Università Politecnica delle Marche
Facoltà di Ingegneria
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Anlisi e proposta di algoritmi per trovare la distanza minima di codici LDPC

Improving Information Set Decoding for Low-Density Parity-Check codes

Relatore:
Dott. Paolo Santini

Tesi di laurea di:
Rahmi El Mechri

Anno Accademico 2023-2024

Sommario

Un codice a correzione d'errore ha capacità di correzione strettamente legate alla distanza minima tra le parole in codice che lo compongono, dette codeword. Per codici lineari, i più utilizzati nella pratica, la distanza minima è pari al peso della codeword non nulla con peso minore, ossia con il minor numero di simboli diversi da zero. Trovare la parola a peso minimo di codici generici è un problema NP-*hard*, ovvero, gli algoritmi esistenti impiegano un tempo che cresce esponenzialmente con la dimensione del codice considerato. La famiglia di algoritmi più utilizzata a questo scopo è Information Set Decoding (ISD). Informalmente, si può pensare agli algoritmi ISD come una variante degli attacchi a forza bruta, nel quale si fanno assunzioni sulla distribuzione dei simboli nelle codeword. In questa tesi proponiamo un algoritmo ISD, chiamato SparseISD, per trovare la parola a peso minimo di codici Low-Density Parity-Check (LDPC). Questi codici presentano una matrice di parità sparsa, cioè con la maggioranza degli elementi nulli. SparseISD sfrutta questa caratteristica per creare condizioni di ricerca favorevoli. Studiamo il problema presentando un ensemble di codici creato come generalizzazione dell'ensemble dei codici random, che sono tipicamente il bersaglio degli algoritmi ISD. Dimostriamo che i codici LDPC presentano una distribuzione di peso che tende a quella dei codici random. Presentiamo inoltre, una implementazione proof of concept di SparseISD, per dimostrarne il funzionamento.

Abstract

The error correction capabilities of an error correcting code are strictly related to the minimum distance between any two codewords in it. Therefore, finding the minimum distance of codes is of great importance. For linear codes, this can be done by finding the minimum non null codeword, which is a well known NP-*hard* problem. Information Set Decoding (ISD) is the most efficient family of algorithms for finding the minimum weight codeword of generic codes. We investigate the use of ISD algorithms with Low-Density Parity-Check (LDPC) codes, a family characterized by sparse parity-check matrices. A new ISD algorithm, called SparseISD, is proposed, together with a novel ensemble of codes that comprehends LDPC codes. We prove that these codes have a weight distribution analog to those of random codes, that are, conventionally, the target of ISD algorithms. SparseISD performs better than state of the art ISD algorithms when applied to codes from our ensemble. A proof-of-concept implementation is presented.

Contents

Acronyms

GV Gilbert-Varshamov Bound. 22

GWCP Given Weight Codeword Problem. 11, 17

ISD Information Set Decoding. ii, 2, 3, 12–14, 16–18, 20, 31, 32, 37–39, 49, 51

LDPC Low-Density Parity-Check. ii, iii, 2–4, 9–11, 39, 49, 51

MDPC Moderate-Density Parity-Check. iv, 10, 42

ML Maximum Likelihood. 11, 12

MLD Maximum Likelihood Decoding. 1, 11

PGE Partial Gaussian Elimination. 12, 15, 32

SDP Syndrome Decoding Problem. 11

SIMD Single Instruction Multiple Data. 50

List of Figures

List of Tables

1 Introduction

Error correcting codes, commonly referred to as codes, play a fundamental role in the world of telecommunications. When information is transmitted over a noisy communication channel, many kinds of disturbances can cause an alteration of the original message. Codes were born to make such errors less disrupting. The idea is simple: given a length-k string $\mathbf{m}$, defined over some alphabet, we add r redundant symbols to $\mathbf{m}$, and link them through some mathematical relationship. This step is called encoding, and the result is a length-n string $\mathbf{c}$ called codeword. Clearly, $n = k + r$. Codes can be thought of as a set of codewords. If on reception the received message $\mathbf{x}$ does not satisfy the defined relationship, then $\mathbf{x} \notin \mathscr{C}$, therefore some error happened. When the amount of error is limited, one might even be able to recover the original message. This second step is called decoding. It can be formulated as the problem of finding the codeword $\mathbf{c} \in \mathscr{C}$ that resembles $\mathbf{x}$ the most; this formulation takes the name of Maximum Likelihood Decoding (MLD).

The wide majority of error correcting codes used in practice are linear codes: a linear code $\mathscr{C}$ is a linear subspace of the vector field defined over the finite prime field $\mathbb{F}_q^n$. Then, considering additive communication channels, we can model received messages as $\mathbf{x} = \mathbf{c} + \mathbf{e}$, where $\mathbf{e}$ is the length-n additive error. We endow $\mathbb{F}_q^n$ with a metric to measure distances between its elements with some function $\mathrm{dist} : \mathbb{F}_q^n \to \mathbb{N}$. When communication channels with uniform probability of error across messages are considered, such as our case, MLD is equivalent as finding the closest codeword to the received message. Therefore, we can reformulate the problem as finding the codeword $\mathbf{c} \in \mathscr{C}$, such that $\mathrm{dist}(\mathbf{x}, \mathbf{c}) \leq \mathrm{dist}(\mathbf{x}, \mathbf{c}')$ for any $\mathbf{c}' \in \mathscr{C}$. A direct implication is that for each codeword $\mathbf{c}$ we can define a region in space $V(\mathbf{c}) \subset \mathbb{F}_q^n$, containing the elements that have $\mathbf{c}$ as closest codeword. This region coincides with the Voronoi region centered in $\mathbf{c}$. Hence, given a message $\mathbf{x}$, MLD consists in finding the center of the Voronoi region that contains $\mathbf{x}$. It is now easy to see that MLD fails when the error brings the message to a different Voronoi region, i.e., when $\mathbf{x} = \mathbf{c} + \mathbf{e}$, and $\mathbf{x} \in V(\mathbf{c}'), c \neq c'$. The smallest amount of error for which decoding might fail is $\mathbf{e} = \frac{\mathrm{d}_{min}}{2}$, where d_{min} is the minimum distance between any two codewords in the code. Guaranteed unique decoding for $\mathbf{e} > \left\lfloor \frac{\mathrm{d}_{min}-1}{2} \right\rfloor$ is impossible. This is an important bound on the error capacity of a code, and it is a measure of its decoding performances. For a linear code $\mathscr{C}$, the minimum distance is equivalent to the minimum Hamming weight of any non null codeword $\mathbf{c} \in \mathscr{C}$, that is the number of non zero symbols. It comes without saying, that knowing this bound is of great importance.

Searching for the nearest neighbor and finding the minimum weight for a generic code are NP-*complete* problems. This implies that, in the worst case, solving these problems takes a time exponential on the parameters of the code. For this reason, many codes are built with an underlying algebraic structure, such that polynomial-time decoding is possible. This is the case of the Berlekamp-Massey algorithm for Reed-Solomon codes, and Patterson algorithm for Goppa codes. Other codes use heuristics, that don't guarantee optimal decoding, but have great performances. This is the case of Low-Density Parity-Check (LDPC) codes.

LDPC codes are a family of codes introduced by Robert Gallagher in his PhD theis [4]. They are characterized by a parity-check matrix $\mathbf{H}$, used when decoding, that is sparse. This means that the wide majority of elements in $\mathbf{H}$ are null. A consequence of this property is that many computations can be skipped and, for this reason, these codes have efficient encoding and decoding algorithms. Decoding, in particular, uses fast iterative algorithms based on graphs, called message-passing algorithms, Together with their great error correction capabilities, this makes LDPC one of the most studied family of codes. In spite of this, little is known about algorithms for finding the minimum distance of such codes. This is because their construction relies on heuristic procedures, that do not guarantee specific code properties. As a result, existing algorithms are not only exponential, but are also solely based on heuristics.

The best known family of generic decoders is Information Set Decoding (ISD). These algorithms can be easily tweaked to search for the minimum distance of a code. The first ISD algorithm was proposed in 1962 by Eugène Prange [8]. Many improvements were later proposed, but all algorithms have the same common structure. An ISD algorithm is a randomized algorithm that takes as input a parity-check matrix $\mathbf{H}$ and a weight w. It performs transformations on $\mathbf{H}$, and searches for codewords of weight w that respect a defined weight distribution. It can be thought of as a smart brute force attack, where instead of performing exhaustive search on the entire search space, a probabilistic distribution is used. Its running time depends on the used parameters, and on the input parity-check matrix. When one possesses some information about the structure of the input code, the procedure can be sped up. Matter of fact, ISD algorithms are designed to attack code-based cryptographic schemes, where codes are hidden in such a way that appear as random codes, so that no assumption can be made on the structure beneath. Currently, security parameters for many code-based cryptographic schemes are based on the computational complexity of attacking them using these algorithms.

1.1 Our contribution

In this thesis we propose an ISD variant, based on the existing ISD algorithm proposed by Jacques Stern [9]. Stern algorithm is arguably one of the most used and studied ISD variant, because of its good compromise between time and space complexity. It is based on finding collisions between lists trough the meet-in-the-middle approach. Our algorithm, which we named SparseISD, is tailored to find the minimum distance of LDPC codes. It exploits the sparsity of parity-check matrix to create favorable conditions when performing collision search. We provide an asymptotic analysis of the computational complexity of our algorithm.

Before describing SparseISD, we present a novel ensemble of binary codes that comprehends the family of LDPC codes. The model used for random codes was generalized, defining ensembles of parity-check matrices with elements sampled from the binomial distribution with probability ν. Proofs about important properties of codes in our ensemble are provided, in particular, we show that when ν grows with n faster than $O(\frac{\ln(n)}{n})$, our codes follow the same weight distribution of random codes. This is instrumental to our results, since ISD algorithms are usually studied for random codes, and as ν approaches $\frac{1}{2}$ our ensembles tends to the one of random codes. Doing so, we were able to study the minimum distance of our codes and consequently, analyze the performances of ISD. The proposed algorithm, and the proposed ensemble, were validated through numerical experiments. We provide a comparison of the average computational complexity of SparseISD and Stern algorithm, when applied to code from our ensemble. More precisely, we analyzed the cases of $\nu = \frac{\ln(n)^2}{n}$ and $\nu = \frac{\sqrt{n}}{n}$. On average, our algorithm performs better than Stern algorithm, especially for higher code rates. For $R = 0.8$, we were able to obtain an asymptotic speedup of $2^{21}\times$.

Finally, we have implemented a proof-of-concept implementation of SparseISD, performing tests on small-sized codes with known-distance from our ensemble for validation.

1.2 Thesis organization

The remainder of the thesis is organized as follows:

- In **Chapter 2** we provide the necessary background knowledge on computational complexity and coding theory;
- **Chapter 3** introduces a novel ensemble of binary codes, comprehending LDPC, along with proofs on its properties;
- We formally describe SparseISD in **Chapter 4**, providing an asymptotic analysis of its average computational complexity;
- **Chapter 5** discusses important characteristic of our proof-of-concept implementation, and shows the results of tests performed on small-sized codes;
- **Chapter 6** provides a critical analysis of the work done, outlying the direction for future developments;
- Finally, in **Chapter 7** we summarize the key contributions in this thesis.

2 Preliminaries

2.1 Notation

2.1.1 Mathematical notation

We use p to denote a prime number, and q a prime power $q = p^m$, m a positive integer. We denote by $\mathbb{F}_q$ the finite field of order q, and by $\mathbb{F}_q^n$ the vector field whose elements are length-n vectors with values in $\mathbb{F}_q$. We denote the order of a given set A as $|A|$. To denote matrices and vectors we use respectively bold uppercase letters and bold lowercase letters. Given a matrix $\mathbf{M}$, we denote by $\mathbf{M}^\top$ its transpose. We denote both the null matrix and the null vector as $\mathbf{0}$, with the dimension made clear from context. The identity matrix of size k is indicated by $\mathbf{I}_k$.

$\mathscr{P}_n$ is used to indicate the group of length-n permutations. Let $\pi \in \mathscr{P}_n$ and $a = (a_1, \cdots, a_n)$, we have $\pi(a) = (a_{\pi^{-1}(1)}, \cdots, a_{\pi^{-1}(n)})$. Given an event Event the probability that it happens is denoted as $\mathbb{P}[\text{Event}]$. Let X be a random variable, we use $\mathbb{E}[X]$ to denote its mean value. We express that x is sampled from a distribution $\mathcal{D}$ with $x \sim \mathcal{D}$. We write $x \overset{\$}{\leftarrow} A$ to denote that x is sampled uniformly at random from A.

2.1.2 Asymptotic notation

When describing the asymptotic behavior of functions we make extensive use of Landau's notation. For functions $f(n)$ and $g(n)$, with n positive, we write

- $f(n) \sim g(n)$ if
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1;$$

- $f(n) = o(g(n))$ if
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0;$$

- $f(n) = O(g(n))$ if $\exists c, n_0$ positive such that $\forall n \geq n_0$
$$|f(n)| \leq c \cdot |g(n)|;$$

- $f(n) = \Omega(g(n))$ if $\exists c, n_0$ positive such that $\forall n \geq n_0$
$$|f(n)| \geq c \cdot g(n).$$

For the binomial coefficient we will use the following two asymptotic expansions:

- if $w = o(n)$ then
$$\log_2 \binom{n}{w} = w \cdot \log_2 \left(\frac{n}{w}\right) \cdot (1 + o(1));$$

- if $w = \Omega(n)$ then
$$\log_2 \binom{n}{w} = n \cdot h\left(\frac{w}{n}\right) \cdot (1 + o(1));$$
Where
$$h(x) = -x \log_2(x) - (1-x)\log_2(1-x)$$
is the *binary entropy function.*

2.1.3 Computational complexity

We say that a problem Π is NP if a nondeterministic Turing machine can solve it in polynomial-time. NP-*complete* is a subclass of problems in NP (NP-*complete* $\subseteq$ NP) such that for every $\Pi \in$ NP-*complete*, there exists a polynomial-time reduction from every problem in NP to Π. Currently, there is no known algorithm that runs in polynomial time on a deterministic Turing machine, capable of solving any of the problems in NP-*complete*. If such an algorithm were to exist, then the entire NP class would collapse into P, the class of problems for which such an algorithm exists. It is speculated that P $\neq$ NP, but this has not been proven. Nevertheless, the complexity class refers to worst-case instances. Matter of fact, a problem in NP-*complete* may admit an average-case polynomial-time algorithm. From here on, unless stated otherwise, when we refer to complexity, we will be referring to deterministic machines, since, with the necessary simplifications, they model the computers we use.

2.2 Error Correcting Codes

A code is a subset $\mathscr{C} \subseteq \mathbb{F}_q^n$. Elements in this subsets are called *codewords*. If the subset $\mathscr{C}$ is a linear subspace of $\mathbb{F}_q^n$, then we refer to it as *linear code* of length n. More specifically:

Definition 2.2.1 (Linear Code)**.** Let $k, n \in \mathbf{N}, 1 \leq k \leq n$. An $[n, k]$ *linear code* $\mathscr{C}$ over $\mathbb{F}_q$ is a linear subspace of $\mathbb{F}_q^n$ of dimension k.

