Chapter 8

Error Correction and Concatenation

The preceding chapters have addressed properties of constrained systems and construction of encoders which encode user data sequences into constrained sequences. In practice, these constrained codes, as applied in data recording systems, may be viewed as part of the modulation/demodulation process of input/output signals of the system. Most systems require the use of some form of error-correction coding (ECC) in addition to constrained coding of the input signal or symbol sequence. It is therefore natural to investigate the interplay between these two forms of coding and the possibilities for efficiently combining their functions into a single coding operation, in analogy to the coded-modulation techniques now in wide use in data transmission.

In this chapter, we give, in the first four sections, a very brief introduction to error-correcting linear block codes (the rough idea of error-correction coding was discussed briefly in Section 1.3). This includes description of the basic properties of linear block codes, finite fields, and culminates with a discussion of the celebrated Reed-Solomon codes.

In Section 8.5, we consider three schemes for concatenating ECC codes and constrained codes. The third scheme involves a data compression idea. This idea is formalized in Section 8.6, and the performance in this context of some specific compression codes is given in Section 8.7. Then, in Section 8.8, we show how a dual version of state-splitting ideas from Chapter 5 can be used to give a general construction of compression codes that are useful in this context.

In Chapter 9, we will consider codes which have combined error-correction and constrained coding properties—in particular codes that are designed to handle error mechanisms that arise in magnetic recording.

8.1 Error-Correction Coding

An (n, M) *(block) code* over a finite alphabet F is a nonempty subset $\mathcal{C}$ of size M of F^{n}. The elements of the alphabet are referred to as *symbols*. The parameter n is called the *code length* and M is the *code size*. The *dimension* (or *information length*) of $\mathcal{C}$ is defined by $\mathsf{k} = \log_{|F|} M$, and the *rate* of $\mathcal{C}$ is $R = \mathsf{k}/\mathsf{n}$. The elements of a code are called *codewords*.

In addition to the parameters, code length and code size, there is a third parameter, called the minimum distance, which gives a rough sense of the robustness of the code to channel noise. For this we need to define the following notion of distance.

The *Hamming distance* between two words $\mathbf{x}, \mathbf{y} \in F^{\mathsf{n}}$ is the number of coordinates in which $\mathbf{x}$ and $\mathbf{y}$ differ. We denote the Hamming distance by $\Delta(\mathbf{x}, \mathbf{y})$.

It is easy to verify that Hamming distance satisfies the following properties of a metric for every three words $\mathbf{x}, \mathbf{y}, \mathbf{z} \in F^{\mathsf{n}}$:

- $\Delta(\mathbf{x}, \mathbf{y}) \geq 0$, with equality if and only if $\mathbf{x} = \mathbf{y}$.
- Symmetry: $\Delta(\mathbf{x}, \mathbf{y}) = \Delta(\mathbf{y}, \mathbf{x})$.
- The triangle inequality: $\Delta(\mathbf{x}, \mathbf{y}) \leq \Delta(\mathbf{x}, \mathbf{z}) + \Delta(\mathbf{z}, \mathbf{y})$.

Let $\mathcal{C}$ be an (n, M) code over F with $M > 1$. The *minimum distance* of $\mathcal{C}$ is the minimum Hamming distance between any two distinct codewords of $\mathcal{C}$. That is, the minimum distance d is given by

$$\mathsf{d} = \min_{\mathbf{c}_1, \mathbf{c}_2 \in \mathcal{C} \,:\, \mathbf{c}_1 \neq \mathbf{c}_2} \Delta(\mathbf{c}_1, \mathbf{c}_2) \ .$$

An (n, M) code with minimum distance d is called an $(\mathsf{n}, M, \mathsf{d})$ *code*.

Example 8.1 The binary $(3, 2, 3)$ *repetition code* is the code $\{000, 111\}$ over $F = \{0, 1\}$. The dimension of the code is $\log_2 2 = 1$ and its rate is $1/3$. □

Example 8.2 The binary $(3, 4, 2)$ *parity code* is the code $\{000, 011, 101, 110\}$ over $F = \{0, 1\}$. The dimension is $\log_2 4 = 2$ and the code rate is $2/3$. □

Given an (n, M, d) code $\mathcal{C}$ over F, let $\mathbf{c} \in \mathcal{C}$ be a codeword transmitted over a noisy channel, and let $\mathbf{y} \in F^{\mathsf{n}}$ be the received word. By an *error* we mean the event of changing an entry in the codeword $\mathbf{c}$. The number of errors equals $\Delta(\mathbf{y}, \mathbf{c})$, and the error locations are the indices of the entries in which $\mathbf{c}$ and $\mathbf{y}$ differ. The task of error correction is to recover the error locations and the error values.

The following result shows that for any code with minimum distance d, there is a procedure that can correct up to $\lfloor (\mathsf{d}-1)/2 \rfloor$ many errors.

Proposition 8.1 *Let $\mathcal{C}$ be a block code over F with code length n and minimum distance d. For a received word $\mathbf{y}$, let $\mathcal{D}(\mathbf{y})$ denote the codeword in $\mathcal{C}$ that is closest (with respect to Hamming distance) to $\mathbf{y}$. If codeword $\mathbf{c}$ is transmitted, $\mathbf{y}$ is received, and $\Delta(\mathbf{y}, \mathbf{c}) \leq (\mathsf{d}-1)/2$ then $\mathcal{D}(\mathbf{y}) = \mathbf{c}$.*

Proof. Suppose to the contrary that $\mathbf{c}' = \mathcal{D}(\mathbf{y}) \neq \mathbf{c}$. By definition,

$$\Delta(\mathbf{y}, \mathbf{c}') \leq \Delta(\mathbf{y}, \mathbf{c}) \leq (d-1)/2 \ .$$

So, by the triangle inequality,

$$\mathsf{d} \leq \Delta(\mathbf{c}, \mathbf{c}') \leq \Delta(\mathbf{y}, \mathbf{c}) + \Delta(\mathbf{y}, \mathbf{c}') \leq \mathsf{d}-1 \ ,$$

which is a contradiction. $\square$

8.2 Linear Codes

Most of the theory of ECC has focused on linear codes. Such a code is defined as a finite-dimensional vector space over a finite field. We assume that the reader is familiar, from a course in elementary linear algebra, with the notion of a vector space or linear space. But since such a course does not necessarily treat finite fields, we give a brief introduction to finite fields in Section 8.3; in particular, we give in Section 8.3 a construction of the finite field $\mathrm{GF}(q)$. For a more thorough introduction to ECC, we refer the reader to any of the excellent textbooks on the subject, such as [LinCo83], [Mcl77] or [Wic95].

8.2.1 Definition

An $(\mathsf{n}, M, \mathsf{d})$ code $\mathcal{C}$ over a finite field $F = \mathrm{GF}(q)$ is called *linear* if $\mathcal{C}$ is a linear sub-space of F^{n} over F, namely, for every $\mathbf{c}_1, \mathbf{c}_2 \in \mathcal{C}$ and $a_1, a_2 \in F$ we have $a_1\mathbf{c}_1 + a_2\mathbf{c}_2 \in \mathcal{C}$.

The *dimension* of a linear (n, M, d) code $\mathcal{C}$ over F is the dimension of $\mathcal{C}$ as a linear sub-space of F^{n} over F. If k is the dimension of $\mathcal{C}$, then we say that $\mathcal{C}$ is a linear $[\mathsf{n}, \mathsf{k}, \mathsf{d}]$ code over F. The difference $\mathsf{n} - \mathsf{k}$ is called the *redundancy* of $\mathcal{C}$.

Every basis of a linear $[\mathsf{n}, \mathsf{k}, \mathsf{d}]$ code $\mathcal{C}$ over $F = \mathrm{GF}(q)$ contains k codewords, the linear combinations of which are distinct and generate the whole set $\mathcal{C}$. Therefore, $|\mathcal{C}| = M = q^{\mathsf{k}}$ and the code rate is $R = (\log_q M)/\mathsf{n} = \mathsf{k}/\mathsf{n}$.

Words $\mathbf{y} = y_1 y_2 \ldots y_{\mathsf{n}}$ over a field F—in particular, codewords of a linear $[\mathsf{n}, \mathsf{k}, \mathsf{d}]$ code over F—will sometimes be denoted by $(y_1 \ y_2 \ \ldots \ y_{\mathsf{n}})$, to emphasize that they are elements of a vector space F^{n}.

Example 8.3 The $(3,4,2)$ parity code over GF(2) is a linear $[3,2,2]$ code since it is spanned by (1 0 1) and (0 1 1). □

The *Hamming weight* of $\mathbf{e} \in F^{\mathsf{n}}$ is the number of nonzero entries in $\mathbf{e}$. We denote the Hamming weight by $\mathsf{w}(\mathbf{e})$. Note that for every two words $\mathbf{x}, \mathbf{y} \in F^{\mathsf{n}}$,

$$\Delta(\mathbf{x}, \mathbf{y}) = \mathsf{w}(\mathbf{y} - \mathbf{x}) = \Delta(\mathbf{y} - \mathbf{x}, \mathbf{0}),$$

where $\mathbf{0}$ denotes the (all-)zero codeword, which is an element of any linear code (since a linear space always contains the zero vector). The following result characterizes the minimum distance of a linear code in terms of its minimum Hamming weight.

Proposition 8.2 *Let $\mathcal{C}$ be a linear $[\mathsf{n},\mathsf{k},\mathsf{d}]$ code over F. Then*

$$\mathsf{d} = \min_{\mathbf{c} \in \mathcal{C} \setminus \{\mathbf{0}\}} \mathsf{w}(\mathbf{c}) \ .$$

Proof. Since $\mathcal{C}$ is linear,

$$\mathbf{c}_1, \mathbf{c}_2 \in \mathcal{C} \quad \Longrightarrow \quad \mathbf{c}_1 - \mathbf{c}_2 \in \mathcal{C} \ .$$

Now, $\Delta(\mathbf{c}_1, \mathbf{c}_2) = \mathsf{w}(\mathbf{c}_1 - \mathbf{c}_2)$ and, so,

$$\mathsf{d} = \min_{\mathbf{c}_1, \mathbf{c}_2 \in \mathcal{C} \,:\, \mathbf{c}_1 \neq \mathbf{c}_2} \Delta(\mathbf{c}_1, \mathbf{c}_2) = \min_{\mathbf{c}_1, \mathbf{c}_2 \in \mathcal{C} \,:\, \mathbf{c}_1 \neq \mathbf{c}_2} \mathsf{w}(\mathbf{c}_1 - \mathbf{c}_2) = \min_{\mathbf{c} \in \mathcal{C} \setminus \{\mathbf{0}\}} \mathsf{w}(\mathbf{c}) \ .$$

□

8.2.2 Generator Matrix

A *generator matrix* of a linear $[\mathsf{n},\mathsf{k},\mathsf{d}]$ code over F is a $\mathsf{k} \times \mathsf{n}$ matrix whose rows form a basis of the code.

