Master Project
Winter Semester 2005-2006

List-decoding algorithms

Student : Julie Marc

Supervisor : Andrew Brown.
Responsible Professor: Amin Shokrollahi.

Acknowledgments

I would like to thank Prof. Shokrollahi for giving me the opportunity to realize this project and Andrew Brown for his availability and his advice. I would like also to express my gratitude to the secretary of the ALGO-LMA group, Natascha Fontana, for her kindness and for the practical assistance. Finally special thanks to my family and friends who have supported me throughout my work.

Contents

Introduction **6**

1 Coding and decoding theory **9**
- 1.1 Theoretical reminder 9
- 1.2 Reed-Solomon codes 13
- 1.3 Decoding of Reed-Solomon codes 14

2 The Welch-Berlekamp algorithm **17**
- 2.1 Presentation of the algorithm 17
- 2.2 Error bound 20

3 The Sudan algorithm **21**
- 3.1 Presentation of the algorithm 21
  - 3.1.1 Overview 21
  - 3.1.2 Formal analysis 23
- 3.2 Error bound 26
- 3.3 Implementation 27

4 The Guruswami-Sudan algorithm **31**
- 4.1 The Johnson bound 31
- 4.2 Presentation of the algorithm 33
  - 4.2.1 Overview 35
  - 4.2.2 Formal analysis 36
- 4.3 Error bound 40
- 4.4 Complexity analysis and implementation 40
  - 4.4.1 Results analysis 41
- 4.5 Size of the output list 48
  - 4.5.1 Results analysis 50

5 The Parvaresh-Vardy algorithm **55**
- 5.1 Presentation of the algorithm 55
- 5.2 Error bound 63
- 5.3 Parvaresh-Vardy algorithm, general case 63
- 5.4 Comparison with the Guruswami-Sudan algorithm 65

5.5 Implementation 65

6 The PVGS algorithm **69**

6.1 PVGS algorithm 69

6.1.1 Step 1, Parvaresh-Vardy algorithm 70

6.1.2 Step 2, Guruswami-Sudan algorithm 74

6.2 Error bound 75

6.3 Size of the output list 76

6.4 Comparison with the previous algorithms 77

6.5 PVGS algorithm, general case 80

6.5.1 Cases $M > 2$ 80

6.5.2 Other cases 81

6.6 Conclusion 83

7 Soft-decision decoding algorithm **85**

7.1 The Koetter-Vardy algorithm 86

7.2 Construction of the multiplicity matrix 89

7.3 Conclusion 93

Conclusion **93**

A Some results **97**

Bibliography **98**

Introduction

The goal of this project is to present the main list-decoding algorithms for Reed-Solomon codes, implement them and study more specifically some of their properties and possible improvements.

Reed-Solomon codes were invented in 1960 by Irving S. Reed and Gustave Solomon. However, at that time digital technology was not advanced enough to make practical use of this new concept. The key to the application of Reed-Solomon codes was the invention of an efficient decoding algorithm by Elwyn Berlekamp in the early 1960's. Although the notion of list-decoding was introduced as early as 1960, the first significant result for list-decoding is due to Sudan in 1997. Subsequently, a number of improvements on this algorithm, and new list-decoders have been presented. Today Reed-Solomon codes are in widespread use. A large proportion of the error-correction circuits in operation decode Reed-Solomon codes. For example CD, DVD players and most computer hard drives use these codes.

We will begin this work with some theoretical background. In the first chapter we will define Reed-Solomon codes and the List-decoding Problem. The second chapter will introduce the Welch-Berlekamp algorithm, upon which the other algorithms presented in this project are based.

In chapters 3 and 4, we will present the Sudan and Guruswami-Sudan list-decoding algorithms. We will start with an overview of each one before detailing the proofs of efficiency and correctness. We have also implemented both of these algorithms. For the Sudan algorithm, we will particularly discuss the implementation of the Roth-Rückenstein algorithm, which is used to find roots of bivariate polynomials. Our implementation of the Guruswami-Sudan algorithm will allow us to study the link between its efficiency and the parameters used. We will notably observe that in practice the Guruswami-Sudan cannot always achieve its theoretical error correction capacity. We will also make simulations to study the size of its output list and compare our experimental results with the theory.

Chapter 5 will present in detail the Parvaresh-Vardy algorithm. It partly

improves the performances of the Guruswami-Sudan algorithm, using slightly different codes (subsets of Reed-Solomon codes). We have also implemented this algorithm, mainly to show that it can be done. Moreover it will help us discuss the difficulties in producing an implementation that is really efficient in practice.

Chapter 6 will present a new algorithm that improves all previous results. It is based on both the Guruswami-Sudan and Parvaresh-Vardy algorithms. We will study it in detail and show its improvements over the Guruswami-Sudan and Parvaresh-Vardy algorithms. We have also implemented this algorithm, in order to test whether it can really be used in practice.

Finally, chapter 7 will briefly presents another way to decode: soft-decision decoding, which uses probabilistic reliability information of the channel. Thus, this chapter contains a description of the Koetter-Vardy algorithm which uses some ideas from the Guruswami-Sudan algorithm.

Chapter 1

Coding and decoding theory

Before studying List-decoding, we need to review basic notions of coding theory. Here we suppose that the reader has a basic knowledge in algebra and coding theory since we will define the notions rather briefly.

The problem of the communication of information is of great importance today. This problem can be defined as follows. One or more people would like to transmit information to one or more receivers through a channel. Phone and internet and satellite are examples of such channels. Unfortunately there may be perturbations on the channel and transmitted information can be corrupted, i.e., information can be lost or wrongly transformed by the "noise" and interferences on the channel.

We cannot avoid the perturbations but we can introduce redundancy in information in order to improve the reliability of transmission. Typically, if one has to transmit a message which consists of a finite string of symbols that are elements of some finite alphabet, e.g., a finite field, one extends the string of message symbols to a longer string. Now we assume that the symbols of the message and of the coded message are elements of the same finite field $\mathbb{F}_q$. Coding means to encode a block of k chosen message symbols $m_1 \dots m_k,\ m_i \in \mathbb{F}_q$, into a codeword $c_1 c_2 \dots c_n$ of n symbols $c_j \in \mathbb{F}_q$, where $n > k$. Thus, the quantity $n - k$ is sometimes called the redundancy. Figure 1.1 gives a simple scheme of a communication through a channel.

We will now give the nine definitions needed before proceeding further.

1.1 Theoretical reminder

Definition 1.1 (Linear code)**.** *Let H be a matrix of dimension $(n-k) \times n$ and of rank $n-k$ with entries in $\mathbb{F}_q$. The set C of all vectors $\mathbf{c} \in \mathbb{F}_q^n$ such that $H\mathbf{c}^T = \mathbf{0}$ is called a linear code of parameters $[n, k]$ over $\mathbb{F}_q$; n is called*

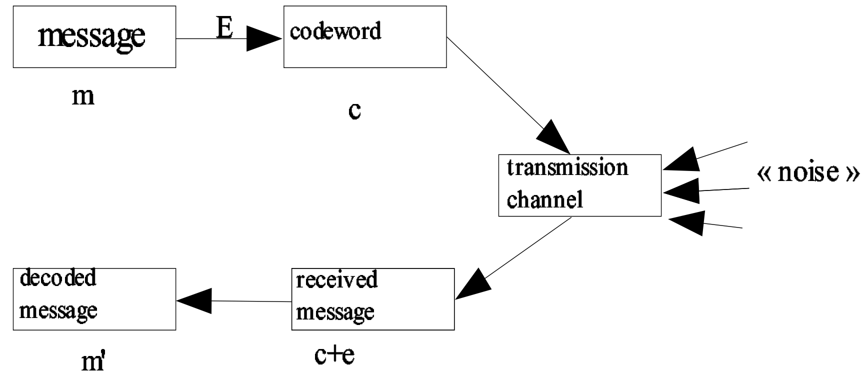

Figure 1.1: Communication through a noisy channel.

the length of the code and k the dimension of the code. The elements of C are called codewords.

Suppose now that we want to encode the message $\mathbf{m} \in \mathbb{F}_q^k$ using the code C. In order to do that, we use the encoding function.

Definition 1.2 (Encoding function)**.** *Let C be a $[n, k]$ code over $\mathbb{F}_q$. An injective function $E : \mathbb{F}_q^k \to C \subseteq \mathbb{F}_q^n$ is called an encoding function.*

Definition 1.3 (Hamming distance)**.** *Let $\mathbf{x}, \mathbf{y}$ be two vectors in $\mathbb{F}_q^n$. Then:*

(i) *The Hamming distance $d(\mathbf{x}, \mathbf{y})$ between $\mathbf{x}$ and $\mathbf{y}$ is the number of coordinates in which $\mathbf{x}$ and $\mathbf{y}$ differ.*

(ii) *The Hamming weight $w(\mathbf{x})$ of $\mathbf{x}$ is the number of non-zero coordinates of $\mathbf{x}$.*

It follows immediately from this definition that $w(\mathbf{x}-\mathbf{y}) = d(\mathbf{x}, \mathbf{y})$. Furthermore the Hamming distance is a metric on $\mathbb{F}_q^n$. As mentioned before, the codeword can be corrupted during the transmission and thus the receiver will receive information which may be different from the sent one. Algebraically speaking, the received vector $\mathbf{c} + \mathbf{e}$ (cf. figure 1.1) may be none of the codewords of C and the decoded message $\mathbf{m}'$ may be different from the original message $\mathbf{m}$. The goal of the receiver is to retrieve the original codeword and, of course, the original message. In the next section we will focus on the problem of decoding. We now define more formally a (memoryless) transmission channel.

Definition 1.4 (Memoryless channel). *A memoryless channel is the simplest communication system. This channel is defined by the three following objects:*

1. *a finite input alphabet* $\mathcal{V}_X$*,*
2. *an output alphabet* $\mathcal{V}_Y$*,*
3. *the probability mass function* $P(Y|X)$ *specifying the conditional distribution of the output* Y *given the input* X*. Moreover a given output* Y_i *only depends of the corresponding input* X_i*, i.e.,*

$$P(Y_1, \dots, Y_n | X_1, \dots, X_n) = \prod_{i=1}^{n} P(Y_i|X_i).$$

This is the "memoryless" condition.

An example of memoryless channel is the q-ary symmetric channel where $\mathcal{V}_X = \mathcal{V}_Y = \{\alpha_1, \dots, \alpha_q\}$, α_i's distinct in $\mathbb{F}_q$ and $P(Y \neq \alpha_i | X = \alpha_i) = p$, $\forall i$.

Definition 1.5 (e-error-correcting code). *For* $e \in \mathbb{N}$ *a linear code* C *is called e-error-correcting if for any* $\mathbf{y} \in \mathbb{F}_q^n$ *there is at most one* $\mathbf{c} \in C$ *such that* $d(\mathbf{y}, \mathbf{c}) \leq e$*.*

Definition 1.6 (Minimum distance).

$$d_C = \min_{\substack{\mathbf{u},\mathbf{v} \in C \\ \mathbf{u} \neq \mathbf{v}}} d(\mathbf{u}, \mathbf{v}) = \min_{\mathbf{0} \neq \mathbf{c} \in C} w(\mathbf{c})$$

is called the minimum distance of the linear code C*.*

Proposition 1.1. *A code* C *with minimum distance* d_C *can correct up to* e *errors if* $e \leq \frac{d_C - 1}{2}$*.*

For illustration, see Figure 1.2. In this picture, a ball with center $\mathbf{c_i}$ and radius $d_C/2$ represents all vectors $\mathbf{z} \in \mathbb{F}_q^n$ such that $d(\mathbf{c_i}, \mathbf{z}) \leq d_C/2$. Thus, there is no intersection between the balls, otherwise the minimum distance would be $< d_C$. For some received vector $\mathbf{y}$, there is only one codeword within distance $\frac{d_C - 1}{2}$ from it.

In this project, we will use the notation $[n, k, d_C]_q$ for a linear code of parameters n, k, d_C over $\mathbb{F}_q$. We need two more definitions.

Definition 1.7 (Information rate). *The information rate (or simply rate) of a code* C *with parameters* n, k *is the ratio* $\kappa = \frac{k}{n}$*.*

This ratio k to n is a measure of the fraction of information which is non redundant.

Definition 1.8 (Error-rate). *Let* e *be a fixed number of error and* n *the length of the transmitted codeword. Then the error rate is* $\varepsilon = \frac{e}{n}$*.*

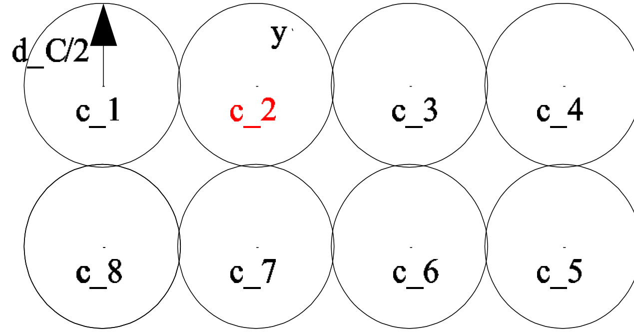

Figure 1.2: There is only one codeword ($\mathbf{c_2}$) within a distance $\frac{d_C-1}{2}$ from the received vector $\mathbf{y}$.

In an ideal world we would like to have a code which can transmit as much information as possible and which has codewords "far apart". Unfortunately there is some bound on this aim.

Proposition 1.2 (Singleton bound)**.** *For a $[n, k, d_C]_q$ code C, the following inequality holds*

$$k + d_C \leq n + 1.$$

Proof: Consider the linear subspace $W \subseteq \mathbb{F}_q^n$ defined by

$$W := \{(a_1, \dots, a_n) \in \mathbb{F}_q^n | a_i = 0 \ \forall i \geq d_C\}.$$

Every $\mathbf{a} \in W$ has Hamming weight $\leq d_C - 1$, thus $W \cap C = 0$. Since $\dim W = d_C - 1$ we get

$$\begin{aligned} k + (d_C - 1) &= \dim C + \dim W \\ &= \dim(C + W) - \dim(C \cap W) \\ &= \dim(C + W) \\ &\leq n \end{aligned}$$

□

Thus, since $d_C \leq n - k + 1$, a large redundancy $n - k$ is desirable in point of view of error-correction. However it decreases the efficiency of the code since n symbols are required to transmit only k symbols of information.

Definition 1.9 (MDS codes)**.** *Codes with $k + d_C = n + 1$ are called MDS (maximum distance separable).*

We have now the tools to define the codes in which we are interested: the Reed-Solomon codes.

1.2 Reed-Solomon codes

These codes have a well-known mathematical structure and useful properties. This is why these codes are often used in practice.

Definition 1.10 (Reed-Solomon codes). *Let q be a prime power, $n \leq q$ and $k \leq n$. Choose n distincts elements $\alpha_1, \dots, \alpha_n \in \mathbb{F}_q$. (Here we need the property $n \leq q$). Let $\mathbb{F}_q[x]_{<k}$ be the set of the polynomials over $\mathbb{F}_q$ of degree $< k$. The encoding function*[1] $E : \mathbb{F}_q[x]_{<k} \cong \mathbb{F}_q^k \to \mathbb{F}_q^n$ *for a Reed-Solomon code is defined as follows:*

$$E(m) = (m(\alpha_1), \dots, m(\alpha_n)).$$

Thus a Reed-Solomon code C of dimension k and length n over $\mathbb{F}_q$ is constructed as follows

$$C := \{(m(\alpha_1), \dots, m(\alpha_n)) | m \in \mathbb{F}_q[x]_{<k}\}.$$

Observe that the weight of a codeword is given by

$$\begin{aligned} w(c) &= n - |\{i \in \{1, \dots, n\} | m(\alpha_i) = 0\}| \\ &\geq n - \deg m \\ &\geq n - (k-1). \end{aligned}$$

Hence the minimum distance d_C of C satisfies the inequality $d_C \geq n+1-k$. Furthermore since Reed-Solomon codes are obviously linear, $d_C \leq n+1-k$ by the Singleton bound. Thus Reed-Solomon codes are MDS codes. Finally we introduce the Generalized Reed-Solomon codes.

Definition 1.11 (Generalized Reed-Solomon codes). *Let $\alpha_1, \dots, \alpha_n$ be distinct elements of $\mathbb{F}_q$ and let $v_1, \dots, v_n$ be non-zero elements of $\mathbb{F}_q$. The encoding function $E : \mathbb{F}_q[x]_{<k} \to \mathbb{F}_q^n$ is given by*

$$E(m) = (v_i \cdot m(\alpha_1), \dots, v_n \cdot m(\alpha_n)).$$

Thus a Generalized Reed-Solomon code, $GRS_k(\alpha, v)$, contains all the vectors

$$(v_i \cdot m(\alpha_1), \dots, v_n \cdot m(\alpha_n))$$

with $m(x) \in \mathbb{F}_q[x]_{<k}$ (for a fixed $k \leq n$).

[1] The isomorphism I between $\mathbb{F}_q[x]_{<k}$ and $\mathbb{F}_q^k$ is defined by $I : \mathbb{F}_q[x]_{<k} \to \mathbb{F}_q^k$, $m(x) = \sum_{i=0}^{k-1} m_i x^i \mapsto \mathbf{m} = (m_0, \dots, m_{k-1})$.

1.3 Decoding of Reed-Solomon codes

First we consider the problem of decoding for general codes. This problem can be formalized in the following way: given a (possibly corrupted) received vector $\mathbf{r} \in \mathbb{F}_q^n$ of an encoding of message $\mathbf{m}$, find $\mathbf{m}$. This definition is informal, but we guess that we are looking for the message $\mathbf{m}$ such that $\Pr(\mathbf{m} \text{ was sent} | \mathbf{r} \text{ is received})$ is maximum. This probability depends on the used probabilistic channel and on the distribution of the messages. When the channel is the q-ary symmetric channel and if we assume that all messages are equiprobable, the previous problem can be simplified to the following one:

Nearest Codeword Problem: For a fixed code C and a given vector $\mathbf{r} \in \mathbb{F}_q^n$, find $\mathbf{c} \in C$ that minimizes $d(\mathbf{r}, \mathbf{c})$, where d is the Hamming distance.

This problem is hard. In particular it asks for error-correction possibly well beyond the minimum distance of the code. It means that $\mathbf{r}$ may be at Hamming distance $> d_C$ from the nearest codeword. To obtain algorithmic results, we need to impose some restrictions. First, we should try to decode only well-known codes (with good properties, with known minimum distance,...). Secondly, we should only consider cases where the number of errors is limited. These restrictions lead to the following problem:

Unambiguous Decoding Problem: For a fixed code C and a given vector $\mathbf{r} \in \mathbb{F}_q^n$, find a codeword $\mathbf{c} \in C$ such that $d(\mathbf{r}, \mathbf{c}) < d_C/2$ if such a codeword exists. Here d_C is the minimum distance of the code C.

In this problem, we suppose that d_C is known. Unfortunately, it is generally hard to compute the minimum distance of an arbitrary code. In the situation of Unambiguous Decoding Droblem, we are sure that the answer is unique by proposition 1.1. Moreover, we will see next that the answer is computable in a reasonable time for Reed-Solomon codes, provided d_C is given. Actually the complexity is still reasonable for larger values of errors ($e > d_C/2$), but in such situations, the answer is no longer unique. This is the following problem:

Bounded Distance Decoding Problem: For a fixed code C, a given error bound e and a given vector $\mathbf{r} \in \mathbb{F}_q^n$, find *one* codeword $\mathbf{c} \in C$ such that $d(\mathbf{r}, \mathbf{c}) \leq e$ if such a codeword exists.

This problem is solvable for some codes and for some bounds e. This is achieved by solving a slightly harder problem called "List-decoding Problem". We can now give the formulation of this problem.

List-decoding Problem: For a fixed code C, a given error bound e

and a given vector $\mathbf{r} \in \mathbb{F}_q^n$, find a list S of *all* codewords $\mathbf{c} \in C$ such that $d(\mathbf{r}, \mathbf{c}) \leq e$.

Step one of the List-decoding Problem is to find the complete and correct subset S of codewords. And step two is to select the right codeword (i.e., the original codeword) within the subset. There are several solutions for step two like asking for a new transmission and looking for intersection between the subsets. One can also suppose that the receiver knows the subject of the transmission or can correlate with other parts of the transmission and thus can retrieve the good codeword from the subset by choosing the most coherent one. The receiver can also simply compare the received vector $\mathbf{r}$ with the codewords in the list and then choose the codeword which is closest to $\mathbf{r}$. For all these reasons, the question of the size of the output list needs to be detailed.

As mentioned earlier, we are particularly interested in Reed-Solomon codes (def. 1.10). We will start decoding Reed-Solomon codes with the Welch-Berlekamp algorithm. This algorithm is an unambiguous decoding algorithm of which the list decoding algorithms are a generalization as we will observe later. Before moving on, note that the Unambiguous Decoding Problem for Reed-Solomon codes can be written in a slightly different way from the general one. This formulation is called the Unambiguous Polynomial Reconstruction Problem.

Unambiguous Polynomial Reconstruction Problem: Given n distincts elements $x_1, \ldots, x_n \in \mathbb{F}_q$, $y_1, \ldots, y_n \in \mathbb{F}_q$ and parameters k and $t \geq \frac{n+k}{2}$, find a polynomial $f \in \mathbb{F}_q[x]$ of degree at most $k-1$, such that $f(x_i) = y_i$ for at least t values of $i \in [n]$.

Let us make the link between the notation here and the one we used in the definition of Reed-Solomon codes (def. 1.10). The x_i's are the points at which we evaluate the message polynomial and correspond to the α_i's, the y_i's are the n coordinates of the received vector $\mathbf{r}$ and the output polynomial $f(x)$ is the message $f(x)$ that the encoder has evaluated at the n points $\alpha_1, \ldots, \alpha_n$. Furthermore, if d_C is the minimum distance of the considered Reed-Solomon code C, we know that $d_C = n-k+1$ and hence the condition $t \geq \frac{n+k}{2}$ implies $n-t \leq \frac{d_C-1}{2} < \frac{d_C}{2}$, where $n-t$ is the number of errors. Thus it is clear that the Unambiguous Polynomial Reconstruction Problem and the Unambiguous Decoding Problem for Reed-Solomon codes are equivalent.

Chapter 2

The Welch-Berlekamp algorithm

In the following chapters 3-5, we will present different list-decoding algorithms which are known today. As mentioned in chapter 1, we begin with the Welch-Berlekamp algorithm which is not a list-decoding algorithm but which is the basis of the next algorithms.

2.1 Presentation of the algorithm

A key definition used by the Welch-Berlekamp algorithm is the notion of error location polynomial.

Definition 2.1 (Error locating polynomial)**.** *Let t, k be parameters. Given a vector $\mathbf{x}$ and a received vector $\mathbf{y}$ such that there exists a polynomial $f(x)$ with degree $\leq k-1$ such that $f(x_i) = y_i$ for at least t values of i, a polynomial $E(x)$ is called an error locating polynomial if it satisfies:*

1. $\deg(E(x)) \leq n - t$
2. $E(x) = 0$ *if* $f(x_i) \neq y_i$

Observe that this polynomial $E(x)$ is not easier to find than f but it has useful properties. Moreover such a polynomial always exists. Take a set $T \subseteq [n]$ that contains all the error locations and satisfies $|T| = n - t$ and let $E(x) = \prod_{i \in T}(x - x_i)$. Finally, we define one more polynomial, $N(x) := f(x) \cdot E(x)$. Finding these polynomials separately is hard, but finding them together along the following properties is easier.

- $E(x_i) = 0$ if $f(x_i) \neq y_i$ and $\deg(E) \leq n - t$.
- $N(x) = f(x)E(x)$ and $\deg(N) \leq n - t + k - 1$

- $\forall i \in [n], \quad N(x_i) = f(x_i)E(x_i) = y_iE(x_i)$

The first two properties are obvious. The third one might require an explanation. If $f(x_i) = y_i$, there is nothing to prove. If $f(x_i) \neq y_i$, we know that $E(x_i) = 0$ and thus $f(x_i)E(x_i) = y_iE(x_i) = 0$. We are now ready to describe the Welch-Berlekamp algorithm, cf table 2.1.

Welch-Berlekamp algorithm:
Input: $n, d, t \geq \frac{n+k}{2}$ and n pairs $\{(x_i, y_i)\}_{i=1}^n$ with x_i's distincts. **Step 1:** Find polynomials N and E, if they exist, satisfying the following constraints:
$\left.\begin{array}{l} \deg(E) \leq n-t, \\ \deg(N) \leq n-t+k-1, \\ \forall i \in [n], \quad N(x_i) = y_iE(x_i). \end{array}\right\} (1)$
Step 2: Compute $f(x) = N(x)/E(x)$.
Output: $f(x)$ or "No such polynomial", if $E(x)$ doesn't divide $N(x)$.

Table 2.1: Welch-Berlekamp algorithm

At this stage it is not clear if this algorithm is efficient and we do not know if the polynomials E and N really exist. We first note that this algorithm can be implemented to run in polynomial time. It is obvious that the second step can be implemented to run in polynomial time and so we just have to prove this for the first step. What we need to find in the first step are the coefficients $E_0, \ldots, E_{n-t}$ and $N_0, \ldots, N_{n-t+k-1}$ such that for all i

$$\sum_{j=0}^{n-t+k-1} N_j x_i^j = y_i \sum_{j=0}^{n-t} E_j x_i^j.$$

For every i this constraint is a linear constraint in the unknowns E_j and N_j. Thus, solving the system above amounts to solving a linear system and hence can be done in polynomial time.

Now that we have proved the efficiency, we need to establish the correctness. For this part, assume that f is a polynomial of degree $\leq k-1$ that agrees with the given set of points at all but $n-t$ points.

