Chapter 9

LDPC codes

Low Density Parity Check (LDPC) codes make up a class of block codes that are characterized by a sparse parity check matrix. They were first described in Gallager's thesis at the beginning of the 60s [9.21]. Apart from the hard input decoding of LDPC codes, this thesis proposed iterative decoding based on belief propagation (BP). This work was forgotten for 30 years. Only a few rare studies referred to it during this dormant period, in particular, Tanner's which proposed a generalization of the Gallager codes and a bipartite graph [9.53] representation.

After the invention of turbo codes, LDPC codes were rediscovered in the middle of the 90s by MacKay *et al.* [9.39], Wilberg [9.64] and Sipser *et al.* [9.52]. Since then, considerable progress concerning the rules for building good LDPC codes, and coding and decoding techniques, have enabled LDPC codes to be used, like turbo codes, in practical applications.

This chapter gives an overview of the encoding and decoding of LDPC codes, and some considerations about hardware implementations.

9.1 Principle of LDPC codes

LDPC codes are codes built from the simplest elementary code: the single parity check code. We therefore begin this chapter by describing the single parity check code and its soft in soft out decoding before dealing with the construction of LDPC codes.

9.1.1 Parity check code

Definition

A parity equation, represented graphically by Figure 9.1, is an equation linking n binary data to each other by the *exclusive or*, denoted $\oplus$ operator. It is satisfied if the total number of 1s in the equation is even or null.

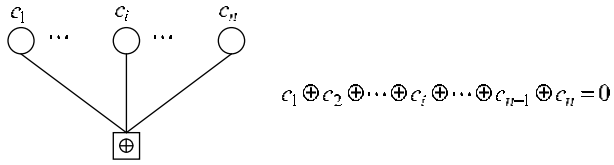

Figure 9.1 – Graphic representation of a parity equation.

The circles represent the binary data c_i, also called *variables*. The rectangle containing the *exclusive or* operator represents the parity equation (also called the parity constraint, or parity). The links between the variables and the operator indicate the variables involved in the parity equation.

Parity code with three bits

We consider that the binary variables c_1, c_2 and c_3 are linked by the parity constraint $c_1 \oplus c_2 \oplus c_3 = 0$, and that they make up the codeword (c_1, c_2, c_3). We assume that we know the log likelihood ratio (LLR) $L(c_1)$ and $L(c_2)$ of variables c_1 and c_2: what can we then say about the LLR $L(c_3)$ of variable c_3?
We recall that $L(c_j)$ is defined by the equation:

$$L\left(c_j\right) = \ln\left(\frac{\Pr\left(c_j = 1\right)}{\Pr\left(c_j = 0\right)}\right) \tag{9.1}$$

There are two codewords in which bit c_3 is equal to 0: codewords (0,0,0) and (1,1,0). Similarly, there are two codewords in which bit c_3 is equal to 1: codewords (1,0,1) and (0,1,1). We deduce from this the following two equations in the probability domain:

$$\begin{cases} \Pr(c_3 = 1) = \Pr(c_1 = 1) \times \Pr(c_2 = 0) + \Pr(c_1 = 0) \times \Pr(c_2 = 1) \\ \Pr(c_3 = 0) = \Pr(c_1 = 0) \times \Pr(c_2 = 0) + \Pr(c_1 = 1) \times \Pr(c_2 = 1) \end{cases} \tag{9.2}$$

Using the expression of each probability according to the likelihood ratio function, deduced from Equation (9.1):

$$\begin{cases} \Pr(c_j = 1) = \frac{\exp(L(c_j))}{1+\exp(L(c_j))} \\ \\ \Pr(c_j = 0) = 1 - \Pr(c_j = 1) = \frac{1}{1+\exp(L(c_j))} \end{cases}$$

we have:

$$L(c_3) = \ln\left[\frac{1+\exp(L(c_2)+L(c_1))}{\exp(L(c_2))+\exp(L(c_1))}\right] \triangleq L(c_1) \oplus L(c_2) \tag{9.3}$$

Equation (9.3) enables us to define the switching operator $\oplus$ between the two LLRs of the variables c_1 and c_2.

Applying function $\tanh(x/2) = \frac{\exp(x)-1}{\exp(x)+1}$ to Equation (9.3), the latter becomes:

$$\begin{aligned}\tanh\left(\frac{L(c_3)}{2}\right) &= \frac{\exp(L(c_0))-1}{\exp(L(c_0))+1} \times \frac{\exp(L(c_1))-1}{\exp(L(c_1))+1} \\ &= \prod_{j=0}^{1} \tanh\left(\frac{L(c_j)}{2}\right)\end{aligned} \tag{9.4}$$

It is practical (and frequent) to separate the processing of the sign and the magnitude in Equation (9.4) which can then be replaced by the following two equations:

$$\operatorname{sgn}(L(c_3)) = \prod_{j=0}^{1} \operatorname{sgn}(L(c_j)) \tag{9.5}$$

$$\tanh\left(\frac{|L(c_3)|}{2}\right) = \prod_{j=0}^{1} \tanh\left(\frac{|L(c_j)|}{2}\right) \tag{9.6}$$

where the sign function $\operatorname{sgn}(x)$ is such that $\operatorname{sgn}(x) = +1$ if $x \geq 0$ and $\operatorname{sgn}(x) = -1$ otherwise.

Processing the magnitude given by Equation (9.6) can be simplified by taking the inverse of the logarithm of each of the terms of the equation, which gives:

$$|L(c_3)| = f^{-1}\left(\sum_{j=1,2} f(|L(c_j)|)\right) \tag{9.7}$$

where function f, satisfying $f^{-1}(x) = f(x)$, is defined by:

$$f(x) = \ln(\tanh(x/2)) \tag{9.8}$$

These different aspects of the computation of function $\oplus$ will be developed in the architecture part of this chapter.

Expression (9.6) in fact corresponds to the computation of (9.3) in the Fourier domain. Finally, there is also a third writing of the LLR of variable c_2 [9.65, 9.23]:

$$\begin{aligned}L(c_3) = \operatorname{sign}(L(c_1))\operatorname{sign}(L(c_2))\min(|L(c_1)|, |L(c_2)|) \\ - \ln(1+\exp(-|L(c_1)-L(c_2)|)) \\ + \ln(1+\exp(-|L(c_1)+L(c_2)|))\end{aligned} \tag{9.9}$$

This other expression of operator $\oplus$ can easily be processed by using a look-up table for function g defined by:

$$g(x) = \ln\left(1 + \exp\left(-|x|\right)\right) \tag{9.10}$$

Practical example
Let us assume that $\Pr(c_1 = 1) = 0.8$ and $\Pr(c_2 = 1) = 0.1$. We then have,

$$\Pr(c_1 = 0) = 0.2 \quad \text{et} \quad \Pr(c_2 = 0) = 0.9$$

It is therefore more probable that $c_1 = 1$ and $c_2 = 0$. A direct application of Equation (9.2) then gives

$$\Pr(c_3 = 0) = 0.26 \quad \text{and} \quad \Pr(c_3 = 1) = 0.74$$

c_3 is therefore more probably equal to 1, which intuitively is justified since the number of 1s belonging to the parity check equation must be even. Using Equation (9.3) gives

$$L(c_3) = L(c_1) \oplus L(c_2) = (-1.386) \oplus (2.197) = -1.045$$

that is, $\Pr(c_3 = 0) = 0.26$ again. We find the same result again using (9.7) and (9.9).

Parity check code with n bits

We can now proceed to the parity check equation with n bits. We consider that the binary variables $c_1, \cdots, c_n$ are linked by the parity constraint $c_1 \oplus \cdots \oplus c_n = 0$ and that they make up the codeword $(c_1, \cdots, c_n)$. The LLR of the variables $\{c_j\}_{j=1..n, j\neq i}$ is assumed to be known and we search for the LLR of variable c_i. It is then simple to generalize the equations obtained for the parity code with 3 bits. Thus, taking the operator defined by (9.2) again:

$$L(c_i) = L(c_1) \oplus L(c_2) \oplus \cdots \oplus L(c_{j\neq i}) \oplus \cdots \oplus L(c_n) = \bigoplus_{j\neq i} L(c_j) \tag{9.11}$$

Similarly, the hyperbolic tangent rule is expressed by:

$$\tanh\left(\frac{L(c_i)}{2}\right) = \prod_{j\neq i} \tanh\left(\frac{L(c_j)}{2}\right) \tag{9.12}$$

or, separating the sign and the magnitude:

$$\operatorname{sgn}\left(L(c_i)\right) = \prod_{j\neq i} \operatorname{sgn}\left(L(c_j)\right) \tag{9.13}$$

$$|L(c_i)| = f^{-1}\left(\sum_{j\neq i} f\left(|L(c_j)|\right)\right) \tag{9.14}$$

where f is defined by Equation (9.8).

9.1.2 Definition of an LDPC code

Linear block codes

Linear block codes (see Chapter 4) can be defined by a parity check matrix $\mathbf{H}$ of size $m \times n$, where $m = n - k$. This matrix can be seen as a linear system of m parity check equations. The words $\mathbf{c}$ of the code defined by $\mathbf{H}$ are the binary words whose n bits simultaneously satisfy the m parity check equations. This system of linear equations is represented graphically in Figure 9.2 for the case of the Hamming binary block code of length $n = 7$.

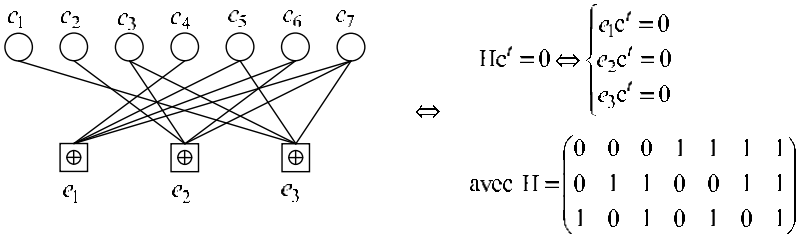

Figure 9.2 – Graphic representation of a block code: example of a Hamming code of length 7.

Such a representation is called the *bipartite graph* of the code. In this graph, branches link two different classes of nodes to each other:

- The first class of nodes called *variable nodes*, correspond to the bits of the codewords ($c_j,\ j \in \{1, \cdots, n\}$), and therefore to the columns of $\mathbf{H}$.
- The second class of nodes, called *parity check nodes*, correspond to the parity check equations ($e_p,\ p \in \{1, \cdots, m\}$), and therefore to the rows of $\mathbf{H}$.

Thus, to each branch linking a variable node c_j to a parity check node e_p corresponds the 1 that is situated at the intersection of the j-th column and the p-th row of the parity check matrix.

By convention, we denote $P(j)$ (respectively $J(p)$) all the indices of the parity nodes (respectively variable nodes) connected to the variable with index j (respectively to the parity with index p). We denote by $P(j)\backslash p$ (respectively $J(p)\backslash j$) all the $P(j)$ not having index p (respectively, all the $J(p)$ not having index j). Thus, in the example of Figure 9.2, we have

$$P(5) = \{1, 3\} \quad \textit{and} \quad J(1)\backslash 5 = \{4, 6, 7\}$$

A *cycle* on a bipartite graph is a path on the graph which makes it possible to leave a node and to return to this same node without passing twice through the same branch. The size of a cycle is given by the number of branches contained

in the cycle. The graph being bipartite, the size of the cycles is even. The size of the shortest cycle in a graph is called the *girth*. The presence of cycles in the graph may degrade the decoding performance by a phenomenon of self-confirmation during the propagation of the messages. Figure 9.3 illustrates two cycles of sizes 4 and 6.

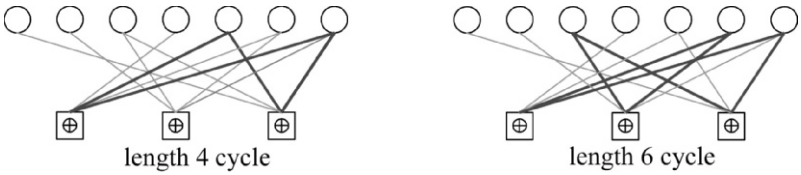

Figure 9.3 – Cycles in a bipartite graph.

Low density parity check codes

LDPC codes are linear block codes, the term low density coming from the fact that parity check matrix **H** contains a low number of non null values: it is a *sparse* matrix. In the particular case of binary LDPC codes studied here, the parity check matrix contains a small number of 1s. In other words, the associated bipartite graph contains a small number of branches. The adjective "low" mathematically means that when the length n of a codeword increases, the number of 1s in the matrix increases in $O(n)$ (compared to an increase in $O(n^2)$ of the number of elements of the matrix if the rate remains fixed).

The class of LDPC codes generates a very large number of codes. It is convenient to divide them into two sub-classes:

- regular LDPC codes
- irregular LDPC codes

An LDPC code is said to be regular in the particular case where parity check matrix **H** contains a constant number d_c of 1s in each row, and a constant number d_v of 1s in each column. We then say that the variables are of degree d_v and that the parities are of degree d_c. The code is denoted a (d_v,d_c) regular LDPC code. For example, the parity check matrix of a regular LDPC code (3,6) contains only 3 non zero values in each column, and 6 non zero values in each row. Figure 9.4 presents an example of a regular (3,6) code of size $n = 256$ obtained by drawing randomly the non zero positions. Out of the 256x128 inputs of the matrix, only 3x256 are non zero, that is, around 2.3%. This percentage tends towards 0 if the size of the code, for a fixed rate, tends towards infinity.

