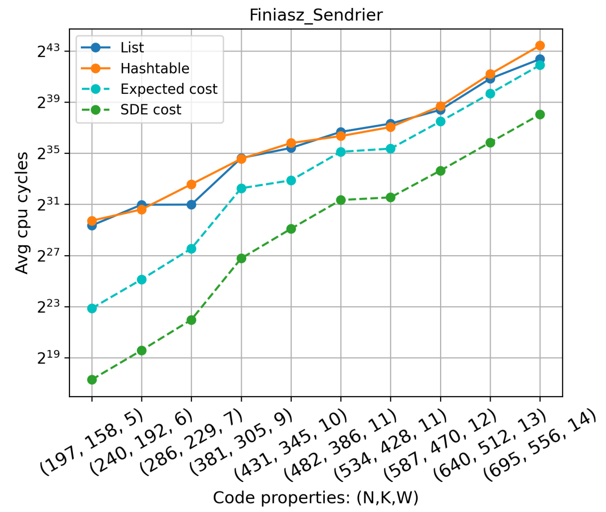


(c) Leon avg cpu cycles to find the target error (d) Stern avg cpu cycles to find the target error

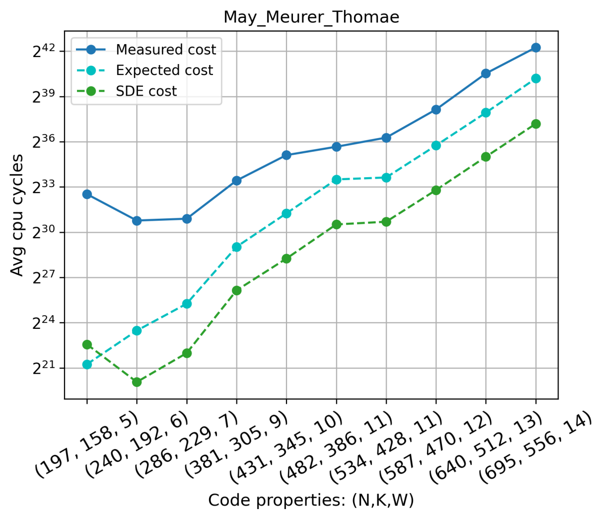
Figure 4.16: Prange, Lee-Brickell, Leon and Stern algorithms evaluation (mceliece tests)



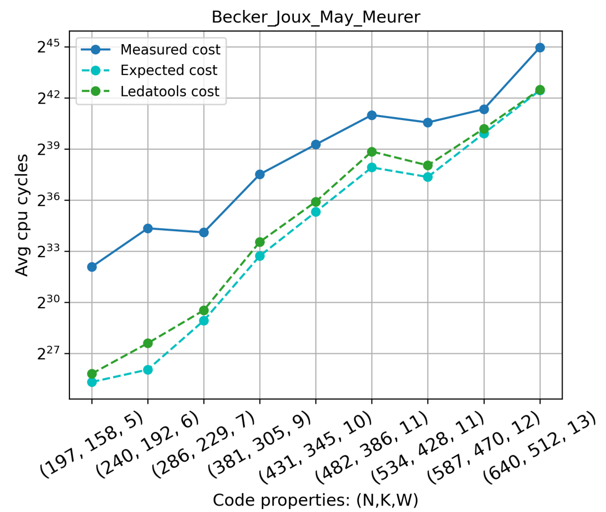
(a) Ball-Collision avg cpu cycles to find the target error



(b) Finiasz-Sendrier avg cpu cycles to find the target error

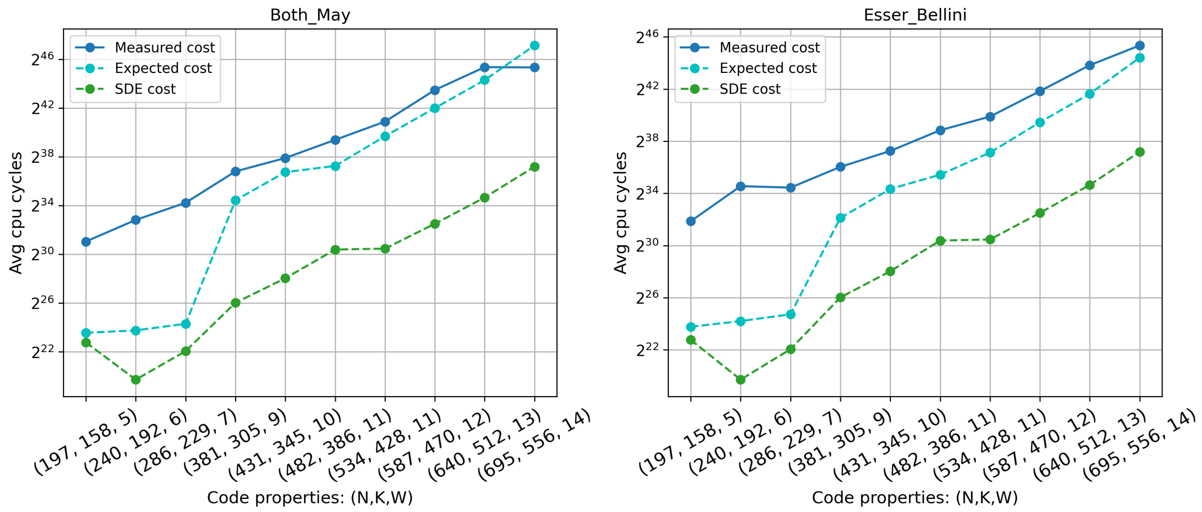


(c) MMT avg cpu cycles to find the target error



(d) BJMM avg cpu cycles to find the target error

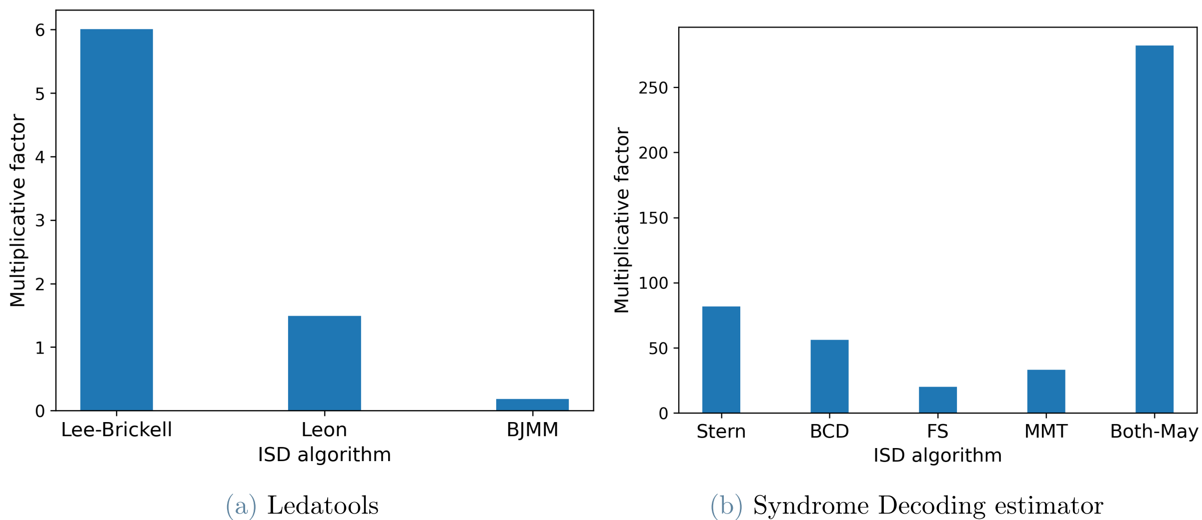
Figure 4.17: Ball-Collision, Finiasz-Sendrier, May-Meurer-Thomae and Becker-Joux-May-Meurer algorithms evaluation (mceliece tests)



(a) Both-May avg cpu cycles to find the target error (b) Esser-Bellini avg cpu cycles to find the target error

Figure 4.18: Both-May and Esser-Bellini algorithms evaluation (mceliece tests)

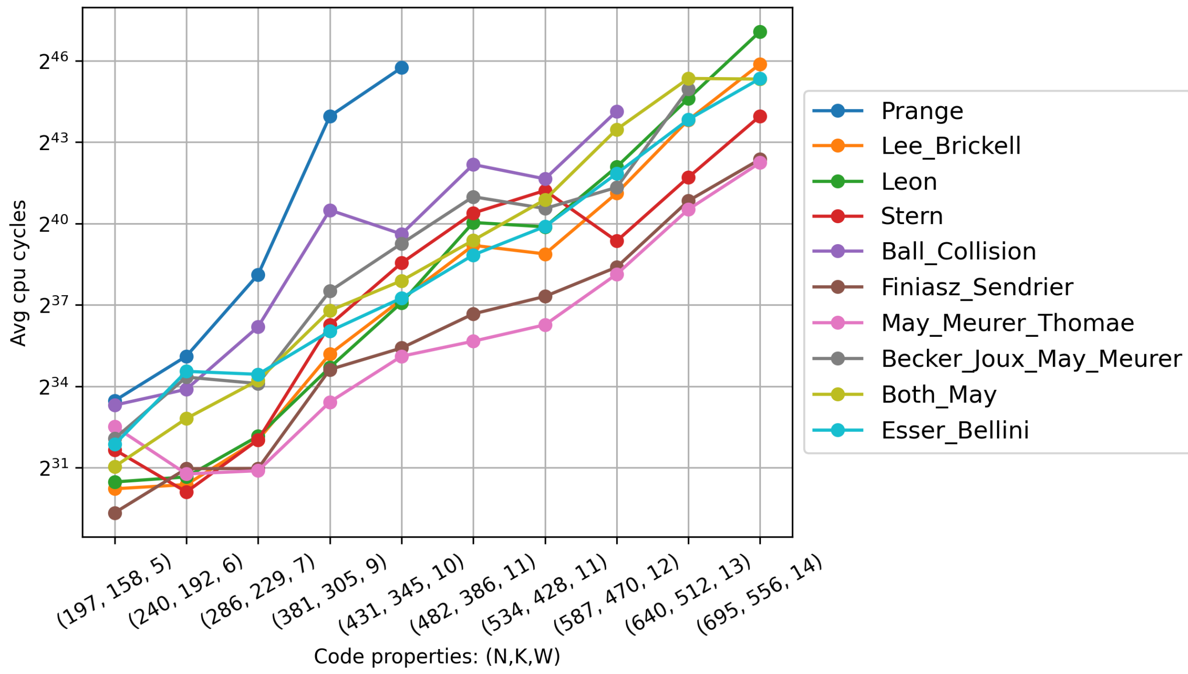
Even here we have measured the multiplicative factor that models how many elementary abstract operations the machine done for each clock cycle. For both the estimators, considering the greatest code tested, we have computed the ratio between the cost returned by the estimator and the measured cost, or viceversa if this latter is greater than the estimate. We can see the results in the Figure 4.19.