Proposition 2.1. *There exists a pair of polynomials E and N satisfying (1) with the property that $N(x)/E(x) = f(x)$.*

Proof: We just take E to be an error-locating polynomials for f and the given set of points, and let $N(x) = f(x)E(x)$. Then N and E satisfy all the requirements.

□

Now we need to check that the output polynomial f is always the same whatever the pair N, E may be. Note that we will not be able to prove that there is a unique such pair. Instead we will show that each pair have the same ratio $N(x)/E(X)$ (and by the claim above this ratio must be f).

Proposition 2.2. *Any solutions (N, E) and (N', E') of (1) satisfy*

$$\frac{N(x)}{E(x)} = \frac{N'(x)}{E'(x)}$$

or equivalently $N(x)E'(x) = N'(x)E(x)$.

Proof: Note that the degree of the polynomials $N(x)E'(x)$ and $N'(x)E(x)$ is at most $2n + k - 1 - 2t$. Furthermore, since these polynomials satisfy (1), we have for every $i \in [n]$,

$$y_i E(x_i) = N(x_i) \quad \text{and} \quad N'(x_i) = y_i E'(x_i).$$

Multiplying the two we get:

$$y_i E(x_i) N'(x_i) = y_i E'(x_i) N(x_i).$$

Now we claim that actually

$$E(x_i) N'(x_i) = E'(x_i) N(x_i).$$

If $y_i \neq 0$ we can divide by y_i and the equality is obvious. If $y_i = 0$, we observe that $N(x_i) = 0 = N'(x_i)$ (from (1) again) and so we have $E(x_i)N'(x_i) = 0 = E'(x_i)N(x_i)$. Now if $n > 2n + k - 1 - 2t$, i.e., if $n < 2t - k + 1$, the polynomials $N \cdot E'$ and $N' \cdot E$ agree on more points than their degree and hence they are identical. The hypothesis of the Welch-Berlekamp algorithm ensures that $t \geq \frac{n+k}{2}$ and consequently $n < 2t - k + 1$ as desired.

□

With these two propositions, we have actually proved the following theorem:

Theorem 2.3. *The Welch-Berlekamp algorithm solves the Unambiguous Decoding Problem (and so the Unambiguous Polynomial Reconstruction Problem) for Reed-Solomon codes in $O(n^3)$ time.*

2.2 Error bound

Finally, notice that the Welch-Berlekamp algorithm can correct a fraction of errors up to

$$\frac{e}{n} = \frac{n-t}{n} = \frac{n - \frac{n-k}{2}}{n} = \frac{1 - \frac{k}{n}}{2} = \frac{1-R}{2}.$$

For future comparisons between the algorithms we will often use the error-rate $\frac{e}{n}$ rather than e.

Chapter 3

The Sudan algorithm

The Welch-Berlekamp algorithm can solve the Unambiguous Decoding Problem (p. 14) for Reed-Solomon codes. However this algorithm can only correct up to $n - t < \frac{d_C}{2}$ errors, where d_C is the minimum distance of the code. We have seen that this bound (called sometimes the error-correction bound) on the number of errors is the necessary condition to be sure that there will be a unique output codeword. Thus, to break this barrier, we must use a list-decoding algorithm. In this section we present the algorithm of Madhu Sudan which was the first polynomial time algorithm that can decode beyond the unique error-correction bound.

This algorithm is a generalization of the Welch-Berlekamp algorithm and has been presented in [2]. We will first describe briefly the Sudan algorithm to understand the idea of the algorithm and to have a first bound on the number of errors it can correct and next prove the correctness and the efficiency more formally. We will then express the Welch-Berlekamp algorithm as a special case of the Sudan algorithm.

3.1 Presentation of the algorithm

3.1.1 Overview

We still consider the same problem, the Polynomial Reconstruction Problem (p. 15) but now we are no longer in the case of unambiguous decoding. Thus, the bound on t is no longer $t \geq \frac{n+k}{2}$ but need to be defined. For notational convenience, we write the Polynomial Reconstruction Problem (list-decoding) in a slightly different way.

List Polynomial Reconstruction Problem:
Input: A field $\mathbb{F}_q$; n distincts pairs of elements $\{(x_i, y_i)\}_{i=1}^{n}$ from $\mathbb{F}_q \times \mathbb{F}_q$ and integers k and t.

Output: A list of all functions $f : \mathbb{F}_q \to \mathbb{F}_q$ satisfying:

$$f(x) \text{ is a polynomial of degree at most } k-1 \text{ with } \big|\{i | f(x_i) = y_i\}\big| \geq t. \tag{3.1}$$

Suppose that the message polynomial is a polynomial $f_1(x)$, thus $f_1(x_i) \neq y_i$ for at most $e := n - t$ values of $i \in [n]$. We call the e points (x_i, y_i) at which $f_1(x_i) \neq y_i$ error points. Moreover suppose that $f_2(x), \ldots, f_s(x)$ are the other polynomials satisfying (3.1). Each of these polynomials contains a certain numbers of errors points, i.e., each of them evaluates to y_i at a certain number of points x_i at which $f_1(x_i) \neq y_i$. Finally assume that $H(x, y)$ is a non identically zero polynomial which "contains" the possible remaining errors, i.e., $H(x_i, y_i) = 0$ for the points $\{(x_i, y_i \neq f_1(x_i))\}_{i=1}^{e}$ at which none of the polynomials $f_2(x), \ldots, f_s(x)$ evaluates to y_i. Observe that there always exists such a polynomial $H(x, y)$ (it can be the polynomial 1 if all the errors points are already explained by $f_1(x), \ldots, f_s(x)$). Thus we define

$$Q(x, y) := (y - f_1(x)) \cdot (y - f_2(x)) \cdot \ldots \cdot (y - f_s(x)) \cdot H(x, y). \tag{3.2}$$

By the previous assumptions, note that

$$Q(x_i, y_i) = 0 \ \forall i \in [n].$$

If on top of that $Q(x, y)$ is identically zero in $y = f_1(x), \ldots, f_s(x)$, it suffices to find the y-roots of Q and we will obtain the solution polynomials $f_1(x), \ldots, f_s(x)$ (and possibly others). To summarize, we look for a polynomial $Q(x, y)$ not identically zero which satisfies:

1. $Q(x_i, y_i) = 0 \ \forall i \in [n]$
2. $Q(x, f_j(x)) \equiv 0$ for $j = 1, \ldots, s$.

Observe that there may be several polynomials Q that satisfy these two constraints. This comes from the fact that there are different possible values for the polynomial $H(x, y)$. However any Q that satisfies these constraints can be expressed as (3.2) and $f_1(x), f_2(x), \ldots, f_s(x)$ are y-roots of it. Thus, any non identically zero polynomial Q that satisfies the two constraints is good.

The second condition $Q(x, f_j(x)) \equiv 0$ is satisfied if the degree of the polynomial $p(x) := Q(x, f_j(x))$ is strictly smaller than its number of zeros. $p(x)$ is a polynomial in x and its degree is at most:

$$(\text{max. degree in } x \text{ of } Q(x, y)) \cdot 1 + (\text{max. degree in } y \text{ of } Q(x, y)) \cdot (k - 1) \tag{3.3}$$

since the degree of $f_j(x)$ is at most $k-1$ for $j = 1, 2, \ldots, s$. Let us call the quantity (3.3) l. The polynomial $p(x)$ is zero at points (x_i, y_i) at which $f_j(x_i) = y_i$ and by assumption this quantity is at least t. Thus $p(x) \equiv 0$ if $t > l$.

Thus we look for a polynomial $Q(x, y)$ which has (maximum degree in $x) \cdot 1+$ (maximum degree in $y) \cdot (k-1)$ at most l. As in the Welch-Berlekamp algorithm, this is equivalent to solving a linear system where the variables are the coefficients of Q. How many coefficients q_{ij} does such a polynomial have? Actually this number (say U) is the number of monomials $x^i y^j$ such that $i+(k-1)j \leq l$. We can estimate U as follows. We have $l+1$ possibilities for i. Once i is chosen we have on average $\frac{l}{2(k-1)}$ possibilites for j. Thus $U \approx (l^2 + l)/2(k-1)$. However, we have only n equations (n times the first condition). In order to make sure to have a non-zero solution for $Q(x, y)$ in the homogenous linear system, U must be strictly larger than n, namely

$$U \approx \frac{l^2 + l}{2(k-1)} > n.$$

Solving that we obtain that there exists a non-zero solution to the linear system if

$$l = \sqrt{2(k-1)n}.$$

Since we require $t > l$, all the conditions are satisfied if $t > \sqrt{2(k-1)n}$. Thus, we have found that the algorithm can correct up to $e = n - t < n - \sqrt{2(k-1)n}$ errors. This algorithm is called the Sudan algorithm. The next section proves the formal correctness and efficiency of the Sudan algorithm.

3.1.2 Formal analysis

First we define more fomally the "(maximum degree in x) $\cdot$ 1+(maximum degree in y) $\cdot$ $(k-1)$" that we have used before.

Definition 3.1 (weighted degree). *For weights $w_x, w_y \in \mathbb{Z}^+$, the (w_x, w_y)-weighted degree of a monomial $q_{ij}x^i y^j$ is $iw_x + jw_y$. The (w_x, w_y)-weighted degree of a polynomial $Q(x, y) = \sum_{ij} q_{ij} x^i y^j$ is the maximum, over the monomials with non-zero coefficients, of the (w_x, w_y)-weighted degree of the monomial.*

We can now present the Sudan algorithm, cf. table 3.1. We want to prove that the algorithm of table 3.1 can be run in polynomial time and works correctly. Note that Step 3 can be solved in polynomial time for example with the Roth-Rückenstein algorithm (cf [12], [11]) and we will cover it in the next section. It remains to consider Step 1 and 2. In order to do that, we need to prove several claims. For all these claims, we fix the set of pairs $\{(x_1, y_1), \ldots, (x_n, y_n)\}$.

Sudan Algorithm:

Input: $n, k, t;\ \{(x_1, y_1), \ldots, (x_n, y_n)\}$.
Step 1: Parameter l to be set later.
Step 2: Find *any* function $Q : \mathbb{F}_q^2 \to \mathbb{F}_q$ satisfying

$$\left.\begin{array}{l} Q(x,y) \text{ has } (1, k-1) - \text{weighted degree at most } l, \\ \forall i \in [n],\ Q(x_i, y_i) = 0, \\ Q \text{ is not identically zero.} \end{array}\right\} (3)$$

Step 3: Factor the polynomial Q into irreducible factors.
Output: All the polynomials f such that $(y - f(x))$ is a factor of Q and $f(x_i) = y_i$ for at least t values of i from $[n]$.

Table 3.1: Sudan algorithm

Proposition 3.1. *If a function $Q : \mathbb{F}_q^2 \to \mathbb{F}_q$ satisfying (3) exists, then one can be found in time poly(n).*

Proof: Let $Q(x,y) = \sum_{j=0}^{\lfloor \frac{l}{k-1} \rfloor} \sum_{m=0}^{l-j(k-1)} q_{mj} x^m y^j$. Then we wish to find coefficients q_{mj} satisfying the second constraint of (3), i.e.,

$$\sum_{j=0}^{\lfloor \frac{l}{k-1} \rfloor} \sum_{m=0}^{l-j(k-1)} q_{mj} x_i^m y_i^j = 0,$$

for every $i \in [n]$. This is a linear system in the unknowns q_{mj} and hence if a solution exists then one can be found in polynomial time.

□

Now, we need to find under what conditions this homogenous linear system has a solution. First note that we have n linear constraints (one for each $i \in [n]$). The number of unknowns q_{mj} is the number of monomials of $(1, k-1)-$weighted degree at most l. This number can be estimated in the following way:

$$\sum_{j=0}^{\lfloor \frac{l}{k-1} \rfloor} \sum_{m=0}^{l-j(k-1)} 1 \geq \frac{l(l+2)}{2(k-1)}.$$

Then, we know that there exists a non-zero solution of the system if the number of unknowns is larger than the number of constraints, i.e., if

$$\frac{l(l+2)}{2(k-1)} > n \Rightarrow l^2 + l > 2n(k-1)$$

We obtain that $l = \sqrt{2n(k-1)}$ satisfies this equation. Thus, we have proved the following claim.

Proposition 3.2. *If $l = \sqrt{2n(k-1)}$ then there exists a function $Q(x, y)$ satisfying (3).*

Note that the bound l given in proposition 3.2 is sufficient to establish the results of the Sudan algorithm. However, doing the proofs with more details (rounding issues) will give a finer bound, cf [2].

Notice the difference with the proofs of the Welch-Berlekamp algorithm. Here we have shown that there always exists a solution thanks to the properties of a homogenous linear system but we do not present an explicit solution. For the Welch-Berlekamp algorithm we had proved that a non-zero solution exists by showing that the error-locating polynomial satisfies the conditions.

We now need to make sure that the output list of the Sudan algorithm will give us all the polynomials $f(x)$ such that $f(x_i) = y_i$ for at least t values of i from n.

Proposition 3.3. *If $Q(x, y)$ is a function satisfying (3), $f(x)$ is a function satisfying (3.1) and $t > l$, then $(y - f(x))$ divides $Q(x, y)$.*

Proof: Consider the function $p(x) := Q(x, f(x))$ and observe that p is a polynomial in x. Since the $(1, k-1)-$weighted degree of $Q(x, y)$ is at most l, we have that $k + j(k-1) \leq l$ for any monomial $q_{mj}x^m y^j$. Thus the term $q_{mj}x^m(f(x))^j$ has degree at most l (because f has degree has most $k-1$). Thus $p(x)$ has degree at most l.

To show that $p(x)$ is identically zero, one possibility is to show that $p(x)$ has more zeroes than its degree. We know that for every $i \in [n]$ such that $f(x_i) = y_i$ we have $Q(x_i, y_i) = Q(x_i, f(x_i)) = 0 = p(x_i)$. Since $t > l$, we have that p has more zeros than its degree and hence is identically zero.

If we consider $Q(x, y)$ as a polynomial in y with coefficients in $\mathbb{F}_q[x]$, by the polynomials remainder theorem, we know that if $Q(x, f(x)) \equiv 0$, then $(y - f(x))$ divides $Q(x, y)$.

$\square$

With the two previous propositions, we are now sure that this algorithm is efficient for each $t > l$ and $l = \sqrt{2n(k-1)}$. Thus, this algorithm correctly solves the List Polynomial Reconstruction Problem (p. 22) in polynomial time for $t > \sqrt{2n(k-1)}$. However, as mentioned earlier, this is also important to be interested in the number of polynomials in the output list. The most important thing to observe here is that the number of output polynomials is at most $\lfloor \frac{l}{(k-1)} \rfloor$, since it is the degree of y in $Q(x, y)$. Thus with

$l = \sqrt{2n(k-1)}$, the bound becomes $\lfloor\sqrt{\frac{2n}{k-1}}\rfloor$. Sudan in [2], lemma 6, gives a more precise bound on the number of polynomials f of degree $k-1$ agreeing with t out of n distinct points $\{(x_i, y_i)\}$. We will only explore in the next chapter the question of the size of the output list more in details since the Sudan algorithm is included in the Guruswami-Sudan algorithm.

We have said before that the Sudan algorithm is a generalization of the Welch-Berlekamp algorithm. We will give a formulation of the latter as a special case of the former.

First, note that in the Welch-Berlekamp algorithm we can transform the third constraint of (1) (cf. table 2.1) into

$$\forall i \in [n]\ N(x_i) - y_i E(x_i) = 0.$$

Now we write $Q(x, y) = N(x) - yE(x)$ to have the same notation as in the Sudan algorithm. Thus, (1) becomes:

$$\left.\begin{array}{l} \deg(E) \leq n-t, \\ \deg(N) \leq k-1+n-t, \\ \forall i \in [n], \quad Q(x_i, y_i) = N(x_i) - y_i E(x_i) = 0. \end{array}\right\} (1)'$$

Let $l = n - t + k - 1$. Thus we can make the correspondence between (1)' and (3) in table 3.1. The second constraint of (3) is exactly the third constraint of (1)'. The first constraint of (3) gives us the condition that $Q(x, y) = N(x) - yE(x)$ has $(1, k-1)-$weighted degree at most $n-t+k-1$. This implies that $\deg(E) \leq n-t$ and $\deg(N) \leq k-1+n-t$. Finally, $Q(x, y) \not\equiv 0$ implies that neither $N(x)$ nor $E(x)$ is identically zero.

3.2 Error bound

The Sudan algorithm can correct up to $e = n - t < n - \sqrt{2(k-1)n}$ errors. Thus this algorithm can correct a fraction of errors up to $1 - \sqrt{2(k-1)/n} \approx 1 - \sqrt{2R}$. Comparing this result with the Welch-Berlekamp one, $\frac{1-R}{2}$, we obtain that

$$1 - \sqrt{2R} > \frac{1-R}{2},$$

for $R < 0.1716$. In fact, doing the calculation with the precise bounds, Sudan obtains in [3] that his algorithm is strictly better than the Welch-Berlekamp one for $R < 1/3$. With this algorithm, we achieve a first lower bound for t, the number of agreements. We have $t > \sqrt{2n(k-1)}$.
We will see in the next chapters that this is possible to do much better in terms of the allowed error-rate.

3.3 Implementation

In this section we discuss the implementation of the Sudan algorithm. Implementing Step 2 of table 3.1 is a straightforward process as it amounts to solving a linear system. The only thing that can be slightly tricky is finding the coefficients of the matrix. However, paying attention to the indices gives the right solution.

As mentioned above, Step 3 can be solved with the Roth-Rückenstein algorithm, (cf [11]), which is not to difficult to implement. R.J. McEliece [12] improves it by adding a depth-first search and a stopping rule. It seems that there is an error in this stopping rule. Indeed, this stopping rule implies forgetting plausible solutions in some cases (not very frequent cases, but possible as observed in our simulations). We will briefly present the Roth-Rückenstein before studying its practical implementation and discussing the problem of the stopping rule of McEliece.

Recall that the factorization problem is this: given a polynomial $Q(x,y)$ in $\mathbb{F}_q[x,y]$ and an integer k, find all polynomials $f(x) \in \mathbb{F}_q[x]$ of degree $\leq k-1$ such that $(y-f(x))|Q(x,y)$. If $Q(x,y)$ is a bivariate polynomial such that $x^m|Q(x,y)$, but $x^{m+1} \nmid Q(x,y)$, define

$$\langle\langle Q(x,y)\rangle\rangle = \frac{Q(x,y)}{x^m}.$$

Suppose now that $f(x) = a_0 + a_1 x + \ldots + a_{k-1}x^{k-1}$ is a y-root of $Q(x,y)$. We will see that the coefficients $a_0, a_1, \ldots, a_{k-1}$ can be "picked out" one at a time. The following lemmas and theorems are from [12] and given without proofs.

Lemma 3.4. *If $(y-f(x))|Q(x,y)$, then $y = f(0) = a_0$ is a root of the equation $Q_0(0,y) = 0$, where $Q_0(x,y) = \langle\langle Q(x,y)\rangle\rangle$.*

We now proceed by induction, defining three sequences of polynomials $f_j(x), T_j(x,y)$, and $Q(x,y)$, for $j = 0,1,\ldots,k-1$, as follows. Initially, $f_0 := f(x)$ and $Q_0(x,y) = \langle\langle Q(x,y)\rangle\rangle$. For $j \geq 1$, define

$$f_j(x) := \frac{(f_{j-1}(x) - f_{j-1}(0))}{x} = a_j + \ldots + a_{k-1}x^{k-1-j} \tag{3.4}$$

$$T_j(x,y) := Q_{j-1}(x, xy + a_{j-1}) \tag{3.5}$$

$$Q_j(x,y) := \langle\langle T_j(x,y)\rangle\rangle \tag{3.6}$$

Theorem 3.5. *Given $f(x) = a_0 + \ldots + a_{k-1}x^{k-1} \in \mathbb{F}_q[x]_{<k}$ and $Q(x,y) \in \mathbb{F}_q(x,y)$, define the sequences $f_j(x)$ and $Q_j(x,y)$ as in (3.4) and (3.6). Then for any $j \geq 1$, $(y-f(x))|Q(x,y)$ if and only if $(y-f_j(x))|Q_j(x,y)$.*

Corollary 3.6. *If $(y - f(x))|Q(x, y)$ then $y = a_j$ is a root of the equation*

$$Q_j(0, y) = 0, \quad \textit{for } j = 0, \ldots, k-1.$$

Corollary 3.7. *If $y|Q_k(x, y)$, i.e., if $Q_k(x, 0) = 0$, then $f(x) = a_0 + \ldots + a_{k-1}x^{k-1}$ is a y-root of $Q(x, y)$.*

The paper [11] gives an efficient way to compute $Q(x, xy+a)$ from $Q(x, y)$. In [12] R.J. McEliece proposes to consider a tree structure with a root, the vertex 0 and where each edgesdescending from vertex u corresponds to a root of the equation $Q_u(0, y) = 0$. The terminal vertices are either vertices corresponding to $y-$root of $Q(x, y)$ or vertices whose degree exceeds $k - 1$. The degree of a vertex u, deg$[u]$, gives the numbers of vertices between the root 0 and u. Actually for each vertex u the Roth-Rückenstein adapted by McEliece does the following:

$$\text{IF} \quad Q_u(x, 0) = 0 : \ \textit{root found, output it.} \tag{3.7}$$

$$\text{ELSE IF} \quad \deg[u] < k - 1 : \textit{Compute the solutions of } Q(0, y) = 0 \textit{ and update the parameters.} \tag{3.8}$$

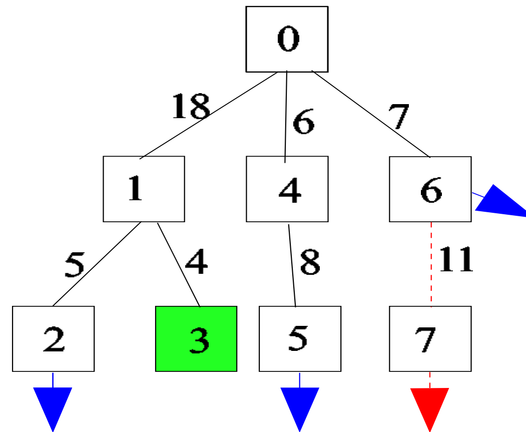

Figure 3.1: Application of the Roth-Rückenstein algorithm.

Figure 3.1 gives an example of the application of the algorithm with $k = 2$. In this tree, $\{18, 6, 7\}$ are the roots of $Q_0(0, y)$. The terminal vertices with a blue arrow give a solution, i.e., $Q_u(x, 0) = 0$ for these vertices. The solution for the vertex $u = 2$ is thus $18+5x$. The green vertex $u = 3$ does not give a solution, this is the case where $Q_3(x, 0) \neq 0$ and $\deg[3] = 1 \not< k-1 = 1$.

We observe that the algorithm stops the exploration of a branch when it finds a solution. However, in some cases, we have that $f(x) = a_0 + \ldots + a_i x^i$ and $g(x) = f(x) + a_{i+1}x^{i+1} + \ldots + a_{k-1}x^{k-1}$ are both solutions. In this case, $g(x)$ will not be found since we will have stopped at the output of $f(x)$. Thus in Figure 3.1, if 7 and $7 + 11x$ are solution, the algorithm will output 7 at vertex 6 and will not find the solution $7 + 11x$ at vertex 7.
As mentioned, this problem of missing a solution rarely happens, for 10000 simulations for a code over $\mathbb{F}_{16}$, it happens 10 times. It is explained by the fact that the stopping rule stops too early. Therefore the solution can be found by transforming the condition (3.7) by the following one:

$$\text{IF} \quad Q_u(x,0) = 0 \text{ AND } \deg[u] = k-1 : \; root\ found,\ output\ it. \quad (3.9)$$

Thus, the algorithm will stop the exploration of a branch only when it reaches a vertex u of degree $= k-1$. Notice that with (3.9) all solutions are found. In the previous example, we thus obtain at vertex 6, that $Q_6(x,0) = 0$ but $\deg[6] = 0 < k-1 = 1$. Then the algorithm goes to (3.8) and finds 11 and 0 as roots corresponding to vertex 7 and 8, cf Figure 3.2. In the next step, $Q_7(x,0) = 0$ and $\deg[7] = 1$, thus the algorithm outputs the solution $7+11x$. It goes next to vertex 8 where $Q_8(x,0) = 0$ and $\deg[8] = 1$ and outputs the solution 7.

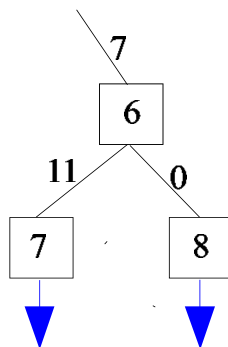

Figure 3.2: Roth-Rückenstein algorithm without stopping rule. Tree of Figure 3.1 from vertex 6

Of course, for a large k, if $f(x) = a_0$ is a solution and if there is no solution of the form $f(x) + g(x)$, $\deg g(x) > 0$, the algorithm will explore unnecessarily some vertices. However it is essential for finding all the solutions.

Chapter 4

The Guruswami-Sudan algorithm

In this section we will present the Guruswami-Sudan algorithm, introduced in [1], which corrects up to $n-1-\lfloor\sqrt{n(k-1)}\rfloor$ errors. Thus, we have a new lower bound for t, the number of agreements, i.e., $t > \sqrt{n(k-1)}$. To justify this algorithm, let us observe a proof of the Johnson bound that uses graph theory. The idea of the Johnson bound is to give an upper bound on the number of polynomials that have a certain property. Here the polynomials in which we are interested are the polynomials that are solution of the List Polynomial Reconstruction Problem, (p. 22).