The irregularity profile of the variables of an irregular LDPC code is defined by the polynomial $\lambda(x) = \sum \lambda_j x^{j-1}$ where coefficient λ_j is equal to the ratio between the accumulated number of 1s in the columns (or variable) of degree j

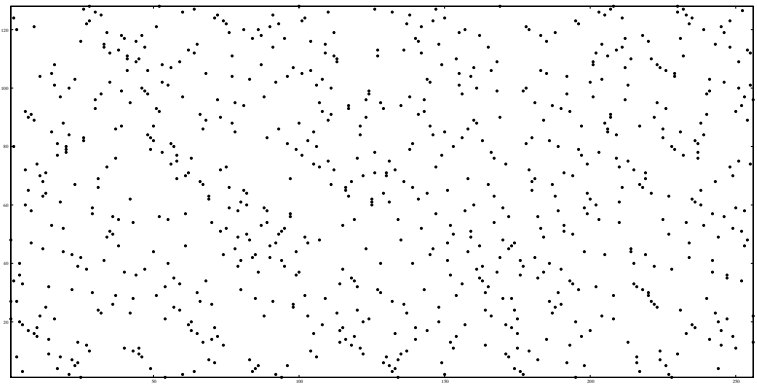

Figure 9.4 – Parity check matrix of a regular (3,6) LDPC code of size $n = 256$ and rate $R = 0,5$.

and the total number E of 1s in matrix $\mathbf{H}$. For example, $\lambda(x) = 0,2x^4 + 0,8x^3$ indicates a code where 20% of the 1s are associated with variables of degree 5 and 80% with variables of degree 4. Note that, by definition, $\lambda(1) = \sum \lambda_j = 1$. Moreover, the proportion of variables of degree j in the matrix is given by

$$\bar{\lambda}_j = \frac{\lambda_j / j}{\sum\limits_k \lambda_k / k}$$

Symmetrically, the irregularity profile of the parities is represented by the polynomial $\rho(x) = \sum \rho_p x^{p-1}$, coefficient ρ_p being equal to the ratio between the accumulated number of 1s in the rows (or parity) of degree p and the total number of 1s denoted E. Similarly, we obtain $\rho(1) = \sum \rho_j = 1$. The proportion $\bar{\rho}_p$ of columns of degree p in matrix $\mathbf{H}$ is given by

$$\bar{\rho}_p = \frac{\rho_p / p}{\sum\limits_k \rho_k / k}$$

Irregular codes have more degrees of freedom than regular codes and it is thus possible to optimize them more efficiently: their asymptotic performance is better than that of regular codes.

Coding rate

Consider a parity equation of degree d_c. It is possible to arbitrarily fix the values of the $d_c - 1$ first bits; only the last bit is constrained and corresponds to the redundancy. Thus, in a parity matrix $\mathbf{H}$ of size (m,n), each of the m rows corresponds to 1 redundancy bit. If the m rows of $\mathbf{H}$ are independent, the code then has m redundancy bits. The total number of bits of the code being

n, the number of information bits is then $k = n - m$ and the coding rate is $R = (n - m)/n = 1 - m/n$. Note that in the case where the m rows are not independent (for example, two identical rows), the number of constrained bits is lower than m. We then have $R > 1 - m/n$.

In the case of a regular LDPC code (d_v,d_c), each of the m rows has d_c non zero values, that is, a total of $E = md_c$ non zero values in matrix $\mathbf{H}$. Symetrically, each of the n columns contains d_v non zero values. We deduce from this that E satisfies $E = nd_v = md_c$, that is, $m/n = d_v/d_c$. The rate of such a code then satisfies $R \geqslant (1 - d_v/d_c)$.

In the case of an irregular code, the expression of the rate is generalized, taking into account each degree weighted by:

$$R \geqslant 1 - \frac{\sum_p \rho_p/p}{\sum_j \lambda_j/j} \tag{9.15}$$

Equality is reached if matrix $\mathbf{H}$ is of rank m.

9.1.3 Encoding

Encoding an LDPC code can turn out to be relatively complex if matrix $\mathbf{H}$ does not have a particular structure. There exist generic encoding solutions, including an algorithm with complexity in $O(n)$, requiring complex preprocessing on the matrix $\mathbf{H}$. Another solution involves directly building matrix $\mathbf{H}$ so as to obtain a systematic code very simple to encode. It is this solution, in particular, that was adopted for the standard DVB-S2 code for digital television transmission by satellite.

Generic encoding

Encoding with a generator matrix

LDPC codes being linear block codes, the coding can be done via the generator matrix $\mathbf{G}$ of size $k \times n$ of the code, such as defined in Chapter 4. As we have seen, LDPC codes are defined from their parity check matrix $\mathbf{H}$, which is generally not systematic. A transformation of $\mathbf{H}$ into a systematic matrix $\mathbf{H}_{sys}$ is possible, for example with the Gaussian elimination algorithm. This relatively simple technique, however, has a major drawback: the generator matrix $\mathbf{G}_{sys}$ of the systematic code is generally not sparse. The coding complexity increases rapidly in $\mathrm{O}\left(n^2\right)$, which makes this operation too complex for usual length codes.

Coding with linear complexity

Richardson *et al.* [9.49] proposed a solution enabling quasi-linear complexity encoding, as well as *greedy* algorithms making it possible to preprocess parity check matrix $\mathbf{H}$. The aim of the preprocessing is to put $\mathbf{H}$ as close as possible to a lower triangular form, as illustrated in Figure 9.5, using only permutations

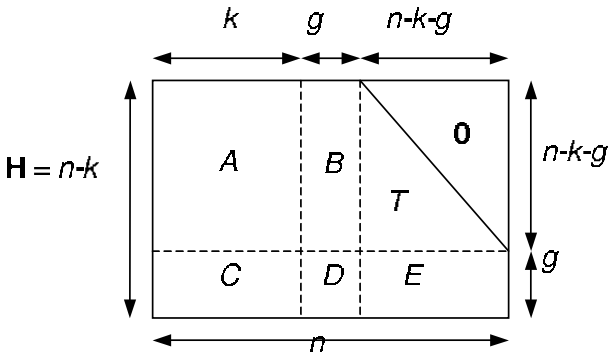

Figure 9.5 – Representation in the lower pseudo-triangular form of the parity check matrix H.

of rows or columns. This matrix is made up of 6 sparse sub-matrices, denoted A,B,C,D,E and T, the latter being a lower triangular sub-matrix. Once the preprocessing of $\mathbf{H}$ is finished, the encoding principle is based on solving the system represented by the following matrix equation:

$$\mathbf{c}\mathbf{H}^T = 0 \tag{9.16}$$

The codeword searched for is decomposed into three parts: $\mathbf{c} = (\mathbf{d}, \mathbf{r}_1, \mathbf{r}_2)$, where $\mathbf{d}$ is the systematic part that is known and where the redundancy bits searched for are split into two vectors $\mathbf{r}_1$ and $\mathbf{r}_2$, of respective size g and $n-k-g$. After multiplication on the right-hand side by matrix $\begin{pmatrix} I & 0 \\ -ET^{-1} & I \end{pmatrix}$, Equation (9.16) becomes:

$$A\mathbf{d}^t + B\mathbf{r}_\mathbf{1}^t + T\mathbf{r}_\mathbf{2}^t = 0 \tag{9.17}$$

$$\left(-ET^{-1}A + C\right)\mathbf{d}^t + \left(-ET^{-1}B + D\right)\mathbf{r}_\mathbf{1}^t = 0 \tag{9.18}$$

Equation (9.18) enables $\mathbf{r}_1$ to be found by inverting $\Phi = -ET^{-1}B + D$. Then Equation (9.17) enables $\mathbf{r}_2$ to be found by inverting T. Many time-consuming operations can be done once and for all during preprocessing. All the operations repeated during the encoding have a complexity in $\mathrm{O}(n)$ except the multiplication of $\left(-ET^{-1}A + C\right)\mathbf{d}^t$ by square matrix $\left(-\Phi^{-1}\right)$ of size $g \times g$ which after inversion is no longer sparse, hence a complexity in $\mathrm{O}\left(g^2\right)$. It is shown in [9.49] that we can obtain a value of g equal to a small fraction of n: $g = \alpha n$ where α is a sufficiently low coefficient for $\mathrm{O}\left(g^2\right) << \mathrm{O}(n)$ for values of n up to 10^5.

Specific constructions

Coding with a sparse generator matrix

One idea proposed by Oenning *et al.* [9.45] involves directly building a sparse systematic generator matrix, so the coding is performed by simple multiplication and the parity check matrix remains sparse. These codes are called Low-Density Generator-Matrix (LDGM) codes. Their performance is however poor [9.36], even if it is possible to optimize their construction [9.22] and lower the error floor.

Encoding by solving the system $cH^T = 0$ obtained by substitution

Mackay *et al.* [9.40] propose to constrain the parity matrix so that it is composed of the three sub-matrices A, B and C arranged as in Figure 9.6.

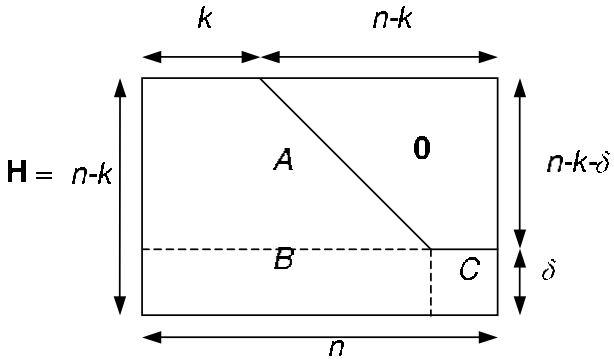

Figure 9.6 – Specific construction of parity check matrix H facilitating the encoding.

Systematic encoding is performed by solving Equation (9.16), which means solving $(n - k - \delta)$ equations by substitution. Each row of the parity matrix containing a low number of 1s, this operation is linear with n. The remaining δ equations are solved by inversion of matrix C defined in Figure 9.4. That leads to multiplication by a non-sparse matrix and therefore a complexity in $O\left(\delta^2\right)$. Bond *et al.* [9.7] and Hu *et al.* [9.29] have proposed building parity check matrices with $\delta = 0$. In [9.25] Haley *et al.* define a class of codes enabling equation 9.12 to be solved by an iterative algorithm similar to the one used for the decoding.

Cyclic coding

The classes of LDPC codes defined by finite geometry or by projective geometry [9.29, 9.46, 9.34, 9.1, 9.58] enable cyclic or pseudo-cyclic codes to be obtained (see Section **??** for further details about this type of construction).The codes thus obtained can be encoded efficiently by using shift registers. In addition, they offer good properties in terms of the distribution of cycle length ($\geqslant 6$). The main drawback is that the cardinal of these classes of code is rela-

tively small. These classes therefore offer only a very limited number of possible *size – rate – irregularity profile* combinations.

Summary

Table 9.1 summarizes the different possible types of coding encountered in the literature. In practice, the conventional encoding of block codes by a generator matrix is not used for LDPC codes due to the large the length of the codewords. The codes obtained by projective or finite geometry cannot be optimized (optimal design of the irregularity profiles). There therefore remain only codes built to facilitate the encoding by solving the equation $cH^T = 0$ by substitution, such as the one chosen for the DVB-S2 standard.

Type of encoding		Complexity	Remarks
Generic	Generator matrix	$\sim O(n^2)$	Not used in practice
	Pseudo-linear encoding	$\sim O(n)$	A lot of preprocessing
Ad hoc construction	Solving $\mathbf{cH}^T = 0$ by substitution	$\sim O(n)$ (si $\delta = 0$)	Possible loss of performance
	Cyclic or pseudo-cyclic	$\sim O(n)$	Limited number of possible combinations of the different parameters

Table 9.1 – Summary of the different possible encodings.

Analogy between an LDPC code and a turbo code

Figure 9.7 presents a turbo code in the form of a bipartite graph proposed by Tanner [9.53], thus showing the very close relation which links the family of turbo codes and that of LDPC codes.

In the case of a turbo code, the constraints are greater than in the case of an LDPC code since elementary codes are convolutional codes. But in the same way as for LDPC codes, a word is a codeword if and only if the two constraints of the graph are respected. Note here that the degree of the bits of a turbo code is two for the information bits and one for the redundancy bits.

Similarly, a product code can also be represented by a bipartite graph. The number of constraint nodes is then $2\sqrt{n}$ (compared with 2 for the turbo code and $n/2$ for an LDPC code with rate 0,5) and the latter have an intermediate complexity between that of the turbo code and that of the LDPC code. Turbo codes and LDPC codes are thus the two extremities of the spectrum of "composite" codes. The former contain only two very complex constraint nodes, the

latter, very many constraint nodes, each node being made up of the simplest linear code possible (parity code). Note that from this representation of the bipartite graph, an infinite number of more or less exotic codes can be built.

It is interesting to note that encoding LDPC codes is tending to be performed more and more like the encoding of turbo codes (in a serial concatenation). The precursors were, without doubt, the *Repeat Accumulate* codes proposed in [9.16] whose encoding is composed of a repetition code, an interleaver and an accumulator. These codes are then decoded by an algorithm of the LDPC type with an adapted schedule. In the literature, we can now find many variants of this type of encoding, which involves combining elementary encoders and interleavers.

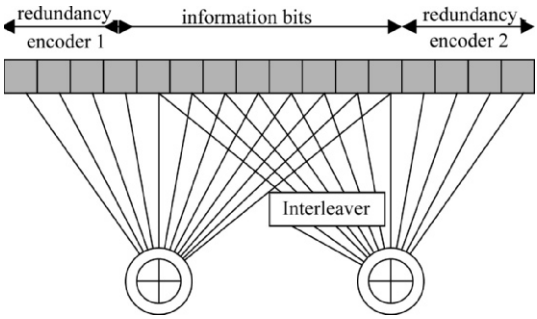

Figure 9.7 – Representation of a turbo code in the form of a bipartite graph.