A linear code can be represented by one of its *generator matrices* **G** or the corresponding *parity-check matrix* **H**, formally defined as:

Definition 2.2.2 (Generator Matrix)**.** A *generator matrix* $\mathbf{G} \in \mathbb{F}_q^{k \times n}$ of an $[n, k]$ linear code $\mathscr{C}$ is a matrix that has the code as image.

$$\mathscr{C} = \left\{\mathbf{xG} | \mathbf{x} \in \mathbb{F}_q^k\right\} \tag{2.1}$$

Definition 2.2.3 (Parity-Check Matrix)**.** A *parity-check matrix* $\mathbf{H} \in \mathbb{F}_q^{n-k \times n}$ of an $[n, k]$ linear code $\mathscr{C}$ is a matrix that has the code as kernel.

$$\mathscr{C} = \left\{\mathbf{x} \in \mathbb{F}_q^n | \mathbf{xH}^\top = \mathbf{0}\right\} \tag{2.2}$$

Theorem 2.2.1.
If $\mathbf{G} = [\mathbf{I}_k | \mathbf{A}]$ *is a generator matrix of an* $[n, k]_q$ *code* $\mathscr{C}$*, then* $\mathbf{H} = \left[-A^\top | I_{n-k}\right]$ *is a parity check matrix of* $\mathscr{C}$*.*

Given a linear code $\mathscr{C} \in \mathbb{F}_q^n$, a generator matrix **G** for $\mathscr{C}$ specifies an encoder for the code, as by eq. (2.1), given a word $\mathbf{u} \in \mathbb{F}_q^k$, the corresponding codeword $\mathbf{c} \in \mathscr{C}$ is $\mathbf{c} = \mathbf{uG}$. Given a parity-check matrix **H** for $\mathscr{C}$ and a vector $\mathbf{x} \in \mathbb{F}_q^n$, to verify if **x** is a codeword of the given code it is sufficient to check if eq. (2.2) is satisfied, that is verifying if multiplying **x** by the transpose of **H** yields a zero vector. The result of this operation is called *syndrome*. When the syndrome is nonzero it can be inferred that some error $\mathbf{e} \in \mathbb{F}_q^n$ was applied to the original message **c**.

$$\mathbf{x} = \mathbf{c} + \mathbf{e}, \quad \mathbf{x}, \mathbf{e} \in \mathbb{F}_q^n, \mathbf{c} \in \mathscr{C} \tag{2.3}$$

Under certain conditions the original codeword can be recovered, using special algorithms called decoders, presented in section 2.3. Error correction is possible due to the presence of $r = n - k$ redundancy symbols in codewords. The ratio between the length n and the dimension k of a linear code is called *code rate* R, and it represents the rate of information bits in transmitted bits when the code is used for communication.

The capability of a code to correct errors is closely related to distances between codewords. To measure distances between elements in a vector space, a distance function, or *metric*, must be defined. We endow $\mathbb{F}_q$ with the *Hamming metric.*

Definition 2.2.4 (Hamming distance)**.** Let $\mathbf{x}, \mathbf{y} \in \mathbb{F}_q^n$. The *Hamming distance* $\mathrm{d_H}(\mathbf{x}, \mathbf{y})$ is the number of coordinates in which $\mathbf{x}$ and $\mathbf{y}$ differ.

Definition 2.2.5 (Hamming weight)**.** Let $\mathbf{x} \in \mathbb{F}_q^n$. The *Hamming weight* $\mathrm{wt}_H(\mathbf{x})$ is the number of nonzero coordinates of $\mathbf{x}$.

For the sake of simplicity for the remainder of this dissertation distance d and weight wt will be used as synonyms to denote respectively Hamming distance $\mathrm{d_H}$ and Hamming weight wt_H.

Definition 2.2.6. (Minimum distance) Let $\mathscr{C}$ be a linear code. The minimum distance between any two codewords d_{min}, such that

$$\forall \mathbf{c}, \mathbf{c}' \in \mathscr{C}, \mathbf{c}' \neq \mathbf{c}' \quad d_{min} \leq \mathrm{d}(\mathbf{c}, \mathbf{c}')$$

is known as *distance of $\mathscr{C}$*.

We can refer to an $[n, k]$ linear code with known distance d as $[n, k, d]$ linear code.

For each codeword $\mathbf{c} \in \mathscr{C}$ we can define the *Voronoi region* $V(\mathbf{c}) \subseteq \mathbb{F}_q^n$, that is the region in space such that for any $\mathbf{x} \in V$, $\mathbf{c}$ is the closest codeword.

$$\forall \mathbf{x} \in V(\mathbf{c}), \forall \mathbf{c}' \in \mathscr{C}, \mathbf{c}' \neq \mathbf{c}', \quad \mathrm{d}(\mathbf{x}, \mathbf{c}) < \mathrm{d}(\mathbf{x}, \mathbf{c}') \tag{2.4}$$

The operating principle of many decoders is to find the closest codeword when a vector with nonzero syndrome is received. For this reason, the larger these regions are, the higher the weight $\text{wt}(\mathbf{e})$ of error $\mathbf{e}$ that can be applied to a codeword $\mathbf{c}$ without ending in the region of a different codeword $V(\mathbf{c}')$. The radius of the smallest of these regions constraints the amount of error the code is capable to correct, and it can be determined from the distance of the code d.

Proposition 2.2.1. ***Error Correction Capacity***
Let $\mathscr{C} \subseteq \mathbb{F}_q^n$ be a $[n, k, d]$ linear code. Then, the code can correct up to $\lfloor \frac{d-1}{2} \rfloor$.

The ratio $\frac{d}{n}$ is called *relative distance*, and it is a measure of the error correction capability of a given code.

Theorem 2.2.2.
Let $\mathscr{C}$ be a linear code, d_{min} its minimum distance, w_{min} the minimum weight of any nonzero codeword $\mathbf{c} \in \mathscr{C}$, then

$$d_{min} = w_{min}.$$

It follows that the error correction capacity of a linear code can be determined by finding its minimum weight codeword, as the minimum distance can be inferred from this.

We denote by $\mathscr{S}_{n,w}$ the *Hamming sphere* with radius w, that is the set of length-n vectors with weight w.

The weight distribution of a code $\mathscr{C}$ is given by $\{m_{\mathscr{C}}(0), m_{\mathscr{C}}(1), ..., m_{\mathscr{C}}(n)\}$, where $m_{\mathscr{C}}(w) \subseteq \mathscr{S}_{n,w}$ counts the number of weight-w codewords in $\mathscr{C}$.

2.2.1 Low-Density Parity-Check Codes

LDPC codes are a family of codes introduced by R. G. Gallagher in his 1963 PhD dissertation [4]. We define LDPC formally:

Definition 2.2.7. A **Low-Density Parity-Check (LDPC) code** $\mathscr{C}$ is a linear block code characterized by a sparse parity-check matrix $\mathbf{H}$.

This means that $\mathbf{H}$ has a wide majority of zeroes and relatively few nonzero elements. This feature allows for fast encoding and decoding algorithms, as many computations can be skipped. This property can also be seen when representing **H** using a *Tanner graph.*

Definition 2.2.8. A **Tanner graph** is a bipartite graph G with vertex set $V \cup W$, $V \cap W = \emptyset$ created from a parity-check matrix **H**. Vertices in V and W correspond respectively to the columns and rows of H. Two nodes $v_i \in V$ and $w_j \in W$ are connected by an edge if and only if the $(s,t)^{th}$ entry of **H** is nonzero.

Tanner graphs can be used to represent any parity-check matrix, but when it is used to represent LDPC the resulting graph is sparse too, meaning the nodes in it have low degree (are connected to a few nodes). In other words the graph presents few relatively few edges. An example of Tanner graph can be seen in fig. 2.1. **H**

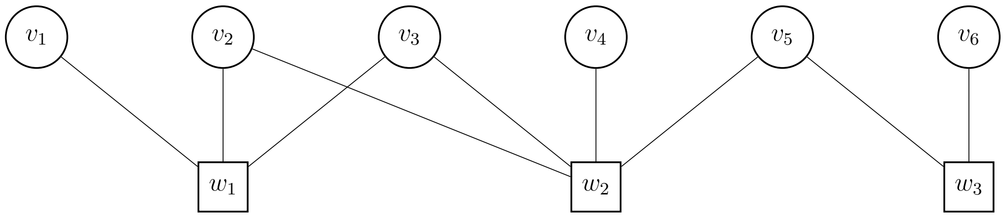


Figure 2.1: Example of Tanner Graph

sparseness can be quantified in different ways. We will use γ to denote the density of H, which is the ratio between the number of ones and the number of elements in the matrix. Thanks to this property encoding and decoding algorithms for LDPC codes achieve great error correction capabilities and are very efficient. For these reasons they are widely employed in many communication standards. Typically, the number of nonzero elements per row in LDPC is $O(\log(n))$. In the [1] cryptographic scheme MDPC are used instead, that are codes with a sparse parity-check matrix but with a higher number of nonzero elements than typical LDPC codes.

Definition 2.2.9. A **Moderate-Density Parity-Check (MDPC) code** $\mathscr{C}$ is a linear block code characterized by a parity-check matrix **H** with row weight $O\left(\sqrt{n}\right)$.

2.3 Decoding

In coding theory, decoders are algorithms capable of correcting received messages in a communication channel, yielding the original sent message. Put formally, a decoder is an algorithm that is capable of determining the codeword **c** when a vector $\mathbf{y} = \mathbf{c} + \mathbf{e}$ is received, **e** being the applied error on the codeword over the communication channel. To be precise will be considering *hard-decision* decoding, meaning that a received word $\mathbf{y}$ is defined over the same alphabet as the transmitted codeword $\mathbf{c}$.

The codeword that is most likely to have been transmitted is the closest to the received word. For this reason the problem of finding such a codeword is also called *MLD*, and decoders based on determining the closest codeword are called *Maximum Likelihood (ML) decoders*. When the decoding problem is expressed using the parity-check matrix $\mathbf{H}$, it is reffered to as *Syndrome Decoding Problem (SDP)*, and it can be stated as follows:

Problem 1. ***Syndrome Decoding Problem (SDP)***
Let $\mathbf{H} \in \mathbb{F}_q^{r\times n}$, $\mathbf{s} \in \mathbb{F}_q^r$ *and* $\mathbf{w} \in \mathbb{N}$. *Determine if there exists a vector* $\mathbf{e} \in \mathbb{F}_q^n$ *such that* $\mathrm{wt}(\mathbf{e}) = w$ *and* $\mathbf{He}^\top = \mathbf{s}$.

SDP was proven to be NP-*complete* after a polynomial-time reduction from the Three-Dimensional Matching problem was found in 1978 by Berlekamp, McEliece and van Tilborg [2]. In the same paper they have conjectured that the problem of finding the minimum distance of a linear code belongs to the NP-*complete* class too. This was later proven by Vardy in 1997 [10]. The problem is formulated as the decision problem of finding a codeword with a given weight w, denoted as *Given Weight Codeword Problem (GWCP)*:

Problem 2. ***Given Weight Codeword Problem (GWCP)***
Let $\mathbf{H} \in \mathbb{F}_q^{r\times n}$, *and* $\mathbf{w} \in \mathbb{N}$. *Determine if there exists a vector* $\mathbf{c} \in \mathbb{F}_q^n$ *such that* $\mathrm{wt}(\mathbf{c}) = w$ *and* $\mathbf{Hc}^\top = \mathbf{0}$.

To find the minimum distance is sufficient to iterate an algorithm for GWCP, increasing w until an affirmative answer is obtained. SDP and GWCP were proven to be equivalent problems. A proof of such equivalence can be found in [11]. Note that this results implies that an algorithm that finds the minimum distance for *any* class of code is expected to run on exponential time, but it doesn't say anything on the complexity for a specific class, such as finding the minimum distance for LDPC codes. For what concerns decoding, algorithms based on the geometrical task of finding the closest codewords are indeed impractical for most codes. Decoding is typically done using algorithms that are based on the inherent structure of the different families. Examples of such algorithms are Belief Propagation [4] for LDPC codes, Patterson algorithm [6] for Goppa codes, and the Berlekamp-Massey algorithm [3] for Reed-Solomon codes, all of which run in polynomial-time, although, unlike ML decoders, they might not achieve the optimal solution.

2.4 Information Set Decoding

Information Set Decoding (ISD) is a family of generic decoders that can operate on any linear code. With slight modifications these algorithms can be used to search for the minimum distance of a code. The name is derived from *information sets*:

Definition 2.4.1 (Information Set)**.** Let $\mathscr{C}$ be an $[n, k]$ linear code with generator matrix $\mathbf{G} \in \mathbb{F}_q^{k\times n}$. For any set of k independent columns of $\mathbf{G}$, the corresponding set of coordinates forms an information set. Given the corresponding parity check matrix $\mathbf{H} \in \mathbb{F}_q^{r\times n}$, for any set of of $n - k$ independent columns of $\mathbf{H}$ the set of coordinates of the complementary k columns forms an information set.

ISD algorithms receive as input a description for the code, typically the parity-check matrix $\mathbf{H}$ and the desired weight w, and return an element X in the powerset of $\mathscr{C}_w$, i.e. a set of weight-w codewords in $\mathscr{C}$.

$$\begin{aligned} \mathrm{ISD}\colon \mathbb{F}_q^{r\times n} \times \{0, \dots, n\} &\longrightarrow \mathscr{P}(\mathscr{C}_w), \\ (\mathbf{H}, w) &\longmapsto X. \end{aligned}$$

The first ISD algorithm was proposed by Prange in 1962 [8]. All subsequent proposed versions are an improvement to the original one and follow a common procedure, highlighted in algorithm 1. We can divide this procedure in three main steps:

- *Partial Gaussian Elimination (PGE)*: a random permutation π of length n is sampled. Then PGE with parameter $\ell \in \mathbb{N}, 1 \leq \ell \leq n - k$ is performed. This is the equivalent of performing a change of basis on the permuted parity-check matrix, in order to obtain a matrix with the following structure:

$$\left(\begin{array}{c|c} \mathbf{A} \in \mathbb{F}_q^{\ell\times(k+\ell)} & \mathbf{0} \in \mathbb{F}_q^{\ell\times(n-k-\ell))} \\ \hline \mathbf{B} \in \mathbb{F}_q^{(n-k-\ell)\times(k+\ell)} & \mathbf{I}_{n-k-\ell} \end{array}\right).$$

  Note that if the rightmost $n - k - l$ columns form a matrix whose rank is $< k$, then PGE cannot be performed. In these cases, a new permutation is sampled.

- *Solving the small instance*: after applying transformations in the previous step, we have:

$$\mathbf{c} = (\mathbf{c}', \mathbf{c}'') \in \pi(\mathscr{C}) \iff \begin{cases} \mathbf{A}\mathbf{c}'^\top = 0, \\ \mathbf{B}\mathbf{c}'^\top + \mathbf{c}''^\top = \mathbf{0}. \end{cases} \tag{2.5}$$

Notice how the first equation in eq. (2.5) implies that $\mathbf{c}'$ is a codeword of length $k+l$ of the code whose parity-check matrix is $\mathbf{A}$. The search for $\mathbf{c}'$ is restricted by assuming a defined weight distribution over the codeword, specifically a weight partition, and some low weight w. We will refer this step as the Solve subroutine.