Example 8.4 The matrix

$$G = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

is a generator matrix of the $[3,2,2]$ parity code over GF(2), and so is the matrix

$$\hat{G} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix} .$$

In general, the $[\mathsf{n}, \mathsf{n}{-}1, 2]$ *parity code* over a field F is defined as the code with a generator matrix

$$G = \left(\begin{array}{c|c} & -1 \\ I & -1 \\ & \vdots \\ & -1 \end{array} \right) ,$$

where I is the $(\mathsf{n}{-}1) \times (\mathsf{n}{-}1)$ identity matrix. □

Example 8.5 The $(3, 2, 3)$ repetition code over GF(2) is a linear $[3, 1, 3]$ code generated by

$$G = \left(\begin{array}{ccc} 1 & 1 & 1 \end{array} \right) .$$

In general, the $[\mathsf{n}, 1, \mathsf{n}]$ *repetition code* over a field F is defined as the code with a generator matrix

$$G = \left(\begin{array}{cccc} 1 & 1 & \ldots & 1 \end{array} \right) .$$

□

Let $\mathcal{C}$ be a linear $[\mathsf{n}, \mathsf{k}, \mathsf{d}]$ code over F and G be a generator matrix of $\mathcal{C}$. We can encode information words into codewords of $\mathcal{C}$ by regarding the former as vectors $\mathbf{u} \in F^{\mathsf{k}}$ and using a mapping $F^{\mathsf{k}} \to \mathcal{C}$ defined by

$$\mathbf{u} \mapsto \mathbf{u}G .$$

Since $\operatorname{rank}(G) = \mathsf{k}$, we can apply elementary operations to the rows of G to obtain a $\mathsf{k} \times \mathsf{k}$ identity matrix as a sub-matrix of G.

A $\mathsf{k} \times \mathsf{n}$ generator matrix is called *systematic* if it has the form

$$\left(\, I \, | \, A \, \right) ,$$

where I is a $\mathsf{k} \times \mathsf{k}$ identity matrix and A is a $\mathsf{k} \times (\mathsf{n}{-}\mathsf{k})$ matrix.

Not always does a code $\mathcal{C}$ have a systematic generator matrix. However, we can always permute the code coordinates to obtain an *equivalent* (although different) code $\hat{\mathcal{C}}$ for which the first k columns of any generator matrix are linearly independent, in which case the code has a systematic generator matrix. The code $\hat{\mathcal{C}}$ has the same length, dimension, and minimum distance as the original code $\mathcal{C}$.

When using a systematic generator matrix $G = (\, I \,|\, A \,)$ for encoding, the mapping $\mathbf{u} \mapsto \mathbf{u}G$ takes the form $\mathbf{u} \mapsto (\, \mathbf{u} \,|\, \mathbf{u}A \,)$; that is, the information vector is part of the encoded codeword.

8.2.3 Parity-check matrix

Let $\mathcal{C}$ be a linear $[\mathsf{n},\mathsf{k},\mathsf{d}]$ code over F. A *parity-check matrix* of $\mathcal{C}$ is an $r \times \mathsf{n}$ matrix H over F such that for every $\mathbf{c} \in F^{\mathsf{n}}$,

$$\mathbf{c} \in \mathcal{C} \qquad \Longleftrightarrow \qquad H\mathbf{c}^\top = \mathbf{0}\ .$$

In other words, the code $\mathcal{C}$ is the (right) kernel, $\ker(H)$, of H in F^{n}. We thus have

$$\mathrm{rank}(H) = \mathsf{n} - \dim\ker(H) = \mathsf{n} - \mathsf{k}\ .$$

So, in (the most common) case where the rows of H are linearly independent we have $r = \mathsf{n}-\mathsf{k}$.

Let G be a $\mathsf{k} \times \mathsf{n}$ generator matrix of $\mathcal{C}$. The rows of G span $\ker(H)$ and, in particular,

$$HG^\top = 0 \qquad \Longrightarrow \qquad GH^\top = 0\ .$$

Also,

$$\dim\ker(G) = \mathsf{n} - \mathrm{rank}(G) = \mathsf{n} - \mathsf{k}\ .$$

Hence, the rows of H span $\ker(G)$. So, a parity-check matrix of a linear code can be computed by finding a basis of the kernel of a generator matrix of the code.

In the special case where G is a systematic matrix $(\,I \,|\, A\,)$, we can take the $(\mathsf{n}-\mathsf{k}) \times \mathsf{n}$ matrix $H = (\,-A^\top \,|\, I\,)$ as a parity-check matrix.

Example 8.6 The matrix

$$\begin{pmatrix} 1 & 1 & \ldots & 1 \end{pmatrix}$$

is a parity-check matrix of the $[\mathsf{n},\mathsf{n}-1,2]$ parity code over a field F, and

$$\left(\begin{array}{c|c} & -1 \\ \; I \; & -1 \\ & \vdots \\ & -1 \end{array}\right)$$

is a parity-check matrix of the $[\mathsf{n},1,\mathsf{n}]$ repetition code over F. □

Example 8.7 The linear $[7,4,3]$ *Hamming code* over GF(2) is defined by the parity-check matrix

$$H = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}\ .$$

A corresponding generator matrix is given by

$$G = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} .$$

One can check exhaustively that the minimum distance of this code is indeed 3. □

The following theorem provides a characterization of the minimum distance of a linear code through any parity-check matrix of the code.

Theorem 8.3 *Let H be a parity-check matrix of a linear code $\mathcal{C} \neq \{\mathbf{0}\}$. The minimum distance of $\mathcal{C}$ is the largest integer d such that every set of $d-1$ columns in H is linearly independent.*

Proof. Write $H = (\,\mathbf{h}_1\ \mathbf{h}_2\ \ldots\ \mathbf{h}_\mathsf{n}\,)$ and let $\mathbf{c} = (c_1\ c_2\ \ldots\ c_\mathsf{n})$ be a codeword in $\mathcal{C}$ with Hamming weight $t > 0$. Let $J \subseteq \{1, 2, \ldots, \mathsf{n}\}$ be the set of $|J| = t$ indexes of the nonzero entries in $\mathbf{c}$. By $H\mathbf{c}^\top = \mathbf{0}$ we have

$$\sum_{j \in J} c_j \mathbf{h}_j = \mathbf{0} \ ,$$

namely, the t columns of H that are indexed by J are linearly dependent.

Conversely, every set of t linearly dependent columns in H corresponds to at least one nonzero codeword $\mathbf{c} \in \mathcal{C}$ with $\mathsf{w}(\mathbf{c}) \leq t$.

Given d as defined in the theorem, it follows that no nonzero codeword in $\mathcal{C}$ has Hamming weight less than d, but there is at least one codeword in $\mathcal{C}$ whose Hamming weight is d. □

Example 8.8 For an integer $m > 1$, the $[2^m-1, 2^m-1-m, 3]$ Hamming code over $F = \mathrm{GF}(2)$ is defined by an $m \times (2^m-1)$ parity-check matrix H whose columns range over all the nonzero elements of F^m. Every two columns in H are linearly independent and, so, the minimum distance of the code is at least 3. In fact, the minimum distance is exactly 3, since there are three dependent columns, e.g., $(0\ \ldots\ 0\ 0\ 1)^\top$, $(0\ \ldots\ 0\ 1\ 0)^\top$, and $(0\ \ldots\ 0\ 1\ 1)^\top$.□

8.3 Introduction to Finite Fields

Roughly speaking, a finite field is a finite set of elements in which notions of addition and multiplication are defined subject to the following conditions:

- The usual rules (associative, distributive and commutative) of arithmetic hold.
- Each element has an additive inverse (i.e., a "negative" of itself) and a multiplicative inverse (i.e., a "reciprocal"); in other words, you can subtract and divide elements of the field and still remain in the field.

The simplest non-trivial field is $GF(2)$, which consists of just two elements: $\{0, 1\}$ with modulo-2 arithmetic, i.e.,

$$\begin{array}{ccccc} 0+0 & = & 1+1 & = & 0 \\ 0+1 & = & 1+0 & = & 1 \\ 0\cdot 0 & = & 0 & & \\ 1\cdot 1 & = & 1 & & \\ 0\cdot 1 & = & 1\cdot 0 & = & 0 \end{array}$$

More generally, for a prime number p, the *prime field* $\mathrm{GF}(p)$ consists of the elements $\{0, 1, \ldots, p-1\}$, with arithmetic modulo p. It can be shown that $\mathrm{GF}(p)$ contains a *primitive element*, defined as an element α such that the powers of α exhaust all nonzero elements of the field.

Example 8.9 In GF(7) we have

$$2\cdot 4 = 3\cdot 5 = 6\cdot 6 = 1\cdot 1 = 1\ .$$

The elements 3 and 5 are primitive elements:

$$\begin{array}{ccccccccccc} 3^0 & = & 1 & = & 5^0 & \qquad & 3^3 & = & 6 & = & 5^3 \\ 3^1 & = & 3 & = & 5^5 & & 3^4 & = & 4 & = & 5^2 \\ 3^2 & = & 2 & = & 5^4 & & 3^5 & = & 5 & = & 5^1 \end{array}$$

and in fact they are the only primitive elements. □

It turns out that the size of any finite field is a power of a prime number $q = p^h$. Such a field is denoted $\mathrm{GF}(q)$ and is defined as the set of all polynomials of degree less than h in an indeterminate x with coefficients in $\mathrm{GF}(p)$. Addition is then defined as the usual polynomial addition (using arithmetic modulo p).

In order to define multiplication, we make use of an *irreducible polynomial*, $P(x)$ of degree exactly h, with coefficients in $\mathrm{GF}(p)$, i.e., a polynomial that cannot be factored nontrivially over $\mathrm{GF}(p)$. It can be shown that such a polynomial exists for each prime p and integer h.

Example 8.10 The following are all the irreducible polynomials over GF(2) with degree at most 4:

degree 1: $\quad x,\ x+1$

degree 2:	x^2+x+1
degree 3:	$x^3+x+1,\ x^3+x^2+1$
degree 4:	$x^4+x+1,\ x^4+x^3+1,\ x^4+x^3+x^2+x+1$

□

We then define multiplication in GF(q) by multiplication of polynomials followed by reduction by $P(x)$: precisely, to find the product of $a(x)$ and $b(x)$, first form the ordinary product $s(x) = a(x)b(x)$ and then compute the remainder of $s(x)$ when divided by $P(x)$. Irreducbility of $P(x)$ is required in order to guarantee that each nonzero element of GF(q) has a multiplicative inverse.

Example 8.11 Let $F = \mathrm{GF}(2)$ and $P(x) = x^3+x+1$. We construct the field GF(2^3) as the set of polynomials over F of degree less than 3, resulting in Table 8.1, where the elements of the field are written as polynomials in the indeterminate x. The third column in the table

000	0	0
001	1	1
010	x	x
011	$x+1$	x^3
100	x^2	x^2
101	x^2+1	x^6
110	x^2+x	x^4
111	x^2+x+1	x^5

Table 8.1: The field GF(2^3)

expresses each nonzero element in the field as a power of the monomial $x = 0\cdot 1+1\cdot x+0\cdot x^2$. Indeed,

$$\begin{aligned}
x^3 &\equiv x+1 \pmod{P(x)}\ ,\\
x^4 &\equiv x\cdot x^3 \equiv x(x+1) \equiv x^2+x \pmod{P(x)}\ ,\\
x^5 &\equiv x\cdot x^4 \equiv x(x^2+x) \equiv x^3+x^2 \equiv x^2+x+1 \pmod{P(x)}\ ,\\
x^6 &\equiv x\cdot x^5 \equiv x(x^2+x+1) \equiv x^3+x^2+x \equiv x^2+1 \pmod{P(x)}\ ,\\
&\text{and}\\
x^7 &\equiv x\cdot x^6 \equiv x(x^2+1) \equiv x^3+x \equiv 1 \pmod{P(x)}\ .
\end{aligned}$$

□

Just as for prime fields, any finite field GF(q) contains a primitive element. In the preceding example, the monomial x is primitive. In general the monomial x may not be a primitive element of the field. However, it turns out that there is always a choice of the irreducible polynomial $P(x)$ such that the monomial x is a primitive element. Such a polynomial is called a *primitive polynomial*.