The similarity between turbo codes and LDPC codes is even greater than can be assumed from the representations in the form of bipartite graphs. Indeed, it is shown in [9.36], [9.44] and in Section 6.2 that it is possible to represent a turbo code in the form of an LDPC matrix. The resemblance stops there. Parity check matrix $\mathbf{H}$ of a turbo code contains many rectangular patterns (four 1s making a rectangle in matrix $\mathbf{H}$), that is, many cycles of length 4, which make the algorithms for decoding LDPC codes, to be described below, inefficient.

9.1.4 Decoding LDPC codes

Decoding an LDPC code is done using the same principle as decoding a turbo code by an iterative algorithm called a belief propagation algorithm. Each variable node sends to the parity nodes with which it is associated a message about the estimated value of the variable (*a priori* information). The set of *a priori* messages received enables the parity constraint to compute then return the extrinsic information. The successive processing of the variable then parity nodes make up one iteration. At each iteration, there is therefore a bilateral exchange of messages between the parity nodes and variable nodes, on the arcs of the bipartite graph representing the LDPC code. At the level of the receiver, the

method for quantifying the sequence received, $\mathbf{r}$, determines the choice of decoding algorithm.

Hard input algorithm

Quantization on one bit involves processing only the sign of the samples received. Hard input decoding algorithms are based on the one proposed by Gallager under the name of algorithm A [9.21]. These decoders of course offer lower performance than those of soft input decoders. They are only implemented for very particular applications like optical communications, for example [9.17]. These algorithms will not be considered in the remainder of this chapter.

Belief propagation algorithm

When quantization is done on more than one bit, the decoding may use soft inputs: the *a priori* probability of the received symbols. In the case of binary codes and in the logarithm domain, we use the *a priori* log likelihood ratio (LLR) of samples r_j :

$$L\left(r_j \,|c_j\right) = \ln\left(\frac{p\left(r_j \,|c_j = 0\right)}{p\left(r_j \,|c_j = 1\right)}\right) \tag{9.19}$$

where c_j is the j-th bit of the codeword and $r_j = 2c_{j-1} + b_j$. In the case of the additive white Gaussian noise channel, the noise samples b_j follow a centred Gaussian law with variance σ^2, that is:

$$p\left(r_j \,|c_j\right) \quad = \quad \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[\frac{\left(r_j - (2c_j - 1)\right)^2}{2\sigma^2}\right] \tag{9.20}$$

Combining (9.19) and (9.20), intrinsic information I_j can be defined :

$$I_j \triangleq L\left(r_j \,|c_j\right) = -\frac{2r_j}{\sigma^2} \tag{9.21}$$

Each iteration of the BP algorithm is decomposed into two steps:

1. Processing the parities:

$$Z_{j,p} = 2\tanh^{-1}\left[\prod_{j' \in J(p)/j} \tanh\frac{L_{j',p'}}{2}\right] \tag{9.22}$$

2. Processing the variables:

$$L_{j,p} = I_j + \sum_{p' \in P(j)/p} Z_{j',p'} \tag{9.23}$$

The iterations are repeated until the maximum number of iterations N_{it} is reached. It is possible to stop the iterations before N_{it} when all the parity equations are satisfied. This enables either a gain in mean throughput, or a limit in consumption.

We call L_j the total information or the LLR of bit j. This is the sum of the intrinsic information I_i and the total extrinsic information Z_j which is by definition the sum of the extrinsic information of branches $Z_{j,p}$:

$$Z_j \triangleq \sum_{p \in P(j)} Z_{j,p} \tag{9.24}$$

We therefore have $L_j = I_j + Z_j$ and Equation (9.23) can then be written:

$$L_{j,p} = L_j - Z_{j,p} = I_j + Z_j - Z_{j,p} \tag{9.25}$$

The BP algorithm is optimal in the case where the graph of the code does not contain any cycle: all the schedules[1] give the same result. As LDPC codes involve cycles, their decoding by the BP algorithm can lead to phenomena of self-confirmation of the messages which degrade the convergence and make the BP algorithm distinctly sub-optimal. However, these phenomena can be limited if the cycles are large enough.

The first schedule proposed is called the "flooding schedule" [9.35]. It involves successively processing all the parities then all the variables.

Flooding schedule algorithm

Initialization :

1- $n_{it} = 0$, $Z_{j,p}^{(0)} = 0 \quad \forall p \quad \forall j \in J(p)$, $I_j = 2y_j/\sigma^2 \quad \forall j$

Repeat until $n_{it} = N_{it}$ or until the system has converged towards a codeword :

2- $n_{it} = n_{it} + 1$

3- $\forall j \in \{1, \cdots, n\}$ do: {Computation of the variable towards parity messages}

4- $Z_j^{(n_{it})} = \sum_{p \in P(j)} Z_{j,p}^{(n_{it}-1)}$ and $L_j^{(n_{it})} = I_j + Z_j^{(n_{it})}$

5- $\forall p \in P(j)$:

$$L_{j,p}^{(n_{it})} = I_j + \sum_{p' \in P(j)/p} Z_{j,p'}^{(n_{it}-1)} = I_j + Z_j^{(n_{it})} - Z_{j,p}^{(n_{it}-1)}$$

[1] By schedules, we mean the order in which each parity and each variable is processed.

6- $\forall p \in \{1, \cdots, m\}$ faire : {Computation of the parity towards variable message}

7- $\forall j \in J(p)$: $Z_{j,p}^{(n_{it})} = \underset{j' \in J(p)/j}{\oplus} L_{j,p'}^{(n_{it})}$

The bits decoded are then estimated by $\text{sgn}\left(-L_j^{(n_{it})}\right)$.
It is interesting to note that it is possible to modify the algorithm by "ordering" the flooding schedule depending on the parity nodes. The latter are then processed serially, and the algorithm becomes:

3'- $\forall j \in \{1, \cdots, n\}$: $Z_j^{(n_{it}+1)} = 0$

4'- $\forall p \in \{1, \cdots, m\}$ do:

5'- Computation of the input messages

$$\forall j \in J(p) \quad L_{j,p}^{(n_{it})} = I_j + Z_j^{(n_{it})} - Z_{j,p}^{(n_{it}-1)}$$

6'- Computation of the extrinsic information

$$\forall\, j \in J(p) \quad Z_{j,p}^{(n_{it})} = \bigoplus_{j' \in J(p)/j} L_{j,p'}^{(n_{it})}$$

7'- Update for the following iteration

$$\forall\, j \in J(p) \quad Z_j^{(n_{it}+1)} = Z_j^{(n_{it}+1)} + Z_{j,p}^{(n_{it})}$$

A similar organization of computations for the variable nodes will be called "distributed computation" since the computations linked with a variable node will be distributed during one iteration. In Section 9.2, the different types of schedule will be detailed then generalized.

It must also be noted that the notion of iteration (the computation of all the messages of the graph once and only once) is not strict. Thus, Mao *et al.* [9.42] proposed a variant of the flooding schedule in order to limit the impact of the effect of the cycles on the convergence. This variant, called "probabilistic scheduling", involves not processing some variables at each iteration. The choice of these variables is random and depends on the size of the smallest cycle associated with this variable: the smaller the cycle, the lower the probability of processing the variables involved in this cycle. This method limits phenomena of self-confirmation introduced by short cycles. It enables convergence to be obtained more rapidly than with the flooding schedule. The architectures linked with this schedule will not be discussed in this chapter.

9.1.5 Random construction of LDPC codes

The construction of an LDPC code (or of a family of LDPC codes) must naturally be done so as to optimize the performance of the code while minimizing the hardware complexity of the associated decoder.

Building a code remains a delicate problem so we refer the reader wishing to explore the subject further to the references given in this chapter. The problem of building an LDPC code adapted to decoding hardware will be dealt with in Section 9.2.

Optimizing an LDPC code is carried out in three steps:

- *a priori* optimization of the irregularity profiles of the parity and variable nodes;
- construction of matrices $\mathbf{H}$ of an adequate size respecting the irregularity profile and maximizing the length of the cycles;
- if necessary, selection or rejection of the codes using the minimum distance criterion or the performance computed by simulation.

Optimization of irregularity profiles

We make the hypothesis of codes with infinite size and an infinite number of iterations. Indeed, this enables optimization of their asymptotic characteristics (irregularity profile, rate) as a function of the channel targeted. Two techniques exist: the density evolution algorithm and its Gaussian approximation, and the extrinsic information transfer chart.

The *density evolution* algorithm was proposed by Richardson [9.48]. This algorithm calculates the probability density of messages $L_{j,p}$ and $Z_{j,p}$ after each new iteration. The algorithm is initialized with the probability density of the input samples, which depends on the level of noise σ^2 of the channel. Using this algorithm enables us to know the maximum value of σ^2 below which the algorithm converges, that is, such that the error probability is lower than a given threshold. It is also possible to determine by linear programming an irregularity profile that gives the lowest possible threshold.

A simplification of the density evolution algorithm proposed by Chung *et al.* [9.14, 9.13], is obtained by approximating the real probability densities by Gaussian densities. The interest of Gaussian density approximation is that it suffices to calculate the evolution of a single parameter by making the hypothesis that these Gaussian densities are consistent, that is, the variance is equal to two times the mean. Indeed, assuming that the "all zero"word was sent, at initialization we have ($n_{it} = 0$):

$$L_{j,p}^{(0)} = -\frac{2r_j}{\sigma^2} \quad \text{with} \quad r_j \sim N\left(-1, \sigma^2\right) \tag{9.26}$$

$$\text{therefore } L_{j,p}^{(0)} \sim N\left(\frac{2}{\sigma^2}, \frac{4}{\sigma^2}\right)$$

We denote:

$m_j^{(0)} = \frac{2}{\sigma^2}$ the average of the consistent Gaussian probability density of variable c_j of degree d_v sent to parities e_p of degree d_c which are connected to it,

$\mu_p^{(n_{it})}$ the mean of messages $Z_{j,p}^{(n_{it})}$.

To follow the evolution of the average $m_j^{(n_{it})}$ during the iterations n_{it} , it then suffices to take the mathematical expectation of Equations (9.22) and (9.23) relative to the variable and parity processing , which gives:

$$\Psi\left(\mu_p^{(n_{it})}\right) = \Psi\left(m_j^{(n_{it})}\right)^{d_c-1} \text{ with } \Psi(x) = E\left[\tanh(x/2)\right],\ x \sim N(m, 2m) \quad (9.27)$$

$$m_j^{(n_{it}+1)} = \frac{2}{\sigma^2} + (d_v - 1)\,\mu_p^{(n_{it})} \quad (9.28)$$

Thus for a regular (d_v, d_c) LDPC code and for a given noise with variance σ^2, Equations (9.27) and (9.28) enable us, by an iterative computation, to know if the mean of the messages tends towards infinity or not. If such is the case, it is possible to decode without errors with a codeword of infinite size and an infinite number of iterations. In the case of an irregular code, it suffices to make the weighted mean on the different degrees of Equations (9.27) and (9.28).

The maximum value of σ for which the mean tends towards infinity, and therefore for which the error probability tends towards 0, is the threshold of the code. For example, the threshold of a regular code (3,6), obtained with the density evolution algorithm, is $\sigma_{\max} = 0.8809$ [9.13], which corresponds to a minimum signal to noise ratio of $\frac{E_b}{N_0}_{\min} = 1.1$dB.

Another technique derived from extrinsic information transfer (EXIT) charts[2] proposed by ten Brink [9.55, 9.56] enables the irregularity profiles to be optimized. Whereas the density evolution algorithm is interested in the evolution of the probability densities of the messages during the iterations, these charts are interested in the transfer of mutual information between the input and the output of the decoders of the constituent codes [9.56]. The principle of these charts has also been used with parameters other than mutual information, like the signal to noise ratio or error probability [9.3, 9.2]. It has also been applied to other types of channels [9.15].

[2] The principle of building EXIT charts is described in Section 7.6.3

Optimization of cycle size

Optimization of the irregularity profiles being asymptotic, we now have to build a parity check matrix of finite size. This phase can be performed randomly: we draw the non zero inputs of the parity check matrix at random, respecting as far as possible the irregularity profile of the nodes. It is also possible to build codes by randomly drawing permutations of an elementary matrix which are then concatenated. Another way to build LDPC codes is the deterministic construction of a matrix (finite and projective geometry).

In all cases, we must pay attention to the cycles present in the graph of the code. The belief propagation decoding algorithm assumes that the cycles that would deteriorate the independence of the messages entering a node do not exist. In practice, the presence of cycles in the graph is inevitable, but if they are large enough, the independence of the messages remains a good approximation. Building good LDPC codes must therefore ensure the absence of smaller cycles, those of size 4. Very many solutions are proposed in the literature to build LDPC codes. For example, Campello *et al.* [9.9] propose optimizing the size of the minimum cycle for a given rate. Hu *et al.* [9.65] suggest building the graph branch by branch in order to avoid at maximum the lowest cycle sizes (*Progressive Edge Geometry* or PEG). Zhang *et al.* [9.67] build LDPC codes whose smallest cycles are size 12, 16 or 20, but the variables are only degree 2. Tian *et al.* [9.57] use the fact that not all the small size cycles have the same influence and omit only those that are the most penalizing.