- *Producing solutions*: once $\mathbf{c}'$ has been found, one can easily compute $\mathbf{c}''$ from the second equation of the linear system eq. (2.5). Codewords of the form $(\mathbf{c}', \mathbf{c}'')$ are produced, and the ones that have the desired weight w are added to the solution. Not that any such codeword corresponds to the permutation of a codeword in $\mathscr{C}_w$

The assumed weight partition during Solve is what really differentiates the different versions of ISD, and determines the probability of finding a codeword in each iteration. Matter of fact, the average number of iterations needed to find a codeword is function of the probability that the weight distribution is followed. For instance, in Prange algorithm, decoding is performed searching for an error vector that has nonzero elements outside of the information set.

Algorithm 1: General ISD structure

```
   Data: subroutine Solve, parameter $\ell \in \mathbb{N}$, $1 \le \ell \le r$
   Input: $\mathbf{H} \in \mathbb{F}_2^{r \times n}$, $w \in \mathbb{N}$
   Output: set $Y \in \mathscr{C}_w$
1  repeat
2      Sample $\pi \xleftarrow{\$} S_n$;
3      Apply PGE on $\pi(\mathbf{H})$;
4  until PGE Is successful;
5  $X = \mathsf{Solve}(\mathbf{A}, \ell)$;
6  Set $Y = \varnothing$;
7  for $\mathbf{c}' \in X$ do
8      Compute $\mathbf{c}'' = -\mathbf{c}'\mathbf{B}^\top$;
9      if $\mathrm{wt}(\mathbf{c}') + \mathrm{wt}(\mathbf{c}'') == w$ then
10         Update $Y \leftarrow Y \cup \{\pi^{-1}((\mathbf{c}0, \mathbf{c}''))\}$;
11 return $Y$;
```

The framework we have just used to describe ISD is a very general way to analyse algorithms within this family, but allows to identify the main quantities we will use for our analysis. We have yet not specified how the Solve subroutine is carried out, yet its functioning, that changes across ISD variants, heavily influences the computational cost of the whole procedure. For the moment, to keep this analysis as general as possible, consider that it will only return the codewords in $\pi(\mathscr{C}_w)$ that satisfy some constraints. We will denote this constraint with:

$$f_{\mathsf{ISD},\pi} : \mathbb{F}_q^{k+\ell} \to 0, 1$$

And assume that whenever $f_{\mathsf{ISD},\pi}(\mathbf{c}')$ is equal to 1, the codeword will be among the outputs of the subroutine.

2.4.1 Computational Cost

We analyse the cost of an ISD algorithm, considering a generic Solve subroutine. This later allows us to establish the computational cost of specific variant by describing the computational cost terms associated with the subroutine.

Crucial quantities for this evaluation are the success probability and the average number of codewords found for each iteration.

When considering generic codes, the probability that a chosen permutation is valid is only a function of:

(i) the desired weight $\mathbf{w}$;

(ii) the weight distribution constraints imposed by the considered ISD variant.

Proposition 2.4.1. ***Cost of one iteration***
On average, one iteration of ISD uses a number of elementary operations (sums and multiplications) over $\mathbb{F}_q$ counted by

$$O\left(\frac{n(n-k+l)^2 + \mathbb{E}[t_{\mathsf{Solve}}] + \mathbb{E}[|X|]}{p_{\mathsf{inv}(\ell)}}\right)$$

Where:

- $p_{\mathsf{inv}} = \prod_{i=\ell+1}^{n-k} 1 - q^{-i}$;

- t_{Solve} *is the cost of the subroutine* Solve*;*

- $|X|$ *is the number of solutions found for the small instance.*

Proof. Performing PGE requires a number of operations which is well counted by $n(n-k+\ell)^2$ (for instance, see [7]). The number of times we need to repeat the PGE step on average, corresponds to the reciprocal of the probability that the chosen permutation π places, on the rightmost side of $\pi(\mathbf{H})$, $n-k-\ell$ columns which form a basis for a space with dimension ℓ. Assuming that all columns of $\mathbf{H}$ behave as random vectors over $\mathbb{F}_q$, with length $n-k$, we get that this probability is

$$p_{\mathsf{inv}}(\ell) = \prod_{i=0}^{n-k-\ell-1}\left(1-\frac{q^i}{q^{n-k}}\right) = \prod_{i=0}^{n-k-\ell-1}\left(1-q^{-(n-k-i)}\right) = \prod_{i=\ell+1}^{n-k}\left(1-q^{-i}\right)$$

$\square$

Remark 2.4.0.1.
The term $O(\mathbb{E}[|X|])$ is slightly optimistic, since we are omitting some polynomial factors. The execution of instructions 8-9 in algorithm 1 requires to:

(i) compute $-\mathbf{c}'\mathbf{B}^\top$;

(ii) check Hamming weights.

With a schoolbook approach the calculation of $-\mathbf{c}'\mathbf{B}^\top$ would require $O((k+\ell)^2(n-k-l))$ operations. However, $\mathbf{c}'$ is expected to have low weight as we are looking for the minimum distance, and some precomputations can be used, drastically reducing this cost. The cost is reduced even further if the use the early abort technique is considered; by checking the codeword weight on-the-run, most of times it allows to stop the computation of $\mathbf{c}''$ as soon as the target weight is exceeded. For these reasons we expect that the cost of instructions $8-10$ is very limited, thus can be safely neglected, so that the cost of instructions $7-10$ corresponds to $O(\mathbb{E}[|X|])$, that is the average number of performed iterations.

We consider it helpful to give, as a reference, the computational cost of applying a *brute-force attack* when searching for the distance of a code, that is, asymptotically, equal to the cost of total enumeration of weight-w codewords:

$$O\left(\binom{n}{w}(q-1)^w\right).$$

We do this to highlight how, even though ISD is intractable for very large instances, it is still a major improvement over brute-force attacks.

2.4.2 Lee-Brickell algorithm

Lee-Brickell algorithm [5], proposed in 1988, is an ISD algorithm where the codeword weight distribution is split into 2 partitions, one having length k and weight p, the other length $n-k$ and weight $w-p$. If we consider $p = w$ the behaviour of the algorithm is the same as Prange's. Lee-Brickell algorithm's standard form for the parity-check matrix $\mathbf{H}$ can be see in fig. 2.2. It is easy to see that the algorithm uses $\ell = 0$. This means that the algorithm does not consider matrix $\mathbf{A}$, since it has 0 rows. The resulting weight partitioning is shown in fig. 2.3. For the latter reasons, Lee-Brickell algorithm's Solve subroutine only consists in the enumeration of the weight-p length-k subcodewords. A list $\mathscr{X}$ of length $\binom{k}{p}(q-1)^p$ is created, and for each entry, the righmost partition is obtained by applying eq. (2.6).

In the original formulation, Lee-Brickell's algorithm was proposed for codes over the binary field $\mathbb{F}_2$, and was later generalized [7] to be used over arbitrary finite fields $\mathbb{F}_q$.

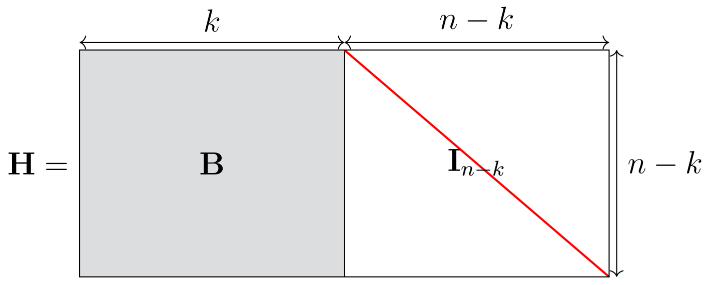


Figure 2.2: $\mathbf{H}$ standard form for Lee-Brickell algorithm

$$\mathbf{B}\mathbf{x}_1^\top = \mathbf{x}_2^\top \tag{2.6}$$

Proposition 2.4.2. ***Performances of Lee-Brickell ISD***

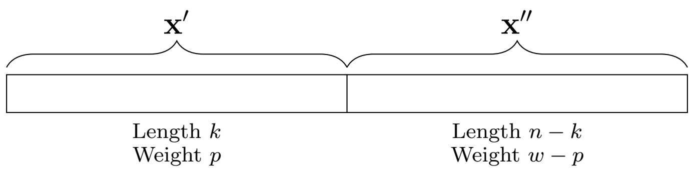


Figure 2.3: Lee-Brickell algorithm codeword partitioning

Algorithm 2: Lee & Brickell Solve subroutine

```
Data: $p \in \mathbb{N}$, $0 \leq k$
Input:
Output: set X with the enumeration of weight-$p$ length-$k$ codewords
1 Set $\mathscr{X} = \{\mathbf{x}' | \mathbf{x}' \in \mathscr{S}_{k,p}\}$
2 return $\mathscr{X}$
```

The time complexity of Lee-Brickell algorithm's Solve *subroutine, with parameter* $p \in \mathbb{N}, 0 \leq p \leq min\{w, k\}$*, is*

$$t_{\mathsf{Solve}}(p) = \binom{k}{p}(q-1)^p$$

The probability that a codeword $\mathbf{c} \in \mathscr{C}_w$ *is returned is*

$$p_{\mathsf{ISD}}(w) = \frac{\binom{k}{p}\binom{n-k}{w-p}}{\binom{n}{w}}$$

Proof. Lee-Brickell's Solve subroutine only consist in the enumeration of codewords with length k and weight p. These result in the enumeration of $\binom{k}{p}(q-1)^p$, hence the stated t_{Solve} is obtained.

The success probability is given by the probability of having assumed the correct weight distribution, that is represented by the number of codewords following the distribution, over the total number of weight-w codewords. This clearly results in the p_{ISD} stated in the proposition.

□

2.4.3 Stern algorithm

Stern's ISD variant [9] was proposed in 1989, and it was originally intended for solving the Given Weight Codeword Problem (GWCP), unlike other variants. It is one of the most used, as well as on the fastest on a classical computer. Stern algorithm is based on performing a collision search using the *meet-in-the-middle* technique on the leftmost partition, by further segmenting it in two partitions. A description of the Solve subroutine is given in algorithm 3. The standard form of the parity-check for Stern algorithm can be seen in fig. 2.4, and it leads to the system of equations in eq. (2.7).

Like Lee-Brickell algorithm, Stern's original formulation considered binary codes ($\mathbb{F}_2$), and was later generalized by Peters [7] to codes over $\mathbb{F}_q$.

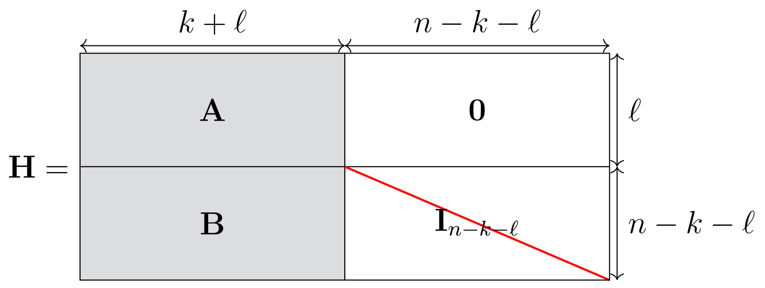


Figure 2.4: **H** standard form for Stern algorithm

$$\begin{cases} \mathbf{A}\mathbf{x}_1^\top = \mathbf{0}, \\ \mathbf{B}_1\mathbf{x}_1^\top = \mathbf{x}_2^\top \end{cases} \tag{2.7}$$

Algorithm 3: Stern Solve subroutine

```
Data: $p \in \mathbb{N}$, $0 \le \left\lfloor \frac{k+\ell}{2} \right\rfloor$
Input: $\mathbf{A} \in \mathbb{F}_q^{\ell \times (k+\ell)}$, $\ell \in \mathbb{N}$
Output: set $X$ with solutions of the small instance, with weight $2p$ equally
        partitioned
1 Write $\mathbf{A} = (\mathbf{A}', \mathbf{A}'')$, where $\mathbf{A}' \in \mathbb{F}_q^{\ell \times \left\lfloor \frac{k+\ell}{2} \right\rfloor}$, $\mathbf{A}'' \in \mathbb{F}_q^{\ell \times \left\lceil \frac{k+\ell}{2} \right\rceil}$;
2 Set $\mathscr{L}_1 = \left\{ (\mathbf{x}', \mathbf{x}'\mathbf{A}'^\top) \mid \mathbf{x}' \in \mathscr{S}_{\left\lfloor \frac{k+\ell}{2} \right\rfloor, p} \right\}$;
3 Set $\mathscr{L}_2 = \left\{ (\mathbf{x}'', -\mathbf{x}''\mathbf{A}''^\top) \mid \mathbf{x}'' \in \mathscr{S}_{\left\lceil \frac{k+\ell}{2} \right\rceil, p} \right\}$;
4 Compute $\mathscr{X}$, the set of all pairs $(\mathbf{x}', \mathbf{x}'') \in \mathscr{S}_p \times \mathscr{S}_p$ such that
    $\mathbf{x}'\mathbf{A}'^\top = -\mathbf{x}''\mathbf{A}''^\top$;
5 return $\mathscr{X}$
```

In order to analyze the performances of Stern ISD, we need to describe how the meet-in-the-middle approach works. The basic idea is finding $\mathbf{x}_1$ trough a collision search on two lists created through enumeration. We partition $\mathbf{x}_1$ in two components $\mathbf{x}_1', \mathbf{x}_1''$, of equal length $\frac{(k+\ell)}{2}$ (assuming $k+\ell$ even for the sake of simplicity) and equal weight p, as shown in fig. 4.2. Then, two lists $\mathscr{L}_1$ and $\mathscr{L}_2$ are created through the enumeration of $\mathscr{S}_p$, and for each entry the partial syndrome is calculated. This is done by splitting matrix $\mathbf{A}$ in two submatrices $\mathbf{A}', \mathbf{A}'' \in \mathbb{F}_q^{\ell \times \frac{k+\ell}{2}}$; elements in $\mathscr{L}_1$ and $\mathscr{L}_2$ are multiplied respectively by $\mathbf{A}'^\top$ and $-\mathbf{A}''^\top$. A valid codeword must satisfy the system in eq. (2.7), hence we must find $(\mathbf{x}_1', \mathbf{x}'')$ pairs such that:

$$\mathbf{x}_1'\mathbf{A}'^\top = -\mathbf{x}_1''\mathbf{A}''^\top \tag{2.8}$$

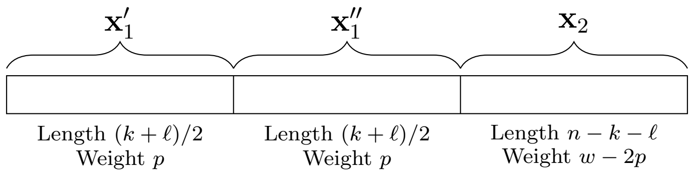


Figure 2.5: Stern algorithm codeword partitioning

We look for such pairs by merging $\mathscr{L}_1$ and $\mathscr{L}_2$. This can be efficiently computed using a sorting algorithm and applying a binary search approach, taking time

$$O\left(\max\left\{|\mathscr{L}_1| \cdot \log_2(|\mathscr{L}_1|), |\mathscr{L}_2| \cdot \log_2(|\mathscr{L}_2|)\right\}\right).$$

If we neglect the usage of floors and ceilings, by assuming $k+\ell$ even, lists sizes are given by:

$$L = |\mathscr{L}_1| = |\mathscr{L}_2| = \binom{\frac{k+\ell}{2}}{p}(q-1)^p. \tag{2.9}$$

Getting rid of the logarithmic factor in our asymptotic estimation, and taking consideration that the resulting list $\mathscr{X}$ needs to be somehow allocated, we can consider the overall cost of the merging the two lists to be

$$O(\max\{|\mathscr{L}_1|, |\mathscr{L}_2|, |\mathscr{X}|\}) = O(\max\{|L|, |\mathscr{X}|\}).$$

When the two merged lists are formed by elements without any relevant structure, we can safely consider that each pair of elements results in a collision with a probability equal to the alphabet length to the power of the words length. In our case, each pair of elements in $\mathscr{L}_1$ and $\mathscr{L}_2$ result in a collision with probability $q^{-\ell}$. This is a frequently employed heuristic, that corresponds to the assumption that each entry of the associated syndromes is uniformly distributed over $\mathbb{F}_q$. When this is true, we can set

$$|\mathscr{X}| = |\mathscr{L}_1| \cdot |\mathscr{L}_2| q^{-\ell} = L^2 q^{-\ell}. \tag{2.10}$$

At this point, the rightmost partition of the codeword is achieved by applying the second equation in eq. (2.7).