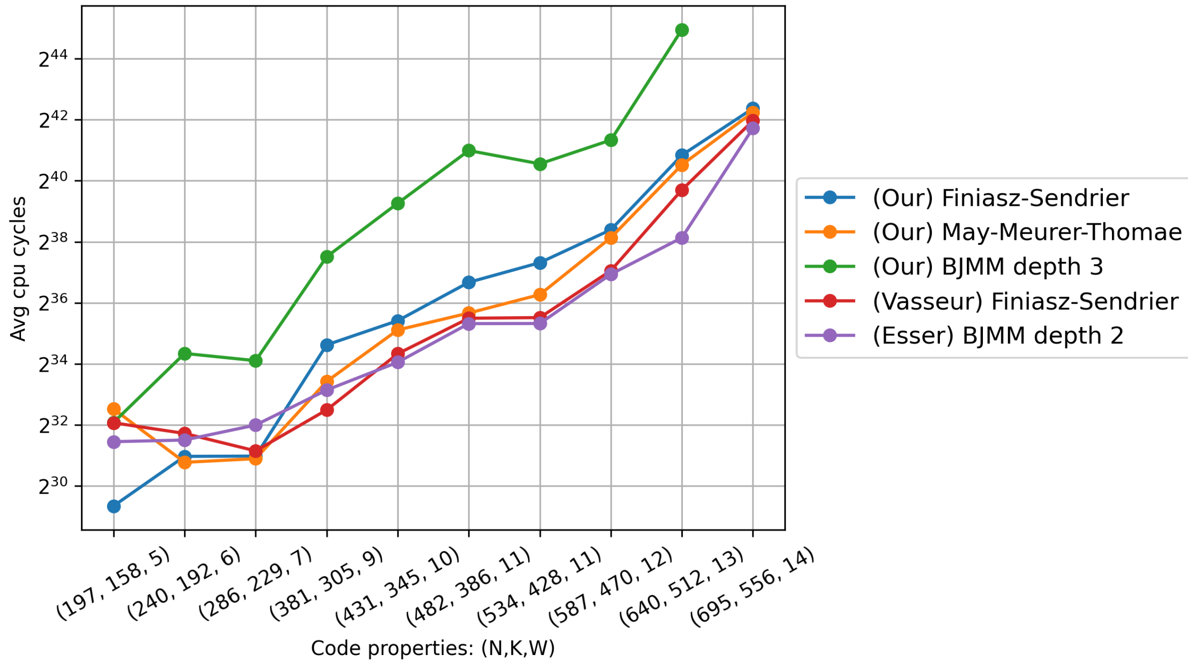
(a) Ledatools (b) Syndrome Decoding estimator

Figure 4.19: Multiplicative factor between the measured cost and the estimate returned by the estimators (mceliece tests)

The reasoning explained in the syndrome tests is valid even here: for each ISD the multiplicative factor is constant between different runs, but since some algorithms have more calculations to do in one iteration or more accesses to the memory than others, it varies depending on the ISD in analysis.
As before, in Figure 4.20 two plots are reported: one collects the results between all the ISD implemented in this work to see a direct comparison among them while the other shows the comparison between our Finiasz-Sendrier's algorithm with the one implemented by Vasseur and our MMT and BJMM with the BJMM at depth 2 implemented by Esser, May and Zweydinger with the mceliece tests.
Here, the fastest ISD algorithms implemented in this thesis for this kind of tests are resulted to be the Finiasz-Sendrier and the May-Meurer-Thomae algorithms as before where the latter has the best performances with the mceliece tests. Different from before, the Esser-Bellini algorithm has better performances compared to the ones of the Both-May. The Becker-Joux-May-Meurer with depth 2 implemented by Esser, May and Zweydinger for the mceliece tests behaves better than our MMT and BJMM different from the the case in which the syndrome tests have been used.
As happened in the syndrome tests, when we consider small tests that are solvable in less than one second, the difference between the measured cost and the expected one is higher due to the overhead of the threads. Because of that, the plots with the small tests without using threads are reported in Figure 4.21: we can notice that the estimated cost and the measured cost without using the threads are almost identical as we expect, different from the case where we use threads with simple tests in which the difference between the measured cost and the estimated cost is very high.

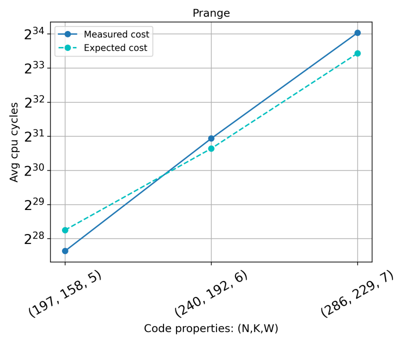


(a) Comparison between our ISD algorithms

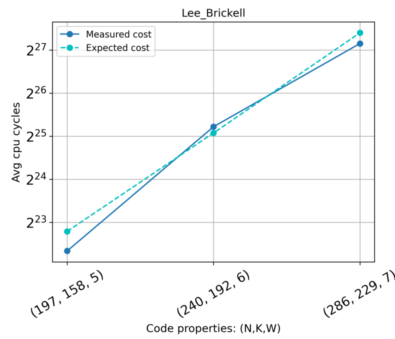


(b) Comparison between our ISD with the Vasseur [30] and the Esser,May,Zweydinger [14] implementations

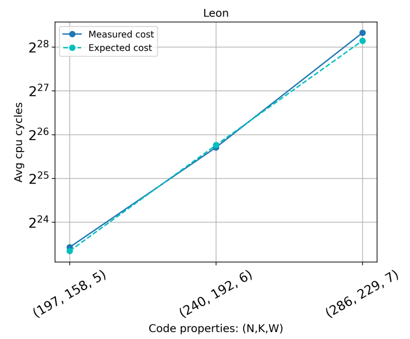
Figure 4.20: Computational cost comparison between all the ISD algorithms (mceliece tests)



