DESIGN AND ANALYSIS OF COMMUNICATION SYSTEMS WITH HIGH ERROR CORRECTION CAPABILITY THROUGH OPTIMIZATION

by

Banu Kabakulak

B.S., Industrial Engineering, Boğaziçi University, 2007

B.S., Mathematics, Boğaziçi University, 2007

M.S., Industrial Engineering, Boğaziçi University, 2010

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Industrial Engineering
Boğaziçi University
2018

ACKNOWLEDGEMENTS

Foremost, I would like to thank my thesis supervisor Z. Caner Taşkın for introducing me to LDPC codes in telecommunications field, and for his great support as an advisor. Not only did he guide me in every step of the thesis process, but he has also been a great mentor. It has been a privilege working with him.

I am also greatful to my thesis supervisory committee members Ali Emre Pusane, Ali Tamer Ünal and İbrahim Muter for their inspiring guidance at every step of the thesis. I have benefited and enlightened much from İ. Kuban Altınel through his extensive academic and personal experiences. I also like to thank to Barış Yıldız who joined them as the final member of my thesis jury. My thesis profited considerably from their insightful and constructive advices.

I would like to thank all my current and former colleagues for the supportive atmosphere they have created at Boğaziçi University Industrial Engineering Department. Especially my lab mates at MMS Lab have been of great assistance: Zeynep Şuvak, Kübra Tanınmış and Betül Ahat. Thank you all!

My special thanks go to my family for their moral support, guidance, patience and love.

I gratefully acknowledge the financial support of TÜBİTAK through 1001 Research Project Program with Grant No. 113M499.

ABSTRACT

DESIGN AND ANALYSIS OF COMMUNICATION SYSTEMS WITH HIGH ERROR CORRECTION CAPABILITY THROUGH OPTIMIZATION

Channel coding is the term used for the collection of techniques that are employed in order to minimize errors which occur during the transmission of digital information from one place to another. Low–density parity–check (LDPC) code family takes attention with its channel capacity–approaching error correction capability and sparse parity–check matrix representation. Sparsity property of the matrix gives rise to the development of heuristic iterative decoding algorithms with low complexity. Ease of the application of iterative decoding algorithms brings the advantage of low decoding latency. In spite of these benefits of LDPC codes, receiver can obtain erroneous information because of both structural properties of LDPC codes and non–optimal decoders.

In the first part of this thesis, we develop optimization–based LDPC decoding algorithms for a communication system with high error performance and we compare its performance with the existing methods in the literature. Error performance of a communication system can still be improved by determining and eliminating small cycles in LDPC codes that cause iterative decoding algorithms to halt or terminate without a conclusive result during the decoding process. At the second place, we implement heuristic and optimization–based approaches for efficiently designing high quality LDPC codes of practically relevant dimensions. We carry out extensive computational experiments to assess the efficiency of proposed methods.

ÖZET

YÜKSEK HATA DÜZELTME YETENEĞİNE SAHİP İLETİŞİM SİSTEMLERİNİN ENİYİLEME YOLUYLA TASARIM VE ANALİZİ

Kanal kodlaması, sayısal bilginin bir yerden başka bir yere iletimi sırasında meydana gelebilecek hataları en aza indirgeyen tekniklerin bütününe verilen isimdir. Düşük–yoğunluklu eşlik–denetim (LDPC) kod ailesi kanal kapasitesine giderek yaklaşan hata düzeltme yeteneği ve seyrek eşlik–denetim matrislerine sahip olması ile dikkat çekmiştir. Matrisin seyreklik özelliği, çok düşük karmaşıklığa sahip olan sezgisel yinelemeli kod çözme algoritmalarının geliştirilmesine olanak vermektedir. Yinelemeli kod çözme algoritmalarının kolaylıkla uygulanabilmesi, düşük kod çözme gecikmesi avantajını da beraberinde getirmektedir. LDPC kodlarının bu faydalarına rağmen, gerek LDPC kodlarının yapısal özellikleri sebebiyle gerekse kod çözücünün hata giderme yeteneğinin yetersizliği sebebiyle alıcı tarafından okunan bilgi hatalar içerebilir.

Bu tezde, ilkin düşük hata ile çalışan bir iletişim sistemi tasarlayabilmek için eniyileme tabanlı LDPC kod çözme algoritmaları geliştirilmiş ve etkinliği literatürdeki yöntemlerle karşılaştırılmıştır. İletişim sisteminin başarımı, LDPC kodlarının yinelemeli kod çözme algoritmalarıyla çözülmesi sırasında algoritmanın ilerleyişinin durmasına veya bir sonuç bulamamasına sebep olan küçük çevrimlerin belirlenmesi ve bertaraf edilmesi durumunda daha da artabilir. İkinci kısmında, gerçek uygulamalarda kullanılabilecek boyutta, yüksek kaliteli LDPC kodlarının hızlı şekilde tasarlanabilmesine olanak veren eniyileme tabanlı LDPC kod tasarım yaklaşımları uygulanmıştır. Geliştirilen yöntemlerin etkinliği kapsamlı bilgisayısal deneylerle sınanmıştır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

C	Set of check nodes
c_j	Check node j
$d_j(d_i)$	Degree of $c_j(v_i)$ in Tanner graph
dc_j	Target degree of c_j
dc_j^s	Slack for degree of c_j
dv_i	Target degree of v_i
dv_i^s	Slack for degree of v_i
f_i	ith bit of the decoded vector
$\mathbf{G}$	Generator matrix
$\mathbf{H}$	Parity–check matrix
h_s	Horizontal step size
k	Length of the original information
k_j	An auxiliary integer variable
m	$n - k$, number of rows in $\mathbf{H}$
m_s	Width of the ribbon of a convolutional code
$N(c_j)(N(v_i))$	Set of variable (check) nodes adjacent to $c_j(v_i)$
n	Length of the encoded information, number of columns in $\mathbf{H}$
p	Error probability in BSC
r	h_s/v_s ratio
T	Target girth
V	Set of variable nodes
v_i	Variable node i
v_s	Vertical step size
w	Height of the window
w_{jS}	1 if local codeword S of c_j is selected, 0 otherwise
X_{ji}	(j, i) entry of the $\mathbf{H}$ matrix
$\hat{\mathbf{y}}$	Received vector

ε_j	Set of feasible local codewords for c_j
γ_i	Log–likelihood ratio for bit i
μ_j	A dual variable for constraints (4.9)
τ_{ij}	A dual variable for constraints (4.10)
ζ_j	Optimum objective function value of Subproblem(j)

LIST OF ACRONYMS/ABBREVIATIONS

ABCW	All Binary Complete Window
ABFW	All Binary Finite Window
ABRW	All Binary Repeating Window
BC	Branch–and–Cut
BCM	Best Combination Model
BER	Bit Error Rate
BP	Branch–and–Price
BSC	Binary Symmetric Channel
CC	Convolutional Code
CP	Constraint Programming
CW	Complete Window
DLPM	Dual Linear Programming Master
EM	Exact Model
EMD	Exact Model Decoder
FW	Finite Window
GFM	Girth Feasibility Model
IP	Integer Programming
IPM	Integer Programming Master
LDPC	Low–Density Parity–Check Code
LDPC CC	LDPC Convolutional Code
LDPC SC	LDPC Spatially–Coupled Code
LEM	Linear Relaxation of EM
LP	Linear Programming
LPM	Linear Programming Master
MDD	Minimum Degree Deviation Model
MDD^r	Relaxed Minimum Degree Deviation Model
PEG	Progressive Edge Growth
RLPM	Restricted Linear Programming Master

RW	Repeating Window
SBCW	Some Binary Complete Window
SBFW	Some Binary Finite Window
SBRW	Some Binary Repeating Window

1. INTRODUCTION

Telecommunication is the transmission of messages from a transmitter to a receiver over a potentially unsafe communication environment. In digital communication systems, code symbols are messages and they are transmitted in the form of electromagnetic radiation. In parallel to the rapid developments in technology, digital communication systems find several application areas: messaging via digital cellular phones, fiber optic internet, TV broadcasting or agricultural monitoring through digital satellites, and receiving high quality images under NASA's Juno and Pluto missions [1, 2] are some examples of digital communication.

In practice, numerous transmitter–receiver pairs use the same communication environment such as air or space. Hence, radio waves, electrical signals, and light waves over fiber optic channels will accumulate some amount of noise on the medium. The noise in the environment can cause transmission errors or failures. Channel coding is the term used for the collection of techniques that are employed in digital communications to ensure a transmission is received with minimal or no errors. These techniques encode the original information by adding redundant bits. When the receiver receives information, decoder estimates the original information by detecting and correcting errors in the received vector with the help of redundant bits.