The resulting codeword partitioning is depicted in fig. 2.5.

Proposition 2.4.3. ***Performances of Stern ISD***
The time complexity of Stern algorithm's Solve *subroutine, with parameters* $p, \ell \in \mathbb{N}$, *where* $0 \leq p \leq \lfloor \frac{k+\ell}{2} \rfloor$, $0 \leq \ell \leq n-k$, *is*

$$t_{\mathsf{Solve}}(p, \ell) = 2L + \frac{L^2}{q^\ell},$$

where $L = \binom{\frac{k+\ell}{2}}{p}(q-1)^p$. *The probability that a codeword* $\mathbf{c} \in \mathscr{C}$ *is returned is*

$$p_{\mathsf{ISD}} = \frac{\binom{(k+\ell)/2}{p}^2 \binom{n-k-\ell}{w-2p}}{\binom{n}{w}}.$$

Proof. The cost t_{Solve} of the Solve subroutine is given by the cost of lists enumeration, and the application of the meet-in-the-middle. By combining eq. (2.9) and eq. (2.10) we get the value of t_{Solve} specified in the proposition.

The success probability is given by the probability of having assumed the correct weight distribution, that is represented by the number of codewords following the distribution, over the total number of weight-w codewords. This clearly results in the p_{ISD} stated in the proposition.

$\square$

3 Generalized LDPC Ensemble

To analyze our algorithm we need to define the set of codes we will be working on. To do so, we need to introduce a specific ensemble of codes, characterized by three parameters:

- code length n;
- code rate R;
- density ν of the parity-check matrix $\mathbf{H}$.

In our ensemble we will only consider binary codes, hoping to generalize this work in the future, expanding the ensemble characterization to codes over arbitrary finite fields $\mathbb{F}_q$.

But first, let us describe what a random code is.

3.1 Random Codes

We denote by $\mathcal{U}_{n,R}$ the uniform distribution of linear codes with length n and dimension $k = Rn$, over $\mathbb{F}_2$. It is well known that sampling a uniform random, $Rn \times n$ matrix over $\mathbb{F}_2$ has full rank Rn with overwhelming probability. For random codes the average weight distribution can be easily estimated.

Theorem 3.1.1.
Let $\mathscr{C} \sim \mathcal{U}_{n,R}$ and $m_w^{\text{rnd}} = \mathbb{E}\left[m_{\mathscr{C}}(w)\right]$; then

$$m_w^{\text{rnd}} = \binom{n}{w} 2^{-n(1-R)}$$

Let R be a constant and define $\mu_w^{\text{rnd}} = \lim_{n\to\infty} \frac{1}{n} \cdot \log_2(m_w)$; then:

- *if $w = o(n)$, then*

$$\mu_w^{\mathrm{rnd}} = \frac{w}{n}\log_2\left(\frac{n}{w}\right)$$

- *if $w = \Omega(n)$, then*

$$\mu_w^{\mathrm{rnd}} = h(\frac{w}{n}) - (1-R)$$

For random codes a well known result esablishes their minimum distance.

Theorem 3.1.2. ***Gilbert-Varshamov Bound (GV) bound***
Let $\mathbf{H} \xleftarrow{\$} \mathbb{F}_2^{n(1-R)\times n}$ and $\mathscr{C}$ be the linear code whose parity-check matrix is $\mathbf{H}$. Then, with overwhelming probability, $\mathscr{C}$ has minimum distance $d = \delta^{\mathrm{rnd}} n$, where $\delta \in [0;1]$ is called relative minimum distance and $\delta^{\mathrm{rnd}} = h^{-1}(1-R)$.

This result is obtained by setting d as the minimum value of w such that $\mu_w \geq 0$.

3.2 Our ensemble

Definition 3.2.1. Let $R \in [0;1]$, $n \in \mathbb{N}$ and $\nu \in R, 0 < \nu < \frac{1}{2}$. Then, we define $\mathcal{B}_{n,R,\nu}$ as the distribution that return matrices with size $(1-R)n \times n$, and such that each entry in distributed according to the Binomial distribution with parameter ν. We write $\mathbf{H} \leftarrow \mathcal{B}_{n,R,\nu}$ when $\mathbf{H}$ is sampled according to $\mathcal{B}_{n,R,\nu}$, and $\mathbb{P}[\mathbf{H}]$ to indicate the probability that a specific matrix $\mathbf{H}$ is returned as output.

We aim to treat $\mathcal{B}_{n,R,\nu}$ as a distribution for parity-check matrices. Thus, the right kernel of each $\mathbf{H}$ can be deemed as a linear code with length n and dimension $\geq Rn$. The average weight distribution can be easily recovered.

For this analysis we will make use of the piling-up lemma:

Theorem 3.2.1. ***Piling-up lemma***
Let $x1, \ldots, x_t$ be t random variables taking values in $\mathbb{F}_2$. Then

$$\mathbb{P}[x_1 + \cdots + x_t = 0] = \frac{1}{2} + 2^{t-1}\prod_{i=1}^{t}\left(\mathbb{P}[x_i = 0] - \frac{1}{2}\right).$$

Theorem 3.2.2.
Let $\mathbf{H} \leftarrow \mathcal{B}_{n,R,\nu}$, $\mathscr{C}$ be the code with parity-check matrix $\mathbf{H}$ and $m_w = \mathbb{E}[m_{\mathscr{C}}(w)]$;

then

$$m_w = \binom{n}{w} \cdot 2^{-n(1-R)} \cdot (1 + (1-2\nu)^w)^{-n(1-R)}.$$

Proof. Let $r = (1-R)n$, S_w be the Hamming sphere with radius w and $\mathbb{P}[\mathbf{H}]$ denote the probability that $\mathbf{H}$ is sampled from the distribution $\mathcal{B}_{n,R,\nu}$. The average weight distribution is derived as

$$m_w = \sum_{\mathbf{H} \in \mathbb{F}_2^{r \times n}} |\{\mathbf{x} \in S_w \cap \ker(\mathbf{H})\}| \cdot \mathbb{P}[\mathbf{H}] = \sum_{\mathbf{H} \in \mathbb{F}_2^{r \times n}} \sum_{\mathbf{x} \in S_w} f(\mathbf{x}, \mathbf{H}) \cdot \mathbb{P}(\mathbf{H}),$$

where $f(\mathbf{x}, \mathbf{H}) = 1$ if $\mathbf{x} \in \ker(\mathbf{H})$ and 0 otherwise. We can swap the two sums in the above equation and get

$$m_w = \sum_{\mathbf{x} \in S_w} \sum_{\mathbf{H} \in \mathbb{F}_2^{r \times n}} f(\mathbf{x}, \mathbf{H}) \cdot \mathbb{P}(\mathbf{H}).$$

We now show that $f(\mathbf{x}, \mathbf{H})$ depends only on the weight of $\mathbf{x}$. First, we indicate by $\mathcal{B}_{n,\nu}$ the Bernoulli distribution with parameter ν. Then $\mathbf{x} \in \ker(\mathbf{H})$ if and only if, for every row $\mathbf{h}_i$ of $\mathbf{H}$, it holds that $\mathbf{h}_i \mathbf{x}^\top = 0$. Sampling $\mathbf{H}$ from $\mathcal{B}_{n,R,\nu}$ corresponds to getting r samples $\mathbf{h}_1, \ldots, \mathbf{h}_r$ from $\mathcal{B}_{n,R,\nu}$, thus, we have $\mathbf{H}x^\top = 0$ if and only if $\mathbf{h}_i \mathbf{x}^\top = 0$ for every i. For each $\mathbf{x}$, the overall number of valid $\mathbf{h}_i$ is

$$2^{n-w} \cdot \sum_{\substack{0 \le j \le w \\ j \text{ even}}} \binom{w}{j}.$$

Indeed, the above quantity counts the number of vectors $\mathbf{h}_i$ which overlap with $\mathbf{x}$ in an even number of positions. The probability to get such a vector is

$$m_w = \binom{n}{w} \cdot \left(\sum\right) \nu^j (1-\nu)^{w-j}.$$

Considering that there are r rows, and that the probability depends only on the weight of $\mathbf{x}$, we get

$$m_w = \binom{n}{w} \cdot \left(\sum_{\substack{0 \le j \le w \\ j \text{ even}}} \binom{w}{j} \nu^j (1-\nu)^{w-j} \right)^r.$$

To conclude the proof, we simplify the expression by considering that $\sum_{\substack{0 \le j \le w \\ j \text{ even}}} \binom{w}{j} \nu^j (1-$

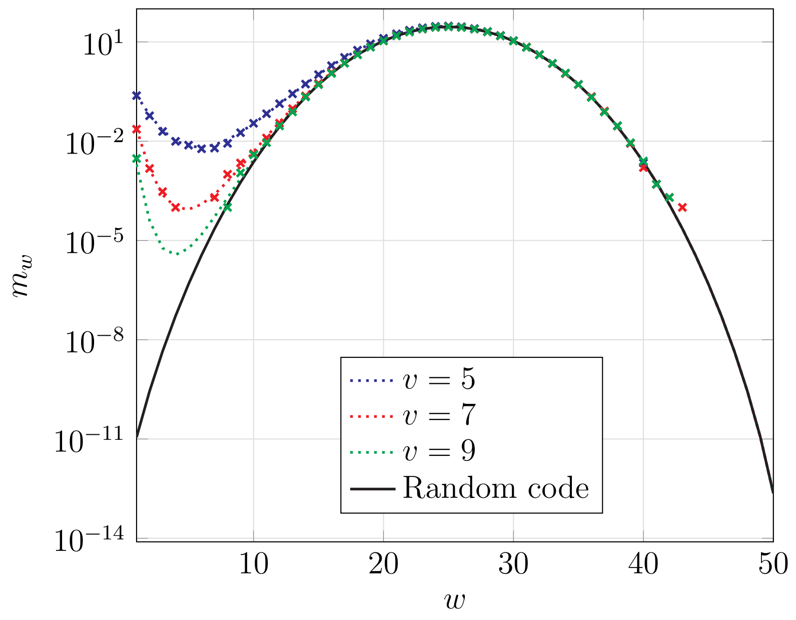


Figure 3.1: Average weight distribution for codes with $n = 50$, $r = 42$ and several values of v. Dotted lines report the theoretical values while cross marks correspond to empirical values, which have been obtained by averaging over 10 000 matrices. The black line shows the average weight distribution for a random code with the same values of n and r. v is the average column weight.

$\nu)^{w-j}$ corresponds to the probability that the sum of w independent Bernoulli variables with parameters ν is equal to 0. This is a special case of the piling-up lemma (theorem 3.2.1) in which all distributions are the same. Hence we rewrite this probability as

$$\frac{1}{2} + 2^{w-1} \prod_{i=1}^{w} (\mathbb{P}[x_i = 0] - \frac{1}{2}) = \frac{1}{2} + 2^{w-1} \prod_{i=1}^{w} (\frac{1}{2} - \nu) = \frac{1}{2}(1 + (1 - 2\nu)^w).$$

$\square$

See fig. 3.1 for a comparison between the formula and the well known distribution for random codes. Moreover, we have compared the theoretical average distribution with the emiprical one, estimated by sampling a large number o $\mathcal{B}_{n,R,\nu}$, computing the weight enumerator function by exhaustive search over all codewords and then averaging over all attempts.

Evidently the sparsity γ of parity-check in this ensemble is equal to ν.

3.3 Bias with respect to random codes

We now study the bias in the weight distribution of our ensemble relative to that of random codes. We performed numerical experiments on codes with constant rate $R = 0.4$, and using several values of ν and n. We have analyzed how $\frac{1}{n}\log_(m_w)$ grows, which represent the *normalized weight distribution.* The results of these experiments are represented in fig. 3.2. We can see how, apart from $\nu = \frac{\ln(n)}{n}$, for all densities the weight distribution of the ensemble converges to that of random codes with sufficiently large length n. Instead, for $\nu = \frac{\ln(n)}{n}$, the behavior of the weight distribution differs significantly from that of random codes.

Trying to formalize the achieved results, we were able to prove the following four facts:

1. the coefficient m_w is always of the form $m_w = m_w^{\text{rnd}} \cdot g(n,w)$, where $g(n,w) = (1+(1-2\nu)^w)^{n(1-R)}$. In particular, $g(n,w) \geq 1$ for any n and w;
2. when ν is constant, the coefficients $g(n,w)$ has an impact on the weight distribution. In particular, the minimum distance is in $o(n)$;
3. for any $\nu = O\left(\frac{\ln(n)}{n}\right)$, the coefficient $g(n,w)$ has an impact on the weight distribution. In particular, the minimum distance is $o(n)$;
4. for any ν such that $\frac{\ln(n)}{n} = o(\nu)$, there is no significant difference with respect to random codes.

We rewrite the coefficients in the weight distribution in a way which makes the difference cleared.