4.1 The Johnson bound

We model the problem as follows: given a received vector $\mathbf{y}$ and codewords $c_1, \ldots, c_S$ that are the codewords which correspond to the polynomials in the output list (and thus S is the size of the list), we construct a bipartite graph B with vertex set $L \cup R$ and edges being a subset of $L \times R$ as follows:

Left vertices: $L = [n]$, corresponding to the n coordinates of the received vector.
Right vertices: $R = [S]$, corresponding to the S codewords $c_1, \ldots, c_S$.
Edges: There is an edge between $i \in L$ and $j \in R$ if and only if $y_i = (c_j)_i$.

We need now to adapt the properties of codes and polynomials to the graph B. The first thing we know about the code that contains the codewords $c_1, \ldots, c_S$ is that it has minimum distance $d_C = n - k + 1$. It means that two codewords cannot agree on more than $k-1$ indices. In particular this implies that two codewords cannot agree with $\mathbf{y}$ on the same subset of k indices. Thus the graph B does not contain a $K_{k,2}$, the complete bipartite

graph with k vertices on the left and 2 vertices on the right, as a subgraph.

The second thing we know about the polynomials and the codewords in the list is that each codeword has at least t coordinates that agree with $\mathbf{y}$. In the graph B, it means that each vertex on the right is adjacent to at least t vertices on the left. We can throw away some edges in the graph, so that right vertices have degree exactly t. Notice that it does not change the problem and B still does not have any $K_{k,2}$ after this deletion.

We will see that these properties will allow us to give an upper bound on the size of the list, S.

Lemma 4.1. *If a bipartite graph $B = (L, R, E)$, with $|L| = n$ and $|R| = S$, has right degree t and no $K_{k,2}$, then*

$$S \leq \frac{n(t-(k-1))}{t^2-(k-1)n}, \text{ provided } t^2 > n(k-1).$$

Proof: Let v_i denote the degree of the ith vertex in L. Since each vertex on the right has degree t and there are S vertices on the right, the total number of edges is $S \cdot t = \sum_i v_i$.
Let p_i denote the probability, when two distinct codewords c_{k_1} and c_{k_2} are picked uniformly at random from R, that both c_{k_1} and c_{k_2} are adjacent to i. We have

$$p_i = \frac{\begin{pmatrix} v_i \\ 2 \end{pmatrix}}{\begin{pmatrix} S \\ 2 \end{pmatrix}} = \frac{v_i^2 - v_i}{S^2 - S}.$$

Now consider the expected number of common neighbours that distinct codewords c_{k_1} and c_{k_2} have. This quantity is given by

$$\sum_i p_i = \frac{1}{S^2-S} \sum_{i=1}^{n} (v_i^2 - v_i).$$

Here we can use the property that two codewords agree in at most $k-1$ coordinates and thus we get

$$\frac{1}{S^2-S} \sum_{i=1}^{n} (v_i^2 - v_i) \leq k-1.$$

We now use the Cauchy-Schwartz inequality and the inequality above becomes:

$$\frac{1}{n} \left(\sum_{i=1}^{n} v_i \right)^2 - \sum_i v_i \leq (k-1)(S^2 - S).$$

$$\Leftrightarrow \frac{S^2t^2}{n} - St \leq (k-1)(S^2 - S).$$

$$\Leftrightarrow S\left(\frac{t^2}{n} - (k-1)\right) \leq t - (k-1).$$

Thus we have

$$S \leq \frac{n(t-(k-1))}{t^2-(k-1)n} \text{ provided } t^2 > n(k-1).$$

□

Then we obtain the following theorem.

Theorem 4.2. *Given points* $\{(x_i, y_i)\}_{i=1}^n$ *there are at most* $\frac{n(t-(k-1))}{t^2-(k-1)n}$ *polynomials* f *of degree at most* $k-1$ *that satisfy* $|\{i \in [n] | f(x_i) = y_i\}| \geq t$ *if* $t > \sqrt{n(k-1)}$.

We would like to design an algorithm that can match the above theorem, i.e., an algorithm that can find the (at most) $\frac{n(t-(k-1))}{t^2-(k-1)n}$ polynomials of degree at most $k-1$ that satisfy $|\{i \in [n] | f(x_i) = y_i\}| \geq t$ for $t > \sqrt{n(k-1)}$. The Guruswami-Sudan is such an algorithm and thus can correct up to $e < n - \sqrt{n(k-1)}$ errors.

4.2 Presentation of the algorithm

The Sudan and Guruswami-Sudan algorithms can be used to solve the problem of list-decoding for Generalized Reed-Solomon codes.

Before explaining the underlying ideas of the Guruswami-Sudan algorithm, let us consider briefly the List Polynomial Reconstruction Problem for Generalized Reed-Solomon codes. As done before for Reed-Solomon codes, let us make the correspondence between the List Polynomial Reconstruction Problem (p. 22) and the Generalized Reed-Solomon definition. It is obvious that the x_i's are the α_i's. If $\mathbf{r}$ is the received vector, the y_i's are not only the r_i, but we have $y_i = r_i/v_i$, as easily checked.

The Guruswami-Sudan algorithm is a generalization of the Sudan algorithm. Thus, we will first recall the ideas of the Sudan algorithm. In the first phase the algorithm finds a polynomial $Q(x, y)$ of $(1, k-1)$-weighted degree at most l such that $Q(x_i, y_i) = 0$ for all $\{(x_i, y_i)\}_{i=1}^n$. Then in a second phase it finds all small degree roots of Q, i.e., finds all polynomials f of degree at most $k-1$ such that $Q(x, f(x)) \equiv 0$ or equivalently $(y - f(x))$ is a factor of $Q(x, y)$; and these polynomials f form candidates for the output. The two important assertions for success are that

1. Q must have a sufficiently large degree in comparison with the number of constraints given by $Q(x_i, y_i) = 0$.

2. If Q and f have low degree in comparison to the number of points at which $y_i = f(x_i)$ then the polynomial $Q(x, f(x))$ will be identically zero and $(y - f(x))$ will be a factor of Q.

Proposition 3.2 tells us that the condition $l = \sqrt{2n(k-1)}$ will ensure that the first assertion is satisfied and proposition 3.3 says that the condition $t > l$ will guarantee that the second assertion is satisfied. These two constraints are somewhat "opposed". The Guruswami-Sudan algorithm can reduce this contradiction. The plan of this algorithm is the same as all the others. We will find a polynomial Q of low weighted degree such that at the point (x_i, y_i) the polynomial Q has a singularity of multiplicity r and not only 1 for all $i \in [n]$. Informally, a singularity is a point where the curve given by $Q(x, y) = 0$ intersects itself.

Traditionally, one uses the partial derivatives of Q to define singularities. But here we will work over fields with positive characteristic and the derivatives over these fields are not well-behaved. Intuitively it is clear what it means for a curve to pass through the origin once: the constant term has zero coefficient. In the same way, if a curve passes through the origin twice it means that the constant term as well as the monomials of degree one (i.e., x and y) have zero as coefficients. We can guess from these cases that a curve Q passes through the origin at least r times if the monomials of total degree less than r have a zero coefficient. We need to extend this idea to a arbitrary point (x_i, y_i). To this end, we can shift the coordinate system such that the point (x_i, y_i) is the origin. In the shifted system, we require that the monomials of Q with degree less than r have zero coefficients.

In the first phase, the conditions that each point (x_i, y_i) is a singularity of multiplicity r will give us more constraints and so the degree of Q will become larger. However in the second phase we will have a big advantage. Indeed, in the second phase, we are interested in the points (x_i, y_i) such that $y_i = f(x_i)$, because at these points we have $Q(x_i, y_i) = Q(x_i, f(x_i)) = 0$. The number of points such that $y_i = f(x_i)$ is still at least t, as for the Sudan algorithm, but now at each of these t points we have a zero of multiplicity r, and so $Q(x, f(x))$ has a zero of multiplicity r for each point (x_i, y_i) such that $y_i = f(x_i)$. Hence, we need only $1/r$th as many singularities as regular points and this is where the advantage comes from. r will be a parameter in the Guruswami-Sudan algorithm, we will choose it sufficiently large to handle as many errors as feasible but we have to be careful because the running time increases with r.

4.2.1 Overview

We will first make an overview of this algorithm to have a general idea of its error-correcting capacity. The considered problem is still the List Polynomial Reconstruction Problem (p.22). The starting ideas are the same as the ones for the Sudan algorithm, cf section 3.1.1. We assume that the message polynomial is a polynomial $f_1(x)$ and the polynomials $f_2(x), \dots, f_s(x)$ are the other polynomials satisfying (3.1). Each polynomial contains a certain number of error points (i.e., evaluates to y_i at points x_i with $f_1(x_i) \neq y_i$). Finally, we assume that $H(x, y)$ is a polynomial that contains the remaining error points. There always exists such a polynomial. We can now define $Q(x, y)$ as in (3.2) and observe that by the previous assumptions

$$Q(x_i, y_i) = 0 \ \forall i \in [n].$$

Actually, as explained above, we require that $Q(x, y)$ has a zero of multiplicity r at each point $\{(x_i, y_i)\}_{i=1}^{n}$. Still following the same scheme as in section 3.1.1, we would like that $Q(x, y)$ be identically zero in $y = f_1(x), \dots, f_s(x)$. Then finding the $y-$roots of $Q(x, y)$ will give the sought polynomials. Thus we look for a polynomial $Q(x, y)$ not identically zero which satisfies:

1. $Q(x, y)$ has a zero of multiplicity r at each point $\{(x_i, y_i)\}_{i=1}^{n}$.
2. $Q(x, f_j(x)) \equiv 0$ for $j = 1, 2, \dots, s$

Observe that each Q that satisfies these conditions, also satisfies (3.2) and so is good for our purpose. The second condition $Q(x, f_j(x)) \equiv 0$ is clearly satisfied if the polynomial $p(x) := Q(x, f_j(x))$ is divisible by a polynomial of degree larger than its degree. $p(x)$ is a polynomial in x and has degree at most

$$(\text{max. degree in } x \text{ of } Q(x, y)) \cdot 1 + (\text{max. degree in } y \text{ of } Q(x, y)) \cdot (k - 1) \tag{4.1}$$

since the degree of $f_j(x)$ is at most $k - 1$ for $j = 1, 2, \dots, s$. Let us call this quantity l. We will prove in the next section that $p(x)$ is divisible by a polynomial of degree rt. Thus $p(x) \equiv 0$ if $rt > l$.

Thus we look for a polynomial $Q(x, y)$ which has (maximum degree in x)$\cdot 1+$ (maximum degree in y)$\cdot (k-1)$ at most l. The problem is exactly the same as in the Sudan algorithm (section 3.1.1). The number of coefficients of Q is then the number of monomials $x^i y^j$ such that $i + (k - 1)j \leq l$. We have estimated U previously and we have obtained $U \approx (l^2 + l)/2(k - 1)$.

The difference with the Sudan algorithm comes in the number of constraints. For each point (x_i, y_i), we require that the coefficients of monomials of degree less than r of the polynomial Q, in the shifted system such that

(x_i, y_i) is the origin, have value zero. Assuming that the shift can be done in polynomial time and is a linear transformation (which will be proven in the next section), we only need to count the number of monomials of degree less than r. This number is the number of monomials $x^i y^j$ such that $i + j < r$. This quantity (say V) can be estimated as follows. We have r possibilites for i. Once i is chosen it remains on average $r/2$ possibilities for j. Thus $V \approx r^2/2$. Since we have n points, the total number of constraints is $nV = nr^2/2$. In order to have a nontrivial solution to the homogenous linear system, we must have more unknowns (coefficients of Q) than constraints (given by $Q(x_i, y_i) = 0$ r times), namely

$$U \approx \frac{l^2 + l}{2(k-1)} > nV \approx \frac{nr^2}{2}.$$

Thus, we obtain that there exists a solution Q to the linear system if $l = r\sqrt{(k-1)n}$. Since we require also $rt > l$, we finally obtain that the algorithm solves the List Polynomial Reconstruction Problem if $t > \sqrt{(k-1)n}$. Thus we obtain a constraint on $e = n - t$, the number of errors that the Guruswami-Sudan algorithm can correct: $e < n - \sqrt{(k-1)n}$. The next section proves the formal correctness and efficiency of the Guruswami-Sudan algorithm.

4.2.2 Formal analysis

We first give an exact definition of what it means for a curve to "pass through a point r times".

Definition 4.1. *A polynomial $Q(x, y)$ passes through the point (α, β) at least r times if all the coefficients of total degree less than r of the polynomial $Q_{\alpha,\beta}(x, y) := Q(x + \alpha, y + \beta)$ are zero. The polynomial $Q_{\alpha,\beta}(x, y)$ is called a shifted polynomial.*

We can describe the coefficients of the shifted polynomial with linear transformation of the coefficients of the polynomial $Q(x, y)$. The relation is given by

$$(q_{\alpha,\beta})_{ij} = \sum_{a \geq i} \sum_{b \geq j} \binom{a}{i} \binom{b}{j} q_{ab} \alpha^{a-i} \beta^{b-j}. \tag{4.2}$$

We can now describe the Guruswami-Sudan algorithm for the List Polynomial Reconstruction Problem. The steps are presented in table 4.1

We now prove the correctness of the algorithm and that it can be implemented to run in polynomial time. We need also to set the parameters r and l. The assertions we need to prove are very similar to the ones for the previous algorithms. First, we need to prove that there exists a polynomial Q as sought in Step 1.

Guruswami-Sudan algorithm:

Inputs: $n, k, t, \{(x_i, y_i)\}_{i=1}^{n}$, where $x_i, y_i \in \mathbb{F}_q$.
Step 0: Parameters r, l to be set later.
Step 1: Find a polynomial $Q(x, y)$ such that the $(1, k-1)-$ weighted degree of Q is $\leq l$ and such that the following conditions hold:

1. Q is not identically zero.
2. For every $i \in [n]$, (x_i, y_i) is a singularity of multiplicity r of Q. Let $Q^{(i)}$ be the shifted polynomial in (x_i, y_i). Using the above definition of singularity and shifted polynomial, we obtain more specifically:

$$\left.\begin{array}{c} \forall i \in [n], \forall j_1, j_2 \geq 0, \text{ s.t. } j_1 + j_2 < r \\ (q_{j_1 j_2})^{(i)} := \sum_{a \geq j_1} \sum_{b \geq j_2} \binom{a}{j_1} \binom{b}{j_2} q_{ab} x_i^{a-j_1} y_i^{b-j_2} = 0. \end{array}\right\} \quad (4)$$

Step 2: Find all polynomials $f \in \mathbb{F}_q[x]$ of degree $\leq k-1$ such that $(y - f(x))$ is a factor of $Q(x, y)$. For each polynomial f check if $f(x_i) = y_i$ for at least t values of $i \in [n]$, and if so, include f in output list.
Output: f according to Step 2.

Table 4.1: Guruswami-Sudan algorithm

We have already noticed that the constraints are linear constraints in the coefficients of Q. Thus we have a homogenous linear system and if there is a nontrivial solution, we can find it in polynomial time. In order to make sure that there will be a nontrivial solution, we need some conditions. First, observe that the number of constraints in (4) in table 4.1 is

$$\sum_{i=0}^{r-1} (r - i) = r^2 - \frac{r^2 - r}{2} = \binom{r+1}{2}$$

for every $i \in [n]$, thus the total number of constraints is

$$n \binom{r+1}{2}.$$

The number of unknowns, i.e., the number of monomials of $(1, k-1)-$weighted degree at most l, can be computed precisely as follows:

$$\sum_{j_2=0}^{\lfloor \frac{l}{k-1} \rfloor} \sum_{j_1=0}^{l-(k-1)j_2} 1 = \sum_{j_2=0}^{\lfloor \frac{l}{k-1} \rfloor} (l + 1 - (k-1)j_2)$$

$$= (l+1)\left(\left\lfloor \frac{l}{k-1} \right\rfloor + 1\right) - \frac{(k-1)}{2} \left\lfloor \frac{l}{k-1} \right\rfloor \left(\left\lfloor \frac{l}{k-1} \right\rfloor + 1\right).$$

This quantity can be lower bounded by

$$\left(\left\lfloor\frac{l}{k-1}\right\rfloor+1\right)\left(l+1-\frac{l}{2}\right)$$
$$\geq \frac{l}{k-1}\cdot\frac{l+2}{2}.$$

If the number of constraints is strictly smaller than the number of unknowns, the linear system has a non-zero solution. Thus if

$$\frac{l}{k-1}\cdot\frac{l+2}{2} > n\binom{r+1}{2}, \tag{4.3}$$

then there exists a polynomial Q satisfying the two properties in Step 1. We have a first constraint on the parameters r and l.

In Step 2, we would like to find all the polynomials f that satisfy (3.1). In order to find under what conditions such f are roots of Q, we need to prove the following proposition. In this proposition Q can be any polynomial returned in Step 1 of the algorithm.

Proposition 4.3. *If (x_i, y_i) is an input point and f is any polynomial such that $y_i = f(x_i)$, then $(x - x_i)^r$ divides $p(x) := Q(x, f(x))$.*

Proof: Let $f'(x)$ be the polynomial given by $f'(x) = f(x + x_i) - y_i$. Notice that $f'(0) = f(x_i) - y_i = 0$. Hence $f'(x) = xf''(x)$, for some polynomial $f''(x)$. Now, consider $p'(x) := Q^{(i)}(x, f'(x))$. We first argue that $p'(x - x_i) = p(x)$. To see this, observe that

$$p(x) = Q(x, f(x)) = Q^{(i)}(x-x_i, f(x)-y_i) = Q^{(i)}(x-x_i, f'(x-x_i)) = f'(x-x_i).$$

Now, by construction, $Q^{(i)}$ has no coefficients of total degree less than r. Thus by substituting $y = xf''(x)$ for y, we are left with a polynomial g' such that x^r divides $p'(x)$. Shifting back we have $(x-x_i)^r$ divides $p'(x-x_i) = p(x)$.

□

Consider now the polynomial $p(x) := Q(x, f(x))$, where $f(x)$ is a polynomial satisfying (3.1). This polynomial is a polynomial in x and has degree at most l by the definition of weighted degree and properties of Q. By the above proposition, for every i such that $y_i = f(x_i)$, $(x - x_i)^r$ divides $p(x)$. Since $f(x)$ satisfies (3.1), there are at least t values of $i \in [n]$ such that $f(x_i) = y_i$. Thus if S is the set of i such that $y_i = f(x_i)$, then $\prod_{i\in S}(x - x_i)^r$ divides $p(x)$ and by our assumption $|S| \geq t$. Hence, we have a polynomial of degree at least rt that divides a polynomial of degree at most l. If

$$rt > l \tag{4.4}$$

we have that p is identically zero and then $(y - f(x))$ divides Q. We have proved the following proposition and we have a second constraint on r and l.

Proposition 4.4. *If $f(x)$ is a polynomial of degree at most $k-1$ such that $y_i = f(x_i)$ for at least t values of $i \in [n]$ and $rt > l$, then $(y - f(x))$ divides Q.*

We have now two constraints on the parameters r and l, (4.3) and (4.4). Guruswami and Sudan in [1] prove the following proposition that give values for r and t.

Proposition 4.5. *If n, t, k satisfy $t^2 > (k-1)n$, then for the settings*

$$r := 1 + \left\lfloor \frac{(k-1)n + \sqrt{(k-1)^2n^2 + 4(t^2 - (k-1)n)}}{2(t^2 - (k-1)n)} \right\rfloor$$

and

$$l = rt - 1,$$

$n\begin{pmatrix} r+1 \\ 2 \end{pmatrix} < \frac{l(l+2)}{2(k-1)}$ *and $rt > l$ both hold.*

Proof: If we set $l := rt - 1$ in the algorithm, $rt > l$ certainly holds. Using $l = rt - 1$, we now need to satisfy the constraint

$$n\begin{pmatrix} r+1 \\ 2 \end{pmatrix} < \frac{(rt-1)(rt+1)}{2(k-1)}$$

which simplifies to $r^2t^2 - 1 > (k-1)n(r^2 + r)$ or equivalently,

$$r^2(t^2 - (k-1)n) - (k-1)nr - 1 > 0.$$

Hence it suffices to pick r to be an integer greater than the larger root of the above quadratic, and therefore picking

$$r := 1 + \left\lfloor \frac{(k-1)n + \sqrt{(k-1)^2n^2 + 4(t^2 - (k-1)n)}}{2(t^2 - (k-1)n)} \right\rfloor$$

suffices.

□

With the previous propositions we have proved the following theorem:

Theorem 4.6. *The Guruswami-Sudan algorithm with inputs n, k, t and the points $\{(x_i, y_i) : 1 \le i \le n\}$, correctly solves the List Polynomial Reconstruction Problem in polynomial time provided $t > \sqrt{(k-1)n}$.*

4.3 Error bound

Finally, observe that the Guruswami-Sudan algorithm can correct a fraction of error up to $\frac{e}{n} = 1 - \sqrt{\frac{k-1}{n}} \approx 1 - \sqrt{R}$, where R is the rate of the code. Since $1 - \sqrt{R} \geq \frac{1-R}{2}$ for all $R \leq 1$, the Guruswami-Sudan algorithm can always correct as many or more errors than the Welch-Berlekamp algorithm. It is clear that the Guruswami-Sudan algorithm is also at least as good as the Sudan algorithm and often better since $1 - \sqrt{R} \geq 1 - \sqrt{2R}$ for $R \in [0, 1]$.

4.4 Complexity analysis and implementation

We will now be interested in the complexity of this algorithm and in its practical implementation. Observe that in Step 1 of table 4.1, we have to build a matrix and to find its kernel. As explained previously the number of constraints is n times the number of monomials with total degree less than the multiplicity r. Thus there are $nr(r+1)/2 = O(nr^2/2)$ constraints. The number of variables is the number of coefficients of the polynomial Q. Q has about $l(l+2)/2(k-1) = O(l^2/2(k-1))$ coefficients. Thus, the size of the matrix in Step 1 is

$$O(l^2/2(k-1)) \times O(nr^2/2).$$

Setting $l := rt - 1$ and $t = \sqrt{n(k-1)}$ as suggested in proposition 4.5, we obtain the size of the matrix:

$$O(r^2 n/2) \times O(r^2 n/2).$$

To solve the linear system, we can use simple Gaussian elimination and the complexity will be $O(r^6 n^3)$. Notice that with the algorithm due to Koetter [9], the complexity is only $O(r^4 n^2)$. This algorithm does not inverse the matrix to find the polynomial Q but uses a slightly different way. In our implementation, we prefer to use available Gaussian elimination. However, it is important to observe here that the complexity of the Guruswami-Sudan algorithm is at least (according to what we know today) $O(r^4 n^2)$. In Step 2 of table 4.1, the complexity of the Roth-Rückenstein algorithm is negligible compared with Step 1.

In the above presentation of this algorithm (cf Section 4.2), we first choose the parameters n, k and t, then we set l and r according to proposition 4.5. With these settings we are sure that there will exist a solution since the two constraints (4.3) and (4.4) are satisfied. However, these two constraints were obtained with some slight simplifications and then the r given in proposition 4.5 is not the smallest possible r for a given t. It means that there exist smaller values of r such that the algorithm is still efficient. Since the complexity of the algorithm depends on r, it is important to have the smallest

possible r. For this reason, we will look at the Guruswami-Sudan algorithm in a slightly different way.

Instead of choosing t and setting r and l as functions of t as in proposition 4.5, we will choose r, the multiplicity, then find the smallest l which satsifies the following equation

$$(l+1)\left(\left\lfloor\frac{l}{k-1}\right\rfloor+1\right)-\frac{k-1}{2}\left\lfloor\frac{l}{k-1}\right\rfloor\left(\left\lfloor\frac{l}{k-1}\right\rfloor+1\right)>n\frac{(r+1)r}{2}. \quad (4.5)$$

and finally find the smallest t (which is the same as the largest e) such that

$$rt > l. \quad (4.6)$$

Observe that we keep the floor functions in the equation (4.5) to obtain precise solutions. Our method is as follows. For given n and k we start with $r = 1$, then we compute l according to (4.5) and $e = n - t$ according to (4.6). We continue increasing r until we reach the r (we call it r_{theo}) such that the algorithm can correct the theoretical maximum number of errors $e_{theo} = n - 1 - \lfloor\sqrt{(k-1)n}\rfloor$. Since the complexity of the algorithm is in $O(n^2r^4)$, it will not be efficient if r (and of course n) are too large. With this method we can observe when r becomes large and so when the size of the matrix is too large to allow an inversion in reasonable time (in less than one hour for example on a computer with 1.5 Ghz and 512 Mo RAM). Observe that for very large matrix (>1'000'000 × >1'000'000) there are also memory problems and the matrix cannot even be constructed.

4.4.1 Results analysis

Observe the following arrays for a better understanding. First, we take the Reed-Solomon code with $[n, k, d_C] = [16, 4, 13]$ over $\mathbb{F}_{16}$. For this code $e_{theo} = 9$. We obtain

Multiplicity	Errors	Size of the matrix
0	6	17×16
1	7	18×16
2	8	51×48
28	9	6501×6496

Table 4.2: $[16, 4, 13]$ Reed-Solomon code over $\mathbb{F}_{16}$

The first column gives the multiplicity r and the second the number of errors that the Guruwsami-Sudan algorithm can correct with this multiplicity. We give only the multiplicities that improve the previous number of errors. Thus for this code the multiplicities $r = 3$ until $r = 27$ does not

allow the algorithm to correct more than 8 errors. The last column gives the size of the matrix. Finally, the first line with $r = 0$ gives the error-correction bound, i.e., $e_{WB} = \lfloor \frac{d_C}{2} \rfloor$ the number of errors we can correct with the Welch-Berlekamp algorithm. For this line the size of the matrix is not $O(r^2 n/2) \times O(r^2 n/2)$ but $O(n) \times O(n)$.