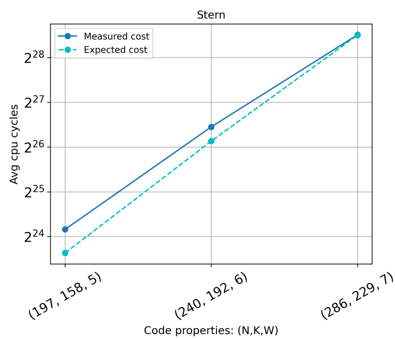
(a) Prange average cpu cycles



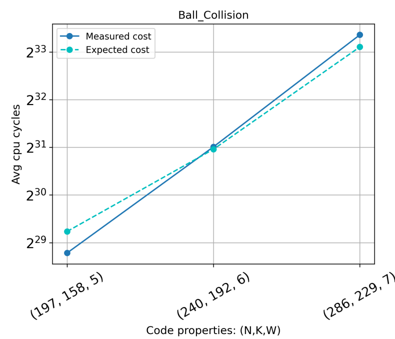
(b) Lee-Brickell average cpu cycles



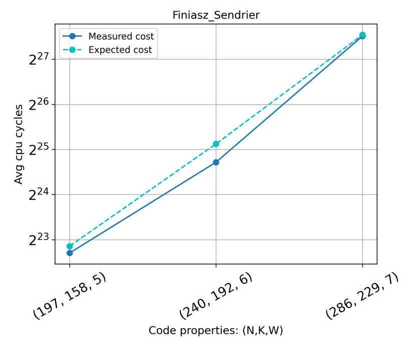
(c) Leon average cpu cycles



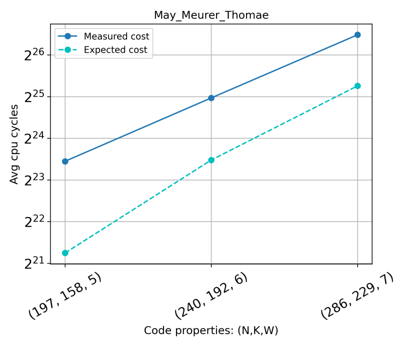
(d) Stern average cpu cycles



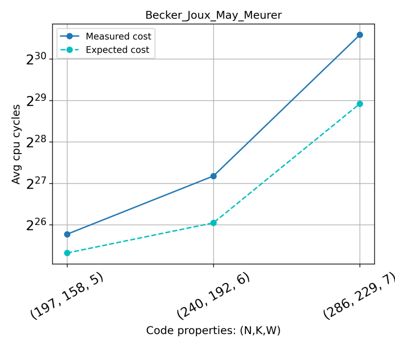
(e) Ball-Collision avg cpu cycles



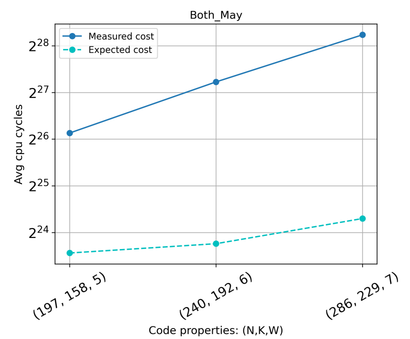
(f) Finiasz-Sendrier avg cpu cycles



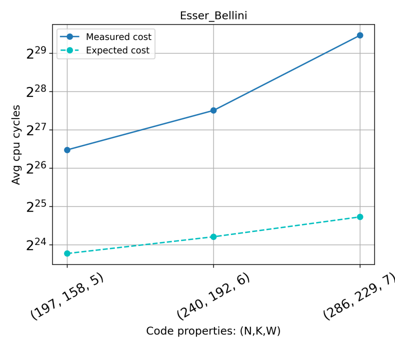
(g) MMT average cpu cycles



(h) BJMM average cpu cycles

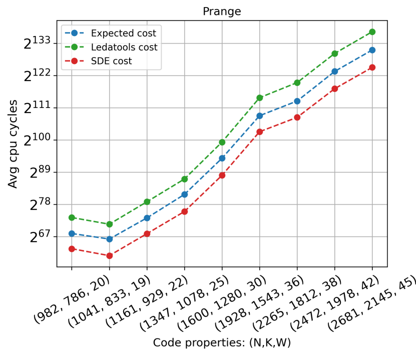


(i) Both-May average cpu cycles

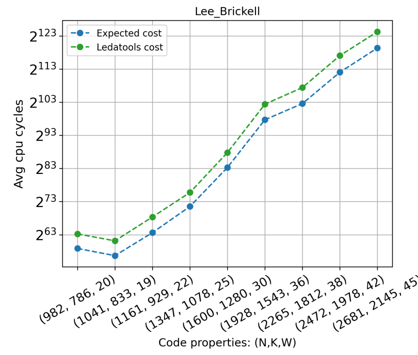


(j) Esser-Bellini average cpu cycles

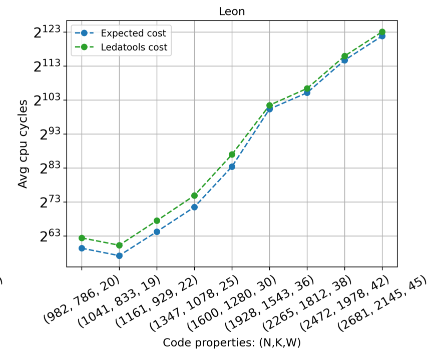
Figure 4.21: Evaluation of the ISD algorithms without using threads to find the target error for small tests (mceliece tests)



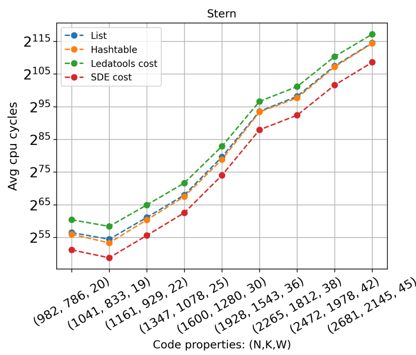
(a) Expected cost of Prange



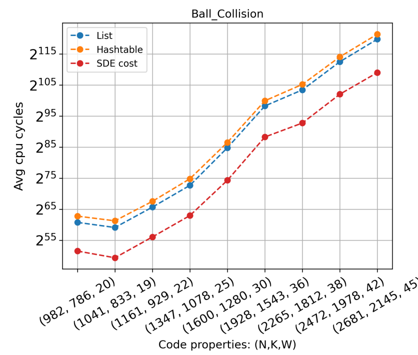
(b) Expected cost of Lee-Brickell



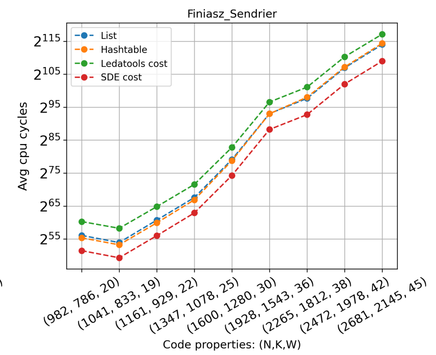
(c) Expected cost of Leon



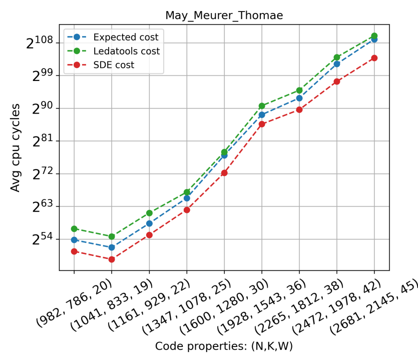
(d) Expected cost of Stern



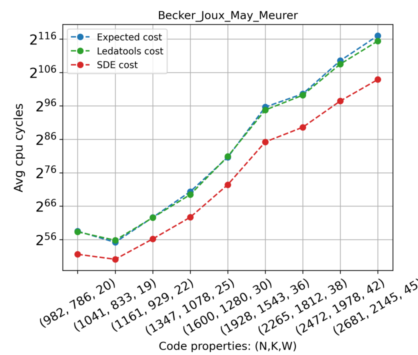
(e) Expected cost of BCD



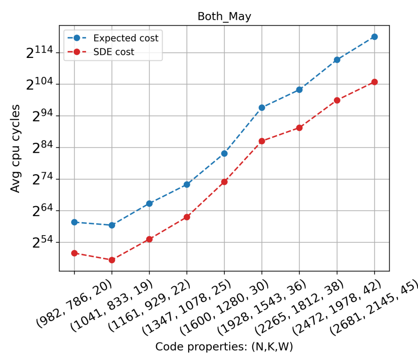
(f) Expected cost of Finiasz-Sendrier



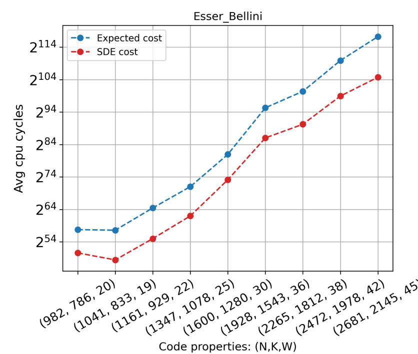
(g) Expected cost of May-Meurer-Thomae



(h) Expected cost of Becker-Joux-May-Meurer



(i) Expected cost of Both-May



(j) Expected cost of Esser-Bellini

Figure 4.22: Expected total cost of the ISD algorithms to find the target error with complex tests (mceliece tests)

Last, on the previous page at Figure 4.22, we have produced the plots reporting the estimated cost complexity for each ISD using tests who can take from days to several months to complete. In the following we can see one plot who collects the expected cost measured for all the ISD algorithms in case of complex tests.

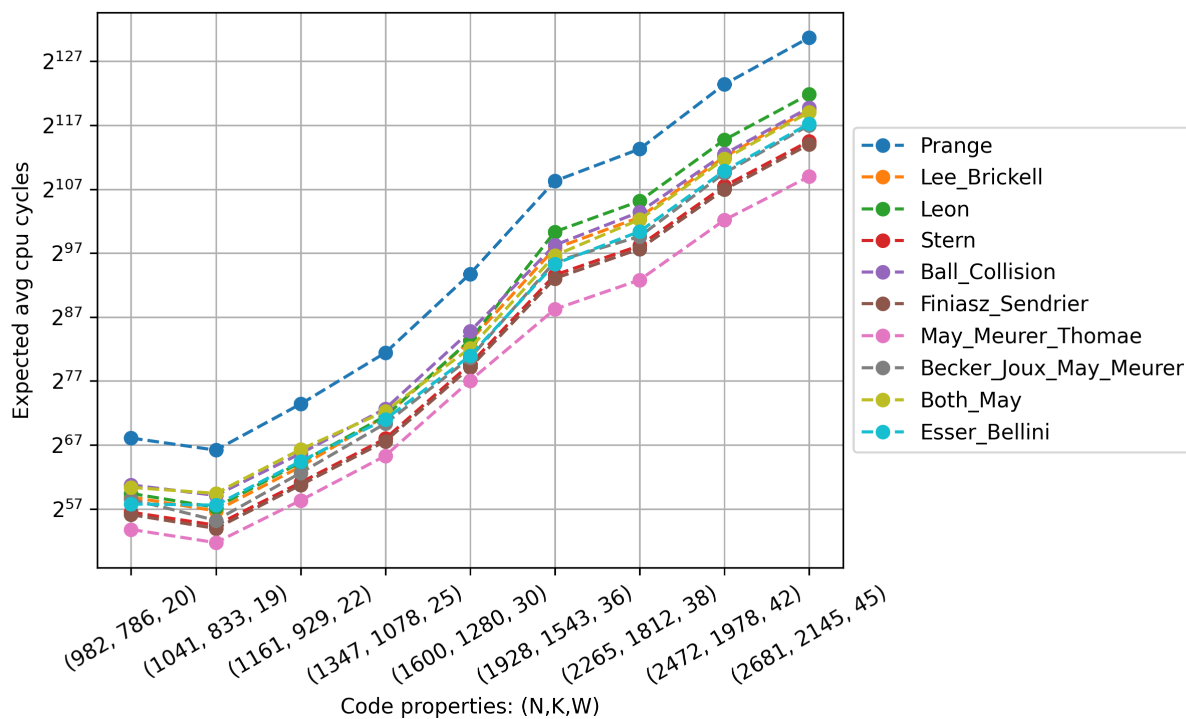


Figure 4.23: Expected total cost for all the ISD algorithms with complex tests (mceliece tests)

As before, we can see how with the increasing of the size of the tests the cost for finding an error increases exponentially: the syndrome decoding problem, as we expected, starts becoming intractable with greater code properties.
The NIST has proposed five security categories for the McEliece cryptosystem: a parameter set matches the security level of the category one,three or five if the scheme instantiated with this set is at least as hard to break as AES-128, AES-192 or AES-256 respectively. In the following table we can see the parameter sets proposed for the category one,three and five.

