CONTROLLED GIRTH, STRUCTURED LDPC CODES

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree

of

MASTER OF TECHNOLOGY

in

ELECTRONICS AND COMMUNICATION ENGINEERING

(With Specialization in Communication Systems)

By

VINAY BHARDWAJ

G 14285 L

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE - 247 667 (INDIA)

JUNE, 2008

CANDIDATE'S DECLARATION

I hereby declare that the work, which is presented in this dissertation report entitled, " **CONTROLLED GIRTH, STRUCTURED LDPC CODE** " towards the partial fulfillment of the requirements for the award of the degree of **Master of Technology** with specialization in **Communication Systems**, submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, Roorkee (India) is an authentic record of my own work carried out during the period from June 2007 to June 2008, under the guidance of **Dr. Arun Kumar, Professor and Dr Nagendra P Pathak, *Assistant Professor, Department of Electronics and* Computer Engineering, Indian Institute of Technology Roorkee.**

I have not submitted the matter embodied in this dissertation for the award of any other Degree or Diploma.

Date: 27 Jun 08

Place: Roorkee

[illegible]

VINAY BHARDWAJ

CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of my knowledge and belief.

Date: 27 Jun 08

Place: Roorkee

NPpathak 27/6/08

(Dr Nagendra P Pathak)
Asst Professor
Department of E & CE
Indian Institute of Technology
Roorkee-247667

[illegible]

(Dr Arun Kumar)
Asst Professor
Department of E & CE
Indian Institute of Technology
Roorkee-247667

ACKNOWLEDGEMENTS

This dissertation would not have existed without the help, support and love of many people. The foremost one is from my guide , **Dr Arun Kumar**. His endless support encouraged me throughout the course of post graduate studies. I would like to express my deepest appreciation and gratitude to him for his guidance and support during my studies.

I wish to thank my co-guide, **Dr Nagendra P Pathak**. His constant support, encouragement and comments helped me so much in order to complete this thesis. The fruitful discussions with him gave me the insightful understanding of fundamental problems of communication engineering.

I would like to also thank to **Dr D.K. Mehra,** Professor and HOD, Department of E & CE, IIT Roorkee, for his blessings and guidance.

My sincere thanks are due to **Dr Jin Lu,** Sun Microsystems , Louisville, USA and **Dr Jose M.F. Moura** Professor, Department of Electrical and computer engineering, Carnegie Mellon university, USA for their timely replies to my queries.

Last but not the least, my deepest love and gratitude is devoted to my family, especially my wife Dr Chetna Bhardwaj and my little daughter Trijya. This is all for them.

VINAY BHARDWAJ

ABSTRACT

Low Density Parity Check (LDPC) codes have stimulated a lot of interest in recent years due to their capacity approaching performance and availability of fast decoding algorithms. LDPC codes can be represented by bipartite graphs known as Tanner graphs. The girth of the graph is the length of the shortest cycle of the graph. It is well known that iterative decoding on LDPC codes performs well when the underlying Tanner graph has high girth. Therefore the design of LDPC codes having high girth is desirable. An elementary Turbo Structured Tanner graph construction for a family of regular LDPC codes attaining desired high girth is designed and obtained in this dissertation. The design studied in the dissertation provides flexibility in choice of parameters like column weight ($j \geq 2$), row weight k and arbitrary girth g.

Turbo Structured low density parity check (TS-LDPC) codes are regular codes whose Tanner graph is composed of two trees connected by an interleaver. TS-LDPC codes with good girth properties are easy to construct. Careful design of the interleaver component prevents short cycles of any desired length in its Tanner graph. Simulation results demonstrates that the bit error rate (BER) performance at low signal to noise ratio (SNR) is competitive with error performance of Random LDPC codes of same size, with better error performance at high SNR.

The issue of complexity of construction of codes is also addressed in the dissertation. Here a linear encoding algorithm for a type of Turbo structured LDPC codes. This encoding can be developed with few restrictions and alterations in Tanner graph of TS LDPC codes.

CONTENTS

Candidate's Declaration and Certificate			i
Acknowledgements			ii
Abstract			iii
Table of Contents			iv
CHAPTER 1		**Introduction And Outline Of Dissertation**	1
	1.1	Digital Communication Systems	1
	1.2	Dissertation Motivation	2
	1.3	Statement of the Problem	5
	1.4	Dissertation Outline	5
CHAPTER 2		**Theory Of LDPC Codes**	7
	2.1	Background	7
	2.2	Definition	7
	2.3	Tanner Graph	8
	2.4	Regular and irregular LDPC codes	9
	2.5	Decoding of LDPC Codes	9
	2.5.1	Sum product algorithm	10
	2.6	Cycles, Girth and Performance	15

CHAPTER 3		**Turbo Structured LDPC Codes**	17
	3.1	Introduction	17
	3.2	Turbo Design of LDPC codes	18
	3.3	LDPC codes using turbo designs	19
	3.4	Interleaver Design	21
	3.4.1	(p–q)-Alternate-Decimal Format	22
	3.4.2	Avoiding Short Type I Cycles—Digit-Wise Reversal	23
	3.4.3	Avoiding Type II Cycles—Grouping and Shifting	25
	3.4.3a	Connection rule to avoid type II cycles	26
	3.4.5	Minimum Number of Groups in Each Tree	29
	3.5	Construction of TS LDPC codes	31
	3.5.1	Upper tree (T_U)	32
	3.5.2	Lower tree (T_L)	32
	3.5.3	Interleaver	33
	3.5.4	Shift Matrix S	33
	3.5.5	Algorithm for Shift Matrix S	33
	3.5.6	Parity check matrix H	36
	3.6	Simulation and Discussion	37
	3.6.1	TS-LDPC codes with column weight 3, rate 2/3 and girth g=8	38
	3.6.2	(6084,3,12) TS-LDPC codes girth g=8	40
	3.6.3	(6666,3,6) TS-LDPC codes girth g=10	41
	3.6.4	(6220,3,4) TS-LDPC codes girth g=12	42
	3.6.5	(1446,4,6) TS-LDPC codes girth g=8	43
	3.7	Efficient Memory Utilisation	44

	3.8	Conclusions	44
CHAPTER 4		**Efficient Encoding For TS-LDPC Codes**	45
	4.1	Introduction	45
	4.2	Encoding friendly TS-LDPC (EFTS-LDPC) Codes	45
	4.3	Linear-Complexity Encoding of TS-LDPC codes	46
	4.4	Encoding algorithm for EFTS-LDPC Codes	49
	4.4.1	EFTS-LDPC Codes Encoding: Example	50
	4.5	Computational complexity	52
	4.6	Simulation and discussion	52
	4.6.1	(8051,3,6) EF TS-LDPC codes girth g=10	53
	4.6.2	(1993, 3, 8) TS-LDPC codes girth g= 8	54
	4.7	Conclusions	55
CHAPTER 5		**Conclusion And Future Work**	56
GLOSSARY			57
REFERENCES			60
APPENDIX		**Matlab Source code**	I

1.1 Digital Communication Systems

The fundamental problem of communications is to reproduce at one point a digital message selected at another point as exactly as possible [1]. To solve this fundamental problem, communication engineers have designed sophisticated systems to transmit messages over hostile noisy channels. The general block diagram of a digital communication system is illustrated in Figure 1.1.

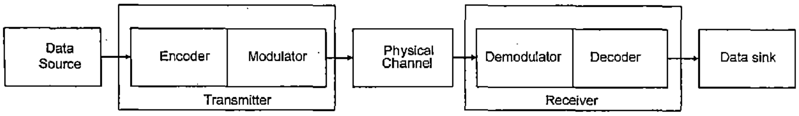


Fig 1.1 Block diagram of a digital communication system

At the transmitter, digital messages are processed before they are sent to the physical channel. The objective of this process is twofold: (i) to choose proper signal waveforms to avoid bad effects of the physical channel and (ii) to be able to detect these waveforms easily at the receiver end. At the receiver, a computational algorithm is needed to detect the transmitted waveforms as precisely as possible with a practical complexity.

In digital communication systems, the modulator is the block that maps the information to the physical channel. Due to the continuous nature of most physical channels, the modulator needs to transform discrete waveforms to continuous waveforms that adapt to the channel.

The demodulator processes the received continuous waveforms that are corrupted by random factors of the physical channel. Usually, the demodulator tries to replace the received continuous waveforms by finite-dimensional vectors to enable the calculation of the decision variable based on the joint density functions of random variables. In practice, matched filters are often used in digital communication systems to transform continuous waveforms to sufficient statistics.

The coding process in digital communications approaches the problem digitally or in the discrete time domain. The encoder can be divided into two blocks, namely the source encoder and the channel encoder. In this dissertation, we are only concerned with the channel encoder and shall refer to it simply as the encoder. The encoder also tries to create waveforms to transmit effectively the information message against random factors of the channel as in the modulator block, but this is done in the discrete domain. The encoder implements this task by inserting redundant information in the message in a controlled manner. At the receiver, the decoder recovers the original information from the discrete outputs of the demodulator with the help of this redundant information.

1.2 Dissertation Motivation

In his seminal paper "A Mathematical Theory of Communication", Shannon [1] defined the capacity of a communication channel which predicted the rates at which transmission systems can transmit and receive information reliably over a noisy channel. Shannon suggested that the capacity is achievable with good channel codes. A channel code is good in a sense that the decoders of the code at rate slightly smaller than the channel capacity is error-free asymptotically. In his proof, Shannon used random coding to achieve the capacity. However, from an engineering perspective, random coding is too complex to implement because random codes lack structures. For fifty years, researchers have been searching for practical capacity-achieving error-correcting codes. Recently, with a reasonable complexity, LDPC, Turbo and related codes have been shown to perform only several tenths of a dB away from the capacity.

The main idea of error-correcting codes is to add redundancy that is correlated to the information to be transmitted so that the receiver can exploit the correlation between the information bits and the redundancy bits and then correct or detect errors caused by channels.

There are two major classes of codes, namely, block codes and convolutional codes. Examples of block codes are Hamming codes, Bose- Chaudhuri -Hocquenghem (BCH) codes, Reed-Solomon (RS) codes [2] and newly rediscovered LDPC codes. Block codes like Hamming, BCH and RS codes have nice mathematical structures. However, there is a limitation when it comes to code lengths. A bounded-distance decoding algorithm is usually employed in decoding block codes and, except LDPC codes[3,4], it is generally hard to use soft decision decoding for block codes.

Convolutional codes are represented with finite state machines, in which going from a start state to a next state is called a state transition. A sequence of state transitions is called trellis path and a sequence of allowed state transitions constitutes a valid trellis path. The decoder can output the most probable valid trellis in the maximum likelihood (ML) or maximum *a posteriori* (MAP) sense based on received signal. The decoding algorithms can be based on hard-decision data or soft-decision data and produce soft output. Soft-output decoding algorithms are especially useful in iterative decoding, in which a concatenated coding system is used and the soft output of the decoder of a component code can be further processed by the decoder of the other component code without loss of information due to quantization.

Both codes provide good coding gain but there is still a gap from capacity. Recently, there has been a great improvement in coding techniques towards achieving this limit. The new approach uses pseudo-random codes and suboptimum decoders, called iterative decoders, instead of the optimum one. Due to this iterative decoding technique, the performance of coded systems is significantly improved due to the ability to increase the code length and, simultaneously, still keep a reasonable computational complexity for the receivers. In 1993, Turbo code was invented by Berrou, Glavieux and Thitimajshima [5] . This awkward code is constructed by parallel concatenation of two convolutional codes through a random interleaver. Iterative decoding is applied in the decoder where the soft output is exchanged between the two component decoders. The performance of original Turbo code approaches the Shannon limit within 0.5 dB with this iterative decoding technique. In wake of Turbo codes, Gallager's low density parity check (LDPC) codes were rediscovered and it was shown that with long block-length codes they also achieve near Shannon limit performance [6]. Low density parity check codes are linear block codes with *sparse* parity check matrices generated randomly and these codes can also be decoded iteratively.

The performance of Turbo codes and LDPC codes with a very long block length depends on the convergence of the iterative decoding algorithms. So far, the best known error control code over additive white Gaussian noise (AWGN) channels is an irregular LDPC code in whose empirical performance achieves the bit error rate (BER) of 10^{-6} within 0.04 dB of Shannon limit with a block length of 10^{7} [7]. For LDPC codes with short and medium block lengths, the error performance of a LDPC coded system depends on both the convergence property and the Hamming distance spectrum of the code. The convergence property determines how the performance of the iterative decoder can approach that of the maximum likelihood decoder. On the other hand, the Hamming distance spectrum of the codes determines the performance of the ML receiver.

Compared to Turbo codes, LDPC codes are more flexible in construction in terms of the code rate and other parameters. Moreover, there is an error floor on the error performance of Turbo codes due to the poor Hamming distance spectra of these codes. On the other hand, the error performance of LDPC codes does not clearly show an error floor because of the good Hamming distance property of the codes. This fact makes finite-length LDPC codes a good candidate for applications that require very low bit error rate (BER), and simultaneously, low delay. Another advantage of LDPC codes is the ability to implement fully parallel decoders thanks to the mechanism of their decoding algorithm. The parallel iterative decoding algorithms of LDPC codes are easily implemented in VLSI [8]. Due to these advantages, LDPC codes are proposed for most future data applications such as wireless, wire line communications and storage systems. Low-density parity-check (LDPC) codes, [3,4], are being considered in numerous applications including digital communication systems and magnetic recording channels. Their bit-error rate (BER) performance using iterative decoding is close to the Shannon limit, [1].

There are several ways to specify LDPC codes. In this dissertation, the code parity-check matrix H and its associated parameters are specified by bipartite Tanner graph, [9]. Kou, Lin, and Fossorier [10], [11] developed LDPC codes based on finite geometries and incidence structures. Finite- geometry LDPC codes can be designed over a wide variety of block lengths and code rates and achieves good minimum distances. Finite-geometry LDPC codes have girth 6. Another method to construct structured

4-cycle-free regular LDPC codes is based on balanced incomplete block designs (BIBD) [12]–[13]. A BIBD is defined as a collection of equal size blocks, comprising elements drawn from a set V, such that each pair of distinct elements (x, y) of V occurs in exactly λ blocks of B . BIBD-based codes are well structured, free of 4-cycles, i.e., with girth g=6 , but exhibit a large number of 6-cycles in their Tanner graphs. Finite-geometry codes and BIBD codes are examples of cyclic and quasi-cyclic [14] codes. For these codes with column weight $j \geq 3$ the girth is less than or equal to 12 [15]. Hu, Eleftheriou, and Arnold propose in [16] a non algebraic method named progressive edge-growth (PEG). They present examples of codes of girth g=8 by progressively establishing edges between bit and check nodes in an edge-by-edge manner. PEG optimizes the placement of a new edge on the Tanner graph with the goal of maximizing the local girth. Reference [16] constructed LDPC codes from circulant permutation matrices. It presents conditions for such codes to achieve girth up to $g \leq 12$.

Motivated by the successes and potential of LDPC codes and the technique of iterative decoding, this dissertation studies LDPC codes whose tanner graph [8] of the associated parity check matrix **H** exhibit a specific architecture that is stimulated by Turbo codes. This turbo structured LDPC code can have desired girth and designed easily with parameters like number of ones in a row and column.

1.3 *Statement of the problem*

To design structured LDPC codes for controlled and large girth with desired column and row weights (or code rate).

To develop a linear complexity encoding method for turbo structured LDPC codes.

1.4 Dissertation Outline

This dissertation implements turbo structured LDPC codes, further extends the encoding design and algorithm presented in [22] and simulation results are also obtained. A more extensive study of turbo structured LDPC codes is presented in[19,22,25].

This dissertation includes five chapters. The first chapter gives an introduction to the digital communication systems, the motivation and the outline of the thesis.

In Chapter 2, the background of error control coding and LDPC codes are presented. The relevant concepts introduced in this chapter are helpful for the exposition of subsequent chapters.

In chapter 3, design and concept of Turbo structured LDPC is discussed.

In chapter 4, a linear complexity encoding algorithm for a type of TS-LDPC codes—encoding friendly TS-LDPC (EFTS-LDPC) codes is explained. Building of EF TS LDPC code for a particular TS LDPC code is also explained.

In chapter 5, the conclusions are given. Possibility to extend present work and future scope of the dissertation is also presented in this chapter.

THEORY OF LDPC CODES CHAPTER 2

2.1 Background

Low-density parity-check (LDPC) codes are forward error-correction codes, proposed in 1962 by Gallager [3,4]. Due to difficulties in the hardware implementation, it was largely neglected for over 35 years. In the mean time the field of forward error correction was dominated by highly structured algebraic block and convolutional codes. Despite the enormous practical success of these codes, their performance fell well short of the theoretically achievable limits set down by Shannon. Since the discovery of turbo codes in 1993 by Berrou et al. [5] and the rediscovery of LDPC codes by Mackay and Neal in 1995 [6], there has been renewed interest in turbo codes and LDPC codes because their bit error rate performance approaches asymptotically the Shannon limit [1]. Commonly, a graph, the Tanner graph [9], is associated with the code and an important parameter affecting the performance of the code is the girth of its Tanner graph.