Selecting the code by the impulse method

The decoding performance using the belief propagation algorithm is improved by avoiding small sized cycles. But it is also important to have "good" error correcting codes, that is to say, those that have a large minimum distance. The impulse method was first proposed by Berrou *et al.* [9.4, 9.5] to evaluate the minimum Hamming distance of a turbo code. It was then adapted to the case of LDPC codes by Hu *et al.* [9.26]. It thus enables us to simply verify that the minimum distance of the code designed is sufficient to reach the error rate targeted for the application required.

Selecting the code by simulation

Two codes of the same size and the same rate, built with the same irregularity profiles, not having a short cycle and having the same minimum distance can nevertheless have a fairly different performance. These differences can be explained by two phenomena: the existence of "parasitic" fixed points introduced by the sub-optimality of the iterative decoding algorithm which increase the binary error rate in relation to the theoretical value [9.33]. The number of codewords with minimum distance also influences the performance of the code.

Figure 9.8 shows the performance of an LDPC code for different sizes and different rates in the case of a DVB-S2 decoder implemented on an Altera Stratix80 FPGA.

LDPC codes therefore have an excellent theoretical performance. This must however be translated by simplicity in their hardware implementation to enable these codes to be used in practice. That is why particular attention must be paid to LDPC decoder architectures and implementations.

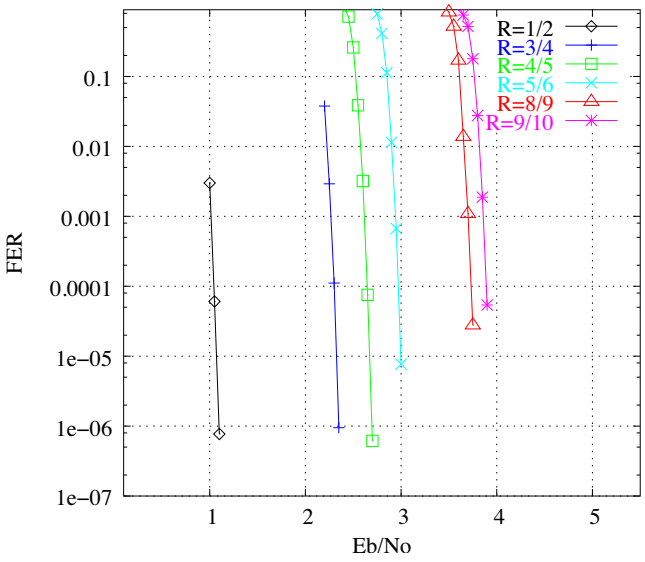

Figure 9.8 – Packet error rate (or Frame error rate, FER) obtained for codeword sizes of 64 kbits and different rates of the DVB-S2 standard (50 iterations, fixed point). With the permission of TurboConcept S.A.S, France.

9.1.6 Some geometrical constructions of LDPC codes

To complete the random constructions presented above, we list below some deterministic constructions leaving much less room, if any, for random ones.

Cayley / Ramanujan constructions

Margulis [9.43] was the first to propose algebraic LDPC codes. Then Rosenthal and Votonbel [9.50, 9.51] extended these results to obtain high expansion factor graphs with high girths, using Ramanujan graphs instead of Cayley graphs. Some drawbacks were raised by MacKay and Postol [9.38] about these codes (error floor and low minimum distance for some sets of parameters).

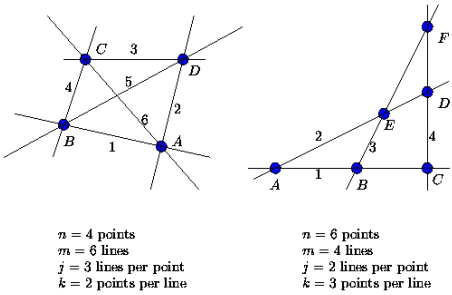

Figure 9.9 – Simple illustration of finite geometry

Kou, Lin and Fossorier's Euclidian / Projective Geometry LDPC

These LDPC codes are built on finite geometry [9.34]. Finite geometry is based on n points and m rows, such that each row contains k points and each point is on j lines. Two lines are either parallel or they have only one common point. A parity check matrix $\mathbf{H} = (h_{ij})$ can be built by assuming that $h_{ij} = 1$ if and only if the i-th row contains the j-th point. Of course, j and k have to be small compared to m and n in order to build an LDPC code that is called *type I*. An example of simple finite geometry is illustrated in Figure 9.9.

These codes are cyclic. When considering the transposed version of $\mathbf{H}$, we can obtain quasi-cyclic LDPC codes which are referred to as *type II* codes. Two kinds of finite geometry are used: Euclidean geometry (EG) and projective geometry (PG).

Constructions based on permutation matrices

Tanner *et al.* have proposed an algebraic construction of LDPC parity-check matrices based on an idea of Tanner [9.54]. A regular (j, k) code of length $n = kp$ and $m = jp$ parity checks can be obtained, assuming that p is a prime number and j, k are chosen among the prime factors of $p - 1$. The structure of the parity check matrix is:

$$H = \begin{pmatrix} I_1 & I_a & I_{a^2} & \cdots & I_{a^{k-1}} \\ I_b & I_{ab} & I_{a^2b} & \cdots & I_{a^{k-1}b} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ I_{b^{j-1}} & I_{ab^{j-1}} & I_{a^2b^{j-1}} & \cdots & I_{a^{k-1}b^{j-1}} \end{pmatrix}$$

where I_x is the identity matrix whose columns have been right-shifted x times, and where a and b are two non-zero elements in $\mathbf{F}_p$ of order j and k respectively.

Matrices based on Pseudo random generators

An important drawback of random constructions is that the parity check matrix has to be saved in memory, which takes up a lot of room for long codes. Prabhakar and Narayanan [9.47] found an interesting solution to circumvent this issue by using linear congruential sequences to design the parity-check matrix. Hence, after a non-complex computation, the generator outputs the address of the non-zero entries of the matrix. This solution has been implemented by Verdier *et al.* [9.62]. Girths of at least 6 can be obtained by correctly choosing the parameters of the generator.

Array-based LDPC

Array codes are two-dimensional codes that have been proposed for detecting and correcting burst errors [9.6]. When viewed as binary codes, the parity check matrix of array codes exhibit sparseness, which can be exploited for decoding them as LDPC codes using the BP algorithm [9.19]. Therefore, array codes provide the framework for defining a family of LDPC codes that lend themselves to deterministic constructions [9.18]. The parity check matrix of an array-based LDPC code is:

$$H = \begin{pmatrix} I & I & I & \cdots & I \\ I & \alpha & \alpha^2 & \cdots & \alpha^{k-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ I & \alpha^{j-1} & \alpha^{2(j-1)} & \cdots & \alpha^{(j-1)(k-1)} \end{pmatrix}$$

where I is the $p \times p$ identity matrix, p being an odd prime number, α is the I matrix whose rows have been shifted once, and $j, k \geq p$.

BIBDs, Latin rectangles

A block design is an incidence system [9.63] (v, k, λ, r, b) in which a set X of v points is partitioned into a family A of b subsets (blocks) in such a way that any two points determine λ blocks with k points in each block, and each point is contained in r different blocks. It is also generally required that $k < v$, which leads to a balanced incomplete block design (BIBD) of the LDPC code. The five parameters are not independent, but satisfy the two relations

$$\begin{aligned} vr &= bk \\ \lambda(v-1) &= r(k-1) \end{aligned}$$

A BIBD is therefore commonly simply written as (v, k, λ), since b and r are given in terms of v, k, and λ by

$$b = \frac{v(v-1)\lambda}{k(k-1)} \qquad r = \frac{\lambda(v-1)}{k-1}$$

A BIBD is said to be symmetric if $b = v$ (or, equivalently, $r = k$).

These constructions have been widely studied in the literature. For example, MacKay and Davey [9.37] first proposed the use of Steiner triple systems to design short LDPC codes. Johnson and Weller [9.30, 9.31], and also B. Vasic [9.60], presented a family of LDPC codes based on Kirkman triple systems. A design based on anti-Pasch Steiner systems is also presented in [9.59], and mutually orthogonal Latin rectangles (MOLR) are used in [9.61].

9.2 Architecture for decoding LDPC codes for the Gaussian channel

When the belief propagation algorithm is implemented, the general architecture of decoders for LDPC codes can be performed with the help of generic node processors (GNP) modelling either the parity processing, or the variable processing. This section describes the different possible implementations of these processors after analysing the decoding complexity of LDPC codes. The different possibilities for controlling this GNP-based architecture enables us to define three classes of schedule for the belief propagation algorithm: a two-pass schedule, a "vertical" schedule and a "horizontal" schedule. This original, unified presentation of the architectures of decoders for LDPC codes enables us to cover many existing architectures published so far, and to synthesize innovatory architectures.

9.2.1 Analysis of the complexity

The decoding complexity of LDPC codes is directly linked with the number of branches in the bipartite graph of the code, or with the number of 1s in the parity check matrix. The iterative decoding belief propagation algorithm has two steps. At each step, we have to calculate information $L_{j,p}$ or $Z_{j,p}$ which is associated with the branch linking variable j to parity p. Let us denote B the number of branch in the bipartite graph of the LDPC code. For example, in the case of a regular code (d_v, d_c) of size n, the number of branches B is given by:

$$B = d_v n = d_c m \tag{9.29}$$

The computing power P_c necessary to decode LDPC codes is then defined as the number of branches to process per clock cycle. This parameter depends on:

- the number k of information bits to transmit per codeword,
- the number of branches B,
- the data rate D of information desired,
- the maximum number of iterations N_{it},

- the clock frequency f_{clk}.

In one second, the number of codewords to process in order to obtain an information data rate D is equal to D/k (words/second). In the worst case, decoding a codeword requires computing $B \times N_{it}$ branches. To guarantee a data rate D, an architecture must provide the power to compute $D \times B \times N_{it}/k$ branches per second. The minimum computing power P_c to provide per clock cycle is therefore:

$$P_c = \frac{BN_{it}D}{kf_{clk}} \text{ (branches/cycle)} \tag{9.30}$$

Note that, for a fully parallel architecture in which each node of the graph is associated with a processor, all the branches of the graph are processed in one clock cycle. The computing power is then $(P_c)_{\max} = B$. There is no practical interest in trying to go beyond this power since the critical path then becomes longer.

9.2.2 Architecture of a generic node processor (GNP)

The computations performed in a variable node processor (VNP) and in a parity node processor (PNP) have an identical dependency between the inputs and the outputs. Indeed, for both the PNP and VNP, the d outputs are calculated from the d inputs, with the i-th output depending on all the inputs less the i-th input. It is thus possible to represent the different processor architectures abstractly by a generic node processor. The latter will then be specialized according to the decoding algorithm used. The GNP therefore receives at its input d messages $(e_i)_{i=1..d}$ and produces at its output d messages $(s_j)_{j=1..d}$ defined by:

$$s_j = \bigotimes_{i \neq j} e_i \tag{9.31}$$

Operator $\otimes$ is a generic associative-commutative operator for computations, whose implementation will be specified below. The condensed expression 9.31 means that the operator is applied to all the variables e_i for $i \neq j$.
Figure 9.10 illustrates the three main versions of GNP parallel architectures:

- *Direct architecture*: the computations of the d output messages are performed independently (Figure 9.10(a)). The computations of the different outputs can also be factorized. The number of $\otimes$ components traversed is of the order of $\log_2(d)$.

- *Trellis architecture (Forward-Backward type)*: This architecture corresponds to a particular factorized form of parallel architecture which has great regularity, but whose number of $\otimes$ operators is linear with d.

- *Total sum architecture*: this architecture is possible only if the generic operator $\otimes$ allows an inverse denoted $inv_{\otimes}$. In this case, the generic operator

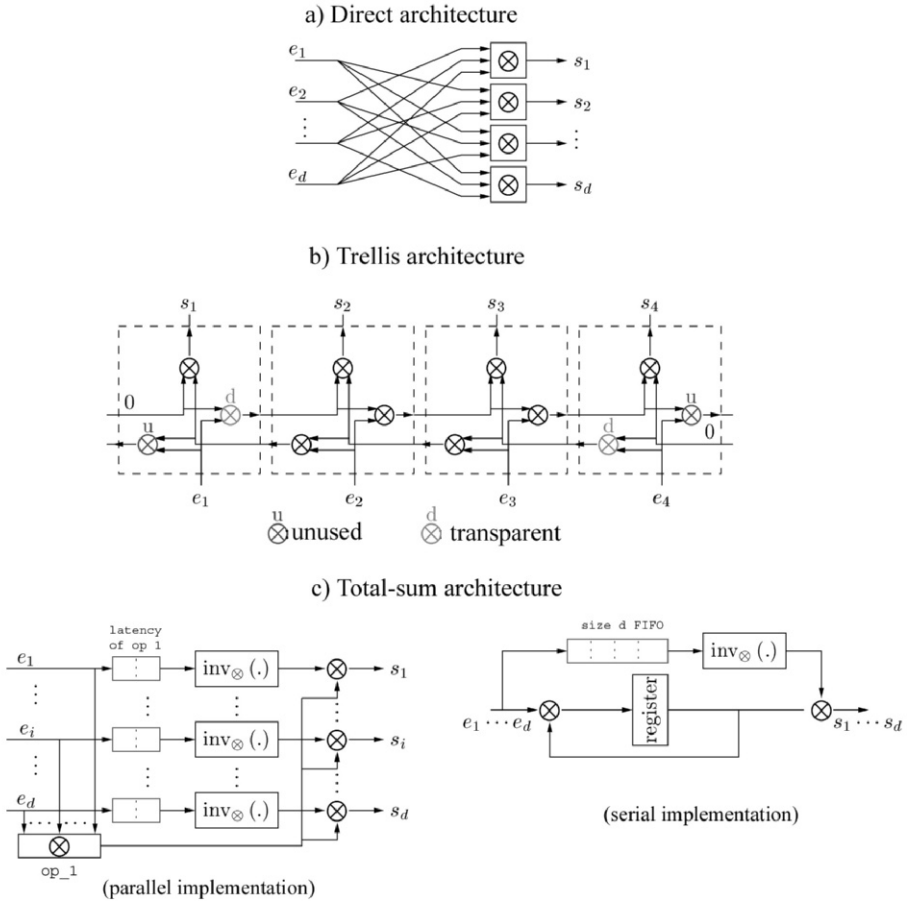

Figure 9.10 – The different "compact mode" architectures for implementing the generic operator $\otimes$.

is applied at all the inputs (total sum) then each output is calculated by eliminating the contribution of the corresponding input, with the help of the inverse operator.