Table 4.1: Parameter sets suggested by NIST for McEliece cryptosystem

Category	n	k	w
1	3488	2720	64
3	4608	3360	96
5	6688	5024	128
5	6960	5413	119
5	8192	6528	128

We can conclude, thanks to the testing results, that the McEliece cryptosystem with the parameter sets proposed by the NIST in Table 4.1 is considered secure from the point of view of the Information Set Decoding algorithms. This is true because, as we can notice from the last plot, the syndrome decoding problem just with the smallest set proposed by the NIST will have a bigger complexity compared to the one retrieved by the best ISD algorithm with the largest test taken in consideration. Since the complexity of the syndrome decoding problem grows exponentially, the problems with the parameter sets in category 3 and 5 will be more complex and intractable too.

5 | Conclusions and future developments

In this thesis we focused on how to solve the syndrome decoding problem, the basis on which the code-based cryptosystems are founded on. This problem has been solved using the best algorithms currently known in the state of the art called Information Set Decoding algorithms. After a brief chapter in which the basic of the coding theory and the functioning of McEliece and Niederrieter cryptosystems are explained, in the second chapter we have analyzed different routines that have been used inside the Information Set Decoding algorithms. We have analyzed different procedures to compute a systematic form of a binary matrix, various binary searches to find an element inside a list efficiently, a smart way to compute all the possible combinations using an array and two sorting algorithms for ordering lists. Then, in the first part of the experimental evaluation chapter, we have tested the different RREF methods, the binary search variants and the two sorting algorithms to see which variant performs best and unterstand which one to use inside the ISD algorithms implementation.
In Chapter 3, first we have studied what is an information set decoding algorithm and then, we have presented all the ISD implemented reporting the pseudocode of each algorithm with the relative computational and spatial complexity. The algorithms implemented are: Prange, Lee-Brickell, Leon, Stern, Ball-Collision Decoding, Finiasz-Sendrier, May-Meurer-Thomae, Becker-Joux-May-Meurer, Both-May and Esser-Bellini. These algorithms have been tested in the second part of the experimental evaluation part in Chapter 4: here the results of a concrete evaluation of the Information Set Decoding algorithms has been reported to understand how they behave in practice. Two family of codes have been taken in consideration: the first has a code rate $\mathrm{R} = 0.5$ and the weight of the target error close to the Gilbert-Varshamov distance, while the second has a code rate $\mathrm{R} = 0.8$ and corresponds to instances of the syndrome decoding problem on which the security of the classic McEliece cryptosystem relies. As expected, with the increasing dimensions of the code in the tests, the complexity of the ISD algorithms for finding the target error grows exponentially. We have seen how the measured cost complexity is in

line with the estimated one computed multiplying the measured cost of a single iteration with the number of expected iterations: thanks to that, we have been able to produce the testing results even for complex tests that might be run for months. Analyzing the results of the McEliece tests with high dimension codes we have concluded that the McEliece cryptosystem instantiated with the parameter sets proposed by the NIST is secure, since the syndrome decoding problem becomes intractable for the Information Set Decoding algorithms using these parameter sets.
A possible future development can be moving the current implementation of the information set decoding algorithms in GPU to see how the performances change.

Bibliography

[1] May, a.; meurer, a.; thomae, e. decoding random linear codes in $\mathcal{O}(2^{0.054n})$. in proceedings of the advances in cryptology—asiacrypt 2011—17th international conference on the theory and application of cryptology and information security, seoul south korea, 4–8 december 2011; pp. 107–124.

[2] M. Albrecht and G. Bard. *The M4RI Library*. The M4RI Team. URL `https://bitbucket.org/malb/m4ri`.

[3] M. R. Albrecht and C. Pernet. Efficient decomposition of dense matrices over gf (2). *arXiv preprint arXiv:1006.1744*, 2010.

[4] N. Aragon, J. Lavauzelle, and M. Lequesne. decodingchallenge.org, 2019. URL `http://decodingchallenge.org`.

[5] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini. A finite regime analysis of information set decoding algorithms. *Algorithms*, 12(10), 2019. ISSN 1999-4893. doi: 10.3390/a12100209. URL `https://www.mdpi.com/1999-4893/12/10/209`.

[6] G. V. Bard. Accelerating cryptanalysis with the method of four russians. Cryptology ePrint Archive, Report 2006/251, 2006. `https://ia.cr/2006/251`.

[7] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[8] A. Becker, A. Joux, A. May, and A. Meurer. Decoding random binary linear codes in $2^{n/20}$: How 1+ 1= 0 improves information set decoding. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 520–536. Springer, 2012.

[9] D. J. Bernstein, T. Lange, and C. Peters. Smaller decoding exponents: ball-collision decoding. In *Annual Cryptology Conference*, pages 743–760. Springer, 2011.

[10] D. J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal. Ntru prime: Reducing attack surface at low cost. In C. Adams and J. Camenisch, editors, *Se-*

lected Areas in Cryptography – SAC 2017, pages 235–260, Cham, 2018. Springer International Publishing. ISBN 978-3-319-72565-9.

[11] D. Blackman and S. Vigna. Scrambled linear pseudorandom number generators. *ACM Trans. Math. Softw.*, 47(4), sep 2021. ISSN 0098-3500. doi: 10.1145/3460772. URL `https://doi.org/10.1145/3460772`.

[12] L. Both and A. May. Decoding linear codes with high error rate and its impact for lpn security. In *International Conference on Post-Quantum Cryptography*, pages 25–46. Springer, 2018.

[13] A. Esser and E. Bellini. Syndrome decoding estimator. Cryptology ePrint Archive, Report 2021/1243, 2021. `https://ia.cr/2021/1243`.

[14] A. Esser, A. May, and F. Zweydinger. Mceliece needs a break – solving mceliece-1284 and quasi-cyclic-2918 with modern isd. Cryptology ePrint Archive, Report 2021/1634, 2021. `https://ia.cr/2021/1634`.

[15] M. Finiasz and N. Sendrier. Security bounds for the design of code-based cryptosystems. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 88–105. Springer, 2009.

[16] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 01 1962. ISSN 0010-4620. doi: 10.1093/comjnl/5.1.10. URL `https://doi.org/10.1093/comjnl/5.1.10`.

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

[18] Intel. Intel® intrinsics guide. URL `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#`.

[19] P. J. Lee and E. F. Brickell. An observation on the security of mceliece's public-key cryptosystem. In *Advances in Cryptology - EUROCRYPT '88, Workshop on the Theory and Application of of Cryptographic Techniques, Davos, Switzerland, May 25-27, 1988, Proceedings*, volume 330 of *Lecture Notes in Computer Science*, pages 275–280. Springer, 1988. doi: 10.1007/3-540-45961-8_25.

[20] J. S. Leon. A probabilistic algorithm for computing minimum weights of large error-correcting codes. *IEEE Trans. Inf. Theory*, 34:1354–1359, 1988.

[21] R. J. McEliece. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report*, 44:114–116, Jan. 1978.

[22] W. Muła, N. Kurz, and D. Lemire. Faster population counts using avx2 instructions. *The Computer Journal*, 61(1):111–120, 2018.

[23] H. Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Prob. Contr. Inform. Theory*, 15(2):157–166, 1986.

[24] E. Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, 8(5):5–9, 1962. doi: 10.1109/TIT.1962.1057777.

[25] scandum. *Binary Search.* URL `https://github.com/scandum/binary_search`.

[26] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948. doi: 10.1002/j.1538-7305.1948.tb01338.x.

[27] P. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994. doi: 10.1109/SFCS.1994.365700.

[28] V. M. Sidelnikov and S. O. Shestakov. On insecurity of cryptosystems based on generalized reed-solomon codes. 1992.

[29] J. Stern. A method for finding codewords of small weight. In *International Colloquium on Coding Theory and Applications*, pages 106–113. Springer, 1988.

[30] V. Vasseur. *Dumer ISD implementation.* URL `https://github.com/vvasseur/isd`.

[31] T. Wang. Integer hash function, 2007. URL `http://web.archive.org/web/20071223173210/http://www.concentric.net/~Ttwang/tech/inthash.htm`.

[32] H. S. Warren Jr. The quest for an accelerated population count. *Beautiful code: leading programmers explain how they think*, pages 147–60, 2007.

[33] P. Wegner. A technique for counting ones in a binary computer. *Communications of the ACM*, 3(5):322, 1960.