Among the codes that are used in the decoding process at receiver, low–density parity–check (LDPC) code family has received attention with its high error detection and correction capabilities. LDPC codes were first proposed by Gallager in 1962 and today they are used in wireless network standard (IEEE 802.11n), WiMax (IEEE 802.16e) and digital video broadcasting standard (DVB-S2) [3]. They have sparse parity–check matrices, i.e. $\mathbf{H}$ matrix, and can alternatively be represented by bipartite graphs known as Tanner graphs [4]. Iterative decoding algorithms, which have low complexity and low decoding latency due to the sparsity property of parity–check matrix, are developed on Tanner graph [5, 6]. In spite of these benefits of LDPC codes, receiver can obtain erroneous information because of both structural properties of LDPC codes and non–optimal decoders. In order to overcome these errors, better LDPC codes need to be designed and decoding algorithms need to be improved.

For a communication system with high error performance the main focus in first part of the thesis is development of optimization–based decoding algorithms. Taking into account that the proposed decoding algorithms in the literature are heuristic approaches, one can argue that the inherent error correction capabilities in LDPC codes are not fully utilized by existing decoding algorithms. For this purpose, we model the decoding problem, which can be defined as estimating the original information correctly when the sent and received information is different, as an optimization problem and develop solution methods as given in Chapters 4 and 5.

Error performance of a communication system can still be improved by determining and eliminating small cycles in Tanner graph of an LDPC code that cause iterative decoding algorithms to halt or terminate without a conclusive result during the decoding process. Methods proposed in the literature are either heuristic designs, which do not guarantee the best code design, or optimization–based approaches that do not execute fast enough for real dimensional LDPC codes. One of the focal points of this thesis is to develop optimization–based approaches for efficiently designing high quality LDPC codes of practically relevant dimensions. For this aim, we introduce mathematical models to design LDPC code with given smallest cycle length in its Tanner graph and develop optimization techniques in order to solve the models for practical code lengths in Chapter 6.

In the next section, we summarize the literature related with LDPC decoding and code design. We give some preliminary information about LDPC codes in Chapter 3. We explain our work on LDPC decoding in Chapters 4 and 5. In particular, we introduce a decoder based on a branch–and–price method in Section 4.2 and sliding window decoders for LDPC convolutional codes in Chapter 5. We give the details of our branch–and–cut algorithm to design LDPC codes without small cycles in Chapter 6. We give the computational results of these methods in corresponding sections. We list our concluding remarks and comments on future work in Chapter 7.

2. LITERATURE SURVEY

In this section, we summarize the related literature about LDPC decoding and code design. While discussing the current status of the literature, we aim to state the gaps in the literature that we filled in with this thesis.

Maximum likelihood (ML) decoding is the optimal decoding algorithm in terms of minimizing error probability. Since ML decoding problem is known to be NP–hard, iterative message–passing decoding algorithms for LDPC codes are preferred in practice [7]. However, these heuristic decoding algorithms do not guarantee optimality of decoded vector and they may fail to decode correctly when the graph representing an LDPC code includes cycles. Feldman *et al.* use optimization methods and they develop linear relaxation based maximum likelihood decoding algorithms for LDPC and turbo codes in [8, 9]. However, the proposed models do not allow decoding in an acceptable amount of time for codes with practical lengths.

LDPC convolutional codes, first introduced by Elias in 1955, differ from block codes in that the encoder contains memory and the encoder outputs, at any time unit, depend both on the current inputs and on the previous input blocks [10]. LDPC convolutional codes find application areas such as deep–space and satellite communication starting from early 1970s. LDPC convolutional codes can be decoded with Viterbi algorithm, which provides maximum–likelihood decoding by exaustive search, by dividing the received vector into smaller blocks of bits. Although Viterbi algorithm has a high decoding complexity for convolutional codes with long block lengths, it can easily implemented on hardware due to its highly repetitive nature [11, 12].

For long block lengths, sequential decoding algorithms such as Fano algorithm [13] and later stack algorithm that is developed by Zigangirov [14] and independently by Jelinek [15] fit well. On the contrary to Viterbi algorithm, computational complexity of a sequential decoding algorithm is independent of the block length. While Viterbi algorithm finds the best codeword by enumerating all possibilities exhaustively, sequential decoding is suboptimal since it focuses on a certain number of likely codewords [16].

Being sequentially decodable, LDPC convolutional codes are better than LDPC block codes in encoding for the cases where information is obtained continuously. Although LDPC convolutional codes provide short–delay and low–complexity in decoding, they are not in communication standards such as WiMax and DVB-S2. This is since application–oriented optimization of LDPC convolutional codes is not investigated thoroughly yet [17].

In this thesis, we first consider LDPC codes and propose a branch–and–price decoding algorithm for the mathematical formulation given in [18]. We give the details of the algorithm and the computational results in Section 4.2. Then, we consider LDPC convolutional codes and propose optimization based sliding window decoders that can give a near optimal decoded codeword for a received vector of practical length (approximately $n = 4000$) in an acceptable amount of time. The mathematical formulation and proposed decoding algorithms are explained in Chapter 5. Our proposed decoders can be used in a real–time reliable communication system since they have low decoding latency. Besides, they are applicable in settings such as deep–space communication system due to their high error correction capability.

Iterative decoding algorithm decides on whether the code symbol is 0 or 1 by calculating probabilities for the code symbols and estimate the original information.The calculated probabilities are dependent on each other if there are cycles on the Tanner graph. In order to minimize code symbol estimation errors, designing LDPC codes to maximize the smallest cycle length, i.e. *girth*, is useful. In order to improve a given LDPC code, certain edges are exchanged within Tanner graph to eliminate small cycles without simultaneously creating any others in [19]. A heuristic approach, called Progressive Edge Growth (PEG) which is based on adding edges to the Tanner graph iteratively without constructing small cycles, is given in [20].

Bit–Filling heuristic in [21] starts with a large girth target and decreases target as it inserts the edges to Tanner graph one–by–one. The heuristic terminates when a prescribed girth is met. A randomized approach in [22] can create irregular LDPC codes with high error correction capability. Algebraic properties of $\mathbf{H}$ matrix is considered in [23] to obtain a regular LDPC code. In literature, interleaver methods are proposed for designing Turbo LDPC codes with girth at least 8 [24].

PEG algorithm is adjusted to generate regular LDPC codes in [25] and irregular LDPC codes in [26] for improving the error correction performance. A protograph is a Tanner graph with a relatively small number of nodes. Design of LDPC codes with simple protographs is investigated in [27] to obtain infinite dimensional LDPC codes. Different works in the literature focus on the design of LDPC codes with large girth using the protograph [28, 29].

A method that can build quasi–cyclic LDPC codes with girth at least 6 using Vandermonde matrices is introduced in [30]. In [31], an upper bound on the girth of quasi–cyclic LDPC codes is given. Quasi–cycle constraints are added to PEG algorithm in order to obtain regular and irregular quasi–cyclic LDPC codes in [32]. Other studies also use PEG algorithm for this code family [33] – [35]. For the same code family, a lifting method is given in [36] and generalized polygones are used in [37]. Patent [38] describes a method for quasi–cyclic LDPC codes without stopping sets and guarantees the girth is at least 8. The authors use their results to design hierarchical quasi–cyclic LDPC codes [39]. Independent tree–based heuristic of [40] can iteratively construct regular LDPC codes whose girth values are better than the ones obtained by PEG. The common point of these methods in the literature is that they are heuristic methods without an optimality guarantee.

In this thesis, we propose an integer programming formulation to generate LDPC codes with a given girth value and develop a branch–and–cut algorithm for its solution in Chapter 6. We investigate structural properties of the problem to improve our algorithm by applying a variable fixing scheme, adding valid inequalities and utilizing an initial solution generation heuristic. Our computational results indicate that our proposed methods significantly improve solvability of the problem. To the best of our knowledge, our work is the first in the literature that investigates the LDPC code design problem from an optimization point of view.

3. LDPC CODES PRELIMINARIES

Transmitter sends information to receiver through communication channel in a digital communication system. Communication channel is common for many transmitter–receiver pairs in use which creates noise in the environment. Transmission of the information is affected from the noise, which may result in lost or value change of some information bits. In coding theory, encoding original information improves transmission security [41].

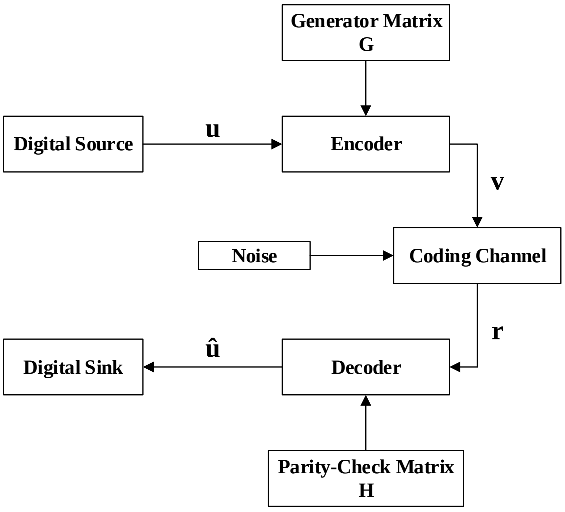


Figure 3.1. Digital communication system diagram.