2.2 Definition

We know that a linear block code C of length n is uniquely specified by a generator matrix G or a parity check matrix H. If it is specified by a parity check matrix H, code C is simply the null space of H. An n- tuple $v = (v_0, v_1 \ldots\ldots, v_{n-1})$ over GF(2) is a code word if and only if $vH^T = 0$. This simply says that the code bits of a codeword in C must satisfy a set of parity check equations specified by the row of H. LDPC codes are specified in terms of their parity check matrices [2].

An LDPC code is defined as the null space of a parity check matrix H that has the following structural properties:

1. Each row consists of j 1's.
2. Each column consists of k 1's.
3. The number of 1's common between any two columns , is no greater than 1; that is λ=0 or 1, and
4. Both j and k are small compared with the length of the code and the numbers of rows in H.

Since, both j and k are small compared with the code length and number of rows in the matrix, hence, H has a small density of 1's. For this reason, H is said to be a low density parity check matrix and the code specified by H is called LDPC code.

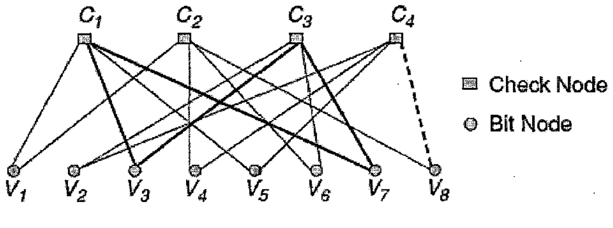
There are two different possibilities to represent LDPC codes: like all linear block codes they can be described via matrices or they can be represented graphically. Graphical representation for LDPC codes was introduced by Tanner which provides a complete representation of the code and also helps to describe the decoding algorithm[9]. Graphical representation discussed below is used in the dissertation to design and implement LDPC cod.

2.3 Tanner Graph

Let $G = \{(V, E)\}$ be a graph, where V is a set of vertices or nodes V and E is a set of edges E connecting the vertices. The degree of a node V is the number of edges incident on V. In an undirected graph, a series of successive edges forming a continuous curve passing from one vertex to another is called a chain.

$$\mathbf{H} = \begin{array}{cccccccc|c} V_1 & V_2 & V_3 & V_4 & V_5 & V_6 & V_7 & V_8 & \\ \hline 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & C_1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & C_2 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & C_3 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & C_4 \end{array}$$

(a)



(b)

Fig 2.1: (a) Parity Check Matrix H of a LDPC code (b) Corresponding Tanner Graph G

A chain of a node where the initial and the terminal nodes are the same and that does not use the same edge more than once is called cycle or in other words it is possible for a path to begin and end at the same vertex. Such a closed path is called Cycle. No vertex on a cycle (except for the initial and the final vertex) appears more than once. The length of a cycle is the number of its edges. The length of the shortest cycle in a graph is called the girth of the graph [9].

The graph G is bipartite if the set of vertices V can be decomposed into two disjoint sets V_1 and V_2 such that no two vertices within either V_1 or V_2 are connected by an edge. It is well known from graph theory that a graph with at least two nodes is bipartite if and only if all its cycles are of even length. Tanner graphs are bipartite where the two disjoint sets V_1 or V_2 collect the bit nodes and the check nodes: each bit of a codeword is assigned a bit node, and each parity-check equation is assigned a check node. The figure 2.1 shows a 4×8 $\boldsymbol{H}$ matrix and its Tanner graph $\boldsymbol{G}$, respectively. The m rows of $\boldsymbol{H}$ correspond to the check nodes and the n columns to the bit nodes of the Tanner graph $\boldsymbol{G}$; they are represented by filled squares and filled circles in Figure 2.1, respectively. A 1 in row C_i and column V_j in H is represented by an edge in the Tanner graph connecting the associated check node C_i and the bit node V_j ; for example, the 1 in row 4 and column 8 in **H** in Figure 2.1 is illustrated by the dashed line between C_4 and V_8 .The bold solid lines *(C_1, V3)*, *(V_3,C_3)*, *(C_3,V_7)*, and *(V_7,C_1)* depict a cycle in the Tanner graph; this turns out to be the shortest cycle in this graph, sȯ that its girth is $g = 4$.

2.4 Regular and irregular LDPC codes

A LDPC code is called regular if j and k are constant for every column and every row. It's also possible to see the regularity of code while looking at the graphical representation [2]. There is the same number of incoming edges for every v-node and also for all the c-nodes. If H is low density but the numbers of 1's in each row or column aren't constant the code is called a irregular LDPC code.

2.5 Decoding of LDPC Codes

An LDPC code can be decoded in various ways, namely majority logic (MLG) decoding, bit flipping (BF) decoding, weighted BF decoding, a posteriori probability (APP) decoding and iterative decoding based on belief propagation (IDBP) (commonly known as sum product algorithm (SPA)) [2]. The first two types are hard decision decoding, the last two are soft decision decoding, and the third one is in between. MLG decoding is simplest one in decoding complexity. BF decoding requires a little more decoding complexity but gives better error performance than MLG decoding. APP decoding and the SPA decoding provide much better error performance but require larger complexity than MLG and BF decoding. The weighted BF offers a good trade-off between error performance and decoding complexity. SPA decoding gives the best error performance among the five types of decoding of LDPC codes and yet is practically implementable.

In some algorithms, such as bit-flipping decoding, the messages are binary and in others, such as *belief propagation* decoding, the messages are probabilities which represent a level of belief about the value of the codeword bits. It is often convenient to represent probability values as log likelihood ratios, and when this is done belief propagation decoding is often called sum-product decoding since the use of log likelihood ratios allows the calculations at the bit and check nodes to be computed using sum and product operations

2.5.1 Sum product algorithm

The sum-product algorithm is a soft decision message-passing algorithm. The input bit probabilities are called the *a priori* probabilities for the received bits because they were known in advance before running the LDPC decoder. The bit probabilities returned by the decoder are called the *a posteriori* probabilities. In the case of sum-product decoding these probabilities are expressed as *log-likelihood ratios*.

For a binary variable x it is easy to find p($x = 1$) given p($x = 0$), since p($x = 1$) = 1-p ($x = 0$) and so we only need to store one probability value for x. Log likelihood ratios are used to represent the metrics for a binary variable by a single value:

$$L(x) = \log(\frac{p(x=0)}{p(x=1)}) \tag{2.1}$$

Where we use *log* to mean $\log_e$. If $p(x = 0) > p(x = 1)$ then $L(x)$ is positive and the greater the difference between $p(x = 0)$ and $p(x = 1)$, i.e. the more sure we are that $p(x) = 0$, the larger the positive value for $L(x)$. Conversely, if $p(x = 1) > p(x = 0)$ then $L(x)$ is negative and the greater the difference between $p(x = 0)$ and $p(x = 1)$ the larger the negative value for $L(x)$. Thus the sign of $L(x)$ provides the hard decision on x and the magnitude $|L(x)|$ is the reliability of this decision. To translate from log likelihood ratios back to probabilities we note that

$$p(x=1)=\frac{p(x=1)/p(x=0)}{1+p(x=1)/p(x=0)}=\frac{e^{-L(x)}}{1+e^{-L(x)}} \qquad (2.2)$$

and

$$p(x=0)=\frac{p(x=0)/p(x=1)}{1+p(x=0)/p(x=1)}=\frac{e^{L(x)}}{1+e^{L(x)}} \qquad (2.3)$$

The benefit of the logarithmic representation of probabilities is that when probabilities need to be multiplied log-likelihood ratios need only be added, reducing implementation complexity.

The aim of sum-product decoding is to compute the *maximum a posteriori probability* (MAP) for each codeword bit, $P_i = P\{c_i = 1|N\}$, which is the probability that the i-th codeword bit is a 1 conditional on the event N that all parity-check constraints are satisfied. The extra information about bit i received from the parity-checks is called *extrinsic* information for bit i.

The sum-product algorithm iteratively computes an approximation of the MAP value for each code bit. However, the a posteriori probabilities returned by the sum-product decoder are only exact MAP probabilities if the Tanner graph is cycle free. Briefly, the extrinsic information obtained from a parity check constraint in the first iteration is independent of the a priori probability information for that bit (it does of course depend on the a priori probabilities of the other codeword bits). The extrinsic information provided to bit i in subsequent iterations remains independent of the original a priori probability for bit i until the original a priori probability is returned back to bit i via a cycle in the Tanner graph. The correlation of the extrinsic information with the original a priori bit probability is what prevents the resulting posteriori probabilities from being exact.

In sum-product decoding the extrinsic message from check node j to bit node *i*, $E_{j,i}$ is the LLR of the probability that bit i causes parity-check j to be satisfied. The probability that the parity-check equation is satisfied if bit i is a 1 is

$$P_{j,i}^{ext} = \frac{1}{2} - \frac{1}{2} \prod_{i' \in B_j, i' \neq i} (1 - 2P_{i'}^{\text{int}}) \qquad (2.4)$$

Where $P_{j,i}^{ext}$ is the current estimate, available to check j, of the probability that bit i' is a one. The probability that the parity-check equation is satisfied if bit I is a zero is thus 1-$P_{j,i}^{ext}$. Expressed as a log-likelihood ratio,

$$E_{j,i} = LLR(P_{j,i}^{ext}) = \log(\frac{1 - P_{j,i}^{ext}}{P_{j,i}^{ext}}) \qquad (2.5)$$

And substituting (2.4) gives

$$E_{j,i} = \log(\frac{\frac{1}{2} + \frac{1}{2} \prod_{i' \in B_{j,i' \neq i}} (1 - 2P_{j,i}^{int})}{\frac{1}{2} - \frac{1}{2} \prod_{i' \in B_{j,i' \neq i}} (1 - 2P_{j,i}^{int})}) \qquad (2.6)$$

Using the relationship

$$\tanh(\frac{1}{2} \log(\frac{1-p}{p})) = 1 - 2p \, , \qquad (2.7)$$

Gives

$$E_{j,i} = \log(\frac{1 + \prod_{i' \in B_{j,i' \neq i}} \tanh(M_{j,i'}/2)}{1 - \prod_{i' \in B_{j,i' \neq i}} \tanh(M_{j,i'}/2)}) \qquad (2.8)$$

Where

$$M_{j,i'} = LLR(P_{j,i'}^{int}) = \log(\frac{1-P_{j,i'}^{int}}{P_{j,i'}^{int}}) \tag{2.9}$$

Alternatively, using the relationship

$$2\tan^{-1}(p) = \log(\frac{1+p}{1-p}) \tag{2.10}$$

(2.8) can be written as :

$$E_{j,i} = 2\tan^{-1}(\prod_{i' \in B_{j,i' \neq i}} \tanh(M_{j,i'}/2))$$

Each bit has access to the input a priori LLR, r_i, and the LLRs from every connected check node. The total LLR of the *i*-th bit is the sum of these LLRs:

$$L_i = LLR(P_{i'}^{\text{int}}) = r_i + \sum_{j \in A_i} E_{j,i} \tag{2.11}$$

However, the messages sent from the bit nodes to the check nodes, $M_{j,i}$, are not the full LLR value for each bit. To avoid sending back to each check node information which it already has, the message from the i-th bit node to the jth check node is the sum in without the component Ej,i which was just received from the j-th check node:

$$M_{j,i} = \sum_{j' \in A_i, j' \neq j} E_{j',i} + r_i \tag{2.12}$$

The sum-product algorithm is shown in Algorithm . Input is the log likelihood ratios for the a priori message probabilities

$$r_i = \log(\frac{p(c_t = 0)}{p(c_t = 1)}) \tag{2.13}$$

the parity-check matrix H and the maximum number of allowed iterations, I_{max}. The algorithm outputs the estimated a posteriori bit probabilities of the received bits as log likelihood ratios.

Algorithm : Sum-Product Decoding

```
procedure DECODE(y)

    $I = 0$                                         Initialization
    for $i = 1 : n$ do
        for $j = 1 : m$ do
            $M_{j,i} = r_i$
        end for
    end for

    repeat
        for $j = 1 : m$ do                          Step 1: Check messages
            for $i \in B_j$ do
                $E_{j,i} = \log\left(\frac{1 + \prod_{i' \in B_{j,i' \neq i}} \tanh(M_{j,i'}/2)}{1 - \prod_{i' \in B_{j,i' \neq i}} \tanh(M_{j,i'}/2)}\right)$
            end for
        end for

    for $i = 1 : n$ do . Test
        $L_i = \sum_{j \in A_i} E_{j,i} + r_i$
        $z_i = \begin{cases} 1, \rightarrow L_i \leq 0 \\ 0, \rightarrow L_i > 0 \end{cases}$
    end for
    if $I = I_{max}$ or $Hz^{-T} = 0$ then
```

```
            Finished
        else
            for i = 1 : n do                         Step 2: Bit messages
                for j ∈ A_i do
            $M_{j,i} = \sum_{j' \in A_i, j' \neq j} E_{j',i} + r_i$
                end for
            end for
            I = I + 1
            end if
        until Finished
    end procedure
```

The computational complexity and decoding delay (or decoding time) of the SPA increase as the number of decoding iterations increases. Long decoding delays are not desirable in high speed communication and data storage systems. If an LDPC code has a large MLG error correcting capability, such as a long finite geometry LDPC code , it is possible to device a hybrid SPA decoding scheme with MLG decoding to shorten the decoding iteration process and hence the decoding delay with only a small degradation in error performance. Based on many experimental results it is observed that SPA decoding of long finite geometry LDPC codes converges very fast.

2.6. Cycles, Girth and Performance

It is well known that with a cycle-free Tanner graph, the sum-product algorithm terminates naturally in a finite number of steps and yields optimal decoding in the sense that the symbol error probability is minimized [17]. However, cycle-free Tanner graphs have poor BER performance: their minimum distance is two at code rates $r > 1/2$, and their error floors occur at unacceptable values of SNR [18]. When Tanner graphs have cycles, the resulting sum-product algorithm is suboptimal [17]. Cycles, especially short cycles in the Tanner graph, lead to inefficient decoding and prevent the sum-product algorithm from converging to the optimal decoding result [18], taxing the performance of the LDPC decoders. Intuitively, the girth determines the smallest number of iterations for a message sent by a node (in the shortest cycle of the graph) to propagate back to the node itself. This causes loss of independence in the extrinsic information merged on a node in the iterative decoding through the successive iterations; Gallager [3] showed that the number of independent iterations M is proportional to the girth g of the Tanner graph, in particular, $M < g/4 \leq M + 1$. A second reason relates to the minimum distance of the code. Tanner [9] derives a lower bound on the minimum distance d_{min}, this lower bound increases with the girth g of the code. Therefore, LDPC codes with large girth are to be preferred.

To increase the girth g and avoid short cycles, the parity-check matrix **H** should be sufficiently sparse. In turn, this means that the block length must be large enough. In fact, Gallager [3] provides a loose lower bound on the block length n given the girth g of a regular LDPC code. For a parity-check matrix **H** with column weight j and row weight k, the bound on the block length n is

1) for $g = 4m + 2$, $n \geq \sum_{i=1}^{m+1} S_i$, where $S_1 = 1$, and for $i \geq 2, S_i = j(j-1)^{(i-2)}(k-1)^{(i-1)}$

2) for g=4m, $n \geq \sum_{i=1}^{m} L_i$, where $L_i = k(j-1)^{(i-1)}(k-1)^{(i-1)}, \forall i$

For example, to construct an $(n, j = 3, k = 12)$ LDPC code with girth $g = 12$, its block length n should satisfy $n \geq 6,084$. Of course, this lower bound is just that; even if n exceeds the bound for a given g there might not exist a regular LDPC code with this block length n and the desired girth g.

TURBO STRUCTURED LDPC CODES CHAPTER 3

3.1 Introduction

The turbo-structured Low Density Parity Check (TS-LDPC) codes can be designed with arbitrary large girth by appropriately choosing the code block length n. TS-LDPC codes are hardware friendly: they are regular and structured, and their parity check matrix **H**, which can be very large, is determined from a much smaller object, the shift matrix **S**, so that their memory requirements can be negligible [19].