Finally, we mention that elements of $\mathrm{GF}(q)$ can be viewed as h-dimensional vectors over the field $\mathrm{GF}(p)$ and this way define a vector space of dimension h over $\mathrm{GF}(p)$. In this way, we may regard the symbols of a code over $\mathrm{GF}(2^8)$ as bytes.

8.4 The Singleton bound and Reed-Solomon codes

Theorem 8.4 (The Singleton bound) *For any* $(\mathsf{n}, M, \mathsf{d})$ *code over an alphabet of size* q*,*

$$\mathsf{d} \le \mathsf{n} - \lceil \log_q M \rceil + 1 \; .$$

Proof. Let $\ell = \lceil \log_q M \rceil - 1$. Since $q^\ell < M$, there must be at least two codewords that agree in their first ℓ coordinates. Hence, $\mathsf{d} \le \mathsf{n} - \ell$. □

For a linear $[\mathsf{n}, \mathsf{k}, \mathsf{d}]$ code over $\mathrm{GF}(q)$ the Singleton bound becomes

$$\mathsf{d} \le \mathsf{n} - \mathsf{k} + 1 \; .$$

This can also be seen from a parity-check matrix of the code: since the rank of a parity-check matrix is $\mathsf{n}{-}\mathsf{k}$, every set (so at least one set) of $\mathsf{n}{-}\mathsf{k}{+}1$ columns in that matrix is linearly dependent.

For linear codes, the Singleton bound can also be obtained by considering a systematic generator matrix of the code: the Hamming weight of each row is at most $\mathsf{n}{-}\mathsf{k}{+}1$.

A code is called *maximum distance separable (MDS)* if it attains the Singleton bound with equality. Note that a linear MDS code can correct a certain number, e, of symbols in error within each codeword if its redundancy, $\mathsf{n}{-}\mathsf{k}$, satisfies $\mathsf{n}{-}\mathsf{k} = 2e$; in other words, for a linear MDS code, two bytes of redundancy are sufficient (and in fact necessary by the Singleton bound) to correct each error. So, a code with two bytes of redundancy can correct one error in each codeword, and a code with four bytes of redundancy can correct two errors in each codeword, and so on.

The following are simple examples of linear MDS codes over $F = \mathrm{GF}(q)$:

- The whole space F^{n}, which is a linear $[\mathsf{n}, \mathsf{n}, 1]$ code over F.
- The $[\mathsf{n}, \mathsf{n}{-}1, 2]$ parity code over F.
- The $[\mathsf{n}, 1, \mathsf{n}]$ repetition code over F.

The following family of MDS codes is among the most widely used error-correction codes today.

Let $\alpha_1, \alpha_2, \ldots, \alpha_{\mathsf{n}}$ be distinct elements of $F = \mathrm{GF}(q)$. A *Reed-Solomon code* over F is a linear $[\mathsf{n}, \mathsf{k}, \mathsf{d}]$ code with the parity-check matrix

$$H = \begin{pmatrix} 1 & 1 & \ldots & 1 \\ \alpha_1 & \alpha_2 & \ldots & \alpha_{\mathsf{n}} \\ \alpha_1^2 & \alpha_2^2 & \ldots & \alpha_{\mathsf{n}}^2 \\ \vdots & \vdots & \vdots & \vdots \\ \alpha_1^{\mathsf{n}-\mathsf{k}-1} & \alpha_2^{\mathsf{n}-\mathsf{k}-1} & \ldots & \alpha_{\mathsf{n}}^{\mathsf{n}-\mathsf{k}-1} \end{pmatrix} .$$

This construction requires $\mathsf{n} \leq q$.

Proposition 8.5 *Every Reed-Solomon code is MDS.*

Proof. Every $(\mathsf{n}-\mathsf{k}) \times (\mathsf{n}-\mathsf{k})$ sub-matrix of H has a *Vandermonde* form

$$B = \begin{pmatrix} 1 & 1 & \ldots & 1 \\ \beta_1 & \beta_2 & \ldots & \beta_{\mathsf{n}-\mathsf{k}} \\ \beta_1^2 & \beta_2^2 & \ldots & \beta_{\mathsf{n}-\mathsf{k}}^2 \\ \vdots & \vdots & \vdots & \vdots \\ \beta_1^{\mathsf{n}-\mathsf{k}-1} & \beta_2^{\mathsf{n}-\mathsf{k}-1} & \ldots & \beta_{\mathsf{n}-\mathsf{k}}^{\mathsf{n}-\mathsf{k}-1} \end{pmatrix} ,$$

where $\beta_1, \beta_2, \ldots, \beta_{\mathsf{n}-\mathsf{k}}$ are distinct elements of the field. Now, the determinant of B is given by

$$\det(B) = \prod_{j>i} (\beta_j - \beta_i)$$

and, therefore, $\det(B) \neq 0$ and B is nonsingular. It follows that every set of $\mathsf{n}-\mathsf{k}$ columns in H is linearly independent and, so, $d \geq \mathsf{n} - \mathsf{k} + 1$. $\square$

Finally, we mention that it is possible to extend Reed-Solomon codes to length $q+1$; such a code is called an *extended Reed-Solomon code*, and these codes are MDS as well [MacS77, Chapter 10].

8.5 Concatenation of ECC and constrained codes

As mentioned in Section 1.3, an ECC encoder and a constrained encoder are usually concatenated as follows: messages are first passed through an error-correction encoder and then a constrained encoder before being transmitted across the channel. At the receiver, data is decoded via the constrained decoder and then the error-correction decoder. This scheme, illustrated in Figure 8.1(a), is called *standard concatenation.*

Typically, the ECC code is a Reed-Solomon code, owing to their excellent error-correction properties and excellent decoding properties. In addition, in order to provide for protection against bursty errors, the ECC scheme involves *m-way-interleaving* of codewords, which we explain as follows. Suppose that the ECC encoder generates m consecutive codewords:

$$\begin{array}{rcl} \mathbf{c}^1 & = & x_1^1 x_2^1 \ldots x_{\mathsf{n}}^1 \\ \ldots & = & \ldots \\ \mathbf{c}^m & = & x_1^m x_2^m \ldots x_{\mathsf{n}}^m \end{array} .$$

Instead of recording these codewords one after another, we first transmit the first symbol of each codeword:

$$x_1^1 x_1^2 \ldots x_1^m$$

and then the second symbol of each codeword

$$x_2^1 x_2^2 \ldots x_2^m,$$

etc. In this way, the errors in a contiguous burst are spread among several codewords, thereby reducing the likelihood of overwhelming the error correcting capability of the code.

It is natural for the constrained encoder to be nearer the channel since its purpose is to produce sequences that are designed to pass through the channel with little likelihood of corruption. On the other hand, standard concatenation requires that the constrained decoder severely limit error propagation, and this precludes the possibility of constrained encoders based on long block lengths.

In *modified concatenation* (sometimes called reversed concatenation) the order of concatenation is reversed as shown in Figure 8.1(b). This idea is due to Bliss [Bli81] and Mansuripur [Man91] and more recently to Immink [Imm97] and Fan and Calderbank [FC98]; the latter demonstrated some advantages of modified concatenation even with constrained encoders based on relatively short block lengths. A user data sequence $\mathbf{u}$ is first encoded via a high rate encoder E1 into a constrained sequence $\mathbf{w}$. In order to achieve the high rate (very close to capacity), long block lengths must be used. If $\mathbf{w}$ is then transmitted across a noisy (binary) channel, a small burst error or even a single isolated error in the received sequence $\hat{\mathbf{w}}$ could affect much or possibly all of the decoded sequence $\hat{\mathbf{u}}$, yielding enormous error propagation. To avoid this, error correction is incorporated and used to correct all errors before decoding $\hat{\mathbf{w}}$. This is done by computing a sequence $\mathbf{r}$ of parity symbols on $\mathbf{w}$, and then encoding $\mathbf{r}$ into a constrained sequence $\mathbf{y}$ via a second constrained encoder E2 which is less efficient (namely, has a lower rate) than the first encoder; yet it operates on shorter blocks. Both constrained sequences $\mathbf{w}$ and $\mathbf{y}$ are then transmitted across the noisy channel. The sequence of parity symbols should be chosen to allow correction of a prescribed typical channel error event in $\mathbf{w}$, such as bursts of errors up to a certain length. The decoder attempts to recover $\mathbf{u}$ from the possibly corrupted versions $\hat{\mathbf{w}}$ of $\mathbf{w}$ and $\hat{\mathbf{y}}$ of $\mathbf{y}$. Since the constrained encoder E2 uses short block lengths, it is presumably subject to very little error propagation. Then the decoded version, $\hat{\mathbf{r}}$, of $\hat{\mathbf{y}}$ can be used to correct $\hat{\mathbf{w}}$, without fear of error propagation (i.e., the error events in $\hat{\mathbf{w}}\hat{\mathbf{r}}$ look roughly like the raw channel error events in $\hat{\mathbf{w}}\hat{\mathbf{y}}$). In this way, $\mathbf{w}$ is recovered error-free; decoding $\mathbf{w}$ via the first constrained decoder recovers $\mathbf{u}$ error-free.

One of Immink's key contributions in [Imm97] was the realization that $\mathbf{w}$, being an encoded version of $\mathbf{u}$, is longer than $\mathbf{u}$, so that it may be necessary to increase the number of parity symbols $\mathbf{r}$ for the error-correcting code to achieve the same performance. In addition, for long bursts, the effect of a burst of channel errors is magnified relative to the standard concatenation scheme, since the bursts are not first decoded by the constrained decoder. Immink's solution, shown in Figure 8.1(c), to this problem was to *compress* the sequence $\mathbf{w}$ in a lossless (one-to-one) manner into a sequence $\mathbf{s}$, and then compute the sequence of parity symbols $\mathbf{r}$ based on $\mathbf{s}$; for instance, $\mathbf{s}$ (respectively, $\mathbf{r}$) could be the information (respectively, parity) portions of a Reed-Solomon code. So the parity sequence $\mathbf{r}$, and therefore also the E2-encoded sequence $\mathbf{y}$, can be made shorter, thereby lowering the overhead of the error-correction scheme. At the channel output, the received sequence $\hat{\mathbf{w}}$ is compressed to a sequence $\hat{\mathbf{s}}$, and the ECC decoder recovers $\mathbf{s}$ from $\hat{\mathbf{r}}$ and $\hat{\mathbf{s}}$. Then the decompressor recovers $\mathbf{w}$, and the constrained decoder D1 recovers $\mathbf{u}$.

At one extreme, one could compress $\mathbf{w}$ back to $\mathbf{u}$ (in which case $\mathbf{s}$ would be the same as $\mathbf{u}$). But then a small channel error in $\hat{\mathbf{w}}$ could corrupt all of $\hat{\mathbf{s}}$ *before* error correction. Instead, the compression scheme will guarantee that such a channel error can corrupt only a limited number of bytes in $\hat{\mathbf{s}}$.