Figure 3.1 shows information flow in a digital communication system. In Figure 3.1, let the original information be a binary vector $\mathbf{u} = (u_1 u_2 ... u_k)$ of k–bits, i.e. $u_i \in \{0, 1\}$. Encoder adds redundant parity–check bits to vector $\mathbf{u}$ by utilizing a $k \times n$ generator matrix $\mathbf{G}$. That is codeword $\mathbf{v} = (v_1 v_2 ... v_n)$ of n–bits, where $n \geq k$ and $v_i \in \{0, 1\}$, is obtained through operation $\mathbf{v} = \mathbf{uG}$. In a codeword $\mathbf{v}$, there are k information bits and $(n - k)$ parity–check bits, which are used to test whether there are errors in the transmission. For integrity of the communication, codeword $\mathbf{v}$ should be in the null space of $\mathbf{H}$ matrix, i.e. $\mathbf{vH}^{\mathrm{T}} = \mathbf{0}$ (mod 2) holds.

After transmission, receiver gets vector **r** of n–bits as shown in Figure 3.1. Decoder detects whether the received vector **r** includes errors or not by checking the expression $\mathbf{rH}^{\mathrm{T}}$ is equal to vector $\mathbf{0}$ in (mod 2) or not. In the case **r** is erroneous, decoder attempts to determine the error locations and fix them [42]. As a result, the information **u** sent from the source is estimated as $\hat{\mathbf{u}}$ at the sink. In the literature, there are models for noisy channels on which the information is transmitted. Among these, in this work the main focus will be on the binary symmetric channels (BSC). As shown in Figure 3.2, in BSC an error occurs with probability p and the transmitted bit flips. The transmission is completed without any errors with probability $1 - p$ [43].

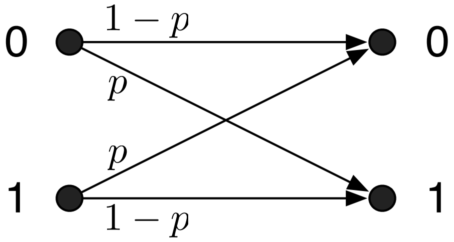


Figure 3.2. Binary symmetric channel.

(n, k) LDPC codes are members of linear block codes that can be represented by a parity–check matrix **H** of dimension $(n - k) \times n$. The only difference of LDPC codes from linear block codes is that **H** matrix of LDPC code is sparse, i.e. the number of ones at every row and column of **H** matrix is forced to be very small. The common property of the codes in (3, 6)–regular LDPC code family is that **H** matrix has only 3 ones at each column and 6 ones at each row independent from the dimension of **H**. This means, for (3, 6)–regular LDPC code with dimension 1500×3000, only % 0.2 of the matrix elements are nonzero. The expression “regular” in the name of the code family means there is a constant number of ones at each row and column of the matrix. An example, (10, 5) LDPC code that is (3, 6)–regular given below.

One can obtain a $k \times n$ generator matrix **G**, which is not necessarily unique, from $(n - k) \times n$ parity-check matrix **H** by carrying out binary arithmetic. Vectors **v** that satisfy the equation $\mathbf{vH}^{\mathrm{T}} = \mathbf{0}$ (mod 2) are codewords. One can observe that each row of generator matrix **G** is a codeword, since $\mathbf{GH}^{\mathrm{T}} = \mathbf{0}$ (mod 2) holds for any $(\mathbf{G}, \mathbf{H})$

$$\mathbf{H} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Figure 3.3. A parity–check matrix from (10, 5) LDPC code family.

pair. From geometrical point of view, the codewords are in the null space of **H** matrix and **G** matrix constitutes a basis for the null space. For any original information **u**, encoded vector $\mathbf{v} = \mathbf{uG}$ (mod 2) is a codeword, since $(\mathbf{uG})\mathbf{H}^{\mathrm{T}} = \mathbf{0}$ (mod 2) is satisfied. The channel decoder concludes that whether the received codeword **r** has changed or not by checking the value of expression $\mathbf{rH}^{\mathrm{T}}$ is equal to vector **0** in (mod 2) or not [44].

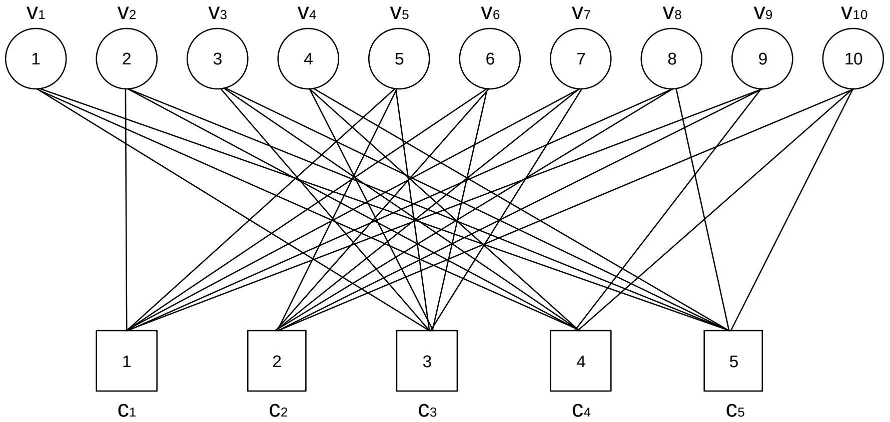


Figure 3.4. Tanner graph representation of the parity–check matrix given in Figure 3.3.

An LDPC code can alternatively be represented as Tanner graph, which is a sparse bipartite graph, corresponding to **H** matrix [4]. On one part of Tanner graph there is a variable node i (v_i), $i \in \{1, ..., n\}$, for each bit of received vector. Each row of **H** matrix represents a parity–check equation and corresponds to a check node j $(c_j), j \in \{1, ..., n-k\}$, in the other part of Tanner graph. A check node is said to be satisfied if its parity–check equation is equal to zero in (mod 2). The set of adjacent check (variable) nodes to a variable node i (check node j) is represented by $N(v_i)(N(c_j))$. The *degree* of v_i (c_j) is the number of adjacent check nodes (variable nodes) on Tanner graph. That is degree of v_i is $d_i = |N(v_i)|$ and c_j is $d_j = |N(c_j)|$. Hence, **H** matrix is the adjacency matrix of Tanner graph. This representation of LDPC codes is practical due to the advantage of applying iterative decoding algorithms easily. Figure 3.4 shows Tanner graph representation of **H** matrix defined in Figure 3.3.

```
Input: Received vector, $\hat{\mathbf{y}}$
1. Calculate all parity–check equations
2. If all check nodes are satisfied, Then STOP.
3. Else Calculate the number of all unsatisfied parity–check
        equations for each received bit, $u_i$ for bit $i$.
4 - A.   Let $l = \text{argmax}_i\{u_i\}$. If $u_l > d_l/2$, Then flip bit $l$.
4 - B.   If $u_i > d_i/2$, Then flip bit $i$.
5. End If
6. If stopping is satisfied, Then STOP.
7. Else Go to Step 1.
8. End If
Output: A feasible decoded codeword, or no solution.
```

Figure 3.5. Gallager A and B algorithms.

We can generate a regular **H** matrix by permuting identity matrices. In permutation codes, a $(3, 6)-$ regular code with codelength $n = 6 \times s$ can be obtained by randomly shuffling the columns of a $s \times s$ identity matrix and putting 6 of them next to each other. We have a $(3, 6)-$ regular code with $m = 3 \times s$ many rows and $n = 6 \times s$ many columns by repeating this process 2 more times and putting each submatrix down to each other.

The sparsity property of LDPC codes allows to apply iterative decoding algorithms, such as Gallager A and B given in Figure 3.5, with low complexity [42]. In Gallager A and B, v_i is incident to d_i many check nodes on Tanner graph and u_i many of them are unsatisfied. A bit i is candidate to be flipped, if $u_i > d_i/2$. At each iteration, Gallager A flips only a candidate bit i with largest u_i value. On the other hand, in Gallager B, all candidate bits are flipped. Gallager A guarantees to decrease the number of unsatisfied check nodes, since it flips only one bit at each iteration. This is not for sure in Gallager B due to multiple flipping at an iteration. In the following chapters, we explain the details of methods that we develop for LDPC codes.

4. LDPC DECODING WITH HIGH ERROR CORRECTION CAPABILITY

4.1. Introduction

In this chapter, we summarize our work on designing a decoding algorithm with high error correction capability for LDPC codes. We give the details of our Branch–and–Price (BP) algorithm in Section 4.2. We consider to improve the performance of BP algorithm with some existing and developed feasible solution generation techniques in Section 4.3.