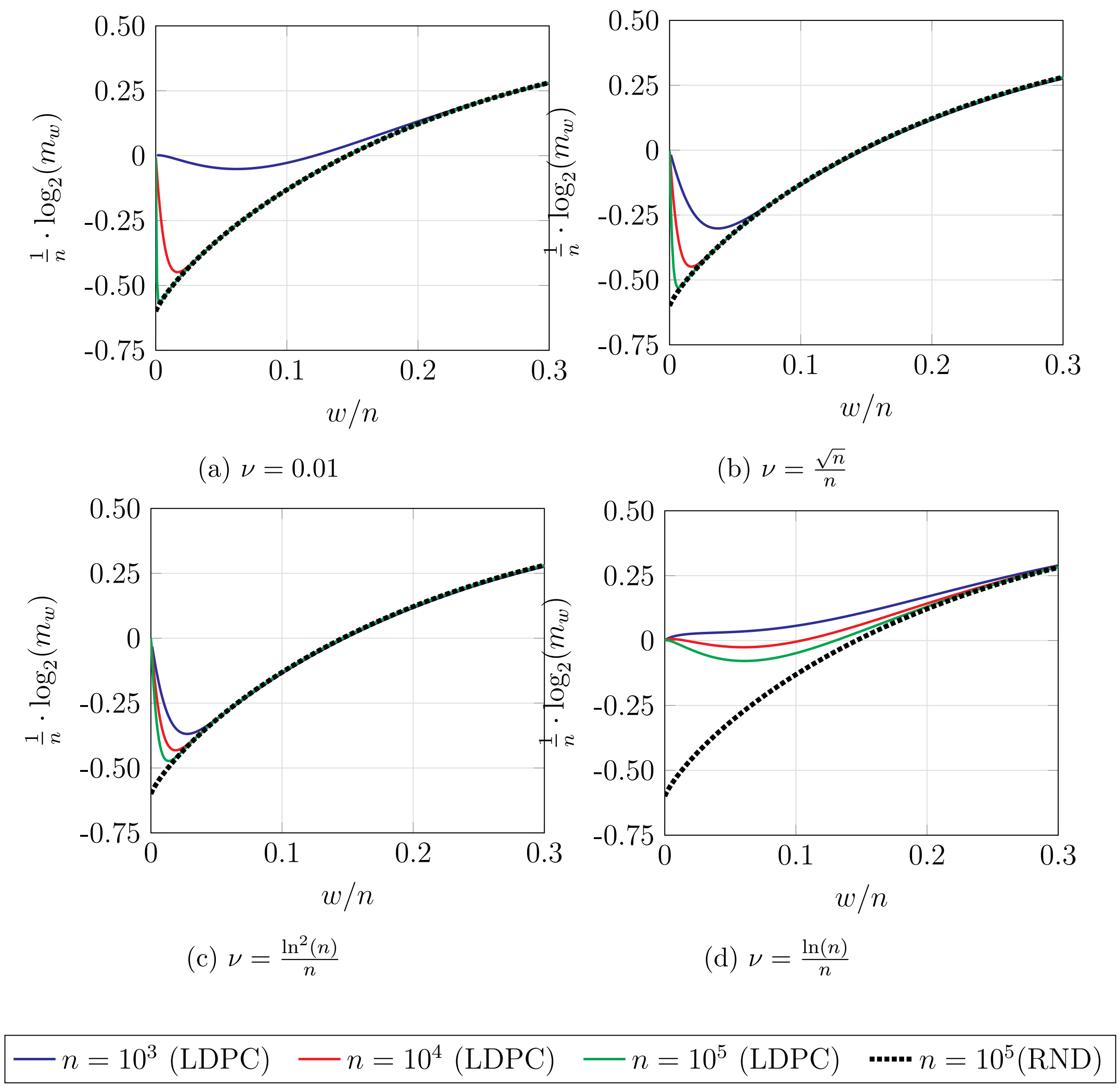
Theorem 3.3.1.
For codes sampled from $\mathcal{B}_{n,R,\nu}$, we have

$$m_w = m_w^{\text{rnd}} \cdot (1+(1-2\nu)^w)^{(n(1-R))},$$

$$\mu_w = \mu_w^{\text{rnd}} + (1-R)\cdot \log_2(1+(1-2\nu)^w).$$

We observe that the only difference is due to the term

$$g(n,w) = n\cdot(1-R)\cdot\log_2(1+(1-2\nu)^w),$$



(a) $\nu = 0.01$

(b) $\nu = \frac{\sqrt{n}}{n}$

(c) $\nu = \frac{\ln^2(n)}{n}$

(d) $\nu = \frac{\ln(n)}{n}$

Figure 3.2: Value of $\frac{1}{n} \cdot \log_2(m_w)$, for several values of n and ν, and $R = 0.4$.

which we now study in several meaningful regimes. First, we notice that if $\nu = \frac{1}{2}$, then $m_w = m_w^{\text{rnd}}$ and $\mu_w = \mu_w^{\text{rnd}}$.

We start by analyzing the behavior with ν constant.

Theorem 3.3.2.
Let $\nu \leq \frac{1}{2}$ be a constant and d denote the minimum value of w such that $\mu_w \geq 0$. Then, for increasing n, we have $d = \Omega(n)$; in particular, $d = \delta \cdot n$ and $\delta \sim \delta^{\text{rnd}}$.

Proof. As n grows, d becomes a function of n. We first show that d cannot $o(n)$.

Indeed, in this regime we can use the binomial asymptotic expansion

$$\frac{1}{n} \cdot \log_2 \binom{n}{d} = \frac{d}{n} \log_2 \left(\frac{n}{d}\right) \cdot (1 + o(1)).$$

Neglecting low order terms, asymptotically μ_d is

$$\frac{d}{n} \cdot \log_2 \left(\frac{n}{d}\right) - (1 - R) \cdot (1 - \log_2(1 + (1 - 2\nu)^d)).$$

If $d = o(n)$, the term $\frac{d}{n} \cdot \log_2(\frac{n}{d})$ asymptotically vanishes, while $1 - \log_2(1 + (1 - 2\nu)^d)$ is always positive. Indeed d is constant, then $(1 - 2\nu)^d$ is constant as well, and less than 1, so that

$$1 - \log(1 + (1 - 2\nu)^d) > 1 - \log_2(1 + 1) = 0$$

If instead d grows with n, we get that $(1 - 2\nu)^d$ tends to 0, hence $\log_2(1 + (1 - 2\nu)^d)$ tends to 0 as well. In both cases, the term $1 - \log(1 + (1 - 2\nu)^d)$ tends to a positive constant. This implies that for sufficiently large n, the value of μ_w is negative.

The situation changes when one considers $d = \Omega(n)$. We still have $(1 - 2\nu)^d = o(1)$, hence $\log_2(1 + (1 - 2\nu)^d) = o(1)$, but now

$$\binom{n}{d} = 2^{n \cdot h(\delta) \cdot (1 + o(1))}.$$

Then, apart from low order terms, μ_d is $h(R) - (1 - R) = \mu_d^{\text{rnd}}$.

□

We now consider what happens when ν grows with n.

Theorem 3.3.3.
Let $\nu = f(n)/n$ with $f(n) > 0$ for every $n > 0$, $\lim_{n\to\infty} f(n) = \infty$ and $f(n) = o(n)$, i.e., $\lim_{n\to\infty} \frac{1}{n} \cdot f(n) = 0$. *Then, for sufficiently large n:*

- *for any w such that $w \cdot f(n) = o(n)$, μ_w is non negative only if $w \leq n \cdot e^{-(1-R) \cdot f(n)}$;*
- *for any w such that $w \cdot f(n) = \Omega(n)$, μ_w is always negative*
- *for any $w = \Omega(n), \mu_w \sim \mu_w^{\text{rnd}}$*

Proof. Since $\lim_{n\to\infty}\frac{n}{f(n)}=0$, we get

$$(1-2\nu)^w=\left(1-\frac{2f(n)}{n}\right)^w=\left(\left(1-\frac{2f(n)}{n}\right)^{\frac{n}{f(n)}}\right)^{\frac{w\cdot f(n)}{n}}\sim e^{\frac{-2w\cdot f(n)}{n}}.$$

Therefore, the expression for μ_w becomes

$$\begin{aligned}\mu_w&=\frac{1}{n}\cdot\log_2\binom{n}{2}-(1-R)(1-\log_2(1+(1-2\nu)^w))\\&\sim\frac{1}{n}\cdot\log_2\binom{n}{w}\end{aligned}\tag{3.1}$$

We now distinguish between the three cases, as stated in the theorem.

- Case $w\cdot f(n)=o(n)$: we have $\lim_{n\to\infty}e^{-\frac{2w\cdot f(n)}{n}}=e^0=1$. We use the following expansion: for $x=o(1)$, it holds that

  $$1-\log_2(1+e^{-x})=\frac{1}{2\ln(2)}\cdot x+o(x)$$

  In our case $x:=\frac{w\cdot f(n)}{n}$ hence

  $$1-\log_2\left(1+e^{\frac{-2w\cdot f(n)}{n}}\right)=\frac{w\cdot f(n)}{n\cdot\ln(2)}+o\left(\frac{2w\cdot f(n)}{n}\right)=\frac{w\cdot f(n)}{n\cdot\ln(2)}+o(1).$$

  Since $\frac{w\cdot f(n)}{n}=o(1)$ and $\lim_{n\to\infty}f(n)=\infty$, w grows slower than $\frac{n}{f(n)}$ so

  $$0\le\lim_{n\to\infty}\frac{w}{n}\le\lim_{n\to\infty}\frac{n}{n\cdot f(n)}=\lim_{n\to\infty}\frac{1}{f(n)}=0.$$

  Hence $w=o(n)$ and

  $$\frac{1}{n}\cdot\log_2\binom{n}{2}=\frac{w}{n}\cdot\log_2\left(\frac{n}{w}\right)=o(1),$$

  thus
  $$\mu_w=\frac{w}{n}\cdot\left(\log_2\left(\frac{n}{2}\right)-\frac{(1-R)\cdot f(n)}{\ln(2)}\right)+o(1).$$

  This is non negative whenever $\log_2\left(\frac{n}{2}\right)-\frac{(1-R)\cdot f(n)}{\ln(2)}\ge0$, which after some manipulations results into

$$w \leq n \cdot e^{-(1-R)\cdot f(n)}.$$

- Case $w \cdot f(n) = \Omega(n)$: now, $w = \Omega\left(\frac{n}{f(n)}\right)$. Since $\lim_{n\to\infty} f(n) = \infty$, we have $\frac{w}{n} = \Omega\left(\frac{1}{f(n)}\right) = o(1)$. Thus also inthis case, $w = o(n)$ and $\log_2\binom{n}{w} = w \cdot \log_2\left(\frac{n}{w}\right) + o(1)$. Let $\lim_{n\to\infty} \frac{w\cdot f(n)}{n} = \zeta$, with $\zeta > 0$ being a constant. Then,

$$\beta = \lim_{n\to\infty} 1 - \log_2\left(1 + e^{\frac{-2w\cdot f(n)}{n}}\right) = 1 - \log_2\left(1 + e^{-2\zeta}\right) > 1 - \log_2(2) = 0.$$

  We are then left with

$$\begin{aligned}\mu_w &= \lim_{n\to\infty} \frac{w}{n} \cdot \log_2\left(\frac{n}{w}\right) - \beta(1-R)\\ &= \lim_{n\to\infty} \underbrace{\frac{\zeta}{f(n)} \cdot \log_2\left(\frac{f(n)}{\zeta}\right)}_{\to 0} - \beta(1-R)\\ &= -\beta(1-R) < 0\end{aligned}$$

- Case $w \cdot = \Omega(n)$: we now have $\lim_{n\to\infty} \frac{w\cdot f(n)}{n} = \infty$. We start again from eq. (3.1) and consider that

$$\lim_{n\to\infty} \log_2\left(1 + \underbrace{e^{\frac{-2w\cdot f(n)}{n}}}_{e^{-\infty}\to 0}\right) = \log_2(1) = 0.$$

  In other words, in this regime, $\lim_{n\to\infty} g(n, w) = 1$, hence $\mu_w \sim \mu_w^{\text{rnd}}$.

$\square$

As a special case of the above theorem, we have a significant deviation in the weight distribution of codes drawn from the ensemble only if $\nu = O\left(\frac{\log(n)}{n}\right)$.

Theorem 3.3.4.
For $w \cdot f(n) = o(n)$, $\mu_w \geq 0$ only if $f(n) \leq \frac{\ln(n)}{1-R}$. For $f(n) = \alpha \cdot \ln(n), \mu_w \geq 0$ for any $w \leq n^{1-\alpha(1-R)}$.

Proof. From theorem 3.3.3 we have that

$$\mu_w \geq 0 \implies w \leq n \cdot e^{-(1-R)\cdot f(n)}.$$

We are considering positive weights $w \geq 1$, hence we want

$$\begin{aligned} 1 &\leq n \cdot e^{-(1-R)\cdot f(n)} = e^{\ln(n)-(1-r)\cdot f(n)} \\ \implies 0 &\leq \ln(n) - (1-R) \cdot f(n) \\ \implies f(n) &\leq \frac{\ln(n)}{(1-R)} \end{aligned}$$

Setting $f(n) = \alpha \cdot \ln(n)$, and considering the largest w such that $w \leq n \cdot e^{-\alpha(1-R)\cdot\ln(n)}$, we get the thesis.

□

3.4 Degenerate codes

We now show that, if the density is too low, sampling from our distribution leads to parity-check matrices that are badly formed, e.g., have a large number of null columns. We call these codes *degenerate*. This is coherent with the results of theorem 3.3.3: we show that, whenever the density is $O\left(\frac{\ln(n)}{n}\right)$, with high probability the code contains a large number of null columns. This somehow motivates the fact that, for such low densities, the minimum distance becomes subexponential in n.

Theorem 3.4.1.
Let $\nu = O\left(\frac{\ln(n)}{n}\right)$. Then, a parity-check matrix sampled from $\mathcal{B}_{n,R,\nu}$ has an average number of null column which is at least 1.

Proof. The probability that a column is null is $(1-\nu)^{(1-R)}$. By hypothesis $\nu = o(1)$, hence

$$(1-\nu)^{n(1-R)} = \left((1-\nu)^{\frac{1}{v}}\right)^{\nu\cdot n(1-R)} \sim e^{-\nu\cdot n(1-R)}$$

Hence, the average number of null columns is $\sim n \cdot e^{-\nu\cdot n(1-R)}$. Requiring it to be at least 1 we get

$$\begin{aligned} 1 &\leq n \cdot e^{-\nu\cdot n(1-R)} = e^{\ln(n)-\nu\cdot n(1-r)} \\ \implies \nu &\leq \frac{\ln(n)}{n(1-R)} \leq \frac{\ln(n)}{n} \end{aligned}$$

□

Other properties of this degenerate codes can be proven, such as the the probability that there are equal columns. We do not enter into such detail, and from now on we exclude degenerate cases from our analysis and consider only densities that grow faster then $\frac{\ln}{n}$, in other words, is ν is such that:

$$\frac{\ln(n)}{n} = o(\nu).$$

From theorem 3.3.3, it follows that the ensembles of codes we are considering have minimum distances properties very close to those of random codes. This is a property we are interested in, as in code-based cryptography, schemes that are more susceptible to attacks are those using codes with tight inherent structure, as this might be exploited. The randomness of our ensembles is, therefore, a desired property. Furthermore, the knowledge of the weight distributions help us to get estimates on the performances of ISD algorithms.

4 Algorithm

In this chapter we present our variant of ISD, based on Stern's algorithm, but tailored to exploit the sparsity of low-density parity-check matrices. Consequently, for the remainder of this dissertation we will refer to this algorithm as *SparseISD*.

We analyze the performances of the algorithm when used with codes from the ensembles we introduced in the previous chapter. For this reason, we characterize the algorithm with the assumption that $q = 2$.

4.1 Rationale

In Stern's algorithm the small instance is produced in form of a matrix with small support. Put differently, this means the matrix can be used as generator of a subcode of the dual code $\mathscr{C}^\top$, with support size less than n. This is obtained via PGE. Applying gaussian elimination on a $n - k \times n$ matrix, any set of ℓ rows yields a matrix with at least $n - k - l$ null columns. In other words, the selection of ℓ rows gives a matrix with support size $\leq k + \ell$. This can be less than $k + \ell$ only if some other columns are null: each column is null with probability $2^{-\ell}$, and the average number of extra null columns is $(k + \ell)2^{-\ell}$. Stern's algorithm is normally optimized by setting $l = \Omega(n)$, thus the average number of extra null columns decays exponentially with n.