A | Appendix A

Table A.1: Optimal parameters returned by the Ledatools estimator

Code (n, k, w)	$\mathbf{p}$ (LB)	$\mathbf{p}$ (Leon)	$\boldsymbol{\ell}$ (Leon)	$\mathbf{p}$ (Stern)	$\boldsymbol{\ell}$ (Stern)	$\mathbf{p}$ (FS)	$\boldsymbol{\ell}$ (FS)	$\mathbf{p}$ (MMT)	$\boldsymbol{\ell}$ (MMT)	$\boldsymbol{\ell_1}$ (MMT)	$\mathbf{p}$ (BJMM)	$\boldsymbol{\ell}$ (BJMM)	Δ_1 (BJMM)	Δ_2 (BJMM)
Syndrome tests														
(100,50,14)	2	3	3	4	6	6	14	4	14	9	4	14	2	0
(130,65,18)	2	3	3	4	7	6	15	4	13	7	4	15	2	0
(150,75,20)	2	3	3	6	14	6	15	4	14	7	4	15	2	0
(180,90,24)	2	3	3	6	15	6	16	4	14	6	4	20	2	0
(200,100,27)	2	3	3	6	16	6	17	4	14	5	4	16	2	0
(220,110,29)	2	3	4	6	16	6	18	4	14	4	4	16	2	0
(250,125,32)	2	3	4	6	17	6	18	4	15	4	4	16	2	0
(270,135,34)	2	3	4	6	18	6	19	4	15	3	4	18	2	0
(300,150,38)	2	3	4	6	18	6	19	4	16	3	4	17	2	0
(320,160,40)	2	3	4	6	19	6	19	4	16	2	4	17	2	0
(350,175,44)	2	3	4	6	19	6	20	4	16	1	4	17	2	0
McEliece tests														
(197,158,5)	2	2	3	4	14	4	16	4	22	19	4	17	2	0
(240,192,6)	2	2	3	4	14	4	16	4	22	18	4	22	2	0
(286,229,7)	2	2	3	4	15	4	15	4	21	17	4	18	2	0
(381,305,9)	2	2	3	4	16	4	16	4	22	18	4	19	2	0
(431,345,10)	2	2	3	4	16	4	17	8	47	39	4	19	2	0
(482,386,11)	2	2	3	4	16	4	17	8	48	40	4	19	2	0
(534,428,11)	2	2	3	4	17	4	17	8	49	41	4	20	2	0
(587,470,12)	2	2	3	4	17	4	18	8	50	42	4	20	2	0
(640,512,13)	2	2	3	4	17	4	18	8	51	42	4	20	2	0
(695,556,14)	2	2	3	4	18	4	18	8	52	43	4	20	2	0

Table A.2: Optimal parameters returned by the Syndrome Decoding estimator

(n,k,w)	$\mathbf{p}$ (Stern)	$\boldsymbol{\ell}$ (Stern)	$\mathbf{p}$ (BCD)	$\boldsymbol{\ell}$ (BCD)	$\mathbf{z}$ (BCD)	$\mathbf{p}$ (FS)	ℓ (FS)	$\mathbf{p}$ (MMT)	ℓ (MMT)	ℓ_1 (MMT)	$\mathbf{p}$ (BJMM)	$\boldsymbol{\ell}$ (BJMM)	Δ_1 (BJMM)	Δ_2 (BJMM)	$\mathbf{p}$ (BM)	$\boldsymbol{r_1}$ (BM)	$\boldsymbol{w_1}$ (BM)	Δ_1 (BM)
Syndrome tests																		
(100,50,14)	4	3	4	5	2	4	6	4	6	2	4	10	2	0	4	2	0	0
(130,65,18)	4	4	4	5	2	4	5	4	7	2	8	18	0	0	4	3	0	0
(150,75,20)	4	6	4	6	2	4	6	4	7	2	8	16	0	0	4	2	0	0
(180,90,24)	4	6	4	6	2	4	6	4	7	2	8	17	0	0	4	3	0	0
(200,100,27)	4	5	4	6	2	4	6	4	7	2	8	18	0	0	4	3	0	0
(220,110,29)	4	6	4	7	2	4	7	4	8	2	8	18	0	0	4	3	0	0
(250,125,32)	4	6	4	7	2	4	7	4	8	2	8	19	0	0	4	3	0	0
(270,135,34)	4	8	4	7	2	4	7	4	8	2	8	19	0	0	4	3	0	0
(300,150,38)	4	6	4	7	2	4	8	4	8	2	8	19	0	0	4	4	0	0
(320,160,40)	4	8	4	8	2	4	8	4	8	2	8	20	0	0	4	3	0	0
(350,175,44)	4	8	4	8	2	4	8	4	9	2	8	20	0	0	4	4	0	0
McEliece tests																		
(197,158,5)	2	7	2	4	2	2	4	4	6	2	4	14	2	0	2	3	0	0
(240,192,6)	2	7	2	4	2	2	4	4	12	2	4	20	2	0	4	4	0	0
(286,229,7)	2	7	2	4	2	2	4	4	13	2	4	21	2	0	4	7	0	0
(381,305,9)	2	6	2	4	2	2	4	4	13	2	4	22	2	0	4	2	0	0
(431,345,10)	2	6	4	12	2	4	12	4	13	2	4	23	2	0	4	2	0	0
(482,386,11)	2	6	4	12	2	4	12	4	13	2	4	23	2	0	4	3	0	0
(534,428,11)	2	7	4	13	2	4	13	4	14	2	4	24	2	0	4	2	0	0
(587,470,12)	4	14	4	13	2	4	13	4	14	2	4	24	2	0	4	2	0	0
(640,512,13)	4	14	4	13	2	4	13	4	14	2	8	28	0	0	4	2	0	0
(695,556,14)	4	13	4	13	2	4	13	4	14	2	8	29	0	0	4	6	0	0

List of Figures

1.1 Error correcting code over a noisy channel 5

2.1 Partial systematic form of H 16

3.1 Situation with the transformed matrix and vectors after calling the RREF 45
3.2 Weight distribution in Prange algorithm 49
3.3 Weight distribution in Lee-brickell algorithm 52
3.4 Weight distribution in Leon algorithm 54
3.5 Weight distribution in Stern algorithm 58
3.6 Weight distribution in Ball-Collision Decoding algorithm 62
3.7 Weight distribution in Finiasz-Sendrier algorithm 66
3.8 Weight distribution in May-Meurer-Thomae algorithm 69
3.9 Lists of MMT algorithm 70
3.10 Weight distribution in BJMM algorithm 74
3.11 Lists of BJMM algorithm 76
3.12 Weight distribution in Both-May algorithm 80
3.13 Lists of Both-May algorithm 82
3.14 Lists of Esser-Bellini algorithm 86
3.15 Array of unsigned int for representing a binary vector 89
3.16 Bidimensional array of unsigned int for representing a binary matrix 90

4.1 Comparison between different RREF procedures 97
4.2 Comparison between different partial RREF procedures 97
4.3 Comparison between binary range search variants: percentage speedup of the execution time with respect to the one of the standard binary range search 98
4.4 Comparison between sorting algorithms: percentage speedup of the execution time of Quicksort with respect to the one of the Djbsort 100
4.5 Expected computational cost returned by the Ledatools and the Syndrome Decoding estimator 101
4.6 Prange, Lee-Brickell, Leon and Stern algorithms evaluation (syndrome tests) 104

4.7 Ball-Collision Decoding, Finiasz-Sendrier, May-Meurer-Thomae and Becker-Joux-May-Meurer algorithms evaluation (syndrome tests) 105
4.8 Both-May and Esser-Bellini algorithms evaluation (syndrome tests) 106
4.9 Multiplicative factor between the measured cost and the estimate returned by the estimators (syndrome tests) 107
4.10 Computational cost comparison between all the ISD algorithms (syndrome tests) 108
4.11 Evaluation of the ISD algorithms without using threads to find the target error for small tests (syndrome tests) (1) 109
4.12 Evaluation of the ISD algorithms without using threads to find the target error for small tests (syndrome tests) (2) 110
4.13 Expected total cost of the ISD algorithms to find the target error with complex tests (syndrome tests) (1) 111
4.14 Expected total cost of the ISD algorithms to find the target error with complex tests (syndrome tests) (2) 112
4.15 Expected total cost for all the ISD algorithms with complex tests (syndrome tests) 113
4.16 Prange, Lee-Brickell, Leon and Stern algorithms evaluation (mceliece tests) 114
4.17 Ball-Collision, Finiasz-Sendrier, May-Meurer-Thomae and Becker-Joux-May-Meurer algorithms evaluation (mceliece tests) 115
4.18 Both-May and Esser-Bellini algorithms evaluation (mceliece tests) 116
4.19 Multiplicative factor between the measured cost and the estimate returned by the estimators (mceliece tests) 116
4.20 Computational cost comparison between all the ISD algorithms (mceliece tests) 118
4.21 Evaluation of the ISD algorithms without using threads to find the target error for small tests (mceliece tests) 119
4.22 Expected total cost of the ISD algorithms to find the target error with complex tests (mceliece tests) 120
4.23 Expected total cost for all the ISD algorithms with complex tests (mceliece tests) 121

List of Tables

2.1 Time complexities of the RREF procedures 22
2.2 Space complexities of the RREF procedures 22

4.1 Parameter sets suggested by NIST for McEliece cryptosystem 122

A.1 Optimal parameters returned by the Ledatools estimator 130
A.2 Optimal parameters returned by the Syndrome Decoding estimator 131