In constrained coding, as we have presented it in this text, unconstrained user sequences are encoded, in a lossless manner, into a constrained sequences; after being passed through a channel the constrained sequences are decoded to unconstrained user sequences.

In lossless data compression, the roles of encoder and decoder are reversed: constrained sequences from a source are encoded into unconstrained sequences where no distortion is permitted upon decompression. This duality between constrained coding and lossless data compression has been noted by several authors (see for example [Ari90], [Ker91], [MLT83], [TLM27]). In these works, data compression techniques such as arithmetic coding have been applied to constrained coding. In the remainder of this chapter, we apply constrained coding to lossless data compression, to obtain compressors that can be used in the scheme of Figure 8.1(c).

8.6 Block and sliding-block compressible codes

A *rate $p : q$ block-compressor* for a constrained system S is simply a one-to-one mapping from the set S_q, of words in length q in S, into the set of unconstrained binary words of length p (*the reader should not confuse S_q with the q-th power system, S^q*). Clearly, a necessary and sufficient condition for the existence of a rate $p : q$ block compressor for S is:

$$|S_q| \leq 2^p . \tag{8.1}$$

Immink gives in [Imm97] two simple examples: a rate $8 : 11$ block compressor for the $(1, 12)$-RLL constraint and a rate $8 : 13$ block compressor for the $(2, 15)$-RLL constraint. For $p = 8$, these values of q are optimal: a simple computation reveals that condition (8.1) would be violated for any rate $8 : 12$ block compressor for the $(1, 12)$-RLL constraint and any rate $8 : 14$ block compressor for the $(2, 15)$-RLL constraint.

Clearly, $p = 8$ is a good choice owing to the availability of high performance, high efficiency, off-the-shelf Reed-Solomon codes. But allowing other values of p can give added flexibility in the choice of compression schemes (provided that p and the symbol alphabet of the ECC are somewhat compatible). Clearly, it is desirable to have a small compression rate p/q, and smaller compression rates can be achieved by larger block lengths p and q. But the capacity of the constraint imposes a lower bound on the compression rates, as we show next.

Since $|S_{qm}| \leq |S_q|^m$ for any choice of positive integers q and m, it follows that

$$\mathsf{cap}(S) = \lim_{m\to\infty} (1/(qm)) \cdot \log |S_{qm}| \leq (1/q) \cdot \log |S_q| ;$$

that is, the limit in the definition of capacity is taken over elements each of which is an upper bound on $\mathsf{cap}(S)$. Combining this with (8.1) yields

$$\mathsf{cap}(S) \leq (1/q) \cdot \log |S_q| \leq p/q . \tag{8.2}$$

Thus, to obtain compression rates p/q close to capacity, we need to take q (and hence p) sufficiently large so that $(1/q) \cdot \log |S_q|$ is close enough to capacity. This approach has several drawbacks. First, such schemes can be rather complex. Secondly, if the typical burst error length is short relative to q, then the compression code may actually expand the burst. Third, even if the typical burst error is of length comparable to q, it may be aligned so as to affect two or more consecutive q-codewords, and therefore two or more consecutive unconstrained p-blocks; this "edge-effect" can counteract the benefits of using compression codes.

The foregoing discussion leads us to consider a more general class of compression codes, in particular *lossless sliding-block compression codes*. Such a code consists of a *compressor* and an *expanding coder* (in short, *excoder*), which acts as a "decompressor". The compressor is a sliding-block code from sequences of q-codewords of S to unconstrained sequences of p-blocks over $\{0, 1\}$; that is, a q-codeword $\mathbf{w}$ is compressed into a p-block $\mathbf{s}$ as a time-invariant function of $\mathbf{w}$ and perhaps some m preceding and a upcoming q-codewords. The excoder, on the other hand, will have the form of a finite-state machine. Just as in conventional constrained coding, the *sliding-block window length* is defined as the sum $\mathsf{m}+\mathsf{a}+1$.

We next present a precise definition of the model of compressors and excoders considered in this chapter. For the sake of convenience, we start with excoders and then base the definition of compressors on that of the matching excoders. Let S be a constraint over an alphabet Σ, let Φ be a set of size n, and let m and a be nonnegative integers. An (m,a)-*sliding-block compressible* (S,n)-*excoder* is a graph $\mathcal{E}$ in which the edges are labeled by elements of Σ and, in addition, each edge is endowed by a *tag* from Φ so that the following holds:

(X1) the outgoing edges from each state are assigned distinct tags from Φ; in particular, each state will have at most n outgoing edges;

(X2) S is contained in the constraint (over Σ) that is presented by $\mathcal{E}$; and—

(X3) if $\mathbf{w}$ is a word in $S_{\mathsf{m}+\mathsf{a}+1}$, and $e_{-\mathsf{m}}e_{-\mathsf{m}+1}\ldots e_0\ldots e_{\mathsf{a}}$ and $e'_{-\mathsf{m}}e'_{-\mathsf{m}+1}\ldots e'_0\ldots e'_{\mathsf{a}}$ are sequences of edges that form two paths in $\mathcal{E}$ both labeled by $\mathbf{w}$, then the tags of e_0 and e'_0 agree.

Often we will apply this definition to a constrained system S whose alphabet consists of q-codewords in another constraint S', in which case $S_{\mathsf{m}+\mathsf{a}+1}$ will consist of words of length $(\mathsf{m}+\mathsf{a}+1)q$ in S' (see the definition below of a rate $p:q$ excoder).

The definition of an (m,a)-sliding-block compressible (S,n)-excoder $\mathcal{E}$ bears similarity to that of a tagged (m,a)-sliding-block decodable (S,n)-encoder: the main difference is in the containment relationship between S and the constraint presented by $\mathcal{E}$. Here, $\mathcal{E}$ must generate every word in S and, in addition, it may generate words that are not in S; note, however, that condition (X3) applies only to those paths in $\mathcal{E}$ that generate words in S.

One could replace condition (X3) by a weaker condition that would correspond to the lossless condition for encoders given in Section 4.1. But this may not be adequate for the compression desired in the scheme of Figure 8.1(c).

Condition (X3) induces a mapping $\mathcal{C} : S_{\mathsf{m}+\mathsf{a}+1} \to \Phi$, which, in turn, defines the *sliding-block compressor* of $\mathcal{E}$ as follows. For any positive integer ℓ, the compressor maps every word

$$\mathbf{w} = w_{-\mathsf{m}}w_{-\mathsf{m}+1}\ldots w_0w_1\ldots w_{\ell-1}w_\ell\ldots w_{\ell+\mathsf{a}-1}$$

in $S_{\mathsf{m}+\mathsf{a}+\ell}$ into a pair $(v,\mathbf{s})$, where v is an *initial state* in $\mathcal{E}$, which can be the initial state of any path in $\mathcal{E}$ labeled by $w_0w_1\ldots w_{\ell+\mathsf{a}-1}$, and

$$\mathbf{s} = s_0s_1\ldots s_{\ell-1}$$

is a tag sequence in Φ^ℓ defined by

$$s_i = \mathcal{C}(w_{i-\mathsf{m}} w_{i-\mathsf{m}+1} \ldots w_i \ldots w_{i+\mathsf{a}}) \; , \quad 0 \le i < \ell \; .$$

Observe that the excoder can recover the sub-word $w_0 w_1 \ldots w_{\ell-1}$ of $\mathbf{w}$ by reading the labels along the (unique) path of length ℓ in $\mathcal{E}$ that starts at v and is tagged by $\mathbf{s}$.

In the examples, given later in this chapter, $\mathcal{E}$ will be (m, a)-*definite on* S: if $\mathbf{w}$ is a word in $S_{\mathsf{m}+\mathsf{a}+1}$, and $e_{-\mathsf{m}} e_{-\mathsf{m}+1} \ldots e_0 \ldots e_{\mathsf{a}}$ and $e'_{-\mathsf{m}} e'_{-\mathsf{m}+1} \ldots e'_0 \ldots e'_{\mathsf{a}}$ are two paths in $\mathcal{E}$ both generating $\mathbf{w}$, then $e_0 = e'_0$; note that (m, a)-definiteness on S is stronger than condition (X3).

Next, we show how rate $p : q$ compression codes can be described through (S, n)-excoders and compressors. Let S be a constrained system that is presented by a labeled graph G. We now define an (m, a)-*sliding-block compressible excoder for* S *at rate* $p : q$ to be an (m, a)-sliding-block compressible $(S^q, 2^p)$-excoder. The tag set Φ is taken as $\{0, 1\}^p$, namely, the set of all possible values of any p-block. So, a rate $p : q$ excoder for S maps p-blocks into q-codewords in a state-dependent manner; the respective compressor, in turn, maps a sequence of q-codewords into a sequence of p-blocks, where the i-th q-codeword is compressed into a p-block through a mapping applied to the i-th q-codeword, as well as m preceding and a upcoming q-codewords. A *block excoder for* S *at rate* $p : q$ is a rate $p : q$ excoder for S with one state. Note that the corresponding sliding-block compressor is simply a block compressor, as defined at the beginning of this section.

We next establish a necessary condition for the existence of (S, n)-excoders.

Proposition 8.6 *Let* S *be a constraint. There is an* (m, a)*-sliding-block compressible* (S, n)*-excoder only if* $\mathsf{cap}(S) \le \log n$.

Proof. Let $\mathcal{E}$ be an (m, a)-sliding-block compressible (S, n)-excoder and let V and Φ be the set of states and the set of tags of $\mathcal{E}$, respectively. Suppose that S' is an irreducible constraint contained in S such that $\mathsf{cap}(S') = \mathsf{cap}(S)$.

Let $\mathbf{w}$ be a word in S'_ℓ (and hence in S_ℓ). Since S' is irreducible, the word $\mathbf{w}$ can be extended to a word $\mathbf{w}'\mathbf{w}\mathbf{w}'' \in S'_{\mathsf{m}+\mathsf{a}+\ell}$. The compressor of $\mathcal{E}$ maps the word $\mathbf{w}'\mathbf{w}\mathbf{w}''$ into a pair $(v, \mathbf{s})$, where $v \in V$ and $\mathbf{s} \in \Phi^\ell$, and the (unique) path in $\mathcal{E}$ that starts at v and is tagged by $\mathbf{s}$ generates the word $\mathbf{w}$. We have thus obtained through the compressor a one-to-one mapping from S'_ℓ into $V \times \Phi^\ell$; so,

$$|S'_\ell| \le |V| \cdot n^\ell \; .$$

The result follows by taking the limit as $\ell \to \infty$ and using the definition of capacity. □

Proposition 8.6 implies that there is a sliding-block compressible excoder for S at rate $p : q$ only if $\mathsf{cap}(S^q) \le p$ or, equivalently, $\mathsf{cap}(S) \le p/q$. The latter inequality is exactly the reverse of Shannon's bound on the rate of (conventional) constrained encoders. The bound (8.2) amounts to the special case of Proposition 8.6 for block excoders.

The next result, which we establish in Section 8.8, states that for finite-type constraints the condition in Proposition 8.6 is not only necessary, but also sufficient for the existence of sliding-block compressible excoders.