Turbo structures have been used in [20],[21] to construct LDPC codes. These codes combine two LDPC codes as component codes on the *encoder* side. The structure of the turbo *encoder,* replaces the recursive convolutional codes commonly used in turbo codes with a tree code—a specific LDPC code whose associated graph for the code-generating matrix , not for the parity-check matrix **H** , is a tree. In [21], the *encoder* is a parallel concatenation of two LDPC codes without an interleaver, as shown on the left of Fig. 3.1

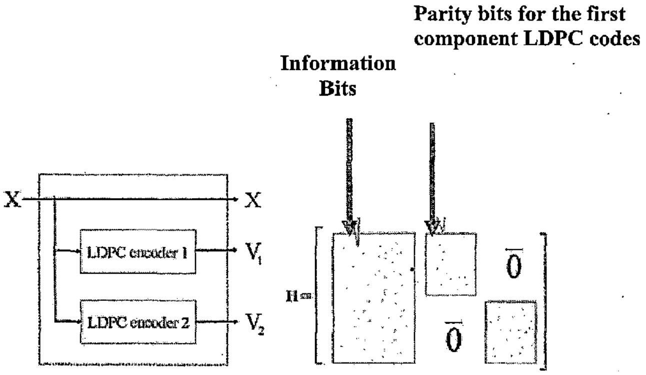


Fig 3.1 Left: The encoder structure for the concatenated LDPC codes in [20]. Right: The parity-check matrix H for the concatenated LDPC codes in [20], [21].

3.2 Turbo Design of LDPC codes

This design is similar to the turbo structure: two tree structures are interconnected to create LDPC codes with large girth[22]. The factor graph of such a code contains two height-balanced sub-trees, denoted as an upper-tree T_U, for which the leaf nodes are variable (bit) nodes, and a lower-tree T_L for which the leaf nodes are check nodes. Thee height of T_U and T_L, say, its number of tiers, is represented by *h*. The first tier of T_U contains only one check node-the root, as shown in figure 3.2.

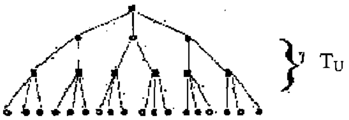


Fig. 3.2. Upper tree T_U of a turbo-like LDPC code with column weight 3, row weight 4 and height 4 [19]

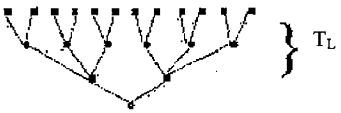


Fig.3.3 Lower tree *TL* of a turbo-like LDPC code with column weight 3, row weight 4, and height 4 [19]

On the other hand, the root of T_L (shown in figure 3.3) is a variable (bit) node. The two trees are "combined" in a turbo-like manner such that the leaf-nodes of T_L are connected to the leaf-nodes of T_U as shown in Fig 3.4. The structure formed by connecting edges between the leaf-nodes of T_U and T_L is called "interleaver." All the variable nodes have uniform degree *j* and all the check nodes have the same degree *k*, except for the roots of T_U and T_L, whose degrees are set to be k - 1 and j - 1 respectively.

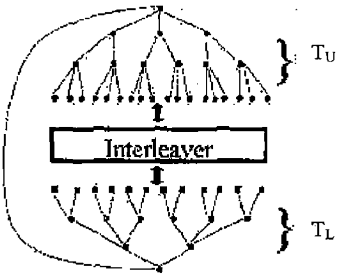


Fig. 3.4 TS LDPC code with column weight 3, row weight 4 and height 4[19]

For example, a turbo-like LDPC code with $h = 4$, $j = 3$ and $k = 4$ is shown in figure 3.4. To make the code exactly regular, the root (a check node) of T_U and the root (a variable node) of T_L are connected directly by an edge, hence forming a regular LDPC code. For a given code rate r, just choose the two parameters j and k to satisfy the equation $r = 1-j/k$, for example, for $r = 8/9$ and column weight 3, simply let $j = 3$ and $k = 27$ in each of the component trees.

The significance of the above structures is that in isolation, no cycle exists in either of the sub-trees. The cycles in turbo-like LDPC codes are introduced by the interleaver. In subsequent paragraphs we will discuss methods to devise interleavers that guarantee that the resulting turbo-like LDPC codes have girth greater than or equal to 8.

3.3 LDPC codes using turbo designs

The goal of this section is to build turbo-like LDPC codes with large girth. By construction, each leaf-node in the upper tree T_U is connected to $q = j - 1$ leaf-nodes in T_L. This means this TS-LDPC interleaver is a 1-to-q mapping while, usually, interleavers are a one to one mapping between elements of two sets with the same size. For convenience, "auxiliary nodes" (represented by solid triangles) as shown in figure 3.5 are introduced. For each leaf node in the upper tree T_U, since it connects to $j - 1$ check nodes in the lower tree T_L, add $j - 1$ auxiliary node to it and let these auxiliary nodes its descendants. Similarly, auxiliary nodes to the lower tree T_L are introduced, such that each leaf node of T_L has $k - 1$ auxiliary node as its descendants. There is a one to one correspondence between auxiliary nodes of T_U and auxiliary nodes of T_L. Figure 3.6 shows a path connecting T_U's leaf node *A* to T_L's leaf node *B* through T_U's auxiliary node C and T_L's auxiliary node *D*. That means, in the original factor graph, nodes *A* and *B* are directly connected by an edge.

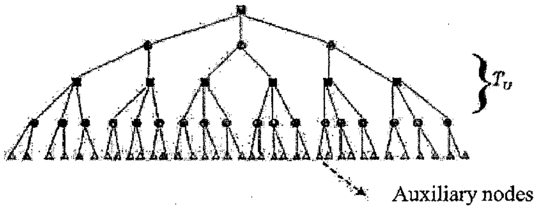


Fig. 3.5. Auxiliary nodes of T_U [19]

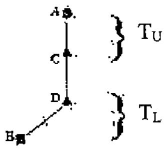


Fig. 3.6. Auxiliary nodes of T_U and T_L

Therefore, it can be equivalently expressed as finding an appropriate one-to-one mapping between auxiliary nodes of T_U and auxiliary nodes of T_L that guarantees large girth. As mentioned, no cycles exist in any of the sub-trees in isolation, so cycles present in the codes must contain "auxiliary nodes," which is at least four (two auxiliary nodes of T_U and two auxiliary nodes of T_L). Cycles can be divided into two disjoint categories: type-I and type-II cycles.

Type of cycle depends upon how many auxiliary nodes a cycle contains. Type-I cycles contain four and only four auxiliary nodes and are denoted by C-I; type-II cycles contain more than four auxiliary nodes and are denoted by C-II. Same can be clearly seen in following figure 3.7.

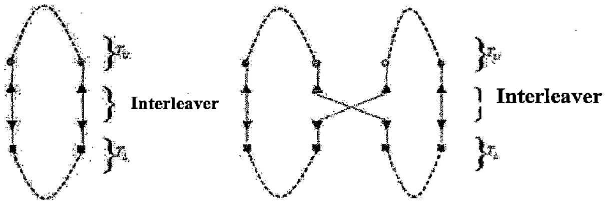


Fig. 3.7 Left: A type I cycle. Right: A type II cycle. [22]

3.4 Interleaver Design

It is clear that in Turbo structured LDPC design, cycles are present only in interleaver, hence it is designed to prevent cycles of length smaller than the desired girth *g*. The interleaver is designed by specifying the rules of how to connect the leaf bit nodes in the upper tree T_U to the leaf check nodes in the lower tree T_L Interleaver designs for turbo codes have been extensively studied in [23,24]. The functions of the interleavers for turbo codes are to avoid low-weight codewords and to decrease the correlation between the extrinsic information and the input data sequence. They are not designed to construct Tanner graphs with large girth. Hence, we can not directly borrow the existing interleaver designs for turbo codes, say, the S -random interleaver, for designing TS-LDPC codes. Here it is required to develop new interleavers that suit the structure of TS-LDPC codes and lead to TS-LDPC codes with large girth.

In subsequent paragraphs p – q alternate-decimal indexing is used, as discussed in [22]. Nodes in the tanner graph are labeled using p-q alternate decimal indexing which helps in establishing connection between leaf nodes of T_U and T_L. This p-q indexing is described in detail in [22]. In this indexing technique rules to prevent cycles of length smaller than the desired girth g are designed .This is achieved by categorizing the cycles in two types: type I and type II cycles. Section 3.4.2 specifies the connecting rules to avoid type I cycles. Type II cycles are considered in Section 3.4.3, they are prevented by "grouping and shifting:" The leaf nodes are grouped and connected with leaf nodes in distinct trees whose labels are appropriately shifted. Section 3.4.4 discusses the number of groups needed. Based on the discussions in Sections 3.4.1-3.4.4 the detailed algorithm to construct shift matrix for TS-LDPC codes is presented in 3.5.5.

3.4.1. (p–q)-Alternate-Decimal Format

Let the sub trees T_U and T_L have height h, the bit nodes degree j, and the check nodes degree k. Let $p = k - 1$ and $q = j - 1$. For T_U with h tiers, there are $[(k - 1)(j - 1)]^{h/2}$ auxiliary nodes of T_U. The interleaver design problem is addressed algebraically by indexing all the auxiliary nodes of T_U and T_L. Auxiliary nodes of T_U are indexed from 0 to $[(k - 1)(j - 1)]^{h/2}$ -1 in the following format – the p-q alternate decimal format, where p= k-1 and q = j-1. There will be h digits in the p-q alternate decimal indexing to label all the auxiliary nodes in T_U. These h digits are numbered from 1 to h, starting from rightmost. The odd coordinates take values from 0 to $q - 1$, and the even coordinates take values from 0 to $p - 1$. The position of each digit is referred as its coordinate . Similarly, all the auxiliary nodes of T_L are indexed from 0 to $[(k - 1)(j - 1)]^{h/2}$ -1 and represent also all these indices in $(q - p)$ alternate-decimal[25].

To be concrete lets see an example where the upper tree has height $h = 4$, the index X_{p-q} of an auxiliary node in the upper tree T_U is

$$X_{p-q} = \underbrace{x_4}_{p}\underbrace{x_3}_{q}\underbrace{x_2}_{p}\underbrace{x_1}_{q} \qquad 3.1$$

In (3.1), x_i , i = 1,2,....4, represents the ith digit. The coordinate of x_i is i and the corresponding value of X_{p-q} in decimals is

$$X_{p-q} = (x_4 * p^1 q^2 + x_3 * p^1 q^1 + x_2 * q^1 + x_1)_{10} \qquad 3.2$$

where $(\cdot)_{10}$ represents the decimal value of $Xp-q$. From hereon, the "indices" of nodes are referred as p-q alternate-decimal or q-p alternate-decimal representations.

3.4.1a *Lemma 1:*

(a) Let the distance between two auxiliary nodes A and B within T_U be $d_U(A,B)$. X_A and X_B are indices for A and B, respectively. If i is the leftmost coordinate where the digits of X_A and X_B differ from each other, then $d_U(A,B) = 2i$.

(b) Denote the distance between two auxiliary nodes a and B within T_L as $d_L(A,B)$. X_A and X_B are indices for A and B . If i is the leftmost coordinate where the digits of X_A and X_B differ from each other, then $d_L(A,B) = 2i$.

Proof: Detailed proof is given in [22]. As discussed in [22] part (a) of the lemma is proved first. From the definition of the p-q alternate-decimal indexing, the first common ancestor R of the auxiliary nodes A and B is in the (h+1-i) tier of T_U. Where i is the leftmost coordinate where the digits of X_A and X_B differ from each other and h denotes the number of tiers in T_U. For example, let $X_A = 0000$ and $X_B = 1000$ be the indices of two auxiliary nodes A and B of T_U . Since the leftmost coordinate where the digits of X_A and X_B differ from each other is the fourth, then the first common ancestor R of the auxiliary nodes A and B is the root of T_U , located in the first tier of T_U .

To find the shortest path in that connects A and B , one has to go up i tiers from A to reach its first common ancestor R and then go downwards i tiers from R to B . Therefore, the distance through the tree T_U between A and B is then $d_U(A,B) = 2i$. Part (b) is also similar to part (a).

3.4.2 Avoiding Short Type I Cycles—Digit-Wise Reversal

Interleaver design can be done with a simple concept —*digit-wise reversal*, [8]. For an index in *Xp−q* in p-q alternate-decimal form with h digits, its digit-wise reversal interchanges the ith digit and the (h+1-i)th digit for i=1,2,......,h/2. the digit-wise reversal operator is represented by $\pi_d(.)$. For the index *Xp−q* in (3.1), its digit-wise reversal is

$$\pi_d(X_{p-q}) = x_1 x_2 x_3 x_4$$

$$= (x_1 \times qp^2 + x_2 \times qp + x_3 \times qp + x_4)_{10} \qquad (3.3)$$

Digit-wise reversal interleaver advantage can be understood in the following theorem [22].

3.4.2a ***Theorem 1:*** Connecting the auxiliary nodes indexed by $X_{p\text{-}q}$ in T_U to the auxiliary nodes indexed by π_d ($X_{p\text{-}q}$) in T_L guarantees that any resulting type I cycle is at least of length 2h , where h denotes the number of tiers in T_U .

Proof: As discussed in [22] from its definition, a type I cycle contains four auxiliary nodes A_1 ,A_2 ,A_3 , and A_4 as shown on the left in Fig. 3.8 Their associated indices are X_1, X_2 , X_3 , and X_4, respectively.

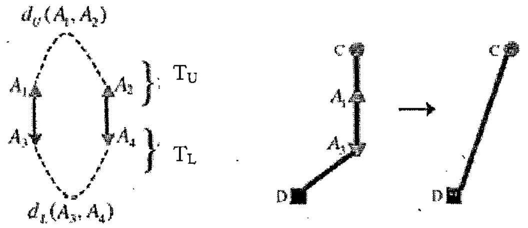


Fig.3.8 . Left: Type I cycle with four auxiliary nodes A_1 ,A_2 ,A_3 , and A_4 .
Right: A path of length 3 that contains two auxiliary nodes is actually a path of length 1.[22]

The distance between A_1 and A_2 within the tree T_U is denoted as d_U (A_1 A_2), and the distance of A_3 and A_4 within T_L as $d_L(A_3,A_4)$. According to the left plot in Fig. 3.8 the length of a type I cycle is:

$$L' = d_U(A_1, A_2) + d_L(A_3, A_4) + 2 \qquad (3.4)$$

However, as auxiliary nodes A_1 ,A_2 ,A_3 , and A_4 are imaginary nodes, the type I cycle does not contain them as vertices. For example, the path of length 3 with two auxiliary nodes A_1 and A_3 shown on the right in Fig. 3.8 is actually a path of length in the Tanner graph. Therefore, the actual cycle length is , i.e.,

$$L = L' - 4 = d_U(A_1, A_2) + d_L(A_3, A_4) - 2 \qquad (3.5)$$

The distance d_U (A_1 A_2) is related to the value of their indices. By Lemma 1, d_U (A_1 A_2) = 2i , where i is the leftmost coordinate where the digits of X_1 and X_2 differ from each other. Let h represent the height of T_U After digit-wise reversal, the digits of X_1 and X_2 at the coordinate i become the digits of π_d (X_1) and π_d (X_2) at the coordinate h+1-i , respectively. So, the digits of π_d (X_1) and π_d (X_2) at the coordinate h+1-i are different. According to the connecting rule stated in the theorem, $X_3 = \pi_d$ (X_1) and

X_4 =, π_d (X_2) so the digits of X_3 and X_4 at the coordinate h+1-i are different. Therefore, by *Lemma 1*

$$d_L(A_3, A_4) \geq 2(h+1-i) \qquad (3.6)$$

By (3.5), the length of such a type I cycle is then

$$L = d_U(A_1, A_2) + d_L(A_3, A_4) \geq 2h \qquad (3.7)$$

From the above analysis, all type I cycles that result from following the connection rule in Theorem 1 are at least of length 2h. hence it is clear from above theorem that to increase the length of type I cycles one need simply to increase the number of tiers in the upper and lower trees.

3.4.3 Avoiding Type II Cycles—Grouping and Shifting

The connection rule in Theorem 1 prevents short type I cycles. The connection rule to avoid short type II cycles as discussed below. To exclude short type II cycles, a different concept grouping and shifting is introduced.

Shifting Shift S is defined as to be a constant in q –p alternate-decimal format that is added to the original index π_d (X_{p-q}) to form a new index. This can be illustrated with an example. Let π_d (X_{p-q}) $= x_1x_2x_3x_4$,the shift S $= s_1s_2s_3s_4$, and $\dot{+}$ represents the digit-wise addition (with no carry). Then

$$\pi_d(X_{p-q}) \dot{+} S = y_1y_2y_3y_4 \qquad (3.8)$$

In (3.8), $y_i = x_i \dot{+} s_i = \mathrm{mod}\,(x_i + s_i\,,\, div_i)$, where div_i = p if i is even and div_i = q if i is odd. In a similar fashion, the digit-wise subtraction is represented by $\dot{-}$.

Grouping The auxiliary nodes of T_U are divided into groups of the same size according to their indices. Those auxiliary nodes whose indices have the same t leftmost digits are placed in the same group. The auxiliary nodes T_L of can, likewise, be classified into groups based also on whether their indices have the same leftmost t digits.

