Belief revision and small loops in Gallager-type error-correcting codes

MICHAEL DOUBEZ

Master of Sciences by Research in Pattern Analysis and Neural Networks

ASTON UNIVERSITY

September 2000

ASTON UNIVERSITY

Belief revision and small loops in Gallager-type error-correcting codes

MICHAEL DOUBEZ

Master of Sciences by Research in Pattern Analysis and Neural Networks, 2000

Thesis Summary

Gallager-type error-correcting codes are low density parity check codes (LDPC) which may in specific case nearly saturate Shannon's bound. They are based on the construction of two very sparse matrices, various structures of which have been studied in recent years. We will mainly focus here on constructions that have recently been studied by Kanter and Saad. The aim of this project is to examine the performance of two different decoding LDPC algorithms, belief propagation and belief revision within this framework, as well as that of different coding methods (MN vs. Gallager codes). We will also look at the effect of removing small loops in the matrices on the code's performance.

Keywords: MN code, Cascading codes, Gallager code, Belief Revision, Belief Propagation, Small loop

Acknowledgements

I would like to thanks Dr. David Saad for his help throughout this thesis and his patient reading of the various drafts.

I am also grateful to Renato Vincente who gave me references, documentations and help when the need arose.

A lot of thanks to David J.C. MacKay, reader in natural philosophy at the university of Cambridge for his web site `http://131.111.48.24/mackay/` on which I made extensive use of the resources and papers available.

Finally, I am grateful to Nicolas Brodu for his useful reading and critics of this thesis and for cheering me up during the whole year.

Contents

Chapter 1

Introduction

In the information driven modern world, we rely more and more on reliable transmission and storage of data. Whether on a CD-ROM or in the various methods of telecommunication, the data may be corrupted by noise from the environment: static electricity in the atmosphere, dust on a sensitive surface and so on.

As our needs increase we get closer to the current technology whether in terms of telecommunication bandwidth or in terms of storage capacity. So that the data should not only be stored or transmitted reliably but also efficiently.

In this thesis, we concentrate on the binary case and all equations employ the modulo 2 arithmetic. The work presented here may easily be extended to include different channels and code/message representation.

1.1 Communication over a noisy channel

In 1948, Shannon examined the question of achieving error-free communication in a noisy channel. For a channel of additive noise, the problem is sketched in Figure 1.1.

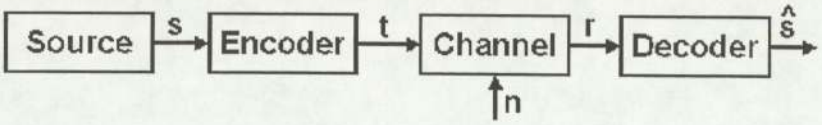


Figure 1.1: Diagram of a general error-correcting system. A source message $\mathbf{s}$ is encoded to a message $\mathbf{t}$ transmitted through a noisy channel which adds some noise $\mathbf{n}$ to the code word. The received message $\mathbf{r}$ is then decoded to recover an estimate of the original message $\hat{\mathbf{s}}$ or declare an error if the recovery process fails.

When encoding a source message $\mathbf{s}$ to a transmitted message $\mathbf{t}$ also known as codeword, redundancy is introduced to compensate for the noise added during transmission. The code rate R is the ratio between the actual information transmitted and the code word length M which in the case of unbiased messages is $R = N/M$.

Shannon proved that for any channel of capacity C, there exists a rate R for which codes capable of achieving perfect retrieval exist, but the proof he provided was non-constructive [1], meaning that we know such a code exists but we don't know how

to achieve it. The theoretically achievable rate R is equal to the channel capacity C defined in the case of the binary symmetric channel by Equation (1.1).

$$C = 1 - H_2(f_n) \tag{1.1}$$

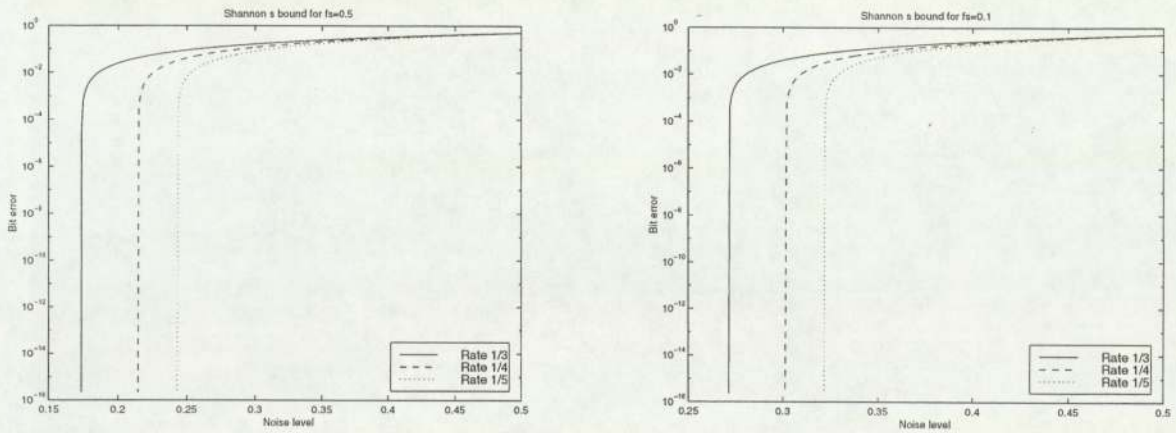
$$H_2(p) = -p \cdot log_2(p) - (1-p) \cdot log_2(1-p) \tag{1.2}$$

with f_n being the flip rate (or noise density)

In the case of biased messages of density f_s in a binary symmetric channel, the actual error free code rate achievable R_c is $R_c = N \cdot H_2(f_s)/M$. In the same way, if a certain amount of bit error p_b is allowed then the code rate achieving Shannon's bound R_c takes the general form:

$$R_c = \frac{N \cdot H_2(f_s)}{M \cdot (1 - H_2(p_b))} \tag{1.3}$$

A plot of the error free code rate achievable defined by the Equation 1.3 is presented in Figure 1.2 for N/M equals to 1/3, 1/4, 1/5



(a) (b)

Figure 1.2: Shannon's bound for a binary symmetric channel and some parameter. (a) Shannon's bound for unbiased messages. (b) Shannon's bound for highly biased messages ($f_s = 0.1$).

A code which saturates Shannon's bound represents the most efficient way of transmitting data as the length of the messages transmitted is then the smallest achievable to retrieve perfectly the data up to the given noise level f_{shannon}.

1.1.1 Error-correcting codes

Error-correcting codes aim at encoding the messages such that after corruption, we are able to retrieve the original message. This is achieved by adding redundancy to the data (creating correlations between the message bits). Then, the encoded messages are able to bear theoretically corruption up to the noise level related to the channel's capacity. In practice, it is very hard to find such a coding scheme that will saturate Shannon's bound.

The general principle of error correcting code is to map with an injective function the messages onto codewords of higher length/dimensionality. Therefore, the transmitted message have a bigger distance between them such that corruption by noise could be tolerated. The transmitted message are called codewords because they do not span the entire space and a message is a codeword if and only it has an antecedent in the space of the source messages by the function of mapping(by the encoding process). The decoded message is related as the closest uncorrupted codeword to the corrupted version. This is illustrated in Figure 1.3.

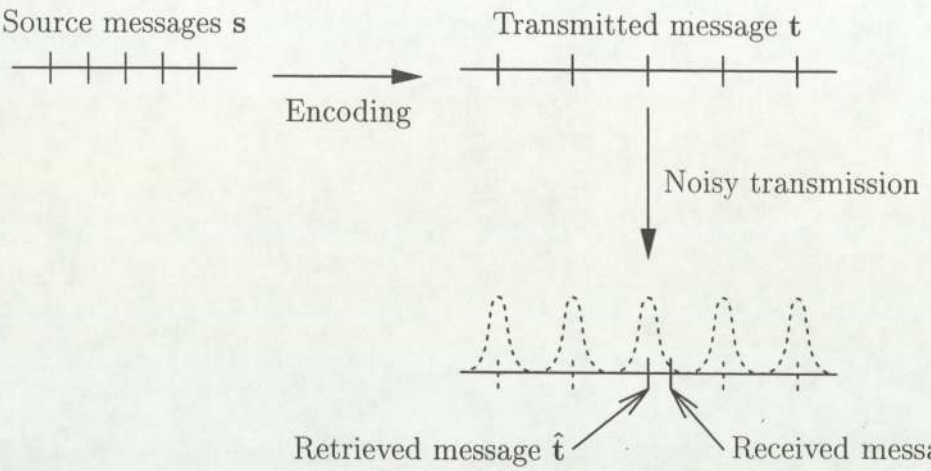


Figure 1.3: Illustration of the error-correcting process in term of distance between codewords. The source messages are mapped to a space of higher dimensionality thus increasing the distance between them. During the transmission, some noise is added. The retrieved message is the the nearest codeword to the received corrupted codeword.

Independently of the method used, we still have residual decoding errors that can be represented in different ways :

percentage of failure the percentage of messages for which the decoding process declared a failure. In the case of iterative decoding, that happens when the decoding process reaches the maximal allowed number of iterations. In the case of Figure 1.3, that happens when the decoding process could not determine the nearest codeword to the corrupted one.

block error(BER) The probability that the decoded message differs from the original one while the decoding process didn't declare a failure. In the case of Figure 1.3, that happens when the decoding process is wrong in determining the transmitted codeword.

bit error(p_b) The probability that a decoded message bit differs from that of the source message in case the decoding process didn't declare a failure. In the case of Figure 1.3, that happens when the decoding process is wrong in determining the transmitted codeword and the difference is measured in terms of overlap between the retrieved message and the transmitted one.

total bit error(tpb) The probability that a decoded message bit differs from that of the source message whether or not the decoding process declared a failure.

The different performance measures will be more relevant to specific tasks.

We will be interested in the bit error if the transmitted data is insensitive to errors in a small fraction of the bits, for instance in transmitting multimedia data where a few pixel changed in a picture or a glitch in sound will not be significant. And we will be more interested in block error if the message must be decoded perfectly like in the transmission of a program or test data values. The percentage of failure indicates the percentage of messages that can be recovered successfully to some degree. Having low error rate is meaningless if the percentage of decoded messages is low (especially in storing information where retransmission is impossible). The total bit error indicates the difference between the original message and the solution suggested by the decoding algorithm. It informs us on whether or not the algorithm was close to the desired solution. It is important if we want to use the data even in the case of a failure in the decoding process.

1.1.2 Linear codes

In the case of linear codes, the message $\mathbf{s}$ is encoded into a transmitted message $\mathbf{t}$ by a linear transformation via a binary generator matrix G:

$$\mathbf{t} = G^T \cdot \mathbf{s} \tag{1.4}$$

Then, in the cases studied, the received message is $\mathbf{r} = \mathbf{t} + \mathbf{n} = G \cdot \mathbf{s} + \mathbf{n}$, corrupted by an M dimensional binary vector$\mathbf{n}$.

We introduce the parity-check matrix H such that $H \cdot G^T = 0$. Then, the problem of estimating the vectors $\mathbf{s}$ and $\mathbf{n}$ from $\mathbf{r} = G^T \cdot \mathbf{s} + \mathbf{n}$ can be transformed by finding the syndrome vector $\mathbf{z} = H \cdot \mathbf{r} = H \cdot G^T \cdot \mathbf{s} + H \cdot \mathbf{r} = H \cdot \mathbf{n}$ and finding the most probable noise vector estimate $\hat{\mathbf{n}}$ such that

$$\mathbf{z} = H \cdot \mathbf{n} \tag{1.5}$$

For instance, a classical example is the (7,4) Hamming code where 7 and 4 are respectively the codeword and the source message length. For a specific realization:

$$G^T = \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{matrix} \qquad H = \begin{matrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{matrix}$$

We can easily verify that $H.G^T = 0$ and the encoding/decoding process in a specific scenario are described as follow:

encoding:	$\mathbf{s} = (1,0,1,1)$	$\Rightarrow$	$\mathbf{t} = G^T \cdot \mathbf{s} = (1,0,1,1,1,0,0)$
corrupting:	$\mathbf{n} = (0,0,0,1,0,0,0)$	$\Rightarrow$	$\mathbf{r} = \mathbf{t} + \mathbf{n} = (1,0,1,0,1,0,0)$
decoding:	$\mathbf{z} = H \cdot \mathbf{r}$	$\Rightarrow$	$\mathbf{z} = (1,1,0)$

Therefore, since $\mathbf{z}$ equals the fourth column of H, the noise corrupted the fourth bit and then $\hat{\mathbf{t}} = (1,0,1,1,1,0,0)$ and $\hat{\mathbf{s}} = (1,0,1,1)$.

This example is very simple due to the small message length. In addition, this code can correct only one bit flip. To correct more than a single bit flip with linear codes, we need to increase the codeword length and, for an arbitrary number of bit flip, there is then no clear practical decoding because as the number of unit elements in the parity check matrix increases, the number of combination to decode from the syndrome vector increases exponentially. This decoding problem is unfortunately NP-hard but we can still obtain an approximated solution by belief revision or belief propagation that will be presented in Chapter 2.

1.2 Gallager-type error correcting codes

Gallager's low-density parity-check code were introduced by Gallager in 1962 [2]. Several variations have been considered over the years; we will concentrate here on the two main variations, the original Gallager code and the MN code.

1.2.1 Gallager codes

Gallager codes are basically linear codes based on the construction of a very sparse (see Appendix B) parity-check matrix $H = [A|B]$ where H is a $(M-N) \times M$ matrix composed of two sub-matrix : an $(M-N) \times N$ random sparse matrix A and an $(M-N) \times (M-N)$ invertible random sparse matrix B. Then, the generator matrix G is constructed as $G = [I_N|(B^{-1}A)^T]$. It is easy to verify that $H \cdot G^T = 0$.

So, the encoding/decoding procedure is as follow: a source message $\mathbf{s}$ is encoded to a transmitted message $\mathbf{t}$ by the Equation (1.6). Some noise $\mathbf{n}$ is added to $\mathbf{t}$ during transmission and the message $\mathbf{r}$ is received (Equation (1.7)). Then we compute the syndrome vector $\mathbf{z}$ that gives us a relation between $\mathbf{z}$, H and $\mathbf{n}$ in Equation (1.8).

$$\mathbf{t} = G^T \cdot \mathbf{s} \tag{1.6}$$

$$\mathbf{r} = G^T \cdot \mathbf{s} + \mathbf{n} \tag{1.7}$$

$$\mathbf{z} = H \cdot (G^T \cdot \mathbf{s} + \mathbf{n}) = H \cdot \mathbf{n} \tag{1.8}$$

The success of convolutional codes, the computational limitations of the time and the fact that Gallager had no efficient decoding algorithm to carry out the decoding process made these codes all but disappear until MacKay and Neal rediscovered them in 1995 while introducing the MN codes [7].

1.2.2 MN code

Basically, MN code is an instance of Gallager codes but instead of taking the generator matrix $G = [I_N|(B^{-1}A)^T]$, one uses $G^T = [B^{-1}A]$. The decoding procedure is a bit different as well because we don't have the property $H \cdot G^T = 0$. The encoding and transmission are kin to Gallager's in Equation (1.9) but the syndrome vector computed

in Equation (1.10) as a different form.

$$\mathbf{t} = B^{-1}A \cdot \mathbf{s} \qquad \Rightarrow \mathbf{r} = B^{-1}A \cdot \mathbf{s} + \mathbf{n} \tag{1.9}$$

$$\mathbf{z} = B \cdot (B^{-1}A \cdot \mathbf{s} + \mathbf{n}) = A \cdot \mathbf{s} + B \cdot \mathbf{n} \qquad \Rightarrow \mathbf{z} = [A|B] \cdot [\mathbf{s}|\mathbf{n}] \tag{1.10}$$

During the decoding process, we not only retrieve the noise vector $\mathbf{n}$ but also the source vector $\mathbf{s}$ itself from the syndrome vector $\mathbf{z}$. MN codes use a linear transformation but are not linear codes [7].

1.3 MN codes - Cascading codes

In this thesis, we determine the construction of a parity check matrix as the way the unit elements are distributed in it. We will make a distinction between the configuration which we define as the process by which the messages are encoded and decoded (i.e. the Gallager vs. MN configuration) and the code construction which we define as the use of specific matrices in a specific configuration. For instance, cascading codes differ from MN codes by the construction of the parity check matrix and the MN code differs from Gallager code by its configuration.

Deciding on the configuration leaves the problem of designing a good matrix construction for it is open.

1.3.1 The construction of matrices

Designing a successful construction, having in mind the decoding process of the form $H \cdot \mathbf{x} = \mathbf{z}$ calls for a trade off [11]. On the one hand, the less unit elements there are in a row of the matrix, the more informative they are in the decoding process as there are less possibilities such that $z_i = \sum_{j \in \{j/H(i,j)=1\}} x_j$ in the system $H \cdot \mathbf{x} = \mathbf{z}$. On the other hand, the more units there are in a column, the more reliable the decoding process is because it has more votes from the check nodes and the contribution of a single wrong message is therefore less critical.

1.3.2 MN construction

In MN constructions, the parity-check matrix is filled by putting C units element per column[1] and having a number of ones per row as uniforming as possible. In this way, each decoded bit has the same contribution from the syndrome vector. It is a good idea to have an odd number of one per column in order to force a decision (otherwise, the votes could be even) and usually, MacKay suggests $C = 3$ as in Figure 1.4 for rate 1/3.

The principal difficulty to build these codes is to ensure the invertibility of the submatrix B (in $H = [A|B]$). MacKay usually generates the submatrices independently: B is generated randomly and some row and column are deleted to make it invertible, then A is generated with the right dimensions to fit with B.

It is to note that these MN codes are based on a regular construction meaning that the number of unit element per row and per column is uniform. Recent studies have shown that irregular construction have better performance [6].

[1] on average but as uniforming as possible

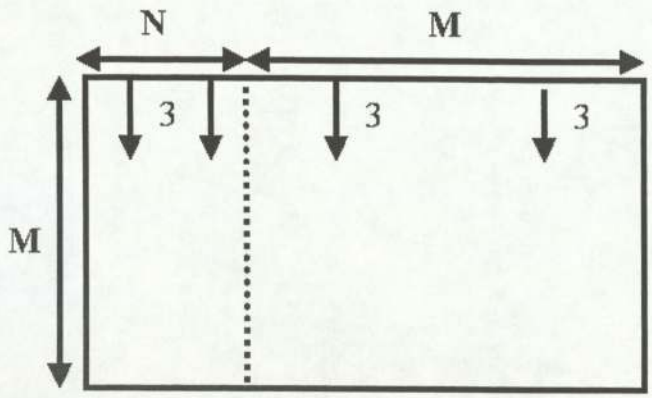


Figure 1.4: MN parity check matrix construction; example for $C = 3$. The source message length is N, M is the transmitted message length. The dotted line represent the separation between the two submatrix A and B that compose the parity check matrix $H = [A|B]$. Arrows with a number stand for the number of unit element per row/column.

1.3.3 Cascading construction

The cascading code is an instance of the MN codes but the construction of the matrix is different, based of insights gained from statistical physics [4]. It is an irregular construction of several submatrices which have the property of a fixed K unit elements per row and C units per column in the submatrix A of the parity check matrix $H = [A|B]$ and L unit per row in the submatrix B. The placement of the unit elements in B is systematic: if $L = 2$ then $B(i,j) = \delta(i,j) + \delta(i,j-5)$ otherwise if $L = 1$ $B(i,j) = \delta(i,j)$. An example of construction for the rate 1/3 is presented in Figure 1.5.

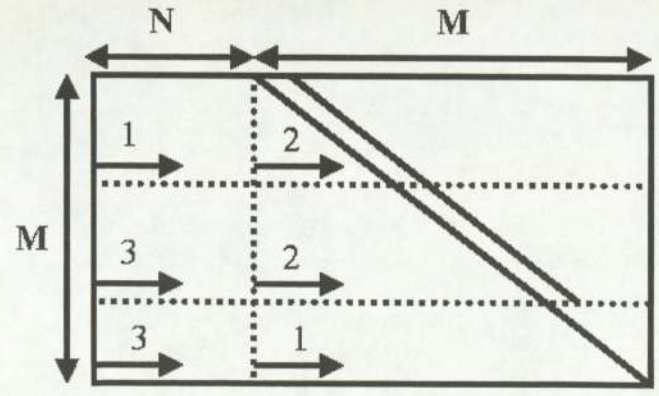


Figure 1.5: Cascading parity check matrix construction [4] example for rate 1/3. An arrow with a number corresponds to the number of unit elements in a row/column in the specified sub-matrices. Dotted lines correspond to the limits of the sub-matrices. Full lines correspond to lines of unit elements, N is the source message length, M is the transmitted message length (=$3N$ in this case).