List of Algorithms

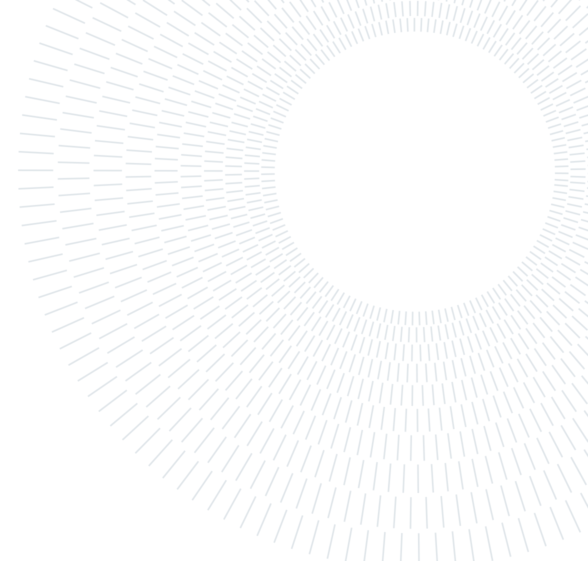
2.1.1 Reduced Row Echelon Form 13
2.1.2 RedRowEchelonForm: Reusing Existing Pivot 19
2.1.3 Partial Reduced Row Echelon Form 20
2.1.4 PartialRedRowEchelonForm: Optimized 21
2.1.5 Method of the Four Russians Inversion (M4RI) 24
2.2.1 Standard Binary Search 26
2.2.2 Boundless Binary Search 27
2.2.3 Doubletapped Binary Search 28
2.2.4 Monobound Binary Search 28
2.2.5 Tripletapped Binary Search 29
2.2.6 Monobound Quaternary Search 30
2.2.7 Monobound Interpolated Binary Search 31
2.2.8 Adaptive Binary Search 32
2.2.9 Boundless Binary Range Search 33
2.3.1 NextComb 36
2.3.2 Init Partial Sums 38
2.3.3 NextColSum Optimized 39
2.4.1 Hoare Partition Scheme 41
2.4.2 Quicksort 41
2.4.3 Djbsort 42

3.1.1 Basic Structure of an ISD algorithm 46
3.2.1 Prange algorithm 51
3.2.2 Lee-Brickell algorithm 53
3.2.3 Leon algorithm 56
3.2.4 Stern algorithm using lists for finding collisions 60
3.2.5 Stern algorithm using hash table for finding collisions 61
3.2.6 Ball-Collision algorithm using lists for finding collisions 64
3.2.7 Finiasz-Sendrier algorithm using lists for finding collisions 67
3.2.8 May-Meurer-Thomae algorithm 71

3.2.9 Becker-Joux-May-Meurer algorithm 77
3.2.10 Nearest Neighbor Search algorithm 81
3.2.11 Both-May algorithm 83
3.2.12 Indwik-Motwani Search algorithm 85
3.2.13 Esser-Bellini algorithm 88
3.3.1 Brian Kernighan's test for computing the Hamming weight 92

List of Symbols

Variable	Description
C	Linear code
G	Generator matrix of a linear code
H	Parity-check matrix of a linear code
n	Length of the linear code (number of columns of H)
k	Rank of the linear code
r	Number of rows of H ($r = n$ - k)
$\boldsymbol{s}$	Syndrome
$\boldsymbol{e}$	Target error
w	Weight of the target error to be found
$\widehat{\boldsymbol{e}}$	Permuted target error
$\widehat{\boldsymbol{s}}$	Permuted syndrome
p	Error affected positions in the second part of $\widehat{\boldsymbol{e}}$ (used from Lee-Brickell)
ℓ	Length of the first part of $\widehat{\boldsymbol{e}}$ for Leon,Stern,BCD and extra parameter for computing the partial RREF for FS,MMT,BJMM
z	Weight of the z part of the error in Ball-Collision algorithm
Δ	Extra weight that will cancel out during the additions (used from BJMM algorithm)
λ	Number of coordinates used for the Indwik-Motwani search



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

A Concrete Evaluation of Information Set Decoding Techniques

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Emanuele Lunardi

Advisor: Prof. Alessandro Barenghi

Academic year: 2020-2021

---

1. Introduction

Nowadays, the most used public-key cryptosystems are based on the hardness of factoring very large numbers or on the intractability of the discrete logarithm problem. Thanks to the Shor's algorithm, we already know that in the future, when it will be possible to build sufficiently performing quantum computers, these problems will be solvable in polynomial time.
Therefore, there is a need to study alternative cryptosystems based on different hard mathematical problems that will be resistant in the era of the quantum computers. One of this alternative is code-based cryptography on which the McEliece and Niederreiter cryptosystems are based on. The functioning of these cryptosystems is founded on the hardness of the syndrome decoding problem, or equivalently, the decoding of a random linear code.
In this work we are going to analyze and implement the algorithms with the best complexities in the current state of the art that solves the syndrome decoding problem. They are called Information Set Decoding algorithms and we have implemented the following ones to concrete evaluate them for understanding how they behave in practice: Prange, Lee-Brickell, Leon, Stern, Ball-Collision Decoding, Finiasz-Sendrier, May-Meurer-Thomae, Becker-Joux-May-Meurer, Both-May and Esser-Bellini.

2. McEliece Cryptosystem and the Syndrome Decoding Problem

The McEliece cryptosystem is an asymmetric encryption algorithm developed by Robert McEliece in 1978 [5] based on the hardness of decoding a random linear code. The working principle is the following: a linear code C is chosen from the Goppa codes family capable of correcting w errors and being indistinguishable from random codes. Then, a random $k \times k$ non-singular binary matrix S and a random $n \times n$ permutation matrix P are selected and the $k \times n$ matrix $\widetilde{G} = SGP$ is computed, where G is the generator matrix of the code chosen from the Goppa family. The public key of this cryptosystem is composed by $\langle \widetilde{G}, w \rangle$ while the private key is $\langle S, P, G \rangle$. This scheme works thanks to the hardness of decoding random linear code: the original code with generator matrix G is hidden by the code generated with the matrix $\widetilde{G}$ obtained perturbating randomly G with the matrices S and P.
A problem equivalent to the decoding random linear code is the so called syndrome decoding problem on which the Niederreiter cryptosystem, a variant of the McEliece one, is based

on. Instead of working with the generator matrix of the code C, this scheme exploits the binary parity-check matrix H of the code following the same logical reasoning(using different matrices size) obtaining a public key composed by $\langle \widetilde{H}, w \rangle$ and a private key by $\langle S, P, H \rangle$. Since the decoding random linear code and the syndrome decoding problem are equivalent, we will focus our attention to the latter because the binary parity-check matrix has smaller dimensions with respect to the generator matrix and so, we can solve the problem with less effort. It follows a formal definition. Let $\widetilde{H} \in \mathbb{F}_q^{(n-k)\times n}$ be a parity check matrix, $s = \widetilde{H}e^T \in \mathbb{F}_q^{n-k}$ the syndrome and w an integer. The Syndrome Decoding Problem asks to find an error $e \in \mathbb{F}_q^n$ with $\mathtt{HW}(e) \leq w$ such that $\widetilde{H}e^T = s$. This problem is NP-hard and if we can solve it we are able to break the code-based cryptosystems of McEliece and Niederrieter. In the next section, we are going to study the Information Set Decoding algorithms for solving the syndrome decoding problem.

3. Information Set Decoding Algorithms

All the Information Set Decoding algorithms have the goal to find a target error e having in input a binary parity-check matrix H, a syndrome s and a weight w. First, let's define what is an information set: having a parity-check matrix $H \in \mathbb{F}_2^{r\times n}$, we define **IS** as an information set with size k if and only if rank $H_{\mathbf{IS}^*} = |\mathbf{IS}^*| = n - k = r$ where $\mathbf{IS}^* = \{0, \ldots, n-1\} \setminus \mathbf{IS}$.
All the algorithms have the same basic structure: they choose an information set **IS** with size k of H that divides the error $\mathbf{e}$ in two parts, $\boldsymbol{e}_{\mathbf{S}^*}$ and $\boldsymbol{e}_{\mathbf{S}}$, where $\boldsymbol{e}_{\mathbf{S}^*}$ are the bits in $\boldsymbol{e}$ indexed by the information set $\mathbf{IS}^*$, while $\boldsymbol{e}_{\mathbf{S}}$ are the ones indexed by **IS**, and then try to guess in the $\boldsymbol{e}_{\mathbf{S}}$ part a certain weight p. The phase of choosing an information set is simply done transforming the matrix H in input in reduced row echelon form obtaining $\widehat{H} = [I_r \quad V]$, while the guessing part depends on the specific ISD chosen. In Figure 1 we can see the syndrome decoding problem structure after applying the RREF method with the error splitting thanks to the information set found.