In the array 4.2, for $e_{theo} = 9$ we see that we have to invert a matrix of size 6000×6000. For practical implementation it is not reasonable. Thus we can say that $e = 8$ is the practical limit for an implementation. We call this number of errors e_{prac} and the associated r: r_{prac}. Consider now the $[255, 144, 112]$ Reed-Solomon code over $\mathbb{F}_{256}$. For this code $e_{theo} = 64$.

Multiplicity	Errors	Size of the matrix
0	56	256×255
3	57	1530×1535
4	59	2550×2553
6	60	5355×5364
7	61	7140×7145
12	62	19890×19907
23	63	70380×70401
573	64	41935005×41935053

Table 4.3: $[255, 144, 112]$ Reed-Solomon code over $\mathbb{F}_{256}$

We observe in table 4.3 that for multiplicities $r = 1, 2$ the Guruswami-Sudan algorithm cannot correct more that 56 errors. With multiplicity $r = 3$ the algorithm begins to be better than the Welch-Berlekamp algorithm. Depending on the algorithm we use for solving the linear system, the practical limit e_{prac} will be 60 or 61 in the array 4.3, but it is clear that it seems pointless to try to correct $e_{theo} = 64$ errors since we have to deal with a matrix of size $42'000'000 \times 42'000'000$.

It seems difficult to set a precise "efficient" bound for r, i.e., the precise r for given n and k at which the size of the matrix will be too large. However, if we observe the results above, we note that there is often a large distance between the r_{theo} (resp. 28 and 573 in the arrays 4.2 and 4.3) which can correct exactly e_{theo} errors and the other r and thus, a simple logical selection of r will give an efficient algorithm.

Sometimes the Guruswami-Sudan algorithm does not bring any useful improvements compared with the Welch-Berlekamp algorithm. This happens for example with the $[255, 223, 33]$ Reed-Solomon code over $\mathbb{F}_{256}$ for which $e_{theo} = 17$. The array 4.4 shows the results. Since it implies to invert a very big matrix to correct $e_{theo} = 17$ errors, an implementation of the Guruswami-

Multiplicity	Errors	Size of the matrix
0	16	256×255
112	17	1613640×1613656

Table 4.4: $[255, 223, 33]$ Reed-Solmon code over $\mathbb{F}_{256}$

Sudan algorithm can actually correct only 16 errors in a reasonable time. This is not better than the $e_{WB} = \lfloor 33/2 \rfloor = 16$ errors that the Welch-Berlekamp algorithm can correct.

It would be interesting to have similar results for more than three codes, for example for all Reed-Solomon codes over $\mathbb{F}_{256}$. We continue with the same method as above: for given n and k we increase r from 1 to r_{theo} and compute for each r between these two bounds the corresponding number of errors that the Guruswami-Sudan can correct. Instead of presenting the results in arrays which give each r and the corresponding error-correcting capacity, we will give the results in a graph (Figure 4.1). We choose an numbers of errors $e_j = e_{theo} - j$, $j = 1, \ldots, 5$ and draw for each code the required multiplicity to correct e_j errors with the Guruswami-Sudan algorithm. The case $e_0 = e_{theo} - 0 = e_{theo}$ is not considered since there are too large differences between the r's and the graph becomes unreadable.

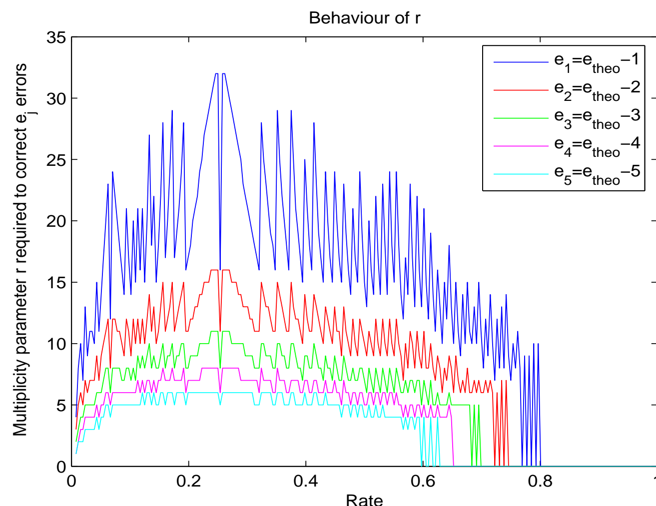

Figure 4.1: Multiplicity r required to correct $e_j = e_{theo} - j$ errors for $j = 1, \ldots, 5$ against the rate for the codes over $\mathbb{F}_{256}$

We can observe several things in Figure 4.1 (numerical results are given in Appendix A). Each curve corresponds to a different $e_j = e_{theo} - j$. As observed in the arrays before, for a given rate R, the multiplicity parame-

ter r increases slowly during a time and suddenly becomes much larger. It confirms our observation that r_{theo} is often much larger than the previous r's.

We notice that the curves are not very regular, there are some jumps. Observe for example the large jump at rate $R \sim 0.25$ for the curve $e_1 = e_{theo} - 1$. Indeed the jump is between the $[256, 64, 193]$ code (we call it C_1) for which $r = 32$ and the $[256, 65, 192]$ code (C_2) for which $r = 16$ and the $[256, 66, 191]$ code (C_3) for which $r = 32$. Tables 4.5, 4.6 and 4.7 give the detailed values for these three codes. The last column gives the degree l of Q, i.e., the smallest l that satisfies (4.5).

Multiplicity	Errors	Degree l of Q
0	96	$--$
1	107	148
2	116	279
3	119	408
4	121	536
5	123	664
6	124	791
8	125	1046
11	126	1427
16	127	2063
32	128	4095

Table 4.5: $C_1 = [255, 64, 193]$ Reed-Solomon code over $\mathbb{F}_{256}$

Multiplicity	Errors	Degree l of Q
0	96	$--$
1	106	149
2	115	281
3	118	411
4	120	540
5	122	669
6	123	797
8	124	1054
11	125	1438
16	126	2079

Table 4.6: $C_2 = [255, 65, 192]$ Reed-Solomon code over $\mathbb{F}_{256}$

For given n and r, if we compute l according to equation (4.5) with k and $k+1$, the solutions are very close. However a slight difference between the two l's suffices to give two different t's. Take for example the two codes C_2

Multiplicity	Errors	Degree l of Q
0	95	$--$
1	105	150
2	114	283
3	117	414
4	119	544
5	121	674
6	122	803
8	123	1062
11	124	1449
16	125	2095
32	126	4159

Table 4.7: $C_3 = [255, 66, 191]$ Reed-Solomon code over $\mathbb{F}_{256}$

and C_3. For a given multiplicity $r = 2$ the degree l will be respectively 281 and 283. The difference is small but it suffices to obtain 115 errors corrected for C_2 and only 114 for C_3. There are now three possible cases.

The first one is when the two codes have the same e_{theo}. This is precisely the case for the two codes C_2 and C_3 for which $e_{theo} = 127$. Since C_2 "gains" one error at each multiplicity, i.e., for a given multiplicity C_2 corrects one error more than C_3, C_3 needs an additional multiplicity to correct $e_{theo} - 1$ errors (here C_3 needs $r = 32$) and the jump comes from that.

The second possible case is when the code that "loses" i errors at each given multiplicity has a e_{theo} $h \neq i$ units smaller than the other code. This is the case for C_1 and C_2. C_2 "loses" $i = 1$ error at each multiplicity but C_2 has a e_{theo} $h = 2 \neq 1$ units smaller than the e_{theo} of C_1. Thus, in fact, for a given e_j such that $e_j = e_{theo} - j$ C_2 "gains" 1 error. Thus the jump comes from that.

Finally if the code that "loses" i errors at each given multiplicity has a e_{theo} i units smaller that the other code, there is a compensation and no jump. This is the case for C_1 and C_3. C_3 "loses" 2 errors at each multiplicity but C_3 has a $e_{theo} = 126$ which is 2 units smaller than the e_{theo} of C_1.

Is is also interesting to observe the perfomance consistency of a given code over the others. In other words, if for a given $e_j = e_{theo} - j$ a code with rate R_1 needs a multiplicity r_1 larger than r_2, the multiplicity for the code with rate R_2, it will be also the case for another $e_i \neq e_j$. Actually, examining the numerical results more precisely, it appears that it is generally the case and when it is not, r_1 and r_2 are very close.

The third interesting thing (that we cannot observe in Figure 4.1) is the smaller r such that all Reed-Solomon codes over $\mathbb{F}_{256}$ with $n = 256$ can correct up to e_{theo} errors (or in other word the larger r_{theo}). This $r = 8255$ and it is obtained for the code $[256, 66, 191]$. The rate of this code is 0.2578 and we will actually show next that the worst codes arise for $R \sim 0.25$.

Notice that to obtain figure 4.1 we have applied our method for each code over $\mathbb{F}_{256}$. It is very long, since we have to try every r until we reach r_{theo}. To avoid that, it would be comfortable to have an expression of r as a function of e, n and k. However observe that without knowing k and n we cannot find an expression for l from the equation (4.5). However if we simplify (4.5) slightly by removing the floor function we obtain that

$$l > -1-\frac{1}{2}(k-1)+\frac{1}{2}\sqrt{(4-4(k-1)+(k-1)^2+4r^2n(k-1)+4rn(k-1))}.$$

With the constraint $rt > l$ we obtain that

$$t = n - e > \frac{-1-\frac{1}{2}(k-1)+\frac{1}{2}\sqrt{(4-4(k-1)+(k-1)^2+4r^2n(k-1)+4rn(k-1))}}{r}.$$

Assuming that

$$n - e - 1 \geq \frac{-1-\frac{1}{2}(k-1)+\frac{1}{2}\sqrt{(4-4(k-1)+(k-1)^2+4r^2n(k-1)+4rn(k-1))}}{r}$$

and solving this equation for r we obtain a (complicated) formula for r (cf Appendix A). This allows us to obtain values for r for a given number of errors and in particular for $e_{theo} = n - 1 - \lfloor\sqrt{(k-1)n}\rfloor$. This method has the advantage to take less time than solving (4.5) for given n and k and for each r until r_{theo}. However due to the simplifications the values are not always exact (although very close) and this why we do not use this method.

As explained, computation quickly takes more time if we increase the number of considered rates. However we can do it in a slightly different way. Instead of trying every r until we reach $e = e_{theo}$, we can fix a r_{prac}, i.e., we assume that the algorithm is no longer practical if $r > r_{prac}$, and observe how many errors each code can correct with this r_{prac}, solving equation (4.5) and (4.6). The advantage here is that we have to compute one l and one t for each rate when we had to compute l and t for each rate *and* each r before. Of course, the results before were finer since we could observe the behaviour of e in function of r. However, we save running time here.

Figure 4.2 shows the behaviour of the difference $e_{theo} - e_{prac}$ as a function of

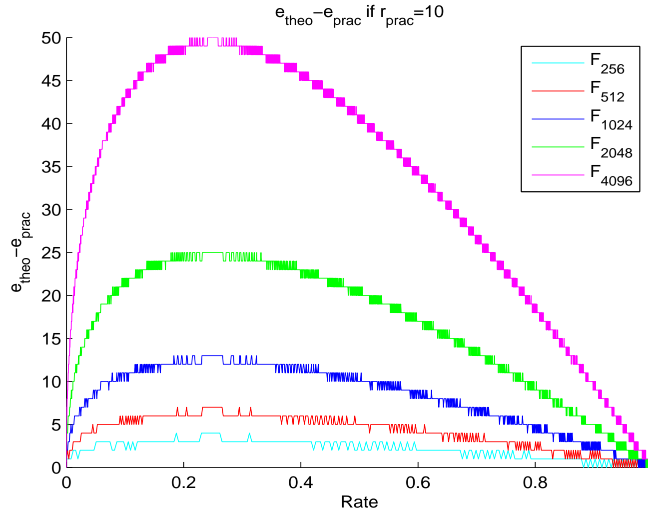

Figure 4.2: Difference between e_{theo} and e_{prac} assuming that the Guruswami-Sudan algorithm is no longer practical if $r > 10$

the rate for $r_{prac} = 10$. The curves correspond to the Reed-Solomon codes over (resp. from the top) $\mathbb{F}_{4096}, \mathbb{F}_{2048}, \mathbb{F}_{1024}, \mathbb{F}_{512}, \mathbb{F}_{256}$. As expected, all the curves have the same shape. If we represented the difference of the fraction of errors rather than the difference of errors, the curves would superimpose. Observe now the same figure but for $r_{prac} = 4$, cf Figure 4.3. As expected, the curves still show the same behaviour. Note that the maximum difference between the theoretical number of errors and the chosen practical one comes to 118 for 84 codes over $\mathbb{F}_{4096}$. These codes have a rate between 0.21973 and 0.2661.

Examining Figures 4.2 and 4.3, the rates that have the maximum difference seem to be between 0.2 and 0.4. Mathematically, given r, n and l (computed with equation (4.5) without the floor functions), if we look for the k that maximizes the difference, we obtain that $R \sim 0.25$ which is in conformity with our observations in Figures 4.1, 4.2 and 4.3. Of course, there are several R that have the same difference and thus this result is only indicative.

The goal of this section was to demonstrate the fact that the multiplicity r of the points in the Guruswami-Sudan algorithm has to be bounded to avoid memory and time problems for practical implementation. Indeed, even though the running time is always polynomial, it quickly gets too large for practical purposes. So we often have $e_{theo} \neq e_{prac}$ where e_{prac} is determined by the available computing resources.

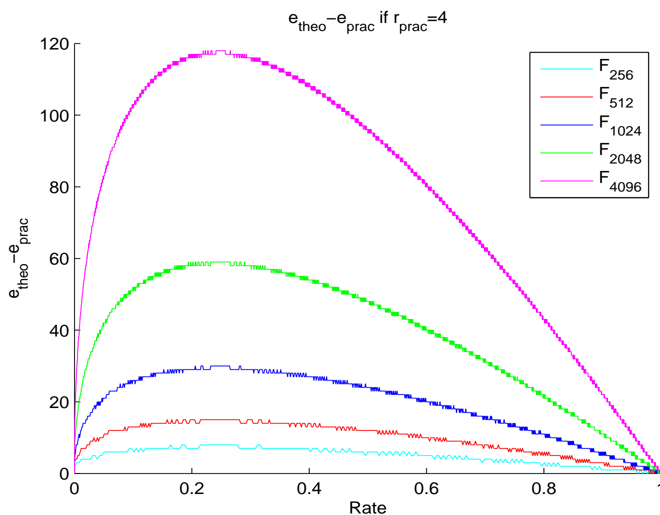

Figure 4.3: Difference between e_{theo} and e_{prac} assuming that the Guruswami-Sudan algorithm is no longer practical if $r > 4$

4.5 Size of the output list

As mentioned earlier, the size of the output list is an important practical parameter to consider. We already know that the size of the list is bounded by a polynomial, and that it is possible to recover the original codeword with a new transmission or by other previously cited means. However it may happen that a new transmission is not possible and in this case it would be useful to have only one codeword in the list. For this reason, we will study the size of the output list for several situations.

Before making simulations, let us observe the theoretical results concerning the size of the list. If the Guruswami-Sudan algorithm is used with multiplicity r it can correct up to e_r errors (we call e_r the decoding radius), then if e_r or fewer errors occur, the original codeword will certainly be in the list. Thus, we will only be interested in the average number of "non original" codewords in the list. The following results are from [12].

Let $\mathcal{C}$ be an $[n, k, d]$ Reed-Solomon code over $\mathbb{F}_q$ with redundancy $red := n-k$. Let $\mathcal{C}^*$ be the set of non-zero codewords of $\mathcal{C}$ and let $\mathbf{e}$ be an arbitrary vector of length n over $\mathbb{F}_q$. We make the following definitions

$$f(\mathbf{e}, e_r) = |\{\mathbf{m} \in \mathcal{C}^* : d(\mathbf{e}, \mathbf{m}) \leq e_r\}|$$

$$D(u, e_r) = \sum_{\mathbf{e}:w(\mathbf{e})=u} f(\mathbf{e}, e_r).$$

The interpretation is this. If the transmitted codeword is $(0, 0, \ldots, 0)$ and $\mathbf{e}$ is received, $f(\mathbf{e}, e_r)$ represents the number of non-zero codewords with distance e_r or less from $\mathbf{e}$. If $f(\mathbf{e}, e_r) = m$, we say that $\mathbf{e}$ is m-tuply falsely decodable. By linearity if $\mathbf{c}$ is the transmitted codeword, $\mathbf{e}$ is the error pattern and $\mathbf{r} = \mathbf{c} + \mathbf{e}$ is the received vector, then $f(\mathbf{e}, e_r)$ is also the number of non original codewords at distance $\leq e_r$ from $\mathbf{r}$. Thus, $D(u, e_r)$ is the total number of falsely decodable words of weight u, where a m-tuply falsely decodable word is counted m times.

Theorem 4.7. *Consider a bounded distance decoder for Reed-Solomon codes with decoding radius e_r. If $w(\mathbf{e}) = u$ then the average number of non original codewords (averaged over all patterns of weight u) in the decoding sphere of radius e_r is given by*

$$L(u, e_r) = \frac{D(u, e_r)}{\binom{n}{u}(q-1)^u}. \tag{4.7}$$

If $P(u, e_r)$ denotes the probability that there exists at least one non original codeword within distance e_r of the received vector $\mathbf{r}$,

$$P(u, e_r) \leq L(u, e_r), \quad \textit{for all } u \textit{ and } e_r, \tag{4.8}$$

$$P(u, e_r) = L(u, e_r), \quad \textit{if } 2e_r \leq red. \tag{4.9}$$

Proof: If $\mathbf{e}$ is the error pattern, the number of non original codewords at distance e_r or less from $\mathbf{r}$ is, by definition, $f(\mathbf{e}, e_r)$. Since there are $\binom{n}{u}(q-1)^u$ error patterns of weight u, the average of $f(\mathbf{e}, e_r)$ over all error patterns of weight u is

$$\frac{\sum_{\mathbf{e}:w(\mathbf{e})=u} f(\mathbf{e}, e_r)}{\binom{n}{u}(q-1)^u} = \frac{D(u, e_r)}{\binom{n}{u}(q-1)^u},$$

which proves (4.7). To prove (4.8) and (4.9), we note that if X is a random variable assuming nonnegative integer values, and $p_i = \Pr(X = i)$, then

$$\Pr(X > 0) = \sum_{i \geq 1} p_i \leq \sum_{i \geq 1} i p_i \leq E(X),$$

with equality if and only if $\Pr(X \geq 2) = 0$. If X represents the number of non original codewords within distance e_r of $\mathbf{r}$, the above inequality is equivalent to (4.8). To prove (4.9), we note that if $2e_r \leq red = n - k$, it is impossible for a sphere of radius e_r to contain two or more codewords, i.e., $\Pr(X \geq 2) = 0$.

□

To compute the average number of non original codewords within distance e_r from $\mathbf{r}$, we need to compute the numbers $D(u, e_r)$. Previously, several authors have presented more or less precise estimates of these numbers. The most precise expression was given by Cheung [13] in 1989 and the Cheung solution will be used for our tests and benchmarks. We will not reproduce this expression here since it is a bit complex.

4.5.1 Results analysis

Now, we would like to compare the theoretical results with the practical ones in the case of the Guruswami-Sudan algorithm. To this end, we need to simulate a large number of decodings with random errors and random messages. Our method is as follows:

1. Choose a $[n, k, d_C]$ Reed-Solomon code over $\mathbb{F}_q$.
2. Choose a multiplicity r and compute the maximum number of errors e_r that the Guruswami-Sudan can correct with this r.
3. Compute a random message and compute the corresponding codeword.
4. Choose the number u of errors occurring ($u \leq e_r$).
5. Compute the received vector $\mathbf{r}$ and randomly insert u errors.
6. Apply the Guruwsami-Sudan algorithm with multiplicity r and memorize the size of the output list.
7. Repeat the steps 3-6 i times.

We could also take always the zero message and randomly insert the u errors. It is obvious that to have meaningful results, i must be large. However we have already observed that for some codes the time may be very long, because the multiplicity parameter becomes too large. Of course, we could do the simulations only for errors that are smaller than e_{prac}, but if $e_{prac} \ll e_{theo}$ the results will not be complete. Moreover, it would be a quite superfluous to make comparisons for all codes. Thus, we choose several codes which are practical ($e_{prac} = e_{theo}$). We choose two Reed-Solomon codes over $\mathbf{F}_{16}$, the $[16, 2, 15]$ code and the $[16, 3, 14]$ code. For the first one and for the second one up to $r = 2$ we have done $100'000$ simulations and $10'000$ for the second one with $r = 6$.

			List size						
r	e_r	u	1	2	3	4	$I(u, e_r)$	$P(u, e_r)$	$L(u, e_r)$
1	10	9	98'519	1'481			0.0148	0.0148	0.0143
1	10	10	97'599	2'401			0.0240	0.0240	0.0236
2	11	10	73'084	24'989	1'887	40	0.2888	0.2692	0.2928
2	11	11	65'793	30'644	3'445	118	0.3789	0.3421	0.3817

Table 4.8: $[16, 2, 15]$ Reed-Solomon code over $\mathbb{F}_{16}$

For example in the first line of table 4.8 we use the Guruswami-Sudan algorithm with multiplicity $r = 1$ which can correct up to $e_r = 10$ errors and we insert $u = 9$ random errors. Over $100'000$ simulations the list has size one $98'519$ times and size two $1'481$ times.

			List size							
r	e_r	u	1	2	3	4	5	$I(u, e_r)$	$P(u, e_r)$	$L(u, e_r)$
1	8	7	99'990	10				0.0001	0.0001	0.0003
1	8	8	99'884	116				0.0012	0.0012	0.0009
2	9	8	97'230	2'758	12			0.0277	0.0278	0.0273
2	9	9	95'701	4'276	23			0.0432	0.0430	0.0453
6	10	9	4'509	4'112	1'225	149	5	0.7029	0.5491	0.6955
6	10	10	3'540	4'399	1'755	286	20	0.8847	0.6460	0.8667

Table 4.9: $[16, 3, 14]$ Reed-Solomon code over $\mathbb{F}_{16}$

We define $I(u, e_r)$ as the average number (found by simulation) of non original codewords in the output list at distance e_r from the received vector. Thus, we will compare $I(u, e_r)$ and $L(u, e_r)$. Moreover notice that the numbers given for the size of the list include all the codewords in the output list (then including the original codeword). In arrays 4.8 and 4.9, $P(u, e_r)$ denotes the probability (found by simulation) that there exists at least one non original codeword in the output list given by the Guruswami-Sudan algorithm, which is the same thing as the probability that the list contains more than one codeword. $L(u, e_r)$ is computed with the Cheung's formula. For example in the last line of table 4.9 over $10'000$ simulations we have obtained an average number of 0.8847 of non original codewords and a probability of 0.6460 of having more than one codeword in the list. The theoretical expected number of non original codewords in this case is 0.8667.

First, we compare the theoretical average number of non original codewords $L(u, e_r)$ with the one found by simulation $I(u, e_r)$. These values are very close for the most cases as expected. Thus the theoretical results and the practical ones seem to be in conformity. By theorem 4.7, $P(u, e_r) \leq L(u, e_r)$. In our tables 4.8 and 4.9, there are a few P's and L's that do not sastify this. However their values are close enough so that the fact that the P's in

the tables are experimental ones and not theoretical ones, explains the small difference. We are never in the case $2e_r \leq n - k$ and thus we can not check if (4.9) holds.

Finally observe that since the complexity increases with r and since the average number of non original codewords increases with e_r, it is naturally pointless to use the Guruswami-Sudan algorithm with multiplicity r_1 to correct e_1 errors (assuming we know that no more than e_1 errors occur) if it is possible to correct e_1 errors with a multiplicity $r_2 < r_1$.

Tables 4.8 and 4.9 show that in some cases, the probability of having more than one codeword in the list is large. Thus, if we assume that the decoder declares "success" if the list contains exactly one codeword and "failure" otherwise, we observe that with the $[16, 3, 14]$ Reed-Solomon code the decoding radius of the Guruswami-Sudan algorithm is 10 and then

$$\Pr(\text{failure}|\text{exactly 10 errors occur}) = 0.6460.$$

Thus, it would be fair to say that the Guruswami-Sudan algorithm with $r = 6$ can correct only less than 10 errors.

In view of these results, the experiments follow the theory. The most interesting value is the number of non original codewords in the list when $u = e_r = e_{theo}$. Figure 4.4 shows these values for codes of length $16, 32, 64$ over $\mathbb{F}_{16}, \mathbb{F}_{32}$ and $\mathbb{F}_{64}$ respectively.

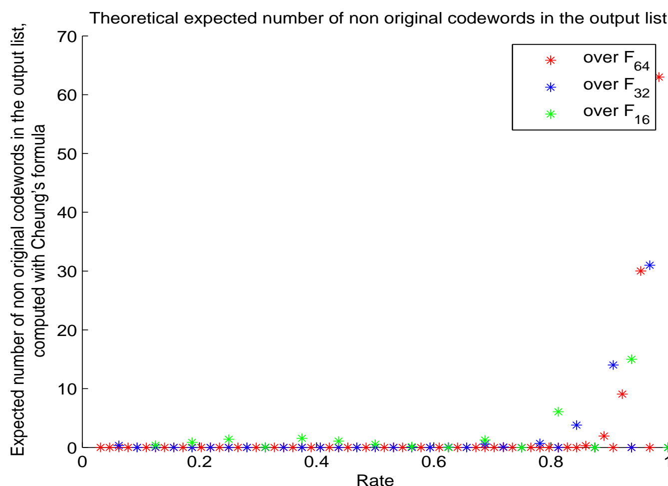

Figure 4.4: Theoretical expected numbers of non original codewords in the output list of the Guruswami-Sudan computed with the Cheung's formula.