The first submatrix ($K = 1$, $L = 2$) is present to break the symmetry at the beginning of the decoding process in the case of unbiased messages. If there was more than one unit element per row in all the rows of the $(M \times N)$ submatrix A, the algorithm would heavily depend on its initial conditions as there is no prior preference for the message bits. This will be explained in detail in the next chapter.

The construction used in the various codes rates [4] are given in the table 1.1.

rate R	A	K	B	L
1/3	$N \times N$	1	$N \times 3N$	2
	$3N/4 \times N$	3	$3N/4 \times 3N$	2
	$5N/4 \times N$	3	$5N/4 \times 3N$	1
1/4	$3N/2 \times N$	1	$3N/2 \times 4N$	2
	$N/2 \times N$	3	$N/2 \times 4N$	2
	$2N \times N$	3	$2N \times 4N$	1
1/5	$3N \times N$	1	$3N \times 5N$	2
	$2N \times N$	3	$2N \times 5N$	1

Table 1.1: Constructions of cascading codes for different code rates

Example of construction:

On Figure1.6, we can see on the left hand side the denser part of the parity check matrix. The upper third ($K = 1$) is clearly less dense than the other lower part ($L = 3$). On the right hand side, we have the systematic submatrix composed of first a bidiagonal and then a diagonal.

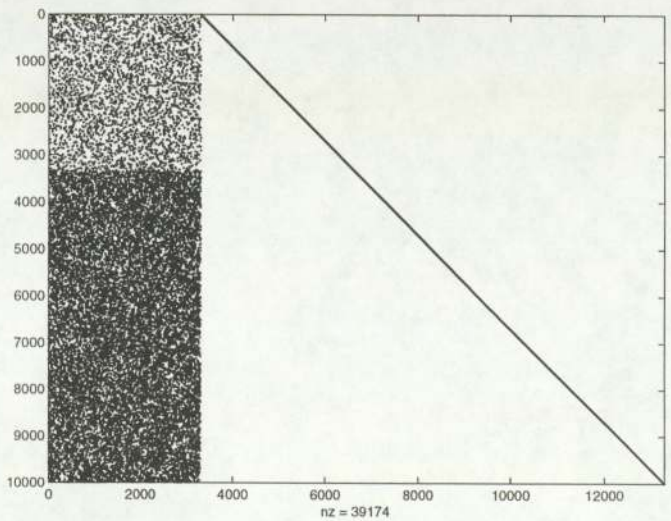


Figure 1.6: Cascading parity check matrix construction [4] example for rate 1/3. The values on the axis correspond to the number of row/column. A dot is drawn each time a unit element is present in the parity check matrix

Chapter 2

Belief propagation versus belief revision

In the first section of this chapter, I present the general decoding process used and emphasise the differences between belief propagation and belief revision. I will also present the motivations underlying this study and the expectations based on previous work related to the comparison between belief propagation and belief revision. Then, in the second section, we will deal with the actual implementation of the decoding process. The third section will present the simulations undertaken, the result of which will be interpreted in the final section.

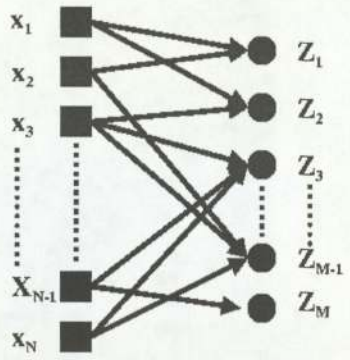
2.1 The decoding process

Our task here is to infer the source message $\mathbf{s}$ given the received message $\mathbf{r}$ with the relation $\mathbf{r} = G^T \cdot \mathbf{s} + \mathbf{n}$. We have seen in the introduction that in the case of MN and Gallager configuration, this can be transformed to finding the most probable vector $\mathbf{x}$ such that $H.\mathbf{x} = \mathbf{z}$ where $\mathbf{x}$ may represent the noise vector or both the signal and noise vectors, depending on the configuration used. Finding the most probable estimate is an NP-hard problem because of the form of the posterior $P(\mathbf{x}|\mathbf{z}) = P(\mathbf{z}|\mathbf{x}) \cdot P(x)/P(\mathbf{z})$ where $P(\mathbf{z}|\mathbf{x})$ takes the value 1 if $H.\mathbf{x} = \mathbf{z}$ and 0 otherwise; so to find $\mathbf{x}$ with the maximal posterior probability, we must go through all possible representation of $\mathbf{x}$ (2^{N+M} possibilities in case of MN configuration).

In this case, we assume that the prior probability distribution of $\mathbf{x}$ is factorizable ($P(\mathbf{x}) = \prod_j P(x_j)$). Given the observed syndrome vector $\mathbf{z}$ we consider the problem onto a Bayesian network which in in the case of this system is a bipartite graph as shown in Figure 2.1. We then use an iterative probabilistic algorithm known as belief update to infer an approximated solution.

We could use directely the recieved message $\mathbf{r}$ given that $\mathbf{r} = G^T \cdot \mathbf{s} + \mathbf{n}$ and then $P(\mathbf{r}|\mathbf{s}) = \prod_j e_j^{t_j(\mathbf{s})}(1 - e_j)^{1-t_j(\mathbf{s})}$ where $t_j(\mathbf{s})$ is the jth bit of the transmitted vector $\mathbf{t} = G^T \cdot \mathbf{s}$ and $e_j = f_n$ if $r_j = 0$ and $1 - f_n$ otherwise. But in this case, the matrix G^T would be dense and the decoding process slower.

In a factor graph, the belief update procedure is called sum-product in the case of belief propagation and min-sum in the case of belief revision (because usually used on



x_1	x_2	x_3	...	x_{N-1}	x_N	
1	1	0	...	0	0	z_1
1	0	1	...	0	0	z_2
0	0	1	...	1	1	z_3
			⋮			
0	1	1	...	0	1	z_{M-1}
0	0	1	...	1	0	z_M

Figure 2.1: Correspondence between the parity-check matrix and the factorgraph. vertices correspond to the unit elements of the parity-check matrix. The rows correspond to the nodes $z_{i1\leq i\leq M}$ of the graph, the column to the nodes $x_{j1\leq j\leq N}$

the log of the probability, otherwise called max-product) [5]. Normally, such algorithms are not expected to work due the presence of loops in the graph however empirically it seems to be useful in providing the correct bit estimate.

2.1.1 Sum-product/min-sum algorithm

Belief update is a message passing algorithm. At each step, we estimate the posterior probabilities $P(x_j|\mathbf{z})$ of each of the bits x_j of $\mathbf{x}$ given $\mathbf{z}$ under the constraint $\mathbf{z} = H \cdot \mathbf{x}$. At each step, nodes send messages to all the other nodes to which they are connected. Given the messages it received each node x_j estimates his probability value and send it to all the nodes it is connected to. As stated in Figure 2.2, at each iteration, first (horizontal step) the nodes x_j send a message q_{ij} to the node z_i which will then (vertical step) send back messages r_{ij} giving the probability of the value of the node x_j given the probability value of the other nodes participating in the check z_i.

Basically, we iterate the message passing algorithm until a vector $\mathbf{x}$ verifying $H \cdot \mathbf{x} = \mathbf{z}$ has been found (successful convergence) or a maximal number of iteration has been reached (declaration of a failure).

To simplify the notations, we will denote $L(i)$ the indexes of all unit elements in row i and $C(j)$ the indexes of the unit elements in the column j. For each unit element of coordinate (i, j) in the parity check matrix, we define the following conditional probabilities.

$$r_{ij}^b = P(z_i|x_j = b) \tag{2.1}$$

$$q_{ij}^b = P(x_j = b|z_{i' \in C(j)\setminus i}) \tag{2.2}$$

The message r_{ij}^b defined in Equation (2.1) provides the probability that the check node z_i send to his parent node x_j and is the probability of having the actual value of z_i given $x_j = b$. The message q_{ij}^b defined in Equation (2.2) provides the probability that $x_j = b$ given the check nodes $z_{i \in C(j)\setminus i}$.

We will denote the prior $P(x_j = b)$ as p_j^b and the pseudo-posterior $P(x_j = b|\mathbf{z})$ as q_j^b.

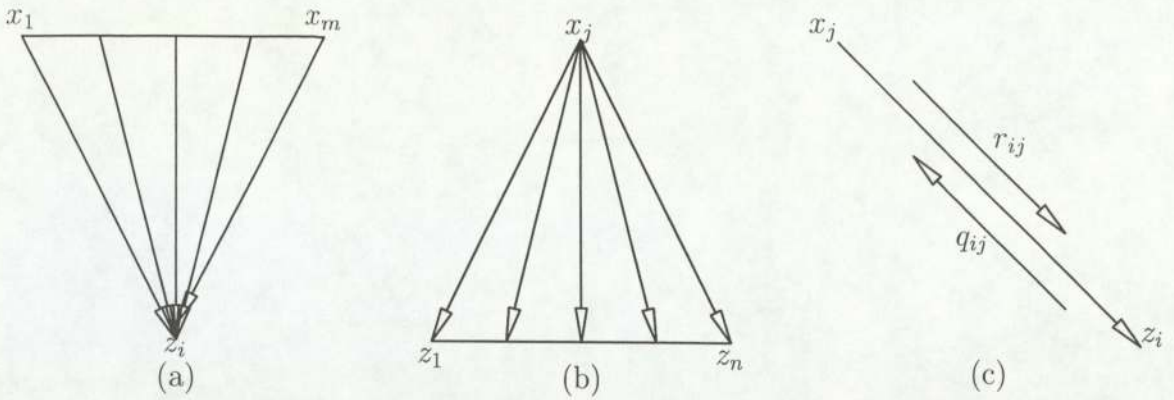


Figure 2.2: The message passing process. The arrows show the causal relationships: the state of z_i is determine by the sum of the incoming x_j; (a)At the horizontal step, we compute the messages going from $\mathbf{x}$ to the check node z_i; (b) At vertical step, we compute the messages going to the source/noise node x_j of the graph from the check vector $\mathbf{z}$. (c) The message going from a node x_i to a node z_j is denoted r_{ij}, the opposite message is denoted q_{ij}

We will now describe in detail an iteration of belief update. The only algorithmic difference between belief revision and belief propagation occurs at the horizontal step.

Initialisation step

This step sets the initial value of the decoded message. A good practice is to set q_{ij}^b to the prior value i.e. for all (i,j) such that $H(i,j)=1$, we set $q_{ij}^b = p_j^b$.

Formally, we should set them to probability $1/2$ but we will see in section 2.2.3 that it has no consequence on the algorithm.

Horizontal step

In this step, we update for each row (check bit z_i) the values r_{ij}^b. Theory provides the relation:

$$P(z_i|x_j=b) = \sum_{\mathbf{x}:x_j=b} P(z_i|\mathbf{x})P(\mathbf{x}|x_j=b) \tag{2.3}$$

In the case of belief propagation, we sum over all the possible configurations of $\{x_{j'\in L(i)\setminus j}\}$ such that $\sum_{j'\in L(i)} x_{j'} = z_i$ given $x_j = b$. This is performed by Equation (2.4).

$$r_{ij}^b = \sum_{x_{j'}/j'\in L(i)\setminus j} P(z_i|x_j=b, x_{j\mathrm{p}rime\in L(i)\setminus j}) \prod_{j'\in L(i)\setminus j} q_{ij'}^b \tag{2.4}$$

In the case of belief revision, we compute the configuration of $\{x_{j'\in L(i)\setminus j}\}$ such that the probability of $\sum_{j'\in L(i)} x_{j'} = z_i$ given $x_j = b$ is maximized. We can see in Equation (2.5) that the sum of Equation (2.4) has been replaced by a max function.

$$r_{ij}^b = \alpha_i \max_{x_{j'}/j'\in L(i)\setminus j} P(z_i|x_j=b, x_{k\in L(i)\setminus j}) \prod_{x_{j'}/j'\in L(i)\setminus j} q_{ij'}^b \tag{2.5}$$

with $\alpha_i = \frac{1}{r_{ij}^0 + r_{ij}^1}$ as a normalization constant.

Vertical step

In this step, we will compute for each column (bit to decode x_j) the values q_{ij}^b and the pseudo-posterior values q_j^b. Bayes rule gives us the relation:

$$P(x_j = b | z_{i' \in C(j) \setminus i}) \propto P(x_j = b) \cdot P(z_{i' \in C(j) \setminus i} | x_j = b) \tag{2.6}$$

which under the assumption of factorizability of the elements $z_{i'}$ becomes:

$$P(x_j = b | z_{i' \in C(j) \setminus i}) \propto P(x_j = b) \prod_{i' \in C(j) \setminus i} P(z_{i'} | x_j = b) \tag{2.7}$$

Therefore, we can compute for all (i, j) such that $H(i, j) = 1$ the messages going from z_i to x_j :

$$q_{ij}^b = \alpha_j p_j^b \prod_{i' \in C(j) \setminus i} r_{i'j}^b \tag{2.8}$$

where $\alpha_j = \frac{1}{q_{ij}^0 + q_{ij}^1}$ is a normalization constant. And if we don't exclude i from the product, we can compute the pseudo posterior:

$$q_j^b = \beta_j p_j^b \prod_{i \in C(j)} r_{ij}^b \tag{2.9}$$

where $\alpha_j = \frac{1}{q_j^0 + q_j^1}$ is again a normalization constant.

Decoding step

We use the pseudo posterior q_j^b to create an estimated vector $\hat{\mathbf{x}}$ such that $\hat{x}_j = 0$ if $q_j^0 \leq 0.5$ and 1 otherwise. Having $H \cdot \hat{\mathbf{x}} = \mathbf{z}$ defines a solution but it doesn't necessarily mean that it is the solution that maximize the posterior $P(\mathbf{x}|\mathbf{z})$; we therefore continue to run the algorithm until the vector stabilize for a certain number of iteration. Since we are only interested in decoding the original message we can relax the halting criterion into obtaining stability for the bits representing the source message estimate

2.1.2 The theoretical differences between belief revision and belief propagation

Belief propagation is also called the maximum marginal (MM) assignment because it computes $\forall i, \max_{x_i} P(x_i|\mathbf{z})$ whereas belief revision is called the maximum a posteriori (MAP) assignment because it computes $\max_{\mathbf{x}} P(\mathbf{x}|\mathbf{z})$. This is processed at the horizontal step while computing the modified term r_{ij}.

Idea of proof: Under the assumption of polytree architecture, nodes with only one parent or one child are bound to exist. This means the value messages send correspond to the true probability and all the nodes in the graph will recursively take the *true* value (this is related to the bucket algorithm). In the same way, the function evaluated at the horizontal step becomes true for all nodes ie. for belief propagation, we marginalize the nodes (we compute $P(x_j|\mathbf{z})$) and for belief revision we maximize $P(\mathbf{x}|\mathbf{z})$.

Normally, belief revision should be what we are looking for because of maximizing the posterior $P(\mathbf{x}|\mathbf{z})$. However, if we are not interested in decoding the message perfectly (block error) but only increasing the overlap between the original message and the decoded one (bit error) then belief propagation performs better.

Another way to put it is that belief propagation evaluates the loss function extracting the ratio of incorrectly decoded bit:

$$L_{\text{belief propagation}}(\mathbf{x}, \hat{\mathbf{x}}) = \frac{1}{N} \sum_j 1 - \delta_{x_j; \hat{x_j}} \tag{2.10}$$

whereas belief revision evaluates the loss function extracting whether the message was correctly decoded or not:

$$L_{\text{belief revision}}(\mathbf{x}, \hat{\mathbf{x}}) = 1 - \prod_j \delta_{x_j; \hat{x_j}} \tag{2.11}$$

A loss function provides a measure for the distance between the original message $\mathbf{x}$ and the recovered message $\hat{\mathbf{x}}$. We can write the loss function as a function of the parameters (source message $\mathbf{s}$, noise message $\mathbf{n}$, parity check matrix H) that minimize the expected loss $E[L]$.

$$E[L] =< L(\mathbf{x}, \hat{\mathbf{x}}) >_{\mathbf{x},\mathbf{n},H} \tag{2.12}$$

where $< f(x,y,z) >_{x,y,z} = \sum_{x,y,z} f(x,y,z) \cdot P(x,y,z)$. Dynamically, a loss function can be assimilated to a cost function we try to minimize.

2.1.3 Expectations

Due to the differences in loss functions, we expect belief revision to perform better than belief propagation in term of the block error obtained because belief revision will minimize the expected loss $E[L_{\text{belief revision}}]$ while belief propagation will minimize the expected loss $E[L_{\text{belief propagation}}]$ resulting in an improved bit error.

$$E[L_{\text{belief revision}}] = \sum_{\mathbf{x},\mathbf{z},H} (1 - \delta(\mathbf{x}; \hat{\mathbf{x}}(\mathbf{z}, H)) P(\mathbf{x}, \mathbf{z}) \tag{2.13}$$

$$E[L_{\text{belief propagation}}] = 1 - \frac{1}{N} \sum_{\mathbf{x},\mathbf{z},H} P(\mathbf{x}, \mathbf{z}) \sum_j \delta(x_j; \hat{x_j}(\mathbf{z}, H)) \tag{2.14}$$

Weiss carried out the same kind of experiment, comparing belief propagation and belief revision decoding performance, in the case of convolutional codes [14][12] and ended up with a significant improvement in block error. Our hope is for a similar result in the case of Gallager-type error correcting codes.

2.2 Implementation of the simulations

In order to carry out the computation, we first generate a sparse parity check matrix H at random according to a specific construction (in this case, the cascading connectivity

construction). Then for each trial, we generate a random source message $\mathbf{s}$, a random noise message $\mathbf{n}$ and we compute the syndrome vector $\mathbf{z}$ with the Equation (2.15) or (2.16).

$$\mathbf{z} = H \cdot [\mathbf{s}|\mathbf{n}] \quad \text{in the case of MN configuration} \tag{2.15}$$

$$\mathbf{z} = H \cdot [\mathbf{n}] \quad \text{in the case of Gallager configuration} \tag{2.16}$$

Then we use the syndrome vector and the appropriate priors for each bit (depending on whether it belongs to the noise or to the source message) to decode and try to retrieve the source vector.

2.2.1 Generation of random vectors

We generate random vectors such that they correspond to the prior probability p^1 of having a unit element at a specific bit of the message. If we iterate on the bits independently, deciding each time at a random number, we will introduce variance in the number of unit elements per vector. Instead, we compute the number of unit elements that should appear in the vector and place them at random. Then, we refer to the probability f defined by the number of unit elements in the vector divided by the length of the message. We are then sure that the prior used in the decoding process is accurate and does not introduce variance in the results.

In order to make relevant comparisons, the generation of source and noise vectors uses distinct random number generator. Each is reinitialized between each simulation at the same value such that, for a certain source message length, the same source messages are always used and, for a certain transmitted message length, at the same noise level, the noise vectors are also the same between different simulations.

In the case of MN code, we generate the source message $\mathbf{s}$ of probability f_s, the noise vector $\mathbf{n}$ of probability f_n and compute $\mathbf{z} = H \cdot [\mathbf{s}|\mathbf{n}]$ providing the priors $\forall j \in [1, N], P(x_j) = f_s$ and $\forall j \in [N, N+M], P(x_j) = f_n$. In the case of Gallager code, we only generate $\mathbf{n}$ and compute $\mathbf{z} = H \cdot \mathbf{n}$ providing the prior $\forall j, p(x_j) = f_n$.

2.2.2 Outputs

The output provided by each trial is as follow:

- a boolean value expressing if the decoder has converged or not with which we can compute the percentage of converging trial.
- the overlap between the original and the decoded message which allows to compute the total bit error. With the help of the convergence indicator, we can compute the bit and block errors.
- the number of iterations required for convergence gives statistics on the convergence speed. If the message has not converged, this value is equal to the maximum number of iterations.