Remember that the parameter ℓ in Stern's algorithm impacts several aspects of the procedure:

- The success probability gets smaller as ℓ increases. It can be increased by enlarging p, but it increases the lists sizes, as well;
- The lists sizes increase with ℓ, as the length of the small instance grows with ℓ;

- The probability that a collision happens for a given pair of list elements decays exponentially with ℓ.

For sparse parity-check matrix $\mathbf{H}$, the situation is rather different as one can find small instances with more convenient parameters. We can see this by considering, again, matrices sampled from $\mathcal{B}_{n,R,\nu}$. By picking a random ℓ, on average the number of non null column is

$$\bar{z} = \mathbb{E}[z] = n(1 - (1 - \nu)^{\ell})$$

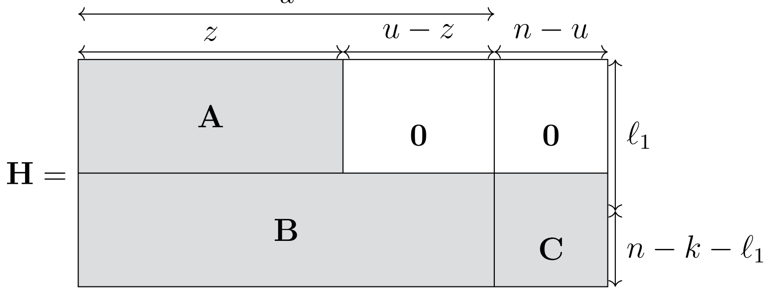
$(1 - v)^{\ell}$ is the probability that a given column of the submatrix created by picking ℓ rows is null. Therefore, $1 - (1 - \nu)^{\ell}$ corresponds to the probability that at least one coordinate is not null. Multiplying by n yields the average number of non null columns. It is easy to see that z can stay small even if ℓ is large. For insance, let us consider the case of $\nu = o(n)$; then

$$(1 - \nu)^{\ell} = \left((1 - \nu)^{\frac{1}{\nu}})^{\ell\nu}\right) \sim e^{-\ell\nu}.$$

The way ℓ is chosen affects the parametrs of the small instance. We split this parameter in two parameters ℓ_1 and ℓ_2, such that $\ell = \ell_1 + \ell_2$. They have a different effect on the size of the small instances of the algorithm, that will be later described.

This matter is studied in detail when analyzing the performances of the algorithm we are presenting. First, we prove a result that will be instrumental in the design of the algorithm.

Lemma 4.1.0.1.
Let $H \in \mathbb{F}_2^{(n-k)\times n}$ be a matrix in the form



Then, $\mathbf{C}$ has rank $\leq$ min $\{n - u, n - k - \ell\}$ and, in particular $\mathbf{C}$ is square $\iff$ $u = k + \ell$

Proof. The proof is obvious and just depends on the dimensions of the matrix **C**.

□

4.2 Description

We obtain a matrix as the one in 4.1 using the following procedure:

1. select at random ℓ_1 rows of $\mathbf{H}$ and denote their supports by $J_1, ..., J_{\ell_1}$
2. set $J = \bigcup_{i=1}^{\ell_1} J_i$ and select a permutation π that moves J to the leftmost positions. Let $z = |J|$;
3. apply a row permutation that brings the select rows in the first ℓ_1 position, then apply a column permutation that places $n - z$ null columns on the right.

Let $\mathbf{H}'$ denote the matrix obtained in this way. All the operation we have just described can be applied using a row permutation matrix $\mathbf{P}_r$ and a column permutation matrix $\mathbf{P}_c$, so that $\mathbf{H}' = \mathbf{P}_r \cdot \mathbf{H} \cdot \mathbf{P}_c$. Now, aiming to produce an identity matrix in the bottomright coner, we apply a change of basis $\mathbf{C}$. Since the first ℓ_1 rows have only null coordinates on the rightmost $n - u$ coordinates, such a matrix can be of the form

$$\mathbf{S} = \begin{pmatrix} \mathbf{I}_{\ell_1} & \mathbf{0} \\ \mathbf{0} & \mathbf{S}' \end{pmatrix}$$

,

with $\mathbf{S}' \in \mathbb{F}_q^{(n-k-\ell_1)\times(n-k-\ell 1)}$. Applying $\mathbf{S}$ on $\mathbf{H}'$, we obtain $\mathbf{H}''$, that is the standard form of the parity-check matrix for our algorithm. We can see how here comes into play the parameter ℓ_2, that we can tune to resize the bottomright identity matrix, and the other submatrices accordingly. In fig. 4.1 it is possible to see a visualization of the transformations operated on the parity check matrix by the algorithm.

Once we achieved the standard form, we search for a solution $\mathbf{x}$ in the form:

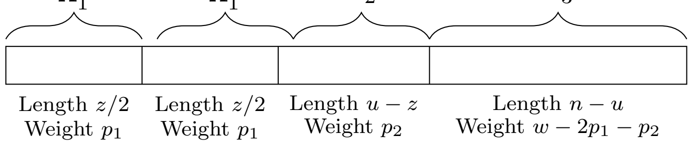
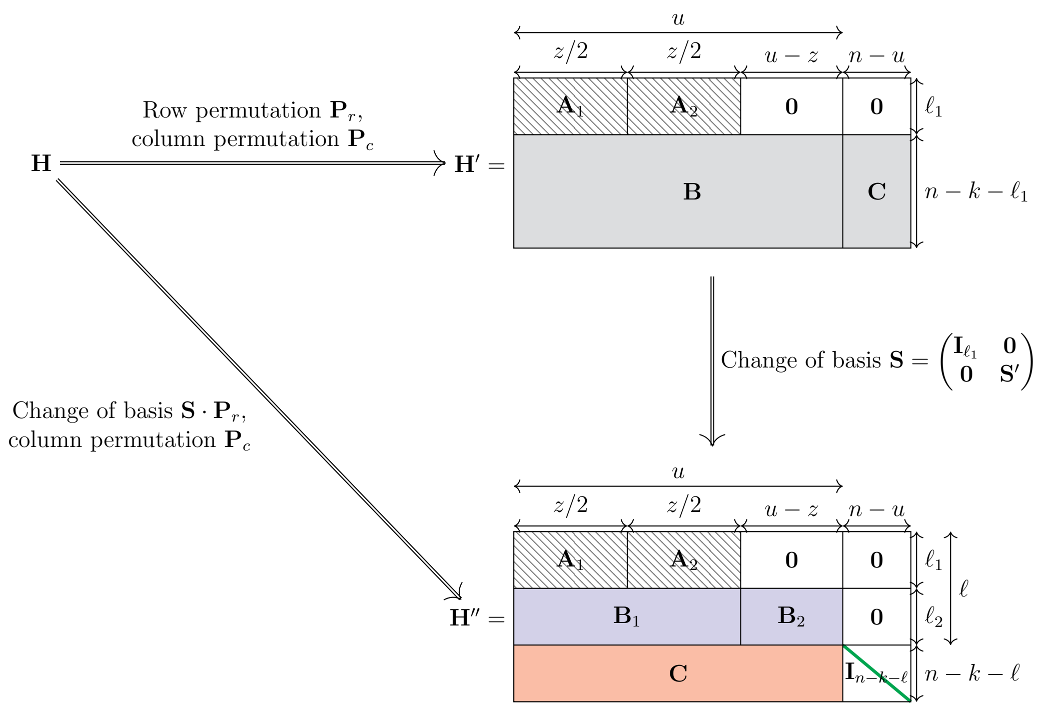

$\mathbf{x}_1'$
$\mathbf{x}_1''$
$\mathbf{x}_2$
$\mathbf{x}_3$
Length $z/2$ Weight p_1
Length $z/2$ Weight p_1
Length $u-z$ Weight p_2
Length $n-u$ Weight $w-2p_1-p_2$

Figure 4.2: SparseISD codeword weight partitioning



Row permutation $\mathbf{P}_r$, column permutation $\mathbf{P}_c$
$\mathbf{H}$
$\mathbf{H}' =$
u
$z/2$
$z/2$
$u-z$
$n-u$
$\mathbf{A}_1$
$\mathbf{A}_2$
$\mathbf{0}$
$\mathbf{0}$
ℓ_1
$\mathbf{B}$
$\mathbf{C}$
$n-k-\ell_1$
Change of basis $\mathbf{S} = \begin{pmatrix} \mathbf{I}_{\ell_1} & \mathbf{0} \\ \mathbf{0} & \mathbf{S}' \end{pmatrix}$
Change of basis $\mathbf{S} \cdot \mathbf{P}_r$, column permutation $\mathbf{P}_c$
$\mathbf{H}'' =$
$\mathbf{B}_1$
$\mathbf{B}_2$
ℓ_2
ℓ
$\mathbf{C}$
$\mathbf{I}_{n-k-\ell}$
$n-k-\ell$

Figure 4.1: Matrix transformations in the algorithm

which gives rise to the following system of equations:

$$\begin{cases} \mathbf{A}_1 \mathbf{x}_1'^{\top} = \mathbf{A}_2 \mathbf{x}_1''^{\top}, \\ \mathbf{B}_1 \mathbf{x}_2^{\top} = \mathbf{B}_2 (\mathbf{x}_1', \mathbf{x}_1'')^{\top}, \\ \mathbf{C}(\mathbf{x}_2, \mathbf{x}_1', \mathbf{x}_1'')^{\top} = \mathbf{x}_3. \end{cases} \tag{4.1}$$

Like Stern algorithm, SparseISD is based on performing meet-in-the-middle to solve the small instances. We solve the system one solution after the other, that is:

1. we find all solutions $(\mathbf{x}_1', x_1'')$ to the first equation, performing meet-in-the-middle; we call the resulting list by $\mathscr{L}_1$;

2. we find $\mathbf{x}_2$ by solving the second equation, performing meet-in-the-middle again; we call the resulting list by $\mathscr{L}_2$;

3. for each entry $(\mathbf{x}_1', \mathbf{x}_1'', \mathbf{x}_2) \in \mathscr{L}_2$, we compute the corresponding $\mathbf{x}_3$ by solving the third equation, and check its weight.

In algorithm 4 we give a formal description of the Solve subroutine of our algorithm, performed after the transformation on the parity-check matrix are applied. Therefore, a parity-check matrix in the form depicted in fig. 4.3 is considered as input for the subroutine.

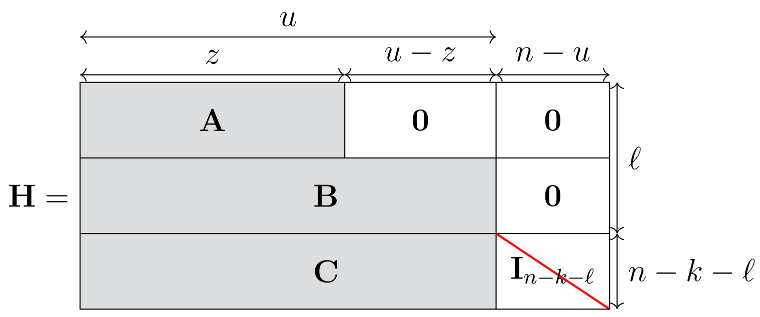


Figure 4.3: **H** standard form for SparseISD algorithm

Algorithm 4: SparseISD Solve subroutine

```
Data: $p \in \mathbb{N}$, $0 \leq \left\lfloor \frac{k+\ell}{2} \right\rfloor$
Input: $\mathbf{A} \in \mathbb{F}_q^{\ell_1 \times z}$, $\ell 1 \in \mathbb{N}$, $\mathbf{B} \in \mathbb{F}_q^{\ell_2 \times u}$, $\ell_2 \in \mathbb{N}$, $u \in \mathbb{N}$
Output: set $X$ with solutions of the small instance, with weight $2p_1 + p_2$

1 Write $\mathbf{A} = (\mathbf{A}_1, \mathbf{A}_2)$, where $\mathbf{A}_1 \in \mathbb{F}_q^{\ell_1 \times \lfloor \frac{z}{2} \rfloor}$, $\mathbf{A}_2 \in \mathbb{F}_q^{\ell \times \lceil \frac{z}{2} \rceil}$;
2 Set $\mathscr{L}_1' = \left\{ (\mathbf{x}_1', \mathbf{x}_1' \mathbf{A}_1^\top) \mid \mathbf{x}_1' \in \mathscr{S}_{\lfloor \frac{z}{2} \rfloor, p} \right\}$;
3 Set $\mathscr{L}_1'' = \left\{ (\mathbf{x}_1'', \mathbf{x}_1'' \mathbf{A}_2^\top) \mid \mathbf{x}_2' \in \mathscr{S}_{\lceil \frac{z}{2} \rceil, p} \right\}$;
4 Compute $\mathscr{L}_1$, the set of all pairs $(\mathbf{x}_1', \mathbf{x}_1'') \in \mathscr{S}_{\lfloor \frac{z}{2} \rfloor, p} \times \mathscr{S}_{\lceil \frac{z}{2} \rceil, p}$ such that
  $\mathbf{x}_1' \mathbf{A}_1^\top = -\mathbf{x}_1'' \mathbf{A}_2^\top$;
5 Write $\mathbf{B} = (\mathbf{B}_1, \mathbf{B}_2)$, where $\mathbf{B}_1 \in \mathbb{F}_q^{\ell_2 \times z}$, $\mathbf{B}_2 \in \mathbb{F}_q^{\ell_2 \times u-z}$;
6 Set $\mathscr{L}_2' = \left\{ (\mathbf{x}_1, \mathbf{x}_1 \mathbf{B}_1^\top) \mid \mathbf{x}_1 \in \mathscr{L}_1 \right\}$;
7 Set $\mathscr{L}_2'' = \left\{ (\mathbf{x}_2, \mathbf{x}_2 \mathbf{B}_2^\top) \mid \mathbf{x}_2 \in \mathscr{S}_{u-z,p} \right\}$;
8 Compute $\mathscr{L}_2$, the set of all pairs $(\mathbf{x}_1, \mathbf{x}_2) \in \mathscr{L}_1 \times \mathscr{S}_{u-z,p}$ such that
  $\mathbf{x}_1 \mathbf{B}_1^\top = -\mathbf{x}_2 \mathbf{B}_2^\top$;
9 return $\mathscr{L}_2$
```

4.3 Complexity Analysis

We now analyze the computational complexity of SparseISD. Clearly, performing the collision search via meet-in-the-middle twice, is what impacts the algorithm's cost the most. The first merge is performed on $\mathscr{L}_1'$ and $\mathscr{L}_1'$. For the sake of clarity, we neglect floors and ceilings, hence we are considering two lists of equal length

$$L_1 = |\mathscr{L}_1'| = |\mathscr{L}_1''| = \binom{z/2}{p_1}. \tag{4.2}$$

With the same assumption made during the description of Stern algorithm, we have that the resulting list $\mathscr{L}_1$, on average has length

$$|\mathscr{L}_1| = |\mathscr{L}_1'| \cdot |\mathscr{L}_1''| q^{-\ell_1} = L_1^2 q^{-\ell_1}.$$

The second meet-in-the-middle involves $\mathscr{L}_2'$ and $\mathscr{L}_2$. The former has length equal to $\mathscr{L}_1$, while the latter has length

$$L_2 = |\mathscr{L}_2''| = \binom{u-z}{p_2}. \tag{4.3}$$

The merge results in list $\mathscr{L}_2$, which has length

$$|L_2| = |\mathscr{L}_2'| \cdot |\mathscr{L}_2''| = L_1^2 q^{-\ell_1} \cdot L_2 q^{-\ell_2} = L_1^2 L_2 q^{-\ell_1-\ell_2}. \tag{4.4}$$

Drawing from this results, let us formally define the algorithm's cost.