We observe that the results are very similar for the three fields. For large rates, the results can be very bad, with more than 60 non original codewords or very good with none non original codewords in the list. It is interesting to notice the alternation between bad and good results for large rates. This is due to the simple observation that $L(u, e_r) = 0$ if $u + e_r \leq d_C - 1$. Thus for two codes with close rates, for example $C_1 = [64, 61, 4]$ and $C_2 = [64, 60, 5]$, we have that $L(e_{theo}, e_{theo}) \approx 30$ for C_1 and $L(e_{theo}, e_{theo}) = 0$ for C_2. In the two cases $u + e_r = 2e_{theo} = 4$, but $d_{C_1} = 4$ and $d_{C_2} = 5$, thus C_2 is in the situation where $u + e_r \leq d_C - 1$.

Chapter 5

The Parvaresh-Vardy algorithm

To overcome the problem of breaking the bound found by Guruswami and Sudan for Reed-Solomon list-decoding, Parvaresh and Vardy in [4] have had the idea to devise new codes, rather than devising a better list-decoder for Reed-Solomon codes. Their idea is to construct a new class of codes which has good properties and can correct a fraction of errors up to $\varepsilon = 1 - \sqrt[M+1]{M^M R^M}$, where R is the rate of the code and $M \geq 1$ is an arbitrary integer parameter.

The results in [4] are based upon two key ideas. The first is the transition from bivariate polynomial interpolation to multivariate interpolation decoding. The second idea is to devise new codes. The construction is still based upon the Reed-Solomon codes, yet differs from them in an essential way. Standard Reed-Solomon encoders (cf definition 1.10), view a message as a polynomial $f(x)$ of degree at most $k-1$, $f(x) \in \mathbb{F}_q$ and produce the corresponding codeword by evaluating $f(x)$ at n distincts elements of $\mathbb{F}_q$. Here, given $f(x)$, we first compute $M-1$ related polynomials $g_1(x), \ldots, g_{M-1}(x)$ and produce the corresponding codeword by evaluating all these polynomials. In the case $M = 1$, the Parvaresh-Vardy algorithm becomes the Guruswami-Sudan algorithm.

5.1 Presentation of the algorithm

We can now describe the construction of the new codes for the simplest nontrivial case $M = 2$. Let us define the needed parameters. Some of them are the same as for the Guruswami-Sudan algorithm:
A field $\mathbb{F}_q, n, k$, n distinct elements $x_1, \ldots, x_n \in \mathbb{F}_q$, a multiplicity paramater r, the number of agreements t.
The others parameters are new and are:
A fixed basis $\{1, \alpha\}$ for $\mathbb{F}_{q^2}$ over $\mathbb{F}_q$, a polynomial $e(x) \in \mathbb{F}_q[x]$ of degree k

and irreducible over $\mathbb{F}_q$ and a positive integer a that satisfies

$$a \geq \left\lceil r \sqrt[3]{\frac{n}{k-1}\left(1+\frac{1}{r}\right)\left(1+\frac{2}{r}\right)} + \frac{1}{k-1} \right\rceil . \tag{5.1}$$

As usual, we define C by describing its encoding function $E : \mathbb{F}_q^k \to C \subseteq \mathbb{F}_{q^2}^n$. The encoding function has two parts. Given k arbitrary message symbols $f_0, f_1, \ldots, f_{k-1} \in \mathbb{F}_q$, the encoder first constructs the polynomial $f(x) = \sum_{i=0}^{k-1} f_i x^i$, and then computes

$$g(x) = (f(x))^a \bmod e(x) \tag{5.2}$$

over $\mathbb{F}_q$. The codeword $(c_1, c_2, \ldots, c_n) \in C$ corresponding to the message $f_0, f_1, \ldots, f_{k-1}$ is then given by

$$c_j = f(x_j) + \alpha g(x_j) \quad \text{for } j = 1, 2, \ldots, n \tag{5.3}$$

It is obvious that the rate of the code C is $R = k/2n$. Moreover, the code C is a subset of a Reed-Solomon code $\mathcal{C}$ of length n and dimension k over $\mathbb{F}_{q^2}$. Thus the minimum distance of C is at least $n - k + 1$. It is also important to notice that in most cases C is not a linear subspace of $\mathcal{C}$ because the computation in (5.2) is, in general, not a linear operation. If $q = p^s$ for a prime p, making C lienar over $\mathbb{F}_q$ requires $a = p^b$, b multiple of s. Making C linear over $\mathbb{F}_{q^2}$ requires $a - 1$ to be a multiple of $q^k - 1$, and leads to the degenerate case where $g(x) \equiv f(x)$.

We need to show that the encoder map in (5.2) and (5.3) can be computed in polynomial time. To this end, it is convenient to think of the polynomials $f(x)$ and $g(x)$ of degree $\leq k-1$ as elements in the extension field

$$\mathbb{K} := \mathbb{F}_q[x]/\langle e(x) \rangle \tag{5.4}$$

of order q^k. Then the computation is simply $\gamma = \beta^a$ in $\mathbb{K}$. Parvaresh and Vardy in [4, Section 3.1] mention an algorithm that compute β^a in polynomial time.

We have seen the construction of the codes. Now we are interested in their decoding. The idea is the same as in the Guruswami-Sudan algorithm. Given the received vector $\mathbf{r} = (r_1, r_2, \ldots, r_n)$ over $\mathbb{F}_{q^2}$, we first decompose each r_j into its two components over $\mathbb{F}_q$, using the fixed basis $\{1, \alpha\}$. Thus we write

$$r_j = y_j + \alpha z_j \quad \text{for } j = 1, 2, \ldots, n$$

where y_j and z_j are in $\mathbb{F}_q$. Thus we obtain the following set of points

$$\mathcal{P} = \{(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_n, y_n, z_n)\} \tag{5.5}$$

where $x_1, x_2, \ldots, x_n$ are the n distinct points chosen at the beginning. As in the Guruswami-Sudan algorithm, we would like to find a polynomial

$Q(x, y, z)$ which passes through (x_j, y_j, z_j) r times for $j = 1, 2, \ldots, n$. Then, we will find the roots of $Q(x, y, z)$, to this end, we will transform it. We can now describe the general form of the Parvaresh-Vardy algorithm in table 5.1 before presenting each step in details.

Parvaresh-Vardy algorithm:

Input: $n, q, k, \{1, \alpha\}, e(x), r, a, t$, n distinct elements $x_1, x_2, \ldots, x_n \in \mathbb{F}_q$.

Step 1: Given a vector $\mathbf{r}$ over $\mathbb{F}_{q^2}$, set up the interpolation set $\mathcal{P}$ and compute the interpolation polynomial $Q(x, y, z) \in \mathbb{F}_q[x, y, z]$ such that

1. Q is not identically zero.
2. Q has a singularity of multiplicity r at each point (x_j, y_j, z_j) for $j = 1, 2, \ldots, n$.
3. Q is the polynomial of the least weighted degree which satisfies the two previous constraints.

Step 2: Find the polynomials $f(x)$ and $g(x) = (f(x))^a \bmod e(x) \in \mathbb{F}_q$ such that $Q(x, f(x), g(x)) \equiv 0$. For each pair f, g compute the corresponding codeword $\mathbf{c}$ according to (5.3) and check if $c_i = r_i$ for at least t $i \in [n]$ and, if so, include f in the output list.

Output: Polynomials $f(x), g(x)$ according to Step 2.

Table 5.1: First presentation of the Parvaresh-Vardy algorithm

In order to find the polynomial Q, we can extend the definition of a shifted polynomial for polynomial in three variables. Thus, the definition 4.1 becomes:

Definition 5.1. *A polynomial $Q(x, y, z)$ passes through the point (α, β, δ) at least r times if all the coefficients of total degree less than r of the polynomial $Q_{\alpha,\beta,\delta}(x, y, z) = Q(x+\alpha, y+\beta, z+\delta)$ are zero. The polynomial $Q_{\alpha,\beta,\delta}(x, y, z)$ is called a shifted polynomial.*

Then, the coefficients of the shifted polynomial can be described as follows:

$$(q_{\alpha,\beta,\delta})_{abc} := \sum_{i \geq a} \sum_{j \geq b} \sum_{k \geq c} \binom{i}{a} \binom{j}{b} \binom{k}{c} q_{ijk} x^{i-a} y^{j-b} z^{k-c}. \quad (5.6)$$

If we use the same notation as in the Guruswami-Sudan algorithm, i.e., $Q^{(i)}$ is the shifted polynomial in (x_i, y_i, z_i), the polynomial $Q(x, y, z)$ is said

to have a zero of multiplicity r at point $(x_i, y_i, z_i) \in \mathbb{F}_q \times \mathbb{F}_q \times \mathbb{F}_q$ if

$$Q_{abc}^{(i)} = 0, \forall a, b, c \geq 0 \text{ s.t. } a + b + c < r. \tag{5.7}$$

Finally, let us define the weighted degree for a trivariate monomial $x^a y^b z^c$. It is defined as for a bivariate monomial. Here we only need the $(1, k-1, k-1)-$weighted degree (since f and g are polynomials of degree $\leq k-1$) and thus we have

$$\text{wtdeg}(x^a y^b z^c) := a + (k-1)b + (k-1)c. \tag{5.8}$$

As explained for the previous algorithms, it should be obvious that $Q(x, y, z)$ can be computed in polynomial time. We need a bound on the weighted degree of Q in order to make sure that there exists such a polynomial. The idea here is the same as in the previous algorithms. We need to find the conditions such that the number of unknowns is larger than the number of constraints. The number of constraints is simply

$$\sum_{i=0}^{r-1} \sum_{j=0}^{r-i-1} (r - i - j) = n\frac{(r+2)(r+1)r}{6}.$$

We could compute the exact number of unknowns with the following addition, where l is the smallest $(1, k-1, k-1)-$weighted degree such that there surely exists a Q of weighted degree l:

$$\sum_{i=0}^{l} \sum_{j=0}^{\lfloor \frac{l-i}{k-1} \rfloor} \sum_{m=0}^{\lfloor \frac{l-i-(k-1)j}{k-1} \rfloor} 1. \tag{5.9}$$

However it suffices to find a lower bound of this equation. Simplifying it, we finally obtain that

$$(5.9) \geq \frac{l^3}{6(k-1)^2}.$$

Then, there exists a polynomial Q that satisfies the constraints if

$$n\frac{(r+2)(r+1)r}{6} < \frac{l^3}{6(k-1)^2}.$$

Thus, the weighted degree of the polynomial Q is at most

$$l = \left\lceil \sqrt[3]{n(k-1)^2 r(r+1)(r+2)} \right\rceil,$$

since with this l we are sure that there exists a solution, but we need the least weighted degree solution and there may be solutions with weighted degree $< l$.

We now have the interpolation polynomial $Q(x, y, z)$ computed in polynomial time. We follow the same way as in the Guruswami-Sudan algorithm, i.e., we need to establish under what conditions $Q(x, f(x), g(x)) \equiv 0$, where $f(x)$ and $g(x)$ are the polynomials used for the construction of the code C. Once again, we will prove that that polynomial $p(x) := Q(x, f(x), g(x))$ is identically zero if the degree of this polynomial is smaller than its number of zeros. Now, the number of agreements is not only the number of points at which $f(x_j) = y_j$ but is defined as the number of points at which both $f(x_j)$ and $g(x_j)$ agree with the received points, i.e.,

$$t := |\{j : f(x_j) = y_j \text{ and } g(x_j) = z_j\}|. \tag{5.10}$$

Actually we will prove the following lemma.

Lemma 5.1. *Let $Q(x, y, z)$ be the polynomial with respect to the set $\mathcal{P}$. Given a pair of polynomials $f(x)$ and $g(x)$ over $\mathbb{F}_q[x]$ and provided*

$$t \geq \left\lceil \sqrt[3]{n(k-1)^2 \left(1 + \frac{1}{r}\right)\left(1 + \frac{2}{r}\right)} + \frac{1}{r} \right\rceil$$

we have $Q(x, f(x), g(x)) \equiv 0$. Namely, the univariate polynomial $p(x) = Q(x, f(x), g(x))$ is identically zero.

Proof: Still following the same plan, we will show that the polynomial $p(x) := Q(x, f(x), g(x))$ has more zeros than its degree and hence is identically zero. Since $\deg f(x), \deg g(x) \leq k - 1$, the polynomial $p(x)$ has degree at most $\Delta := \text{wtdeg} Q(x, y, z)$. On the other hand, for every integer $j \in \{1, 2, \ldots, n\}$ that is counted in (5.10), the interpolation polynomial $Q(x, y, z)$ has a zero of multiplicity r at the point $(x_j, f(x_j), g(x_j))$. Hence the total number of zeros of $p(x)$ in $\mathbb{F}_q$, counted with multiplicity, is at least rt.
Thus if $rt > \Delta$ then $p(x)$ has more zeros than its degree. We do not know Δ but we know that Δ is upper-bounded by $l = \left\lceil \sqrt[3]{n(k-1)^2 r(r+1)(r+2)} \right\rceil$ and then we get that if

$$rt > \left\lceil \sqrt[3]{n(k-1)^2 r(r+1)(r+2)} \right\rceil$$

the polynomial $p(x)$ is identically zero. The lemma follows immediately.

□

Recall that we have received the vector $\mathbf{r}$. We can now establish the following lemma given the number of errors the algorithm of Parvaresh-Vardy can correct.

Lemma 5.2. *Suppose that a codeword* $\mathbf{c} \in C$ *differs from the given vector* $\mathbf{r}$ *in at most*

$$e = \left\lfloor n - \sqrt[3]{n(k-1)^2\left(1+\frac{1}{r}\right)\left(1+\frac{2}{r}\right)} - \frac{1}{r} \right\rfloor \tag{5.11}$$

positions, and let $f(x), g(x)$ *denote the message polynomial(s) that produce* $\mathbf{c}$ *according to the mapping (5.2)-(5.3). Then* $Q(x,y,z)$ *satisfies* $Q(x, f(x), g(x)) \equiv 0$.

Proof: Since $t \geq n - e$, the proof follows from lemma 5.1.

□

It remains to show how to recover the message $f(x)$ from the interpolation polynomial $Q(x,y,z)$, assuming that $Q(x, f(x), g(x)) \equiv 0$. To this end, we first reduce $Q(x,y,z)$ modulo $e(x)$. That is, we compute

$$P(y,z) := Q(x,y,z) \bmod e(x). \tag{5.12}$$

To see that $P(y,z)$ is indeed a bivariate polynomial, first write $Q(x,y,z)$ as an element of $\mathbb{F}_q[x][y,z]$, namely

$$Q(x,y,z) = \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} q_{i,j}(x) y^i z^j \tag{5.13}$$

and let $p_{i,j}(x) = q_{i,j}(x) \bmod e(x)$. Then clearly $p_{i,j}(x)$ can be regarded as an element of the extension field $\mathbb{K}$ defined in (5.4), and therefore $P(y,z) \in \mathbb{K}[y,z]$.

It is important to show that P is not the all zero polynomial. To this end we need the fact that Q is the polynomial of the least weighted degree which has a zero of multiplicity r at the points $\{(x_i, y_i, z_i)\}_{i=1}^n$. Indeed, if $P(y,z) \equiv 0$, it means that $e(x)$ is a factor of $Q(x,y,z)$ and then, there exists the polynomial

$$Q'(x,y,z) := \frac{Q(x,y,z)}{e(x)} \tag{5.14}$$

which has also a zero of multiplicity r at the points $\{(x_i, y_i, z_i)\}_{i=1}^n$ since $e(x)$ is irreducible. The weighted degree of Q' is $\text{wtdeg} Q(x,y,z) - \deg e(x)$. This contradicts the fact that Q is the least weighted degree polynomial which satisfies the interpolation constraints. We can conclude that $P(y,z)$ is not identically zero.

Now we will show that $P(y,z)$ is more "practical" than $Q(x,y,z)$.

Lemma 5.3. *A polynomial* $Q(x,y,z) \in \mathbb{F}_q[x,y,z]$ *satisfies* $Q(x,f(x),g(x)) \equiv 0$ *if and only if it belongs to the ideal of* $\mathbb{F}_q[x,y,z]$ *generated by* $y-f(x)$ *and* $z-g(x)$*, that is, if and only if there exist some polynomials* $A(x,y,z), B(x,y,z) \in \mathbb{F}_q[x,y,z]$ *such that*

$$Q(x,y,z) = A(x,y,z)(y-f(x)) + B(x,y,z)(z-g(x)). \tag{5.15}$$

Proof: It should be obvious that if $Q(x,y,z)$ can be expressed as in (5.15), then $Q(x,f(x),g(x)) \equiv 0$. To prove the converse, fix a monomial order $\prec$ such that $y \prec f(x)$ and $z \prec g(x)$, then divide $Q(x,y,z)$ by $y-f(x)$ and $z-g(x)$ to express it as follows:

$$Q(x,y,z) = A(x,y,z)(y-f(x)) + B(x,y,z)(z-g(x)) + R. \tag{5.16}$$

The polynomial division algorithm guarantees that none of the monomials in the remainder polynomial $R \in \mathbb{F}_q[x,y,z]$ is divisible by the leading term of $y-f(x)$ or $z-g(x)$. But since $y \prec f(x)$ and $z \prec g(x)$, this simply means that $R(x,y,z) = R(x)$. This, in conjunction with (5.16) and the assumption that $Q(x,f(x),g(x)) \equiv 0$, then implies that $R(x) \equiv 0$, and the lemma follows.

□

Lemma 5.4. *Suppose that* $Q(x,f(x),f(g)) \equiv 0$*, and let* β *and* γ *denote the elements of* $\mathbb{K}$ *that correspond to* $f(x)$ *and* $g(x)$ *respectively. Then* $P(\beta,\gamma) = 0$.

Proof: If $Q(x,f(x),g(x)) \equiv 0$, then by lemma 5.3 it can be written in the form

$$A(x,y,z)(y-f(x)) + B(x,y,z)(z-g(x))$$

for some polynomials $A(x,y,z), B(x,y,z) \in \mathbb{F}_q[x,y,z]$. Therefore $P(y,z)$ can be written in the form

$$P(y,z) = \tilde{A}(y,z)(y-\beta) + \tilde{B}(y,z)(z-\gamma) \tag{5.17}$$

where $\tilde{A}(y,z)$ and $\tilde{B}(y,z)$ in $\mathbb{K}[y,z]$ are the remainders of $A(x,y,z)$ and $B(x,y,z)$ upon division by $e(x)$. It is now obvious from (5.17) that $P(\beta,\gamma) = 0$.

□

Here is the crucial part of the decoding algorithm. From the encoding rule (5.2), we know *a priori* that $\gamma = \beta^a$ in $\mathbb{K}$. Hence $P(\beta,\gamma) = 0$ implies that β is a root of the polynomial $H(y) = P(y,y^a)$. But $H(y)$ is a univariate polynomial, so all of its roots can be found using well-known techniques. First, however, we should make sure that $H(y) \neq 0$.

Lemma 5.5. *The polynomial $H(y) = P(y, y^a)$ is not the all-zero polynomial.*

Proof: Assume to the contrary that $H(y) = P(y, y^a) \equiv 0$. This happens if and only if $z - y^a$ is a factor of $P(y, z)$. Clearly, this cannot be the case if $\deg_y P(y, z) < a$. But it follows from (5.12) and (5.8) that

$$\deg_y P(y, z) \leq \frac{\text{wtdeg}Q(x, y, z)}{k-1}. \tag{5.18}$$

l provides an upper bound on wtdeg$Q(x, y, z)$, and our condition for a in (5.1) uses this bound to guarantee that the value of a is strictly greater that the right-hand side of (5.18).

□

We can now establish the following theorem for the special case $M = 2$.

Theorem 5.6. *Given a vector $\mathbf{r} = (r_1, r_2, \ldots, r_n)$ over $\mathbb{F}_{q^2}$, the Parvaresh-Vardy algorithm outputs in polynomial time the list of all codewords of C that differ from $\mathbf{r}$ in at most*

$$e = \left\lfloor n - n\sqrt[3]{4R^2\left(1+\frac{1}{r}\right)\left(1+\frac{2}{r}\right)} - \frac{1}{r} \right\rfloor \tag{5.19}$$

positions, where $r \geq 1$ is an arbitrary multiplicity parameter. The size of this list is at most L^2, where

$$L = \left\lceil r\sqrt[3]{\frac{\left(1+\frac{1}{r}\right)\left(1+\frac{2}{r}\right)}{2R}} \right\rceil + 1. \tag{5.20}$$

Proof: The expression for t in (5.19) follows immediately from (5.11), when taking into account that $R = k/2n$. The list-size is obviously bounded by the degree of $H(y)$. It is clear that $\deg H(y) \leq a \deg_{tot} P(y, z)$, where $\deg_{tot}$ is the total degree. But $\deg_{tot} P(y, z)$ is bounded by the right hand side of (5.18), whence (5.20) now follows by (5.1) and the bound l.

In general, we will always choose a to be just larger than the right hand side of (5.18). Hence, the list-size is essentially bounded by the square of $a \simeq \text{wtdeg}Q(x, y, z)/(k-1)$.

□

Table 5.2 presents the second step of the Parvaresh-Vardy algorithm with more details than in table 5.1.

Parvaresh-Vardy algorithm, Step 2:
Step 2.1: Compute $P(y,z) = Q(x,y,z) \bmod e(x)$.
Step 2.2: Compute $H(y) = P(y, y^a)$.
Step 2.3: Find roots of $H(y)$.
Step 2.4: For each root β of Step 2.3, compute the associated $\gamma = \beta^a$ and the corresponding codeword $\mathbf{c}$ as in (5.3). Check if $c_i = r_i$ for at least t values $i \in [n]$ and, if so, include β (and the associated γ) in the output list.

Table 5.2: Detailed Step 2 of the Parvaresh-Vardy algorithm

5.2 Error bound

If we now consider the error-rate in the asymptotically case $r \to \infty$ we obtain that this algorithm can correct a fraction of errors up to

$$1 - \sqrt[3]{(2R)^2}.$$

We can now briefly discuss the general case $M \geq 2$.

5.3 Parvaresh-Vardy algorithm, general case

In this case, the encoder uses $M-1$ irreducible polynomials $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{M-1}$, where $\mathcal{T}_i = (y, z_i) = (z_i - y^{a_i})$ for all i.

The code C will now be a subset of $\mathbb{F}_Q^n$, where $Q = q^M$. We modify the list of code parameters in previous section as follows:
A fixed basis $\{1, \alpha_1, \ldots, \alpha_{M-1}\}$ for $\mathbb{F}_Q$ over $\mathbb{F}_q$, a sequence of $M-1$ positive integers $a_1, a_2, \ldots, a_{M-1}$ given by

$$a_i := \sum_{j=0}^{i} \left\lceil \sqrt[M+1]{\frac{n}{k-1} \prod_{l=0}^{M} (r+l)} \right\rceil^j . \tag{5.21}$$

All others parameters remain unchanged. Given k arbitrary symbols $f_0, f_1, \ldots, f_{k-1} \in \mathbb{F}_q$, the encoder first constructs the polynomial $f(x) = \sum_{i=0}^{k-1} f_i x^i$, and then computes

$$g_i(x) = (f(x))^{a_i} \bmod e(x) \tag{5.22}$$

for $i = 1, 2, \ldots, M-1$. The codeword $(c_1, c_2, \ldots, c_n) \in C$ corresponding to the message $f_0, f_1, \ldots, f_{k-1}$ is given by

$$c_j = f(x_j) + \sum_{i=1}^{M-1} \alpha_i g_i(x_j) \quad \text{for } j = 1, 2, \ldots, n. \tag{5.23}$$

The rate of the code is $R = k/Mn$. The minimum distance of C is at least $n - k + 1$, since C is a subset of a Reed-Solomon code of length n and dimension k over $\mathbb{F}_Q$.

The multivariate decoding follows the same ideas as the trivariate decoding. Given a received vector $\mathbf{r} = (r_1, r_2, \ldots, r_n)$ over $\mathbb{F}_Q$, we decompose each r_j into its components over $\mathbb{F}_q$ using the basis $\{1, \alpha_1, \ldots, \alpha_{M-1}\}$. Thus we write

$$r_j = y_j + \sum_{i=1}^{M-1} \alpha_i z_{i,j} \quad \text{for } j = 1, 2, \ldots, n,$$

where y_j and $z_{i,j}$ are in $\mathbb{F}_q$. We then set up the interpolation set $\mathcal{P}$ consisting of the n points $(x_j, y_j, z_{1,j}, \ldots, z_{M-1,j})$. As before, the first decoding step consists of computing the interpolation polynomial $Q(x, y, z, z_1, \ldots, z_{M-1})$, defined as the least $(1, k-1, k-1, \ldots, k-1)-$weighted degree polynomial that has a zero of multiplicity r at each point of $\mathcal{P}$.