Then those data are processed in order to obtain statistics on the performances of the code.

2.2.3 Implementation of the belief update method

The vertical step is straightforward to compute but for the horizontal step, in the equations

$$r_{ij}^b = \sum_{x_{j'}/j' \in L(i)\setminus j} P(z_i | x_j = b, x_{k \in L(i)\setminus j}) \prod_{x_{j'}/j' \in L(i)\setminus j} q_{ij'}^b$$

$$r_{ij}^b = \alpha_i \max_{x_{j'}/j' \in L(i)} P(z_i | x_j = b, x_{k \in L(i)\setminus j}) \prod_{x_{j'}/j' \in L(i)\setminus j} q_{ij'}^b$$

we have to realize that $P(z_i | x_j = b, x_{k \in L(i)\setminus j})$ takes only two values:

1 if $\sum_{j' \in L(i)} x_{j'} = z_i$
0 otherwise.

So in fact, we compute the sum (or the max) of $\prod_{x_{j'}/j' \in L(i)\setminus j} q_{ij'}^b$ such that $\sum_{j' \in L(i)} x_{j'} = z_i$. It can be represented by a Markov chain with the $q_{ij'}$ as transition probabilities and the partial sum $z_i + \sum_{j_k} x_{j_k}$ as states:

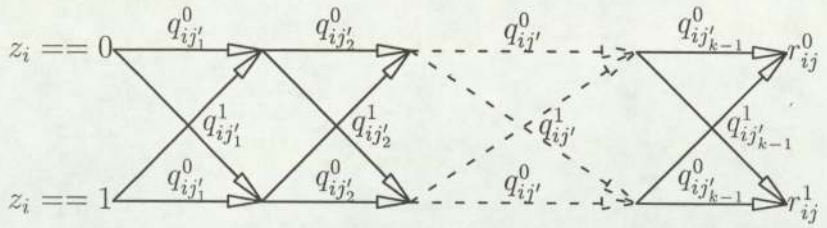


Figure 2.3: Markov chain equivalent to the horizontal step.

And the algorithm is as follows for all rows i and unit element of index j. We will note $r_{ij}^b(\nu)$ the value of r_{ij}^b at the step ν of the Markov chain.

Initialisation

The initial value of the Markov chain starts with the value of z_i: if $z_i = 0$ then $r_{ij}^0(0) = 1$ and $r_{ij}^1(0) = 0$ otherwise $r_{ij}^0(0) = 0$ and $r_{ij}^1(0) = 1$. This is trivial since at this point, the partial sum takes the value of z_i which is given.

Markov chain omitting j

All $j_\nu \in \{H(i, j_\nu) = 1\}$ different from j are part of the Markov chain. For each of them we iterate with the equations 2.17 and 2.18 for belief propagation or with 2.19 and 2.20 for belief revision.

$$r_{ij}^0(\nu+1) = r_{ij}^0(\nu) \cdot q_{ij_\nu}^0 + r_{ij}^1(\nu) \cdot q_{ij_\nu}^1 \tag{2.17}$$

$$r_{ij}^1(\nu+1) = r_{ij}^1(\nu) \cdot q_{ij_\nu}^0 + r_{ij}^0(\nu) \cdot q_{ij_\nu}^1 \tag{2.18}$$

$$r_{ij}^0(\nu+1) = \max\left(r_{ij}^0(\nu) \cdot q_{ij_\nu}^0, r_{ij}^1(\nu) \cdot q_{ij_\nu}^1\right) \tag{2.19}$$

$$r_{ij}^1(\nu+1) = \max\left(r_{ij}^1(\nu) \cdot q_{ij_\nu}^0, r_{ij}^0(\nu) \cdot q_{ij_\nu}^1\right) \tag{2.20}$$

Then, r_{ij}^b is equal to the last value $r_{ij}^b(\nu_{max})$ of the Markov chain. In the case of belief revision, we need to renormalize r_{ij}^b by $r_{ij}^b = r_{ij}^b(\nu_{max})/(r_{ij}^0(\nu_{max}) + r_{ij}^1(\nu_{max}))$. To

make the computation easier, we compute $\delta r_{ij} = r_{ij}^0 - r_{ij}^1$ and $\delta q_{ij} = q_{ij}^0 - q_{ij}^1$ instead of computing r_{ij}^b and q_{ij}^b. The probability p can be recovered from δp by

$$p^0 = \frac{1+\delta p}{2}$$
$$p^1 = \frac{1-\delta p}{2}$$

And the formulas become:

- for belief propagation:

$$\delta r_{ij}(\nu+1) = \delta r_{ij}(\nu) \cdot \delta q_{ij'_\nu}$$

- for belief revision:

$$\delta r_{ij}(\nu+1) = \frac{|\delta r_{ij}(\nu) + \delta q_{ij'_\nu}| - |\delta r_{ij}(\nu) - \delta q_{ij'_\nu}| + 2 \cdot \delta r_{ij}(\nu) \cdot \delta q_{ij'_\nu}}{2 + |\delta r_{ij}(\nu) - \delta q_{ij'_\nu}| + |\delta r_{ij}(\nu) + \delta q_{ij'_\nu}|}$$

The advantage of using the δp representation is for reducing the storage required and for reducing the complexity in the horizontal step.

Initial values

In the implementation, I set the initial values of the q_{ij} to the prior p_j. In fact, we could have set them to 0.5 but then the first horizontal step would have set the $r_{i,j}$ to 0.5 and the first vertical step would have set the q_{ij} to the prior p_j. So, by initializing the q_{ij}, we only gain an iteration and it has no influence on the algorithm.

If in cascading code, the number of unit element per row in the submatrix A of the parity check matrix H=$[A|B]$ was always greater than 1, then, the algorithm wouldn't evolute in the case of unbiased messages (prior for a source message bit equals to 0.5). Because, at the horizontal step, we would always have a δq_{ij} equals to 0 so that all δr_{ij} would be set to 0. Then, at the horizontal step, the q_ij would be reset to their prior which is 0.5 for unbiased messages.

This is the reason why the submatrix of A that have only one element per row is said to break the symmetry in the case of unbiased messages in Section 1.3.3. But it has recently been shown in [16] and [15] that by introducing a weak bias on the prior (ie. we don't set the initial values to exactly 0.5 but a slightly different value), the algorithm evoluates.

Numerical considerations

The convergence of the algorithm is not guaranteed in this case thus I arbitrarily bounded the number of iterations to 200. And since we are inter tested only in the convergence of the source vector bits, I stop when the algorithm has decoded the same source vector bits for 10 iterations.

One should note that due to the numerical approximations and perhaps the small loops[1], the probabilities computed can actually reach 0 or 1. Following the example of MacKay in [7], in order to avoid that, I bounded the probabilities between 10^{-10} and $1-10^{-10}$ (or respectively the values of δq and δr, between $-1+10^{-10}$ and $1-10^{-10}$).

[1] which create some over-confident probabilities. see Chapter 3

2.3 Simulations

In order to compare the performances between belief revision and belief propagation, I have generated a parity check matrix at random respecting the construction of cascading codes the details of which are in F.1.

2.3.1 Performance

The following simulations results are based on 10000 trials of biased and unbiased messages of length $N = 1000$. The same parity check matrix has been used for the different simulations.

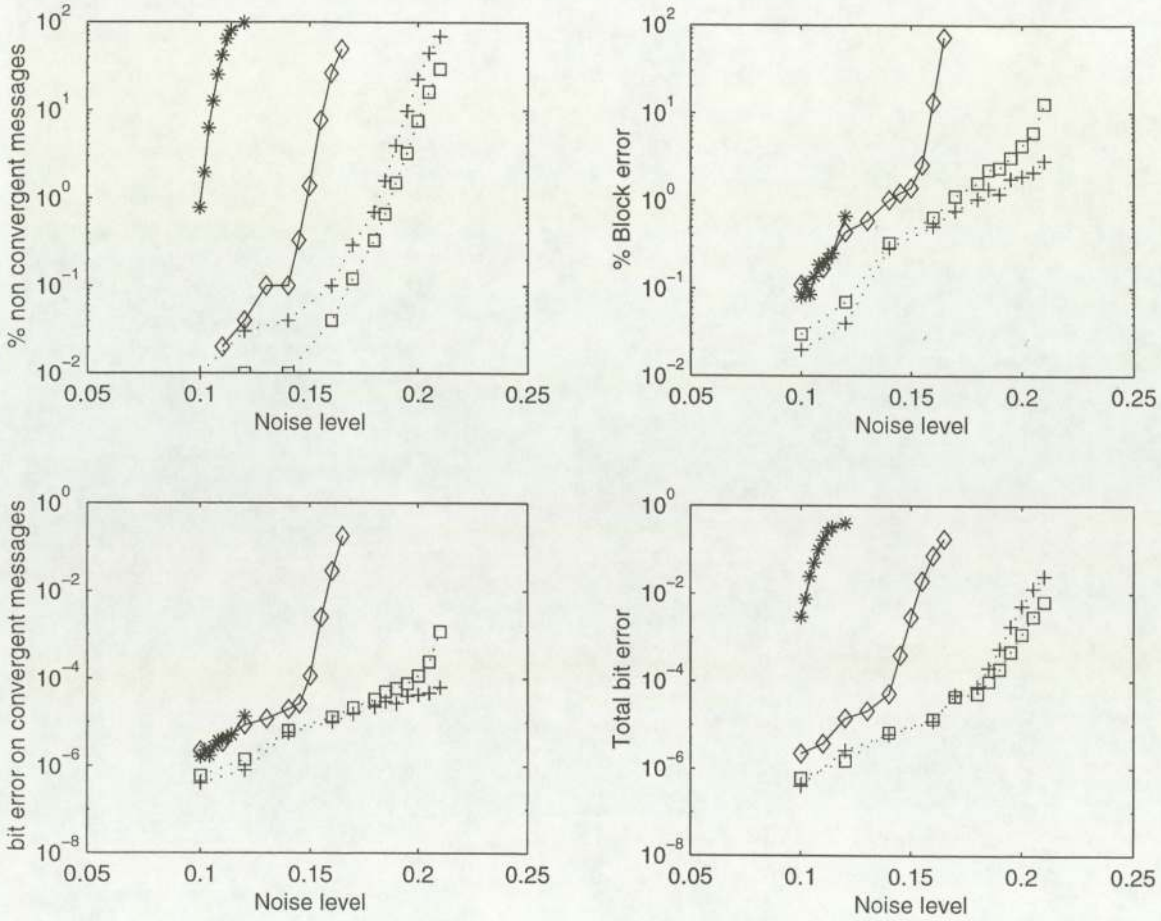


Figure 2.4: Results of comparison between belief propagation and belief revision for cascading code of $R = 1/3$, $N = 1000$

$\Diamond$: Belief propagation, $f_s = 0.5$ $*$: Belief revision, $f_s = 0.5$

$\square$: Belief propagation, $f_s = 0.1$ $+$: Belief revision, $f_s = 0.1$

In Figure 2.4 we can see that there is a great difference between the cases of unbiased and biased messages ($f_s = 0.5$ and $f_s = 0.1$ respectively).

In the case of biased messages, belief revision performs much better regarding both block and bit error; the total bit error is slightly worse, meaning that when belief

revision declares a failure, it is a bit farther from the original message than belief propagation.

In the case of unbiased messages, belief revision and belief propagation are of equivalent performance when belief revision converges.

In both case, belief revision has a worse percentage of convergent cases that renders it impracticable for unbiased messages above noise level 0.12 in this case.

The standard deviation of the bit and total bit errors are in Appendix E.1, they follow the same line has their respective mean: belief revision has a smaller bit error standard deviation in the case of biased messages and an equivalent one in the case of unbiased messages; belief revision has a higher total bit error standard deviation in the case of biased messages and an higher one in the case of unbiased messages; .

2.3.2 Convergence speed

We measure the number of iterations needed for the algorithm to converge, examining their applicability

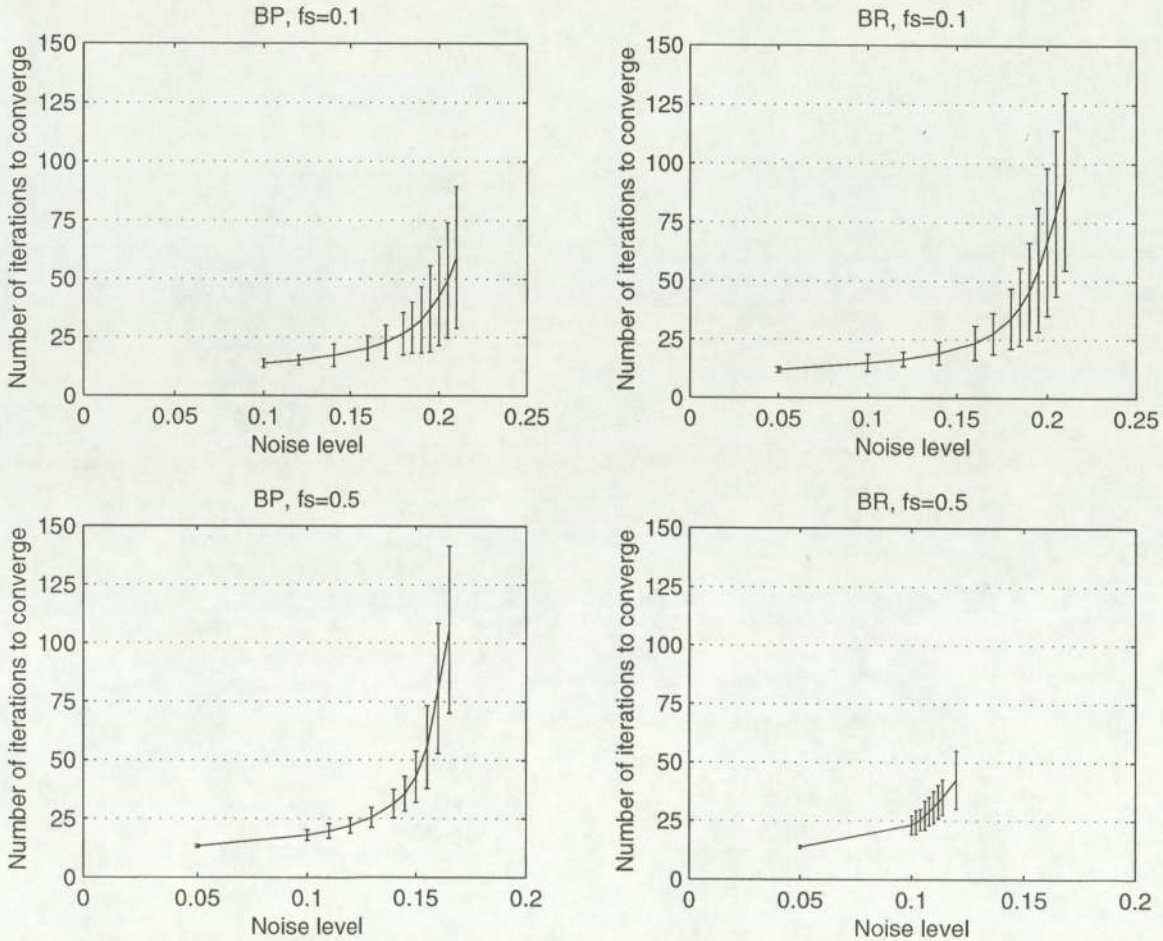


Figure 2.5: Convergence speed of the simulations

In Figure 2.5, we see that belief revision requires more iterations than belief propagation to converge in the case of biased messages. In the case of unbiased messages,

the algorithm converges more quickly but less often. This is suspicious and could mean that the algorithm converges very quickly in a few cases but needs a lot of iteration otherwise. So, in the case of unbiased messages, the poor rate of convergence could be due to the limit we imposed on the number of iteration (200). In order to see if this is the case, we present the curves of the evolving overlaps in Figure 2.6.

2.3.3 Overlap curve

The overlap curve monitors the distance of the current estimate from the original message at iteration i expressed by the Equation (2.21).

$$m_i = -1 + \frac{2}{N} \cdot \sum_{k=1}^{N} \delta(x_k; \hat{x}_k) \tag{2.21}$$

When $m_i = -1$ the decoded message is a mirror image of the original message, while $m_i = 1$ means that we have a perfect estimate. In between, the overlap m_i measures how far from the original message is the decoded message at iteration i. We note that in the case of unbiased messages, the initial overlap is $m_0 = -1 + 2 \cdot \frac{1}{2} = 0$.

In Figure 2.6, I carried out a simulation of 10000 trials reporting the overlap curve on non converging cases in the case of belief revision with unbiased messages at noise level $f_n = 0.11$ ($P(converging) = 41,57\%$). Therefore, we have 4157 overlap curves of non converging cases.

We can see in figure (2.6) that the poor convergence rate of belief revision for unbiased messages is not due to the limitation on the number of iterations since most of the curves end up around 0.2.

2.4 Interpretation of the results

The poor convergence rate of belief revision is certainly due to local minima. In Iba's paper [3], belief revision is called a zero temperature optimizer and belief propagation is called a temperature one optimizer. This means that the fact that belief revision estimate the loss function $1 - \prod \delta_{x_i, \hat{x}_i}$ makes the energy landscape more rugged whereas belief propagation may have a smoother energy landscape.

The issue here is not which methods gives eventually the best result but which has the best dynamical properties. Therefore, belief revision leads to finding a more tortuous path to the global minima and decrease the rate of convergence. This is supported by Figure 2.5 where the number of iteration needed to converge is greater both in mean and in variance for biased messages. The bias on the source messages makes the energy landscape smoother but belief revision still needs more iterations.

Belief propagation has a higher risk of ending up in a local minima as we saw for unbiased messages on the overlap curve. In the remainder of the project, we will use belief propagation because it leads to better performances for unbiased messages and a better rate of convergence in general. In the case of biased messages, it is probably better to use belief revision if we want perfect decoding but the price to pay for it is a longer decoding process and a poor convergence probability.

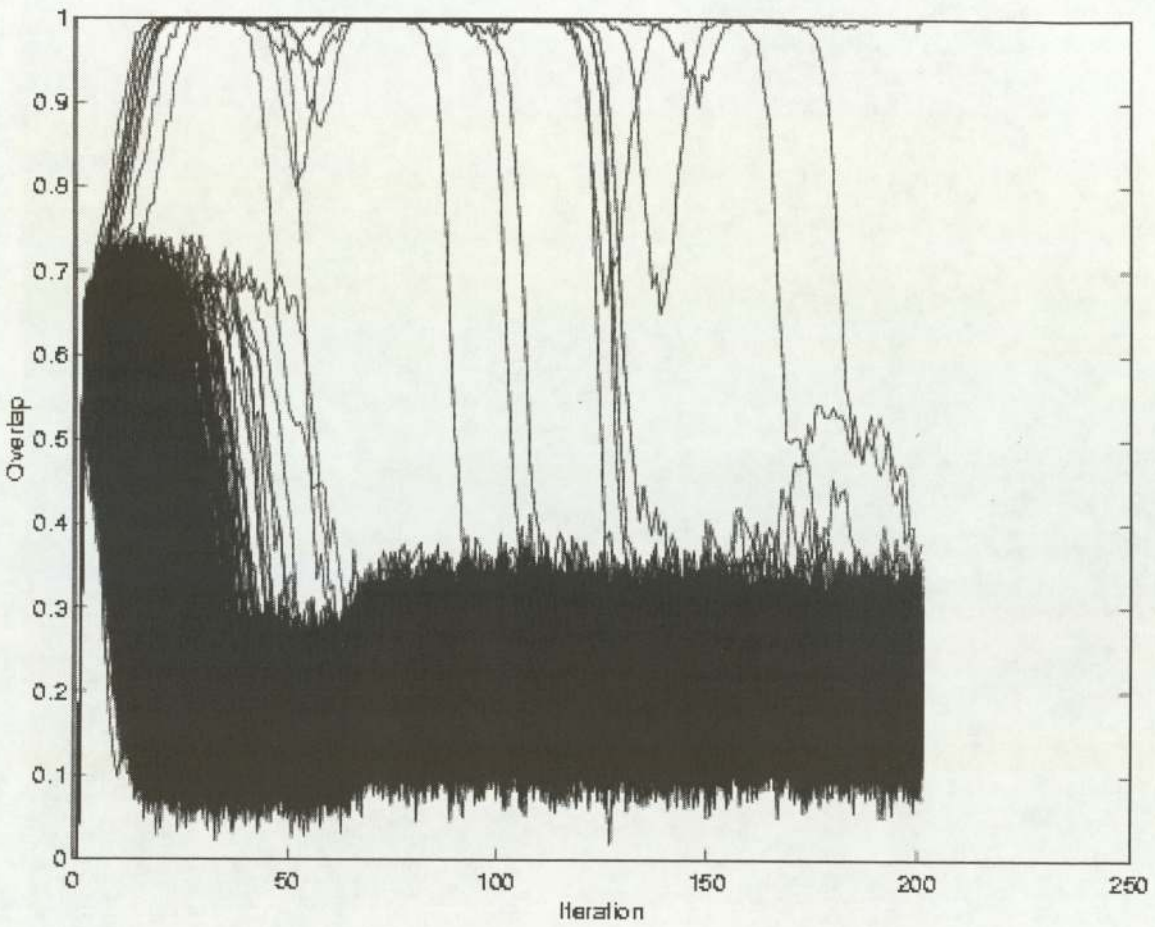


Figure 2.6: Overlap for the decoding process that failed to converge: 4157 curves over 10000 trials for unbiased messages at noise level $f_n = 0.11$ for cascading code $R = 1/3$, $N = 1000$.