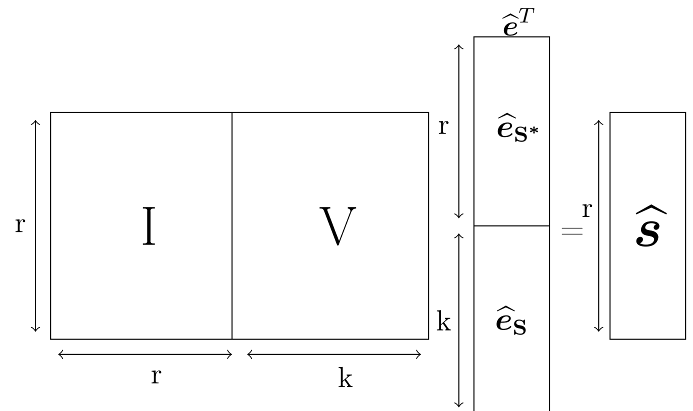


Figure 1: Situation after calling the RREF where an information set has been found

Different methods for computing the RREF of a binary matrix has been taken in consideration with possible optimizations: all of them start with a random shuffling of the matrix columns and then, try to transform the left sub-matrix of the permuted one into an identity matrix. The original syndrome decoding problem has been transformed in an equivalent problem composed of a transformed matrix $\widehat{H} = [I_r \quad V]$ and a transformed syndrome $\widehat{s} = Us^T$, where U is a matrix holding all the row operations applied to H during the RREF transformation. At the end, the error vector relative to the original problem, is obtained applying the permutation initially performed to H to the error vector retrieved from the transformed problem.
After computing a correct systematic form, each ISD algorithm needs to recover the target error: the simplest ones follow a bruteforce technique for guessing p positions set to 1 in $\widehat{\boldsymbol{e}}_{\mathbf{S}}$: the guess is correct if and only if $\mathtt{HW}(\widehat{\boldsymbol{e}}_{\mathbf{S}^*}) = w - \mathtt{HW}(\widehat{\boldsymbol{e}}_{\mathbf{S}}) = w - p$ where $\mathtt{HW}(x)$ indicates the Hamming weight of the binary vector x. When a right guess is found, we obtain the permuted error as the concatenation between $\widehat{\boldsymbol{e}}_{\mathbf{S}^*}$ and $\widehat{\boldsymbol{e}}_{\mathbf{S}}$. If the algorithm hasn't found the wanted target error after it has tried all the possible guesses, it needs to choose another information set calling another time the RREF procedure picking a new initially random permutation and restart the guessing procedure with a new transformed matrix and syndrome. The Information Set Decoding algorithms are probabilistic since their output depends on the permutation picked for computing the RREF form of H. For this reason, each ISD tries to retrieve the error vector repeating a certain num-

ber of times an attempt whose average value depends on the success probability of the single attempt itself. The computational complexity of all the ISD algorithms has the following form:

$$C_{ISD}(n,r,w) = \frac{c_{iter}}{Pr_{succ}} = \\ = \frac{1}{Pr_{succ}}(C_{IS}(n,r) + C_{SEARCH}(n,r,w))$$

where c_{iter} is the complexity of each attempt composed by the computation of the RREF $C_{IS}(n,r)$ and the searching of the error $C_{SEARCH}(n,r,w)$, while Pr_{succ} denotes the success probability of a single attempt. The difference between all the Information Set Decoding algorithms is how the error is retrieved after the call to the RREF, therefore, how $C_{SEARCH}(n,r,w)$ changes. In the following there is a brief description for each ISD algorithm that has been implemented.

Prange. Prange was the first variant of ISD designed and it is based on the idea of guessing a set k of error-free positions in the target vector; from Figure 1 it guesses that $\mathtt{HW}(\widehat{\boldsymbol{e}}_{\mathbf{S}}) = 0$. Therefore, after the computation of the RREF it simply checks if the permuted syndrome $\widehat{\boldsymbol{s}}$ has weight equal to w. When this happens, the original error is obtained applying the permutation to the permuted error $\widehat{\boldsymbol{e}} = [\widehat{\boldsymbol{s}} \quad \mathbf{0}_{1\times k}]$.

Lee-Brickell. The Lee-Brickell algorithm improves Prange allowing p positions set to 1 in the second part of the permuted error obtaining: $\mathtt{HW}(\widehat{\boldsymbol{e}}_{\mathbf{S}}) = p$. It considers all the possible combinations of k size vectors with weight equal to p and for each combination it checks if $\mathtt{HW}(\widehat{\boldsymbol{e}}_{\mathbf{S}^*}) = w - p$. If all the possible combinations have been tested without finding the target error, a new RREF computation is done picking a different permutation and the previous step is repeated with the new matrix.

Leon. The Leon algorithm improves the Lee-Brickell's one assuming that the contribution to the value of the first ℓ bits of the syndrome $\widehat{\boldsymbol{s}}$, $\widehat{\boldsymbol{s}}_{\mathbf{up}}$, comes only from columns in V: this means that there is a run long ℓ bits of zeroes in the first part of $\widehat{\boldsymbol{e}}_{\mathbf{S}^*}$. It performs a pre-check on the ℓ long vector to control if its weight is equal to zero before checking $\mathtt{HW}(\widehat{\boldsymbol{e}}_{\mathbf{S}^*}) = w - p$.

Stern. Stern's algorithm improves Leon's ISD by employing a meet-in-the-middle strategy for finding which set of size p, containing ℓ bit portions of columns of V, adds up to the first ℓ bits of the syndrome. The part of the permuted error with weight p, $\widehat{\boldsymbol{e}}_{\mathbf{S}}$, is splitted into two binary vectors with weight $\frac{p}{2}$. It saves inside a list or an hashtable all the possible $\binom{k/2}{p/2}$ vectors long ℓ with weight $p/2$ relative to the first part of the upper matrix of V and then enumerates all the possible $\binom{k/2}{p/2}$ vectors relative to the second part of the upper matrix of V and tries to find these latter vectors inside the list or the hashtable. The list needs to be sorted and then a binary range search is applied to find collisions. If a match is found, exists a valid candidate pair for constructing $\widehat{\boldsymbol{e}}_{\mathbf{S}}$, and so, it can go on controlling if the lower part has weight equal to $w - p$ to build the complete permuted error.

Ball-Collision Decoding. The Ball-Collision Decoding algorithm applies the same steps as the Stern but allowing z error affected positions in the ℓ part of the permuted error. Also the ℓ part of the error is splitted into two binary vectors with weight $z/2$ and the same procedures of the Stern is applied for populating the list or the hashtable, each time considering all the possible $\binom{k/2}{p/2}\binom{\ell/2}{z/2}$ binary vectors.

Finiasz-Sendrier. The Finiasz-Sendrier algorithm improves the Stern's algorithm removing the ℓ window of zeroes in the permuted error and moving this ℓ region in the part of the error where we need to guess p error bits. Since the p positions to be guessed are picked among the last $k + \ell$ position of the error vector and not among the last k as in the previous ISDs, we need to have only an identity matrix of size $(r - \ell) \times (r - \ell)$ on the upper leftmost portion of $\widehat{H}$ obtaining $\widehat{H} = \begin{bmatrix} I_{r-\ell} & V_{\text{up}} \\ \mathbf{0}_{(r-\ell)\times\ell} & V_{\text{down}} \end{bmatrix}$. This form is obtained computing a partial systematic form of H. Then, the same steps of Stern are done, building the list or the hashtable with all the possible $\binom{(k+\ell)/2}{p/2}$ binary error vectors.

May-Meurer-Thomae. The May-Meurer-Thomae (MMT) algorithm improves the Finiasz-Sendrier's algorithm changing the way in which the p positions of the vector are chosen. Instead of splitting them equally as $\frac{p}{2}$ in the lefmost $\frac{k+\ell}{2}$ columns and $\frac{p}{2}$ in the rightmost $\frac{k+\ell}{2}$ ones, the algorithm picks two disjoint sets $\alpha, \beta \subset \{0, \dots, k+\ell-1\}$. Therefore, the May–Meurer–Thomae algorithm exploits a

time to memory trade-off like the one employed by Stern, applying twice the precomputation strategy. For building the p weight part of the error two layers of lists are used: at layer 1 two lists hold binary vectors with weights $p/2$ that have been retrieved from the four lists at layer 2 holding binary vectors with weight $p/4$.

Becker-Joux-May-Meurer. The BJMM algorithm considers that it is possible to represent the error with weight p, $\widehat{\boldsymbol{e}}_{\mathbf{S}}$, as the sum of two error vectors $\widehat{\boldsymbol{e}}_1$ and $\widehat{\boldsymbol{e}}_2$ with weight equal to $\frac{p}{2} + \Delta$ under the assumption that the extra Δ ones cancel out during the addition. It uses three layers instead of two obtaining at the first layer two lists holding binary vectors with weight $p_1 = \frac{p}{2} + \Delta_1$, at the second four lists holding vectors with weight $p_2 = \frac{p_1}{2} + \Delta_2$ and at the last layer eight lists holding vectors with weight $p_3 = \frac{p_2}{2}$.

Both-May. The Both-May algorithm takes the idea of the BJMM algorithm but goes back computing a full systematic form. It uses two layers of lists and it applies a nearest neighbor search technique for their construction instead of sorting the lists for applying a binary range search. The algorithm doesn't want to find an exact matching between binary vectors but it wants to find pairs of vectors such that, given a target weight w_i, their sum has a weight equal to w_i (approximate matching problem). Both-May algorithm works with a p part of the permuted error long k and not $k + \ell$ like the last algorithms we have seen.

Esser-Bellini. The Esser-Bellini algorithm is a variant of the Both-May algorithm presented in [4]. The algorithm is almost equal to the Both-May algorithm, the only thing that changes is the application of the Indwik-Motwani search instead of the nearest neighbor search to build the lists at the various layers. In a nutshell, before checking the weight w_i of the resulting sum for solving the approximate matching problem, the two binary vectors must be equal in some λ coordinates picked at random. The indwik-motwani search relies on the fact that the sum between the two vectors will have small weight, therefore, for a certain number of λ coordinates, is more likely that on these coordinates the two vectors have the same values with respect to the situation where the sum has larger weight.