It is possible to modify these architectures in order to introduce intermediate *pipeline* registers enabling the critical path to be reduced. There are also architectures of the serial type (Figure 9.10(c)).

In what follows, the degree of parallelism of a GNP will be denoted α_g. This is the number of cycles necessary to process a node (without considering latency due to the *pipeline* processing). Thus, for a parallel architecture capable of

processing one node at each clock cycle, $\alpha_g = 1$, whereas for a serial architecture, $\alpha_g = d$.

Note that in all the GNP architectures presented, we implicitly made the hypothesis that all the inputs were available and that all the outputs had to be generated either simultaneously (parallel architecture), or grouped in time (serial architecture). This kind of GNP control mode is called the "compact mode".

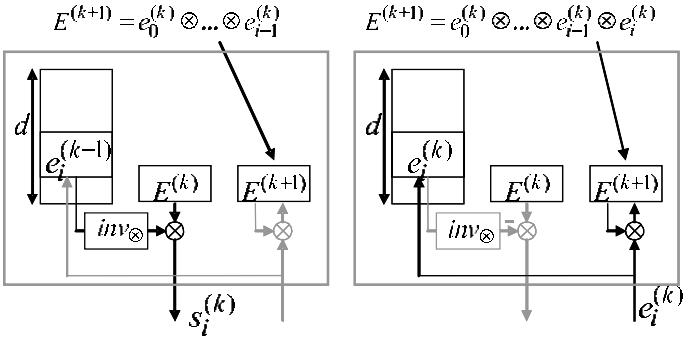

Figure 9.11 – Principle of the distributed mode (delayed update).

It is possible to imagine different execution modes, like the "distributed mode", in which the GNP inputs and outputs are distributed throughout the decoding iteration.

Figure 9.11 shows the distributed mode operation of a processor :

- During the current iteration n_{it}, we consider that the input variables e_i come from the previous iteration $n_{it} - 1$ whereas the output variables belong to the current iteration.
- At the end of an iteration, we assume that the d input variables $(e_i^{(n_{it}-1)})_{i=1..d}$ are memorized in a memory (internal or external to the GNP) as well as the value of $E^{(n_{it})} = \underset{i=1..d}{\otimes} e_i^{(n_{it}-1)}$.
- The GNP can therefore, at the request of the system, calculate the i-th output $s_i^{(n_{it})} = E^{(n_{it})} \otimes inv_\otimes(e_i^{(n_{it}-1)})$.
- This output is sent, via the interleaver, to the opposite node which, once the computation is over, returns $e_i^{(n_{it})}$.
- This new value then replaces $e_i^{(n_{it}-1)}$ in the memory and is also accumulated to obtain the value of $E^{(n_{it}+1)}$ at the end of the iteration. Two accumulation modes are possible:

1. The first mode – delayed update (Figure 9.11) – involves using an accumulation register initialized to zero at each new iteration. This register enables $E^{(n_{it}+1)} = \underset{i=1..d}{\otimes} e_i^{(n_{it})}$ to be calculated directly. This architecture therefore has $d+2$ memory words, d for the inputs $(e_i^{(n_{it}-1)})_{i=1..d}$, a word for $E^{(n_{it})}$ and a word for the accumulation of $E^{(n_{it}+1)}$.
2. The second mode – immediate update – involves replacing the contribution of $e_i^{(n_{it}-1)}$ in $E^{(n_{it})}$ by that of $e_i^{(n_{it})}$ as soon as a new input $e_i^{(n_{it})}$ arrives, that is:

$$E^{(n_{it})} = E^{(n_{it})} \otimes e_i^{(n_{it})} \otimes inv_{\otimes}\left(e_i^{(n_{it}-1)}\right) \tag{9.32}$$

   At the end of the iteration, we thus have $E^{(n_{it}+1)} = E^{(n_{it})}$. This solution offers two advantages in relation to delayed updating:
   - one less memory word;
   - an acceleration in the convergence of the algorithm as the new values of the inputs are taken into account sooner.

Choice of a generic operator

Figure 9.12 gives a "cross-section" view of the belief propagation algorithm on the bipartite graph of the LDPC code. We assume that each branch is split into two to differentiate the variable towards parity messages from the parity towards variable messages. This view shows the great resemblance between processing variables and processing the parities and enables us to imagine other positions of the interconnection networks in the computation cycle. Each position of the interconnection graph is thus translated by a different processing of the parity nodes and the variable nodes. Table 9.2 gives the different computations to be carried out according to the position of the interconnection network.

When the interconnection network is in position 1 (Table 9.2), we again have the classical separation between a variable processor and a parity processor. The latter can then either be performed in the frequency domain (as indicated in Figure 9.11), or directly in the domain of the LLRs via the $\oplus$ operator defined in equation (9.3).

9.2.3 Generic architecture for message propagation

Presentation of the model

PNPs and GNPs are characterized by their architecture and their generic operator, depending on the position of the interconnection network. The architecture presented in Figure 9.13 enables the exchange of messages between these different processors.

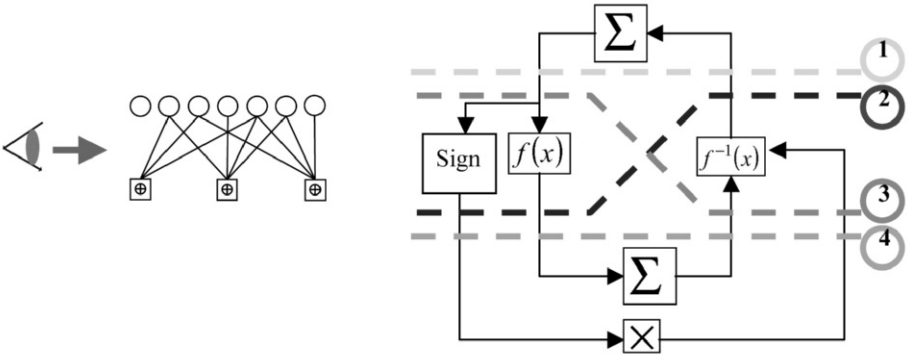

Figure 9.12 – Different positions of the interconnection network obtained by processing the parity nodes in the Fourier domain. These positions separate those parts of the iteration to be performed in the VNP from those performed in the PNP.

Position of the network		1	2	3	4
VNP		Σ	$f \circ \Sigma$	$\Sigma \circ f^{-1}$	$f \circ \Sigma \circ f^{-1}$
PNP module	Fourier	$f^{-1} \circ \Sigma \circ f$	$f^{-1} \circ \Sigma$	$\Sigma \circ f$	Σ
	Direct	$\bigoplus$			
PNP sign		Product of the signs			

Table 9.2 – Value of the generic operator associated with the variable processors (VNPs) and parity processors (PNPs) as a function of the position of the interconnection network.

This architecture is composed of P PNPs which each generate $d = d_p$ messages in α_p clock cycles. These processors therefore have $d'_c = d_c/\alpha_p$ inputs and outputs. They are connected to an interleaving network that is direct and inverse. On the other side of this interleaving network are placed $\left(Pd'_p/d'_v\right)$ VNPs. These processors similarly generate $d = d_v$ messages in α_v clock cycles, and therefore have $d'_v = d_v/\alpha_v$ inputs and outputs.

The degree of parallelism of the architecture is defined by the three parameters P, α_p and α_v. It is possible to obtain all the degrees of parallelism possible, ranging from the completely parallel architecture where $P = m$, $\alpha_p = 1$ and $\alpha_v = 1$ to the completely serial architecture where $P = 1$, $\alpha_c = d_c$ and $\alpha_v = d_v$. Note that such an architecture has a computing power (equation (9.30)) $P_c = P \times d'_c$.

Direct inverse interleaving networks enable the messages associated with the different VNPs to be routed towards different PNPs and vice-versa. This kind of network generally makes it possible to perform several permutations, called space permutations. Another type of permutation, called a time permutation,

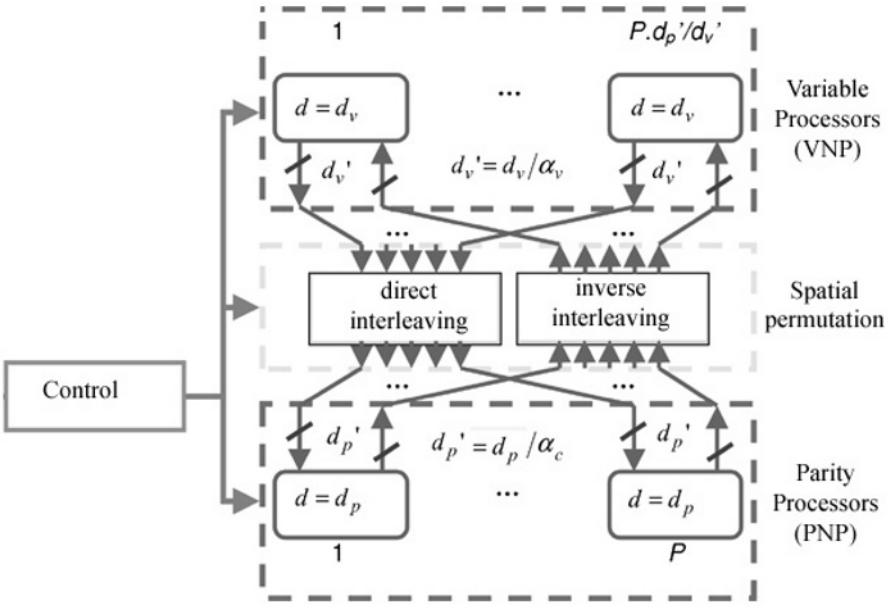

Figure 9.13 – Generic serial-parallel architecture.

makes it possible to randomly access the nodes associated with a same processor of nodes, by memory addressing, for example. The combination of these two types of permutations enables a random interconnection such as that existing between the variable nodes and the parity nodes of the LDPC code.

Example of an implementation

To help clarify ideas about the way to organize the computations and the propagation of the messages in the decoder, and to truly understand the link between the organization of the propagation of the messages and the structure of the LDPC code, Figure 9.14 shows a simple example of decoding an LDPC code of length $n = 12$ and rate $R = 0,5$ (therefore $m = 6$), with $P = 2$, $d_c = 3$, $\alpha = 1$ and $\beta = d_v$. There are therefore $P = 2$ parity node processors and $n/P = 6$ variable node processors. One iteration is performed in $m/P = 3$ steps:

- At the first cycle, reading the information relative to the bits is done in each of the VNPs, each of them containing $n/P = 2$ bits of the codeword (in practice, n/P can be much higher). These bits are shaded in grey in each VNP: this is space permutation.

- This information is then sent to the PNPs via the permutation network, whose address was generated from the cycle number (read into a memory for example): this is time permutation.
- Combining the two describes the random interleaving between the variables and the first two parities of the bipartite graph shown on the right-hand part of the figure.

In a single cycle, the first two parities will therefore be able to be processed. The following two will be processed at the second cycle and so on and so forth, until all the parities of the code have been processed. Note that this technique where the PNP information arrives simultaneously prevents two bits contained in the same VNP being involved in the same parity. Thus, for example, bits 1 and 2 cannot be involved in the same parity otherwise that would lead to a memory conflict. This solution therefore imposes constraints on matrix **H**, if we want it to be decodable by this structure. One solution to relax the constraints involves, for example, entering the data serially into the parities.

9.2.4 Combining parameters of the architecture

A certain number of parameters characterizing LDPC decoder architectures have been defined above:

- Node processors:
  - 3 possible architectures (direct, trellis, total sum)
  - 4 possible positions of the interconnection network (see Figure 9.12)
  - 3 input-output control modes (compact, distributed with delayed or immediate update)
- Message propagation architecture:
  - 3 parameters characterizing the level of parallelism *(P, α, β)*

All the combinations of these different parameters are possible to describe or create an LDPC decoder architecture. Of course, some of these combinations are more or less of interest, depending on the specifications required. For example, the combinations of the control modes between VNPs and PNPs, showing the different possible decoding schedules, are given in Table 9.3.