Chapter 3

The effect of small loops on the performances

The sum-product algorithm provides an exact solution when applied to a Bayesian network without loops. However, empirically, one observes that belief update provides a good approximation in the occurrence of loops in the graph. The probabilities may not be at the exact value but provide a approximation sufficiently good for making a successful Bayesian estimation of the decoded message.

3.1 The effect of small loops

The problem in belief update of loopy networks is that as it is based on a message passing algorithm, the same message will pass the same nodes again and again, artificially increasing their probability and thus their confidence in the related Bayesian decision. This problem is especially apparent with small loops (length 4) because the same message passes more often. Thus, if the nodes belonging to a small loop support the wrong decision and the input coming from nodes external to the loop is no sufficiently strong as shown in Figure 3.1, this may result in an erroneous estimate and a source of block error.

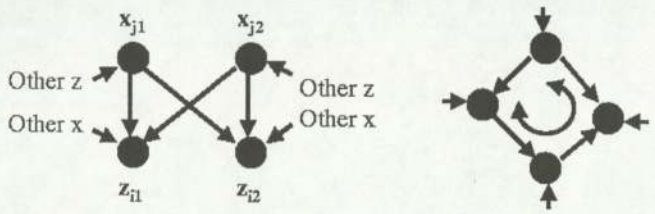


Figure 3.1: A small loop in the factor graph (shown in the left hand side). That means that the messages will continue to contribute to the same nodes indefinitely as shown in the right hand side of the figure; messages pass more often by the same nodes and if the external correlation is not sufficient enough to overcome the overconfidence in posterior created by the small loop, the nodes of the small loop may not converge to the right decoding.

In his experiment, MacKay observed the presence of a flooring effect in the block

error and attributed it to the presence of small loops in the graph because he also observed a significant improvement when he removed those small loops.

We also expect that, at low noise level, small loops will deteriorate the percentage of convergence [13].

3.1.1 The location of small loops

In cascading constructions, there are several ways in which small loops can appear in the parity check matrix $H = [A|B]$. In particular, a small loop may have different effects on the performance depending on the submatrix it is located in. A specific construction is reminded in table 3.1.

submatrix A	submatrix B
SM1 $K = 1$	SM4 L=2
SM2 $K = 3$	SM5 L=2
SM3 $K = 3$	SM6 L=1

Table 3.1: Construction of cascading code for rate 1/3. We identify the submatrices SM1,..., SM6 and indicate the property of the various submatrices: K is the number of unit elements per row with uniform repartition per column. L is the number of unit elements per row with the pattern $B(i,j) = \delta(i,j)$ in the case L=1 and $B(i,j) = \delta(i,j) + \delta(i,j-5)$ in the case $L = 2$

From table 3.1, there is different way in which the loops can appear in the parity check matrix:

- case 1: all the nodes of a small loops are in SM2 or SM3 or both because $K \geq 2$ and so two row can have an overlap greater than or equal to 2.
- case 2: part of the nodes of the loop in submatrix A and part in B; for instance between SM2 and SM5 if in SM2 we have $SM2(i,j) = SM2(i+5,j) = 1$.
- case 3: some small loops with a unit each in {SM1, SM2, SM4, SM5} or {SM2, SM3, SM5, SM6}.

Case 1 is certainly the most probable in random constructions but we will also study the case 2. We will not consider case 3 since it can only contribute a very limited number of loops and has very small probability.

3.1.2 The number of small loops

Small loops are maybe a source of bit error if they contribute strongly toward a wrong estimate and the messages coming from nodes external to the loops are insufficient to make the bits flip back. On the other hand, if he small loops converges to the correct estimate, the probability should converge quickly to 0 or 1. In this case, supposing x_{j_1} and x_{j_2} are in such a small loop converging to the values b_{j_1}, b_{j_2} respectively:

- At the horizontal step
  $r_{ij}^b = \sum_{x_{j'}/j' \in R(i)\backslash j} P(z_i | x_j = b, x_{k \in R(i)\backslash j}) \prod_{x_{j'}/j' \in R(i)\backslash j} q_{ij'}^b$
  becomes
  $r_{ij}^b = \sum_{x_{j'}/j' \in R(i)\backslash \{j,j_1,j_2\}} P(z_i | x_j = b, x_{j_1} = b_{j_1}, x_{j_2} = b_{j_2}, x_{k \in R(i)\backslash \{j,j_1,j_2\}}) \prod_{x_{j'}/j' \in R(i)\backslash j} q_{ij'}^b$
  which is equivalent to setting the nodes x_{j_1} and x_{j_2} to fixed values in this row.

- At the vertical step
  In a small loop, two probabilities $r_{i_1 j}^b = r_{i_2 j}^b 0$ Therefore, in the formula $q_{ij}^b = \alpha_j p_j^b \prod_{i' \in C(j)\backslash i} r_{i'j}^b$ there is necessarily one probability equal to 0 setting $q_{ij}^b = 0$.

- In the end, it is like setting the two nodes x_{j_1} and x_{j_2} in the graph to fixed values.

Thus, the algorithm converges more quickly (in the hypothesis it converges) because there are less cases to consider. An other way to put it is that the nodes in small loops, having high probability, will decrease the complexity of solving the related probabilities of the nodes to which they are connected. Therefore we can expect that increasing the number of small loops will reduce the number of iterations needed for the algorithm to converge.

3.2 Controlling the number of small loops

In order to remove small loops, MacKay first randomly generates a sparse matrix and then deletes the lines and columns responsible for the presence of a small loops [7]. We found this method unsatisfactory in the case of cascading code due to the complex constraints of the construction.

3.2.1 Generating matrices without small loops

The problem is to keep the construction of the matrix (number of unit elements in each row and column and in each submatrix) and build it such that the loops are big. In cascading codes, since the submatrix B if bidiagonal, it is not responsible for any loops so we can allow ourself to keep it as is.

Therefore, we have to avoid creating small loops within A and between A and B. A simple solution is to forbid, during the construction, some sites in each row in order to avoid small loops as described in the following algorithm.

For each submatrix of A, we store the number of unit elements in each row and each column. For each element to add in each row, we compute the distance (as defined later) from the node z_i corresponding to the row to all other nodes x_j that are in the submatrix. Then, we order the nodes first by distance (looking for loops as large as possible) and then, for the nodes of the same distance, we sort them by the number of existing units in the correct column, giving priority to nodes with a low number of existing elements. This heuristic making is aimed at keeping the nodes with the maximum distance and the maximum of flexibility. This is also to avoid being blocked at the end of the construction by having to put more than one node in a row and having only one possibly choice for the column. Then, we select x_j at random from the possible choices we have defined and connect it to the current node z_i (marking a 1 at the indices (i, j) in the parity-check matrix).

A side effect of this method is that we not only eventually remove the loops of length 4 but the remaining loops have very large lengths, this effect increases with the size of the system.

Distance between two nodes

We will call distance between two nodes the minimum length of path connecting them.

In order to compute it, we use an iterative algorithm. We start from a node and store all the nodes adjacent to this node. Those become the nodes of distance 1 from the starting node. The non marked nodes adjacent to the nodes previously stored becomes the nodes of distance 2. We continue until the target node appears in the list of stored nodes or until there are no more nodes unmarked connected to the marked ones. Examining this list we can find the distance between nodes. In case the algorithm ends without returning the target node, there is no path between the nodes and we set the relevant distance to infinity.

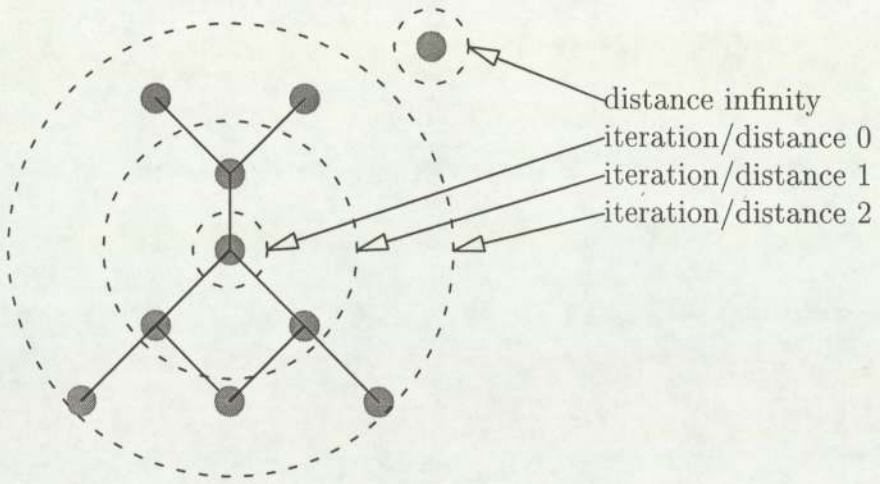


Figure 3.2: Distance between a node and the other nodes in the graph. The circles represents the steps of the greedy algorithm and incidentally the distance to the nodes within them.

To compute the distance between a node and all the others, the complexity of the algorithm increases linearly with the number of rows plus the number of columns of the corresponding matrix. For a matlab-like description of the algorithm, see Appendix D.2.

Construction of a sparse matrix without small loops - an example

At iteration i of the algorithm, we have the parity check matrix in Table 3.2

We compute the distance from z_i to the nodes $\{x_1, .., x_N : W_c > 0\}$ and order them first by loop length and then by the number of links remaining to be added.

So we select at random between x_3 and x_4; let's take x_3 and connect it to z_i (the changes are noted in bold fonts) in Table 3.4

	x_1	x_2	x_3	x_4	x_5	...	x_{M+N}	W_r
z_1	0	0	0	0	0	...	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
z_{i-1}	1	0	0	1	0	...	0	0
z_i	0	0	0	0	0	...	0	3
W_c	0	3	2	2	3	...	0	

Table 3.2: Parity check matrix with large loops. Iteration i, W_r is the number of remaining unit elements to add in each row; W_c is the remaining number of unit elements to add in each column in the submatrix relative to row i

nodes	x_3	x_4	x_2	x_5	x_N	...	x_k
distance	11	11	9	5	5	...	3
W_c	2	2	3	3	2	...	2

Table 3.3: The nodes are sorted first by distance and then by remaining number of unit elements to add in the relative column

	x_1	x_2	x_3	x_4	x_5	...	x_{M+N}	W_r
z_1	0	0	0	0	0	...	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
z_{i-1}	1	0	0	1	0	...	0	0
z_i	0	0	**1**	0	0	...	0	**2**
W_c	0	3	**1**	2	3	...	0	

Table 3.4: Parity check matrix with large loops. Iteration i. A one has been at index $(i, 3)$ creating a loop of length 12. W_c and W_r are decreased accordingly.

We iterate until there is no more elements to add in the row i. We also iterate for all row of the submatrix and for all the submatrices.

One should note that he values of number of element to add per column in a submatrix is computed beforehand such that the repartition is as uniform as possible within the submatrix and the global submatrix A.

3.2.2 Adding small loops

If we want to add a certain number of small loops to a submatrix, we just insert them at random such that a node participates in only one small loops and decrease by 2 the weight of the rows and columns concerned. Then we apply the same algorithm as for creating a code with large loops in order to maximise loop lengths and minimize the effect of loops of small lengths (6 and 8).

The problem with adding small loops arbitrarily is that we can inadvertently create small loops with the submatrix B of the parity check matrix $H = [A|B]$. So we need to control that only the number of small loops we added appears in the graph.

3.2.3 Evaluating the number of influencing loops

Computing the number of small loops in a graph is easily done: we just need to compare every row of the parity check matrix to all other rows and small loops occur when two row have an overlap greater or equal to 2. But MacKay observed a significant improvement when removing the loops of length 4 and 6 although the effect became less clear with the removal of loops of higher length. Therefore, it would be interesting to know their number.

The method used for counting the number of small loops become rapidly non practicable because, for higher loop length, in order to count all the loops of length l, we need to compare all arrangements of $l/2$ rows of the parity check matrix and the complexity is then on the order of $C_M^{l/2} = M!/((l/2)! \cdot (M - l/2)!)$. Counting all the loops in a graph is very difficult, and in this case, not very informative since we are mostly interested in the loops of length 4 and 6 and marginally of loops of higher length.

Graph theory provides us the notion of base of loops: the set of loops from which all loops of a graph can be constructed from by union, difference and intersection. But there is no practical algorithm to obtain the base of loops of minimal length and we could miss some small loops.

Instead, we will concentrate on the loops of minimal length that contribute significantly in the artificial increase of their probability value. The following algorithm provides the exact number of loops of length 4 but only a lower bound for the loops of higher length.

Counting loops

To count loops, we compute the distance between a node and all the other nodes it is connected to without taking into account the direct links (except one).

Then we have contribution of this link to the loops in the graph. We iterate this for all nodes and note down the contribution of the links to loops. Since we are in a bipartite graph, we can avoid double counting the contribution of each link by applying

this algorithm only to one part of the nodes (**x** or **z**). for each loop of length l, l links contribute to this loop. Therefore, by dividing by l the number of contributions c_l of links to the loops of length l, we obtain the number n_l of distinct loops of length l in the graph ($n_l = c_l/l$).

This algorithm gives a lower bound of the number of loops of length l. An example of how this algorithm can be used is provided in Figure 3.3.

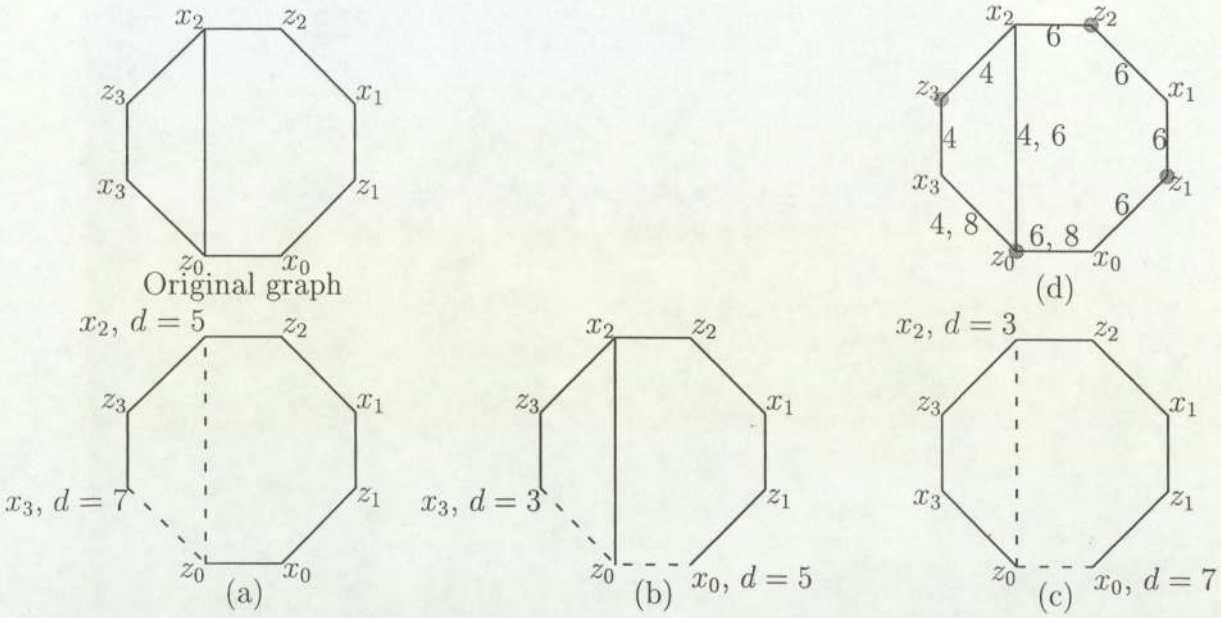


Figure 3.3: Iterations for the contribution of a node to loops in a factor graph. (a) we remove all the links (dashed) from z_0 except the one to x_0. Then we compute the distance between z_0 and $\{x_2, x_3\}$. We know that this link participates in a loop of length 6 with x_2 and a loop of length 8 with x_3. We iterate on the links from z_0 in (b) and (c). We then iterate on all the node z of the graph to obtain the contribution of each link in (d). Eventually, we find $4/4 = 1$ loop of length 4, $6/6 = 1$ loop of length 6 and $2/8 = 1/4$ loop of length 8.

In Figure 3.3, we end up with 1/4 loop of length 8 because the loop of length 8 in this case is composed of loops of smaller length and isn't detected by the algorithm. This is the reason why this algorithm provides only a lower bound.

For a matrix $M \times N$ of density ρ, the complexity of this algorithm is of the order $\rho NM(N + M)$. It takes a few minutes for a 3000×4000 matrix with 11000 unit elements, in C++ on an DS20 Compaq machine. A matlab-like algorithm is provided in Appendix D.3.

3.3 Simulations

3.3.1 Removing small loops

In order to see the effect of removing small loops, I have generate a cascading code which I define as the original construction meaning that it is the random construction where we do not control the number of small loops or where they happen to be. We

observed the presence of 26 small loops in the original construction (see Appendix F.1, 'Original matrix' for details on the repartition of loops) Then I generate a matrix which has the same construction but no small loops.

Performance

The simulation results in Figure 3.4 are based on biased and unbiased messages of length $N = 1000$ and the construction of two matrices corresponding to cascading codes of rate 1/3. Each point represent an average on 10000 trials with belief propagation.