4. RREF Testing

The first step of each ISD algorithm as we have seen is the computation of the RREF of H. Different procedures computing a full RREF or a partial RREF have been taken in consideration and they have been tested to understand which one performs best: the standard RREF, the RREF with reusing pivots, the partial RREF and the partial RREF optimized have been implemented from scratch while the method of M4RI is available publicly in the M4RI library based on the work by Gregory V.Bard in [3]. We have taken a set of challenges having different dimensions of the parity-check matrix to stress the procedures with increasing sizes of H. The matrices tested have rate $R = \frac{n}{k} = 0.5$ and their sizes go from the minimum $r \times n = 25 \times 50$ to the maximum 500×1000. For each code we call 30 times the RREF procedure in consideration and we save the average value of the cost complexity measured in cpu cycles spent for computing the RREF. On the following plots we can see that the M4RI method is the most efficient.

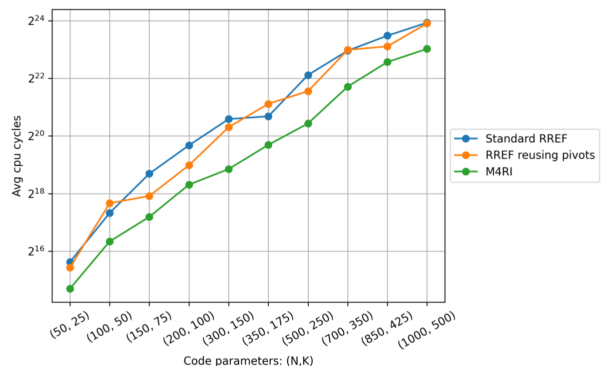


Figure 2: Avg cpu cycles comparison between different RREF procedures

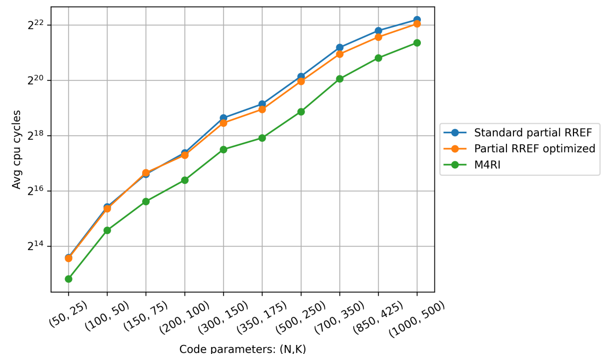


Figure 3: Avg cpu cycles comparison between different partial RREF procedures

5. ISD Algorithms Testing

In this section we are going to present the results obtained by the testing of the implementation of all the ISD algorithms. Two different family of tests have been considered. The first test case is composed by instances of the the so called "Syndrome Decoding" that we can find at [1]. In these tests the codes taken in consideration have a rate $R = 0.5$ and the weight w close to the Gilbert-Varshamov distance.
In the second tests the code rate R is equal to 0.8 resulting in $k = 0.8n$ while the weight of the target error to find is $w = \lceil \frac{n}{5\lceil \log(n) \rceil} \rceil$. The tests in input are taken in the "Syndrome Decoding in the Goppa-McEliece Setting" section at [1] and they are intended to be as close as possible to the problem on which relies the Classic McEliece cryptosystem proposed for the NIST standardization process. Under these conditions, instances with cryptographic size are assumed to be out of reach, so we have considered instances with increasing size, as in the previous tests, to see how hard this problem is in practice. Since different test cases using different linear codes properties have been considered, we have the need to tune the parameters of the ISD algorithms properly (like the parameter p used from Lee-Brickell or the parameter ℓ used in Leon). The optimal parameters of the ISD algorithms have been chosen using two estimators already available: the Ledatools estimator described in [2] and the Esser-Bellini estimator described in [4]. In the following plots, we can see the average computational cost measured in cpu cycles of all the ISD algorithms implemented for the two family of tests taken in consideration.

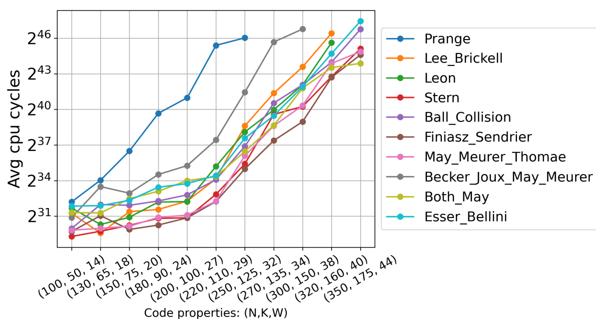


Figure 4: Avg cpu cycles comparison between all the ISD algorithms (syndrome tests)

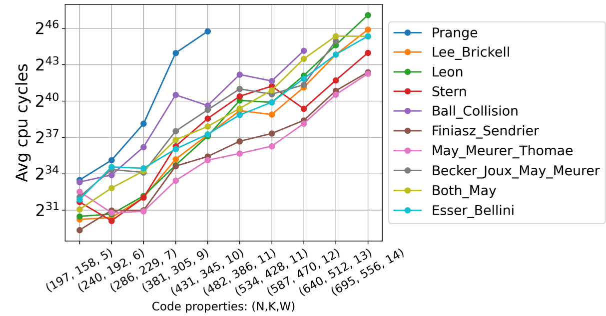


Figure 5: Avg cpu cycles comparison between all the ISD algorithms (mceliece tests)

For both the tests the measured cost complexity is in line with the estimated cost complexity obtained multiplying the cost of a single iteration with the expected number of iteration for finding a target error. The latter term is the reciprocal of the success probability that we have seen in the formula of the computational complexity of the ISD and it depends on which ISD we are using. The cost of a single iteration of an ISD has been measured as the average cost taken from 30 runs of the algorithm: for each test we execute the ISD algorithm only for one iteration not worrying about finding the target error but focusing on the cost of a single iteration (one RREF computation and one call to the search error part). Knowing that, it is possible to estimate the complexity of the ISD algorithms even for tests that could be run for months before finding the target error. In the following plots we can see the estimated cost complexity for complex syndrome tests and mceliece tests for all the ISD implemented.

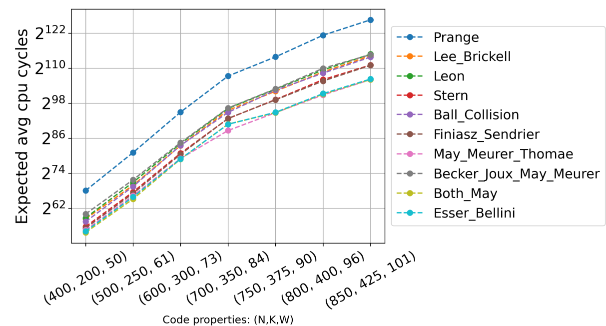


Figure 6: Estimated average cost complexity for complex tests (syndrome tests)

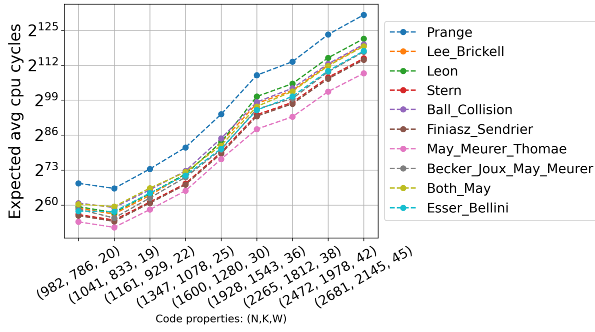


Figure 7: Estimated average cost complexity for complex tests (mceliece tests)

We can see how with the increasing of the size of the tests the cost for finding an error increases exponentially: the syndrome decoding problem, as we expected, starts becoming intractable with greater code properties. The NIST has proposed five security categories for the McEliece cryptosystem: a parameter set matches the security level of the category one, three or five if the scheme instantiated with this set is at least as hard to break as AES-128, AES-192 or AES-256, respectively. For example, in category one the parameter set proposed is (3488,2720,64) while one set from category 5 is (6688,5024,128). We can conclude, thanks to the testing results, that the McEliece cryptosystem with the parameter sets proposed by the NIST is considered secure from the point of view of the Information Set Decoding algorithms. This is true because, as we can notice from the last plot, the syndrome decoding problem just with the smallest set proposed by the NIST will have a bigger complexity compared to the one retrieved by the best ISD algorithm with the largest test taken in consideration. Since the complexity of the syndrome decoding problem grows exponentially, the problems with the parameter sets in category 3 and 5 will be more complex and intractable too.

6. Conclusions

In this thesis we focused on how to solve the syndrome decoding problem, the basis on which the code-based cryptosystems are founded on. This problem has been solved implementing the Information Set Decoding algorithms, the techniques with the best complexity currently known in the state of the art. We have done a concrete evaluation of the Information Set Decoding algorithms to understand their behaviour in practice. We have tested these algorithms with increasing tests size and then, computing the cost of a single iteration for each ISD, we have retrieved the estimated cost complexity for high dimension tests to see that the syndrome decoding problem starts becoming intractable as the code properties increase, confirming the security of the McEliece cryptosystem with the parameter sets proposed by the NIST.

References

[1] Nicolas Aragon, Julien Lavauzelle, and Matthieu Lequesne. decodingchallenge.org, 2019.

[2] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. A finite regime analysis of information set decoding algorithms. *Algorithms*, 12(10), 2019.

[3] Gregory V. Bard. Accelerating cryptanalysis with the method of four russians. Cryptology ePrint Archive, Report 2006/251, 2006. `https://ia.cr/2006/251`.

[4] Andre Esser and Emanuele Bellini. Syndrome decoding estimator. Cryptology ePrint Archive, Report 2021/1243, 2021. `https://ia.cr/2021/1243`.

[5] R. J. McEliece. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report*, 44:114–116, January 1978.