Proposition 8.7 *Let S be a finite-type constrained system with memory m, and let n be a positive integer such that $\mathsf{cap}(S) \le \log n$. Then there is an (m, a)-sliding-block compressible (S, n)-excoder; in fact, this excoder is (m, a)-definite on S.*

Finally, we make some remarks regarding the inclusion of the initial state in the information conveyed from the compressor to the excoder. The cost of transmitting this initial state is quite minimal. Typically, there will be a small number of states and the number of bits required to represent a state is only the logarithm of that number. Also, in the scheme of Figure 8.1(c), one does not really need to expand the entire tag sequence after error correction: since channel decoding takes place after error correction and since the channel decoder has full knowledge of the received (uncompressed) constrained sequence, it need only re-expand the corrected p-blocks in the compressed tag sequence; since a previously corrected portion of the received sequence is very likely to contain state information (for example if the excoder $\mathcal{E}$ is (m, a)-definite on S), no extra state information may be needed at all. Another alternative is to simply compress only those constrained sequences that can be generated from one fixed state of the excoder. When incorporated into Immink's scheme this would entail a loss in capacity, but the loss is very small since the block lengths are so long. A third solution, which is applicable to (m, a)-definite excoders, is to include the first $\mathsf{m}+\mathsf{a}+1$ q-codewords of the (non-compressed) constrained sequence in the bit stream that is protected by the ECC. Thus, the ECC decoder of the receiving end will reconstruct the correct prefix of the constrained sequence, thereby allowing us to recover the state information.

8.7 Application to burst correction

When used in conjunction with the scheme in Figure 8.1(c), there are a number of factors that may affect the choice of a compressor-excoder pair such as the complexity of the compression and decompression and the error-propagation associated with the application of the compression on the receiving end. In particular, we are concerned with how compressors handle raw channel bursts, and their suitability for use with a symbol-based ECC.

The compressor is applied to the channel bit sequence immediately after the channel, so that a benefit of using a low-rate excoder is that the length of a raw channel burst will be roughly decreased by the compression factor p/q, when the length of the burst is long (relative to q). On the other hand, edge effects in the use of a compressor can expand the error length, and this error propagation may dominate for short bursts. In addition, for sliding-block compressible excoders, the sliding-block window length will also extend the burst. Ultimately, the choice of a compressor-excoder pair involves a balance of these four factors:

1. compression rate $p : q$;
2. edge effects (how many extra p-blocks are affected by the phasing of a burst);
3. effect of the sliding-block window length $\mathsf{m}+\mathsf{a}+1$ (i.e., how many extra p-blocks may be affected by each error); and —
4. compatibility between the block length p and the symbol alphabet of the ECC.

We consider here the effect of a channel burst of length L bits on the maximum number of affected p-blocks. Hereafter, by a length of a burst in a sequence over a given symbol alphabet we mean the number of symbols between (and including) the first and last erroneous symbols. Our computation will mainly concentrate on the simplified model where any error in the q-codeword will result in an entirely erroneous p-block upon compression, although in practice it might be possible to mitigate this effect by a proper tag assignment to the excoder (see Section 8.8.3).

The maximum number of q-codewords (including the edge effect) that can be affected by a channel burst of length L bits is either $\lfloor (L-1)/q \rfloor + 1$ or $\lceil (L-1)/q \rceil + 1$, depending on the phasing within a q-codeword where the channel burst starts. For (m, a)-sliding-block compressible excoders, the effect of the memory and anticipation is to expand the number of affected p-blocks by $\mathsf{m}+\mathsf{a}$, so that we get a maximum of

$$N = N(L) = \lceil (L-1)/q \rceil + \mathsf{m}+\mathsf{a}+1 \tag{8.3}$$

affected p-blocks.

Next, we need to translate from a number of erroneous p-blocks to a corresponding number of symbol errors for the ECC. Let the symbol alphabet of the ECC in Figure 8.1(c) be the finite field $\mathrm{GF}(2^B)$; namely, the sequence of p-blocks is regarded as a long bit-stream and sub-divided into non-overlapping blocks of length B bits, each such block being a symbol of the ECC and regarded as a "B-bit byte." We make the assumption that the boundaries between p-blocks align with the boundaries between ECC symbols as often as possible, in particular, every $(pB)/\gcd(p,B)$ bits. We can then calculate the maximum number of ECC symbols that are in error due to a channel burst of length L bits.

Consider our basic unit to be of size $\gcd(p,B)$ bits, so that we are starting with a burst of length $(Np)/\gcd(p,B)$, and looking for the maximum number of affected blocks of length $B/\gcd(p,B)$. This is analogous to finding the maximum number of q-codewords affected by a burst of channel bits, and we obtain the following expression for the number of ECC symbols in error as a function of L and B:

$$D(L,B) = \left\lceil \frac{(Np)/\gcd(p,B) - 1}{B/\gcd(p,B)} \right\rceil + 1 = \left\lceil \frac{Np - \gcd(p,B)}{B} \right\rceil + 1 \; .$$

Putting this together with (8.3) yields

$$D(L,B) = \left\lceil \frac{(\lceil (L-1)/q \rceil + \mathsf{m}+\mathsf{a}+1)p - \gcd(p,B)}{B} \right\rceil + 1 \, .$$

Example 8.12 Consider a $(0,1)$-sliding-block compressible excoder for the $(2,\infty)$-RLL constraint at rate $4:7$ (such as the excoder that we will present in Example 8.16 in Section 8.8.2). Table 8.2 contains the respective values of $D(L,B)$ for $L=40$ and $B=4,5,6,7,8$.

B	$D(40,B)$	ν	ρ	κ
4	8	544	64	840
5	8	1,320	(80)	2,170
6	6	2,340	72	3,969
7	6	5,418	(84)	9,334
8	5	10,280	80	17,850

Table 8.2: Parameters of an ECC used for burst lengths of up to 40 bits in conjunction with a $(0,1)$-sliding-block compressible excoder for the $(2,\infty)$-RLL constraint at rate $4:7$.

The last three columns in Table 8.2 show the parameters of an ECC that consists of $D(L,B)$-way interleaving of an extended Reed-Solomon code of length 2^B+1 over GF(2^B). The overall block length (in bits) of this ECC scheme equals $\nu = \nu(L,B) = (2^B+1)\cdot B \cdot D(L,B)$; the values of ν are listed in the third column of the table. Since Reed-Solomon codes are maximum distance separable, each symbol error of GF(2^B) can be corrected using two redundancy symbols. Therefore, the total number of redundant symbols is $2D(L,B)$. The redundancy (in bits) of the coding scheme thus equals $\rho = \rho(L,B) = 2\cdot B\cdot D(L,B)$; this is the length of "Parity" in Figure 8.1(c). The values of ρ are listed in the fourth column of the table, where numbers in parentheses indicate that smaller redundancy values (and larger ECC block lengths) can be obtained by using a larger value of B.

A block of ν bits at the output of the ECC encoder in Figure 8.1(c) corresponds to $\nu-\rho$ bits at the input of that encoder; those bits, in turn, correspond to $\kappa = \kappa(L,B) = \lfloor (\nu-\rho)\cdot q/p \rfloor$ channel bits at the input of the lossless compressor. The values of κ are listed in the fifth column of Table 8.2. Clearly, a smaller value of the ratio ρ/κ means a smaller overhead introduced by the ECC (when combined with the compression). □

Using the ECC scheme and the notations of Example 8.12, we can fix a number, κ_0, of channel bits and compute for each (maximal) burst length L the respective redundancy ρ obtained by optimizing over all values of B for which $\kappa(L,B) \geq \kappa_0$. As an example, consider a message of 512 user bytes (4,096 bits) in Figure 8.1(c). Selecting the rate $256:466$ code of [Imm97] as the constrained encoder D1, the message is mapped into 7,456 channel bits. Figure 8.2 shows the best redundancy values attained for $\kappa_0 = 7,456$ and $L \leq 300$ using a $(0,1)$-sliding-block compressible excoder at rate $4:7$ for the $(2,\infty)$-RLL constraint. The figure shows the redundancy values also for two other block excoders for this constraint at rates $8:13$ and $4:6$; note that $8:13$ is the rate of the excoder presented in [Imm97]. Thus we see that for longer bursts, the sliding-block excoder requires less redundancy due to a better compression of the burst length, in spite of the longer sliding-block window. (We point out that Figure 8.2 is the same also for $\kappa_0 = 7,427$, which is the number we get when we divide 4,096 by the capacity ($\approx .5515$) of the $(2,\infty)$-RLL constraint.)

8.8 Constructing sliding-block compressible excoders

Our construction of excoders (and respective compressors) follows the lines of the state-splitting algorithm in Figure 5.9 for constructing finite-state encoders.

Recall that in the conventional state-splitting algorithm, we begin with a graph presentation G of the given constraint S. Typically, we assume that G is deterministic, such as the graph $G_{2,\infty}$ in Figure 8.3 which presents the $(2,\infty)$-RLL constraint. The state-splitting algorithm generates an encoder for S through a sequence of state splittings, which are guided by approximate eigenvectors.

The algorithm we present here is very similar; the main difference is that instead of approximate eigenvectors, we will use what we call *super-vectors*. Such a vector will lead us through a sequence of state-splitting operations beginning with the graph G and ending with a graph H with out-degree at most n (i.e., each state has at most n outgoing edges). Then one assigns to the edges of H tags taken from the tag alphabet Φ (of size n) such that at each state all outgoing edges have distinct tags; typically, $n = 2^p$ and $\Phi = \{0,1\}^p$. The tagged version of H will be the (S,n)-excoder $\mathcal{E}$.

In order to guarantee the sliding-block compressibility of $\mathcal{E}$, we will assume that S has finite memory m, in which case we take G as a (necessarily deterministic) graph presentation of S that has memory m. When state splitting is applied to this graph, we are guaranteed to end up with an excoder $\mathcal{E}$ which is (m,a)-definite on S, where a is the number of rounds of out-splittings; in particular, $\mathcal{E}$ will be (m,a)-sliding-block compressible.

8.8.1 Super-vectors

As is the case with approximate eigenvectors, super-vectors will be computed using the adjacency matrix A_G of the graph presentation G of S. Recall that for a deterministic presentation, the adjacency matrix can be used to compute the capacity of S; namely $\mathsf{cap}(S) = \log \lambda(A_G)$, where $\lambda(A_G)$ is the largest (absolute value of any) eigenvalue of A_G.

Recall also that $(A_G)^q = A_{G^q}$ and, so, $(\lambda(A_G))^q = \lambda(A_{G^q})$.

Example 8.13 The adjacency matrix of the graph $G_{2,\infty}$ in Figure 8.3 is

$$A_{G_{2,\infty}} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix} ,$$

and the capacity of the $(2, \infty)$-RLL constraint is given by $\log \lambda(A_{G_{2,\infty}}) \approx \log 1.4656 \approx .5515$. By Proposition 8.6, we will be able to construct a sliding-block compressible excoder for the $(2, \infty)$-RLL constraint only if its rate $p : q$ satisfies $(\lambda(A_{G_{2,\infty}}))^q \leq 2^p$. In the running example of this section, we will choose $p = 4$ and $q = 7$, in which case $(\lambda(A_G))^q \approx 14.5227 \leq 16 = 2^p$.$\Box$

Let A be a nonnegative integer square matrix and let n be a positive integer; e.g., $A = A_G^q$ and $n = 2^p$. An (A, n)-*super-vector* is a nonnegative integer vector $\mathbf{x}$, not identically zero, such that

$$A\mathbf{x} \leq n\mathbf{x} .$$

Note that approximate eigenvectors (from Section 5.2.1) are defined the same way except that the inequality is reversed (with the benefit of hindsight, approximate eigenvectors should probably have been called "sub-vectors").