We will not describe in details this algorithm since the proofs are easy generalization of the description of the case $M = 2$. The steps are the same as for the case $M = 2$ and next, as in (5.12), we reduce the interpolation polynomial modulo $e(x)$ to obtain the polynomial $P(y, z_1, \ldots, z_{M-1})$ in the ring $\mathbb{K}[y, z_1, \ldots, z_{M-1}]$, where $\mathbb{K}$ is defined by (5.4). Then exactly the same argument as in the case $M = 2$ shows that $P(y, z_1, \ldots, z_{M-1})$ is not the all-zero polynomial. The third decoding step still consists of computing

$$H(y) := P(y, y^{a_1}, y^{a_2}, \ldots, y^{a_{M-1}}). \tag{5.24}$$

The fourth and final decoding step is exactly as before: find all the root of $H(y)$ in $\mathbb{K}$, and output this list. We find that

$$\deg H(y) \le a_{M-1} \deg_{tot} P_0 \le \frac{\Delta_M a_{M-1}}{k-1}. \tag{5.25}$$

Since the list-size of the decoder is obviously bounded by $\deg H(y)$ and using the fact that $R = k/Mn$, (5.21) and (5.25), we obtain the following theorem.

Theorem 5.7. *Let q be a power of a prime. Then for all positive integers $r, M, n \le q$, and $k \le n$, there is a code C of length n and rate $R = k/nM$ over $\mathbb{F}_Q = GF(q^M)$, equipped with an encoder E and a decoder D that have the following properties. Given any vector $\mathbf{r} \in \mathbb{F}_Q^n$, the decoder D outputs the list of all codewords that differ form $\mathbf{r}$ in at most*

$$e = \left\lfloor n - n \sqrt[M+1]{M^M R^M \left(1 + \frac{1}{r}\right) \cdots \left(1 + \frac{M}{r}\right)} - \frac{1}{r} \right\rfloor$$

positions (where r, M are arbitrary integer parameters). The size of the list is at most

$$L = \left\lceil r \sqrt[M+1]{\frac{(1+\frac{1}{r})(1+\frac{2}{r})\dots(1+\frac{M}{r})}{MR}} \right\rceil^M + o(1).$$

Moreover, both the encoder E and the decoder D run in time (number of operations in $\mathbb{F}_q$) that is bounded by a polynomials in n and r, for every given $M \geq 1$.

The case $M > 2$ is better than the case $M = 2$ only for few rates. For example, $M = 5$ improves $M = 2$ for rates up to 5.1×10^{-3}, $M = 10$ for rates up to 1.2×10^{-3} and $M = 100$ for rates up to $3.14 times 10^{-6}$.

5.4 Comparison with the Guruswami-Sudan algorithm

Let us now compare the results of this algorithm with the previous ones. The best algorithm we have until now was the Guruswami-Sudan algorithm with its correction bound of $1 - \sqrt{R}$. In the case $M = 2$ for the Parvaresh-Vardy algorithm, we have the first bound on R which is $R \leq k/nM = k/2n \leq 1/2$ since $k \leq n$. Thus the Parvaresh-Vardy can be compared with the Guruswami-Sudan algorithm only for codes with $R \leq 1/2$. We get

$$1 - \sqrt[3]{(2R)^2} > 1 - \sqrt{R} \Leftrightarrow 16R^4 < R^3 \Leftrightarrow R < 1/16.$$

The Parvaresh-Vardy algorithm in the case $M = 2$ brings improvements over the Guruswami Sudan for rates $R < 1/16$. As mentioned just before, the general case $M > 2$ does not improve the Guruswami-Sudan algorithm for rates $R > 1/16$. In conclusion, the Parvaresh-Vardy algorithm brings theoretical improvements for rates $R < 1/16$.

5.5 Implementation

In this chapter we will briefly present the important things to notice for an implementation of the Parvaresh-Vardy algorithm. First, observe that the multiplicity parameter r leads to the same problem as for the Guruswami-Sudan algorithm. When r increases, the size of the matrix also increases and then the running time of Step 1, i.e., the resolution of the linear system, increases too. Even with an algorithm like the one in [9], this computation takes $O(n^{8/3}r^8/k^{2/3})$ operations in $\mathbb{F}_q$. It is a first important thing to observe.

Moreover, we have compared the fraction of errors to compare the Guruswami-Sudan and the Parvaresh-Vardy algorithms. Let us compare the

number of errors they can correct for a given code of rate $R = 15/256 = 30/(2 \cdot 256) < 1/16$. The Guruswami-Sudan algorithm can correct a maximum of 196 errors with a multiplicity $r = 172$. To correct the same number of errors, the Parvaresh-Vardy algorithm needs a multiplicity $r = 934$ and cannot correct 197 errors. Although $1-\sqrt{R} = 0.7579 < 1-\sqrt[3]{(2R)^2} = 0.7605$ in a practical implementation the Parvaresh-Vardy algorithm cannot correct more errors than the Guruswami-Sudan algorithm for the considered code of rate $R = 0.0586$.

The construction of the matrix in Step 1 of table 5.1 is also not so easy. The constraints are given by

$$(q_{\alpha,\beta,\delta})_{abc} := \sum_{i \geq a} \sum_{j \geq b} \sum_{k \geq c} \binom{i}{a} \binom{j}{b} \binom{k}{c} q_{ijk} x_s^{i-a} y_s^{j-b} z_s^{k-c} = 0,$$

for all point $\{(x_s, y_s)\}_{s=1}^n$ and for all $a, b, c \in \mathbb{N}$ with $a+b+c < r$ ($(q_{\alpha,\beta,\delta})_{abc}$ was defined in (5.6)). The unknowns are the coefficients q_{ijk} of Q. Thus we must have a matrix M and a vector $\mathbf{q}$ such that $\mathbf{q}M = 0$ and such that $\mathbf{q}$ represents the coefficients q_{ijk}. Our implementation takes as $\mathbf{q}$:

$$\mathbf{q} = (q_{000}, q_{100}, \ldots, q_{l00}, q_{101}, \ldots, q_{l-(k-1)01}, \ldots, q_{00\lfloor \frac{l}{k-1} \rfloor}, q_{010}, \ldots, q_{0\lfloor \frac{l}{k-1} \rfloor 0}).$$

Thus, the first column of the matrix contains the values

$$\binom{i}{a} \binom{j}{b} \binom{k}{c} x_s^{i-a} y_s^{j-b} z_s^{k-c} \tag{5.26}$$

for the point (x_1, y_1) and the case $a = b = c = 0$ and the order i, j, k corresponding to that of $\mathbf{q}$. Suppose now that $r > 1$. The second column must containt the values (5.26) for the point (x_1, y_1) and the case $a = 1, b = 0, c = 0$. Then the third column for the case $a = 0, b = 1, c = 0$, etc, until all the cases $a + b + c < r$ have been considered. Thus, we have finished with the point (x_1, y_1) and the rest of the matrix is the repetition of the above with the other points (x_s, y_s), $s = 2, \ldots, n$. To summarize, observe that we need seven for loops to construct the matrix. One on s, the index of the point, then three on a, b and c, and finally three on i, j and k.

Another observation in the construction of the matrix are the binomial coefficients in (5.26). They can quickly become large and therefore, long to compute. However if $\mathbb{F}_q$ is an extension field of $\mathbb{F}_2$, the binomial coefficients are either 0 or 1. There exists a solution to compute the parity of a binomial coefficient $C(n, k) := \binom{n}{k}$ avoiding computing $n!$ and $k!$. The solution is to consider the binary representation of n and k. These representations are easily available when programming in C++. We line the two representations

up, with n on top. If k is shorter than n, we can pad the representation of k on the left. Now, we examine each position. If the top digits are always equal or larger than the bottom digits, then $C(n, k)$ is odd and if there is at least one position in which the top digit is smaller than the bottom digit, so $C(n, k)$ is even. This method avoids computing large factorials.

A second difficulty in an implementation of the Parvaresh-Vardy algorithm is Step 2.3 of table 5.2, i.e., is to find the roots of $H(y) \in \mathbb{K}[y]$. Parvaresh and Vardy suggest in [4] to use the Shoup algorithm [7]. However this algorithm is very difficult to implement. In this project we use the library for doing number theory NTL [8] in which some root-finding algorithms, like the one described in [15], are already implemented. They are less efficient that the first cited algorithm but for simulations over $\mathbb{F}_{16}$ that is more than enough.

Recall that the aim is to find the roots of $H(y)$ in $\mathbb{K}[y]$, where $\mathbb{K} = \mathbb{F}_q[x]/\langle e(x) \rangle$. Thus the order of $\mathbb{K}$ is q^k. Assuming that $q = p^s$ we have that $\mathbb{K}$ is isomorphic to the finite field $\mathbb{L} := \mathbb{F}_{p^{ks}}$. Whatever algorithm we use to find the roots, it applies on the field $\mathbb{F}_{p^{ks}}$. Thus we need to transform the coefficient of $\mathbb{H}$ to elements of $\mathbb{L}$, then we can apply the algorithm and find the roots in $\mathbb{L}$. Finally we have to transform the roots to elements of $\mathbb{H}$ again. All of that implies to have efficient functions that deal with multivariate polynomials, that find minimal multivariate polynomials, that transform elements from $\mathbb{H}$ to $\mathbb{L}$ and vice-versa.

What we want to explain here is that the Parvaresh-Vardy algorithm requires the implementation of many of algebraic functions for multivariate polynomials. Moreover, the Parvaresh-Vardy algorithm brings theoretical improvements over the Guruswami-Sudan algorithm but in a practical implementation, the improvements are limited since the parameter r must be very large to correct as many errors as the Guruswami-Sudan algorithm for some rates, as shown in a example at the beginning of this section, and since the improvements of the Parvaresh-Vardy algorithm are only for rates $R < 1/16$.

Chapter 6

The PVGS algorithm

We will present in this chapter a new algorithm, (we will call it PVGS for simplicity), based on both the Parvaresh-Vardy and Guruswami-Sudan algorithms, which enables us to correct a larger fraction of errors than the previous studied algorithms for all rates $R \in (0, 1)$. In fact, we will construct a new way to encode and then apply the known decoding algorithms. Observe that in the Parvaresh-Vardy algorithm we obtain solutions f, g modulo $e(x)$. The PVGS algorithm will use this property of the Parvaresh-Vardy algorithm. Actually, the main idea of the PVGS algorithm is the following:

- Construct f and g in two parts and then apply the Parvaresh-Vardy algorithm to find a part of them (the remainder modulo $e(x)$) and the Guruswami-Sudan algorithm to find the remaining part.

6.1 PVGS algorithm

We can now introduce the PVGS algorithm. We begin with the definition of the needed parameters. Most of them are the parameters of the Parvaresh-Vardy algorithm (cf chapter 5) since we use this algorithm for decoding and we do not repeat here the properties of each one.

Parameters for the PVGS algorithm:
Parameters already defined in the Parvaresh-Vardy algorithm:

- q, n, k, n elements $x_1, \ldots, x_n \in \mathbb{F}_q$, $\{1, \alpha\}, r$, polynomial $e(x)$, the field $\mathbb{K}$.

New parameters and parameters with new definition:

- A positive integer $s \leq n$.
- A positive integer a that satisfies

$$a \geq \left\lceil r \sqrt[3]{\frac{n}{k+s-1}\left(1+\frac{1}{r}\right)\left(1+\frac{2}{r}\right)} + \frac{1}{k+s-1} \right\rceil .$$

We assume that all these parameters are fixed in advance. We define the new code $\mathcal{C}_1$ by describing its encoding map $E : \mathbb{F}_q^{k+2s} \to \mathcal{C}_1 \subseteq \mathbb{F}_{q^2}^n$ as follows. The encoder first chooses $k + 2s$ message symbols $u_0, u_1, \ldots, u_{k+2s-1} \in \mathbb{F}_q$ and constructs polynomials $f_0(x), f_1(x), g_1(x)$ from them such that

1. $f_0(x) = u_0 + \ldots + u_{k-1}x^{k-1} \in \mathbb{F}_q[x]_{<k}$.
2. $f_1(x) = u_k + \ldots + u_{k+s-1}x^{s-1} \in \mathbb{F}_q[x]_{<s}$.
3. $g_1(x) = u_{k+s} + \ldots + u_{k+2s-1}x^{s-1} \in \mathbb{F}_q[x]_{<s}$.

Then, he computes

$$g_0(x) = (f_0(x))^a \bmod e(x) \tag{6.1}$$

over $\mathbb{F}_q$. Define now

$$F(x) := f_0(x) + e(x)f_1(x), \tag{6.2}$$

$$G(x) := g_0(x) + e(x)g_1(x), \tag{6.3}$$

The codeword $\mathbf{c}_{PV} = (c_{PV_1}, c_{PV_2}, \ldots, c_{PV_n}) \in \mathcal{C}_1$ corresponding to the message symbols $u_0, \ldots, u_{k+2s-1}$ is then given by

$$c_{PV_j} = F(x_j) + \alpha G(x_j) \quad \text{for } j = 1, 2 \ldots, n. \tag{6.4}$$

Notice that the encoding is similar to the Parvaresh-Vardy algorithm. The rate of $\mathcal{C}_1$ is $R_1 = \frac{k+2s}{2n}$. As mentioned in the chapter 5, in most cases the code $\mathcal{C}_1$ is not linear.

We have seen the code and its encoding, we can now discuss the decoding of this code. Because the Parvaresh-Vardy algorithm finds f and g modulo $e(x)$, if we apply it to F and G then we will obtain f_0 and g_0 (see (6.2) and (6.3)). Table 6.1 presents the steps of the algorithm. As mentioned before, we will apply first the Parvaresh-Vardy algorithm and then the Guruswami-Sudan algorithm. We now need to detail the steps of the PVGS algorihtm. In particular, we need to define the vectors $\mathbf{r}_{PV}$ and $\mathbf{r}_{GS}$. In Step 1, we apply the Parvaresh-Vardy algorithm and so the decoding steps are already described in the chapter 5 with lemmas and proofs. We only have to adapt some of these results to our case.

6.1.1 Step 1, Parvaresh-Vardy algorithm

The codeword $\mathbf{c}_{PV}$ (6.4) is sent and we receive the vector $\mathbf{r}_{PV} = (r_{PV_1}, \ldots, r_{PV_n})$ over $\mathbb{F}_{q^2}$ where each r_{PV_j} is decomposed into its two components over $\mathbb{F}_q$, namely

$$r_{PV_j} = y_j + \alpha z_j \quad \text{for } j = 1, 2, \ldots, n.$$

PVGS algorithm:
Input: $n, q, k, \{1, \alpha\}, e(x), r, a, t$, n distinct elements $x_1, x_2, \ldots, x_n \in \mathbb{F}_q$, s.
Step 1: Given a received vector $(r_{PV_1}, r_{PV_2}, \ldots, r_{PV_n})$ over $\mathbb{F}_{q^2}$, apply the Parvaresh-Vardy algorithm to find lists for the polynomials $f_0(x)$ and $g_0(x)$. **Step 2:** For each pair $(f_0(x), g_0(x))$ apply the Guruswami-Sudan algorithm on the vector $\mathbf{r}_{GS}$ to find lists for the polynomials $f_1(x)$ and $g_1(x)$. **Step 3:** For each possible $F(x) = f_0(x) + e(x)f_1(x)$ and associated $G(x) = g_0(x) + e(x)g_1(x)$ compute the codeword $\mathbf{c}_{PV}$ as in (6.4) and check if $c_{PV_i} = r_{PV_i}$ for at least t values $i \in [n]$. If so, include $f_0(x), g_0(x), f_1(x), g_1(x)$ in the output list.
Output: Polynomials $f_0(x), g_0(x), f_1(x), g_1(x)$ according to Step 2.

Table 6.1: Presentation of the PVGS algorithm

Thus, we can set up the interpolation set $P = \{(x_i, y_i, z_i)\}_{i=1}^{n}$ exactly as in (5.5). We now need to find the polynomial $Q(x, y, z)$ with the three constraints given in the first step of table 5.1. The first thing to adapt is the considered weighted degree of the sought polynomial Q. The idea of the weighted degree is to "contain" the future degree in x of the polynomial $Q(x, f(x), g(x))$ where $f(x)$ and $g(x)$ are the message polynomials. Here the messages $F(x)$ and $G(x)$ have degree at most $k + s - 1$. Thus the weighted degree for the PVGS algorithm is defined by

$$\text{wtdeg}(x^a y^b z^c) = a + (k + s - 1)b + (k + s - 1)c.$$

The computation of the polynomial Q still amounts to solving a linear system. Nevertheless we need to adapt the conditions ensuring that this linear system must have a non-zero solution. In particular, we want to find a bound for l, the smallest possible weighted degree of Q such that there surely exists a solution to the homogenous system. The number of constraints does not change and is still

$$\frac{n(r+2)(r+1)r}{6}.$$

The number of unknowns is given by

$$\sum_{i=0}^{l} \sum_{j=0}^{\lfloor \frac{l-i}{k+s-1} \rfloor} \sum_{k=0}^{\lfloor \frac{l-i-(k+s-1)j}{k+s-1} \rfloor} 1. \tag{6.5}$$

Simplifying this equation we obtain a lower bound, namely

$$(6.5) > \frac{l^3}{6(k+s-1)^2}.$$

In order to guarantee the existence of Q the number of constraints must be smaller than the number of unknowns, namely

$$\frac{l^3}{6(k+s-1)^2} > \frac{n(r+2)(r+1)r}{6}.$$

Thus, we know that there exists a polynomial Q of degree at most l that has a zero of multiplicity r at each point $\{(x_i, y_i, z_i)\}_{i=1}^n$ where

$$l = \left\lceil \sqrt[3]{n(k+s-1)^2 r(r+1)(r+2)} \right\rceil.$$

We now move on to the adjustment of lemma 5.1 and its proof to establish under what conditions the polynomial $p(x) := Q(x, F(x), G(x))$ is identically zero. To this end, we need to find conditions guaranteeing that $p(x)$ has more zeros than its degree. The degree of $p(x)$ is at most l since the degree of F and G is at most $k+s-1$. On the other hand, for every integer j such that $F(x_j) = y_j$ and $G(x_j) = z_j$, the polynomial $p(x)$ has a zero of multiplicity r at the point $(x_j, F(x_j), G(x_j))$. Thus, with the definition of the number of agreements t given by the adaptation of (5.10):

$$t := \left|\{j : F(x_j) = y_j \text{ and } G(x_j) = z_j\}\right|,$$

the number of zeros of $p(x)$ is at least rt. Consequently if $rt > l$, i.e., if

$$rt > \left\lceil \sqrt[3]{n(k+s-1)^2 r(r+1)(r+2)} \right\rceil,$$

then $p(x)$ must be the zero polynomial. Therefore if

$$t \geq \left\lceil \sqrt[3]{n(k+s-1)^2 \left(1+\frac{1}{r}\right)\left(1+\frac{2}{r}\right)} + \frac{1}{r} \right\rceil,$$

then $p(x) \equiv 0$. Lemma 5.2 obviously continues to hold for the PVGS algorithm, with the difference that now the maximum number of positions in which the received vector $\mathbf{r}_{PV}$ and the true codeword $\mathbf{c}_{PV}$ can differ is

$$e_1 = n - t = \left\lfloor n - \sqrt[3]{n(k+s-1)^2 \left(1+\frac{1}{r}\right)\left(1+\frac{2}{r}\right)} - \frac{1}{r} \right\rfloor. \tag{6.6}$$

Thus if $\mathbf{r}_{PV}$ and $\mathbf{c}_{PV}$ differ in at most e_1 positions, then $Q(x, F(x), G(x)) \equiv 0$. Note that, as of yet, there are no crucial differences from the Parvaresh-Vardy algorithm. In fact, we have not yet used the fact that F and G are

constructed from two polynomials.

It remains to show how to recover the polynomials F and G (actually we recover only f_0 and g_0 in Step 1 of the PVGS algorithm as precised before). The first time we will use the polynomial $e(x)$ is in the reduction modulo $e(x)$ of Q. However, until now, the particularities of F and G are still not taken into account. We then obtain

$$P(y,z) := Q(x,y,z) \bmod e(x).$$

For the same reason as in chapter 5 $P(y,z)$ is not identically zero and thechnical lemma 5.3 continues to hold. The first significant difference comes in the adaptation of lemma 5.4 because of the reduction modulo $e(x)$. Let us write the new lemma.

Lemma 6.1. *Suppose that $Q(x,F(x),G(x)) \equiv 0$, and let β and γ denote the elements of $\mathbb{K}$ that corresponds to $F(x)$ and $G(x)$, respectively. Then $P(\beta,\gamma) = 0$.*

Proof: First observe that β corresponds to $f_0(x)$ and γ to $g_0(x)$ by definition of F and G ((6.2),(6.3)). If $Q(x,F(x),G(x)) \equiv 0$, then by lemma 5.3 it can be written in the form

$$A(x,y,z)(y-F(x)) + B(x,y,z)(z-F(x)) =$$

$$A(x,y,z)y - A(x,y,z)f_0(x) - A(x,y,z)e(x)f_1(x) +$$
$$B(x,y,z)z - B(x,y,z)g_0(x) - B(x,y,z)e(x)g_1(x)$$

for some polynomials $A(x,y,z), B(x,y,z) \in \mathbb{F}_q[x,y,z]$. Therefore, since $P(y,z)$ is a reduction modulo $e(x)$ of Q, it can be written in the form

$$\begin{aligned} P(y,z) &= \tilde{A}(y,z)(y-f_0(x)) + \tilde{B}(y,z)(z-g_0(x)) \\ &= \tilde{A}(y,z)(y-\beta) + \tilde{B}(y,z)(z-\gamma) \end{aligned} \quad (6.7)$$

where $\tilde{A}(y,z)$ and $\tilde{B}(y,z)$ in $\mathbb{K}[y,z]$ are the remainders of $A(x,y,z)$ and $B(x,y,z)$ upon division by $e(x)$. It is now obvious from (6.7) that $P(\beta,\gamma) = 0$.

□

By lemma 6.1 and the known property $g_0(x) = (f_0(x))^a \bmod e(x)$, i.e., $\gamma = \beta^a$ in $\mathbb{K}$, we have that β is a root of $H(y) = P(y,y^a)$. $H(y)$ is not the zero polynomial, the adaptation of lemma 5.5 is straightforward. Then we can find the root of the univariate polynomial $H(y)$. Thus, with the Parvaresh-Vardy algorithm, we obtain a list of f_0 and corresponding g_0 in Step 1 of the PVGS algorithm. However, observe that we cannot check the

number of agreements between $\mathbf{c}_{PV}$ and $\mathbf{r}_{PV}$ since we do not know f_1 and g_1. Thus the list will contain the original f_0 and g_0 but also polynomials that evaluate to a codeword at distance more than e_1 (6.6) from the received vector $\mathbf{r}_{PV}$. Thus, Theorem 5.6 is adapted to our case as follows.

Theorem 6.2. *Given a vector $\mathbf{r}_{PV}$ over $\mathbb{F}_{q^2}$ which differs from the original codeword $\mathbf{c}_{PV}$ constructed as in (6.4) in at most*

$$e_1 = \left\lfloor n - \sqrt[3]{n(k+s-1)^2\left(1+\frac{1}{r}\right)\left(1+\frac{2}{r}\right)} - \frac{1}{r}\right\rfloor$$

positions, the Parvaresh-Vardy algorithm applied in Step 1 of the PVGS algorithm outputs in polynomial time a list of polynomials $f_0(x), g_0(x) \in \mathbb{F}[x]_{<k}$ such that the list contains the polynomials f_0 and g_0 that evaluate to the codeword $\mathbf{c}_{PV}$ for some polynomials $f_1(x), g_1(x) \in \mathbb{F}_q[x]_{<s}$, where $r \geq 1$ is an arbitrary multiplicity parameter. The size of the list is at most L^2 , where

$$L = \left\lceil r\sqrt[3]{\frac{n\left(1+\frac{1}{r}\right)\left(1+\frac{2}{r}\right)}{k+s-1}}\right\rceil + 1.$$

The proof follows immediately from the proof of theorem 5.6 using the adapted lemmas.

6.1.2 Step 2, Guruswami-Sudan algorithm

We have used the Parvaresh-Vardy algorithm adapted to our new code to find possible f_0 and g_0. We need now to find a way to recover $f_1(x)$ and $g_1(x)$. We will only explain the steps for $f_1(x)$. The resolution is the same for $g_1(x)$, it suffices to replace $F(x)$ by $G(x)$, $f_1(x)$ by $g_1(x)$, $f_0(x)$ by $g_0(x)$ and y_j by z_j. To this end, we will use the Guruswami-Sudan algorithm. Let us recall the issue. We know the received vector $\mathbf{r}_{PV}$ and its components $\mathbf{y}$ and $\mathbf{z}$. Moreover we know a list of possible f_0's and corresponding g_0's thanks to the previous application of the Parvaresh-Vardy algorithm. This list contains the original pair (f_0, g_0) by theorem 6.2. Finally, we know that $F(x) = f_0(x) + e(x)f_1(x)$. Thus we have

$$f_1(x_j) = \frac{F(x_j) - f_0(x_j)}{e(x_j)}, \forall j = 1, 2, \ldots, n. \tag{6.8}$$

Substituting the unknown $F(x_j)$'s by the known y_j's we get

$$r_{GS_j} := \frac{y_j - f_0(x_j)}{e(x_j)}, \forall j = 1, 2, \ldots, n. \tag{6.9}$$

Thus we do not know the true values of

$$(f_1(x_1), f_1(x_2), \dots, f_1(x_n)) \tag{6.10}$$

but we can compute $\mathbf{r}_{GS} = (r_{GS_1}, \dots, r_{GS_n})$ which differs from the vector (6.10) in at most e_1 positions, since the errors come from $y_1, \dots, y_n$ and by our hypothesis in Step 1 $(F(x_1), \dots, F(x_n))$ differs from $\mathbf{y} = (y_1, \dots, y_n)$ in at most e_1 (6.6) positions.