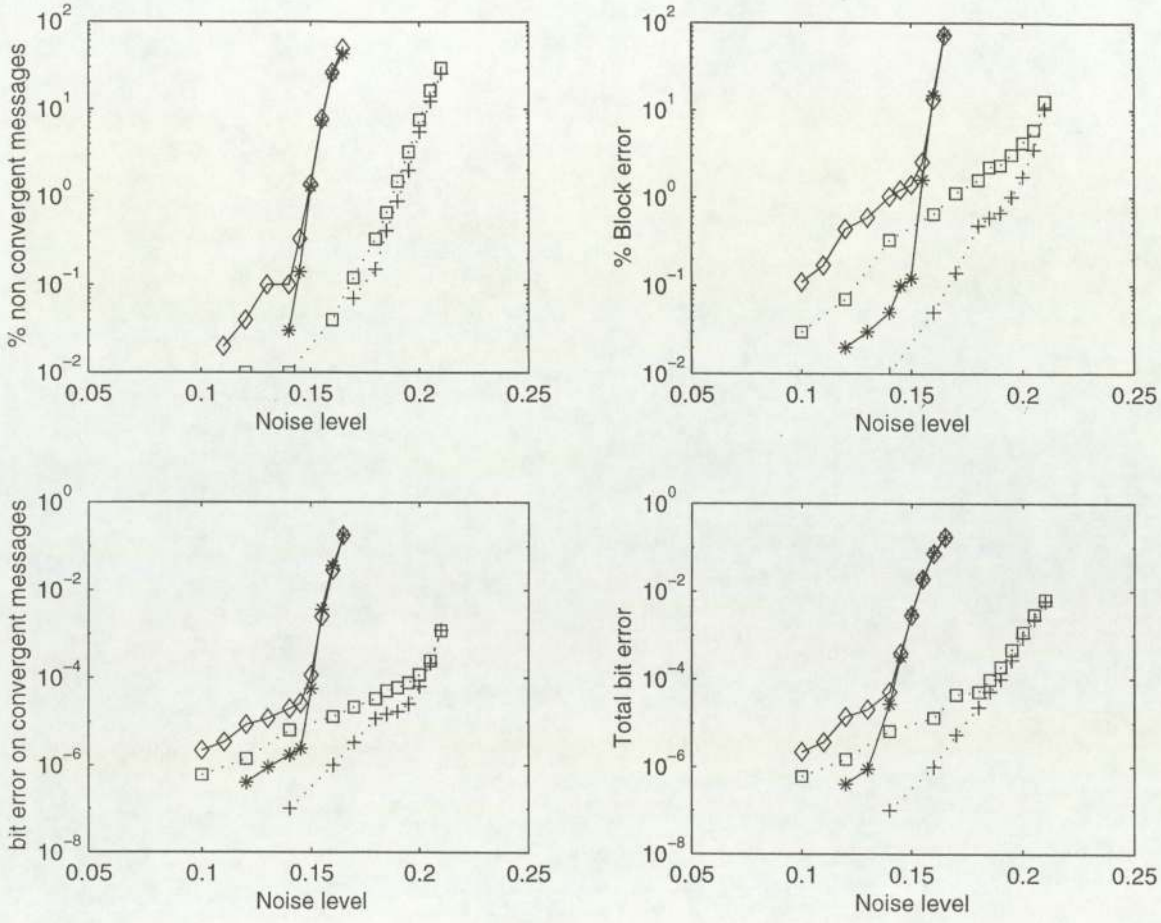


Figure 3.4: Impact of 26 small loops on the performance of cascading code $R = 1/3$, $N = 1000$

$\Diamond$: Original matrix, $f_s = 0.5$ $*$: Large loops, $f_s = 0.5$
$\square$: Original matrix, $f_s = 0.1$ $+$: Large loops, $f_s = 0.1$
full line: $f_s = 0.5$ dotted line: $f_s = 0.1$

We can see on Figure 3.4 that a code without small loops performs better in block and bit errors at low noise level in mean and in variance (see Appendix E.2). There is a very slight improvement in the percentage of convergence in the case of unbiased messages which is clearer in the case of unbiased messages. A higher noise level, there is little difference whether we remove loops or not.

Convergence time

From the same simulations, we examine the effect of small loops on the speed of convergence.

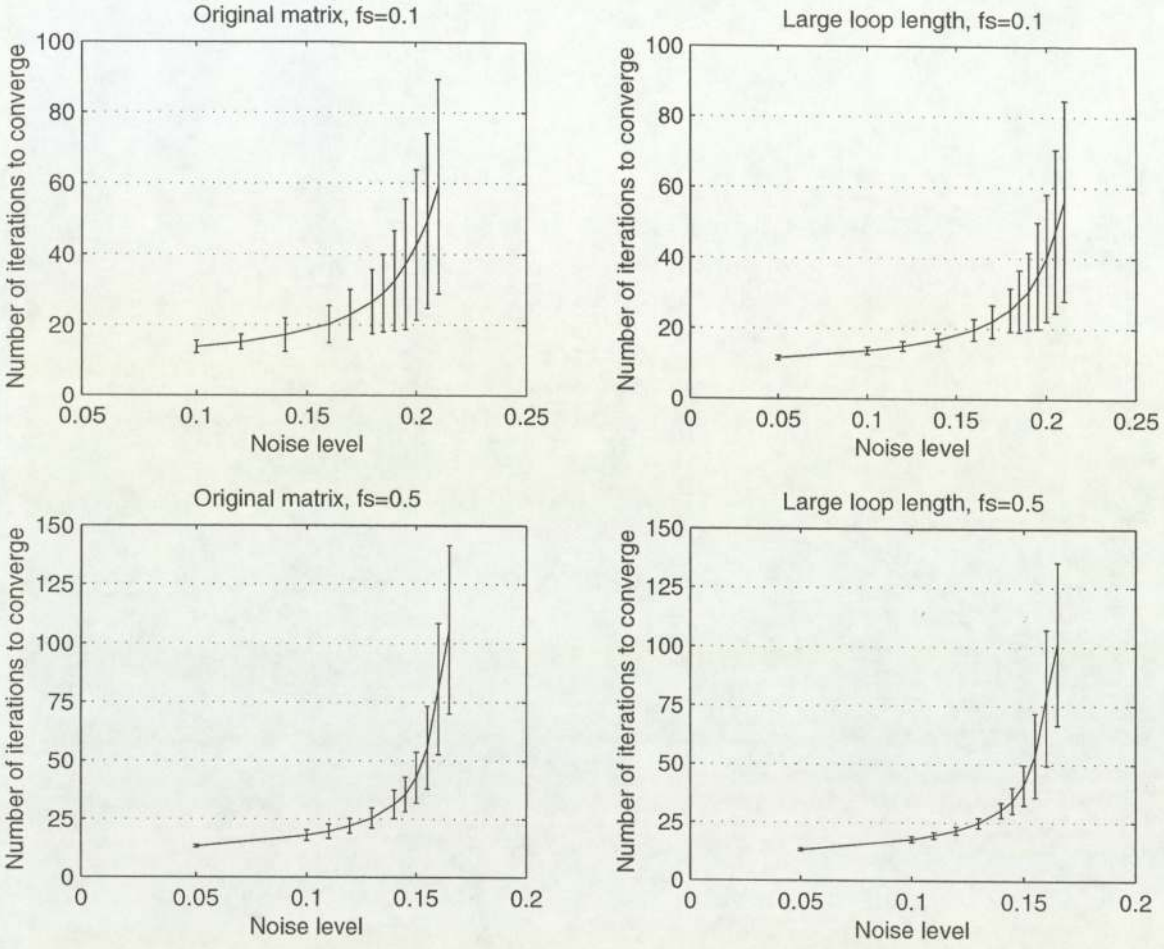


Figure 3.5: Impact of 26 small loops on convergence speed for cascading code $R = 1/3$, N=1000. The curve corresponds to the mean of number of iteration and the bars to the standard deviation

In Figure 3.5, removing small loops improve slightly the mean convergence speed and the standard deviation becomes slightly smaller.

3.3.2 Adding small loops

To evaluate the effect of adding small loops, I artificially added the same number of small loops (26) as in the original matrix in the submatrix of my choice such that a node is never part of two small loops. The other connections are made in order to respect the connectivity and increase other loops length as much as possible as described in Section 3.2.1. We follow the convention and the cases considered of Section 3.1.1

Performance

The simulation results in Figure 3.6 are based on biased and unbiased messages of length $N = 1000$ and the construction of four matrices corresponding to cascading codes of rate 1/3: one with large loops, the others with 26 small loops representing the cases identified in Section 3.1.1. Each point represent an average on 10000 trials with belief propagation.

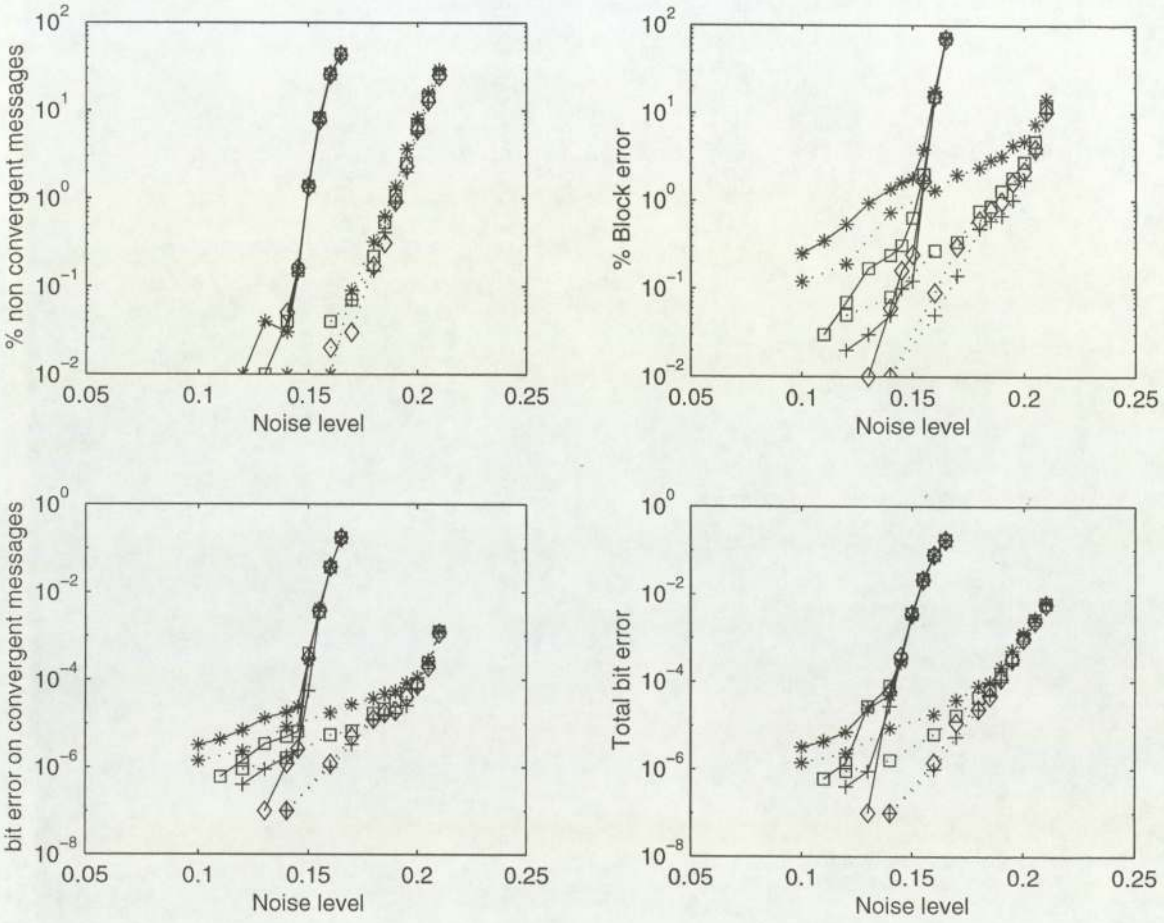


Figure 3.6: Effect of placement of 26 small loops for cascading code $R = 1/3$, N=1000

$+$:	Large loops	$*$:	26 small loops between SM2 and SM3
$\Diamond$:	26 small loops in SM2	$\square$:	26 small loops in SM3
	full line: $f_s = 0.5$		dotted line: $f_s = 0.1$

In the Figure 3.6, the values of percentage of block error and percentage of non-convergent messages bellow 0.1% and the values of bit error bellow 10^{-6} should be ignored because the trials have been carried out on 10000 messages and we reach the simulations precision. In this regard, we can see that placing small loops in SM2 or SM3 has relatively small influence compared to the effect of placing small loops between SM2 and SM5. We could have the impression that small loops in SM2 leads to better performances than adding them in SM3 but other constructions of the same matrices with different seeds have shown that it isn't a characteristic result.

Convergence time

The convergence time curves corresponding to the previous simulations are presented in Figure 3.7.

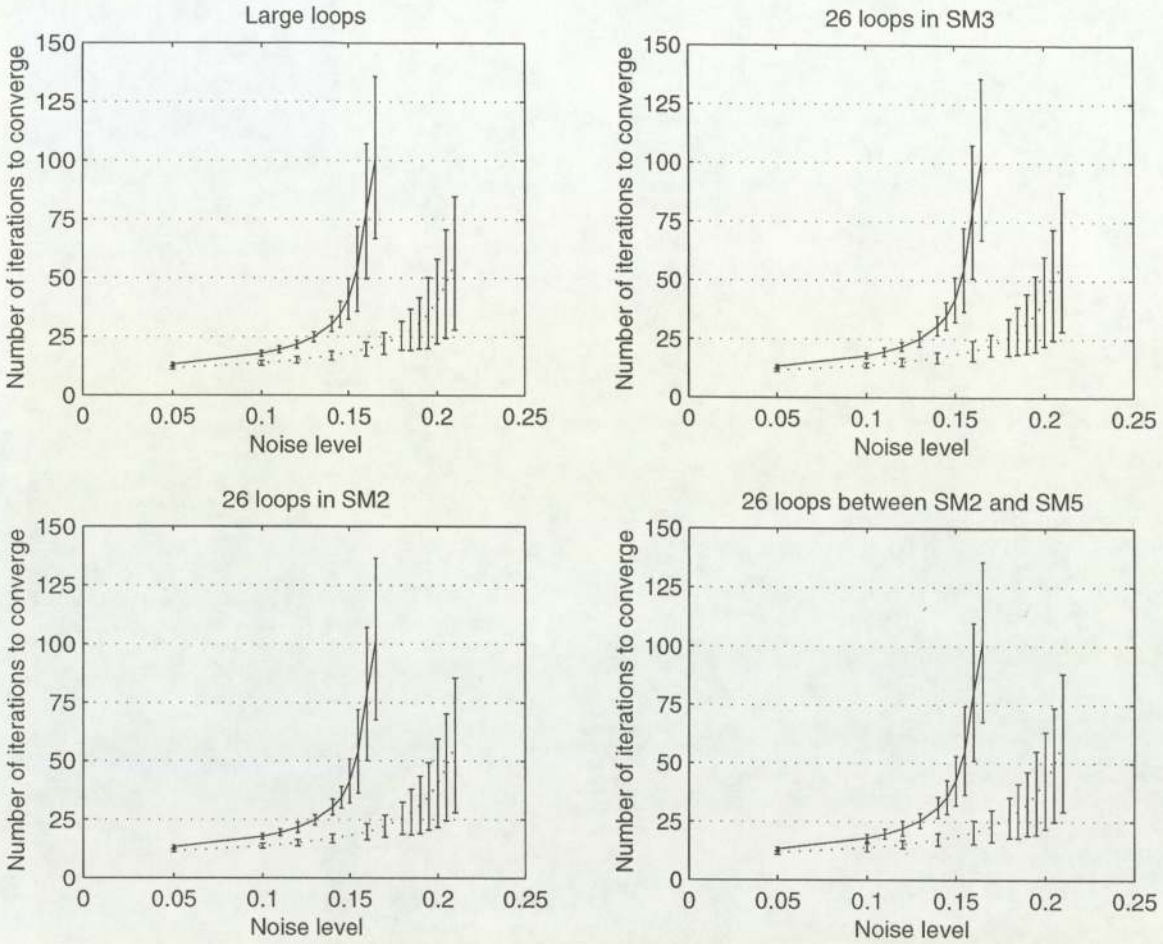


Figure 3.7: Effect on the convergence time of 26 small loops placement for cascading code $R = 1/3$, $N = 1000$ unbiased messages. full line: $f_s = 0.5$. dotted line: $f_s = 0.1$

In the Figure 3.7, we can see that the placement of small loops in themselves has very little influence on the speed of convergence.

3.3.3 The number of small loops.

We artificially build the successive parity check matrix such that SM3 contains an increasing number of small loops and such that the other loops have high length.

Performance

The following simulations use matrices that have 100, 200 and 400 small loops in SM3. We carried out the computations for biased and unbiased messages of length 1000 and $R = 1/3$.

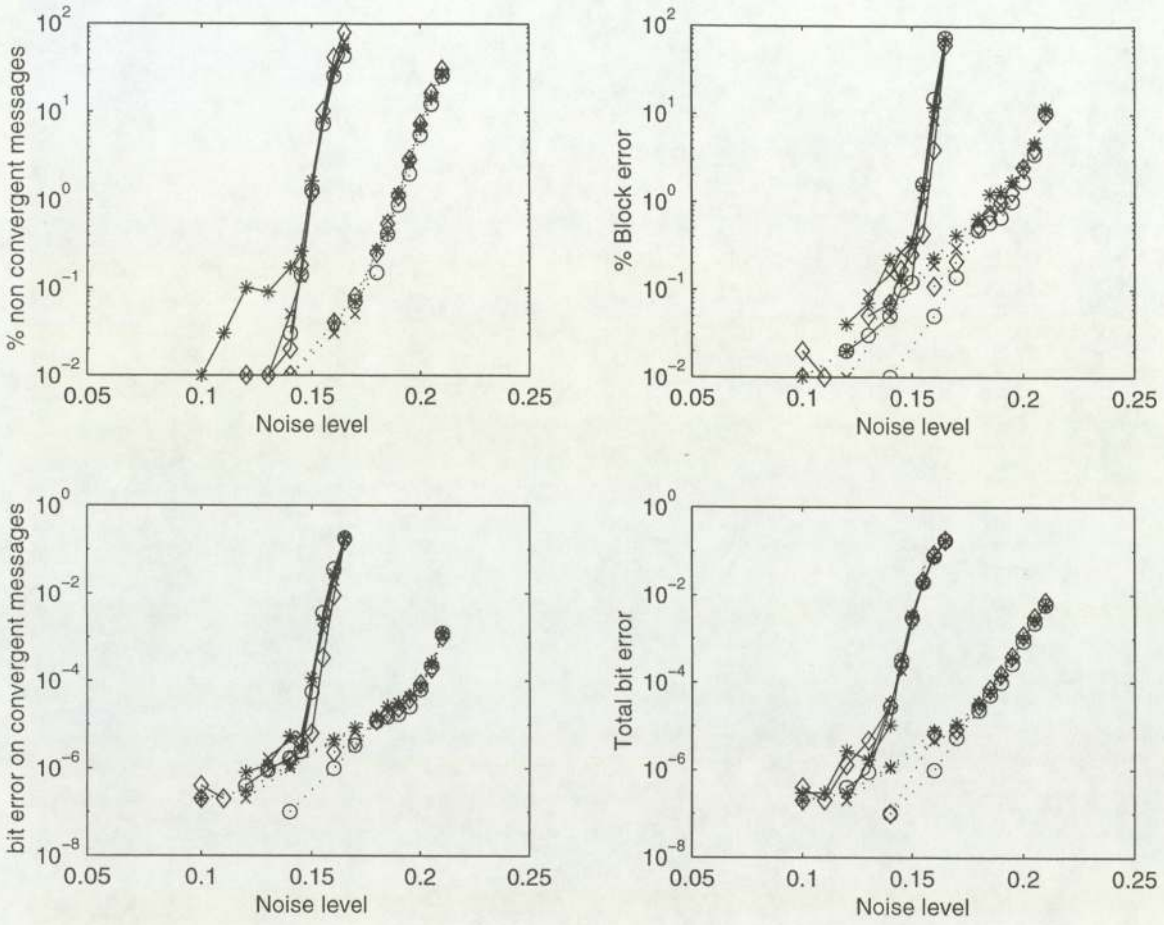


Figure 3.8: Effect of number of small loops in SM3 for cascading code $R = 1/3$, N=1000

- $\circ$: Large loops, $f_s = 0.5$
- $*$: 100 small loops in SM3, $f_s = 0.5$
- $\times$: 200 small loops in SM3, $f_s = 0.5$
- $\Diamond$: 400 small loops in SM3, $f_s = 0.5$

full line: $f_s = 0.5$ dotted line: $f_s = 0.1$

In Figure 3.8, the can see that although the presence of small loops deteriorate the performances at low noise level, there is no clear effect with the increase of their number. In terms of bit and total bit errors standard deviations, in Figure E.3 (Appendix E.3), there is the effect that an increasing number of small loops decrease the variance in bit error and increase the one in total bit error.

In the case of unbiased messages, at higher noise level, the percentage of convergence decrease whereas the block and bit error decrease with the addition of small loops. This is more easily seen in Figure 3.9 where we only include high noise levels.