By a straightforward modification of the corresponding proof for approximate eigenvectors (Theorem 5.4), it follows that for any nonnegative integer square matrix A, there exists an (A, n)-super-vector if and only if $\lambda(A) \leq n$. The proof suggests ways of finding such a vector, but a simpler algorithm is presented in Figure 8.4; this algorithm is the analogue of the Franaszek Algorithm for finding approximate eigenvectors.

Next, we summarize the properties of this algorithm—in particular, the uniqueness of a minimum (A, n)-super-vector $\mathbf{x}^*$ and the fact that the algorithm always converges to $\mathbf{x}^*$. For convenience we assume that A is irreducible.

Proposition 8.8 *Let A be an irreducible integer matrix and let n be a positive integer with $\lambda(A) \leq n$. Then the following holds:*

(a) Any (A, n)-super-vector is strictly positive.

(b) If $\mathbf{x}$ and $\mathbf{x}'$ are (A, n)-super-vectors, then the vector defined by $\mathbf{z} = \min\{\mathbf{x}, \mathbf{x}'\}$ is also an (A, n)-super-vector (here, $\min\{\cdot, \cdot\}$ is applied componentwise).

(c) There is a unique minimum (A, n)-super-vector, i.e., a unique (A, n)-super-vector $\mathbf{x}^*$ such that for any (A, n)-super-vector $\mathbf{x}$ we have $\mathbf{x}^* \leq \mathbf{x}$.

(d) The algorithm in Figure 8.4 eventually halts and returns the vector $\mathbf{x}^*$.

Parts (b) and (d) of the preceding result are analogous to to the corresponding results for approximate eigenvectors (given in Proposition 5.5). On the other hand, for approximate eigenvectors, there is no analogous uniqueness result and there is no clear choice of initial vector for the algorithm.

Proof of Proposition 8.8. (a) Let $\mathbf{x} = (x_u)_u$ be an (A, n)-super-vector. If some entry x_u is 0 then, according to the inequality $A\mathbf{x} \leq n\mathbf{x}$, we have $x_v = 0$ for all v such that $A_{u,v} \neq 0$. By irreducibility of A, this implies that $\mathbf{x}$ is identically zero, contrary to the definition of an (A, n)-super-vector.

(b) Since A is nonnegative, $A\mathbf{z} \leq A\mathbf{x} \leq n\mathbf{x}$ and $A\mathbf{z} \leq A\mathbf{x}' \leq n\mathbf{x}'$; so, $A\mathbf{z} \leq n\mathbf{z}$. Clearly $\mathbf{z}$ has only integer entries. By (a), $\mathbf{x}$ and $\mathbf{x}'$ are strictly positive, and thus so is $\mathbf{z}$; in particular, it is not identically zero.

(c) Let $\mathbf{x}^*$ be obtained by taking the componentwise minimum of all (A, n)-super-vectors. By (b), $\mathbf{x}^*$ is an (A, n)-super-vector, and it is clearly the unique minimum such vector.

(d) Denote by $\mathbf{x}^i$ the value of $\mathbf{x}$ after the i-th iteration of the `while` loop in Figure 8.4, with $\mathbf{x}^0 = (1\ 1\ \ldots\ 1)^\top$. We show by induction on i that $\mathbf{x}^i \leq \mathbf{x}^*$. The induction base $i = 0$ follows from (a). Now, suppose that $\mathbf{x}^i \leq \mathbf{x}^*$ for some i. Since A is nonnegative, we have

$$\frac{1}{n}A\mathbf{x}^i \leq \frac{1}{n}A\mathbf{x}^* \leq \mathbf{x}^* .$$

Therefore, $\mathbf{x}^{i+1} = \max\left\{\left\lceil \frac{1}{n}A\mathbf{x}^i \right\rceil, \mathbf{x}^i\right\} \leq \mathbf{x}^*$, thereby establishing the induction step.

Next we verify that the algorithm halts. Observe that $\mathbf{x}^0 \leq \mathbf{x}^1 \leq \mathbf{x}^2 \leq \ldots \leq \mathbf{x}^*$ and that $\mathbf{x}^i$ are integer vectors. Now, if the algorithm did not halt, there had to be an index i for which $\mathbf{x}^{i+1} = \mathbf{x}^i$. However, that would imply $\left\lceil \frac{1}{n}A\mathbf{x}^i \right\rceil \leq \mathbf{x}^i$ (in which case $\mathbf{x}^i$ would in fact be a super-vector), so the algorithm had to halt in the i-th iteration.

We therefore conclude that the algorithm halts and returns an (A, n)-super-vector $\mathbf{x} \leq \mathbf{x}^*$; in fact, we must have $\mathbf{x} = \mathbf{x}^*$, since $\mathbf{x}^*$ is the unique minimum (A, n)-super-vector. $\Box$

Example 8.14 The adjacency matrix of the graph $G = G^7_{2,\infty}$ is given by

$$A_G = A^7_{G_{2,\infty}} = \begin{pmatrix} 3 & 2 & 4 \\ 4 & 3 & 6 \\ 6 & 4 & 9 \end{pmatrix} .$$

Now,

$$\begin{aligned}
\mathbf{x}^0 &= (1\ 1\ 1)^\top , \\
\mathbf{x}^1 &= \max\left\{\left\lceil \tfrac{1}{16}A_G(1\ 1\ 1)^\top \right\rceil, (1\ 1\ 1)^\top\right\} = \max\left\{(1\ 1\ 2)^\top, (1\ 1\ 1)^\top\right\} = (1\ 1\ 2)^\top , \\
\mathbf{x}^2 &= \max\left\{\left\lceil \tfrac{1}{16}A_G(1\ 1\ 2)^\top \right\rceil, (1\ 1\ 2)^\top\right\} = \max\left\{(1\ 2\ 2)^\top, (1\ 1\ 2)^\top\right\} = (1\ 2\ 2)^\top ,
\end{aligned}$$

and $A_G\mathbf{x}^2 \leq 16\mathbf{x}^2$. Hence, by Proposition 8.8(d), the vector $(1\ 2\ 2)^\top$ is the unique minimum (A_G, n)-super-vector $\mathbf{x}^*$. □

8.8.2 Consistent splittings

As mentioned before, the state-splitting construction of a sliding-block compressible (S, n)-excoder $\mathcal{E}$ starts with a deterministic presentation G of S. Typically, S will be a q-th power of a given constraint S' and G will be the q-th power of a graph presentation of S'; the integer n will be 2^p and $\mathcal{E}$ will thus be a sliding-block compressible excoder for S' at rate $p : q$.

Given S, G, and n, we compute an (A_G, n)-super-vector $\mathbf{x}$ using the algorithm in Figure 8.4. Just as in Section 5.2.1, the entry x_u in $\mathbf{x}$ will be referred to as the *weight of state* u. Using the vector $\mathbf{x}$, we will transform the graph G through out-splitting operations into a graph H such that $(1\ 1\ \ldots\ 1)^\top$ is an (A_H, n)-super-vector (i.e., the weights are all reduced to 1). It is easy to see that this is equivalent to saying that H has out-degree at most n. An (S, n)-excoder $\mathcal{E}$ will then be obtained by assigning tags to the edges of H.

Next we discuss the role of the (A_G, n)-super-vector in more detail. For an edge e in a graph, recall that $\tau_G(e) = \tau(e)$ denotes the terminal state of e.

Given a graph G, a positive integer n, and an (A_G, n)-super-vector $\mathbf{x} = (x_u)_u$, an $\mathbf{x}$-*consistent partition* of G is defined by partitioning the set, E_u, of outgoing edges from each state u in G such that

$$\sum_{e \in E_u^{(r)}} x_{\tau(e)} \leq n x_u^{(r)} \qquad \text{for} \qquad r = 1, 2, \ldots, N = N(u)\ , \tag{8.4}$$

where $x_u^{(r)}$ are nonnegative integers and

$$\sum_{r=1}^{N(u)} x_u^{(r)} = x_u\ . \tag{8.5}$$

The out-splitting based upon such a partition is called an $\mathbf{x}$-*consistent splitting*. The splitting is called *non-trivial* if at least one state u has at least two descendants $u^{(r)}, u^{(t)}$ such that both $x_u^{(r)}$ and $x_u^{(t)}$ are strictly positive. Note that here we have used the same terminology as in Section 5.2.3, but the reader should understand that throughout the remainder of this chapter, the notions of $\mathbf{x}$-*consistent partition* and $\mathbf{x}$-*consistent splitting* are as defined in this paragraph.

Let G' denote the graph after splitting. It is easy to see that the (A_G, n)-super-vector $\mathbf{x}$ gives rise to an *induced* $(A_{G'}, n)$-*super-vector* $\mathbf{x}' = (x'_u)_u$; namely, set $x'_{u^{(r)}} = x_u^{(r)}$. Note that (8.5) asserts that the weights of the descendants of a state u sum to the weight of u.

Example 8.15 Let S be presented by the graph $G = G^7_{2,\infty}$. For each state $u \in \{0, 1, 2\}$ in G, denote by $\mathcal{L}_u$ the set of labels of the outgoing edges from state u in G. Note that the graph G has memory 1, since the label of an edge determines the terminal state of that edge: edges whose labels end with '1' terminate in state 0, edges whose labels end with '10' terminate in state 1, and the remaining edges terminate in state 2. Hence, each set $\mathcal{L}_u$ completely describes the set, E_u, of outgoing edges from state u. We have

$$
\begin{aligned}
\mathcal{L}_0 &= \{0000000, 0000001, 0000010, 0000100, 0001000, 0001001, 0010000, 0010001, 0010010\} \; ; \\
\mathcal{L}_1 &= \{0000000, 0000001, 0000010, 0000100, 0001000, 0001001, 0010000, 0010001, 0010010 \\
&\qquad 0100000, 0100001, 0100010, 0100100\} \; ; \\
\mathcal{L}_2 &= \{0000000, 0000001, 0000010, 0000100, 0001000, 0001001, 0010000, 0010001, 0010010, \\
&\qquad 0100000, 0100001, 0100010, 0100100, 1000000, 1000001, 1000010, 1000100, 1001000, 1001001\} \; .
\end{aligned}
$$

We have shown in Example 8.14 that $\mathbf{x} = (1\ 2\ 2)^\top$ is an $(A_G, 16)$-super-vector. We next perform an $\mathbf{x}$-consistent splitting on G which will result in a graph H in which each state has weight 1. That is, up to tagging, the graph H will be an $(S, 16)$-excoder, namely, a rate $4 : 7$ excoder for the $(2, \infty)$-RLL constraint.