In the case where the controls on the two processors are of the compact flow of inputs-outputs type, the schedule performed is of the flooding type: all the PNPs are processed then all the VNPs. This schedule can easily be used with completely parallel ($P = m$) architectures. For mixed ($P < m$) architectures, the parities cannot all be processed completely before processing the variables. The control of the VNPs in distributed mode with delayed update allows the processing to be done since it guarantees that the new outputs will

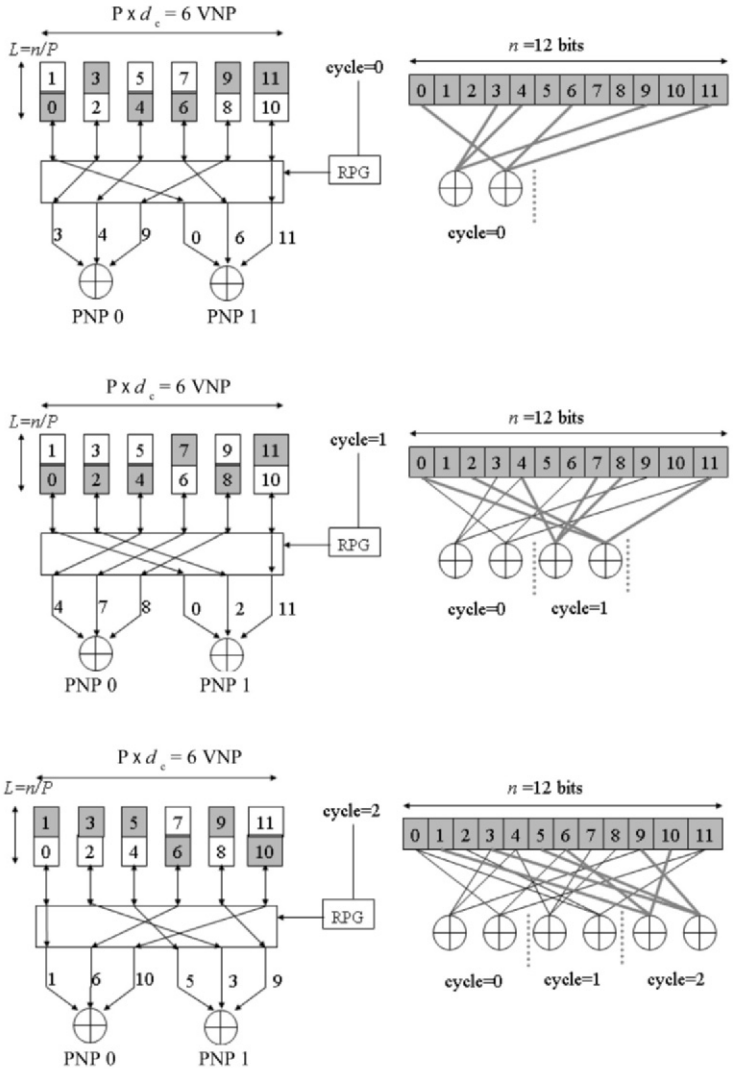

Figure 9.14 – Example of a message propagation architecture: link between the decoder's addressing and code structure.

only be computed when all the parities have been processed. This control mode implements a flooding schedule according to the parities. Symmetrically, we make a flooding schedule appear according to the variables, when the PNPs are in distributed mode and the VNPs are in compact mode. These three types of

			VNP		
			Compact	distributed	
				Delayed update	Immediate update
PNP	Compact		Flooding	Flooding (parity)	Interleaving (horizontal)
	distributed	Delayed update	Flooding (variable)	Branches	
		Immediate update	interleaving (vertical)		

Table 9.3 – Schedules associated with the different combinations of the node processor controls.

schedules converge towards the same values: They do not change the information propagation operation.

When one of the two types of processor is controlled in compact mode and the other in distributed mode with immediate update, we implement a schedule of the horizontal or vertical interleaving (*shuffle*) type. The order in which the processors are activated is similar to the flooding schedule according to the variables or the parities. Only the update of information changes since it is performed as soon as a new input has arrived, thus accelerating the convergence of the code.

The case where the two VNP and PNP processors are controlled in distributed mode is not of great interest. It would in fact correspond to controlling the decoding, branch by branch.

The memory required to implement these different combinations is given in Table 9.4.

			VNP		
			Compact	Distributed	
				Delayed update	Immediate update
PNP	Compact		$B+n$	$3n+g(B,d_c)$	$2n+g(B,d_c)$
	Distributed	Delayed update	$B+n+2m$	$3n+2m+B$	$2n+2m+B$
		Immediate update	$B+n+m$	$3n+m+B$	$2n+m+B$

Table 9.4 – Quantity of memory necessary as a function of the combinations of the different node processor controls.

Parameter B designates, like above, the number of branches in the graph. Each extrinsic branch information must be memorized, whatever the schedule used. In all cases the intrinsic variable information must also be memorized, that

is, n values. When the parity check mode is the compact one, the accumulation of the messages of the n variables in each VNP must be memorized, that is, n memories if we update them immediately, and $2n$ in the opposite case. The reasoning is the same if the VNPs are in compact mode and the PNPs are in distributed mode, but in this case, it is the accumulations of messages in the m parities that must be memorized.

It is sometimes possible, as we shall see later, to memorize the $Z_{j,p}$ messages in a compressed way. The number of messages to memorize then passes from B to $g(B, d_c)$, with g representing a compression function ($g(B, d_c) < B$).

9.2.5 Example of synthesis of an LDPC decoder architecture

The two examples described in this part allow us to show two LDPC decoder architectures using two different schedules. For each of these examples, we will give the values of the parameters characterizing these architectures.

Parameters		Values
Message propagation architecture		($\alpha_p = 1$, $\alpha_v = d_v = 3$, $P = 3$)
Position of the interconnection network		1
VNP	Control	Distributed, delayed update
	Data path	Total sum, serial
PNP	Control	Compact
	Data path	Trellis, parallel

Table 9.5 – Parameters characterizing the flooding schedule architecture.

Flooding schedule (according to parities)

The architecture described here to illustrate the flooding schedule is based on the one proposed initially by Boutillon *et al.* [9.8]. It is schematized in Figure 9.15. In this example, $P = 3$ PNP operate simultaneously in compact mode. As $\alpha_p = 1$ and $\alpha_v = 3$, there are 12 VNPs that operate simultaneously but in distributed mode (only one VNP and one PNP are shown in the figure). Note that the computing power of such an architecture is 12 branches per cycle.

The architecture of the PNPs is of the trellis type, with parallel implementation. The chronogram at the bottom of Figure 9.15 indicates that at time T_1, $d_v = 4$ messages $L_{j,p}(T_1)$ are produced at each PNP. After a latency of $T_2 - T_1$ clock cycles, the $Z_{j,p}(T_2)$ messages leaving are sent to the VNPs. This operation is reproduced m/P times to carry out one complete iteration. The data path of the VNPs is of the total sum type, with a serial implementation. The delayed update is shown by using two memory blocks, one for the extrinsic information during accumulation (Lacc), and another for the total extrinsic information of

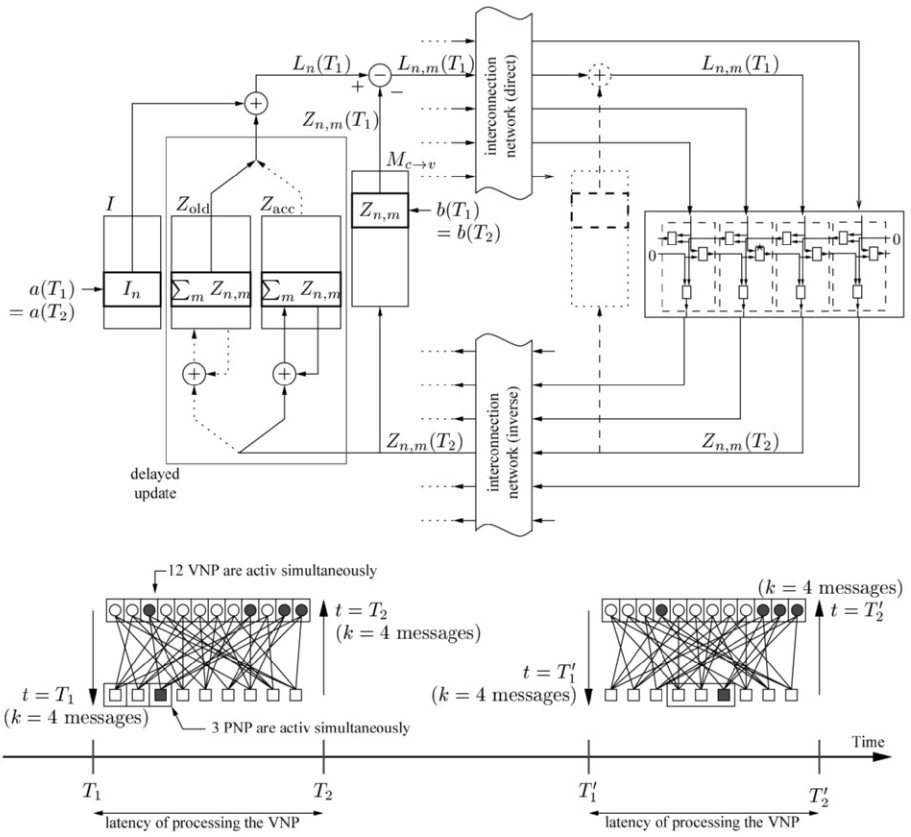

Figure 9.15 – Example of architecture for a flooding schedule (according to parities).

the previous iteration (Lold). At the end of each iteration, the role of these two memories is exchanged. In this architecture, the extrinsic branch information $Z_{j,p}$ can be saved either on the VNP side (solid line in the figure) or on the PNP side (dotted line in the figure), like in the Chen *et al.* [9.12] and Guilloud *et al.* [9.24] architectures.

Horizontal interleaving schedule

This second type of architecture illustrates the horizontal interleaving schedule, proposed by Mansour *et al.* [9.41] in a particular case of the turbo decoding of LDPC codes. In this example, illustrated in Figure 9.16, there are $P = 3$ PNPs that operate simultaneously in compact mode. As $\alpha_p = 4$ and $\alpha_v = 3$, there are 3 VNPs that operate simultaneously in distributed mode, which gives a computing power of 3 branches per cycle.

The data paths of the VNPs and PNPs are both of the total sum type, with a serial implementation. From time T_1, $d_c = 4$ messages $L_{j,p}$ enter serially into the PNP. After a computation latency of $T_2 - T_1$, the messages $Z_{j,p}$ calculated are sent back, again serially, to the VNPs which are controlled in distributed mode. But in this case, the update of the information is immediate. This is translated by using a single block of *Lacc* memory. Thus, the sum of the extrinsic information of the j bits is updated as soon as a new input $Z_{j,p}$ arrives.

Parameters		Values
Message propagation architecture		($\alpha_p = d_c = 4$, $\alpha_v = d_v = 3$, $P = 3$)
Position of the interconnection network		4
VNP	Control	Compact
	Data path	Total sum, serial
PNP	Control	Distributed, immediate update
	Data path	Total sum, serial

Table 9.6 – Values of the parameters characterizing vertical interleaving architecture.

9.2.6 Sub-optimal decoding algorithm

In order to reduce the complexity of the LDPC decoder, many "sub-optimal" decoding algorithms have been proposed. These algorithms are based on the same principle: reduction in complexity and in memory of the (parity or variable) node processors, by replacing the individual computation of the d output messages (with d the degree of the node) by the computation of Δ ($\Delta < d$) distinct values. Of course, using a sub-optimal algorithm generally degrades the performance of the code. A compromise thus has to be found between performance and complexity.

Single message decoding algorithm ($\Delta = 1$)

This is the simplest algorithm since all the outputs of the (variable or parity) node processor are assigned a same single value at each step in the iterative process.

VNP with $\Delta = 1$

In this technique, the VNP simply returns L_j to the parity constraints to which it is connected. Thus, it is no longer necessary to memorize the $Z_{j,p}$ messages since the latter are no longer used by the VNP. There results a significant economy in memory. This algorithm, APP algorithm, was first proposed by Fossorier *et al.* in [9.20] , and taken up again by E. Yeo *et al.* in [9.66].

Note that the hypothesis of independence between the messages leaving and entering a parity node is absolutely not verified. That is why the iterative

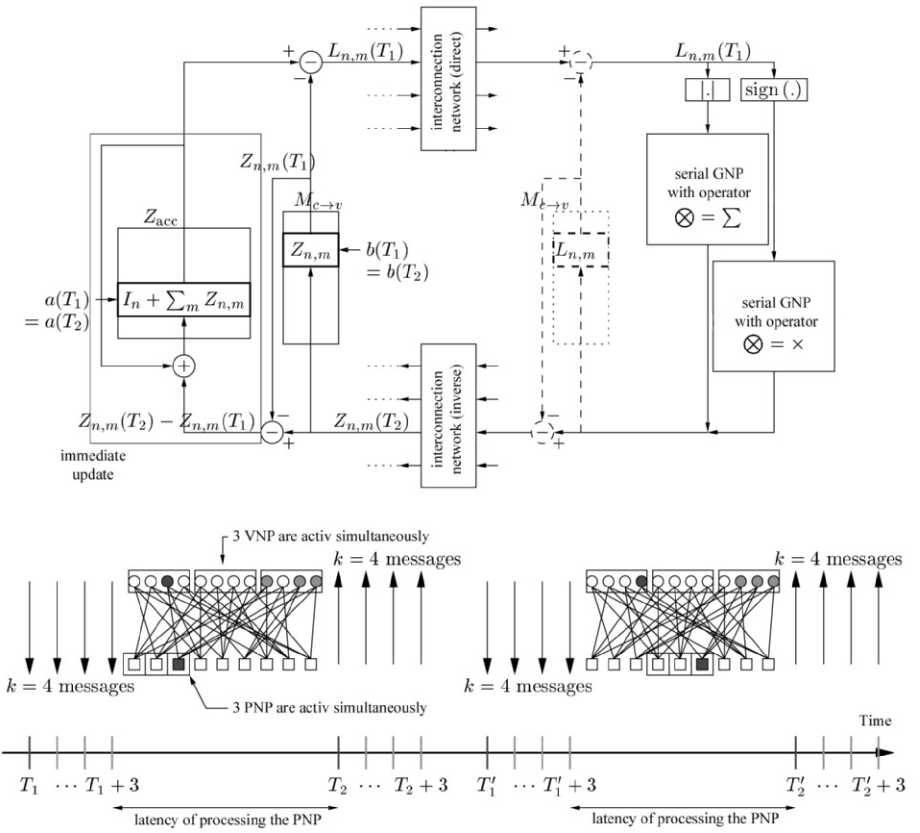

Figure 9.16 – Example of architecture for a vertical interleaving schedule. The serial implementation of the PNP is not detailed in this figure.

decoding algorithm diverges very rapidly as it is subject to the self-confirmation phenomenon: the propagation of the information occurs as if cycles with length 2 existed in the graph.