After clustering the auxiliary nodes into groups, the shift S is considered to be the same when the auxiliary nodes of T_U in the same group are connected to the auxiliary nodes of T_L in the same group. The shift introduced is denoted by $S_{\alpha,\beta}$ when the auxiliary nodes of T_U in the α^{th} group are connected to the auxiliary nodes of T_L in the β^{th} group. For different α and β , the shifts $S_{\alpha,\beta}$ may be the same or different from each other. The mapping rule for the interleaver is discussed below.

3.4.3a.Connection rule to avoid type II cycles

Connect the auxiliary node indexed by X_{p-q} in the α^{th} group in T_U to the auxiliary node indexed by π_d (X_{p-q}) in the β^{th} group in T_L to auxiliary node indexed by

$\pi_d(X_{p-q}+S_{\alpha,\beta}$ in the β^{th} group of T_L.

This rule prevents short type II cycles. But, first, it needs to be ensured that using this rule does not introduce short type I cycles. This is settled in the next theorem that shows that type I cycles do not depend on the shifts $S_{\alpha,\beta}$.

Theorem 2: Connecting the auxiliary node indexed by X_{p-q} in the α^{th} group in T_U to the auxiliary node indexed by π_d (X_{p-q}) + $S_{\alpha,\beta}$ in the β^{th} group in T_L guarantees that any type I cycle formed is at least of length 2h, where h denotes the number of tiers in T_L .

From Theorem 2, any shift $S_{\alpha,\beta}$ in the rule $X_{p-q} \rightarrow \pi_d$ (X_{p-q}) + $S_{\alpha,\beta}$ can be selected, because the length of any type I cycle is guaranteed to be greater than or equal to 2h . The freedom to choose the values of $S_{\alpha,\beta}$ is exploited to avoid short type II cycles.

It is observed that each type II cycle with 2k edges in interleaver is associated with 2k shifts

$$S_{\alpha_1,\beta_1}, S_{\alpha_2,\beta_2}, \ldots\ldots\ldots\ldots S_{\alpha_{2k},\beta_{2k}} \qquad (3.9)$$

Type II cycle is characterized by the shift sequence A= $\{S_{\alpha1,\beta1}, S_{\alpha2,\beta2}, S_{\alpha3,\beta3} \ldots\ldots\ldots S_{\alpha2k,\beta2k}\}$.The index labels of shift sequence characterizing a type II cycle satisfy the following two conditions:

(i) $\alpha_{2t-1} \neq \alpha_{2t}$,t=1,2.....k and $\beta_{2t-1}=\beta_{2t}$, t=12....k.

(ii) $\alpha_{2t} = \alpha_{2t+1}$,t=1,2....k-1 and $\alpha_{2k}= \alpha_1$ and $\beta_{2t} \neq \beta_{2t+1}$ t=1,2...k-1and $\beta_{2k} \neq \beta_1$.

Given a shift sequence A= $\{S_{\alpha1,\beta1}, S_{\alpha2,\beta2}, S_{\alpha3,\beta3} \ldots\ldots\ldots S_{\alpha2k,\beta2k}\}$ that satisfies conditions (i) and (ii) above, its accumulated alternate sum Δ_A to be

$$\Delta_A= \sum_{i=1}^{2k}(-1)^{i+1}S_{\alpha_i,\beta_i} \qquad (3.10)$$

$=S_{\alpha 1,\beta 1} - S_{\alpha 2,\beta 2} + S_{\alpha 3,\beta 3} - .... + S_{\alpha 2i-1,\beta 2i-1} - S_{\alpha 2i,\beta 2i} + ... + S_{\alpha 2k-1,\beta 2k-1} - S_{\alpha 2k,\beta 2k}$. (3.11)

Following theorem helps to eliminate type II cycles with 2k edges in the interleaver.

Theorem 3: Let $A = \{S_{\alpha 1,\beta 1}, S_{\alpha 2,\beta 2}, S_{\alpha 3,\beta 3} .... S_{\alpha 2k,\beta 2k}\}$ be a shift sequence that contains 2k shifts.

$$\Delta_A = \sum_{i=1}^{2k} (-1)^{i+1} S_{\alpha_i,\beta_i} \qquad (3.12)$$

is the accumulated alternate sum of A and has h digits in its q-p alternate-decimal expansion. If Δ_A contains at most $d = h - \ell$ consecutive "0" in its q-p alternate-decimal expansion, then any type II cycles characterized by A has length no less than $2(\ell + k)$.

Proof: In order to prove theorem 3, as discussed in [22] an equivalent proposition will be proved: If there exists a type II cycle with 2k edges in the interleaver and its length is less than $2(\ell+k)$, then the associated $\Delta_A = \sum_{i=1}^{2k} (-1)^{i+1} S_{\alpha_i,\beta_i}$ must contain more than $d = h - \ell$ consecutive digits "0". Fig. 3.9 shows a type II cycle with edges in the interleaver. Let this type II cycle contain N_U edges in T_U and N_L edges in T_L. By assumption, the length of this type II cycle is less than $2(\ell+k)$. Since the length of the type II cycle is $N_U + N_L + 2k$, then

$$N_U + N_L < 2\ell \qquad (3.13)$$

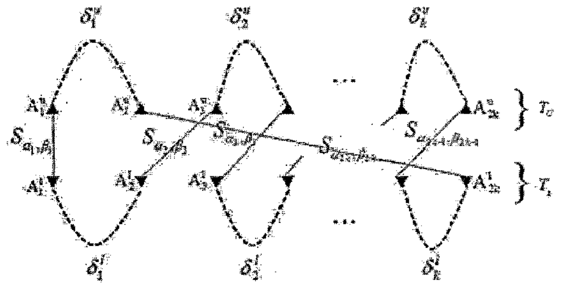


Fig.3.9 . Type II cycle with 2k edges in the interleaver.

Since there are 2k edges in the interleaver, the cycle contains 4k auxiliary nodes, 2k auxiliary nodes $A_1^u, A_2^u, ... A_{2k}^u$ in T_U, and 2k auxiliary nodes in $A_1^l, A_2^l, ... A_{2k}^l$ in T_L. With reference to the plot in Fig. 3.9, and assuming that the index for the auxiliary node

A_1^u is $X_{A_1^u}$, according to the connecting rule presented in Theorem 2, the index for the auxiliary node is $X_{A_1^l} = \pi_d\,(X_{A_1^u}) + S_{\alpha 1,\beta 1}$. Let δ_1^l denote the difference between $X_{A_2^l}$ and $X_{A_1^l}$, i.e., $\delta_1^l = X_{A_2^l} - X_{A_1^l}$. The index for the auxiliary node $X_{A_3^U}$ is $X_{A_1^l} = \pi_d\,(X_{A_2^l} - S_{\alpha 2,\beta 2})$. Again, let $\delta_2^u = X_{A_4^U} - X_{A_3^U}$. The index for the auxiliary node A_3^l is $X_{A_3^l} = \pi_d(X_{A_4^U}) + S_{\alpha 3,\beta 3}$.

Continuing to trace the cycle and finding the relationships between the indices of the auxiliary nodes, at auxiliary node $X_{A_2^U}$, $X_{A_2^U} = \pi_d(X_{A_{2k}^l}) - S_{\alpha 2k,\beta 2k}$ .The relationship between $X_{A_1^U}$ and $X_{A_2^U}$ is $X_{A_1^U} = X_{A_2^U} + \delta_1^U$. Iterating in the definition of $X_{A_2^U}$,we have

$$\sum_{t=1}^{k}(-1)^{i+1}S_{\alpha i,\beta i} + \pi_d(\sum_{s=1}^{k}\delta_s^U) + \sum_{t=1}^{k}\delta_t^l = 0 \tag{3.14}$$

Since the cycle has N_L edges in T_L , then the distance between auxiliary nodes A_{2t-1}^l and A_{2t}^l t=1,2,….k, through T_L is less than or equal to N_L . By *Lemma* 1, it is known that only the rightmost $N_L/2$ digits of δ_t^l , t=1,2,…..k , can be nonzero, the other digits of δ_t^l have to be zero. Note that, for the digit-wise add +, there is no carry. Therefore, only the rightmost $N_L/2$ digits of $\sum_{t=1}^{k}\delta_t^l$ can be nonzero. Similarly, it can be derived that only the rightmost $N_U/2$ digits of $\sum_{t=1}^{k}\delta_s^U$ can be nonzero. From the definition of the digit-wise reversal π_d (.) only the leftmost $N_U/2$ digits of $\pi_d\,(\sum_{t=1}^{k}\delta_s^U)$ can be nonzero. According to (3.14), it can be derived that only the leftmost $N_U/2$ digits and the rightmost $N_L/2$ digits of Δ_A can be nonzero. That means that the intermediate h - $N_L/2$ - $N_U/2$ digits of Δ_A are zero. Since by (3.13) , $N_U + N_L < 2\ell$ then Δ_A contains more than h - $N_L/2$ - $N_U/2$ consecutive zeros. Thus, if a cycle has 2k edges in the interleaver and its length is less than $2(\ell+k)$, then its associated Δ_A contains more than h-ℓ consecutive zeros in its p-q alternate-decimal representation.

3.4.5 Minimum Number of Groups in Each Tree

There is a tradeoff when deciding the number of groups to choose in each tree. On the one hand, to get compact TS-LDPC codes, small number of groups in T_U and T_L are preferred. To reduce the number of groups, the number t of the common leftmost digits of the indices of the auxiliary nodes in the same group should be as small as possible. On the other hand, a smaller number of groups decreases the number of free parameters in the code design.

Lemma 3: To achieve a girth g , the minimum number groups G_U and G_L are

$$G_U \geq ((k-1)(j-1))^{\left\lfloor \frac{(g-2)/4}{2} \right\rfloor}(k-1)^{\left\lfloor \frac{(g-2)}{4} \right\rfloor \bmod 2} \qquad (3.15)$$

$$G_L \geq ((k-1)(j-1))^{\left\lfloor \frac{(g-2)/4}{2} \right\rfloor}(j-1)^{\left\lfloor \frac{(g-2)}{4} \right\rfloor \bmod 2} \qquad (3.16)$$

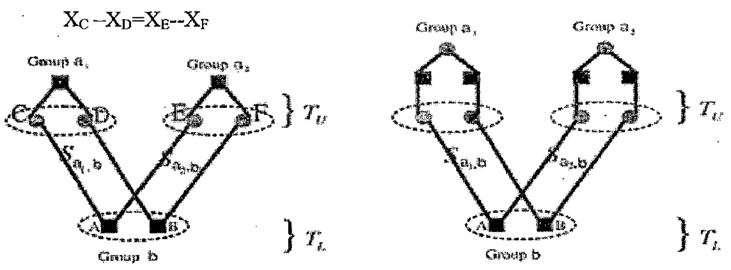


Fig. 3.10. Left: A length 8 cycle that can NOT be excluded by any choice of the shifts $S_{a1,b}$ and $S_{a2,b}$.
Right: A length 12 cycle that can NOT be excluded by any choice of the shifts $S_{a1,b}$ and $S_{a2,b}$ [22]

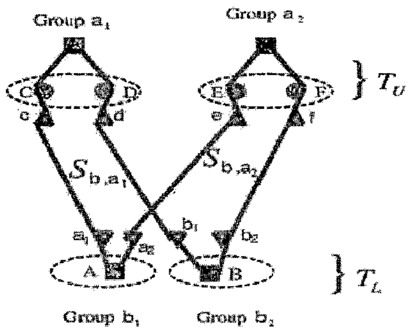


Fig.3.11 Check nodes A and B divided into subgroups to avoid a length 8 cycle. [22]

Proof: As given in [22] an example is discussed first. The length 8 cycles shown on the left in Fig. 3.10 cannot be avoided when the two check nodes A and B are in the same group. To avoid this length 8 cycle, A and B must be in two different groups, as shown in Fig. 3.11. Since A connects to the auxiliary node c and B connects to the auxiliary node d in Fig. 3.11, c and d must connect to check nodes in two different groups. Further, since c and d can be any two auxiliary nodes whose indices are different only in the two rightmost digits, we conclude any two auxiliary nodes whose two rightmost digits are different should be connected to different groups. Since there are (j-1)(k-1) categories of such auxiliary nodes, at least (j-1)(k-1) groups in T_L are needed . Similarly, to avoid the length 12 cycle shown on the right in Fig. 3.10, at least $(j-1)^2(k-1)$ groups in T_L are required .

More generally, to avoid the cycle with length L=4d+4, when d is odd, at least $G_L=(j-1)^{\frac{(d+1)}{2}}(k-1)^{\frac{(d+1)}{2}}$ groups in T_L are needed; when d is even, at least $G_L=(j-1)^{\frac{d}{2}+1}(k-1)^{\frac{d}{2}}$ groups in T_L are necessary.

The above relationship can be compactly written as

$$G_U \geq ((k-1)(j-1))^{\left\lfloor \frac{(g-2)/4}{2} \right\rfloor}(k-1)^{\left\lfloor \frac{(g-2)}{4} \right\rfloor \bmod 2}$$

Now number of groups needed for T$_U$ are discussed . To avoid cycle shown on the left in Fig. 13.12 , the two bit nodes E and F are divided into different subgroups, as shown on the right in Fig. 3.12. Symmetrically, it can be derived

$$G_L \geq ((k-1)(j-1))^{\left\lfloor \frac{(g-2)/4}{2} \right\rfloor}(j-1)^{\left\lfloor \frac{(g-2)}{4} \right\rfloor \bmod 2}$$

where g is the girth to be achieved and G$_U$ is the minimum number of groups needed for T$_U$.

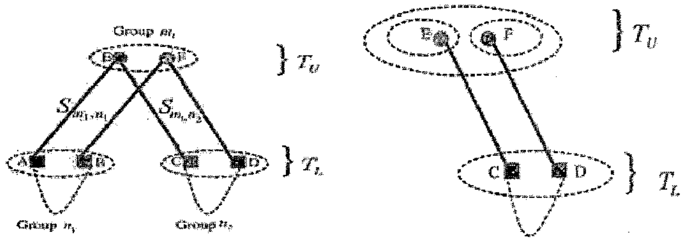


Fig. 3.12 . Left: A type of cycle that can NOT be excluded by choices of shifts S$_{m1,n1}$ and S$_{m1,n2}$ Right: Divide bit nodes E and F into subgroups to avoid cycles.[22]

On the other hand, if the indices of the auxiliary nodes in the same group must have t left most digits in common, G$_U$ and G$_L$ are given by

$$G_U = ((k-1)(j-1))^{\left\lfloor \frac{t}{2} \right\rfloor}(k-1)^{t \bmod 2} \quad (3.17)$$

$$G_L = ((k-1)(j-1))^{\left\lfloor \frac{t}{2} \right\rfloor}(j-1)^{t \bmod 2} \quad (3.18)$$

So to achieve girth g the parameter t that determines the number of groups has to satisfy

$$t \geq \left\lfloor \frac{(g-2)}{4} \right\rfloor \quad (3.19)$$

Equation 3.19 determines the minimum value of the parameter t to achieve a girth g.

3.5 Construction of TS LDPC codes

Design of TS LDPC code with column weight j, row weight k and minimum girth g is discussed in subsequent paragraphs. For given values of j and k (or column weight j and code rate r) and desired girth g following steps are followed for construction of TS-LDPC code :

(a) Number of tiers in upper tree T_U and lower tree T_L h=g-2

(b) Design upper tree T_U and calculate total number of bit node and check nodes in upper tree.

(c) Design lower tree T_L and calculate total number of bit node and check nodes in lower tree.

(d) Calculate number of bit nodes and check nodes in interleaver.

(e) Calculate minimum number of groups for bit nodes and check nodes of interleaver. This will give size of shift matrix. Detailed algorithm for design of shift matrix is explained in subsequent paragraphs.

(f) Obtain interleaver matrix by shift matrix and no of elements in each group of shift matrix.

(g) Based on values of T_U, T_L and interleaver calculate block length of coded word (n) , this gives the desired parity check matrix H. parity check matrix is constructed with T_U , T_L and interleaver .