Since state 0 has weight 1, it will not be split (or more precisely, we split it trivially into one state, namely itself). Since each of the states 1 and 2 has weight 2, we would like to split each into two states of weight 1. Define the *weight of an edge* to be the weight of its terminal state. Now, $A_G (1\ 2\ 2)^\top = (15\ 22\ 32)^\top$, indicating that the total weights of outgoing edges from states 0, 1, and 2 are 15, 22, and 32, respectively. If we can partition the sets of outgoing edges from state 1 and state 2, each into two subsets of edges of total weight at most 16, then it follows that the weights in the graph H obtained from the corresponding out-splitting will all be 1, as desired. This indeed can be done as follows, where each partition element $E_u^{(r)}$ is represented by the respective label set $\mathcal{L}_u^{(r)}$ (labels that correspond to edges with weight 2 are underlined):

$$
\begin{aligned}
\mathcal{L}_0 &= \{\underline{0000000}, 0000001, \underline{0000010}, \underline{0000100}, \underline{0001000}, 0001001, \underline{0010000}, 0010001, \underline{0010010}\} \; ; \\
\mathcal{L}_1^{(1)} &= \{\underline{0000000}, 0000001, \underline{0000010}, \underline{0000100}, \underline{0001000}, 0001001, \underline{0010000}, 0010001, \underline{0010010}\} \; ; \\
\mathcal{L}_1^{(2)} &= \{\underline{0100000}, 0100001, \underline{0100010}, \underline{0100100}\} \; ; \\
\mathcal{L}_2^{(1)} &= \{\underline{0000000}, 0000001, \underline{0000010}, \underline{0000100}, \underline{0001000}, 0001001, \underline{0010000}, 0010001, \underline{0010010}, 1001001\} \; ; \\
\mathcal{L}_2^{(2)} &= \{\underline{0100000}, 0100001, \underline{0100010}, \underline{0100100}, \underline{1000000}, 1000001, \underline{1000010}, \underline{1000100}, \underline{1001000}\} \; .
\end{aligned}
$$

The reader can verify that the sets E_0, $E_1^{(1)}$, $E_1^{(2)}$, $E_2^{(1)}$, and $E_2^{(2)}$ have total weights 15, 15, 7, 16, and 16, respectively, as desired (in fact, the weights of $E_2^{(1)}$ and $E_2^{(2)}$ are forced to be 16).

The resulting split graph H will have five states, 0, $1^{(1)}$, $1^{(2)}$, $2^{(1)}$, and $2^{(2)}$, and the induced (A_H, n)-super-vector is $\mathbf{x}' = (1\ 1\ 1\ 1\ 1)^\top$, implying that the row sums of the adjacency matrix

$$A_H = \begin{pmatrix} 3 & 2 & 2 & 4 & 4 \\ 3 & 2 & 2 & 4 & 4 \\ 1 & 1 & 1 & 2 & 2 \\ 4 & 2 & 2 & 4 & 4 \\ 2 & 2 & 2 & 5 & 5 \end{pmatrix}$$

are all at most 16. □

The following modification of of the corresponding result (Proposition 5.7) for conventional state splitting shows that in general there always is an $\mathbf{x}$-consistent splitting whenever we need one.

Proposition 8.9 *Let G be an irreducible graph which does not have out-degree at most n and let $\mathbf{x}$ be an (A_G, n)-super-vector. Then there is a non-trivial $\mathbf{x}$-consistent splitting of G.*

Proof. By the assumption, some state u in G has out-degree greater than n. By the pigeon-hole principle, there is a subset E of E_u with at most n edges such that n divides $\sum_{e \in E} x_{\tau(e)}$. Partition E_u into two sets $E_u^{(1)} \cup E_u^{(2)}$ where

$$E_u^{(1)} = E \qquad \text{and} \qquad E_u^{(2)} = E_u \setminus E ,$$

and set

$$x_u^{(1)} = (1/n)\Big(\sum_{e \in E} x_{\tau(e)}\Big) \qquad \text{and} \qquad x_u^{(2)} = x_u - x_u^{(1)} .$$

It can be readily verified that the partition $E_u^{(1)} \cup E_u^{(2)}$ indeed implies a non-trivial $\mathbf{x}$-consistent splitting of state u. □

Passage from the (A_G, n)-super-vector $\mathbf{x}$ to the induced $(A_{G'}, n)$-super-vector $\mathbf{x}'$ always preserves the super-vector sum and increases the number of states. Since a super-vector is always a positive integer vector, it follows that repeated applications of Proposition 8.9 beginning with G eventually yield a graph H with an (A_H, n)-super-vector $(1\ 1\ \ldots\ 1)^\top$ and therefore with out-degree at most n. As mentioned earlier, if the original presentation G has finite memory m, then H will be (m, a)-definite on S. Finally, we assign tags to the edges of H, thereby obtaining an (m, a)-sliding-block compressible (S, n)-excoder $\mathcal{E}$. This establishes Proposition 8.7.

It may well be possible to merge states in H, resulting in a simpler excoder. Suppose that u and v are states in H such that every word that can be generated in H by paths starting at u can also be generated by paths starting at v. We can merge state u into state v by redirecting all incoming edges to u into v and then deleting state u with all its outgoing edges. This is the direct analogy of merging in Section 5.5.1.

Merging can add new words to those that are generated by H (and $\mathcal{E}$), and it may also give rise to new paths that present words of S. In particular, it may destroy definiteness on S. However, suppose that there are integers $\mathsf{m}' \geq \mathsf{m}$ and $\mathsf{a}' \geq \mathsf{a}$ such that the following holds: for every word $\mathbf{w} \in S_{\mathsf{m}'}$ that can be generated by a path in H that terminates in u and for every word $\mathbf{w}' \in S_{\mathsf{a}'+1}$ that can be generated in H from v but not from u, we have $\mathbf{w}\mathbf{w}' \notin S_{\mathsf{m}'+\mathsf{a}'+1}$. Under this condition, the merged graph is $(\mathsf{m}', \mathsf{a}')$-definite on S, since definiteness on S involves only words in S, which may be a proper subset of the constraint presented by the merged graph.

Example 8.16 Continuing the discussion in Example 8.15, we observe that $\mathcal{L}_1^{(2)} \subset \mathcal{L}_2^{(2)}$; furthermore, since G has memory 1, edges with the same label in $E_1^{(2)}$ and $E_2^{(2)}$ terminate in the same state of G. It follows that every word that can be generated in H from $1^{(2)}$ can also be generated from $2^{(2)}$.

Let $\mathbf{w}$ be a word that is generated by a path in H that terminates in state $1^{(2)}$. This word is also generated in G by a path that terminates in state 1 and, so, $\mathbf{w}$ ends with '10'. Let $\mathbf{w}'$ be a word that can be generated in H from $2^{(2)}$ but not from $1^{(2)}$. The word $\mathbf{w}'$ necessarily starts with '1' since

$$\mathcal{L}_2^{(2)} \setminus \mathcal{L}_1^{(2)} = \{1000000, 1000001, 1000010, 1000100, 1001000\} .$$

It follows that $\mathbf{w}\mathbf{w}'$ contains the sub-word '101'; as such, it violates the $(2, \infty)$-RLL constraint and, therefore, it does not belong to S. We conclude that the definiteness on S will be preserved upon merging state $1^{(2)}$ into state $2^{(2)}$. Similarly, since $\mathcal{L}_0 = \mathcal{L}_1^{(1)} \subset \mathcal{L}_2^{(1)}$ and $\mathcal{L}_2^{(1)} \setminus \mathcal{L}_0 = \{1001001\}$, we see that states 0 and $1^{(1)}$ can be merged into state $2^{(1)}$ while preserving definiteness on S. Thus, the resulting merged graph, H', has only two states, $a \equiv 2^{(1)}$ and $b \equiv 2^{(2)}$, and it is $(1,1)$-definite on S. In fact, H' is $(0,1)$-definite on S: since $\mathcal{L}_2^{(1)} \cap \mathcal{L}_2^{(2)} = \emptyset$, the initial state of the edge that generates the current codeword is uniquely determined by that codeword.

Finally, we assign tags from $\{0,1\}^4$ to the edges of H' to obtain the $(0,1)$-sliding-block compressible excoder $\mathcal{E}_{2,\infty}$ whose transition table is shown in Table 8.3. In that table, the rows are indexed by the tags (4-blocks) and the columns by the states of $\mathcal{E}_{2,\infty}$. Entry (s, u) in the table contains the label and terminal state of the outgoing edge from state u that is tagged by s. Observe that the tags have been assigned to the edges of $\mathcal{E}_{2,\infty}$ so that tags of edges that have the same label (and initial state) will differ in only the last bit. Thus, whenever the compression of the current 4-block depends on the upcoming 7-codeword, such dependency is limited only to determining the last bit of the 4-block. □

In Table 8.4, we list for each $p \leq 16$ the largest value of q which allows a block excoder at rate $p : q$ as well as the largest value of q which allows a $(0,1)$-sliding-block compressible excoder at rate $p : q$ for the $(2, \infty)$-RLL constraint. The values of q for the block excoder

	a	b
0000	0000000, a	0100000, a
0001	0000000, b	0100000, b
0010	0000010, a	0100100, a
0011	0000010, b	0100100, b
0100	0000100, a	1000000, a
0101	0000100, b	1000000, b
0110	0001000, a	1000100, a
0111	0001000, b	1000100, b
1000	0010000, a	1001000, a
1001	0010000, b	1001000, b
1010	0010010, a	0100001, a
1011	0010010, b	1000001, a
1100	0000001, a	0100010, a
1101	0001001, a	0100010, b
1110	0010001, a	1000010, a
1111	1001001, a	1000010, b

Table 8.3: Excoder $\mathcal{E}_{2,\infty}$.

p	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
q (block)	1	3	4	6	8	10	11	13	15	17	19	21	22	24	26	28
q ((0,1)-sliding block)	1	3	**5**	**7**	**9**	10	**12**	**14**	**16**	**18**	19	21	**23**	**25**	**27**	**29**

Table 8.4: Maximal values of q for existence of block excoders and $(0,1)$-sliding-block compressible excoders for the $(2,\infty)$-RLL constraint.

case can be obtained by (8.1), where the values of $|S_q|$ can be computed using the formulas in [Imm91, Section 5.2] (it can be verified that the same values of q apply also to $(0,0)$-sliding-block compressible excoders). The values of q for the $(0,1)$-sliding-block compressible case were obtained by actually computing $(A^q_{G_{2,\infty}}, 2^p)$-super-vectors and verifying that excoders can be obtained by one round of out-splitting. In particular, the rate $4:7$ is attained by the excoder $\mathcal{E}_{2,\infty}$ in Example 8.16. It is worthwhile pointing out that a rate $5:9=.5555\ldots$ is attainable by a $(0,1)$-sliding-block compressible excoder with ten states. This rate is above the capacity ($\approx .5515$) by less than 1%. The boldface numbers in Table 8.4 indicate values of q which are larger (by 1) than those attainable by block codes. The results in Table 8.4 remain unaffected if one were to replace the $(2,\infty)$-RLL constraint with the $(2,15)$-RLL constraint.

We also mention here the existence of the following two $(0,1)$-sliding-block compressible excoders that might be of practical interest: a rate $8:9$ excoder for the $(0,2)$-RLL constraint, and a rate $5:7$ excoder for the $(1,7)$-RLL constraint; there are no block excoders at such rates for those constraints.