Proposition 4.3.1. ***Performances of SparseISD (1)***
Let ℓ_1, ℓ_2, p_1 and p_2 be the parameters for SparseISD. Let $\ell = \ell_1, \ell_2$ and u such that

$$n - u = n - k - \ell \implies u = k + \ell$$

Let $\bar{z} = \mathbb{E}[z]$. Then, the time complexity of SparseISD's Solve *subroutine, is*

$$t_{\mathsf{Solve}}(\ell 1, \ell 2, p_1, p_2, \bar{z}) = 2L_1 + \frac{L_1^2}{q^{\ell 1}} + L_2 + \frac{L_1^2 L_2}{q^{\ell_1+\ell_2}},$$

where

$$L_1 = \binom{\bar{z}/2}{p_1}, \quad L_2 = \binom{u - \bar{z}}{p_2}.$$

The probability that a codeword $c \in \mathscr{C}_w$ is returned is:

$$p_{\mathsf{ISD}} = \frac{\binom{z/2}{p_1}^2 \binom{u-z}{p_2} \binom{n-u}{w-2p_1-p2}}{\binom{n}{w}}.$$

Proof. The cost t_{Solve} of the Solve subroutine is given by the cost of lists' enumeration, and the application of the 2 meet-in-the-middle procedures. The stated value is obtained as a combination of eq. (4.2), eq. (4.3) and eq. (4.4).

The success probability is given by the probability of having assumed the correct weight distribution, that is represented by the number of codewords following the distribution, over the total number of weight-w codewords. This clearly results in the p_{ISD} stated in the proposition.

□

Proposition 4.3.2. ***Performances of SparseISD (2)***
On average, SparseISD runs in time

$$O\left(\frac{n^3 + n \cdot \left(2L_1 + \frac{L_1^2}{q^{\ell_1}} + L_2 + \frac{L_1^2 L_2}{q^{\ell_1+\ell_2}}\right)}{\binom{\bar{z}/2}{p_1}^2 \binom{u-\bar{z}}{p_2} \binom{n-u}{w-2p_1-p2} / \binom{n}{w}}\right)$$

Proof. The proposition is a direct result of proposition 2.4.1 and proposition 4.3.1. □

When working with the ensemble described in the previous chapter, one can get a precise estimate of the expected value of z, that is the average size of the support of the ℓ_1 randomly selected rows:

$$\bar{z} = \mathbb{E}[z] = n\left(1 - (1-\nu)^{\ell_1}\right) \tag{4.5}$$

This result makes it possible for us to execute numerical estimation of the behavior of our algorithm when working with the different ensembles.

4.4 Comparison with Stern algorithm

Since SparseISD is a modified version of Stern's ISD, it is natural to compare the two algorithms to understand their differences. Both algorithms were analyzed on the two most promising ensembles introduced in chapter 3, namely those with densities $\nu = \frac{\ln(n)^2}{n}$ and $\nu = \frac{\sqrt{n}}{n}$. Numerical experiments were conducted by varying code length n and code rate R.

The results of these experiments show that, in general, our algorithm tends to follow or improve the performances when working with sparse parity-check matrices. However, the entity of the improvement is strictly correlated to code properties. We can see the best improvements for high code rates in both ensembles. Due to the exponential nature of the algorithms, these differences become more appreciable as n grows. The only exception is for $R = 0.4$ and $\nu = \frac{\ln(n)}{n}$, where there's even a slight loss in performances for high n. Nevertheless, even in this case, for tractable code sizes there's still some, although small, improvement.

For the different cases analyzed, we present plots that illustrate the computational cost behavior of the two algorithms. We chose to use a cost coefficient normalized with respect to n, as this provides a clearer understanding of the asymptotic performance across varying code lengths, and ensures a fair comparison. In other words, if ops is the number of operations needed for one algorithms, on the axis we have n, and on the y-axis we are plotting:

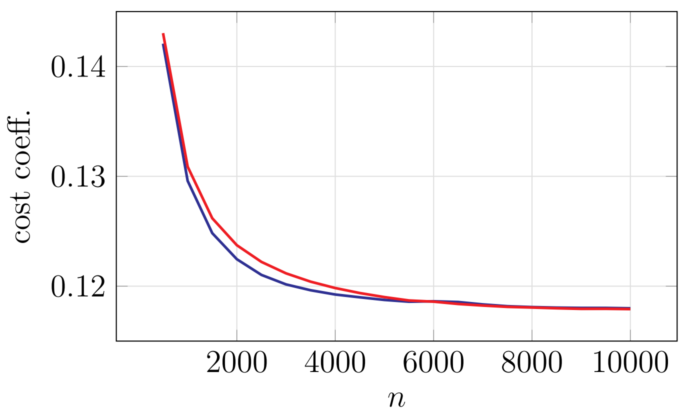
$$\mathsf{cost\_coeff.} = \frac{\log_2(\mathsf{ops})}{n}$$

Additionally, we include tables explicitly showing the computational costs, expressed as powers of 2, to highlight the improvements achieved.

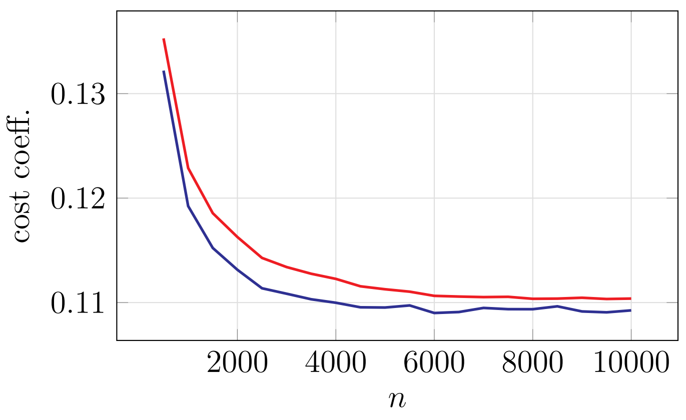
4.4.1 Ensemble with $\nu = \frac{\ln n^2}{n}$ (LDPC)

The first ensemble we examined is the one of codes with density $\nu = \frac{\ln(n)^2}{n}$. On average these codes have a number of nonzero elements per row that is, clearly, $\ln(n)$. This is a typical value for the construction of LDPC codes.

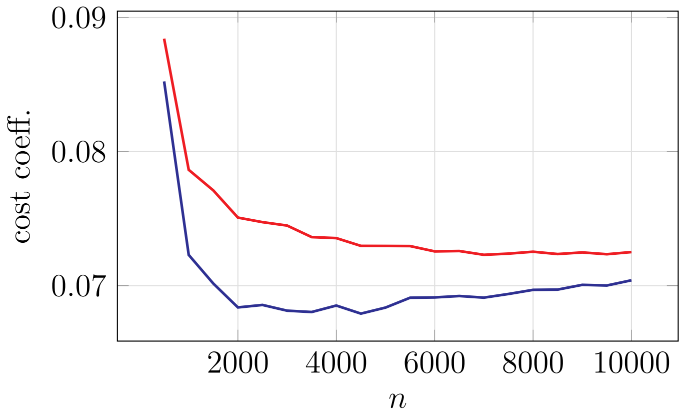
In figure fig. 4.4 we have plotted the cost coefficients for both algorithms. As already stated, as R grows, the difference in performances between our algorithm and Stern ISD grows too.



(a) $R = 0.4$



(b) $R = 0.6$



(c) $R = 0.8$

— SparseISD — Stern

Figure 4.4: Cost coefficients for $\nu = \frac{\ln(n)^2}{n}$, for several values of n and R.

	SparseISD		Stern		Gain
n	cost_coeff.	$\log_2$(ops)	cost_coeff.	$\log_2$(ops)	$\log_2$(gain)
500	0.142	71	0.143	71	0
1,000	0.13	129	0.131	130	1
1,500	0.125	187	0.126	189	2
2,000	0.122	244	0.124	247	2
2,500	0.121	302	0.122	305	2
3,000	0.12	360	0.121	363	3
3,500	0.12	418	0.12	421	2
4,000	0.119	476	0.12	479	2
4,500	0.119	535	0.119	537	1
5,000	0.119	593	0.119	595	1
5,500	0.119	652	0.119	652	0
6,000	0.119	711	0.119	711	−1
6,500	0.119	770	0.118	769	−2
7,000	0.118	828	0.118	827	−1
7,500	0.118	886	0.118	885	−1
8,000	0.118	944	0.118	944	−1
8,500	0.118	1,003	0.118	1,002	−1
9,000	0.118	1,062	0.118	1,061	−1
9,500	0.118	1,121	0.118	1,120	−1
10,000	0.118	1,179	0.118	1,179	−1

Table 4.1: SparseISD and Stern performances with $R = 0.4$, $\nu = \frac{\sqrt{n}}{n}$

	SparseISD		Stern		Gain
n	cost_coeff.	$\log_2$(ops)	cost_coeff.	$\log_2$(ops)	$\log_2$(gain)
500	0.132	66	0.135	67	1
1,000	0.119	119	0.123	122	3
1,500	0.115	172	0.119	177	4
2,000	0.113	226	0.116	232	6
2,500	0.111	278	0.114	285	7
3,000	0.111	332	0.113	340	7
3,500	0.11	386	0.113	394	8
4,000	0.11	439	0.112	449	9
4,500	0.11	492	0.112	502	9
5,000	0.11	547	0.111	556	8
5,500	0.11	603	0.111	610	7
6,000	0.109	654	0.111	663	9
6,500	0.109	709	0.111	718	9
7,000	0.109	766	0.111	773	7
7,500	0.109	820	0.111	829	8
8,000	0.109	874	0.11	882	7
8,500	0.11	931	0.11	938	6
9,000	0.109	982	0.11	994	11
9,500	0.109	1,036	0.11	1,048	12
10,000	0.109	1,092	0.11	1,103	11

Table 4.2: SparseISD and Stern performances with $R = 0.6$, $\nu = \frac{\log(n)^2}{n}$

	SparseISD		Stern		Gain
n	cost_coeff.	$\log_2$(ops)	cost_coeff.	$\log_2$(ops)	$\log_2$(gain)
500	0.085	42	0.088	44	1
1,000	0.072	72	0.079	78	6
1,500	0.07	105	0.077	115	10
2,000	0.068	136	0.075	150	13
2,500	0.069	171	0.075	186	15
3,000	0.068	204	0.074	223	19
3,500	0.068	238	0.074	257	19
4,000	0.069	274	0.074	294	20
4,500	0.068	305	0.073	328	22
5,000	0.068	341	0.073	364	22
5,500	0.069	380	0.073	401	21
6,000	0.069	414	0.073	435	20
6,500	0.069	450	0.073	471	21
7,000	0.069	483	0.072	506	22
7,500	0.069	520	0.072	542	22
8,000	0.07	557	0.073	580	22
8,500	0.07	592	0.072	615	22
9,000	0.07	630	0.072	652	21
9,500	0.07	665	0.072	687	22
10,000	0.07	704	0.073	725	21

Table 4.3: Costs coefficients, $R = 0.8$, $\nu = \frac{\log(n)^2}{n}$

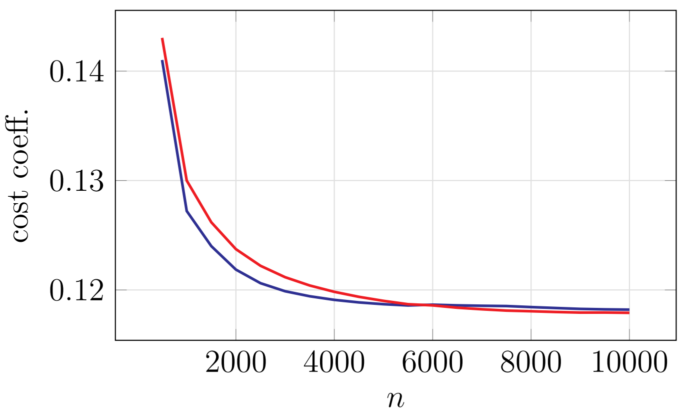
In table 4.1, we observe that for $R = 0.4$, the two algorithms perform similarly overall. However, there is a notable difference at $n = 3000$, where our algorithm is 6 times faster than Stern's. After this peak, performance starts to decline, and by $n = 6500$, our algorithm becomes 4 times slower than Stern's.

For $R = 0.6$ and $R = 0.8$, the number of saved operations increases as n grows. When $R = 0.6$ (see table 4.2), the algorithm reaches its peak improvement at $n = 9500$, running 2^{12} times faster. When $R = 0.8$ (see table 4.3), the improvement is even greater, reaching the same advantage of the previous case already at $n = 2000$, and peaking at a speedup of 2^{22}.

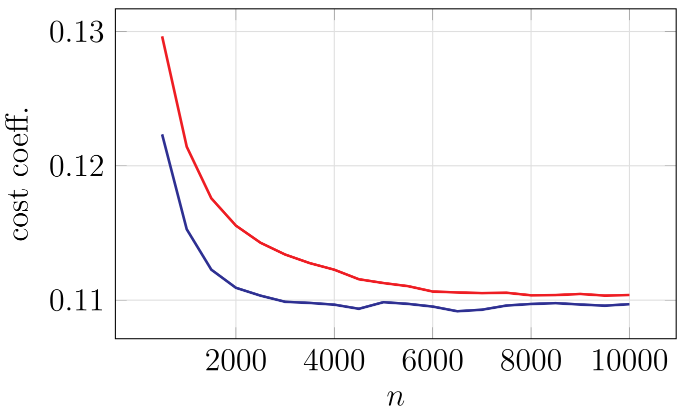
4.4.2 Ensemble with $\nu = \frac{\sqrt{n}}{n}$ (MDPC)

Now we examine the ensemble of codes with density $\nu = \frac{\sqrt{n}}{n}$. It is easy to see that, in this case, the average number of nonzero elements is $\nu = \sqrt{n}$, making codes in this ensemble similar to MDPC codes.

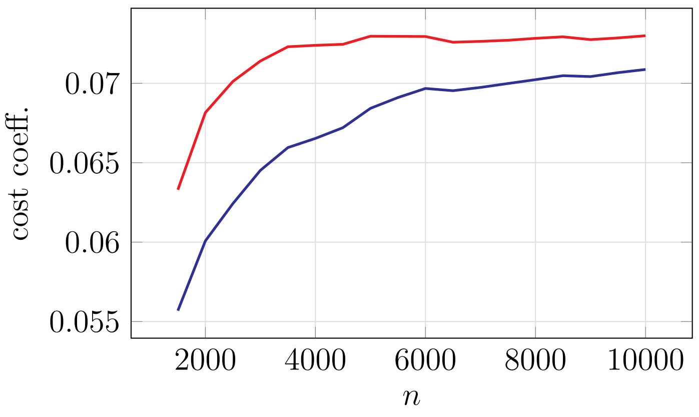
Numerical experiments, plotted in fig. 4.5, yield results similar to those of the previous ensemble,



(a) $R = 0.4$



(b) $R = 0.6$



(c) $R = 0.8$

Figure 4.5: Cost coefficients for $\nu = \frac{\sqrt{n}}{n}$, for several values of n and R.