3.5.1 Upper tree (T_U)

Upper tree T_U is upper part in Tanner graph of LDPC codes. It starts with a check node and has bit nodes as its leaf nodes. Each check node is connected with (k-1) number of bit nodes and each bit node is connected with (j-1) number of check nodes. If the number of tiers in upper tree is h, than total number of bit nodes and check nodes in upper tree can be calculated with following expression.

$$\text{Bit nodes} = \sum_{i=1}^{\frac{h}{2}} (k-1)^i (j-1)^{i-1} \tag{3.20}$$

$$\text{Check nodes} = \sum_{i=0}^{\frac{h}{2}-1} (k-1)^i (j-1)^i \tag{3.21}$$

3.5.2 Lower tree (T_L)

Lower tree T_L is lower part in Tanner graph of LDPC codes. It starts with a bit nodes and has check nodes as leaf nodes .Each check node is connected with (k-1) number of bit nodes and each bit node is connected with (j-1) number of check nodes. If the number of tiers in upper tree is h, than total number of bit nodes and check nodes in upper tree can be calculated with following expression.

$$\text{Check nodes} = \sum_{i=1}^{\frac{h}{2}} (j-1)^i (k-1)^{i-1} \quad (3.22)$$

$$\text{Bit nodes} = \sum_{i=0}^{\frac{h}{2}-1} (j-1)^i (k-1)^i \quad (3.23)$$

3.5.3 Interleaver

Total number of bit nodes and check nodes in interleaver are given by following expression

$$\text{Bit nodes} = (k-1)^{h/2} (j-1)^{h/2-1} \quad (3.24)$$

$$\text{Check nodes} = (j-1)^{h/2} (k-1)^{(h/2-1)} \quad (3.25)$$

3.5.4 Shift Matrix S

Minimum Number of groups G_U and G_L in bit nodes and check nodes, to achieve the girth g can be calculated with following expression.

$$G_U \geq ((k-1)(j-1))^{\left\lfloor \frac{(g-2)/4}{2} \right\rfloor} (k-1)^{\left\lfloor \frac{(g-2)}{4} \right\rfloor \bmod 2} \quad (3.26)$$

$$G_L \geq ((k-1)(j-1))^{\left\lfloor \frac{(g-2)/4}{2} \right\rfloor} (j-1)^{\left\lfloor \frac{(g-2)}{4} \right\rfloor \bmod 2} \quad (3.27)$$

3.5.5 Algorithm for Shift Matrix S

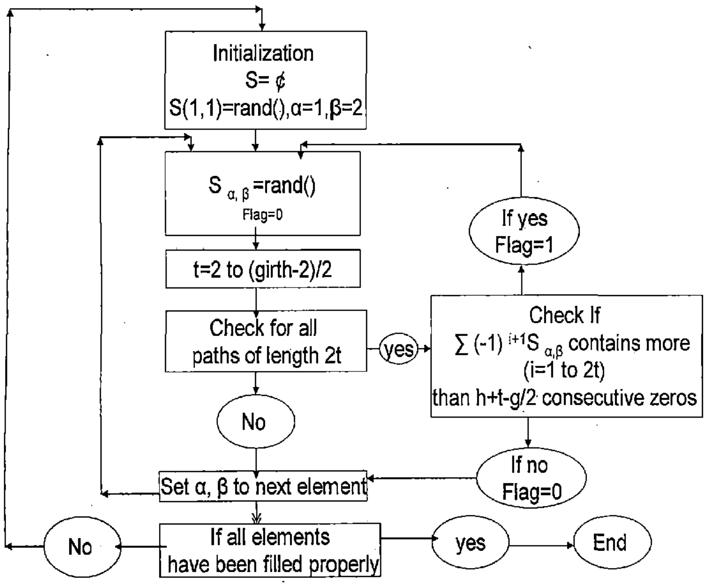
Theorem 3 can be used to reduce the construction of TS-LDPC codes with desired girth g. Designing a matrix S that collects appropriate shifts $S_{\alpha,\beta}$. By choosing suitably these shifts $S_{\alpha,\beta}$, according to Theorem 3, all short type I and type II cycles up to the desired length g-2 can be avoided . An algorithm that finds $S = [S_{\alpha,\beta}]$ for TS-LDPC codes with girth g is discussed in subsequent paragraphs. The matrix S is G_U x G_L, which is much smaller than the TS-LDPC code parity-check matrix H . The algorithm determines shifts $S_{\alpha,\beta}$, one at a time, its value being strongly dependent on the previously determined shifts. Different initial settings of $S_{\alpha,\beta}$ will lead to different matrices . If the algorithm fails to generate a matrix , it is restarted with different initial settings. To construct a TS-LDPC code with column weight j , row weight k , and tier h (number of tiers contained in the upper tree T_U), the number of candidate matrices S is $[(j-1)(K-1)]^{\frac{h}{2}.G_U.G_L}$, which is exponential in the number of groups G_U and G_L. Large girth g may require increasing the number of tiers in the upper and lower trees.

Algorithm 1 TS-LDPC codes with girth g (design shift matrix)

```
Initialization        Set matrix S=Φ, the empty matrix
Determine the elements of the matrix S row by row.
$S_{1,1} \leftarrow$ rand(.) . Set $\alpha = 1$ and $\beta = 2$
Step a.: $S_{\alpha,\beta} \leftarrow$ rand(.) set its flag to 0.
for t=2 to (g-2)/2 do
    for     all closed path of length 2t in the current entries of the shift
      matrix S that   pass the entry $S_{\alpha,\beta}$
      do
       check if     $\sum_{i=1}^{2t}(-1)^{i+1}S_{\alpha i,\beta i}$ contains ore than h+t –(g/2) consecutive zeros
                    { $S_{\alpha 1,\beta 1}, S_{\alpha 2,\beta 2}, \ldots\ldots, S_{\alpha 2t,\beta 2t}$ are the 2t consecutive
              corners of the closed path considered}
       if $\sum_{i=1}^{2t}(-1)^{i+1}S_{\alpha i,\beta i}$   contains more than h+t-(g/2)Consecutive zeros then
                    set the flag to be 1 and stop the for loop;
        else
          keep the flag to 0.
        end if
       end for
    end for
    if        the value of the flag is 1 then
```

```
        discard the current candidate for $S_{\alpha,\beta}$ , go back to set a to select
 another possible vale for $S_{\alpha,\beta}$.
else
        fill the entry($\alpha,\beta$) of S with the current value $S_{\alpha,\beta}$. Set the value of $\alpha$
 and $\beta$ to the next element of S that is to be determined .
        if all the elements of S have already been properly
        chosen then
                go to step b
         else
                go back to step a
          end if,
    end if
 step b: end,    output the shift matrix S
```

The above mentioned algorithm can be presented in flow chart form , as shown below.



Initialization
S= ∅
S(1,1)=rand(),α=1,β=2
S α, β =rand()
Flag=0
t=2 to (girth-2)/2
Check for all paths of length 2t
yes
Check If Σ (-1) i+1S α,β contains more (i=1 to 2t) than h+t-g/2 consecutive zeros
If yes Flag=1
If no Flag=0
No
Set α, β to next element
If all elements have been filled properly
No
yes
End

Fig 3.13: flow chart to design shift matrix

3.5.6 Parity check matrix H

Finally parity check matrix can be build by combining, T_U, T_L and interleaver. Size of the parity check matrix will be given by following expression.

Numbers of rows (n-m) = check nodes of T_U + check nodes of T_L (3.28)

Numbers of columns (n) = bit nodes of T_U + bit nodes of T_L (3.29)

Hence with a given column weight (j), row weight (k) and desired girth g, parity check matrix for Turbo Structured LDPC code can be constructed. As an illustration, above mentioned method can be applied to construct a (6666,3,6) regular LDPC code, with rate r = 0.5 and girth g=10. Its structured is given by 3333 x 6666 parity check matrix H and shown in figure 3.14.

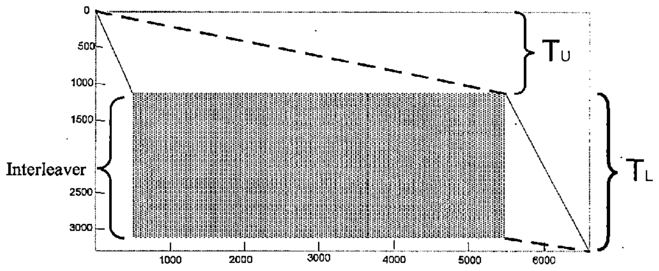


Fig 3.14 Parity-check matrix H for (n, j, k) TS-LDPC code with rate r and girth g (6666,3,6), r = 1/2, g =10.

T_U , T_L and the interleaver component from the constructed matrix can be clearly identified, as labeled in Fig. 3.14. In this matrix, along the solid lines, there is a single 1 in each row, while along the dashed thicker diagonals there are five 1's in each row, so that per row there are six 1's.

3.6 Simulation and Discussion

This section presents simulation results for BER of the regular TS-LDPC codes, which are constructed with arbitrary column weight (j >2) and row weight k and arbitrary girth g. These simulation results are compared with, the BER of randomly constructed regular LDPC of the same size, that is free of 4-cycles [6]. The performance for both the codes is evaluated in additive white Gauss noise (AWGN) channels. The codes are decoded with the sum-product algorithm [6]. The rate normalized SNR is adopted as :

$$\mathrm{SNR} = 10 \log_{10}[E_b / (2r \sigma^2)] \qquad (3.30)$$

where r denotes the code rate. Maximum number of iterations for sum product algorithm (SPA) are kept at 30.

3.6.1 TS-LDPC codes with column weight 3, rate 2/3 and girth g=8

First TS LDPC code for column weight 3 and rate 2/3 is designed. Row weight for rate 2/3 and column weight 3 is 9. Number of tiers in upper and lower tee for girth 8 will be 8-2=6. Calculation for Upper tree, lower tree, interleaver and shift matrix is as follows:

Upper Tree T_U (from equation 3.20 and 3.21)

$$\text{Bit nodes} = \sum_{i=1}^{3} 8^i 2^{i-1} = 2184$$

$$\text{Check nodes} = \sum_{i=0}^{2} 8^i 2^i = 273$$

Interleaver Design (from equation 3.24 and 3.25)

$$\text{Bit nodes} = 8^3 * 2^2 = 2048$$

$$\text{Check nodes} = 2^3 * 8^2 = 512$$

Shift Matrix S (from equation 3.26 and 3.27)

Min G_U =8 Min G_L =2

S-matrix size: 2 X 8

Lower Tree T_L (from equation 3.22 and 3.23)

$$\text{Check nodes} = \sum_{i=1}^{3} 2^i 8^{i-1} = 546$$

$$\text{Bit nodes} = \sum_{i=0}^{2} 2^i 8^{i-1} = 273$$

Final H-matrix

For construction of final parity check matrix H first the elements (α,β) of shift matrix S are calculated (source code for said program is attached as appendix). Elements of shift matrix will give the shift of the respective group. Based on the number of elements in each group interleaver matrix is formed. Finally combination of upper tree T_U , lower tree T_L and interleaver will give parity check matrix.

$$\text{Rows} = 273+546 = 819$$

$$\text{Columns} = 2184+273 = 2457$$

as it is clear that number of columns in parity check matrix are 2457, hence block length of code will be 2457.

Encoding of TS LDPC code with block length 2457

Turbo structured parity check matrix is encoded by constructing generator matrix (G) with the help of parity check matrix. For construction of generator matrix Gauss Jordan method was employed.

Modulation

After encoding the message bits , the encoded word was modulated with BPSK method and same was passed through AWGN channel.

Decoding

At receiving end message is decoded by applying sum product algorithm (as discussed in chapter 2). Maximum number of iteration was kept at 30. BER was calculated for various values of SNR.

Random LDPC code

Random LDPC code of same block length as of TS LDPC code i.e. 2457 and rate 2/3 is constructed [6]. Same message was encoded with random LDPC code and modulated and passed through AWGN channel. At receiver end message was decode with Sum product algorithm, similar to decoding of TS LDPC codes.

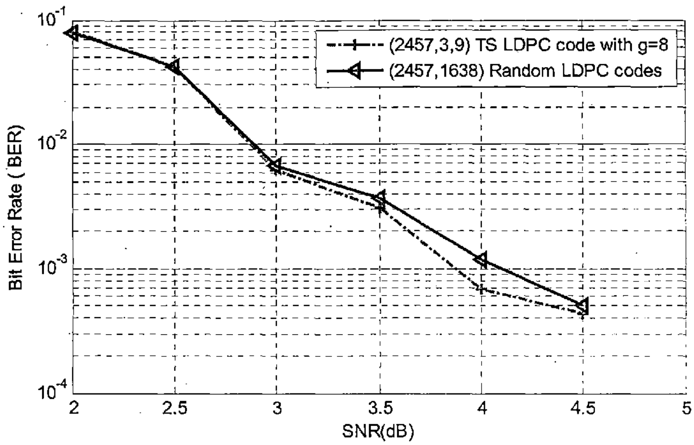


Fig. 3.15 BER performance comparison between a (2457,3,9)TS-LDPC code with girth 8,rate 2/3 and Randomly constructed (2457,1638) LDPC codes

It is clear from BER v/s SNR plot in Fig 3.15 that at lower SNR values performance of TS LDPC code is comparable to Random LDPC codes. In the high SNR region TS-LDPC code outperforms the random code, i.e. at BER 7 x 10^{-6} this gain is apprx 0.3 dB.

3.6.2 (6084,3,12) TS-LDPC codes girth g=8

Next, with keeping the column weight same (j=3) TS LDPC code with rate r=3/4 and girth 8 is constructed. Following calculation shows that similar size random code will be of (6084, 4563) dimension.

Calculated code parameters

T_U = 507 X 5577

T_L = 1014 X 507

Interleaver = 968 X 5324

Min G_U =2 , Min G_L= 11

Size of S matrix = 2 X 11

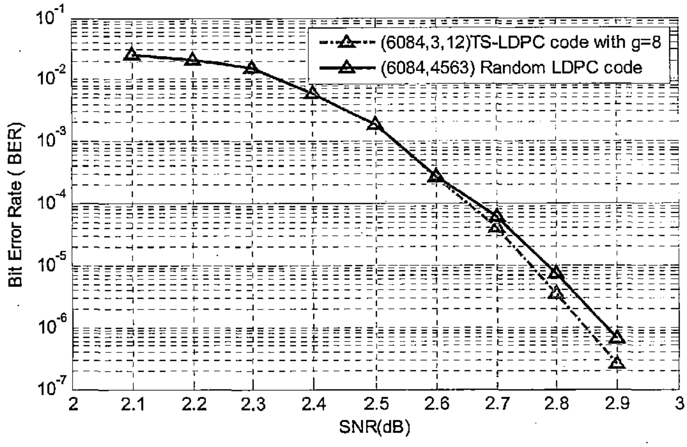


Fig. 3.16 **BER performance comparison between a (6084,3,12)TS-LDPC code with girth 8,rate 3/4 and Randomly constructed (6084,4663) LDPC codes**

It is clear from the above mentioned calculations that if column weight (j) and girth is not changed than to achieve higher code rate , size of parity check matrix or the code block length will be more. It is clear from BER v/s SNR plot in Fig 3.16 BER performance of the TS-LDPC code outperforms that of the random LDPC code at BER < 10^{-5} while at low SNR , both codes have identical error correcting performance.

3.6.3 (6666,3,6) TS-LDPC codes girth g=10

Next, simulations are carried out for TS-LDPC codes while the column weight (j=3) is not changed, but desired girth g is kept at 10. TS LDPC code with rate r=1/2 and girth 10 is constructed. Following calculation shows that similar size random code will be of (6084, 4563) dimension. Calculated code parameters are:

T_U = 1111 X 5555

T_L = 2222 X 1111

Interleaver = 2000 X 5000

Min G_U =10 , Min G_L = 10

Size of S matrix = 10 X 10

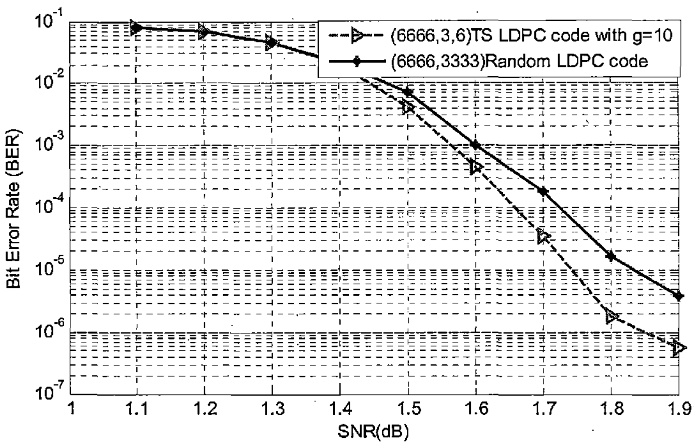


Fig. 3.17 BER performance comparison between a (6666,3,6)TS-LDPC code with girth 10,rate 1/2 and Randomly constructed (6666,3333) LDPC codes

It is clear from BER v/s SNR plot in Fig 3.17 that at lower SNR values performance of TS LDPC code is comparable to Random LDPC codes. In the high SNR region TS-LDPC code outperforms the random code, i.e. at BER 10^{-6} this gain is apprx 0.1 dB. The slope of the BER curve for random LDPC code decreases with the SNR in the high SNR region.