8.8.3 Reduction of edge effect in error propagation

Recall that in the analysis of Section 8.7, we made the conservative assumption that a p-block is wholly corrupted even if only one bit in that p-block is in error. Yet, as we know for block codes, and as we have demonstrated in Example 8.16 for sliding-block compressible excoders, special care in the assignment of tags can reduce the dependency of certain bits in p-blocks on certain channel bits, thereby reducing the effect of error. Specifically, when using the excoder $\mathcal{E}_{2,\infty}$ of Example 8.16, a burst of $L = 40$ channel bits, while affecting up to eight 4-blocks, can corrupt only up to 29 bits (and not 32 bits) in those blocks. This, in turn, allows us to modify Table 8.2 to produce Table 8.5 (the modified entries are indicated by boldface numbers). It follows from Table 8.5 that for the range $841 \leq \kappa \leq 7,778$, the actual

B	$D(40, B)$	ν	ρ	κ
4	8	544	64	840
5	**7**	**1,155**	**70**	**1,898**
6	6	2,340	(72)	3,969
7	**5**	**4,515**	**70**	**7,778**
8	5	10,280	80	17,850

Table 8.5: Modified Table 8.2.

redundancy that will be required is strictly less than dictated by Table 8.2. In particular, for $3,970 \leq \kappa \leq 7,778$, the savings amount to reducing the redundancy from 80 to 70 bits.

Additional savings can be obtained by using an excoder with more states, as we demonstrate in the next example.

Example 8.17 Looking closely at Table 8.3, one can see that the compression of the last bit of the current 4-block depends on the first, second, fourth, and seventh bits of the upcoming 7-codeword. The dependency on the seventh bit has a slight disadvantage in case of short bursts—in particular isolated *bit-shift errors*, where an occurrence of '1' in a constrained sequence is shifted by one position, thereby resulting in two adjacent erroneous bits in the constrained sequence. If those two bits cross the boundary of adjacent 7-codewords, the error may propagate through the compression to up to 9 bits in three 4-blocks.

By allowing more states, we are able to present another rate $4 : 7$ excoder for the $(2, \infty)$-RLL constraint, $\mathcal{E}'_{2,\infty}$, where the compression of the last bit of the current 4-block depends on the first, second, fourth, and fifth bits of the upcoming 7-codeword (whereas the other bits of the 4-block depend only on the current 7-codeword). Here, one bit-shift error in the constrained sequence may affect only two 4-blocks. The excoder $\mathcal{E}'_{2,\infty}$ is obtained through a different out-splitting of state 2 in G (into states $2^{(1')}$ and $2^{(2')}$), resulting in four states, $\alpha \equiv 0 = 1^{(1)}$, $\beta \equiv 1^{(2)}$, $\gamma \equiv 2^{(1')}$, and $\delta \equiv 2^{(2')}$, with a transition table as shown in Table 8.6. The excoder $\mathcal{E}'_{2,\infty}$ is $(1, 1)$-definite (but not $(0, 1)$-definite) on the constraint; still,

	α	β	γ	δ
0000	0000000, γ	0100010, α	0000000, γ	0100010, α
0001	0000000, δ	0100010, β	0000000, δ	0100010, β
0010	0000010, α	0100000, γ	0000010, α	0100000, γ
0011	0000010, β	0100000, δ	0000010, β	0100000, δ
0100	0000100, γ	–	0000100, γ	1000010, α
0101	0000100, δ	–	0000100, δ	1000010, β
0110	0010000, γ	–	0010000, γ	1000000, γ
0111	0010000, δ	–	0010000, δ	1000000, δ
1000	0010010, α	–	0010010, α	1001001, α
1001	0010010, β	0100001, α	0010010, β	0100001, α
1010	0001000, γ	0100100, γ	0100100, γ	0001000, γ
1011	0001000, δ	0100100, δ	0100100, δ	0001000, δ
1100	–	–	1000100, γ	1000001, α
1101	0001001, α	–	1000100, δ	0001001, α
1111	0000001, α	–	0000001, α	1001000, γ
1111	0010001, α	–	0010001, α	1001000, δ

Table 8.6: Excoder $\mathcal{E}'_{2,\infty}$.

the particular tagging of the edges makes it $(0,1)$-sliding-block compressible. Note that states α and β have less than 16 outgoing edges and, so, certain elements of $\{0,1\}^4$ do not tag those edges. $\square$

Problems

To be filled in.

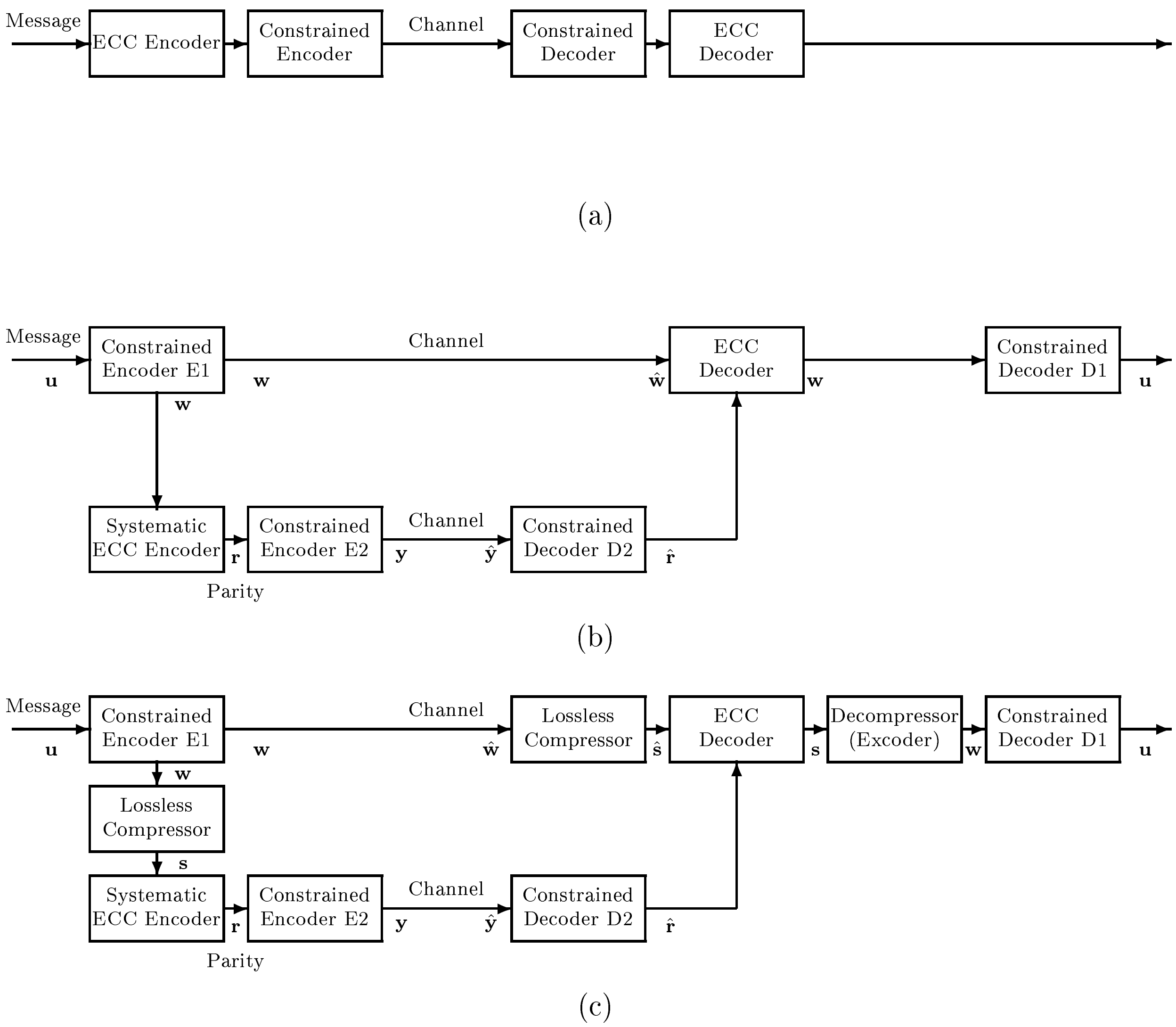

Figure 8.1: (a) Standard concatenation. (b) Modified concatenation. (c) Modified concatenation with lossless compression.

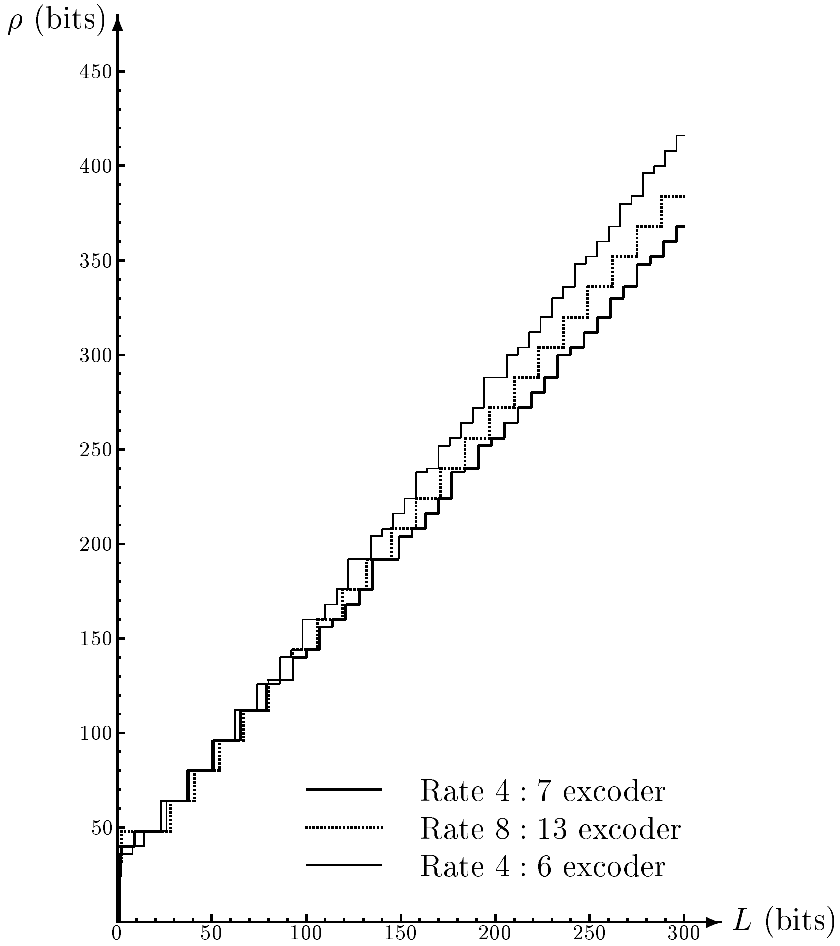

Figure 8.2: Redundancy for 4,096 user bits and various burst lengths using three different excoders for the $(2, \infty)$-RLL constraint.

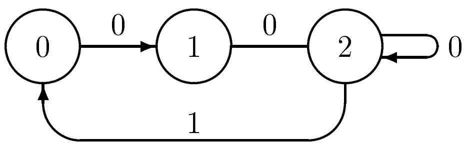

Figure 8.3: Graph presentation $G_{2,\infty}$ of the $(2, \infty)$-RLL constraint.

$\mathbf{x} \leftarrow (1\ 1\ \ldots\ 1)^\top$;
while ($A\mathbf{x} \not\leq n\mathbf{x}$)
    $\mathbf{x} \leftarrow \max\left\{\left\lceil \frac{1}{n} A\mathbf{x} \right\rceil, \mathbf{x}\right\}$;   /* apply $\lceil\cdot\rceil$ and $\max\{\cdot,\cdot\}$ componentwise */
return $\mathbf{x}$;

Figure 8.4: Reversed Franaszek algorithm for computing (A, n)-super-vectors.