However, we have still to prove that we can use the Guruswami-Sudan algorithm. Observe that we actually have a Reed-Solomon code $\mathcal{C}_2$ over $\mathbb{F}_q$. The message polynomial is $f_1(x) \in \mathbb{F}_q[x]_{<s}$ and the codeword $\mathbf{c}_{GS} = (c_{GS_1}, c_{GS_2}, \dots, c_{GS_n}) \in \mathcal{C}_2$ corresponding to this message is given by the evaluation of the message polynomial at the n points $x_1, \dots, x_n$:

$$c_{GS_j} = f_1(x_j) = \frac{F(x_j) - f_0(x_j)}{e(x_j)} \quad \text{for } j = 1, 2, \dots, n.$$

Thus we have a Reed-Solomon code with parameters $[n, s, d_{C_2} = n - s + 1]$. Hence the rate of this code is $R_2 = \frac{s}{n}$. In our case $\mathbf{c}_{GS}$ is the true codeword, $\mathbf{r}_{GS}$ is the received vector. We can now apply the Guruswami-Sudan algorithm with input parameters $n, k = s$ and the points $\{(x_i, r_{GS_i})\}_{i=1}^{n}$. This algorithm will give us a list of all polynomials $f_1(x)$ such that $f_1(x_i) \neq r_{GS_i}$ for at most $e_2 = n - 1 - \lfloor \sqrt{n(s-1)} \rfloor$ values i from $[n]$ as showed in chapter 4.

With the Guruswami-Sudan algorithm we obtain a list for $f_1(x)$ and for $g_1(x)$ in Step 2 of the PVGS algorithm. Each list is associated with a $f_0(x)$ since we apply the algorithm once for each f_0 found in Step 1. Note that there exists a "correspondence" between $f_0(x)$ and $g_0(x)$ by the relation (6.1) and then we can define the correspondence between the list for $f_1(x)$ and $g_1(x)$ as follows:
A list of $f_1(x)$'s corresponds to a list of $g_1(x)$'s if they were found with the same (f_0, g_0) pair.

6.2 Error bound

Let us sum up what we have achieved so far by giving the main theorem of this section.

Theorem 6.3. *The PVGS algorithm with inputs as in table 6.1 can correct a fraction of errors up to*

$$\min\left\{\tau_1 := 1 - \sqrt[3]{\left(\frac{k+s-1}{n}\right)^2}, \tau_2 := 1 - \sqrt{\frac{s-1}{n}}\right\}.$$

Proof: An unknown codeword $\mathbf{c}_{PV}$ (constructed from polynomials f_0, g_0, f_1, g_1) was sent and we have received a vector $\mathbf{r}_{PV}$, we have then found a list for possible $(f_0(x), g_0(x))$ pairs with the conditions that $\mathbf{r}_{PV}$ differs from $\mathbf{c}_{PV}$ in at most

$$e_1 = \left\lfloor n - \sqrt[3]{n(k+s-1)^2\left(1+\frac{1}{r}\right)\left(1+\frac{2}{r}\right)} - \frac{1}{r} \right\rfloor$$

positions. Then, for each $f_0(x)$ and corresponding $g_0(x)$, we have found a list for $f_1(x)$ and $g_1(x)$ with the conditions that the vector $\mathbf{r}_{GS}$, constructed from $\mathbf{r}_{PV}$, differs from $(f_1(x_1), \ldots, f_1(x_n))$ in at most

$$e_2 = n - 1 - \lfloor \sqrt{n(s-1)} \rfloor$$

positions.

Thus, we can correct at most e_1 errors in the first step of table 6.1 and e_2 in the second one. Since the e_1 errors are reflected on the vector $\mathbf{r}_{GS}$ by $\mathbf{r}_{PV}$ and since there is no "new" errors occuring in the second step, the PVGS algorithm can correct

$$\min\{e_1, e_2\} \text{ errors.}$$

In term of fraction of errors, we obtain asymptotically that we can correct a fraction of errors up to

$$\min\left\{\tau_1 = 1 - \sqrt[3]{\left(\frac{k+s-1}{n}\right)^2}, \tau_2 = 1 - \sqrt{\frac{s-1}{n}}\right\}.$$

□

6.3 Size of the output list

There remains to discuss briefly the question of the size of the list. The size is obviously polynomial bounded since the size of the list for the two used algorithms in the decoding steps are polynomial bounded. Recall that in the first step of the PVGS algorithm, the Parvaresh-Vardy gives us a list of $(f_0(x), g_0(x))$ pairs whose size is at most (cf theorems 5.6 and 6.2)

$$L_{PV} = \left\lceil r\sqrt[3]{\frac{n\left(1+\frac{1}{r}\right)\left(1+\frac{2}{r}\right)}{k+s-1}} \right\rceil + 1,$$

while there are at most

$$L_{GS} = \frac{n(t-(s-1)))}{t^2-(s-1)n}, \; t \geq \lfloor \sqrt{n(s-1)} \rfloor + 1$$

polynomials in the output list of the Guruswami-Sudan algorithm (cf theroem 4.2). Moreover, for each pair (f_0, g_0) and each f_1 there are L_{GS} solutions for $G(x)$ (one for each g_1 in the list associated to f_1). Thus the size of the output list of the PVGS algorithm is at most

$$L_{PV} \cdot L_{GS}^2$$

which is polynomial in n and r.

6.4 Comparison with the previous algorithms

It remains to show whether the PVGS algorithm can really correct more errors than the previous ones. We have shown that it can correct a fraction of errors up to

$$\min \left\{ \tau_1 = 1 - \sqrt[3]{\left(\frac{k+s-1}{n}\right)^2}, \tau_2 = 1 - \sqrt{\frac{s-1}{n}} \right\}. \tag{6.11}$$

Assume we have a Parvaresh-Vardy code of rate $R = \frac{k+2s}{2n}$ for fixed k, n and s. With the Parvaresh-Vardy algorithm in the case $M = 2$, we can correct a fraction of errors up to $\tau_{PV} = 1 - \sqrt[3]{(2R)^2}$. Let us compare with τ_1. First, observe that

$$2R - \frac{k+s-1}{n} = \frac{k+2s}{n} - \frac{k+s-1}{n} = \frac{s+1}{n}.$$

It implies that $2R > \frac{k+s-1}{n}$ and so $\tau_{PV} < \tau_1 \ \forall R$.
On the other hand, $R = \frac{k+2}{2n} + \frac{s-1}{n}$ hence $R > \frac{s-1}{n}$ and so $1 - \sqrt{R} = \tau_{GS} < \tau_2$ $\forall R$. Moreover, we have two constraints to satisfy, here they are:

1. $s > 0$.

2. $k + s - 1 \leq n$ to make sure that e_1 is positive.

This proves the following lemma.

Lemma 6.4. *If $\tau_1 = \tau_2$ in (6.11), if $s > 0$ and if $k + s - 1 \leq n$, the PVGS algorithm can correct up to $\tau_1 = \tau_2$ errors. Moreover, in this case the PVGS algorithm is better than both the Guruswami-Sudan and Parvaresh-Vardy algorithms for all rates.*

We have shown that the PVGS algorithm is better than the previous algorithms if $\tau_1 = \tau_2$. We have still to show that this equality happens. To simplify, let us define $\sigma = \frac{s}{n}$. Moreover when n becomes large, we can consider that $\tau_1 = 1 - \sqrt[3]{((k+s)/n)^2}$ and $\tau_2 = 1 - \sqrt{s/n}$. Thus $\tau_1 = \tau_2$ if and only if

$$(2R - \sigma)^{2/3} - \sigma^{1/2} = 0. \tag{6.12}$$

The constraints become

$$\sigma > 0 \quad (6.13)$$
$$\sigma \geq 2R - 1 \quad (6.14)$$

Solving equation (6.12), we find an expression for σ (not reproduced here). Moreover the solution always satisfies the constraints (6.13) and (6.14). Figure 6.1 shows the solutions σ as a function of the rate R and allows also to observe the difference $k/2n$ between R and σ. Since the equation (6.12) under constraints (6.13) and (6.14) has a solution σ for all rates $R \in (0, 1)$ then the PVGS algorithm is better than the Guruswami-Sudan and the Parvaresh-Vardy algorithms for all rates $R \in (0, 1)$ by lemma 6.4.

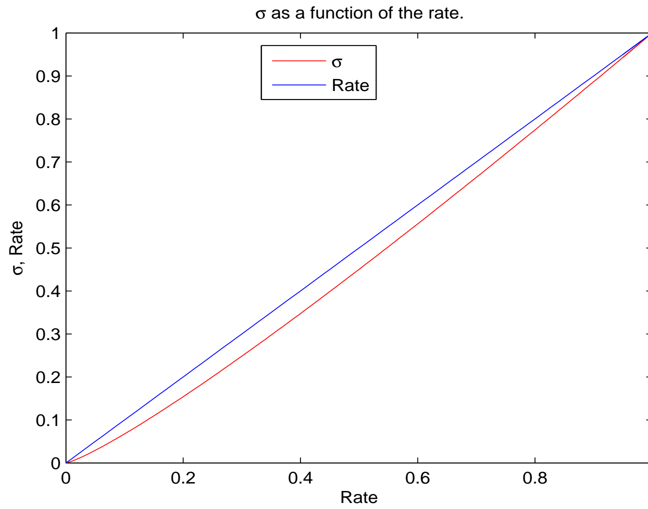

Figure 6.1: σ such that $\tau_1 = \tau_2$

This most interesting result is presented in Figure 6.2 that compares the three algorithms, Parvaresh-Vardy, Guruswami-Sudan and PVGS. First, notice that the Parvaresh-Vardy is better than the Guruswami-Sudan algorithm for rates $R < 1/16$ as explained before. For $R > 0.5$ the Parvaresh-Vardy algorithm becomes not relevant as mentioned in section 5.2. Finally the PVGS algorithm is cleary better than the other algorithms for all $R \in (0, 1)$.

Thus, we have shown that the PVGS algorithm is better than the previous ones if $\tau_1 = \tau_2$. However, there may be a point at which $\tau_1 \neq \tau_2$ and at which $\tau_i = \min\{\tau_1, \tau_2\}$ is larger than τ_{GS} and τ_{PV}. What we explain here is the fact that the equation we have to solve is in fact: find σ which maximizes the minimum of $\{\tau_1, \tau_2\}$. Indeed we look for σ that maximizes the minimum

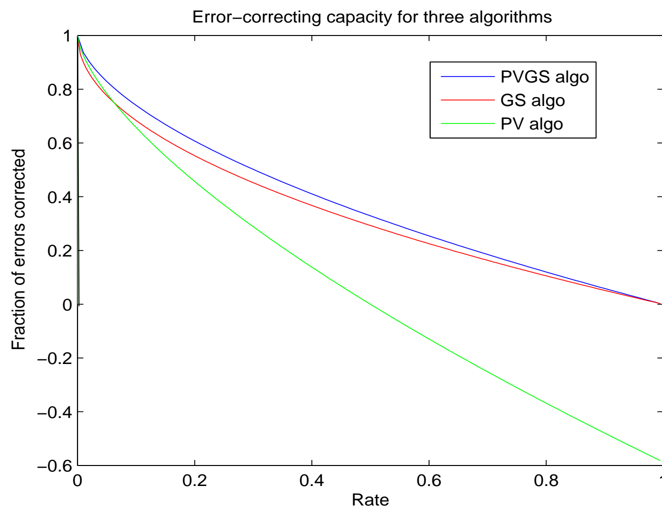

Figure 6.2: Comparison of the error-correcting capacity plotted against the rate of the code for three algorithms.

of the two curves $f = (2R - \sigma)^{2/3}$ and $g = \sigma^{1/2}$. Observe that

$$\frac{\mathrm{d}}{\mathrm{d}\sigma} f = -\frac{2}{3}(2R - \sigma)^{-1/3} < 0$$

since $\sigma \leq R$ and $R > 0$. For g we obtain

$$\frac{\mathrm{d}}{\mathrm{d}\sigma} g = \frac{1}{2\sqrt{\sigma}} > 0.$$

Since f is a strictly increasing function and g a stricly decreasing function and since there exists an intersection between the curves the σ that maximizes the minimum is definitely the one at which $\tau_1 = \tau_2$.

Theorem 6.3 and lemma 6.4 show that the PVGS algorithm is better than the Parvaresh-Vardy and Guruswami-Sudan algorithms for the case $M = 2$. The following section explores the other possible cases, notably the cases $M > 2$.

6.5 PVGS algorithm, general case

6.5.1 Cases $M > 2$

Let us observe what happens if $M > 2$. We have now $F(x), G_1(x), \ldots, G_{M-1}(x)$ defined by

$$\begin{aligned} F(x) &= f_0(x) + e(x)f_1(x) \\ G_1(x) &= g_0(x) + e(x)f_2(x) \\ &\vdots \\ G_{M-1}(x) &= g_{M-1} + e(x)f_M(x) \end{aligned}$$

with

$$\begin{aligned} g_i(x) &= (f_0(x))^{a_i} \bmod e(x) \text{ for } i = 0, \ldots, M-1 \\ \deg f_i &< s \text{ for } i = 1, \ldots, M \\ \deg f_0 &< k \\ \deg e &= k \\ a_i &:= \sum_{j=0}^{i} \left\lceil \sqrt[M+1]{\frac{n}{k+s-1} \prod_{l=0}^{M} (r+l)} \right\rceil^j . \end{aligned}$$

The codeword $\mathbf{c}_{PV} = (c_{PV_1}, \ldots, c_{PV_n})$ is given by

$$c_{PV_j} = f(x_j) + \sum_{i=1}^{M-1} \alpha_i G_i(x_j) \quad \text{for } j = 1, \ldots, n,$$

where x_j and α_i are defined as in section 5.3. The rate of this code is $R_1 = \frac{k+Ms}{Mn}$. The next steps of coding, interpolation, decoding are straightforward generalization of the chapter 5 and this chapter with $M = 2$. Thus, we do not prove all lemmas again and we straight observe the final result. We obtain that the PVGS algorithm with $M > 2$ can correct a fraction of errors up to

$$\min\{\tau_1 = 1 - \sqrt[M+1]{\left(\frac{k+s-1}{n}\right)^M}, \tau_2 = 1 - \sqrt{\frac{s-1}{n}}\}.$$

With the same explanations as before, $\tau_1 > \tau_{PV} = 1 - \sqrt[M+1]{(MR)^M}$ and $\tau_2 > \tau_{GS} = 1 - \sqrt{R}\ \forall R$. Defining $\sigma = s/n$ we get that the PVGS algorithm for $M > 2$ can correct a fraction of errors up to

$$\min\{\tau_1 = 1 - \sqrt[M+1]{(MR - (M-1)\sigma)^M}, \tau_2 = 1 - \sqrt{\sigma}\}.$$

We have seen in the chapter 5 that the Parvaresh-Vardy algorithm for $M > 2$ improves the case $M = 2$ for very few rates. Here the situation is the same. The PVGS algorithm with $M > 2$ does not bring significant improvements over the case $M = 2$. Figure 6.3 shows the error-correcting capacity for $M = 2, 5, 10, 100$ on the interval $R = [0, 0.05]$. For $R > 0.05$, the curves continue decreasing. We observe that the case $M = 5$ brings improvements for rates up to ~ 0.018 while $M = 10$ brings improvements for rates up to ~ 0.005 and $M = 100$ brings so few improvements that they are not observable on figure 6.3.

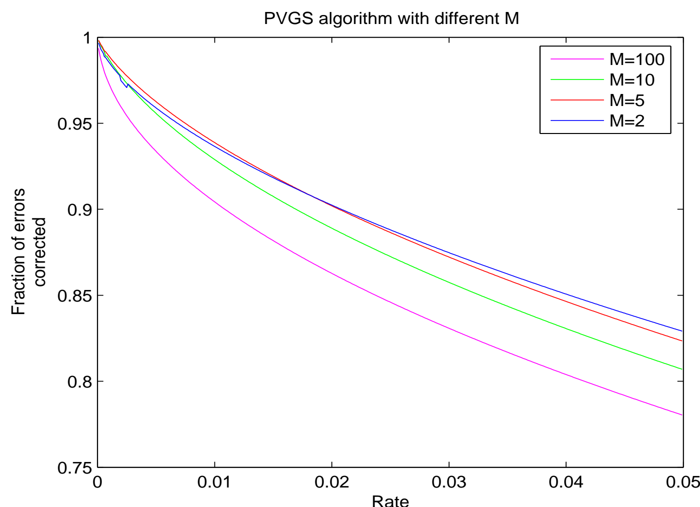

Figure 6.3: Comparison of the error-correcting capacity plotted against the rate of the code for PVGS algorithm in cases $M = 2, 5, 10, 100$.

6.5.2 Other cases

Instead of increasing M we could increase the degree of $F(x)$ and $G(x)$. The idea here is to iterate the PVGS algorithm. Let us define more precisely the new $F(x)$ and $G(x)$.

$$F(x) = f_0(x) + f_1(x)e(x) + f_2(x)e(x)^2,$$

$$G(x) = g_0(x) + g_1(x)e(x) + g_2(x)e(x)^2,$$

where

$$f_0(x) \in \mathbb{F}_q[x]_{<k}, \quad f_1(x) \in \mathbb{F}_q[x]_{<s_1}, \quad f_2(x), g_2(x) \in \mathbb{F}_q[x]_{<s_2},$$

$$g_0(x) = (f_0(x))^a \bmod e(x), \quad g_1(x) = (f_1(x))^a \bmod e(x)$$

and $s_1 < k$. $a, e(x)$ and all the others parameters are as in section 6.1. The codeword $\mathbf{c_{PV}} = (c_{PV_1}, \ldots, c_{PV_n})$ is simply defined by

$$c_{PV_j} = F(x_j) + \alpha G(x_j).$$

The decoder receives the vector $\mathbf{r_{PV}} = (r_{PV_1}, \ldots, r_{PV_n})$ and decomposes it such that

$$r_{PV_j} = y_j + \alpha z_j \quad \text{for } j = 1, 2, \ldots, n.$$

The rate of this code is $R = \frac{k+s_1+2s_2}{2n}$. The considered weighted degree is now $(1, 2k+s_2-1, \ldots, 2k+s_2-1)$. We can then apply the Parvaresh-Vardy algorithm and follow a similar way to section 6.1 to find list for f_0 and g_0. It can correct a fraction of errors up to

$$\tau_1 = 1 - \sqrt[3]{\left(\frac{2k+s_2-1}{n}\right)^2}. \tag{6.15}$$

Defining $F_1(x) = f_1(x) + f_2(x)e(x)$ and $G_1(x) + g_1(x) + g_2(x)e(x)$ observe that we have

$$F_1(x_j) = \frac{F(x_j) - f_0(x_j)}{e(x_j)}, \quad G_1(x_j) = \frac{G(x_j) - g_0(x_j)}{e(x_j)}, \quad j = 1, 2, \ldots, n.$$

Replacing $F(x_j)$ and $G(x_j)$ by y_j and z_j respectively, consider the new codeword $\mathbf{c'_{PV}} = (c'_{PV_1}, \ldots, c'_{PV_n})$ defined by

$$c'_{PV_j} = u_j + \alpha v_j \; j = 1, 2, \ldots, n,$$

where

$$u_j = \frac{y_j - f_0(x_j)}{e(x_j)}, \quad v_j = \frac{z_j - f_0(x_j)}{e(x_j)}.$$

Thus, we have all we need to apply the Parvaresh-Vardy algorithm one more time. It allows us to obtain lists for f_1 and g_1. Notice that we use the fact that $s_1 < k$ to make sure that $e(x)$ does not divide $f_1(x)$ and $g_1(x)$, which is necessary in adaption of lemma 6.1. In this step we can correct a fraction of errors up to

$$\tau_2 = 1 - \sqrt[3]{\left(\frac{k+s_2-1}{n}\right)^2}.$$

The third and last step is the application of the Guruswami-Sudan algorithm to find f_2 and g_2 exactly as done in section 6.1 for f_1 and g_1. We can correct here a fraction of errors up to

$$\tau_3 = 1 - \sqrt{\frac{s_2-1}{n}}. \tag{6.16}$$

Finally the PVGS algorithm in this case can correct up to

$$\min\{\tau_1, \tau_2, \tau_3\}.$$

Since $\tau_1 < \tau_2$ we only need to consider the minimum of τ_1 and τ_3. Recall that the constraints are

1. $s_1 < k$
2. $s_1, s_2 > 0$
3. $2k + s_2 - 1 \leq n$.

Solving this problem we obtain the same results as for the case in section 6.1. More precisely, defining $\sigma_1 := s_1/n, \sigma_2 := s_2/n$, and writing $R_{L_1} = \frac{k_{L_1}+2s}{2n}$ for the rate of the code in section 6.1 and $R_{L_2} = \frac{k_{L_2}+s_1+2s_2}{2n}$ for the code of this section, we obtain that

$$\sigma = \sigma_2,$$

$$\frac{k_{L_1}}{2n} = \sigma_1,$$

$$\frac{k_{L_2}}{2n} = \frac{\sigma_1}{2}.$$

Thus a code of section 6.1 of rate R_{L_1} and a code of this section with the same rate R_{L_2} can correct the same fraction of errors. We can actually check with the three equalities above that

$$(6.16) = 1 - \sqrt{\sigma}$$

and

$$(6.15) = 1 - \sqrt[3]{(2R_{L_1} - \sigma)^2}.$$

Thus the PVGS algorithm with the maximum power of $e(x) > 1$ does not bring improvements.

6.6 Conclusion

The PVGS algorithm can correct a larger fraction of errors than the Guruswami-Sudan and Parvaresh-Vardy algorithms, as shown by Theorem 6.3 and lemma 6.4. With section 6.5 we can conclude that the PVGS algorihtm can be used in practice with $M = 2$ and maximum power of $e(x)$ equals to 1 since the other cases do not bring significant improvements. This is the case that we have implemented.

Chapter 7

Soft-decision decoding algorithm

In this chapter we will briefly present another kind of decoding algorithms. In the previous chapters, we have considered adversarial noise, i.e., the errors can be in any positions. What does it mean? The considered channels were the $q-$ary symmetric memoryless channels (def. 1.4). The input and output alphabets $\mathcal{V}_X$ and $\mathcal{V}_Y$ were identical. For example if the input alphabet was $\mathbb{F}_2$ the receveid vector contained only 0's and 1's. It could not contain 0.4 or 27. The soft-decision decoding algorithms utilizes the probabilistic reliability information concerning the received symbols. For example consider the memoryless channel as in figure 7.1 with a Gaussian noise. In this case, the input alphabet $\mathcal{V}_X$ is different from the output alphabet $\mathcal{V}_Y$. If $\mathcal{V}_X = \mathbb{F}_2$ we send -1 (for 1) and 1 (for 0), but we can receive 0.1 or 0.3 since we receive $y = x + z,\ x \in \mathcal{V}_X, y \in \mathcal{V}_Y, z \sim \mathcal{N}(0, \sigma^2)$. Of course we know that the sent message was not $(0.1, 0.3, \ldots)$, however knowing the properties of the channel we can use such information to retrieve the original codeword.

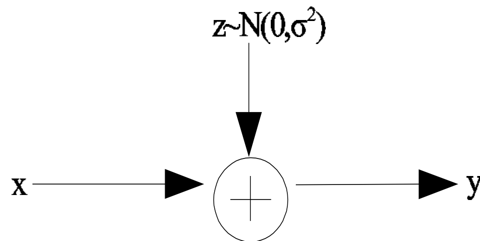

Figure 7.1: A channel with a Gaussian noise.

7.1 The Koetter-Vardy algorithm

In this section we present the Koetter-Vardy algorithm [14], which is a soft-decision decoding algorithms for Reed-Solomon codes. This algorithm proceeds as for the Guruswami-Sudan algorithm for the interpolation and factorization steps. The main difference is in the choices of the multiplicity and the sets of points at which Q must be zero. In the Guruswami-Sudan algorithm, all points have the same multiplicity r and we choose this multiplicity arbitrarily. In [1], Guruwami and Sudan also present a weighted version of their list-decoding algorithm (in which the points have not all the same multiplicity) however the multiplicities are still chosen arbitrarily. In their algorithm, Koetter and Vardy will use probabilistic reliability information of the channel to choose the set of points and their multiplicities in a optimal way.

We can now present the Koetter-Vardy algorithm. We have a memoryless channel (cf definition 1.4) with a finite input alphabet $\mathcal{V}_X$, an output alphabet $\mathcal{V}_Y$ and the $|\mathcal{V}_X|$ probability-mass functions (or probability-density functions in the continuous case)

$$f(\cdot|x) : \mathcal{V}_Y \to \mathbb{R} \quad \text{for all } x \in \mathcal{V}_X.$$

We think of channel input and output as random variables $\mathcal{X}$ and $\mathcal{Y}$, respectively. Moreover we assume that $\mathcal{X}$ is uniformly distributed over $\mathcal{V}_X$. The decoder can easily compute the probability that $\alpha \in \mathcal{V}_X$ was transmitted given that $y \in \mathcal{V}_Y$ was received as follows:

$$\Pr(\mathcal{X} = \alpha | \mathcal{Y} = y) = \frac{f(y|\alpha)\Pr(\mathcal{X} = \alpha)}{\sum_{x \in \mathcal{V}_X} f(y|x)\Pr(\mathcal{X} = x)} = \frac{f(y|\alpha)}{\sum_{x \in \mathcal{V}_X} f(y|x)}. \tag{7.1}$$

We used here the Bayes formula and the assumption that $\mathcal{X}$ is uniform. For Reed-Solomon codes, the input alphabet is always $\mathcal{V}_X = \mathbb{F}_q$. In the remainder of this chapter, we will use the fixed ordering $\alpha_1, \alpha_2, \ldots, \alpha_q$ of the elements of $\mathbb{F}_q$. Given the received vector $\mathbf{y} = (y_1, \ldots, y_n) \in \mathcal{V}_Y^n$ we compute

$$\pi_{ij} := \Pr(\mathcal{X} = \alpha_i | \mathcal{Y} = y_j), \quad \text{for } i = 1, \ldots, q \text{ and } j = 1, \ldots, n \tag{7.2}$$

according to the expression in (7.1).