The decoding problem can be represented with Exact Model (EM) which is given in [18]. The columns and rows of a $(n-k) \times n$ parity–check matrix $\mathbf{H}$ of a binary linear code can be represented with index sets $V = \{1, ..., n\}$ and $C = \{1, ..., n-k\}$, respectively. In EM, H_{ji} is the $(j,i)-$entry of parity–check matrix $\mathbf{H}$, f_i is a binary variable denoting the value of the ith code bit and k_j is an integer variable. Here, $\hat{\mathbf{y}}$ is the received vector.

Exact Model (EM):

$$\min \sum_{i:\hat{y}_i=1} (1-f_i) + \sum_{i:\hat{y}_i=0} f_i \tag{4.1}$$

s.t.

$$\sum_{i \in V} H_{ji} f_i = 2k_j, \ \forall j \in C \tag{4.2}$$

$$f_i \in \{0,1\}, \ \forall i \in V, \tag{4.3}$$

$$k_j \geq 0, \ k_j \in \mathbb{Z}, \ \forall j \in C. \tag{4.4}$$

Constraints (4.2) guarantee that the decoded vector $\mathbf{f}$ satisfies the equality $\mathbf{f}\mathbf{H}^{\mathrm{T}} = \mathbf{0} \pmod 2$. The objective (4.1) minimizes the Hamming distance between the decoded vector $\mathbf{f}$ and the received vector $\hat{\mathbf{y}}$. That is, the aim is to find the nearest codeword to the received vector. Constraints (4.3) and (4.4) set the binary and integrality restrictions on decision variables $\mathbf{f}$ and $\mathbf{k}$, respectively.

An alternative objective function is log–likelihood objective which can be given as

$$\min \sum_{i \in V} \gamma_i f_i. \tag{4.5}$$

Here, γ_i, as given in equation (4.6), is a term that represents the error probability for received bit i. In this equation, $\hat{y}_i$ represents the received value of bit i and f_i is the decoded value of the bit i.

$$\gamma_i = log(\tfrac{Pr(\hat{y}_i|f_i=0)}{Pr(\hat{y}_i|f_i=1)}) \tag{4.6}$$

The linear relaxation of EM (LEM) can be obtained by replacing the constraints (4.3) and (4.4) with the followings:

$$0 \leq f_i \leq 1, \; k_j \geq 0, \; \forall i \in V, \; j \in C. \tag{4.7}$$

Since EM is an integer programming formulation, it is not practical to obtain an optimal decoding using commercial solver for real–sized LDPC codes. Hence, we develop branch–and–price algorithm and sliding window decoders explained in the following sections for decoding problem.

4.2. Branch–and–Price Algorithm

In this section, we introduce a branch–and–price (BP) algorithm for the integer programming formulation in [45] in order to find the nearest codeword to the received vector $\hat{\mathbf{y}}$.

Integer Programming Master (IPM) formulation given in [45] is a maximum likelihood decoder utilizing Tanner graph representation of $\mathbf{H}$ matrix. A *local codeword* can be formed by assigning a value in $\{0,\ 1\}$ to each variable node $i \in N(c_j)$ that is adjacent to c_j. A local codeword is *feasible* if sum of the values of variable nodes $i \in N(c_j)$ is zero in (mod 2). For a check node c_j, the set of feasible local codewords can be given as $\varepsilon_j := \{S \subseteq N(c_j) : |S|\ even\}$. We can satisfy c_j if we set each bit in S to 1, and all other bits in $N(c_j)$ to 0. One can observe that $S = \emptyset$ trivially satisfies a check node and $\emptyset \in \varepsilon_j$ for all c_j.

Integer Programming Master (IPM):

$$\min \sum_{i \in V} \gamma_i f_i \tag{4.8}$$

s.t.

$$\sum_{S \in \varepsilon_j} w_{jS} = 1,\ \forall j \in C \tag{4.9}$$

$$f_i - \sum_{S \in \varepsilon_j, i \in S} w_{jS} = 0,\ \forall \text{ edges } (i,j) \tag{4.10}$$

$$f_i \geq 0,\ \forall i \in V,\quad w_{jS} \in \{0,1\},\ \forall j \in C, \forall S \in \varepsilon_j. \tag{4.11}$$

In IPM model, binary decision variable w_{jS} takes value 1 if feasible local codeword $S \in \varepsilon_j$ of check node c_j is selected and zero otherwise. Hence, decision variables $\mathbf{w}$ represent a feasible solution of parity–check equations and f_i variable represents the decoded value of bit i. We can obtain a trivial solution of IPM with $w_{j\emptyset} = 1$ for all

$j \in C$ and $f_i = 0$ for all $i \in V$. We obtain Linear Programming Master (LPM) model by relaxing the constraints (4.11) as

$$f_i \geq 0,\ \forall i \in V,\quad w_{jS} \geq 0,\ \forall j \in C, \forall S \in \varepsilon_j. \tag{4.12}$$

We define dual variables μ_j for constraints (4.9) and τ_{ij} for constraints (4.10) in LPM and obtain Dual LPM (DLPM) model.

Dual LPM (DLPM):

$$\max \sum_{j \in C} \mu_j \tag{4.13}$$

$$\text{s.t.}$$

$$\sum_{i \in S} \tau_{ij} \geq \mu_j,\ \forall j \in C,\ S \in \varepsilon_j \tag{4.14}$$

$$\sum_{j \in N(v_i)} \tau_{ij} \leq \gamma_i,\ \forall i \in V \tag{4.15}$$

$$\mu_j \text{ free},\ \forall j \in C,\quad \tau_{ij} \text{ free},\ \forall \text{ edges } (i,j). \tag{4.16}$$

We consider a Restricted LPM (RLPM) that has limited number of columns corresponding to w_{jS} variables. At each iteration of our column generation algorithm, we search for columns corresponding to variables w_{jS} that have positive reduced cost, i.e. $\mu_j - \sum_{i \in S} \tau_{ij} > 0$, and add them to RLPM. Such w_{jS} columns are equivalent to the violated constraints from constraints (4.14) in DLPM. If $\zeta_j = \max\{\mu_j - \sum_{i \in S} \tau_{ij} : S \in \varepsilon_j\} > 0$ for some j, then we add the column $\begin{bmatrix} 0 \\ e_j \\ A_k \end{bmatrix}$ for variable w_{jS}. Here, e_j is a $m-$column vector, that has a 1 at jth row and 0 otherwise, and A_k is a $(\sum_{i=1}^{n} d_i)-$column vector which has -1 at kth row if kth edge is the edge (i,j) with $i \in S$. If $\zeta_j = 0\ \ \forall j$, then we are at optimum solution of LPM.

The above discussion means, at each iteration of column generation algorithm, we are trying to solve the following subproblem for each j:

Subproblem(j):

$$\min \sum_{i \in N(c_j)} \tau_{ij} x_i - \mu_j \tag{4.17}$$

s.t.

$$\sum_{i \in N(c_j)} x_i = 2k, \tag{4.18}$$

$$x_i \in \{0,1\}, k \in \mathbb{Z}^+. \tag{4.19}$$

We can solve the jth subproblem with algorithm given in Figure 4.1. The algorithm runs in $\mathcal{O}(n \log n)$ time due to sorting step where n is the number of variable nodes.

```
Input $\tau_{ij}$ values
1. Sort the $\tau_{ij}$ values in nondecreasing order.
   Let $\tau_{ij}^t$ be the $t$th smallest $\tau_{ij}$ value.
2. Set $x_i = 0 \;\; \forall i \in N(c_j)$, set $t = 1$.
3. If $\tau_{i_1,j}^t + \tau_{i_2,j}^{t+1} < 0$, Then set $x_{i_1} = x_{i_2} = 1$, Else STOP.
4. $t \leftarrow t + 2$, go to Step 3.
Output Subproblem($j$) is solved.
```

Figure 4.1. Subproblem j solution algorithm.

As we mentioned before, $w_{j\emptyset} = 1$ for all $j \in C$ is a feasible solution for LPM. Hence, for all $j \in C$ we can take $(j, \emptyset)$ columns for the starting RLPM problem. We can solve LPM to optimality by introducing columns to RLPM until we have $\zeta_j = 0$ for all j. Since our ultimate goal is to solve IPM, we need to branch on decision variables if optimum solution of LPM is fractional. In the next section we discuss the alternative branching strategies in detail.

4.2.1. Branching in BP Algorithm

If we have a fractional optimal solution of LPM, they we have either w_{jS} or f_i variables fractional. Before determining a branching strategy, we will first prove the following proposition.

Proposition 4.1. *In LPM problem, f_i values are integral $\forall i$ if and only if w_{jS} values are integral $\forall (j, S)$.*

Proof. ($\Leftarrow$) Assume that w_{jS} values are integral $\forall (j, S)$. Constraints (4.10) imply that f_i values are integral $\forall i$, since each f_i is the sum of integer numbers. Besides, we observe that w_{jS} values can be either 0 or 1, so do the f_i values.