PNP with $\Delta = 1$

This is the algorithm symmetric to the previous one: the PNP returns a unique value. This technique, which is very efficient in terms of complexity, enables the algorithm to reach its correction capacity in very few iterations, typically 5. Although its correction capacity is very low in relation to the BP algorithm, it is interesting to note that for 5 iterations, such an algorithm is more efficient than the BP algorithm after this same number of iterations. Thus, such procedures can be successfully applied for high data-rate applications where only a reduced number of iterations can be performed.

Sub-optimal PNP algorithms ($\Delta > 1$)

In the state of the art there are three algorithms for $\Delta > 1$ concerning the PNP.

Min-sum or BP-Based algorithm ($\Delta = 2$)

This algorithm proposed by Fossorier *et al.* [9.20] requires no computations in the PNP. Indeed, the authors suggest approximating parity processing algorithm (9.22) by:

$$\begin{cases} |Z_{j,p}| = \underset{j' \in J(p)/j}{\text{Min}} (|L_{j',p}|) \\ \text{sign}\,(Z_{j,p}) = \prod\limits_{j' \in J(p)/j} \text{sign}\,(L_{j',p}) \end{cases} \tag{9.33}$$

Only the computation of the magnitude changes: it is approximated by excess by the minimum of the magnitudes of the messages entering the PNP. Processing in the PNP therefore involves only computing the sign and sorting the two lowest magnitudes of the input messages. Note that this approximation makes the iterative decoding processing independent of the knowledge of the level of noise σ^2 of the channel. The loss in performance is of the order of around 1 dB compared to the *BP* algorithm.

This approximation by excess of the Min-Sum algorithm can however be compensated by simple methods. It is thus possible to reduce the value of $|Z_{j,p}|$ by assigning it a multiplicative factor A strictly lower than 1. It is also possible to subtract from it an offset B (B>0), taking the precaution, however, of saturating the result to zero if the result of $|Z_{j,p}| - B$ is negative. The value of $|Z_{j,p}|$ corrected $|Z_{j,p}|^c$ is therefore:

$$\begin{cases} |Z_{j,p}|^c = A \times \max(|Z_{j,p}| - B, 0) \\ \text{sign}\,(Z_{j,p}) = \prod\limits_{j' \in J(p)/j} \text{sign}\,(L_{j',p}) \end{cases} \tag{9.34}$$

These two variants of the Min-sum algorithm are called *Offset BP-based* and *Normalized BP-Based* [9.10] respectively. The optimization of coefficients A and B enables decoders to be differentiated. They can be constant or variable according to the signal to noise ratio, the degree of the parity constraint, or the processed iteration number, etc.

$\lambda - \min algorithm (\Delta = \lambda + 1)$

This algorithm was presented initially by Hu *et al.* [9.27, 9.28] then reformulated independently by Guilloud *et al.* [9.24]. Function f, defined by Equation (9.35), is such that $f(x)$ is large for low x, and low when x is large. Thus, the sum in (9.35) can be approximated by its λ highest values, that is to say, by the λ lowest values of $|L_{j,p}|$. Once the set denoted $J_\lambda(p)$ of minima λ is obtained, the PNP will calculate $\Delta = \lambda + 1$ distinct magnitudes:

$$|Z_{j,p}| = f\left(\sum_{j' \in J_\lambda(p)/j} f\,|L_{j',p}|\right) \quad \text{with} \quad f\,(x) = \ln\,\tanh\left(\frac{x}{2}\right) \tag{9.35}$$

Indeed, if j' is the index of a bit having sent one of the values of the set of minima, the magnitude is calculated on all the $\lambda - 1$ other minima (λ computations on $\lambda - 1$ values). However, for all the bits, the same magnitude is returned (a computation on λ values). It must be noted that the performance of the $\lambda - \min$ algorithm can be improved by adding a correction factor A and B as defined in equation (9.34).

$A - \min * algorithm(\Delta = 2)$

The last sub-optimal algorithm published so far is called the "$A - \min *$" algorithm and was proposed by Jones *et al.* [9.32]. Here also, the first step is to find index j_0 of the bit having the message with the lowest module: $j_0 = Arg\,\mathrm{Min}_{j \in J(p)}(|L_{j,p}|)$. Then, two distinct messages are calculated:

$$\text{If} \quad j = j_0 : |Z_{j_0,p}| = f\left(\sum_{j \in J(p)/j_0} f\,|L_{j,p}|\right) \tag{9.36}$$

$$\text{If not} \quad j \neq j_0 : |Z_{j,p}| = f\left(\sum_{j \in J(p)} f\,|L_{j,p}|\right) \tag{9.37}$$

A comparison of performance in terms of binary error rate and packet error rate between the sub-optimal algorithms and the BP algorithm is presented in Figure 9.17. The code simulated is an irregular code with size $n = 1008$ and rate $R = 0,5$, built by Hu (*IBM Zurich Research Labs*) with the help of the PEG (*Progressive Edge Growth*) technique [9.29]. The degrees of the parities are thus almost all equal to 8. The degrees of the variables vary from 2 to 15. We see that the sub-optimal algorithm is not necessarily less efficient than the BP algorithm (typically for the $A - \min *$ algorithm). No compensation of extrinsic information was used for these simulations. It is possible to greatly improve the performance of these algorithms by adding this compensation to them, like for the *offset* and *normalized BP-based* algorithms cited above.

9.2.7 Influence of quantization

Quantization is likely to create a degradation in performance by essentially affecting two points: the computation of intrinsic information I_i defined by (9.21) and the computation of branch (9.22) and total (9.24) extrinsic information.

We distinguish two quantization parameters: the number of quantization bits n_q and the quantization dynamic δ. The representable values thus lie between $-\delta$ and $+\delta$. This means that the position of the decimal point is not necessarily defined in the n_q bits or, to put it another way, that the coding dynamic is not necessarily a power of two. Thus, the bit with the lowest weight will be equal to:

$$q_{\mathrm{LSB}} = \frac{\delta}{2^{n_q - 1}} \tag{9.38}$$

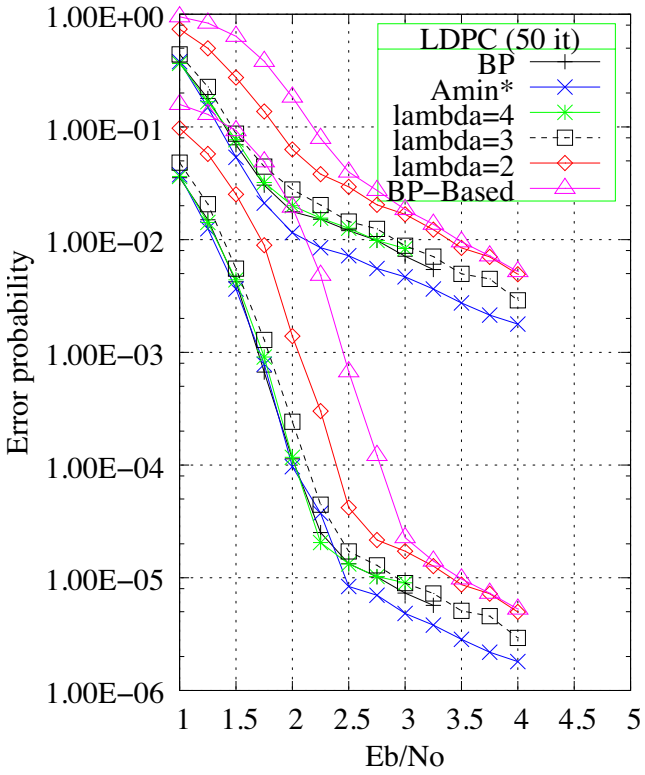

Figure 9.17 – Comparison of performance between the $3-\min$ and $A-\min*$ algorithms in the case of decoding an irregular code C_3.

and we pass from a quantified scalar a_q to a non-quantified scalar a by the relations:

$$\begin{cases} a_q = \text{trunc}\left(a\frac{2^{n_q-1}}{\delta} + 0.5\right), \\ a = a_q\frac{\delta}{2^{n_q-1}} \end{cases} \tag{9.39}$$

where *trunc* designates the truncature operation.
These two parameters can influence the decoding performance. Too low a dynamic allows an error floor to appear on the error rate curves. This floor appears much earlier than that associated with the minimum distance of the code.

However it is important to note that increasing the dynamic without increasing the number of quantization bits increases the value of the bit with the lowest weight, and consequently decreases the precision of the computations done in the PNP. Increasing the dynamic without increasing the number of bits degrades the decoding performance in the convergence zone.

The decoding performance obtained in practice is very close to that obtained with floating decimal points for quantizations on 4 to 6 bits (see the table in Figure 9.18). The influence of the parameters can be studied by the density evolution algorithm [9.15, 9.11].

9.2.8 State of the art of published LDPC decoder architectures

The table in Figure 9.18 groups the main characteristics of LDPC decoder circuits published in the literature so far. The inputs of the table are as follows:

- Circuit: description of the type of circuit used (ASIC or FPGA).
- Authors: reference to authors and articles concerning the platform.
- Architecture:
  - Decoder: indication of the type of decoder (serial, parallel or mixed) with parameters (P, α_p, α_v) associated with the message propagation architecture.
  - Data path: indication for each node processor (variable and constraint) of type of architecture used (direct, trellis or total sum).
  - Control: indication for each node processor (variable and constraint) of type of control used, compact or distributed and, if applicable, of type of update.
  - Position of the interconnection network (between 1 and 4).
- Characteristics of the LDPC code: size, rate and regularity of the LDPC code.
- quantization format: number of bits used to represent the data in the decoder.
- Clock frequency of the chip in MHz.
- Data rate (in bits per second). The information binary rate is obtained by multiplying by the coding rate.
- Maximum number of iterations.

So far, no architecture has been published that describes in detail a decoder using a sub-optimal algorithm. However, we can mention that of Jones *et al.* [9.32], but which contains too little information to be classified here.

Circuit	Authors	Architecture								Characteristics of the code				Coding format (bits)	Clock frequency (MHz)	Rate (Mb/s)	Max number of iterations
		Decoder				Data path		Control		position of inter-connection network	n	m	R	degrees			
		P	α	B		PNP	PNV	PNP	PNV								
Synthesis	Levine et. al. 2000 [9.53]	Serial				Parallel (no LLR)					1200	900	0,25	dv=3; dc=4	8	50	4
		1	1	1													10
DSP - TMS320C6201	Bhatt et. al. 2000 [9.54]	Serial (?)				Serial (?)				1	200	100		dv=3; dc=6	16	200	0,133
ASIC 0.16um (7.5x7 mm²)	Blanksby et. al. 2002 [9.55]	Parallel				Total-Sum		Compact			1024	512		dv=3,6,7,8 dc=6,7	4	64	1000
		M	1	1									0,5				64
FPGA Xilinx Virtex-E 2600	Zhang et. al.2002 [9.56]	Mixed				Trellis	direct	Compact	Compact	3	9216	4608			5	56	54
		3.dc	1	1										dv=3; dc=6			18
ASIC 0.18um (3.1x4.2 mm²)	Mansour et. al. sept 2003 [9.57]	Mixed				Trellis	Total-Sum	Compact	Distributed, immediate update	1	2048	1024			4	125	1600
		64	1	1													1
ASIC 0.18um (9.41 mm²)	Mansour et. al. dec. 2003 [9.58]	Mixed									2304	1536	2/3	irregular	5	200	192
		64	1	1													10
FPGA Xilinx Virtex-II 8000	Chen. Y. et. al. 2003 [9.41]	Mixed				Total-Sum		compact	Distributed, delayed update		8088	4044	0,5	irregular	6	44	80
		24	1	J												212	376
ASIC 0.11um (2.61 mm²)																	25
ASIC 0.13nm (49.6mm² 0.09nm (15.8mm²)	P. Urard et. al. 2005 [9.59]	360								DVB-S2 + BCH					6	200 300	90 135
ASIC 0.13nm (22.74mm²)	F. Kienle et. al. 2005 [9.60]	360								DVB-S2					6	270	255
																	30

Figure 9.18 – State of the art of the different platforms published.

Bibliography

[9.1] PB. Ammar, B. Honary, Y. Kou, and S. Lin. Construction of low density parity check codes: a combinatoric design approach. In *Proceedings of IEEE International Symposium on Information Theory (ISIT'02)*, July 2002.

[9.2] M. Ardakani, T.H. Chan, and F.R. Kschischang. Properties of the exit chart for one-dimensional ldpc decoding schemes. In *Proceedings of Canadian Workshop on Information Theory*, May 2003.