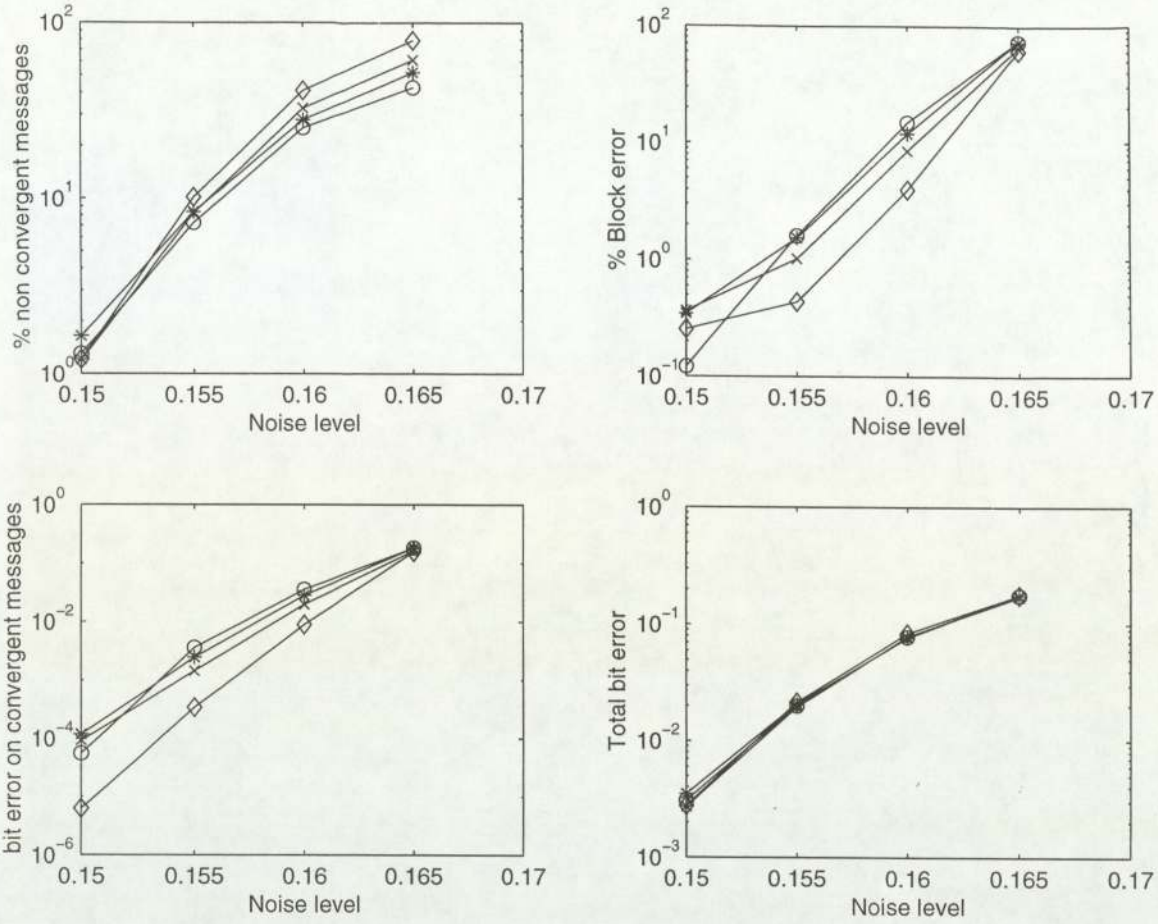


Figure 3.9: Effect of number of small loops in SM3 at high noise level for cascading code $R = 1/3$, N=1000

- $\circ$: Large loops, $f_s = 0.5$
- $*$: 100 small loops in SM3, $f_s = 0.5$
- $\times$: 200 small loops in SM3, $f_s = 0.5$
- $\Diamond$: 400 small loops in SM3, $f_s = 0.5$

full line: $f_s = 0.5$ dotted line: $f_s = 0.1$

In Figure 3.9, we can see the transition between low noise level and high noise level: in terms of bit and block error, the graph without loop used to performs better and around noise level $f_n = 0.155$ the codes with an increasing number of loops performs increasingly better. This is balanced by a decrease in the percentage of converging cases for the codes with loops.

Convergence time

The convergence time curves corresponding to the previous simulations are presented in Figure 3.10.

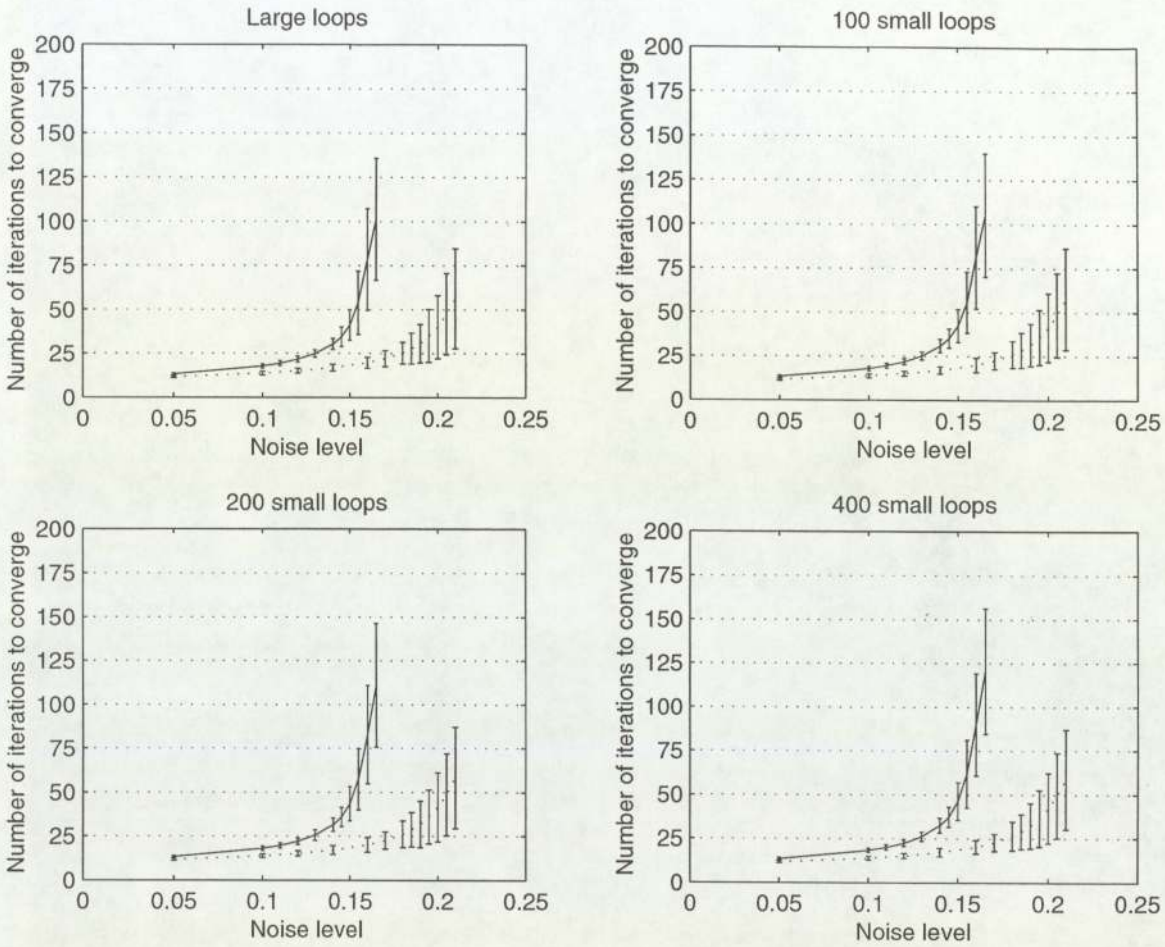


Figure 3.10: Effect on the convergence time of the number of small loops in the last submatrix for cascading code $R = 1/3$, N=1000

From Figure 3.10, we see that the convergence time increases slightly with an increasing number of small loops.

3.4 Interpretation of the results

3.4.1 Removing small loops

The results generally confirm our expectations. A code with large loops improves the performances in all respects at low noise levels: the percentage of converging cases is slightly higher, there is a significant improvement of the block error, bit error and total bit error. There is no significant change in the speed of convergence but it can be due to the relatively small number of small loops present in the original code.

The improvement is not only due to the removal of small loops but also to the fact that we have very large loops. This can be explained by the fact that a message pass less often by the same nodes so that the influence of the loops is reduced. Also, the fact that the length of the loops is of the same order (mainly, loops of length 10 from Appendix F.1) has the consequence that the messages are equally double counted, therefore the belief update approximation is more accurate than with a large range of loop lengths. This idea has been formalized by the notion of unwrapped network. The principle is that when a message comes back to a node it has already visited, we duplicate the network from this node [14]. Thus we trade a loopy network to an infinite polytree where belief update is exact. With a small range of loop length, each node is duplicated an equal amount of time and they have all the same contribution (this is the notion of balanced unwrapped network).

At high noise level, there is no effect from the small loops because as the noise level increase, more and more errors are due to the corruption and the errors due to small loops become irrelevant.

3.4.2 Adding small loops

Placing small loops in SM2 or SM3 has very little influence on the code whether in terms of performance or in terms of convergence speed.

In contrast, placing small loops between SM2 and SM5, leads to a degradation in performance. This is understandable since the part of the loop in SM5 has only a link to a loop to evaluate it pseudo posterior value ($L = 2$). So the probability values of the small loops can only be flipped by the part in SM2 which becomes harder than if all the part of the small loop could be flipped by the contribution of their column as it is the case when we add small loops to SM2 and SM3.

3.4.3 The number of small loops.

In the cases observed, the main influence of increasing the number of small loops is an increase in the bit and block errors variance at low noise level and a decrease in the mean bit and block errors at high noise level in parallel with a lower probability of converging cases.

Therefore, at high noise level, increasing the number of small loops can improve block and bit error. A possible explanation is that while increasing the number of small loops, we increase the correlation between the message estimate bits. Therefore, it is harder to have convergence with a few erroneous bit. This explain the increase in bit error variance and the poorer rate of converging messages. As a side effect, this improve the block and bit errors.

The degradation in the speed of convergence at higher noise level doesn't support the intuition of Section 3.1.2.

Chapter 4

Gallager versus MN configurations

The issue in this chapter is to determine which configuration is more likely to achieve Shannon's bound. In the first section, we highlight the theoretical difference between the two configuration. In the second part, we carry out simulations based on two construction: MacKay's and cascading (see Section 1.3 for the details of the construction). Finally, we try to determine which configuration has achieved the better performance in terms of approaching Shannon's bound.

4.1 The differences between Gallager and MN configuration

Although the two configurations seem similar, the claim [9] is that to achieve Shannon's bound using Gallager's configuration, the number of unit elements per row K should go to infinity. When the actual transition is computed with different K, Gallager configurations goes increasingly closer to Shannon's bound as shown in Figure 4.1 whereas in theory, MN codes need only $K \geq 3$ element per row to saturate Shannon's bound in the case of unbiased messages.

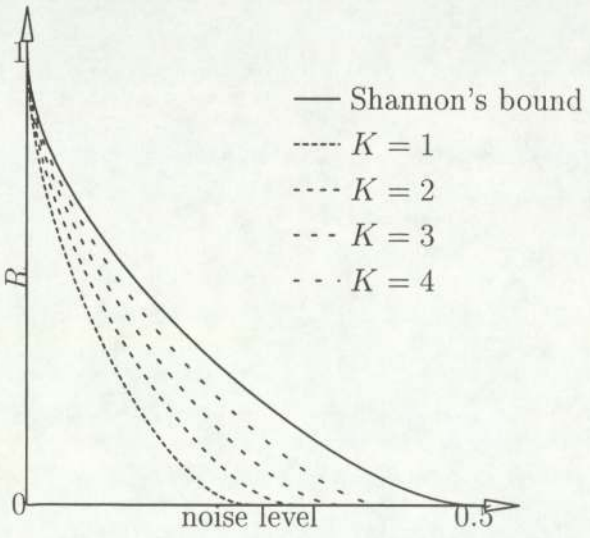


Figure 4.1: Illustration of bounds for Gallager configurations of different K values

In this regard, MN configuration should approach Shannon's bound better than Gallager's configuration.

Basically, the cascading constructions were first designed for MN configurations, however the same construction can be used in the Gallager's configuration. The issue is now if the performances are better when we plug the constructions prepared originally for the MN configuration in a Gallager configuration system as observed by MacKay in his experiments [8].

4.2 Simulations

In order to make a comparison, I have taken a matrix check matrix provided on MacKay's web site. However, if we use the same matrix for MN and Gallager configurations, the rate won't be the same: for a matrix $M \times (N + M)$, the rate for MN is $R = N/M$ and for Gallager is $R = N/(N + M)$ ie. a rate $1/3$ in MN is a rate $1/4$ in Gallager. So, in the simulations, we will measure the distance to Shannon's bound achieved in order to carry out the comparison.

4.2.1 Result for Gallager configuration

Using the two matrix, we carry out the simulation in a Gallager configuration until the limit of the code has been reached.

Performance

The following results are based on unbiased messages of length $N = 3296$ for MN construction and Gallager construction $R \approx 1/4$, over 10000 trials.

We can see in Figure 4.2, that while the MN construction has perfect performances at the beginning, at one point the percentage of convergent message increase and decrease suddenly. At the same time, the block error rises to 100% and stays that way. In contrast, cascading constructions continues farther and eventually presents the same behavior.

The residual block and bit errors for cascading construction are probably due to finite size effects and will resemble a step function with the increase in system size. We can note that all loops of length six or bellow have been removed in the MN construction while we have kept the original construction for the cascading construction, therefore, we can suppose that this effect would disappear with the removal of small loops as done in Chapter 3. We didn't remove the small loops in cascading construction because the goal is not to design an optimal code but to compare the typical performances.

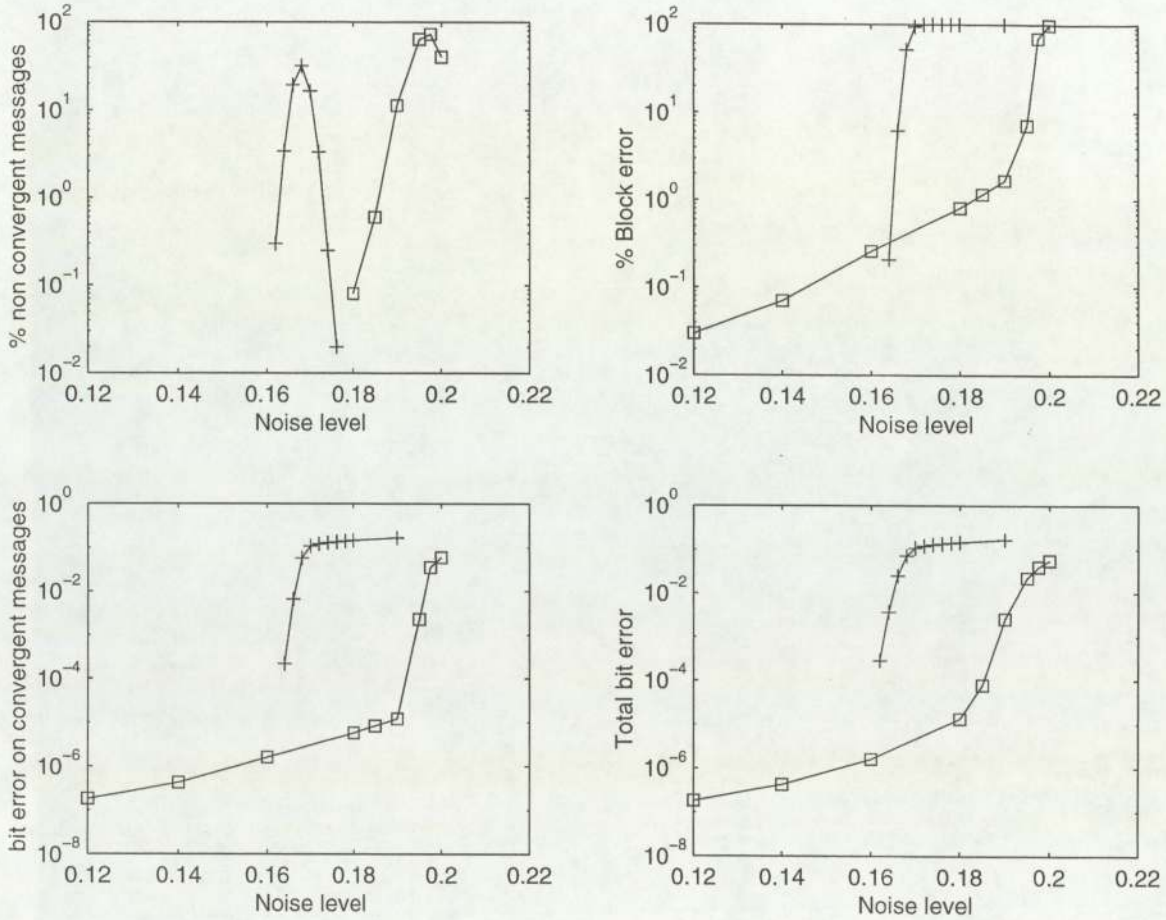


Figure 4.2: Results for Gallager's configuration, source message length=3296, transmitted message length=13298, unbiased messages.
□ : Cascading construction + : MN Construction

Convergence time

The following converging time curves correspond to the previous Gallager configuration simulations.

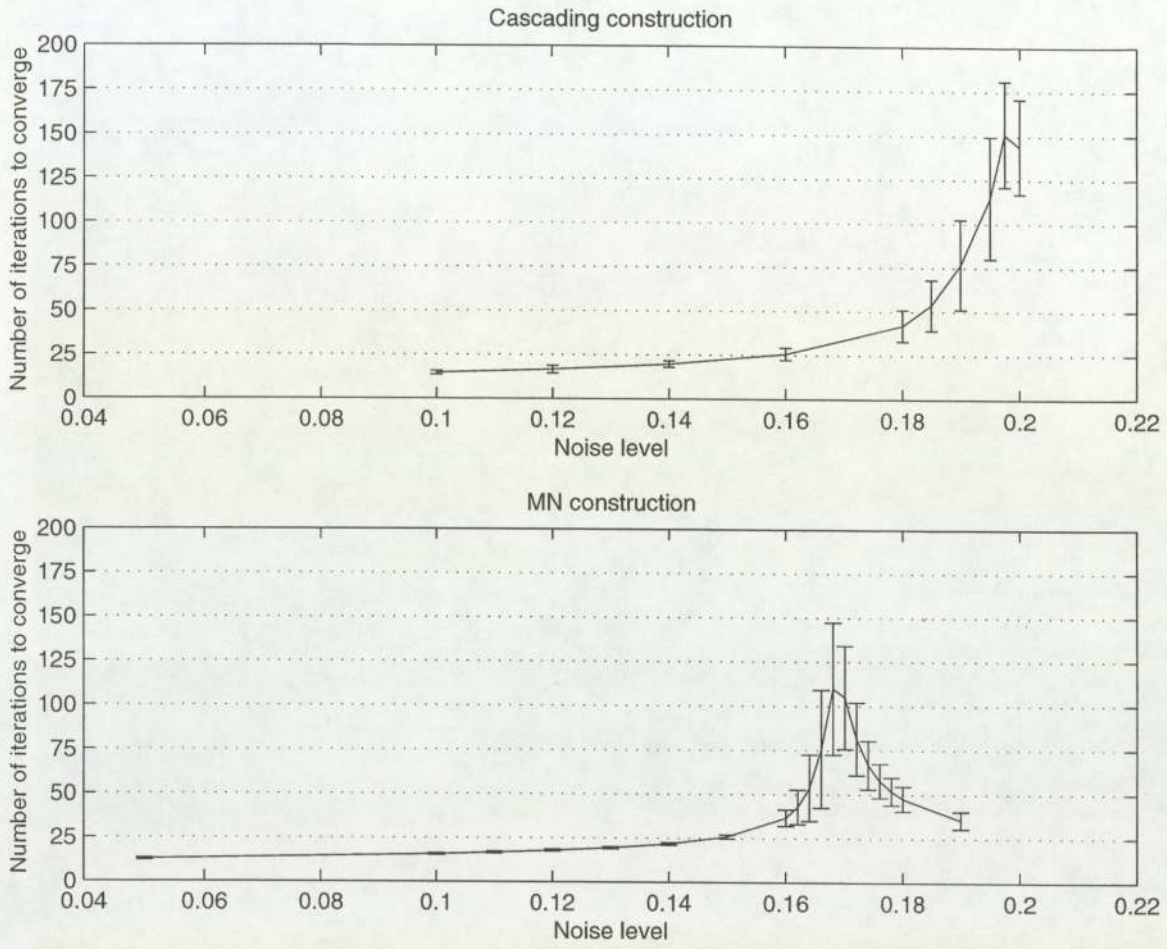


Figure 4.3: Speed of convergence for the Gallager configuration, a comparison between MN and cascading codes.