($\Rightarrow$) Assume for contradiction f_i integral but $\exists j$ such that w_{jS} values are not integral $\forall S$. By constraints (4.9), we know $\sum_{S \in \varepsilon_j} w_{jS} = 1$. Hence, for at least two w_{jS} variables, say $w_{j,S_1} = p$ and $w_{j,S_2} = q$ with $p, q > 0$ and $p + q \leq 1$, we have fractional values. Since $S_1 \neq S_2$, there exists $k \in S_2 \setminus S_1$.

For variable node k and check node j, we have the constraint $f_k = \sum_{S \in \varepsilon_j, k \in S} w_{jS}$ for edge (k, j). Edge (k, j) exists, since $k \in S_2 \in \varepsilon_j$ which implies that $k \in N(c_j)$. We know that $k \notin S_1$, meaning that $w_{j,S_1} = p$ will not be in the sum. This means $f_k = \sum_{S \in \varepsilon_j, k \in S} w_{j,S} \leq 1 - w_{j,S_1} = 1 - p < 1$. Moreover, w_{j,S_2} will be in the sum, since $k \in S_2$. This gives $f_k \geq w_{j,S_2} = q > 0$. As a result, $0 < f_k < 1$ and f_k is a fractional value. This contradicts with our assumption that f_i values are all integral. Hence, we conclude that if f_i integral $\forall i$, then w_{jS} values are also integral $\forall (j, S)$.
Combining two results, we see that f_i values are integral $\forall i$ if and only if w_{jS} values are integral $\forall (j, S)$. $\square$

As a result of this proposition, in order to have an integral solution to the LPM problem, we should either branch on $w_{j,S}$ variables to have integral w_{jS} values or branch on f_i variables to have integral f_i values. Having integral w_{jS} values (or integral f_i values) will guarantee that all decision variables are integral.

4.2.1.1. Branching on w_{jS} variables. In this strategy, we consider to branch on some fractional w_{jS} at a node. This means we have in one branch $w_{jS} = 0$ and $w_{jS} = 1$ in the other branch. In $w_{jS} = 0$ branch, we never select local codeword S for check node j. Let $\mathbf{y} \in \mathbb{B}^n$ be the characteristic vector of S, i.e. $y_i = 1$ if node v_i is in S and $y_i = 0$ otherwise.

Subproblem(j_0)*:*

$$\zeta_j^0 = \min \sum_{i \in N(c_j)} \tau_{ij} x_i - \mu_j \tag{4.20}$$

s.t.

$$\sum_{i \in N(c_j)} x_i = 2k, \tag{4.21}$$

$$|x_i - y_i| = z_i, \ i \in N(c_j) \tag{4.22}$$

$$\sum_{i \in N(c_j)} z_i \geq 1, \tag{4.23}$$

$$x_i \in \{0, 1\}, k \in \mathbb{Z}^+, z_i \in \mathbb{R}. \tag{4.24}$$

Constraints (4.22) and (4.23) guarantee that the selected local codeword characterized by vector $\mathbf{x}$ is different from the prohibited local codeword characterized by vector $\mathbf{y}$ at least in one neighbor v_i. We observe that as we proceed with the branching process, for c_j if we set $w_{j,S_1} = w_{j,S_2} = ... = w_{j,S_r} = 0$ as branch condition, we add the following constraints (4.25) and (4.26) instead of constraints (4.22) and (4.23) to subproblem j:

$$|x_i - y_i^k| = z_i^k, \ i \in N(c_j); \ k = 1, ..., r, \tag{4.25}$$

$$\sum_{i \in N(c_j)} z_i^k \geq 1, \ k = 1, ..., r. \tag{4.26}$$

In $w_{jS} = 1$ branch, we always select local codeword S for c_j. Hence, we satisfy the constraints (4.9) as equality for c_j. This means, we cannot select any other local codeword for c_j. Hence, we solved subproblem j for c_j. For the other check nodes $j' \neq j$, subproblem j' can be solved with the following formulation, where $\mathbf{y}$ is the characteristic vector of set S:

Subproblem(j'_1)*:*

$$\zeta^1_{j'} = \min \sum_{i \in N(c_{j'})} \tau_{ij'} x_i - \mu_{j'} \tag{4.27}$$

$$\text{s.t.}$$

$$\sum_{i \in N(c_{j'})} x_i = 2k, \tag{4.28}$$

$$x_i = y_i, \; i \in N(c_{j'}) \cap N(c_j) \tag{4.29}$$

$$x_i \in \{0, 1\}, k \in \mathbb{Z}^+. \tag{4.30}$$

As we proceed with the branching process, if we have set $w_{j_1,S_1} = w_{j_2,S_2} = ... = w_{j_r,S_r} = 1$ as branch condition, we say that for subproblem j_k the selected local codeword is S_k for $k = 1, 2, ..., r$. For a subproblem $j \notin \{j_1, j_2, ..., j_r\}$, we can solve the subproblem by replacing constraints (4.29) with the following constraints where $\mathbf{y}^k$ is the characteristic vector of set S_k for $k = 1, 2, ..., r$.

$$x_i = y^k_i, \; i \in N(c_{j'}) \cap N(c_{j_k}); \; k = 1, ..., r. \tag{4.31}$$

In more general case, for check nodes $j \in C_1 \subseteq C$ we may have $w_{j,S_1} = w_{j,S_2} = ... = w_{j,S_{r_j}} = 0$ for and for check nodes $j \in C_2 \subseteq C$ we may have $w_{j,S^j} = 1$. Then, we solve subproblem j by selecting S^j as local codeword for $j \in C_2$. For $j \in C_1$, in order to solve subproblem j, we add constraints (4.25) and (4.26) by replacing r with r_j, instead of constraints (4.22) and (4.23). Besides, for the subproblem $j \notin C_2$, we should add constraints (4.31).

However, we observe that we can add at most $(d_j + 1) \cdot 2^{d_j - 1}$−many constraints to Subproblem(j) model for c_j. That is the number of constraints in the subproblem j grows exponentially in terms of d_j (number of neighbors of c_j). Besides, since the structure of the problem has changed after adding these constraints, we can no more make use of Figure 4.1 as a solution procedure. From this analysis, we conclude that branching on w_{jS} variables is not practical in use.

4.2.1.2. Branching on f_i variables. Assume that we solve the RLPM and find that for some v_i, f_i is fractional. Then, we consider to branch the problem by assigning $f_i = 0$ in one branch and $f_i = 1$ in the other branch. We continue to branch on the **f** variables until we have an integral solution in RLPM. In that case, we have an integer feasible solution for LPM problem, which is a feasible solution of IPM.

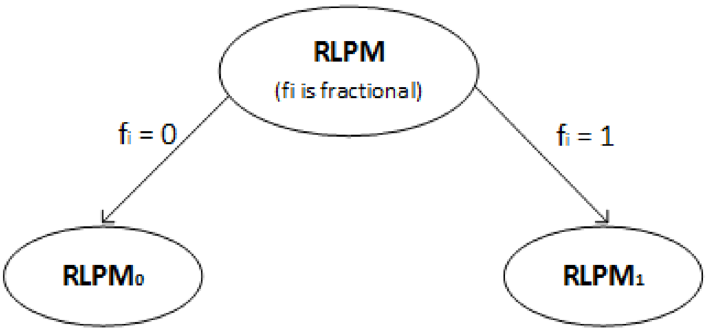


Figure 4.2. Branching strategy.

In $f_i = 0$ branch, constraints (4.10) become $\sum_{S \in \varepsilon_j, i \in S} w_{jS} = 0$ for all (i, j) edges, implying that each check node $j \in N(v_i)$ with $i \in S$ will have $w_{jS} = 0$. This means in $f_i = 0$ branch, we permanently set these w_{jS} values to 0. As a result, we can eliminate the (j, S) columns if check node $j \in N(v_i)$ and $i \in S$ from the RLPM. There can be some other (j, S) columns of RLPM such that $i \notin S$. These (j, S) columns can still be in the $f_i = 0$ branch, since check node $j \in N(v_i)$ and $i \notin S$ with $w_{jS} > 0$ implies $f_i = 0$

and for check node $j \notin N(v_i)$ nodes f_i value is not affected. We name new RLPM in the $f_i = 0$ branch as $RLPM_0$.

The subproblem for c_j in the $f_i = 0$ branch can be given as follows:

Subproblem(j_0):

$$\zeta_j^0 = \min \sum_{l \in N(c_j)} \tau_{lj} x_l - \mu_j \tag{4.32}$$

s.t.

$$\sum_{l \in N(c_j)} x_l = 2k, \tag{4.33}$$

$$x_i = 0, \text{ if } i \in N(c_j), \tag{4.34}$$

$$x_l \in \{0, 1\}, k \in \mathbb{Z}^+. \tag{4.35}$$

The subproblem j determines a local codeword S characterized with $(x_1, x_2, ..., x_{d_j})$ for c_j. For a c_j, if variable node $i \notin N(c_j)$ then selected local codeword S cannot include v_i anyway. Hence, the constraint (4.34) will not be in the subproblem. On the other hand, if variable node $i \in N(c_j)$ then selected local codeword S should not include v_i in order to agree with $f_i = 0$ condition. This is satisfied by adding constraint (4.34).