Definition 7.1 (Reliability matrix). *Let Π be the $q \times n$ matrix with entries π_{ij} defined in (7.2). Π is the reliability matrix and we assume that Π is the input of the soft-decision decoding algorithm. We will sometimes write $\Pi(\alpha, j)$ to refer to the entry found in the jth column of Π in the row indexed by $\alpha \in \mathbb{F}_q$.*

Definition 7.2 (Multiplicity matrix). *A multiplicity matrix is a $q \times n$ matrix M with nonnegative integer entries m_{ij}. The multiplicity of the point (x_j, α_i) is m_{ij}.*

In the Guruswami-Sudan algorithm, the considered set of points was $\{(x_j, y_j)\}_{j=1}^n \in \mathbb{F}_q \times \mathbb{F}_q$ and the multiplicity of each point was r. Thus, the multiplicity matrix in that case has entries $m_{ij} = r$ for i such that $y_j = \alpha_i$ and zero otherwise. In the Koetter-Vardy algorithm, we have to construct a multiplicity matrix M and the idea is to give larger multiplicity to "the most reliable" points using the reliability matrix. We discuss this step in the next section. Once M is computed, the decoder proceeds as in the Guruswami-Sudan algorithm. The decoder looks for a bivariate polynomial $Q(x, y)$ of minimal $(1, k-1)$-weighted degree that has a zero of multiplicity m_{ij} at the point (x_j, α_i) for every i, j such that $m_{ij} \neq 0$. The final step is the factorization step as described in table 4.1, Step 2.

We will now find under what conditions the decoder produces the original codeword $\mathbf{c}$, for a given multiplicity matrix. The ideas are still the same. There must be more unknowns than constraints to find a non identically zero polynomial Q and $Q(x, f(x))$ must have more zeroes than its degree or must be divisible by a polynomial of higher degree than its own to make sure that $f(x)$ (the polynomial that evaluates to $\mathbf{c}$) will be solution. We need a first definition.

Definition 7.3 (Cost)**.** *Given a multiplicity matrix M, we define the cost of M as follows:*

$$C(M) := \frac{1}{2} \sum_{i=1}^{q} \sum_{j=1}^{n} m_{ij}(m_{ij} + 1).$$

Observe that $C(M)$ is exactly the number of constraints on the coefficients of $Q(x, y)$ since a given zero of multiplicity m imposes $m(m+1)/2$ linear constraints. As in the previous chapters, we can always find a non-zero solution $Q(x, y)$ if the $(1, k-1)-$weighted degree Δ is large enough, namely if

$$N_{1,k-1}(\Delta) > C(M)$$

where N_{w_x,w_y} is the number of monomials of $(w_x, w_y)-$weighted degree at most Δ. Thus we define the function

$$l_{w_x,w_y}(\nu) := \min\{\Delta \in \mathbb{Z} : N_{w_x,w_y}(\Delta) > \nu\}. \tag{7.3}$$

Now we define the inner product of two matrices.

Definition 7.4 (Inner product)**.** *Given two matrices A and B over the same field, we define the inner product*

$$\langle A, B, \rangle := \operatorname{trace}(AB^T) = \sum_{i=1}^{q} \sum_{j=1}^{n} a_{ij} b_{ij}.$$

Finally it will be convenient to think of the codewords of the Reed-Solomon code as $q \times n$ matrices over the reals. Specifically, any vector $\mathbf{v} = (v_1, \ldots, v_n)$ over $\mathbb{F}_q$ can be represented by the $q \times n$ real-valued matrix $[\mathbf{v}]$ defined as follows: $[\mathbf{v}]_{ij} = 1$ if $v_j = \alpha_i$, and $[\mathbf{v}]_{ij} = 0$ otherwise. We have one more definition.

Definition 7.5 (Score)**.** *The score of a vector* $\mathbf{v} = (v_1, \ldots, v_n)$ *over* $\mathbb{F}_q$ *with respect to a given multiplicity matrix* M *is defined as the inner product* $S_M(\mathbf{v}) = \langle M, [\mathbf{v}] \rangle$.

To understand what represents the score let us make an example. We set $n = 3, q = 4$. Let $f(x)$ be the message polynomial, $\mathbf{c} = (f(x_1), f(x_2), \ldots, f(x_n))$ the transmitted codeword and

$$M = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \\ m_{41} & m_{42} & m_{43} \end{pmatrix}$$

the multiplicity matrix. Recall that the lines of M correspond to the α's and the columns to the x's. Thus the point (x_2, α_4) has multiplicities m_{42}. Suppose that $c_1 = f(x_1) = \alpha_3, c_2 = f(x_2) = \alpha_1, c_3 = f(x_3) = \alpha_2$, the matrix $[\mathbf{c}]$ associated with $\mathbf{c}$ is then given by

$$[\mathbf{c}] = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

The lines of $[\mathbf{c}]$ correspond to the α's and the columns to its coefficients. For example $[\mathbf{c}]_{31} = 1$ since $c_1 = \alpha_3$. Thus we have that the score of $[\mathbf{c}]$, $S_M([\mathbf{c}])$, is equal to

$$\text{trace}(M, [\mathbf{c}]^T) = \text{trace}(\begin{pmatrix} m_{12} & m_{13} & m_{11} & 0 \\ m_{22} & m_{23} & m_{21} & 0 \\ m_{32} & m_{33} & m_{31} & 0 \\ m_{42} & m_{43} & m_{41} & 0 \end{pmatrix}) = m_{12} + m_{23} + m_{31}.$$

It represents the total multiplicity of the points $(x_2, \alpha_1), (x_3, \alpha_2)$ and (x_1, α_3). These points are "right" points, i.e., points at which there is no error, since $f(x_1) = \alpha_3, f(x_2) = \alpha_1, f(x_3) = \alpha_2$. They correspond to the points (x_i, y_i) at which $f(x_i) = y_i$ in the Guruswami-Sudan algorithm (where $f(x)$ is still the message polynomial).

So we have a polynomial Q which has a zero of multiplicity m_{ij} at the point (x_j, α_i). Its degree is at least l_{w_x, w_y}. If $Q(x, f(x))$ is divisible by a

polynomial of degree larger than l_{w_x,w_y}, then $(y - f(x))$ will be a factor of $Q(x, y)$. We prove this more formally in the following theorem. To make the link with the Guruswami-Sudan algorithm, observe that the following theorem corresponds to proposition 4.4.

Theorem 7.1. *Let C be the cost of a given multiplicity matrix M. Then the polynomial $Q(x, y)$ has a factor $(y - f(x))$ where $f(x)$ evaluates to a codeword $\mathbf{c}$, if the score of $\mathbf{c}$ is large enough compared to C, namely, if*

$$S_M(\mathbf{c}) > l_{1,k-1}(C). \tag{7.4}$$

Proof: By definition of a Reed-Solomon code we know that $f(x_j) = c_j$ for $j = 1, \ldots, n$. We then define $p(x) := Q(x, f(x))$. The degree in x of $p(x)$ is at most $l_{1,k-1}(C)$ by (7.3) and since $f(x)$ has degree at most $k - 1$. We write

$$S_M(\mathbf{c}) = \langle M, [\mathbf{c}] \rangle = m_1 + m_2 + \ldots + m_n.$$

Thus the polynomial $Q(x, y)$ passes through the point (x_j, c_j) with multiplicity at least m_j, for $j = 1, 2, \ldots, n$. Since $f(x_j) = c_j$ for $j = 1, 2, \ldots, n$, it follows from proposition 4.3 and the fact that $x_1, x_2, \ldots, x_n$ are all distincts that $p(x)$ is divisible by the product

$$(x - x_1)^{m_1} \ldots (x - x_n)^{m_n}$$

whose degree is $S_M(\mathbf{c})$. Since the degree of $p(x)$ is at most $l_{1,k-1}(C)$ and by (7.4) we get that $p(x)$ is divisible by a polynomial of degree larger than its degree. Hence $p(x) = Q(x, f(x)) \equiv 0$.

□

7.2 Construction of the multiplicity matrix

This section presents the core contribution of Koetter and Vardy in their paper [14]: an algorithm that converts posterior probabilities derived from the channel output into a choice of interpolation points and their multiplicities.

Let $\mathfrak{M}_{qn}$ denote the set of all $q \times n$ matrices with nonnegative integer entries m_{ij} and let $\mathfrak{M}(C)$ be the finite set of all matrices in $\mathfrak{M}_{qn}$, whose cost is equal to C. Thus

$$\mathfrak{M}(C) := \left\{ M \in \mathfrak{M}_{qn} : \frac{1}{2} \sum_{i=1}^{q} \sum_{j=1}^{n} m_{ij}(m_{ij} + 1) = C \right\}.$$

In view of theorem 7.1, we would like to choose $M \in \mathfrak{M}(C)$ so as to maximize the score of the transmitted codeword $\mathbf{c}$. However, we obviously do not know this codeword. We only know the output of the channel

$(y_1, y_2, \ldots, y_n) \in \mathcal{V}_Y$. We can consider the transmitted codeword as a random vector $\mathcal{X} = (\mathcal{X}_1, \ldots, \mathcal{X}_n)$. To define the probability distribution $P(\cdot)$ of $\mathcal{X}$ we use the reliability matrix Π defining in definition 7.1, namely

$$\begin{aligned} P(x_1, x_2, \ldots, x_n) &:= \prod_{j=1}^{n} \Pr(\mathcal{X}_j = x_j | \mathcal{Y}_j = y_j) \\ &= \prod_{j=1} \Pi(x_j, j) \end{aligned} \quad (7.5)$$

Observe that for a given matrix M, the score of the transmitted codeword is a function of $\mathcal{X}$ given by $S_M(\mathcal{X}) = \langle M, [\mathcal{X}] \rangle$.

Thus, $S_M(\mathcal{X})$ is a random variable. Now the question is: what is the best choice of a multiplicity matrix $M \in \mathfrak{M}(C)$ in this probabilistic setting? Koetter and Vardy choose for several reasons explained in [14, section IV] to compute the multiplicity matrix $M \in \mathfrak{M}(C)$ that maximizes the expected value of $S_M(\mathcal{X})$.

Definition 7.6 (Expected score)**.** *The expected score with respect to a probability distribution $P(\cdot)$ on the random vector $\mathcal{X}$ is defined as follows:*

$$E_P\{S_M(\mathcal{X})\} := \sum_{\mathbf{x} \in \mathcal{V}_X^n} S_M(\mathbf{x}) P(\mathbf{x}) = \sum_{\mathbf{x} \in \mathbb{F}_q^n} \sum_{j=1}^{n} M(x_j, j) P(\mathbf{x}) \quad (7.6)$$

where $M(x_j, j)$ denotes the entry found in the jth column of M in the row indexed by x_j.

Thus we would like to compute the matrix $M(\Pi, C)$ defined as follows

$$M(\Pi, C) := \text{argmax}_{M \in \mathfrak{M}(C)} E_P\{S_M(\mathcal{X})\} \quad (7.7)$$

where the expectation is taken with respect to the probability distribution $P(\cdot)$ in (7.5). Koetter and Vardy in [14, lemma 6] give a useful expression for the expected score, namely

$$E_P\{S_M(\mathcal{X})\} = \langle M, \Pi \rangle. \quad (7.8)$$

Since $\mathcal{X}$ is distributed according to (7.5), then the reliability matrix Π is precisely the expected value of $[\mathcal{X}]$. (7.8) now follows by linearity of expectation:

$$E_P\{S_M(\mathcal{X})\} = E_P\{\langle M, [\mathcal{X}] \rangle\} = \langle M, E_P\{[\mathcal{X}]\} \rangle = \langle M, \Pi \rangle.$$

Referring to the equality (7.6), we observe that increasing m_{ij} from 0 to 1 increases the expected score by π_{ij}, while increasing the cost by 1. In general, increasing m_{ij} from a to $a+1$ always increases the expected score

Koetter-Vardy algorithm:

Input: Reliability matrix Π and a positive integer s, indicating the total number of interpolation points.
Step 1: Set $\Pi^* := \Pi$ and $M :=$ all-zero matrix.
Step 2: Find the position (i, j) of the largest entry π^*_{ij} in Π^*, and set

$$\begin{aligned} \pi^*_{ij} &:= \tfrac{\pi_{ij}}{m_{ij}+2} \\ m_{ij} &:= m_{ij} + 1 \\ s &:= s - 1 \end{aligned}$$

Step 3: If $s = 0$, output M; otherwise go to Step 2.
Output: Multiplicity matrix M.

Table 7.1: Koetter-Vardy algorithm for computing the multiplicity matrix M

by π_{ij} while increasing the cost by $a + 1$. These observations lead to the following algorithm for the computation of the multiplicity matrix M.

Let $\mathcal{M}(\Pi, s)$ denote the multiplicity matrix produced by this algorithm for a given reliability matrix Π and a given number of interpolation points s (counted with multiplicities). The following theorem from [14] shows that this matrix is optimal.

Theorem 7.2. *The matrix $\mathcal{M}(\Pi, s)$ maximizes the expected score among all matrices in $\mathfrak{M}_{qn}$ with the same cost. That is, if C is the cost of $\mathcal{M}(\Pi, s)$, then*

$$\mathcal{M}(\Pi, s) = \text{argmax}_{M \in \mathfrak{M}(C)} \langle M, \Pi \rangle.$$

Proof: With each position (i, j) in the reliability matrix Π, we associate an infinite sequence of rectangles $\mathcal{B}_{ij1}, \mathcal{B}_{ij2}, \ldots$ indexed by the positive integers. Let $\mathfrak{B}$ denote the set of all such rectangles. For each rectangle $\mathcal{B}_{ijl} \in \mathfrak{B}$, we define its $\text{length}(\mathcal{B}_{ijl}) = l, \text{height}(\mathcal{B}_{ijl}) = \pi_{ij}/l$, and

$$\text{area}(\mathcal{B}_{ijl}) = \text{length}(\mathcal{B}_{ijl}) \cdot \text{height}(\mathcal{B}_{ijl}) = \pi_{ij}.$$

For a multiplicity matrix $M \in \mathfrak{M}_{qn}$, we define the corresponding set of rectangles $\sigma(M)$ as

$$\sigma(M) := \{\mathcal{B}_{ijl} : \ 1 \le i \le q, \ 1 \le j \le n, \ 1 \le l \le m_{ij}\}.$$

Observe that the number of rectangles in $\sigma(M)$ is given by $\sum_{i=1}^{q} \sum_{j=1}^{n} m_{ij}$, which is precisely the total number of interpolation points imposed by the

multiplicity matrix M (counted with multiplicities). Furthermore

$$\begin{aligned}
C(M) &= \sum_{\substack{i=1\\j=1}}^{q,n} \frac{m_{ij}(m_{ij}+1)}{2} = \sum_{\substack{i=1\\j=1}}^{q,n} \sum_{l=1}^{m_{ij}} l \\
&= \sum_{\substack{i=1\\j=1}}^{q,n} \sum_{l=1}^{m_{ij}} \text{length}(\mathcal{B}_{ijl}) = \sum_{\mathcal{B}\in\sigma(M)} \text{length}(\mathcal{B}) \\
\langle M, \Pi\rangle &= \sum_{\substack{i=1\\j=1}}^{q,n} m_{ij}\cdot \pi_{ij} = \sum_{\substack{i=1\\j=1}}^{q,n} \sum_{l=1}^{m_{ij}} \pi_{ij} \\
&= \sum_{\substack{i=1\\j=1}}^{q,n} \sum_{l=1}^{m_{ij}} \text{area}(\mathcal{B}_{ijl}) = \sum_{\mathcal{B}\in\sigma(M)} \text{area}(\mathcal{B})
\end{aligned}$$

Thus the cost of M is the total length of all the rectangles in $\sigma(M)$ and the expected score $\langle M, \Pi\rangle$ is the total area of all the rectangles in $\sigma(M)$. It is intuitively clear that to maximize the total area for a given total length, one has to choose the highest rectangles. This is precisely what the algorithm of table 7.1 does. It is now obvious that if the s ***highest*** rectangles in $\mathfrak{B}$ have total length C, then no collection of rectangles of total length at most C can have a larger total area.

□

In order to well understand the Koetter-Vardy algorithm, let us observe more in details what we do in Step 2. We are looking for the s highest rectangles of $\mathfrak{B}$. Let H denote the subset of the s highest rectangles. At the beginning, $H = \emptyset$ and then all rectangles in $\mathfrak{B}$ are available and in particular the ones with length$=1$. Thus the highest rectangle with largest area is $\mathcal{B}_{i_1 j_1 1}$ where (i_1, j_1) is the position of the largest entry $\pi^*_{i_1 j_1}$ in Π, since the π^*_{ij}'s are precisely the heights of the highest rectangles at position (i,j) and, in the first iteration, the π^*_{ij}'s are also the area of the rectangles at position (i,j). Thus $H = \{\mathcal{B}_{i_1 j_1 1}\}$ and hence $\mathcal{B}_{i_1 j_1 1}$ is not yet available. The highest available rectangle associated with position (i_1, j_1) has now length$=2$ and height$=\pi_{i_1,j_1}/2$. This is exactly what the equation

$$\pi^*_{ij} := \frac{\pi_{ij}}{m_{ij}+2}$$

does. It removes from $\mathfrak{B}$ the taken rectangle and updates the height of the highest rectangle available at this position. The equations $m_{ij} = m_{ij} + 1$ and $s = s - 1$ do not require more explanations.

However, observe that the Koetter-Vardy algorithm cannot be used for an arbitrary value of the cost C. The algorithm computes a solution to (7.7) only for those costs that are expressible as the total length of the s highest rectangles in $\mathfrak{B}$ for some s. Fortunately it will generally suffice for the purposes.

The analysis of the performances of the Koetter-Vardy algorithm is a bit longer to be outlined here. Moreover, since this algorithm is based on probabilities, there is no result in terms of number of errors corrected. Thus, we cannot compare easily this algorithm with the Gurusami-Sudan and Parvaresh-Vardy algorithms. Koetter and Vardy, in [14], have compared their algorithm with the Guruswami-Sudan one. They observe that the soft-decision decoding algorithm outperforms the Guruswami-Sudan algorithm by a significant margin. To our knowledge there exists no comparison between the Koetter-Vardy and the Parvaresh-Vardy algorithms.

7.3 Conclusion

The goal of this chapter was mainly to present another way to decode and for channels other than the q-ary symmetric channel. It is also interesting to observe that the algorithm presented in this chapter has a large common part with the Guruswami-Sudan algorithm.

Conclusion

We are now at the end of this work. The goals of the project, presented in the introduction have been achieved. We have first studied a conventional algorithm, the Welch-Berlekamp algorithm, which is the basis of the other algorithms presented. In particular, the Welch-Berlekamp algorithm can be seen as a prelude to the idea of looking for a polynomial Q which is zero at certain points and then finding its $y-$roots. This idea is clearly formulated in the Sudan algorithm which is the first list-decoding algorithm introduced in this work. We then described the Guruswami-Sudan algorithm and its improvements over the Sudan algorithm. These algorithms have been successfully implemented.

By implementing the Guruswami-Sudan algorithm, we observed how its practical running time increases with the multiplicity parameter r. Thus it is not always possible to correct $e_{GS} = n - 1 - \lfloor\sqrt{(k-1)n}\rfloor$ errors in practice since it sometimes requires dealing with a very large matrix. Another interesting result involves the size of the output list, where we observed the conformity between experimental and available theoretical results. In particular, even with more than $\lfloor d_C/2 \rfloor$ errors, in many cases the size of the list will be only 1 with high probability. On the other hand for some codes, correcting the maximum theoretical number of errors (e_{GS}) implies a large list. This leads to potential difficulties in recovering the original codeword from the list.

We then presented the Parvaresh-Vardy algorithm. It still uses the same basic idea (finding a polynomial Q which is zero at some points and has certain properties) but works on new codes (subsets of Reed-Solomon codes). This algorithm can correct a fraction of errors up to $1 - \sqrt[M+1]{(RM)^M}$ and thus improves the results of the Guruswami-Sudan algorithm for rates $R < 1/16$. We implemented this algorithm for codes over $\mathbb{F}_{16}$. However to have a practical implemention for larger fields, additional time would be necessary in order to implement efficient functions that deal with multivariate polynomials.

Finally we have presented the new PVGS algorithm, that improves the

error-correcting capacity of all the previously presented algorithms and for all rates $R \in (0,1)$. This algorithm uses the qualities of both the Guruswami-Sudan and the Parvaresh-Vardy algorithms. Figure 6.2 shows the error-correcting capacities of these three algorithms. We have also discussed the generalization of the PVGS algorithm, i.e., $M > 2$ and maximum degree of $e(x) > 1$), and concluded that these generalizations do not bring significant improvements.

In the last chapter, we briefly introduced soft-decision decoding, to show that there are list-decoders for channels other than the binary symmetric channel. We observed that the Koetter-Vardy algorithm also uses the basic ideas of finding a polynomial Q which is zero at some points with certain multiplicities and then factoring it.

In conclusion, we mention the recent work of Guruswami and Rudra [17], which presents error-correcting codes of rate R that can be list-decoded in polynomial time up to a fraction $(1 - R - \varepsilon)$ of errors, for $0 < R < 1$ and $\varepsilon > 0$. An interesting future work would be to focus on implementation and try to implement practical versions of the algorithms from [17], of the PVGS algorithm and other performant list-decoding algorithms. The comparison between the theoretical results gives precise results concerning which algorithm is the best algorithm. However we have seen that a practical implementation can be very difficult and not efficient for large fields. Thus a work focused on implementation can bring results concerning the possible use of these algorithms in a real situation.

Appendix A

Some results

In section 4.4 the exact expression for r is as follows:

$$r = \frac{(dt - nd + 2t - 2 - d)}{2(nd - t^2 + 2t - 1)} -$$

$$\frac{\sqrt{(4 - 8t - 4ndt - 4d + d^2 + 4nd + 8dt + 4t^2 - 2d^2tn - 2d^2t + 10nd^2 + d^2t^2 - 4dt^2 + n^2d^2)})}{2(nd - t^2 + 2t - 1)}.$$

Bibliography

[1] Venkatesan Guruswami and Madhu Sudan, "Improved Decoding of Reed-Solomon and Algebraic-Geometry Codes", *IEEE Transf. Inform. Theory*, **45:** 1755-1764, Sept. 1999.

[2] Madhu Sudan, "Decoding of Reed-Solomon codes beyond the error correction bound", *J. Complexity*, **13:** 180-193, March 1997.

[3] Madhu Sudan, "Decoding Reed-Solomon Codes beyond the Error-Correction Diameter", *Proc. 35th Annual Allerton Conference on Communication, Control and Computing*, 1997.

[4] Farzad Parvaresh and Alexander Vardy, "Correcting Errors Beyond the Guruswami-Sudan Radius in Polynomial Time", *Proceedings the 46th IEEE Symposium on Foundations of Computer Science (FOCS)*, 285-294, 2005

[5] Algorithmic Introduction to Coding Theory, Course given by Madhu Sudan, 2001.

[6] Erich Kaltofen, "Polynomial Factorization 1987-1991". *LATIN '92*, I. Simon (Ed.), Springer LNCS, **583:** 294-313, 1992.

[7] Victor Shoup, "A fast deterministic algorithm for factoring polynomials over finite fields of small characteristic", *Proc. Intern. Symp. Symbolic and Algebraic Computation (ISSAC)*, 14-21, Bonn, Germany, July 1991.

[8] Victor Shoup, NTL: A Library for doing Number Theory, `http://shoup.net/ntl/`.

[9] Ralf Kötter, "On Algebraic Decoding of Algebraic-Geometric and Cyclic Codes", *Ph.D Thesis*, University of Linköping, Sweden,1996.

[10] David Cox, John Little and Donal O'Shea. *Ideals, Varieties, and Algorithms.* Springer, New-York,1992.

[11] R.M. Roth, G. Ruckenstein. " Efficient decoding of Reed-Solomon codes beyond half the minimum distance", *IEEE Trans. Inform. Theory* **46:** 246-257, 2000.

[12] R. J. McEliece. "The Guruswami-Sudan Decoding Algorithm for Reed-Solomon Codes", *IPN PR 42-153*, January-March 2003, 1-60, May 15, 2003.

[13] K.-M. Cheung. “More on the decoder error probability for Reed-Solomon codes”, *IEEE Trans. Inform. Theory*, **35:** 895-900, July 1989.

[14] Ralf Koetter, Alexander Vardy. “Algebraic Soft-Decision Decoding of Reed-Solomon Codes”, *IEEE Trans. Inform. Theory*, **11:** 2809-2825, November 2003.

[15] Victor Shoup, “A new polynomial factorization algorithm and its implementation”, *Journal of Symbolic Comp.*, **20:** 363-397, 1995.

[16] `http://www-math.cudenver.edu/~wcherowi/jcorn6.html`

[17] Venkatesan Guruswami, Atri Rudra. “ Explicit Capacity-Achieving List-Decodable Codes” *Electronic Colloquium on Computational Complexity (ECCC)* **133**, 2005

[18] `http://en.wikipedia.org/wiki/Reed-Solomon_error_correction`