	SparseISD		Stern		Gain
n	cost_coeff.	$\log_2$(ops)	cost_coeff.	$\log_2$(ops)	$\log_2$(gain)
500	0.141	70	0.143	71	1
1,000	0.127	127	0.13	129	2
1,500	0.124	186	0.126	189	3
2,000	0.122	243	0.124	247	3
2,500	0.121	301	0.122	305	3
3,000	0.12	359	0.121	363	3
3,500	0.119	417	0.12	421	3
4,000	0.119	476	0.12	479	2
4,500	0.119	534	0.119	537	2
5,000	0.119	593	0.119	595	1
5,500	0.119	652	0.119	652	0
6,000	0.119	711	0.119	711	−1
6,500	0.119	770	0.118	769	−2
7,000	0.119	829	0.118	827	−3
7,500	0.119	888	0.118	885	−4
8,000	0.118	947	0.118	944	−4
8,500	0.118	1,005	0.118	1,002	−4
9,000	0.118	1,064	0.118	1,061	−4
9,500	0.118	1,123	0.118	1,120	−3
10,000	0.118	1,182	0.118	1,179	−3

Table 4.4: Costs coefficients, $R = 0.4$, $\nu = \frac{\sqrt{n}}{n}$

	SparseISD		Stern		Gain
n	cost_coeff.	$\log_2$(ops)	cost_coeff.	$\log_2$(ops)	$\log_2$(gain)
500	0.122	61	0.13	64	3
1,000	0.115	115	0.121	121	6
1,500	0.112	168	0.118	176	7
2,000	0.111	221	0.116	231	9
2,500	0.11	275	0.114	285	9
3,000	0.11	329	0.113	340	10
3,500	0.11	384	0.113	394	10
4,000	0.11	438	0.112	449	10
4,500	0.109	492	0.112	502	9
5,000	0.11	549	0.111	556	7
5,500	0.11	603	0.111	610	7
6,000	0.11	657	0.111	663	6
6,500	0.109	709	0.111	718	9
7,000	0.109	765	0.111	773	8
7,500	0.11	821	0.111	829	7
8,000	0.11	877	0.11	882	5
8,500	0.11	933	0.11	938	5
9,000	0.11	987	0.11	994	7
9,500	0.11	1,041	0.11	1,048	7
10,000	0.11	1,097	0.11	1,103	6

Table 4.5: Costs coefficients, $R = 0.6$, $\nu = \frac{\sqrt{n}}{n}$

	SparseISD		Stern		Gain
n	cost_coeff.	$\log_2$(ops)	cost_coeff.	$\log_2$(ops)	$\log_2$(gain)
1,500	0.056	83	0.063	94	11
2,000	0.06	120	0.068	136	16
2,500	0.062	156	0.07	175	19
3,000	0.065	193	0.071	214	20
3,500	0.066	230	0.072	253	22
4,000	0.067	266	0.072	289	23
4,500	0.067	302	0.072	326	23
5,000	0.068	342	0.073	364	22
5,500	0.069	380	0.073	401	21
6,000	0.07	418	0.073	437	19
6,500	0.07	451	0.073	471	19
7,000	0.07	488	0.073	508	20
7,500	0.07	524	0.073	545	20
8,000	0.07	561	0.073	582	20
8,500	0.07	599	0.073	619	20
9,000	0.07	633	0.073	654	20
9,500	0.071	671	0.073	692	20
10,000	0.071	708	0.073	729	21

Table 4.6: Costs coefficients, $R = 0.8$, $\nu = \frac{\sqrt{n}}{n}$

Results are analog to those of the previous case, having larger speedups as R increases. For $R = 0.4$ (see table 4.4) we have an initial gain, but decades rapidly, becoming negative at $n = 6000$. For $R = 0.6$ (see table 4.5) our algorithm performs better than Stern across all n, running 8 to 1024 times faster. For $R = 0.8$ (see table 4.6) there's an even more significant difference in performances, starting from a speedup of 2^{11} for $n = 1500$, and peaking at 2^{23} for $n = 4000$.

5 Proof of Concept Implementation

In this chapter we present a proof of concept implementation of SparseISD. This implementation serves solely to validate the theoretical model of our algorithm. Nevertheless, design choices are justified, laying the foundations for a future optimized and practical implementation.

5.1 Design

We implemented our algorithm using SageMath, a free and open-source mathematical system, that offers a python-based language interface, built on top of other important mathematical packages. There are many reasons behind this choice. First and foremost, SageMath, unlike other programming languages, offers a very simple and clean syntax for algebraic operations. Second of all, it handles error correction codes, and offer related useful functions, such as calculating the hamming weight of a codeword, or finding the minimum distance of a codeword through exhaustive search. This last function, in particular, was used to calculate beforehand the minimum distance of the codes we have benchmarked our algorithm on. More specifically we have used an algorithm supplied by the GUAVA package of the GAP library, trough API available in SageMath. This, together with access to all libraries available for the python language, makes SageMath is the best candidate for our proof of concept implementation.

5.2 Collision search

We implemented a hash-based collision search, where collisions are found creating hash tables and then searching for collisions in keys. Hash tables, also called dictionaries, are data structures that map keys to values, using a hash function to generate unique indexes from keys, from which the values can recovered.

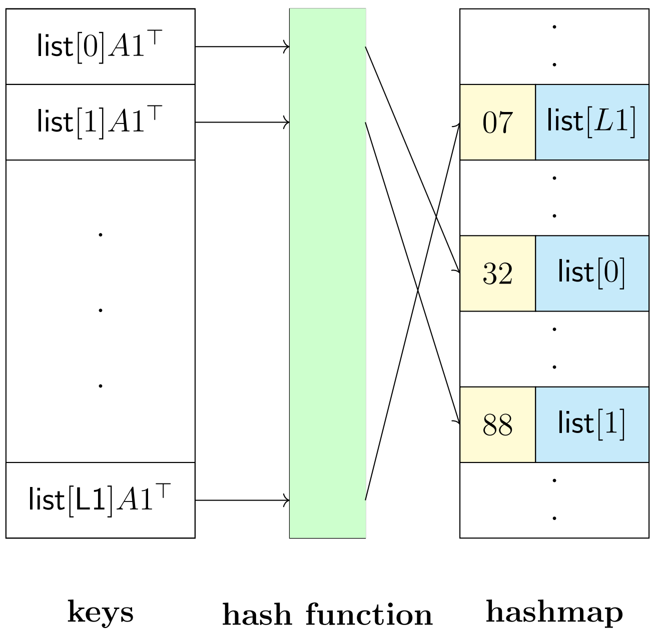


Figure 5.1: Hash tables functioning.

The alternative approach is sorting both lists and then performing a binary search on them, but for large lists this is computationally more expensive. The average time complexity of sorting algorithms is $O(n\log_2(n))$. Same goes for the binary search, which has a $O(n\log_2(n))$. Hence, the cost of the procedure would be $O(n\log(n))$. Hash tables, instead, on average have $O(1)$ insertion time and $O(1)$ lookup time. Therefore, building our hash tables and performing the merge have both a cost of $O(n)$. On the other hand this is has a space complexity of $O(n)$. This might be infeasible for very large instances, which we will not analyze with our proof of concept. The implementation of the collision search can be seen in Listing 5.1.

```
def collision_search(list1, list2, A1, A2):
    hash_table1 = {tuple(x1*A1.T) : x1 for x1 in list1}
    hash_table2 = {tuple(x2*A2.T) : x2 for x2 in list2}

    collisions = []
    for key, x1 in hash_table1.items():
        if key in hash_table2:
            x2 = hash_table2[key]
            collisions.append(vector(GF(2), list(x1) + list(x2)))

    return collisions
```

Listing 5.1: Collision search using hash tables

5.3 Solve subroutine

In Listing 5.2 can see how the implementation of the Solve subroutine in SageMath is really concise. The submatrices are generated at the beginning, then lists are created through enumeration, and finally these lists are used for collision search.

```
def sparseISD_Solve(H, l1, l2, z, p1, p2):
    l = l1 + l2
    A = H[:l1, :z]
    z1 = floor(z/2); z2 = ceil(z/2)
    A1 = A[:,:z1]; A2 = A[:,z1:]
    B = H[l1:l, :k+l]
    B1 = B[:, :z]; B2 = B[:, z:]

    list1_1 = enum_vectors(z1,p1)
    list1_2 = enum_vectors(z2,p1)
    list1 = collision_search(list1_1,list1_2,A1,A2)
    list2 = enum_vectors(k+l−z,p2)
    solutions = collision_search(list1,list2,B1,B2)

    return solutions
```

Listing 5.2: SparseISD Solve subroutine.

5.3.1 Ensemble generation

The generation of parity-check matrices is straightforward. The Matrix method of SageMath accepts a Lambda function for the generation of each element. In Listing 5.3, we can see how we generate elements according to the binomial distribution with probability ν. To avoid the generation of degenerate codes, the parity-check matrix is regenerated until it has no null columns.

```
def generate_matrix(density):
        H = Matrix(GF(2), r, n, lambda i, j: random.random() < density)
        while any(col.is_zero() for col in H.columns()):
            H = Matrix(GF(2), r, n, lambda i, j: random.random() < density)
        return H
```

Listing 5.3: H generation

6 Future developments

Our work was primarily focused on binary codes, but in the future we would like to generalize our work to codes defined over arbitrary finite fields $\mathbb{F}_q$. This means expanding the analysis of our ensemble, so that it consider distributions wider than the classic binomial one. Same goes for our implementation, that's currently only able to operate with binary codes.

We studied and characterized irregular LDPC codes through the introduced ensemble, and we did it generalizing the classic ensemble of binary random codes. Our analysis did not consider Tanner graphs, but in the future we would like to draw some connections between LDPC codes and graphs. Many combinatorial objects, like cycles and trapping sets, arise when studying these codes, and these structure are widely characterized in studied in graph theory. Understanding these relationship further, could help us improve our algorithm and design better heuristics that make use of Tanner graph properties. This is strongly motivated by some results in graph theory that are strictly related to problems in coding theory. For instance, the clique problem, a typical NP-*complete* problem, gets significantly easier when studying sparse graphs, becoming solvable in polynomial time.

For what concerns the implementation of SparseISD, we would like to develop a faster and more practical version. The proof-of-concept, as such, doesn't consider a wide variety of optimizations that could be applied. In the current state, large instances are unfeasible for our algorithm. We intend to reimplement the algorithm using a lower level programming language such a C, opening the doors for a variety of tweakings that can severely fasten the algorithm. Among these techniques, we plan to exploit commonly used ones for ISD implementations, such as:

- **Intermediate sums**: technique used to speed up computations done on sets of vectors. It consists on performing the computations starting from the easier case and use the obtain results to perform the operation on harder cases. The natural application in our algorithm is the calculation of lists used in meet-in-the-middle, starting from codewords of lower weight onward.

- **Early abort**: this technique is used when solutions of an algorithm need to satisfy a constraint, but this can be evaluated before computing the entire solution. An example related to our algorithm, that can help better understand this technique, is when, after performing the Solve subroutine, we compute the rightmost codeword partition. We usually check the codeword weight after computing it all. Applying early abort, one could stop the computation earlier if, after computing some elements, the given weight is already exceeded, discarding the current codeword.

We aim to use system optimization when possible, such as leveraging the Single Instruction Multiple Data (SIMD) parallel computing model, where as single instruction is operated on multiple elements simultaneously. This paradigm is particularly suited when working with vectors and matrices. A direct example, is the computation of the product between 2 matrices, where multiple elements of the resulting matrix could be computed simultaneously. Many modern CPU's instruction sets have SIMD extensions: common ones are **AVX** for x86 architectures, and **NEON** for ARM architectures.

7 Conclusion

In this thesis a new ISD algorithm for finding the minimum distance of LDPC codes was presented. A formal description of the algorithm, along with computational complexity analysis, was provided. The algorithm performances were compared to those of the most commonly used ISD algorithm, which is Stern algorithm. To accomplish this, we introduced a new ensemble of codes, which is a generalization of the ensemble of random codes. We proved that for densities that grow faster than $\frac{\ln(n)}{n}$, LDPC codes have a weight distribution analog to those of random codes. Through numerical experiments it was showed how, for code rates $R \geq 0.6$, our algorithm achieves an asymptotic speedup of $2^{21}\times$. The algorithm was further validated through a proof-of-concept implementation, which was tested on a set of codes from our ensemble. Benchmarks were executed on small instances, but we intend to test larger instances in the future. Results show, as expected, that SparseISD has better performances than Stern algorithm when applied to LDPC codes. We laid the foundations for the development of new ISD algorithms, tailored to exploit the sparsity of parity-check matrices. Furthermore, the presented ensemble shall be useful for studying a variety of algorithms that work with LDPC codes, and it should not be considered merely instrumental for our work. Nevertheless, our algorithm is still exponential-time, but through analysis of the relations between LDPC codes and graphs, we hope to improve our algorithm in the near future.

Bibliography

[1] Nicolas Aragon et al. “BIKE: bit flipping key encapsulation”. In: (2022).

[2] E. Berlekamp, R. McEliece, and H. van Tilborg. “On the inherent intractability of certain coding problems (Corresp.)” In: *IEEE Transactions on Information Theory* 24.3 (1978), pp. 384–386. DOI: 10.1109/TIT.1978.1055873.

[3] B Erlekamp. *ER (1968): Algebraic coding theory.*

[4] R. Gallager. “Low-density parity-check codes”. In: *IRE Transactions on Information Theory* 8.1 (1962), pp. 21–28. DOI: 10.1109/TIT.1962.1057683.

[5] Pil Joong Lee and Ernest F Brickell. “An observation on the security of McEliece’s public-key cryptosystem”. In: *Workshop on the Theory and Application of of Cryptographic Techniques.* Springer. 1988, pp. 275–280.

[6] Nicholas Patterson. “The algebraic decoding of Goppa codes”. In: *IEEE Transactions on Information Theory* 21.2 (1975), pp. 203–207.

[7] Christiane Peters. “Information-Set Decoding for Linear Codes over Fq”. In: *Post-Quantum Cryptography.* Ed. by Nicolas Sendrier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 81–94. ISBN: 978-3-642-12929-2.

[8] E. Prange. “The use of information sets in decoding cyclic codes”. In: *IRE Transactions on Information Theory* 8.5 (1962), pp. 5–9. DOI: 10.1109/TIT.1962.1057777.

[9] Jacques Stern. “A method for finding codewords of small weight”. In: *Coding Theory and Applications: 3rd International Colloquium Toulon, France, November 2–4, 1988 Proceedings 3.* Springer. 1989, pp. 106–113.

[10] Alexander Vardy. “The intractability of computing the minimum distance of a code”. In: *IEEE Transactions on Information Theory* 43.6 (1997), pp. 1757–1766.

[11] Violetta Weger, Niklas Gassner, and Joachim Rosenthal. “A survey on code-based cryptography”. In: *arXiv preprint arXiv:2201.07119* (2022).