3.6.4 (6220,3,4) TS-LDPC codes girth g=12

Next simulations are carried out for TS-LDPC codes while the column weight (j=3) is not changed, but desired girth g is kept at 12. TS LDPC code with rate r=1/4 and girth 12 is constructed. Following calculation shows that similar size random code will be of (6220,1555) dimension. Calculated code parameters are:

T_U = 1555 X 4665

T_L = 3110 X 1555

Interleaver = 2592 X 3888

Min G_U = 6 , Min G_L = 6

Size of S matrix = 6 X 6

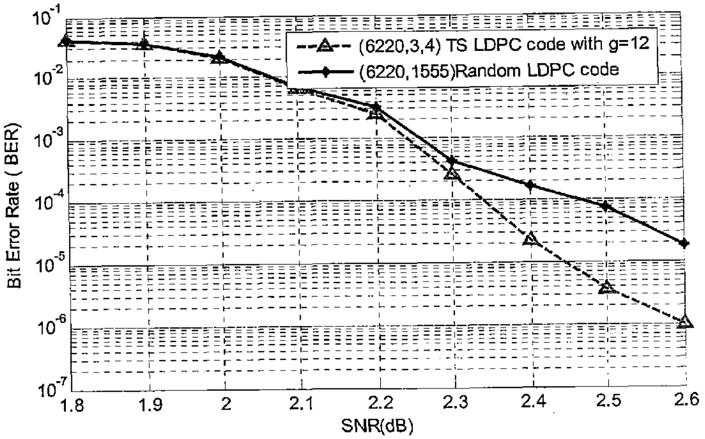


Fig. 3.18 **BER performance comparison between a (62203,4)TS-LDPC code with girth 12, rate 1/4 and Randomly constructed (6220, 1555) LDPC codes**

It is clear from BER v/s SNR plot in Fig 3.18 that at lower SNR values performance of TS LDPC code is comparable to Random LDPC codes. In the high SNR region TS-LDPC code outperforms the random code. At BER $2x10^{-5}$ this gain is 0.2 dB.

3.6.5 (1446, 4, 6) TS-LDPC codes girth g=8

Now simulations are carried out for TS-LDPC codes while the column weight (j=4) is changed and desired girth g is kept at 8. TS LDPC code with rate r=1/3 and girth 12 is constructed. Following calculation shows that similar size random code will be of (1446,482) dimension. Calculated code parameters are:

T_U = 241 X 1205

T_L = 723 X 241

Interleaver = 675 X 1125

Min G_U =3 , Min G_L = 5

Size of S matrix = 3 X 5

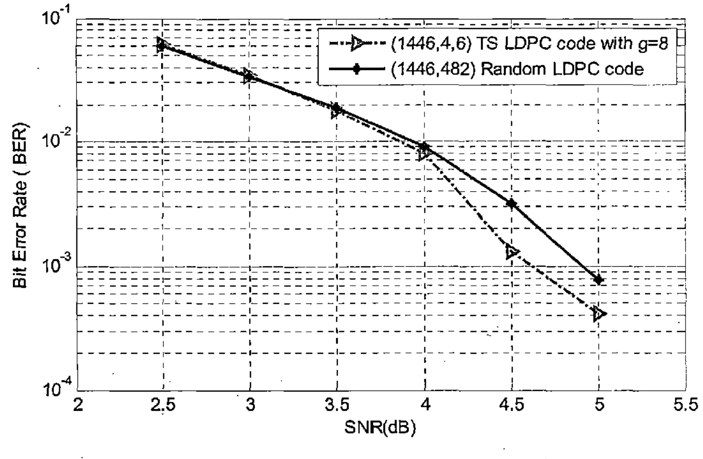


Fig. 3.19 BER performance comparison between a (1446,4,6)TS-LDPC code with girth 8, rate 1/3 and Randomly constructed (1446,482) LDPC codes

It is clear that with smaller block length and lower code rate, high value of SNR is required to achieve better error performance. It is also clear from BER v/s SNR plot in

Fig 3.19 that at lower SNR values performance of TS LDPC code is comparable to Random LDPC codes. In the high SNR region TS-LDPC code outperforms the random code. At BER 1.5×10^{-3} this gain is 0.3 dB.

3.7 Efficient Memory Utilisation

In general, an (n,j,k)-LDPC code is represented by an $m \times n$ parity-check matrix $\mathbf{H}$. Its efficient storage records only the nonzero column indices in each row, hence, at least $n \times j$ indices needs to be stored. In contrast, for a TS-LDPC code we need only to store its small shift generating matrix **S**. For example, for the 3333 x 6666 matrix with girth g=10 shown in Fig 3.14, according to equation 3.15 and 3.16, $G_U = G_L = 10$, and so the shift matrix **S** is 10 x 10 . Hence, instead of storing the 6666 x 3 = 19999 column indices required for generic LDPC codes, TS-LDPC require only storing 10 x 10 = 100 shifts, reducing the memory by a factor of 200. Hence TS-LDPC code not only ensures higher and controlled girth property but results in reduced memory requirements.

3.8 Conclusions

In this chapter design of Turbo Structured LDPC codes are discussed. Turbo Structured LDPC codes with large and controlled girth and flexible code rate guarantees fast convergence of iterative decoding algorithms with reduced memory requirements.

4.1 Introduction

Encoding of LDPC codes can be done similarlily as of linear Block codes. A generator matrix for a code can be found by performing Gauss-Jordan elimination on H to obtain it in the form

$$H = [\ A, I_{n-k}\ ] \tag{4.1}$$

Where A is a (n-k) x k binary matrix and I_{n-k} is size n-k identity matrix. The generator matrix is then

$$G = [\ I_k, A^T\ ] \tag{4.2}$$

However, the drawback of this approach is that, unlike H, the matrix G will most likely not be sparse and so the matrix multiplication

$$c = uG, \tag{4.3}$$

at the encoder will have complexity in the order of n^2 operations. As n is large for LDPC codes, from thousands to hundreds of thousands of bits, the encoder can become prohibitively complex. Hence for LDPC code Encoder should be of liner complexity.

4.2 Encoding friendly TS-LDPC (EFTS-LDPC) Codes

The Tanner graph of an EFTS-LDPC code is derived from Tanner graph of Turbo Structure LDPC. Similar to TS LDPC codes it contains an upper tree T_U and a lower tree T_L that are interconnected by an interleaver ***I***. There are no restrictions on the upper tree T_U of the EFTS-LDPC code, which can be exactly the same as the part of the standard TS-LDPC codes. The T_L of the EF-TSLDPC code is restricted so that the degree of its bit nodes is two. In addition, the root of the T_L is changed from a bit node to a check node, as shown in Fig. 4.1. In addition to design restriction in [22], while constructing lower-tree T_L for EF TS LDPC code, the degree of the root of T_L (check node) should be $(j-1)^{\frac{h}{2}}$ while other check nodes in T_L have uniform degree (k-1) where j, k, h are column weight, row weight, and height of T_L, respectively. One can always build a regular TS LDPC code first, then build a corresponding EF TS LDPC code from the TS LDPC code [22].

The Tanner graph for an EFTS-LDPC code is shown in Fig. 4.1. EFTS-LDPC codes are slightly irregular. These modifications enable EFTS-LDPC codes to be encoded with linear complexity.

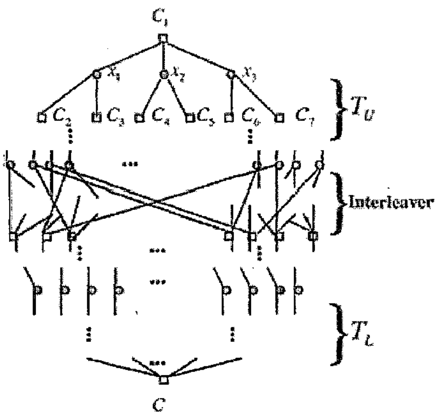


Fig. 4.1. EFTS-LDPC codes: A variant of TS-LDPC codes (EFTS-LDPC codes). [22]

4.3 Linear-Complexity Encoding of TS-LDPC codes

Encoding of EFTS-LDPC codes can be explained using their Tanner graph because LDPC codes are represented by Tanner graph. To achieve linear-complexity encoding, the root (a check node) of the lower tree T_L from the Tanner graph of the EFTS-LDPC codes is removed first. This check node is removed because its degree is $(j-1)^{h/2}$ while other check nodes in T_L have uniform degree (k-1) and number of tiers in T_L will also be same as of T_U. It is shown in the following *Lemma* that removing the root of T_L will not alter the underlying tree structure.

Lemma 4: The parity-check equation denoted by the root of T_L is redundant and can be removed from the parity-check matrix H without changing the underlying code structure.

Proof: As given in [22] two different cases are discussed: the bit node degree of T_U is even; the bit node degree of T_U is odd.

1) The Bit Node Degree j of T_U Is Even: Since all the bit nodes in T_L have uniform degree two by definition, then the degree of all the bit nodes is even, which means that each column of **H** contains an even number of 1's. Hence, the sum of all the

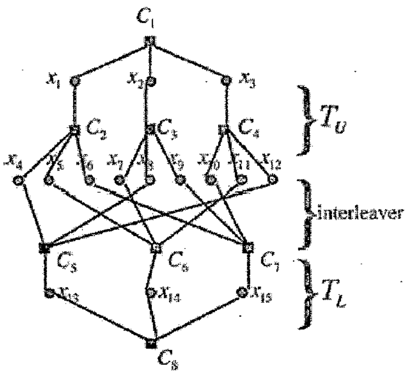


Fig. 4.2. Tanner graph for an EFTS-LDPC code (The bit node degree of T_U is even). [22]

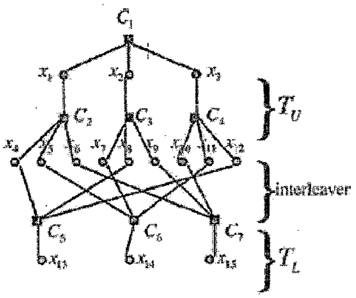


Fig. 4.3. Equivalent representation of the Tanner graph, shown in Fig. 4.2. (The root of T_L is removed.) [22]

rows of **H** in the binary field is a vector of 0's. Therefore, one row of **H** is linearly dependent on the remaining rows and can be removed without affecting the code. Hence one row can be removed without affecting the properties of **H**, for construction of encoding friendly TS LDC code. Best choice is the row that corresponds to the root of

T_L. For example, Fig. 4.2 shows an EFTS-LDPC code. The bit node degree of its T_U is two, an even number. The root of its T_L can be removed to generate an equivalent Tanner graph, as shown in Fig. 4.3

2) The Bit Node Degree j of T_U is Odd: By construction, check nodes in T_L connect to either leaf nodes of T_U or bit nodes of T_L . Since each leaf node of T_U is connected to (j-1) check nodes in T_L and is an odd number, then each leaf node of T_U is connected to an even number of check nodes in T_L . Further, every bit node in T_L is connected to exactly two check nodes in T_L by construction. Hence, every bit node is connected to an even number of check nodes in T_L . If we sum up those parity-check equations denoted by the check nodes in T_L , the summation in the binary field is a vector of 0's. Therefore, one of those parity-check equations in T_L can be removed without changing the underlying code structure. Again the parity-check equation denoted by the root of T_L is chosen for removal.

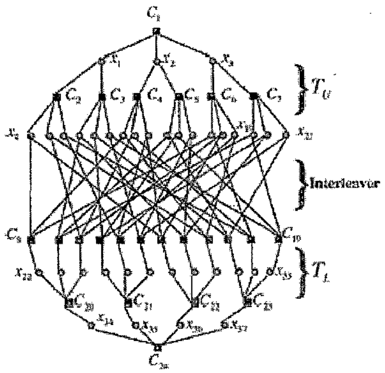


Fig. 4.4. Tanner graph for an EFTS-LDPC code. (The bit node degree of T_U is odd.) [22]

For example, Fig. 4.4 shows an EFTS-LDPC code. The bit node degree of T_U is 3 , an odd number. The root of T_L can be removed from its Tanner graph without changing the code, as shown in Fig. 4.5.

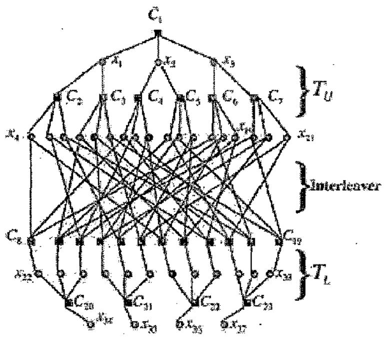


Fig. 4.5. An equivalent representation of the Tanner graph shown in Fig.4.4. (The root of T_L is removed.) [22]

4.4 Encoding algorithm for EFTS-LDPC Codes

Encoding for Encoding Friendly Turbo Structured (EF-TS) LDPC code is done step by step, first upper tree T_U is encoded than lower tree T_U will be encoded based on the information provided by interleaver.

After encoding T_U, it is noticed that all the bit nodes in represent parity bits (Fig 4.5). Let h represent the number of tiers in T_L. Since the degree of every bit node x in T_U is two, the value of x depends only on the values of the bit nodes in the lower tier. Particularly, the values of the bit nodes in tier h-1 of T_L are based only on the values of the leaf nodes of T_U . Hence, the values of the bits in T_L can be computed tier by tier, starting from the bottom tier. Each time the values all the bits in a given tier, say, the $(2i)^{th}$ tier are computed, the values of all the bits in the $(2i-1)^{th}$ tier can be computed . This encoding process keeps going on until the values of all the bits the second tier are known (the first tier has been removed by *Lemma 4*). In this way, all the bits in T_L are encoded. An algorithm for encoding EF-TS LDPC codes is presented below.

Algorithm 2 Encoding algorithm for Ef-TS-LDPC codes (T_U contains h tiers)

```
Initialization
Encode $T_U$ , get the values of all the bit nodes in $T_U$, including the leaf nodes;
Compute values of the bit nodes in tier h of $T_L$ based on the values of the
leaf-nodes of $T_U$ ;
for i=h-3 to step-2 do
        Compute values of the bit nodes in tier i of $T_L$ based on the values of bit nodes in
        tier i+2 of $T_L$ ;
end for
Output the encoding result.
```

4.4.1 EFTS-LDPC Codes Encoding: Example

Encoding with the help of EF-LDPC codes can be understood with an example EFTS-LDPC code for (37, 3, 4). Number of tiers in upper and lower tree are 4. It can be calculated that dimension of upper tree, lower tree and interleaver will be 7 x 21 , 16 x 16 and 12 x 18 respectively. Fig 4.6 shows the upper tree T_U .

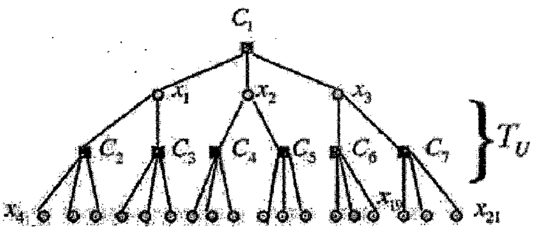


Fig 4.6: Upper Tree T_U of the EFTS-LDPC code

Upper Tree T_U can be encoded as follows:-

1. First acquire the values of the information bits x_2, x_3, x_5, x_6, x_8, x_9, x_{11}, x_{12}, x_{14}, x_{15}, x_{17}, x_{18}, x_{20} and x_{21}.

2. Then compute the parity bit x_1 from the parity check equation

   C1: x1=x2 $\oplus$ x3.

3. Compute the parity bits x_4, x_7, x_{10}, x_{13}, x_{16} and x_{19} from parity check equations as follows :

   C2: x4 = x1 $\oplus$ x5 $\oplus$ x6
   C3: x7 = x1 $\oplus$ x8 $\oplus$ x9
   C4: x10= x2 $\oplus$ x11 $\oplus$ x12
   C5: x13= x2 $\oplus$ x14 $\oplus$ x15
   C6: x16= x3 $\oplus$ x17 $\oplus$ x18
   C7: x19= x3 $\oplus$ x20 $\oplus$ x21

   Hence the finally encoded bits are $x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8,x_9,x_{10},x_{11},x_{12},x_{13},x_{14},x_{15},x_{16},x_{17},x_{18},x_{19},x_{20},x_{21}$.

4. The complexity of above encoding process is only 7*2-1=13 XOR operations.

For encoding of lower tree T_L refer fig 4.7. Steps are as follows

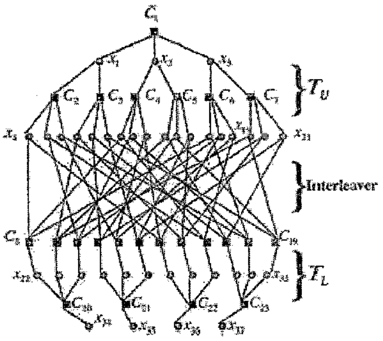


Fig 4.7: Encoding of lower tree T_L of the EFTS-LDPC code

1. Compute the parity bits x_{22} to x_{33} from the check equation C_8 to C_{19} using Parity check equations of interleaver and leaf node bits of upper tree.