As we proceed with branching, say we are at the rth level with the conditions $f_{i_1} = f_{i_2} = ... = f_{i_r} = 0$, we will have constraints (4.36) given below instead of constraint (4.34) in subproblem j:

$$x_{i_k} = 0, \text{ if } i_k \in N(c_j), \ k = 1, ..., r. \tag{4.36}$$

Then, we can solve Subproblem(j_0) by simply applying Figure 4.1 after we discard the $\tau_{i_k,j}$ values and x_{i_k} variables for $i_k \in N(c_j),\ k = 1, ..., r$. If the objective function value $\zeta_j^0 < 0$, then we can introduce column (j, S) to the $RLPM_0$.

In $f_i = 1$ branch, constraints (4.10) become $\sum_{S \in \varepsilon_j, i \in S} w_{jS} = 1$, for all (i, j) edges. The (j, S) columns having $j \in N(v_i)$ and $i \notin S$ with $w_{jS} > 0$ imply that $f_i = 0$. Since this contradicts with the branch condition $f_i = 1$, we conclude that $w_{jS} = 0$ permanently for such columns. Hence, in the $f_i = 1$ branch we can eliminate the (j, S) columns if $j \in N(v_i)$ and $i \notin S$ from RLPM. There can be some other (j, S) columns that have $j \in N(v_i)$ and $i \in S$. Additionally, there can be (j, S) columns having $j \notin N(v_i)$, and they do not affect the value of f_i. We name new RLPM in the $f_i = 1$ branch as $RLPM_1$. We solve $RLPM_1$ and obtain the current optimal dual variables $(\boldsymbol{\mu}^*, \boldsymbol{\tau}^*)$. We solve the subproblem in order to determine the new entering columns.

The subproblem for c_j in the $f_i = 1$ branch can be given as follows:

Subproblem(j_1)*:*

$$\zeta_j^1 = \min \sum_{l \in N(c_j)} \tau_{lj} x_l - \mu_j \tag{4.37}$$

s.t.

$$\sum_{l \in N(c_j)} x_l = 2k, \tag{4.38}$$

$$x_i = 1, \text{ if } i \in N(c_j), \tag{4.39}$$

$$x_l \in \{0, 1\}, k \in \mathbb{Z}^+. \tag{4.40}$$

The subproblem j determines a local codeword S for c_j. For a check node c_j, if variable node $i \notin N(c_j)$ then selected local codeword cannot include v_i anyway. Hence, constraint (4.39) is not in the subproblem. On the other hand, if $i \in N(c_j)$ then selected local codeword S should include v_i in order to agree with the $f_i = 1$ condition.

This is satisfied by adding constraint (4.39).

As we proceed with branching, say we are at the rth level with the conditions $f_{i_1} = f_{i_2} = ... = f_{i_r} = 1$, we have constraints (4.41) given below instead of constraint (4.39) in subproblem j:

$$x_{i_k} = 1, \text{ if } i_k \in N(c_j), \; k = 1, ..., r. \tag{4.41}$$

Then, we can solve Subproblem(j_1) after we plug in $x_{i_k} = 1$ values and obtain an additional constant term from the corresponding $\tau_{i_k,j}$ values. The remaining problem can be solved by applying Figure 4.1 with a small update: if n is even then already constructed set with x_{i_k} variables is an even set, and we can continue adding even number of elements to this set as long as they have negative marginal cost. If n is odd, then we have an odd set initially. Hence, we should consider to add one element at the first iteration. We observe that we have to add the first candidate element even it has a positive marginal cost in order to have an even set, a feasible local codeword. The algorithm will consider to add even number of elements in the next iterations if they have negative marginal cost. But this is not possible when the first candidate element has positive τ_{ij} value. Since we order the τ_{ij} values in nondecreasing order, the remaining elements cannot have a negative marginal cost. Hence, in this case we will stop after adding the first element to the set. If the objective function value $\zeta_j^1 < 0$, then we can introduce column (j, S) to $RLPM_1$.

A more general case in a branch is that we have some f_i variables are set to 0 and some of them are set to 1. When we are at the rth level, we can say that $f_i = 0$ for $i \in N_0$ and $f_i = 1$ for $i \in N_1$, where $N_0 \cup N_1 = \bar{V} \subseteq V$, $|\bar{V}| = r$ and $N_0 \cap N_1 = \emptyset$. In this branch, we have added the following constraints to the subproblem j:

$$x_i = 0, \text{ if } i \in N(c_j) \cap N_0, \text{ and } x_i = 1, \text{ if } i \in N(c_j) \cap N_1. \tag{4.42}$$

In order to solve Subproblem(j), we eliminate the x_i variables for $i \in N(c_j) \cap N_0$ and we plug in the $x_i = 1$ values for $i \in N(c_j) \cap N_1$ to obtain an additional constant term from the corresponding τ_{ij} values. We can solve the remaining problem by applying Figure 4.3, modified Figure 4.1, given below. The algorithm runs in $\mathcal{O}(n \log n)$ time due to sorting step where n is the number of variable nodes.

```
Input: Sets $N_0$ and $N_1$, where $f_i = 0$ for $i \in N_0$ and $f_i = 1$ for $i \in N_1$.
0. Set $x_i = 0$, if $i \in N(c_j) \cap N_0$, and $x_i = 1$, if $i \in N(c_j) \cap N_1$.
   Let $I_j = N(c_j) \setminus (N_0 \cup N_1)$.
1. Sort the $\tau_{ij}$ values in nondecreasing order for $i \in I_j$.
   Let $\tau_{ij}^t$ be the $t$th smallest $\tau_{ij}$ value.
2. Set $x_i = 0 \ \forall i \in I_j$, set $t = 1$.
3. If $|N(c_j) \cap N_1|$ is even
4.    Then set $x_{i_1} = x_{i_2} = 1$ if $\tau_{i_1,j}^t + \tau_{i_2,j}^{t+1} < 0$, otherwise STOP.
5.    $t \leftarrow t + 2$, go to Step 4.
6. Else set $x_i = 1$ for $\tau_{ij}^t$
7.    If $\tau_{ij}^t < 0$, Then $t \leftarrow t + 1$ and go to Step 4, Else STOP.
8. End If
Output: A local codeword $S$ with objective value
        $\zeta_j = \sum_{i \in I_j} \tau_{ij} x_i + \sum_{i \in N(c_j) \cap N_1} \tau_{ij} - \mu_j$.
```

Figure 4.3. Subproblem j on a branch solution algorithm.

From the above analysis, we observe that branching on f_i variables does not change the structure of the subproblems. Hence, we can still find the optimal solution of a subproblem in polynomial time. As a result, in this study we prefer to branch on f_i variables.

The general branch–and–price algorithm for IPM problem is given in Figure 4.4. In Figure 4.4 we try two integer values, namely 0 and 1, for f_i variables. Hence, we can have at most $n(2n + 1)-$many nodes in the branch–and–price tree, i.e. at most $n(2n + 1)-$many problems in the $LIST$. For each element in the $LIST$, we solve a linear programming problem which has at most $e = (n + \sum_{j=1}^{m} 2^{d_j - 1})-$many variables, where d_j is the number of neighbors of a check node j. The problem can be encoded in L input bits, can be solved with Karmarkar's interior point algorithm in $\mathcal{O}(e^{3.5} \cdot L)$ time [46]. Besides, we apply Figure 4.3 for branching which takes $\mathcal{O}(n \log n)$ time as we have seen. As a result, Figure 4.4 runs in $\mathcal{O}(n^2 \cdot e^{3.5} \cdot L)$ time. Since, e grows exponentially in number d_j, Figure 4.4 is an exponential time algorithm.

```
Input: A set of feasible local codewords that constitutes $RLPM$
       ($\emptyset \in \varepsilon_j,\ \forall j$).
0. Set $LIST = \{RLPM\}$, let $\bar{z} = \infty$ and $\underline{z} = -\infty$.
1. While $LIST \neq \emptyset$ Do
2.     Select the last problem in $LIST$, say problem $P$.
       /* depth–first search*/
3.     Solve $P$ and obtain optimal primal $(\mathbf{f}^*, \mathbf{w}^*)$
       and dual $(\boldsymbol{\mu}^*, \boldsymbol{\tau}^*)$ solutions with value $\underline{z}^i$.
    Prunning /* delete P from the LIST */
4.     If $P$ is infeasible, Then prune by infeasibility and go to Step 1.
5.     If $\underline{z}^i \geq \bar{z}$, Then prune by bound and go to Step 1.
6.     If $P$ has an integer optimal solution, Then $\bar{z} = \underline{z}^i$,
       solve the subproblems with Figure 4.3.
7.         If $\zeta_j = 0$ for all $j$, Then prune by optimality, go to Step 1.
8.         Else add the columns with $\zeta_j > 0$ to $P$, go to Step 1.
9.         End If
10.    End If
    Branching /* add P to the LIST */
11.    If $P$ has a fractional optimal solution,
       Then choose a fractional $f_i$
        Left Branch
12.        Let $RLPM_0 = P \cap \{(\mathbf{f}, \mathbf{w}) : f_i = 0\}$,
           add $x_i = 0$ to subproblem $j$, if $i \in N(c_j)$.
13.        Solve the subproblems with Figure 4.3
           and add the columns with $\zeta_j > 0$ to $RLPM_0$.
14.        Add $RLPM_0$ to $LIST$, and go to Step 1.
        Right Branch
15.        Let $RLPM_1 = P \cap \{(\mathbf{f}, \mathbf{w}) : f_i = 1\}$,
           add $x_i = 1$ to subproblem $j$, if $i \in N(c_j)$
16.        Solve the subproblems with Figure 4.3,
           and add the columns with $\zeta_j > 0$ to $RLPM_1$.
17.        Add $RLPM_1$ to $LIST$, and go to Step 1.
18.    End If
19. End While
Output: An integral solution $(\mathbf{f}^*, \mathbf{w}^*)$ to LPM with objective value $\bar{z}$.
```