In terms of the convergence time in Figure 4.3, we show that for the MN construction the number of iterations needed to converge increases up to one point and then decreases when all messages are incorrectly decoded ($BER = 1$). For the cascading construction, we see that 200 iterations are not sufficient to converge at high noise levels. We could expect that the percentage of convergent messages would increase if we increased the number of iteration allowed but it doesn't matter since around half of the messages are decoded correctly at that noise level which is the critical noise level.

4.2.2 Results for MN configuration

We use the same matrices used in the previous Gallager configuration simulations but in MN configuration.

Performance

The following simulators results are based on unbiased messages of length $N = 3296$ for MN and Cascading codes $R \approx 1/3$, over 10000 trials.

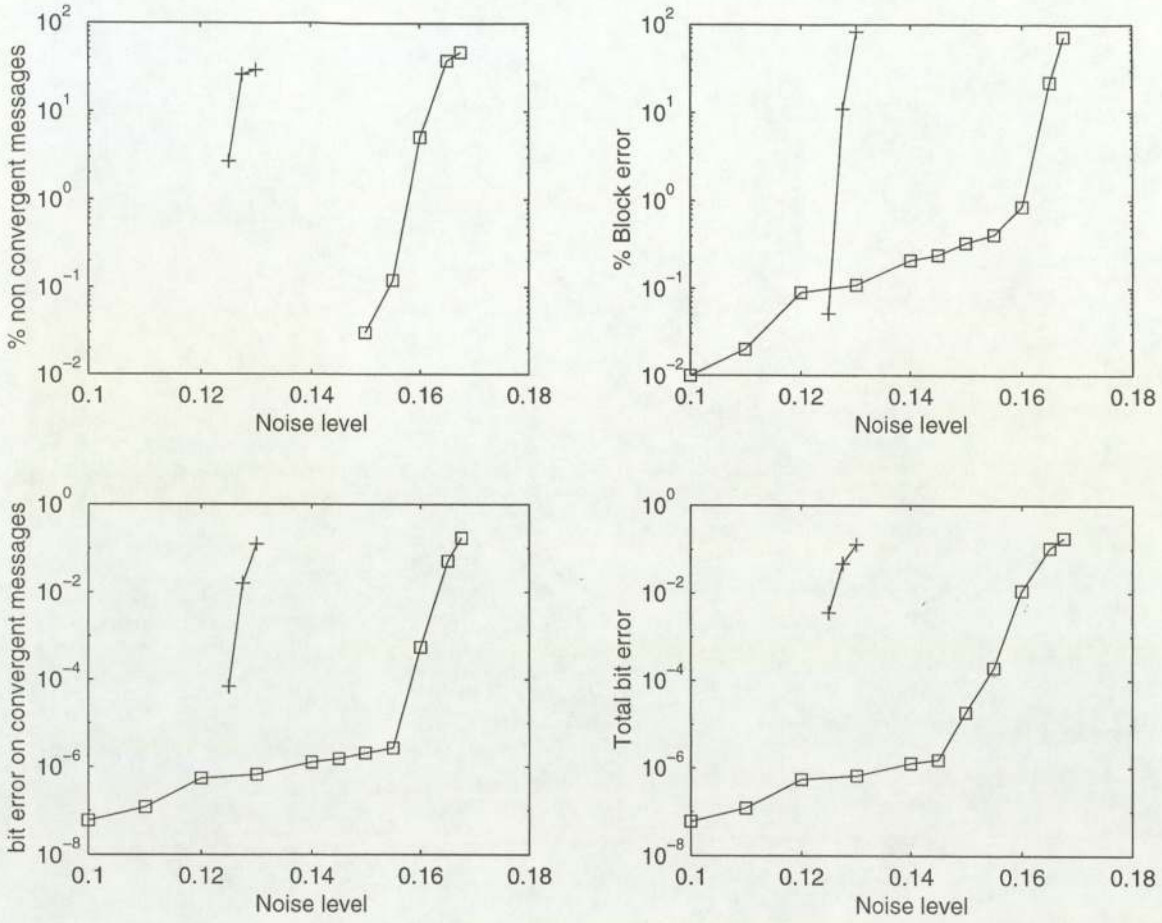


Figure 4.4: Results for MN configuration, source message length=3296, transmitted message length=10002, unbiased messages, $R \approx 1/3$

□ : Cascading construction + : MN Construction

In Figure 4.4, we have approximately the same sub figures as in Gallager configuration except that the decrease in percentage of converging cases doesn't appear.

Convergence time

The following converging time curves correspond to the previous MN configuration simulations.

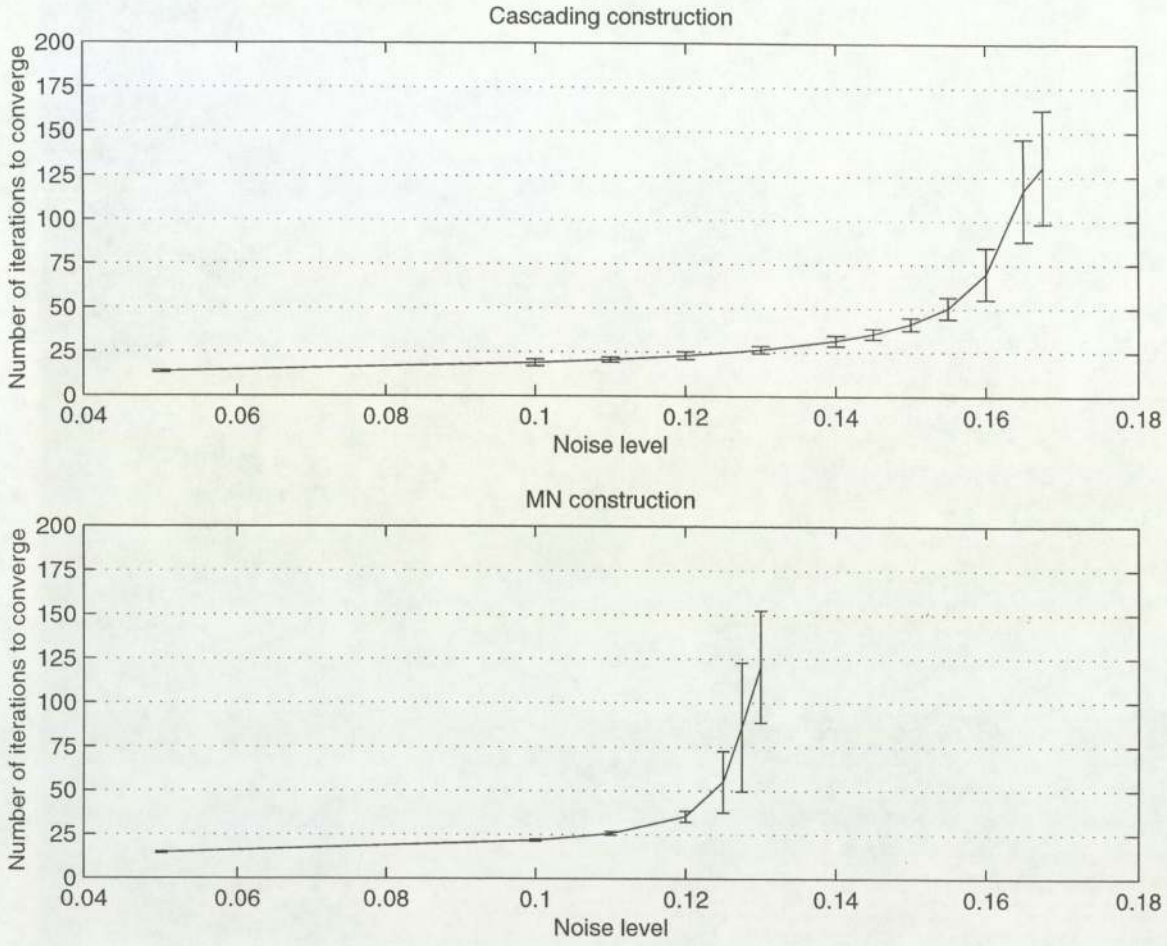


Figure 4.5: Results on the convergence speed for MN configuration

In Figure 4.5, in contrast with the Gallager configuration, the convergence doesn't decrease after a time. This has to be put in relation with the fact that the is no decrease in percentage of converging cases.

4.2.3 Distance to Shannon's bound

In order to compute the distance to Shannon's bound, we have to determine the noise level in which the code reaches its limit.

Choice of criterion for determining the limit of the code

A good criterion to use is the noise level at which the block error equals 50%. If there were no finite size effects or small loops, that degrade the block error rate at small noise levels, the block error curve would have been a step function; the increase in error from

0 to 100% occurring at the limit of the code. Otherwise, the curve takes a smoother shape as illustrated in Figure 4.6.

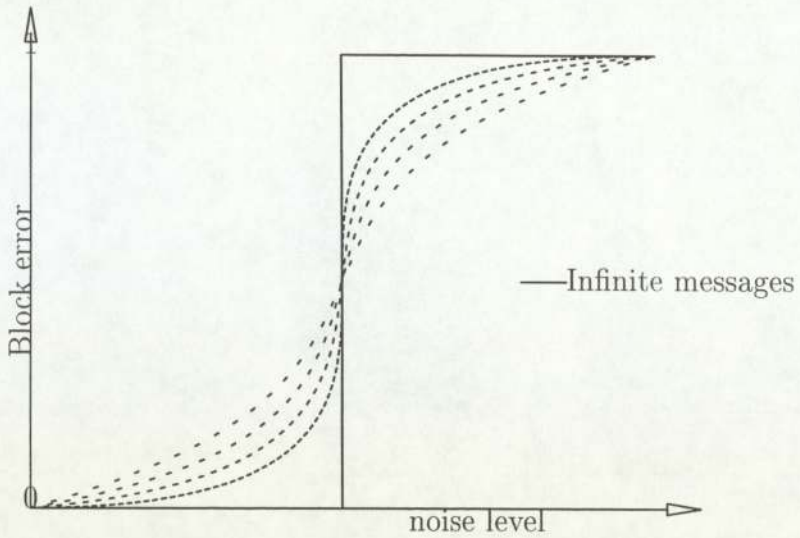


Figure 4.6: Illustration of the finite size effects on the block error. The solid line represents the infinite limit one and the dotted line show the degradation in performance as the message size decrease.

For instance, we can see on the figures 4.4 and 4.2 that the value of block error at 50% correspond to the noise level at which the change in block error, bit error and total bit error is the sharpest and also where the percentage of converging cases reaches it maximum in the Gallager configuration.

Limits of the simulation codes

From the simulations carried out previously, we choose some values near 50% noise level reported table 4.1.

Configuration	Construction	noise level 1	noise level 2	noise level 3
Gallager	MN	$f_n = 0.166$ $BER = 6.18\%$	$f_n = 0.168$ $BER = 51.81\%$	$f_n = 0.17$ $BER = 93.88\%$
	CC	$f_n = 0.195$ $BER = 7.13\%$	$f_n = 0.1975$ $BER = 69.70\%$	$f_n = 0.2$ $BER = 98.67\%$
MN	MN	$f_n = 0.1275$ $BER = 11.04\%$		$f_n = 0.13$ $BER = 83.06\%$
	CC	$f_n = 0.165$ $BER = 21.97\%$		$f_n = 0.1675$ $BER = 72.57\%$

Table 4.1: Values of block error used to interpolate the value of the noise level achieved. noise level 1 and noise level 3 are the values used to interpolate; noise level 2 gives an idea of the value of the noise value f_c at 50% block error when the simulations happened to be near it.

From these values, we interpolate the values of noise level for which a block error of 50% should be reached. We can see from the graphs 4.2 and 4.4 that this points are in sharp increase nearly linear zones thus a linear interpolation is sufficient to determine the noise level achieved f_c (from the values of table 4.1, we can afford a precision at 10^{-4} for the value of f_c).

Distance to Shannon's bound

In order to make the relevant comparisons between the different rates, we will compute the ratio $\frac{H_2(f_c)}{H_2(f_{shannon})}$ ($f_{shannon}$ being Shannon's bound for the corresponding code). This ratio measures the fraction of information corrected by the code relatively to the quantity it corrects at Shannon's bound.

Configuration	Shannon's bound	Construction	f_c	ratio
Gallager	$R = 0.2479$	MN	$f_c = 0.1658$	0.87
	$f_{shannon} = 0.2156$	CC	$f_c = 0.1963$	0.95
MN	$R = 0.3295$	MN	$f_c = 0.1289$	0.83
	$f_{shannon} = 0.1756$	CC	$f_c = 0.1664$	0.97

Table 4.2: Distance to Shannon's bound

We can see that cascading construction can achieve more information correction in MN configuration than in Gallager's. In contrast, MN construction corrects more bit flips in case of Gallager's configuration.

4.3 Interpretation

Gallager configuration

In the case of the MN construction, the sudden rise in percentage of non convergent message in Gallager configuration is due to the fact that we bounded the number of iterations. The decrease in percentage of non convergent messages is easily explained by the appearance of a local minima [10]. We can see that the same effect starts to appear at the highest value of the simulation for the cascading construction.

At one point, a second large local minima appears. The noise level at which the block error is equal to 50% corresponds to the point where the minima of perfect decoding is perfectly balanced by the emerging local.

Distance to Shannon's bound

The MN construction performs better with a Gallager configuration than a MN configuration which is consistent with MacKay's observation. The improvement is on the order of 4% in terms of fraction of information corrected relatively to the quantity it can correct at Shannon's bound.

In contrast, the CC construction performs slightly better (2%) in a MN configuration which is consistent with the theory. We can not conclude on whether Gallager or MN configuration is more likely to saturate Shannon's bound but in the cases studied, it depends on the construction of the matrix chosen.

Chapter 5

Conclusion

In this thesis, we carried out an empirical study on the comparison between belief revision and belief propagation in the specific case of cascading code. We eventually have found that belief revision is unpracticable in the case of unbiased messages and that the similarity of its performance in terms of block error and bit error with belief propagation makes belief propagation a better choice for practical decoding. In the case of biased messages, it may be worthwhile using belief revision to reduce the bit error but at the cost of slower decoding and a poorer percentage of converging cases. We have shown that this is not due to the limitations on the number of iterations but, intuitively, due to the ruggedness of the probability distribution belief revision evaluates.

Concerning the effect of small loops on the code's performances we have the unsurprising result that building a code with large loop lengths improves the performances in all respects and in the case of cascading codes, small loops deteriorate somewhat the bit and block error at low noise but not as much as we would have expected. The most surprising is that the number of small loops, increasing the correlation between the source message estimate, improves the bit and block errors at high noise level at the cost of a poorer probability of convergence. Small loops' contribution to the deterioration of the performance is marginal; even in the case of 400 small loops (ie. 80% of the nodes participate in a small loop), the performances are better than the original code in all respects.

It would be interesting to study the relation between the performances and the heterogeneity in loops rather than simply their number. In the unwrapped network framework, a wide range of loops of different length would create an unbalanced network and some messages contribute more to the posterior than other thus maybe deteriorating the performances.

In the comparison between the Gallager and MN configuration we have results in accordance with the theory ie. MN configuration generally performs better (in this thesis in the cases of Cascading codes) but it is likely to depend on the construction of the matrix: MacKay's construction performs better in the Gallager configuration.

Bibliography

[1] T.M. Cover and Joy A. Thomas. *Elements of Information Theory.* John Wiley & Sons, Inc., N. Y., 1991.

[2] R. G. Gallager. Low density parity check codes. *IRE Trans. Info. Theory*, IT-8:21–28, Jan 1962.

[3] Y. Iba. The Nishimori line and baysesian statistics. *Journal of Physics A*, 32:3875–3888, 1999.

[4] I. Kanter and D. Saad. Error-correcting codes that nearly saturate shannon's bound. *Physical Review Letters*, 83-13:2660–63, 1999.

[5] F. R. Kschischang, B. J. Frey, and H. Loeliger. Factor graphs and the sum-product algorithm. *Submitted to IEEE Transactions on Information Theory, July 1998*, 1998.

[6] A. Shokrollahi M. Luby, M. Mitzenmacher and D. Spielman. Improved LDPCC using irregular graphs and belief propagation. *SRC Technical Note, digital*, 1998.

[7] D. J. C. MacKay. Good error correcting codes based on very sparse matrices. To be submitted to IEEE transactions on Information Theory, 1996.

[8] D.J.C. MacKay. Relationships between sparse graph codes. 2000.

[9] R. Vincente, D. Saad and Y. Kabashima. Error-correcting code on a cactus: a solvable model. *to appear in Europhysics Letters*, 2000.

[10] R. Vincente, D. Saad and Y. Kabashima. Statistical physics of irregular low-density parity-check codes. *to appear in Journal of Physics A*, 2000.

[11] A. Shokrollahi T. Richardson and R. Urbanke. Design of provably good low-density parity-check codes. *Bell Labs, Lucent Technologies*, 1999.

[12] Y. Weiss. Belief propagation and revision in networks with loops. *MIT AI Memo 1616 (CBCL Paper 155).*, 1997.

[13] Y. Weiss. Loopy belief propagation for approximate inference: an empirical study. *Laskey K.B. and Prade H. (editors) Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence, Morgan Kaufmann Publishers, San Francisco*, 1999.

[14] Y. Weiss. Correctness of local probability propagation in graphical models with loops. *Neural Computation*, 12:1–41, 2000.

[15] D. Saad Y. Kabashima, T. Murayama and R. Vicente. The statistical physics of regular low-density parity-check error-correcting codes. *Phys. Rev. Lett.*, 62, 2000.

[16] T. Murayama Y. Kabashima and D. Saad. Cryptographical properties of Ising spin systems. *Phys. Rev. Lett.*, 84, 2000.

Appendix A

Notations used

Symbol	Meaning	Type
CC	Cascading Connectivity code	
MN	MacKay-Neal code	
N	length of source message	Integer
M	length of transmitted message	Integer
R	rate of the code	Real
H_2	Binary entropy function (see Equation (1.2))	Real
K	number of unit element per row in submatrices of CC	Integer
C	number of unit element in submatrices of CC	Integer
L	number of ones per row in the bidiagonal part of CC	Integer
$\mathbf{s}$	source message	Binary vector
$\mathbf{t}$	transmitted message	Binary vector
$\mathbf{r}$	received message	Binary vector
$\mathbf{z}$	syndrome vector	Binary vector
$\hat{\mathbf{s}}$	decoded message	Binary vector
$\mathbf{x}$	any vector ($\mathbf{n}$ in case of Gallager code $[\mathbf{s}\|\mathbf{n}]$ in case of MN code)	Binary vector
p_b	bit error	Real
BER	block error	Real

Appendix B

General definitions

In this thesis, we refer to the messages $\mathbf{s}$, $\mathbf{t}$, $\mathbf{x}$ etc as vectors over the finite field $\{0, 1\}$. A binary variable is termed as a bit.

Weight-density

We call weight $w(\mathbf{x})$ of a vector, the number of unit element present in vector $\mathbf{x}$. If l is the length of the vector, we define the density f_x of the vector $\mathbf{x}$ by:

$$f_x = \frac{w(x)}{l}$$

In the same way, if H is a $M \times N$ binary matrix, the weight $w(H)$ of the matrix is equal to the number of unit elements in the matrix. The density ρ of the matrix H is equal to:

$$\rho = \frac{w(H)}{N \cdot M}$$

Sparseness

A matrix is said to be **very sparse** if the density goes to zero when the dimensions of the matrix go to infinity.

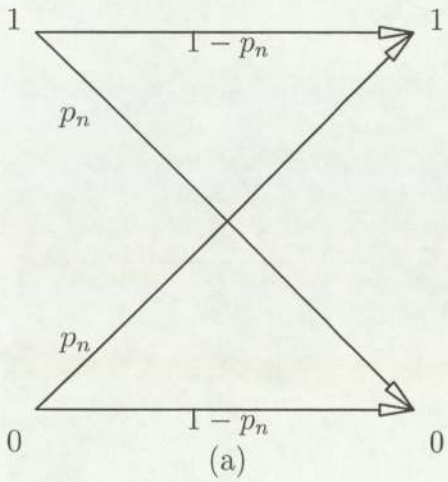
Appendix C

The binary symmetric channel

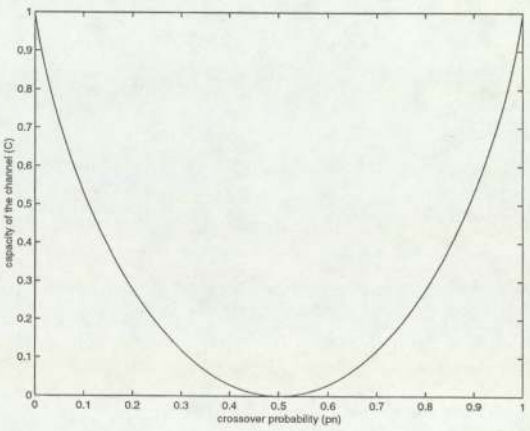
The Binary symmetric channel (BSC) is a symmetric corruption channel. When a vector is transmitted through this channel, each bit of the transmitted vector has a probability p_n of being flipped.