2. Compute the parity bits x_{33}, x_{34}, x_{35} and x_{36}, x_{37}

C20: x34= x22 $\oplus$ x23 $\oplus$ x24

C21: x35= x25 $\oplus$ x2 6 $\oplus$ x27

C22: x36= x28 $\oplus$ x29 $\oplus$ x30

C23: x37= x31 $\oplus$ x32 $\oplus$ x33

Hence final encoded word will be combination of upper tree encoded bits x_1 to x_{21} and lower tree bits x_{22} to x_{37}.

4.5 Computational complexity

Let us evaluate the computational complexity of algorithm. Let k_ℓ, $\ell = 1, 2 \ldots m$ denote the number of bits contained in the ℓ^{th} parity-check equation. Each of the m parity-check equations is used to obtain the value of a parity bit. When employing the ℓ^{th} parity-check equation to determine the value of a parity bit, (k_ℓ-2) XOR operations are needed. So, $\sum_{l=1}^{m}(k_l-2)\,\mathrm{XOR}$ operations are required to obtain all m parity bits. Let $\bar{k}=\frac{1}{m}\sum_{l=1}^{m}k_l$ denote the average number of bits in the m parity-check equations and then the encoding complexity can be expressed as $O(m(\bar{k}-2)$. For LDPC codes with uniform row weight k, the encoding complexity is $O(m(k-2)$, hence it is clear that the encoding process of Encoding Friendly Turbo Structured (EF-TS) LDPC codes can be accomplished in linear time.

4.6 Simulation and discussion

This section presents simulation results for BER of the Encoding Friendly TS-LDPC codes , which are constructed with arbitrary column weight (j >2) and row weight k and arbitrary girth g. The performance for the code is evaluated in additive white Gauss noise (AWGN) channels. The codes are decoded with the sum–product algorithm [6]. SNR is calculated as follows ;

$$\mathrm{SNR} = 10 \log_{10}[E_b / (2r\, \sigma^2)] \qquad (4.4)$$

where r denotes the code rate. Maximum number of iterations for sum product algorithm (SPA) are kept at 30.

4.6.1 (8051,3,6) EF TS-LDPC codes girth g=10

One can always build a regular TS LDPC code first, and then build a corresponding EF TS LDPC code from the TS LDPC code. For example, let us build EF TS LDPC code from the TS LDPC code of (6666,3,6). As per the calculations given chapter 3. number of check nodes in interleaver are 2000. Hence for tier h of T_L, number of check nodes = 2000, for tier h-2, check nodes = (2000/5) = 400, for tier h-4 check nodes = 80, for tier h-6 check nodes = 16, for tier h-7, there are 16 bit nodes, for tier h-8, there is only one check node that connects to all the $(3-1)^{\frac{8}{2}}$ =16 bit nodes in tier h-7. The node in tier h-8 is the root. Hence, the degree of the root of T_L is different from the degree of the other check nodes in T_L. This root is removed for construction of EF TS-LDPC codes, as discussed previously. Now upper tree, lower tree and shift matrix are obtained similar to TS LDPC codes. Combination of these three matrices will give parity check matrix.

Calculated code parameters are

T_U = 1111 X 5555

T_L = 2496 X 2496

Interleaver = 2000 X 5000

Min G_U =10 , Min G_L = 10

Size of S matrix = 10 X 10

It is clear from above parameters that block length of EF TS LDPC code will be 5555 + 2496 = 8051.

Message is encoded with upper tree first and than lower tree, as explained in algorithm (source code is attached as appendix).

Encoded message is modulated (BPSK) and passed through AWGN channel.

At receiver end message is decoded with Sum Product algorithm. Maximum number of iteration are kept at 30.

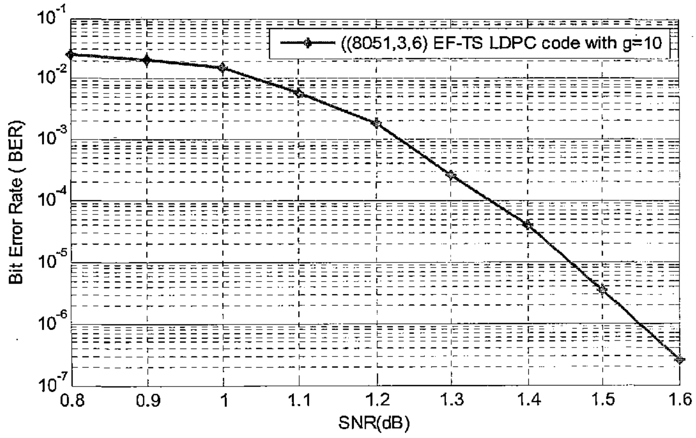


Fig. 4.8 BER performance comparison between a (8051,3,6) EF-TS-LDPC code with girth 10.

It is clear from the calculations that Encoding Friendly Turbo Structured (EF-TS) LDPC code will have larger dimension than corresponding TS-LDPC code. Since this code has larger block length (8051) and is irregular LDPC code, BER of 2 x 10^{-7} is achieved at 1.6 dB SNR only (fig 4.8).

4.6.2 (1993, 3, 8) TS-LDPC codes girth g= 8

Here EF-TS-LDPC code corresponding to (1688, 3, 8) TS-LDPC code with girth 8 is constructed.

T_U = 211 X 1477

T_L = 456 X 456

Interleaver = 392 X 1372

Min G_U = 2 , Min G_L = 7

Size of S matrix = 2 X 7

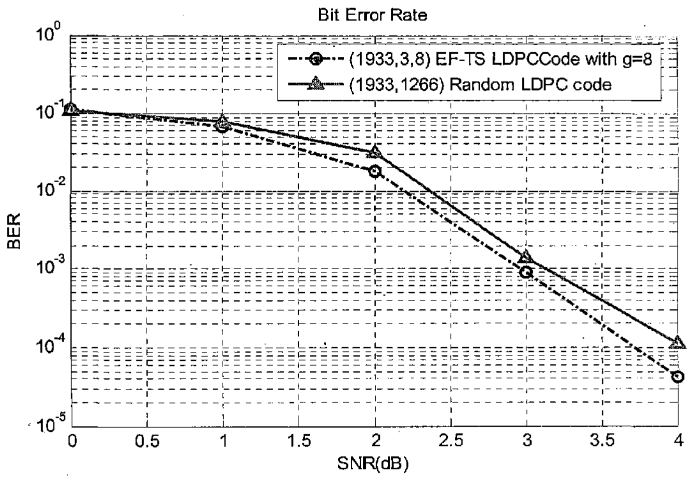


Fig.4.9 BER performance comparison between a (1933,3,8) EF-TS-LDPC code with girth 8 and Randomly constructed (1993,1266) LDPC codes

In Figure 4.9 we compare the performance of (19333,8) EF-TS-LDPC codes with girth g=8 and the random LDPC code (1993,1266) free of length 4 cycles. Simulation results demonstrate that simulation time taken for encoding the EF-TS LDPC code is much - much lower than the Random LDPC code. Random LDPC codes are encoded with the help of creation of Generation matrix (G) with Gauss-Jordan elimination on H. It is also clear that EF-TS LDPC code outperforms random LDPC code at higher SNR value. For Low SNR both these codes have similar error correcting performance.

4.7. Conclusions

In this chapter design and construction of EFTS-LDPC codes with linear complexity encoding are discussed. A regular TS-LDPC code is developed first, and than its corresponding EFTS-LDPC code is constructed. Decoding is done using sum product algorithm.

CONCLUSION AND FUTURE WORK CHAPTER 5

LDPC codes have received much attention in recent years. The advantages of the codes include capacity-approaching performance, and highly-efficient parallel decoding algorithms. LDPC codes are enabling technology for many new applications.

In this dissertation a class of well structured regular LDPC codes is discussed. In the first part of dissertation turbo design for LDPC codes is considered. It is shown through a series of theorems that Turbo Structured (TS) LDPC code can be designed with arbitrary desired column and row weights j and k, hence with any practical rate r and arbitrary girth g. TS-LDPC codes can be designed by specifying a shift matrix S, a much smaller object than the parity check matrix H,. This results in requirement of much less memory to store them. TS-LDPC codes with girth g ≥ 8 have good BER performance than equivalent size and rate random LDPC codes.

In the second part of dissertation a variant of Turbo Structured (TS) LDPC codes- Encoding Friendly (EF) TS-LDPC code is designed. It is shown that with few restrictions and modifications in lower tree T_L of TS-LDPC code, EFTS-LDPC codes can be constructed. One can always build a regular TS LDPC code first, and then build a corresponding EF TS LDPC code from the TS LDPC code. EFTS-LDPC codes encode efficiently in liner complexity.

These characteristics of flexible code rates, arbitrary large girth, good error floor performance, efficient storage, and efficient encoding make TS-LDPC codes attractive for applications such as digital communication systems and data storage systems.

This dissertation decodes Turbo Structured LDPC codes with Sum Product Algorithm (SPA). This work can be further extended by developing a turbo like decoding algorithm which converges faster than sum product algorithm. This decoding method can utilize turbo structure of parity check matrix.

GLOSSARY

Tanner Graph

The Tanner graph is a special type of graph, a bipartite graph, where the nodes divide into two disjoint classes with edges only between nodes in the two different classes.

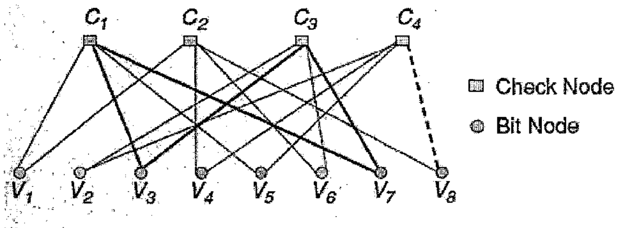


Fig A.1: Tanner graph

Girth

Let $G = \{(V, E)\}$ be a graph, where V is a set of vertices or nodes V and E is a set of edges E connecting the vertices. The degree of a node V is the number of edges incident on V. In an undirected graph, a series of successive edges forming a continuous curve passing from one vertex to another is called a chain. A chain of a node where the initial and the terminal nodes are the same and that does not use the
same edge more than once is a cycle. The length of a cycle is the number of its edges; and the girth g of G is the length of the shortest cycle.

Signal to Noise Ratio

Signal-to-noise ratio (SNR or S/N) is defined as the ratio of a signal power to the noise power corrupting the signal.

Bit Error Rate (BER)

Bit Error Rate is defined as the probability of error per bit in a digital communications system.

Parity Bits

A bit in asynchronous communications used to indicate the type of error checking used in a transmission. A bit appended to the bit pattern for a character so that the number of bits in the pattern and parity bit combined is either even or odd, said to be even or odd parity respectively.

Code Rate

In error correction, the ratio of information divided by information plus redundancy. Codes with a high code rate are desirable because they efficiently use the available channel for information transmission.

Row Weight

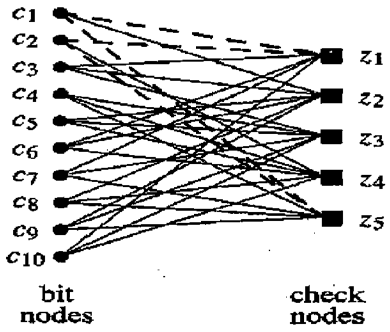
Row weight is the number of ones in one row of the parity check matrix of LDPC codes.

Column Weight

Column weight is the number of ones in one column of the parity check matrix of LDPC codes.

Bit Node and Check nodes

Associated with a parity check matrix H is a graph called the Tanner graph containing two sets of nodes. The first set consists of N nodes which represent the N bits of a codeword; nodes in this set are called "bit" nodes. The second set consists of M nodes, called "check" nodes, representing the parity constraints. The graph has an edge between the nth bit node and the mth check node if and only if nth bit is



1.

Fig A.2: Bit Nodes and Check Nodes

Auxiliary Nodes

By construction, each leaf-node in T_U is connected to $q = j - 1$ leaf-nodes in T_L. This is a one-to-q mapping, while the standard interleaver is a one-to-one mapping between elements of two sets with the same size. To get a standard interleaver, we introduce "auxiliary nodes" (solid triangles) as shown in Figure to facilitate the code design. For each leaf-node in T_U, we add $j-1$ auxiliary nodes as its children. Similarly, each leaf-node in T_L has $k - 1$ auxiliary nodes as its descendants.

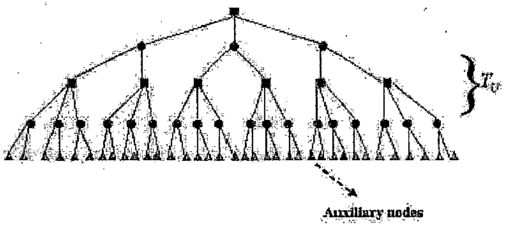


Fig A.3: Auxiliary nodes in upper tree

References

1 C. E. Shannon, "A mathematical theory of communication," *Bell System Tech. J.*,vol. 27, pp. 623–656, July 1948.

2. S. Lin and D. J. Costello Jr., *Error Control Coding*, 2nd ed. New Jersey: Prentice Hall, 2004.

3. R.G. Gallager, *Low-Density Parity Check Codes.* Cambridge, MA: MIT Press, 1963.

4. Gallager, R. G., "Low-density parity-check codes," *IRE Trans. Inf. Theory*, vol. IT-8, no. 1, pp. 21–28, January 1962.

5. Berrou, C., Glavieux, A. and Thitimajshima, P., "Near Shannon limit error-correcting coding and decoding: turbo codes," *Proc. 1993 IEEE International Conference on Communications*, Geneva, Switzerland, vol. 2, pp. 1064–1070, May 1993.

6. MacKay, D. J. C. and Neal, R. M., "Near Shannon limit performance of low density parity check codes," *Electron. Lett.*, vol. 33, no. 6, March 13, 1997.

7. S. Y. Chung, G. D. Forney, T. J. Richardson, and R. L. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," *IEEE Commun. Letters*, vol. 5, pp. 58–60, Feb. 2001.

8. A. J. Blanksby and C. J. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder," *IEEE Journal of Solid-State Circuits*, vol. 37, pp. 404–412, Mar. 2002.

9. R. M. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Inf. Theory*, vol. IT-27, no. 5, pp. 533–547, Sep. 1981.

10. Y. Kou, S. Lin, and M. P. C. Fossorier, "Low-density parity-check codes based on finite geometries: A rediscovery and new results," *IEEE Trans. Inf. Theory*, vol. 47, no. 7, pp. 2711–2736, Nov. 2001.

11. ——, "Low density parity check codes: Construction based on finite geometries," in *Proc. IEEE Globecom 2000*, San Francisco, CA, Nov. 2000, vol. 2, pp. 825–829.

12. J. H. Dinitz and D. R. Stinson, "A brief introduction to design theory," in *Contemporary Design Theory: A Collection of Surveys*, J. H. Dinitz and D. R. Stinson, Eds. New York: Wiley, 1992, ch. 1, pp. 1–12.

13. D. J. C. MacKay and M. C. Davey, "Evaluation of Gallager codes for short block length and high rate applications, in codes," in *Systems and Graphical Models; IMA Volumes in Mathematics and its Applications*, B. Marcus and J. Rosenthal, Eds. New York: Springer-Verlag, 2000, vol. 123, pp. 113–130.

14. R. L. Townsend and E. J.Weldon, "Self-orthogonal quasi-cyclic code," *IEEE Trans. Inf. Theory*, vol. IT-13, no. 2, pp. 183–195, Apr. 1967.

15. M. P. C. Fossorier, "Quasi-cyclic low-density parity-check codes from circulant permutation matrices," *IEEE Trans. Inf. Theory*, vol. 50, no. 8, pp. 1788–1793, Aug. 2004.

16. X. Y. Hu, E. Eleftheriou, and D. M. Arnold, "Progressive edge-growth tanner graphs," in *Proc. IEEE Globecom 2001*, San Antonio, TX, Nov.2001, pp. 995–1001.

17. F.R. Kschischang, B.J. Frey, and H.A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Trans. Inform. Theory,* vol. 47, no. 2, pp. 498–519, Feb. 2001.

18. T. Etzion, A. Trachtenberg, and A. Vardy, "Which codes have cycle-free Tanner graphs?," *IEEE Trans. Inform. Theory,* vol. 45, no. 6, pp. 2173–2181, Sept. 1999.

19. J Lu and J. M. F. Moura, "Turbo design for LDPC codes with large girth," in *IEEE Int. Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, Rome, Italy, Jun. 2003, pp. 90–94.

20. L. Ping and K. Y. Wu, "Concatenated tree codes: A low-complexity, high-performance approach," *IEEE Trans. Inf. Theory*, vol. 47, no. 2,pp. 791–799, Feb. 2001.

21. H. Behairy and S.-C. Chang, "Parallel concatenated Gallager codes for CDMA applications," in *Proc. IEEE Globecom 2001*, San Antonio, TX, Nov. 2006, vol. 2, pp. 1002–1006.