Figure 4.4. *IPM* solution algorithm.

4.2.2. Repairing Infeasibility in Node Relaxations

In the application of Figure 4.4 explained above, we observe that a branch can be prunned although there exists a feasible solution on that branch. This may happen if the currently generated columns are not sufficient to construct a feasible solution on the branch. As an example, consider we are at the $f_2 = 1$ and $f_4 = 1$ branch of Tanner graph in Figure 4.5.

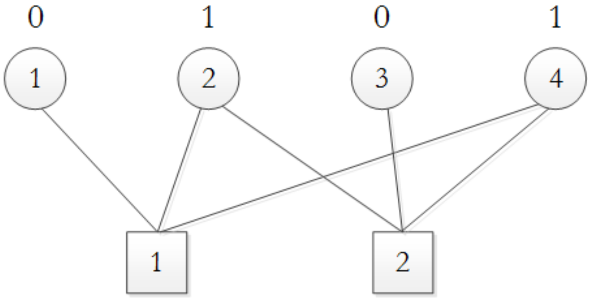


Figure 4.5. An example Tanner graph.

The set of all feasible local codewords for check node 1 is $\varepsilon_1 = \{\emptyset, \{1,2\}, \{1,4\}, \{2,4\}\}$ and for check node 2 is $\varepsilon_2 = \{\emptyset, \{2,3\}, \{2,4\}, \{3,4\}\}$. On the $f_2 = 1$ and $f_4 = 1$ branch, one can see that (0 1 0 1) is a feasible codeword if we can choose local codeword $\{2,4\}$ of check node 1 and $\{2,4\}$ of check node 2. However, we cannot find this feasible solution on the branch if we have only generated the local codewords $\emptyset$, $\{1,2\}$ and $\{1,4\}$ for check node 1 and the local codeword $\emptyset$ for check node 2. Moreover, we cannot find any other feasible solution on this branch with these limited number of local codewords.

In such a case, the $f_2 = 1$ and $f_4 = 1$ branch is prunned by infeasibility by Figure 4.4 although there is a feasible solution for LPM on the branch. In order to overcome this situation, we developed a column generation method based on the dual formulation. Let P be the primal problem representing the RLPM and D is the dual of RLPM. We first prove the following proposition:

Proposition 4.2. *P is infeasible if and only if D is unbounded.*

Proof. From the duality theory, we know that infeasible P implies D is unbounded or infeasible. We know that LPM is bounded since the variables f_i and $w_{jS} \in [0,1]$ and it is feasible since $\mathbf{0}-$codeword is a trivial solution. Then the dual of the LPM is also feasible.

D being the dual of a restricted LPM, will be feasible since it contains the feasible region defined by LPM dual. This means that D cannot be infeasible in any case. From here, we get P is infeasible $\implies$ D is unbounded.

Moreover, we can say that unbounded D implies P is infeasible from the duality theory. As a result, we conclude that P is infeasible $\iff$ D is unbounded. □

At an infeasible branch, either the current P is really infeasible or it occurs to be infeasible since we could not generate the columns that are necessary to construct a feasible solution. Then, we can make use of Proposition 4.2 to generate the required columns for P if there is a feasible solution for LPM on the branch.

If there is a feasible solution on the branch, then D should give a finite optimum solution. Then, we can able to find a dual constraint, a feasible local codeword for primal, such that we can find a finite solution. This dual constraint can be determined using the Farkas' Lemma. Consider the following primal and dual formulations:

Primal:	**Dual:**
$\min \begin{bmatrix} \mathbf{f} & \mathbf{w} \end{bmatrix} \begin{bmatrix} \gamma \\ \mathbf{0} \end{bmatrix}$	$\max\ \mathbf{c} \begin{bmatrix} \boldsymbol{\mu} \\ \boldsymbol{\tau} \end{bmatrix}$
s.t.	s.t.
$\begin{bmatrix} \mathbf{f} & \mathbf{w} \end{bmatrix} \mathbf{A} = \mathbf{c}$	$\mathbf{A} \begin{bmatrix} \boldsymbol{\mu} \\ \boldsymbol{\tau} \end{bmatrix} \leq \begin{bmatrix} \gamma \\ \mathbf{0} \end{bmatrix}$
$\mathbf{f} \geq \mathbf{0}, \mathbf{w} \geq \mathbf{0}$	$\boldsymbol{\mu}$ and $\boldsymbol{\tau}$ unrestricted

Here $\mathbf{f}$ and $\mathbf{w}$ are primal, $\boldsymbol{\mu}$ and $\boldsymbol{\tau}$ are dual decision variables, $\mathbf{A}$ is the matrix for the constraints and $\mathbf{c} = \begin{bmatrix}\mathbf{1} & \mathbf{0}\end{bmatrix}$ is the right–hand–side vector of P. Let System 1 and System 2 are defined as follows:

$$\underline{\text{System 1:}} \quad \begin{bmatrix}\mathbf{f} & \mathbf{w}\end{bmatrix}\mathbf{A} = \mathbf{c} \text{ and } \mathbf{f} \geq \mathbf{0}, \mathbf{w} \geq \mathbf{0}$$

$$\underline{\text{System 2:}} \quad \mathbf{Ad} \leq 0 \text{ and } \mathbf{cd} > 0, \mathbf{d} \text{ unrestricted.}$$

Since primal formulation is infeasible in the current branch, System 1 is infeasible. This means System 2 will have a feasible solution according to the Farkas' Lemma.

Proposition 4.3. *The solution* $\mathbf{d}$ *of System 2 is a recession direction for* D*. The dual objective is unbounded in direction* $\mathbf{d}$*.*

Proof. Let $\begin{bmatrix}\boldsymbol{\mu} \\ \boldsymbol{\tau}\end{bmatrix}$ is a feasible dual solution. Then, $\mathbf{A}\begin{bmatrix}\boldsymbol{\mu} \\ \boldsymbol{\tau}\end{bmatrix} \leq \begin{bmatrix}\gamma \\ \mathbf{0}\end{bmatrix}$, $\boldsymbol{\mu}$ and $\boldsymbol{\tau}$ are unrestricted.

$$\mathbf{A}\begin{bmatrix}\boldsymbol{\mu} \\ \boldsymbol{\tau}\end{bmatrix} + \lambda\mathbf{Ad} \leq \begin{bmatrix}\gamma \\ \mathbf{0}\end{bmatrix}, \text{ since } \mathbf{A}\begin{bmatrix}\boldsymbol{\mu} \\ \boldsymbol{\tau}\end{bmatrix} \leq \begin{bmatrix}\gamma \\ \mathbf{0}\end{bmatrix}, \lambda \geq 0 \text{ and } \mathbf{Ad} \leq 0.$$

This means that $\begin{bmatrix}\boldsymbol{\mu} \\ \boldsymbol{\tau}\end{bmatrix} + \lambda\mathbf{d}$ is also dual feasible for all $\lambda \geq 0$. That is $\mathbf{d}$ is a recession direction for the dual problem.

Besides, the dual objective $\mathbf{c}\left(\begin{bmatrix}\boldsymbol{\mu} \\ \boldsymbol{\tau}\end{bmatrix} + \lambda\mathbf{d}\right) = \mathbf{c}\begin{bmatrix}\boldsymbol{\mu} \\ \boldsymbol{\tau}\end{bmatrix} + \lambda\mathbf{cd}$ is unbounded since $\lambda \geq 0$ and $\mathbf{cd} > 0$.