The capacity of the binary symmetric channel is

$$C_{BSC} = 1 + (1 - p_n)log_2(1 - p_n) + p_n log_2(p_n)$$



(a)



(b)

Figure C.1: The binary symmetric channel: (a) probability transition diagram for each bit transmitted through the channel; (b)Capacity of the binary symmetric channel as a function of the noise density

For the simulation, we don't compute the bit flips one by one because it introduces variance in the noise[1], and therefore variance in the results (see Section 2.2.1). Thus we usually refer to the noise density f_n which is equal to the number of bit flipped divided by the length of the message.

[1]being like a Bernoulli process

Appendix D

Algorithms

D.1 Belief update

In algorithm description language, an iteration of the belief update algorithm is:

%%%%%%%%%%
% Horizontal step %
%%%%%%%%%%
for all (i,j) such that $H(i,j)=1$
  %Initialisation of the Markov chain
  $\delta r_{ij} = 1 - z_i$
  %The Markov chain itself
  for all $j' \neq j$ such that $H(i,j')=1$
    if method==BP
      $\delta r_{ij} = \delta r_{ij} \cdot \delta q_{ij'}$
    else %if method==BR
      $\delta r_{ij} = \frac{|\delta r_{ij}+\delta q_{ij'}|-|\delta r_{ij}-\delta q_{ij'}|+2\cdot\delta r_{ij}\cdot\delta q_{ij'}}{2+|\delta r_{ij}-\delta q_{ij'}|+|\delta r_{ij}+\delta q_{ij'}|}$
    end;
  end;
end;

%%%%%%%%%
% Vertical step %
%%%%%%%%%

for all (i,j) such that $H(i,j)=1$
  $q_{ij}^0 = p_j^0 \prod_{i' \in C(j)\setminus i} 1 + \delta r_{i'j}^b$
  $q_{ij}^1 = p_j^1 \prod_{i' \in C(j)\setminus i} 1 - \delta r_{i'j}^b$
  end;
$\delta q_{ij} = \frac{q_{ij}^0 - q_{ij}^1}{q_{ij}^0 + q_{ij}^1}$

for all (i,j) such that $H(i,j)=1$
  $q_j^0 = p_j^0 \prod_{i \in C(j)} 1 + \delta r_{ij}^b$
  $q_j^1 = p_j^1 \prod_{i \in C(j)} 1 - \delta r_{ij}^b$
end;
$\delta q_j = \frac{q_j^0 - q_j^1}{q_j^0 + q_j^1}$

%%%%%%%%%%
% Decoding step %
%%%%%%%%%%

for all j
  if $\delta q_j <= 0$, $\hat{x_j} = 0$
  else $\hat{x_j} = 1$
  end;
end;

D.2 Algorithm to compute the distance between two nodes

In matlab/algorithm description language, the algorithm is:

```
function distance=computeDistance(graph,node_begin,node_end)

% graph is a representation of the graph corresponding
%       to the parity check matrix
% node_begin and node_end identify the two nodes
%            we want the distance between
% distance return the distance between node_begin and node_end or
%                 Inf if there is no path between them

%%%%%%%%%%%%%%%%%
% Initialisation %
%%%%%%%%%%%%%%%%%
distance=0;
mark(node_begin);          % The function 'mark' marks the parameter node
level=[];                  % Nodes at distance 'distance-1' of node_begin
nextlevel=[node_begin];    % Nodes at distance 'distance' of node_begin

% While node_end is not reached and the is still nodes
while (node_end not in nextlevel) and (nextlevel~=[])
    distance=distance+1;
    level=nextlevel;
    nextlevel=[];
    for all nodes n in graph connected to the nodes in level
        if not marked(n)
            mark(n);
            nextlevel=[nextlevel n];
        end;
    end;
end;

% Trap the case 'no path between node_begin and node_end'
if node_end not in nextlevel
  distance=Inf;
end;
```

If we want the distance of the node `xdebut` to all other nodes, we just mark them with the distance until `nextlevel` is empty. The node not marked are the nodes for which there is no path between the two and they are at distance infinity.

D.3 Approximation of the number of loops

```
function loop=computeLoop(graph)

% graph is a representation of the graph corresponding
%       to the parity check matrix
loop=infinite array of 0;   %will be used to sum the
                            %contributions of the links
                            %to the various loop lengths
for all nodes z in graph
  for all nodes x to which z is connected
    remove the links from z to all other nodes except x in graph;
    dz=compute distance from z to nodes formerly connected to z;
    loop(dz+1)=loop(dz+1)+1;
  end;
end;

%We divide by the length of loops to retrieve the number of loop
for all index i of loop such that loop(i)~=0
  loop(i)=loop(i)/i;
end;
```

Appendix E

Simulation results

E.1 Standard deviation for belief revision vs. belief propagation

I included this graph, because intuitively, we can think that belief revision also decrease the variance in bit error since it strives for a perfect decoding.

Figure E.1 present the bit and total bit errors standard deviation of the simulation carried out in Section 2.3 for the comparison between belief revision and belief propagation. For reminder, the simulations results are based on 10000 trials of biased and unbiased messages of length $N = 1000$, using cascading code.

We can see on Figure E.1 that belief revision a a smaller bit error standard deviation in the case of biased messages and an equivalent one in the case of unbiased messages. In terms of bit error's standard deviation belief revision performs as well as belief propagation for biased messages but worse for unbiased messages.

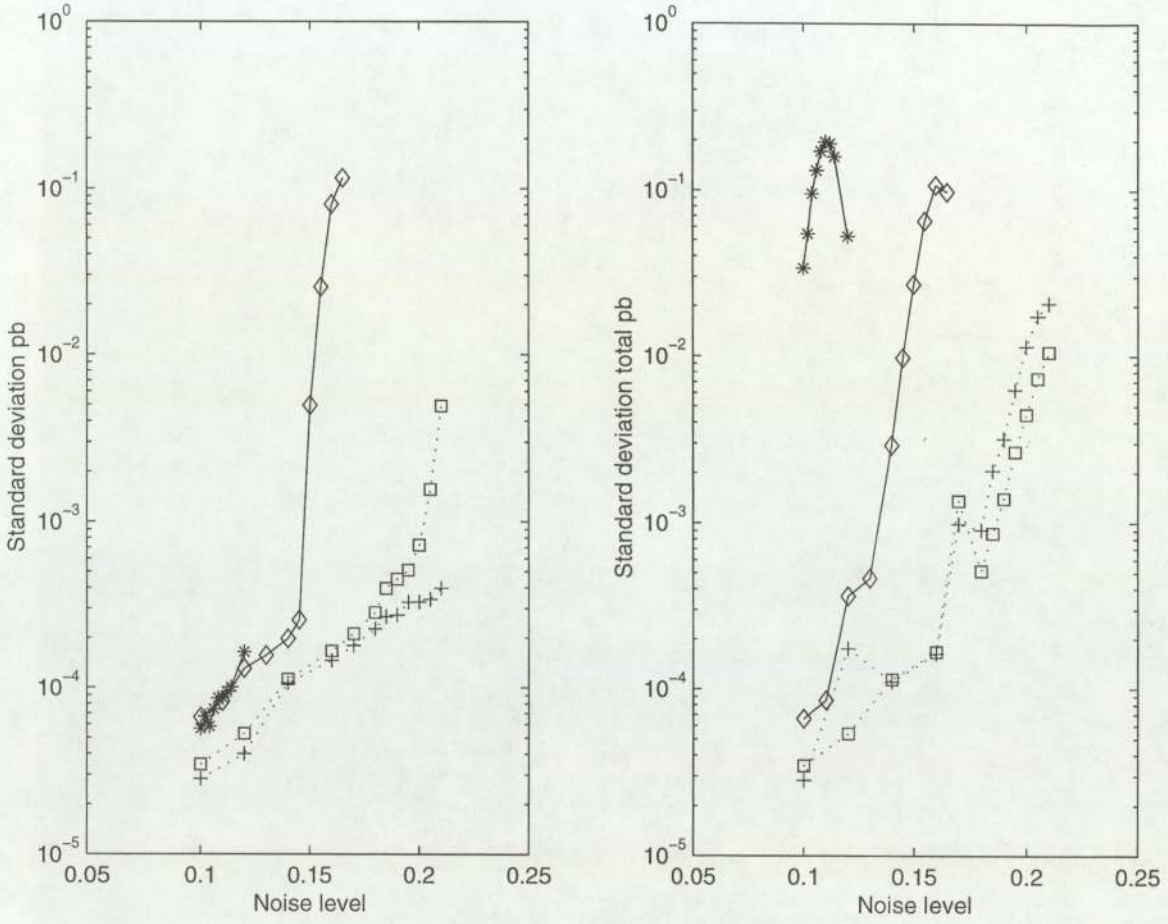


Figure E.1: Standard deviation results of comparison between belief propagation and belief revision for cascading code of $R = 1/3$, $N = 1000$

$\Diamond$: Belief propagation, $f_s = 0.5$ $*$: Belief revision, $f_s = 0.5$

$\square$: Belief propagation, $f_s = 0.1$ $+$: Belief revision, $f_s = 0.1$

E.2 Standard deviation for the removal of small loops

Figure E.2 present the bit and total bit errors standard deviation of the simulation carried out in Section 3.3.1 for the comparison between a loopy graph and a graph with large loops. For reminder, the simulations results are based on 10000 trials of biased and unbiased messages of length $N = 1000$, using cascading code.

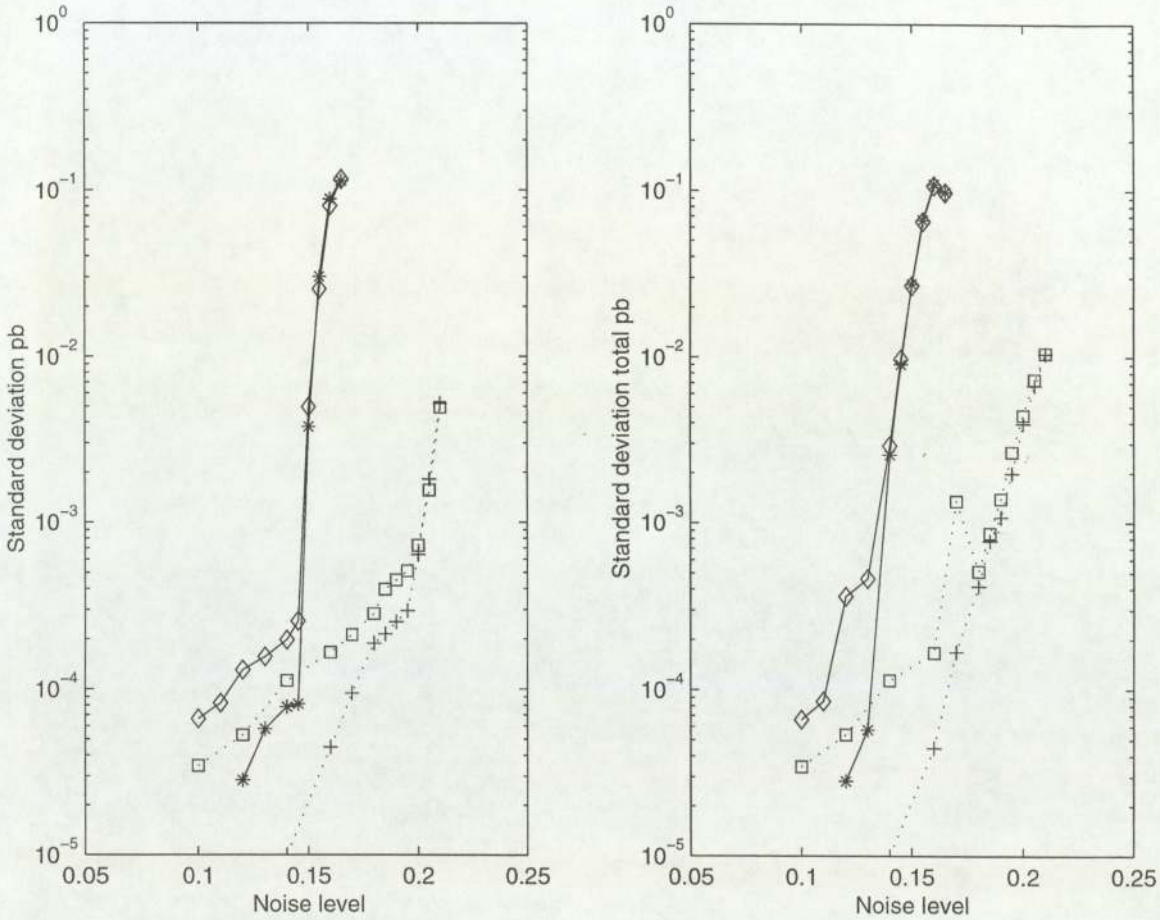


Figure E.2: Impact of 26 small loops on the bit and total bit errors standard deviations of cascading code $R = 1/3$, $N = 1000$

$\Diamond$: Original matrix, $f_s = 0.5$ $*$: Large loops, $f_s = 0.5$
$\square$: Original matrix, $f_s = 0.1$ $+$: Large loops, $f_s = 0.1$
full line: $f_s = 0.5$ dotted line: $f_s = 0.1$

In terms of bit and total bir errors standard deviations, in Figure E.2, removing small loops reduce the variance of both bit and total bit errors

E.3 Standard deviation for the removal of small loops

Figure E.3 present the bit and total bit errors standard deviation of the simulation carried out in Section 3.3.3 for the effect of the number of small loops study. For reminder, the simulations results are based on 10000 trials of biased and unbiased messages of length $N = 1000$, using cascading code.

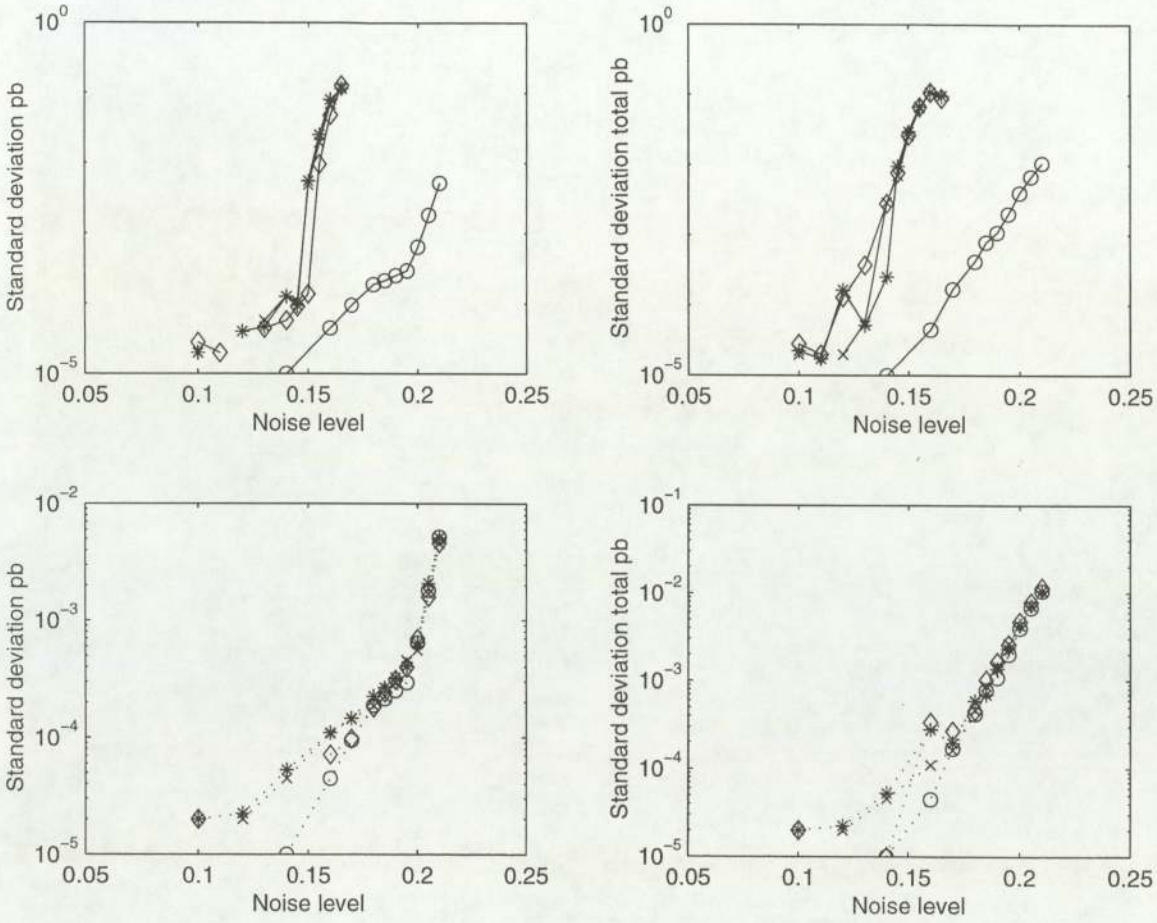


Figure E.3: Effect of number of small loops in SM3 on bit and total bit errors standard deviation for cascading code $R = 1/3$, N=1000

- $\circ$: Large loops, $f_s = 0.5$
- $*$: 100 small loops in SM3, $f_s = 0.5$
- $\times$: 200 small loops in SM3, $f_s = 0.5$
- $\Diamond$: 400 small loops in SM3, $f_s = 0.5$
- full line: $f_s = 0.5$
- dotted line: $f_s = 0.1$

In terms of bit and total bir errors standard deviations, in Figure E.3, an increasing number of small loops decrease the variance in bit error and increase the one in total bit error.

Appendix F

Properties of matrices used

F.1 Cascading code Rate 1/3, length 1000

In order to check the implementation of the parity check matrix construction, I compute the number of unit element per row K and per column C in the submatrix A of $H = [A|B]$ generated.

The figures in Table F.1 were computed by a software in order to verify the conformity of the actual construction of the matrices generated.

submatrix	column 1 - 1000	column 1001 - 4000	Entire submatrix
row 1 - 1000	100% K=1 100% C=1	100% L=2	100% $K+L$=3
row 1001 - 1750	100% K=3 75% C=2 25% C=3	100% L=2	100% $K+L$=5
row 1751 - 3000	100% K=3 25% C=3 75% C=4	100% L=1	100% $K+L$=4
Entire submatrix	33.33% K=1 66.66% K=3 100% C=7	41.66% L=1 58.33% L=2	

Table F.1: Properties of cascading code $R = 1/3$, message length $N = 1000$.

The original matrix is the matrix generated with the only constraint being that it should respect the construction of cascading code. This properties are kept when we modify the loops properties in Table F.2.

	4	6	8	10	12	14	total
Original construction	26	231	1274	1390	221	18	3160
Without small loops	0	0	15	2451	305	14	2785
With 26 small loops in SM2	26	1	8	2427	316	18	2796
With 26 small loops in SM3	26	0	34	2381	339	17	2797
With 26 small loops in SM2 and SM5	26	0	17	2424	313	14	2794
100 small loops in SM3	100	1	21	2352	344	20	2838
200 small loops in SM3	200	0	33	2263	378	19	2893
400 small loops in last submatrix	400	0	12	2163	429	29	2993

Table F.2: Loop information for the different constructions $N = 1000$, $R = 1/3$

F.2 Codes $N = 3296$, $M = 10002$, $R \approx 1/3$

The matrix corresponding to MacKay's construction comes from MacKay's site : `http://131.1`

The Cascading construction matrix has beeen generated without removing the small loops.

Loop length	4	6	8	10	12	14	total
MacKay's construction	0	0	643	2619	9116	4216	16597
Cascading construction	37	248	1899	5146	1938	261	9529

Table F.3: Loop information for the different constructions $N = 3296$, $R \approx 1/3$