22. Jin Lu and M. F. Moura, , "TS-LDPC codes:Turbo Structured Codes With Large Girth," *IEEE* Transaction on Information Theory, vol53,No3, Mar2007 pp 1080-1094.
23. S. Dolinar and D. Divsalar, Weight Distribution for Turbo Codes Using Random and Nonrandom Permutations, JPL, Pasadena, CA, Progr. Rep. 42-122, Aug. 1995, pp. 56–65.
24. P. Popovski, L. Kocarev, and A. Risteski, "Design of flexible-length s-random interleaver for turbo codes," *IEEE Commun. Lett.*, vol. 8, no. 7, pp. 461–463, Jul. 2004.
25. Jin Lu, Jose M.F. Moura and Urs Niesen, " A Class of structured LDPC codes with Large Girth", IEEE Communication Sociaty2004, pp-425-429

```
% Generation of upper tree

function [H]=vingenerate_tu_new(k,j,check,bit)
a=zeros(check,bit);
%t=0;
 for m=1:check

                    if m~=1
                     p=1+floor((m-2)/(j-1))
                    a(m,p)=1;
                    end;
     for n=1:bit
            if (((k-1)*(m-1)< n)&& (n<=(k-1)*m))
                a(m,n)=1;
            end;
               % if (t>0)
                %    a(m,t)=1;
               % end;

     end;


 end;
 H=a;
```

% Generation of lower tree

```
function [H]=vingenerate_tl_new(k,j,check,bit,inter)
a=zeros(check,bit);
t=1;
 for m=1:check
                 p=floor((m+(j-2))/(j-1))
                    a(m,p)=1;

 end;

  for t=inter+1:check

      for n=1:bit

        if (((k-1)*(t-(inter+1))< n)&& (n<=(k-1)*(t-inter)))
                a(t,n)=1;

        end;
      end;
  end;
 H=a;
```

% Generation of Shift matrix

```
function [S]= generate_S_matrix_new(m,n,k,j,p_group,girth)

             h_tier=girth-2;
             a=zeros(m,n);
             flag =zeros(m,n);
             iterations=0;


%NEXT2:
jump1=1;
while jump1==1
       jump1=0;
       iterations=iterations+1;
        for i=1:m
         for jj=1:n

                                a(i,jj)=-1;
                                   flag(i,jj)=1;

         end
        end

        check1=zeros(1,p_group);
   for i=1:p_group
       check1(i)=i-1;
   end

        ran1=randperm(n);
        ran2=randperm(m);
        for i=1:k-1

            a(1,ran1(i))=gen_random1(p_group);
        end

        for i=1:j-1
            a(ran2(i),1)=gen_random1(p_group);
        end
        %a(1,:)=randsrc(1,n,check1);
        %a(:,1)=randsrc( 1,m,check1);

        randd=zeros(1,p_group);
        g=1;
    for i=2:m

    for jj=2:n

        jump2=1;
        checkrow=0;
                    checkcol=0;
                    for u=1:n
                        if a(i,u)>=0
                            checkrow=checkrow+1;
                        end
                    end
                    for u=1:m
                        if a(u,jj)>=0
                            checkcol=checkcol+1;
                        end
                    end



                if  checkrow>=(k-1)
                    jump2=0;

                end

                if  ~(checkcol>=(j-1))

                    %NEXT1:

                    while jump2==1

                           jump2=0;

                          a(i,jj)=gen_rand_cond(p_group,randd,g);
                            flag(i,jj)=0;
                            for t=2:(girth-2)/2
                  if
check_loop_matrix(a,m,n,n,k,j,p_group,2*t,h_tier,girth)


flag(i,jj)=1;%discard the current candidate of a(i,j)

randd(g+1)=a(i,jj);

  g=g+1;
  if g<p_group
  jump2=1;
  end
                  end
                            end


                    if jump2~=1
                            if(g>=p_group)
                          %a[i][jj]=-1;
                            %goto NEXT2;
                            jump1=1;
                            end
                            g=1;
                    end
                    end
                    end
    end
    end
```

```
end
%rearrange
checkcol=0;
                for u=1:m
                        if a(u,jj)>=0
                            checkcol=checkcol+1;
                        end
                    end
for i=1:n


               checkcol=0;
                for u=1:m
                        if a(u,i)>=0
                            checkcol=checkcol+1;
                        end
                end
       if checkcol<(j-1)&&a(m,i)<0
           a(m,i)=gen_random(p_group);
       end

end

S=a;
```

% various functions utilized for shift matrix program

```
function num=gen_random(range)
sum=randperm(range);
num=sum(1)-1;


function num= gen_rand_cond(p,a,n)


         z=randperm(p);
        jump=1;
        nu=0;
    if n>0
    while jump==1
     % NEXT:
     jump=0;
      for i=1:n
             if (z(1)-1)==a[i]

                        z=randperm(p);
                        %goto (NEXT,'gen_rand_cond.m');
                        jump=1;
             end
             if jump~=1

             num=z(1)-1;
             jump=0;
             return;
```

```
              end
         end
       end

        else
            num=z(1)-1;
            return;
       end


function bool =check_loop_matrix(a,m,n,p,k_row,j_col,p_group,cycle_length,h_tier,girth)
        %matrix d
        d=zeros(m,n);

   %matrix b i.e lists of connection b/w nodes
     b=zeros(m*p,m+n);

   %calculate b's elements
         kk=1;
         for i=1:m
           for j=1:n
                 if a(i,j)>=0
                 %v[kk]=kk;
                 d(i,j)=kk;
                 kk=kk+1;
                 else
                 d(i,j)=-1;
                 end
           end
         end
           w=1;

          for h=1:m
              for r=1:n
                    if d(h,r)>0
                      s=1;
                       for j=1:n
                          if d(h,j)>=0&&j~=r
                           b(w,s)=d(h,j);
                           s=s+1;
                          end
                       end

                       for j=1:m
                          if d(j,r)>=0&&j~=h
                          b(w,s)=d(j,r);
                          s=s+1;
                          end
                       end
                        w=w+1;
                    end
              end
          end

        x=m*p;
     count=0;

     for i=1:m
      for j=1:n
          if a(i,j)>=0
              count=count+1;
          end
      end
     end

       comb=zeros(1,count);
       for i=1:count
           comb(i)=i;
       end

     %check cycle
     for cycle=4:cycle_length
         allcomb=nchoosek(comb,cycle);
         size1=size(allcomb);
         for per=1:size1(1)

             permu=perms(allcomb(per,:));
             y=cycle;
             size2=size(permu);
             for z=1:size2(1)
                  v=permu(z,:);
                  if check_loop(v,cycle,b,x,m+n)
                          jump=0;
                          for r=1:y-2
                                   x1=0;x2=0;x3=0;y1=0;y2=0;y3=0;
                                  for s1=1:m
                                      for s2=1:n

                                          if d(s1,s2)==v(r)
                                               x1=s1;
                                               y1=s2;
                                             else if d(s1,s2)==v(r+1)
                                               x2=s1;
                                               y2=s2;
                                                 else
                                                     if
d(s1,s2)==v(r+2)

                                                     x3=s1;
                                                     y3=s2;
                                                     end
                                                  end
                                                 end
                                          end

if(x1==x2&&x2==x3)||(y1==y2&&y2==y3)
                                              %goto NEXT;
                                              jump=1;
                                              break;
                                             end
                                  end
```

```
                        r=y-1;

                    if r==y-1&&jump~=1
                       x1=0;x2=0;x3=0;y1=0;y2=0;y3=0;
                         for s1=1:m
                         for s2=1:n
                                  if d(s1,s2)==v(r)
                                    x1=s1;
                                    y1=s2;
                                  else if d(s1,s2)==v(r+1)
                                      x2=s1;
                                      y2=s2;
                                      else
                                          if d(s1,s2)==v(1)
                                           x3=s1;
                                           y3=s2;
                                          end
                                       end
                                      end
                              end
                             if
(x1==x2&&x2==x3)||(y1==y2&&y2==y3)
                            %goto NEXT;
                            jump=1;
                             end
                         end

                      r=y;
                     if r==y&&jump~=1
                        x1=0;x2=0;x3=0;y1=0;y2=0;y3=0;
                        for s1=1:m
                         for s2=1:n
                                 if d(s1,s2)==v(r)
                                    x1=s1;
                                    y1=s2;
                                  else if d(s1,s2)==v(1)
                                   x2=s1;y2=s2;
                                      else
                                          if d(s1,s2)==v(2)
                                          x3=s1;y3=s2;
                                          end
                                       end
                                      end
                                 end

if(x1==x2&&x2==x3)||(y1==y2&&y2==y3)
                                    % goto NEXT;
                                    jump=1;
                                     end
                        end
                        %printv(v,cycle);
                        if jump~=1

                                 path_values=zeros(1,y);

path_values=return_path_values(a,m,n,v,y) ;
```

```
                                          sum=0;
                                          for mm=1:y
                                                            if mod(mm-
1,2)
                                                            sum=sum-
path_values(mm);
sum=mod(sum,p_group);
                                                            else

sum=sum+path_values(mm);
sum=mod(sum,p_group);
                                                            end
                                          end
                                             p_q=zeros(1,h_tier);

                                                p_q
=p_q_alternate(h_tier,sum,j_col,k_row);

                                                if
check_zeros(p_q,h_tier,h_tier+y-girth/2)
                                                    bool =1;

                                                return ;
                                                end

                                  end
                                end
                             end

                                      end
                                  end
                          end
                  end
             end

             bool=0;


function bool= check_loop(v,n,b,p,q)

     for i=1:n-1
        if ~check(b,p,q,v(i),v(i+1))
        bool=0;
        return;
        end
     end

     if ~check(b,p,q,v(1),v(n))
     bool=0;
     return;
     end

     bool=1;


function [W]= return_path_values(a,m,n,v,s)

          d=zeros(m,n);

          kk=1;
      for i=1:m
      for j=1:n
              if a(i,j)>=0
              d(i,j)=kk;
              kk=kk+1;
              else
              d(i,j)=-1;
              end
      end
      end

         w1=s;
         w=zeros(1,w1);
            for k=1:w1
            for i=1:m
            for j=1:n
                            if d(i,j)==v(k)
                            w(k)=a(i,j);
                            end
            end
            end
            end


function [A] =p_q_alternate( h,x, j, k)
a=zeros(1,h);
   p=k-1;
   q=j-1;
  i=1;
while x>1
     a(i)=mod(x,q);
     x=x/q;
    i=i+1;
   a(i)=mod(x,p);
   x=x/p;
    i=i+1;
end
A=a;

function bool= check_zeros(a,h,zeros)

        count=0;
        for i=1:h
                        if a(i)==0

                                                count=count+1;
                                                if count==zeros
                                                    bool=1;
                                                return ;
                                                end


          else
              count=0;
                      end
        end
              if count>=zeros
                  bool=1;
              return ;
              end
              bool=0;
```

% GENERATION interleaver MATRIX FROM GIVEN SHIFT MATRIX...

```
function [h]= s_hmatrix(w,p)
%w=generate_S_matrix(4,6,3,2,3,4,6)
y=size(w);
rs=y(1);
cs=y(2);
h=zeros(rs*p,cs*p);
        s=w;

      for i=1:rs
       for  j=1:cs
                    if (s(i,j)==-1)
                                for z=i*p+1-p:i*p
                                 for  x=j*p+1-p:j*p
                                  h(z,x)=0;
                                 end
                                end
                    else
                          q=s(i,j);
                          for  x=j*p+1-p:j*p
                          for z=i*p+1-p:i*p
                             h(z,x)=0;
                                 if (x+q+p*(i-1)>i*p)
                                    v=(x+q+p*(i-1));
                                       while v >i*p
                                       v= v-p;
                                       end;
                                       if z==v
                                       h(z,x)=1;
```

```
                                                   end;
                                            end
                                            if x+q+p*(i-1)==z
                                            h(z,x)=1;
                                            end
                                     end
                                     end
                               end
              end
            end
            %lowertree(h,ch,rh);
      end
```

% Generation of parity check matrix

```
function [H]=final_h_matrix(tu,tl,inter)
  size1=size(tu);
  size2=size(tl);
  size3=size(inter);
  row=size1(1)+size2(1);
  col=size1(2)+size2(2);
  h=zeros(row,col);
  for i=1:size1(1)
      for j=1:size1(2)
          h(i,j)=tu(i,j);
      end
  end

   for i=1:size2(1)
      for j=1:size2(2)
          h(i+size1(1),j+size1(2))=tl(i,j);
      end
   end

   for i=1:size3(1)
      for j=1:size3(2)
          h(i+size1(1),j+size1(2)-size3(2))=inter(i,j);
      end
   end
H=h;
```

% Encoding of message with EF TS LDPC code

```
function [h,en]=encode_tu(k,j,m)
[a,b]=size(m);
col=b+b/(k-2);
row=b/(k-2);
tu=vingenerate_tu_new(k,j,row,col);
e=zeros(1,col);
n=1;

for i=1:col
    if(rem(i-1,k-2))
      e(i)=m(n);
      n=n+1;
    end
end

  count=1;
  for i=1:row
        xor=0;
        for s=1:n
            if(tu(i,s))
                xor=mod(xor+e(s),2);
            end
        end
        e(count)=xor;
        count=count+k-2;
  end


  h=tu;
  en=e;
```

% Example program for generation of Turbo structured parity check matrix for (2457, 3, 9) code with girth 8.

```
function [ber1] = PLOT_TS_LDPC_2457
n=2457;
k=9;
j=3;
g=8;
row=819;col=2457;
%upper tree 273 X 2184
tur=273;
tuc=2184;
%lower tree 546X273

tlr=546;
tlc=273;
%interleaver 512 X 2048
ir=512;
ic=2048;
```

```
%grouping of 10
group=256;
%girth=8
g=8;
%shift matrix
sr=2;
sc=8;
%generate shift matrix 2X5
s=generate_S_matrix(sr,sc,k,j,group,g);
hs=s_hmatrix(s,group);
tu=vingenerate_tu_new(k,j,tur,tuc);
tl=vingenerate_tl_new(k,j,tlr,tlc,ir);
H=final_h_matrix(tu,tl,hs);

H(1,row)=1;
%generate messase bits

m=zeros(1,col-row);
for i=1:col-row
    if mod(i,2)
    m(i)=1;
    else
        m(i)=0;
    end
end

[G,hh]=gen_generator_matrix2(H);%generate generater matrix
%h=G;
H=hh;
generator=G;
parity_check=H;
save ber_tsldpc_2457 generator parity_check;
 no=NO;
 amp=AMP;
 scale=zeros(1,col);
 for i=1:col
 scale(i)=1;
 end
  c=mul_GF2(m,G);  %tx=c;
  ber11=zeros(1,5);
 for u=1:5
  modu=bpsk1(c,amp);
  rx=awgn(modu,no);
 c_=decode_ldpc(rx,no,amp,H,scale);
 count=0;                % rx=c_;
 for i=1:col
     if c(i)~=c_(i)
         count=count+1;
     end
 end

 ber11(u)=count/col;
 error_rate=count/col
 end
 ber1=ber11;
 semilogy(dB,BER11,'b-o')
title('Bit Error Rate')
ylabel('BER')
xlabel('Eb/No (dB)')
grid
figure
```

% BPSK modulation

```
function [waveform]=bpsk1(bitseq,amplitude)

for i=1:length(bitseq)
   if bitseq(i)==1
      waveform(i)=amplitude;
   else
      waveform(i)=-amplitude;
   end
end
```

% passing through AWGN channel

```
function [x]=awgn(waveform,No);
%x=awgn(waveform,No);
%For examples and more details, please refer to the LDPC toolkit
tutorial at
%http://arun-10.tripod.com/ldpc/ldpc.htm
NoiseStdDev=sqrt(No/2);
x=waveform + NoiseStdDev*randn(1,length(waveform));
```

% Example program for generation of (1933, 3,8) with girth 8 TS LDPC code and Encoding of message with linear complexity

```
function [ber1] = vin_EFTS_LDPC_37

n=1933;
k=8;
j=3;
g=8;
row=667;col=1933;
%upper tree 211x1477
%lower tree 456 x 456
%interleaver 392x1372
%grouping of 196
%girth=8
%generate shift matrix 2 x 7

%generate message
```

```
mn=1477/(k-1);
mn=mn*(k-2);
m=zeros(1,mn);
for i=1:mn
    if mod(i,2)
    m(i)=1;
    else
        m(i)=0;
    end
end
s=generate_S_matrix_new(2,7,8,3,196,8);
hs=s_hmatrix(s,196)
[tu,e1]=encode_tu(k,j,m)
tl=vingenerate_tl_new(k,2,456,456,392)
H=final_h_matrix(tu,tl,hs)
%H(1,col)=1;
e2=zeros(1,456)
e=zeros(1,1933);
for i=1:1477
    e(i)=e1(i);
end


p=1477;
for i=212:667
    xor=0;
    for s=1:1933
        if H(i,s)
            xor=mod(e(s)+xor,2);
        end
    end
    e(p)=xor;
    p=p+1;
end

   v=e;
```