[9.3] M. Ardakani and F.R. Kschischang. Designing irregular lpdc codes using exit charts based on message error rate. In *Proceedings of International Symposium on Information Theory (ISIT'02)*, July 2002.

[9.4] C. Berrou and S. Vaton. Computing the minimum distance of linear codes by the error impulse method. In *Proceedings of IEEE International Symposium on Information Theory*, July 2002.

[9.5] C. Berrou, S. Vaton, M. Jézéquel, and C. Douillard. Computing the minimum distance of linear codes by the error impulse method. In *Proceedings of IEEE Global Communication Conference (Globecom'2002)*, pages 1017–1020, Taipei, Taiwan, Nov. 2002.

[9.6] M. Blaum, P. Farrel, and H. Van Tilborg. Chapter 22: Array codes. In *Handbook of Coding Theory*. Elsevier, 1998.

[9.7] J.W. Bond, S. Hui, and H. Schmidt. Constructing low-density parity-check codes. *EUROCOMM 2000. Information Systems for Enhanced Public Safety and Security, IEEE AFCEA*, May 2000.

[9.8] E. Boutillon, J. Castura, and F.R. Kschischang. Decoder-first code design. In *Proceedings of 2nd International Symposium on Turbo Codes & Related Topics*, pages 459–462, Brest, France, 2000.

[9.9] J. Campello and D.S. Modha. Extended bit-filling and ldpc code design. *Proceedings of IEEE Global Telecommunications Conference (GLOBECOM'01)*, pages 985–989, Nov. 2001.

[9.10] J. Chen and M. Fossorier. Near optimum universal belief propagation based decoding of low-density parity check codes. *IEEE Transactions on Communications*, 50:406–414, March 2002.

[9.11] J. Chen and M.P.C. Fossorier. Density evolution for two improved bp-based decoding algorithms of ldpc codes. *IEEE Communications Letters*, 6:208–210, May 2002.

[9.12] Y. Chen and D. Hocevar. A fpga and asic implementation of rate 1/2, 8088-b irregular low density parity check decoder. In *Proceedings of IEEE Global Telecommunications Conference (GLOBECOM'03)*, 1-5 Dec. 2003.

[9.13] S.-Y. Chung. *On the Construction of some Capacity-Approaching Coding Schemes*. PhD thesis, MIT, Cambridge, MA, 2000.

[9.14] S.-Y. Chung, T.J. Richardson, and R.L. Urbanke. Analysis of sum-product decoding of low-density parity-check codes using a gaussian approximation. *IEEE Transactions on Information Theory*, 47, Feb. 2001.

[9.15] D. Declercq. Optimisation et performances des codes ldpc pour des canaux non standards. Master's thesis, Université de Cergy Pontoise, Dec. 2003.

[9.16] D. Divsalar, H. Jin, and R. J. McEliece. Coding theorems for turbo-like codes. In *Proceedings of 36th Allerton Conference on Communication, Control, and Computing*, pages 201–210, Sept. 1998.

[9.17] I.B. Djordjevic, S. Sankaranarayanan, and B.V. Vasic. Projective-plane iteratively decodable block codes for wdm high-speed long-haul transmission systems. *Journal of Lightwave Technology*, 22, March 2004.

[9.18] E. Eleftheriou and S. Olcer. Low-density parity-check codes for digital subscriber lines. In *Proceedings of International Conference on Communications*, pages 1752–1757, 28 Apr.-2 May 2002.

[9.19] J. L. Fan. Array codes as low-density parity-check codes. In *Proceedings of 2nd Symposium on Turbo Codes*, pages 543–546, Brest, France, Sept. 2000.

[9.20] M.P.C. Fossorier, M. Mihaljevic, and I. Imai. Reduced complexity iterative decoding of low-density parity-check codes based on belief propagation. *IEEE Transactions on Commununications*, 47:673–680, May 1999.

[9.21] R. G. Gallager. *Low-Density Parity-Check Codes*. MIT Press, Cambridge, MA, 1963.

[9.22] J. Garcia-Frias and Wei Zhong. Approaching shannon performance by iterative decoding of linear codes with low-density generator matrix. *IEEE Communications Letters*, 7:266–268, June 2003.

[9.23] F. Guilloud. *Generic Architecture for LDPC Codes Decoding*. PhD thesis, ENST Paris, July 2004.

[9.24] F. Guilloud, E. Boutillon, and J.-L. Danger. Lambda-min decoding algorithm of regular and irregular ldpc codes. In *Proceedings of 3rd International Symposium on Turbo Codes & Related Topics*, 1-5 Sept. 2003.

[9.25] D. Haley, A. Grant, and J. Buetefuer. Iterative encoding of low-density parity-check codes. In *Proceedings of IEEE Global Telecommunications Conference (GLOBECOM'02)*, Nov. 2002.

[9.26] X.-Y. Hu, M.P.C. Fossorier, and E. Eleftheriou. On the computation of the minimum distance of low-density parity-check codes. In *Proceedings of IEEE International Conference on Communications (ICC'04)*, 2004.

[9.27] X.-Y. Hu and R. Mittelholzer. An ordered-statistics-based approximation of the sum-product algorithm. In *Proceedings of IEEE International Telecommunications Symposium*, Natal, Brazil, 8-12 Sept. 2002.

[9.28] X.-Y. Hu and R. Mittelholzer. A sorting-based approximation of the sum-product algorithm. *Journal of the Brazilian Telecommunications Society*, 18:54–60, June 2003.

[9.29] X.Y. Hu, E. Eleftheriou, and D.-M. Arnold. Progressive edge-growth tanner graphs. In *Proceedings of IEEE Global Telecommunications Conference (GLOBECOM'01)*, Nov. 2001.

[9.30] S.J. Johnson and S.R. Weller. Construction of low-density parity-check codes from kirkman triple systems. In *Proceedings of IEEE Global Telecommunication Conference (GLOBECOM'01)*, volume 2, pages 970–974, 25-29 Nov. 2001.

[9.31] S.J. Johnson and S.R. Weller. Regular low-density parity-check codes from combinatorial designs. In *Proceedings of Information Theory Workshop*, pages 90–92, Sept. 2001.

[9.32] C. Jones, E. Vallés, M. Smith, and J. Villasenor. Approximate min* constraint node updating for ldpc code decoding. In *Proceedings of IEEE Military Communications Conference (MILCOM'03)*, 13-16 Oct. 2003.

[9.33] R. Koetter and P.O. Vontobel. Graph-covers and the iterative decoding of finite length codes. In *Proceedings of 3rd International Symposium on turboCodes & Related Topics*, Sept. 2003.

[9.34] Y. Kou, S. Lin, and M.P.C. Fossorier. Low-density parity-check codes based on finite geometries: A rediscovery and new results. *IEEE Transactions on Information Theory*, 47:2711–2736, Nov. 2001.

[9.35] F.R. Kschischang and B.J. Frey. Iterative decoding of compound codes by probability propagation in graphical models. *IEEE Journal on Selected Areas in Commununications*, 16:219–230, 1998.

[9.36] D. J. C. MacKay. Good error-correcting codes based on very sparse matrices. *IEEE Transactions on Information Theory*, 45(2):399–431, March 1999.

[9.37] D. J. C. MacKay and M. C. Davey. Evaluation of gallager codes for short block length and high rate applications. In B. Marcus and J. Rosenthal, editors, *Codes, Systems and Graphical Models*, volume 123 of *IMA Volumes in Mathematics and its Applications,*, pages 113–130. Springer, New York, 2000.

[9.38] David J.C. MacKay and Michael S. Postol. Weaknesses of margulis and ramanujan-margulis low-density parity-check codes. *Electronic Notes in Theoretical Computer Science*, 74, 2003.

[9.39] D.J.C MacKay and R.M. Neal. Good codes based on very sparse matrices. In *Proceedings of 5th IMA Conference on CryproGraphy and Coding*, Berlin, Germany, 1995.

[9.40] D.J.C MacKay, S.T. Wilson, and M.C. Davey. Comparison of constructions of irregular gallager codes. *IEEE Transactions on Communications*, 47:1449–1454, Oct. 1999.

[9.41] M.M. Mansour and N.R. Shanbhag. Turbo decoder architectures for low-density parity-check codes. In *Proceedings of IEEE Global Telecommunications Conference (GLOBECOM'02)*, 17-21 Nov. 2002.

[9.42] Y. Mao and A.H. Banihashemi. Decoding low-density parity-check codes with probabilistic scheduling. *IEEE Communications Letters*, 5:414–416, Oct. 2001.

[9.43] G. A. Margulis. Explicit construction of graphs without short cycles and low density codes. *Combinatorica*, 2(1):71–78, 1982.

[9.44] R. J. McEliece, D. J. C. MacKay, and J.-F. Cheng. Turbo decoding as an instance of pearle's belief propagation algorithm. *IEEE Journal on Selected Areas in Commununications*, 16:140–152, Feb. 1998.

[9.45] T.R. Oenning and Jaekyun Moon. A low-density generator matrix interpretation of parallel concatenated single bit parity codes. *IEEE Transactions on Magnetics*, 37:737– 741, 2001.

[9.46] T. Okamura. Designing ldpc codes using cyclic shifts. In *Proceedings of IEEE International Symposium on Information Theory (ISIT'03)*, July 2003.

[9.47] A. Prabhakar and K. Narayanan. Pseudorandom construction of low density parity-check codes using linear congruential sequences. *IEEE Transactions on Communications*, 50:1389–1396, Sept. 2002.

[9.48] T.J. Richardson, M.A. Shokrollahi, and R.L. Urbanke. Design of capacity-approaching irregular low-density parity-check codes. *IEEE Transactions on Information Theory*, 47:619–637, Feb. 2001.

[9.49] T.J. Richardson and R.L Urbanke. Efficient encoding of low-density parity-check codes. *IEEE Transactions on Information Theory*, 47:638–656, Feb. 2001.

[9.50] J. Rosenthal and P. Vontobel. Constructions of ldpc codes using ramanujan graphs and ideas from margulis. In *Proceedings of the 38-th Annual Allerton Conference on Communication, Control, and Computing*, pages 248–257, 2000.

[9.51] J. Rosenthal and P. Vontobel. Constructions of regular and irregular ldpc codes using ramanujan graphs and ideas from margulis. In *Proceedings of IEEE International Symposium on Information Theory (ISIT'01)*, page 4, June 2001.

[9.52] M. Sipser and D.A. Spielman. Expander codes. *IEEE Transactions on Information Theory*, 42:1710–1722, Nov. 1996.

[9.53] R. M. Tanner. A recursive approach to low complexity codes. *IEEE Transactions on Information Theory*, IT-271:533–547, Sept. 1981.

[9.54] R.M. Tanner. A [155, 64, 20] sparse graph (ldpc) code. In *Proceedings of IEEE International Symposium on Information Theory*, Sorrento, Italy, June 2000.

[9.55] S. ten Brink. Convergence of iterative decoding. *IEE Electronics Letters*, 35:806–808, May 1999.

[9.56] S. ten Brink. Iterative decoding trajectories of parallel concatenated codes. In *Proceedings of 3rd IEEE ITG Conference on Source and Channel Coding*, pages 75–80, Munich, Germany, Jan. 2000.

[9.57] T. Tian, C. Jones, J.D. Villasenor, and R.D. Wesel. Construction of irregular ldpc codes with low error floors. In *Proceedings of IEEE International Conference on Communications (ICC'03)*, 2003.

[9.58] B. Vasic. Combinatorial constructions of low-density parity check codes for iterative decoding. In *Proceedings of IEEE International Symposium on Information Theory (ISIT'02)*, July 2002.

[9.59] B. Vasic. High-rate low-density parity check codes based on anti-pasch affine geometries. In *Proceedings of IEEE International Conference on Communications (ICC'02)*, volume 3, pages 1332–1336, 2002.

[9.60] B. Vasic, E.M. Kurtas, and A.V. Kuznetsov. Kirkman systems and their application in perpendicular magnetic recording. *IEEE Transactions on Magnetics*, 38:1705–1710, July 2002.

[9.61] B. Vasic, E.M. Kurtas, and A.V. Kuznetsov. Ldpc codes based on mutually orthogonal latin rectangles and their application in perpendicular magnetic recording. *IEEE Transactions on Magnetics*, 38:2346–2348, Sept. 2002.

[9.62] F. Verdier and D. Declercq. A ldpc parity check matrix construction for parallel hardware decoding. In *Proceedings of 3rd International Symposium on Turbo Codes & related topics*, 1-5 Sept. 2003.

[9.63] Eric W. Weisstein. Weisstein. from Mathworld, http://mathworld.wolfram.com/BlockDesign.html.

[9.64] N. Wiberg. *Codes and Decoding on General Graphs.* PhD thesis, Linköping University, 1996.

[9.65] X.-Y.Hu, E. Eleftheriou, D.-M. Arnold, and A. Dholakia. Efficient implementations of the sum-product algorithm for decoding ldpc codes. In *Proceedings of IEEE Global Telecommunications Conference (GLOBECOM'01)*, pages 1036–1036, Nov. 2001.

[9.66] E. Yeo, B. Nikolic, and V. Anantharam. High throughput low-density parity-check decoder architectures. In *Proceedings of IEEE Global Telecommunications Conference (GLOBECOM'01)*, San Antonio, 25-29 Nov. 2001.

[9.67] H. Zhang and J.M.F. Moura. The design of structured regular ldpc codes with large girth. In *Proceedings of IEEE Global Telecommunications Conference (GLOBECOM'03)*, Dec. 2003.