Hence, the solution $\mathbf{d}$ of System 2 is a recession direction and the dual objective is unbounded. □

Since all constraints (4.15) are already exist in all RLPM duals, the candidate constraints that can bound the unbounded dual objective of P can be among the constraints (4.14). This idea is demostrated in Figure 4.6 below, where the dashed line is the constraint that we are trying to find.

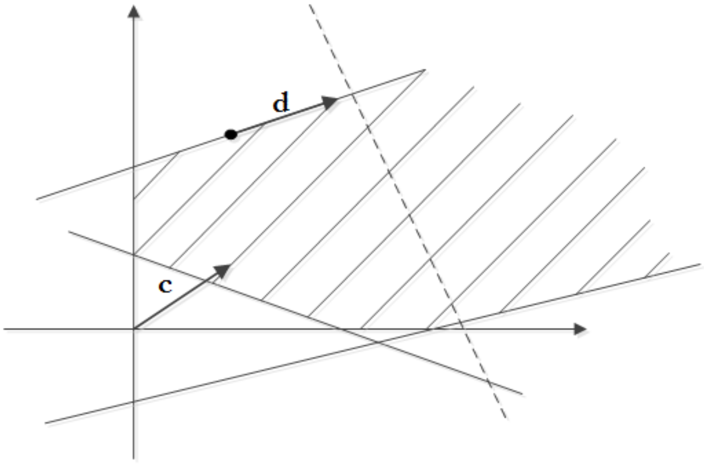


Figure 4.6. Dual constraint generation.

We can observe that for this dual constraint, there exist dual feasible solutions $\begin{bmatrix} \boldsymbol{\mu} \\ \boldsymbol{\tau} \end{bmatrix}$ that the constraint is satisfied. But as we proceed from this feasible point in the direction of $\mathbf{d}$, we will violate the constraint for sufficiently large $\lambda \geq 0$ value. That is for sufficiently large λ, the vector $\begin{bmatrix} \boldsymbol{\mu} \\ \boldsymbol{\tau} \end{bmatrix} + \lambda \mathbf{d}$ will be an infeasible vector.

Let $\mathbf{a} \begin{bmatrix} \boldsymbol{\mu} \\ \boldsymbol{\tau} \end{bmatrix} = \mu_j - \sum_{i \in S} \tau_{ij} \leq 0$ for some $j \in C$ and $S \in \varepsilon_j$ be the constraint that we would like to find. Here, $\mathbf{a}$ is the corresponding coefficients of the constraint.

Proposition 4.4. *An unbounded problem D will be bounded if the constraints such that $\mathbf{ad^t} > 0, \forall t$ are added to the problem D. Here $\mathbf{d^t}$ is a recession direction of D with $\mathbf{cd^t} > 0$.*

Proof. Let $\mathbf{d^t}$ be a recession direction with $\mathbf{cd^t} > 0$ and $\begin{bmatrix} \boldsymbol{\mu} \\ \boldsymbol{\tau} \end{bmatrix}$ be a feasible solution of D . Then, for all $\lambda \geq 0$ the vector $\begin{bmatrix} \boldsymbol{\mu} \\ \boldsymbol{\tau} \end{bmatrix} + \lambda \mathbf{d^t}$ is feasible for problem D.

Let $\mathbf{a} \begin{bmatrix} \boldsymbol{\mu} \\ \boldsymbol{\tau} \end{bmatrix} \leq 0$ be a constraint with $\mathbf{ad^t} > 0$. Then, for sufficiently large λ values the vector $\begin{bmatrix} \boldsymbol{\mu} \\ \boldsymbol{\tau} \end{bmatrix} + \lambda \mathbf{d^t}$ does not satisfy the constraint, i.e. $\mathbf{a} \left(\begin{bmatrix} \boldsymbol{\mu} \\ \boldsymbol{\tau} \end{bmatrix} + \lambda \mathbf{d^t} \right) = \mathbf{a} \begin{bmatrix} \boldsymbol{\mu} \\ \boldsymbol{\tau} \end{bmatrix} + \lambda \mathbf{ad^t} > 0$, since the constraint has the property $\mathbf{ad^t} > 0$.

Hence, adding the constraint $\mathbf{a} \begin{bmatrix} \boldsymbol{\mu} \\ \boldsymbol{\tau} \end{bmatrix} \leq 0$ to the problem D will bound the problem in direction $\mathbf{d^t}$.

Repeating this argument for all directions $\mathbf{d^t}$, we can obtain a bounded objective function value for problem D. □

One can observe that if there is no such constraint, then the problem D is unbounded implying that the restricted primal problem P on the branch is infeasible. Another observation is that there can be more than one constraint that bounds the direction $\mathbf{d^t}$. Then, our aim will be to find the constraint that has the largest $\mathbf{ad^t} > \mathbf{0}$ value.

The $\mathbf{a}$ matrix is the coefficient matrix of a constraint $\mu_j - \sum_{i \in S} \tau_{ij} \leq 0$ for some $j \in C$ and $S \in \varepsilon_j$. Hence, $\mathbf{a}$ has $(m+e)-$many entries where m is the number of check nodes and e is the total number of edges in Tanner graph. The first m entries of $\mathbf{a}$ matrix are the coeffiecients for $\boldsymbol{\mu}$ variables. Then, we have zeros except a 1 for the jth entry. The following e entries are the coefficents for $\boldsymbol{\tau}$ variables and all zero except for the -1 entries for the jth check node and the elements i in the local codeword S. Let

$\mathbf{d^t} = \begin{bmatrix} \mathbf{d}^{\boldsymbol{\mu}} \\ \mathbf{d}^{\boldsymbol{\tau}} \end{bmatrix}$, where $\mathbf{d}^{\boldsymbol{\mu}}$ and $\mathbf{d}^{\boldsymbol{\tau}}$ are the entries of the $\mathbf{d^t}$ corresponding to the indices of the variables $\boldsymbol{\mu}$ and $\boldsymbol{\tau}$, respectively. Then $\mathbf{ad^t} = \mathbf{d}^{\boldsymbol{\mu}} - \sum_{i \in S} \mathbf{d}^{\boldsymbol{\tau}}$ and maximizing $\mathbf{ad^t}$ is equivalent to maximizing $d_j^{\boldsymbol{\mu}} - \sum_{i \in S} d_{ij}^{\boldsymbol{\tau}}$ for each check node j.

Hence, we have to solve the following direction subproblem for each c_j:

Direction Subproblem(j):

$$\min \sum_{i \in N(c_j)} d_{ij}^{\boldsymbol{\tau}} x_i - d_j^{\boldsymbol{\mu}} \tag{4.43}$$

$$\text{s.t.}$$

$$\sum_{i \in N(c_j)} x_i = 2k, \tag{4.44}$$

$$x_i \in \{0, 1\}, k \in \mathbb{Z}^+. \tag{4.45}$$

We observe that the direction subproblem is actually in the same format with the column generation subproblem. Hence, on a branch we can solve the direction subproblem with Figure 4.3 after replacing τ_{ij} and μ_j with $d_{ij}^{\boldsymbol{\tau}}$ and $d_j^{\boldsymbol{\mu}}$, respectively. As a result, we can summarize our dual method for generating dual constraints, i.e. primal columns, with Figure 4.7.

4.2.3. A Pruning Strategy

In a BP algorithm, we are applying three pruning rules, namely prune by optimality, by infeasibility and by value dominance. We will consider an additional pruning rule that is based on the difference between the objective function values of two feasible integral solutions.

As we explained in Section 4.1, there are two alternative objective function definitions in literature for decoding problem. First is Hamming distance (given as equation

```
Input: An infeasible restricted primal problem, $P$
1. Solve the dual Farkas system and obtain a recession direction $\mathbf{d}$
   that $D$ is unbounded.
2. Solve Direction Subproblem($j$) for each check node $j$.
   Add generated local codewords, i.e. columns, to $P$.
3. If no columns generated, Then conclude $P$ is infeasible.
       Prune the branch by infeasibility and STOP.
4. Solve problem $P$.
5. If $P$ is feasible, Then STOP.
6. Else Go to Step 1.
7. End If
Output: A feasible restricted primal problem $P$ or prune $P$ by infeasibility.
```

Figure 4.7. Dual constraint generation algorithm.

(4.1)) and second is log–likelihood (given as equation (4.5)) objectives.

Proposition 4.5. *Log–likelihood and Hamming distance objectives are equivalent. That is both objectives give the same optimum solution set for decoded codeword* $\mathbf{f}$.

Proof. First consider the log–likelihood objective. As it is given in study [9], $\gamma_i = \log[(p)/(1-p)]$ if received bit $\hat{y}_i = 1$ and $\gamma_i = \log[(1-p)/(p)]$ if $\hat{y}_i = 0$ where p is the error probability for BSC. Then, the objective can be written as

$$\min - \sum_{i:\hat{y}_i=1} af_i + \sum_{i:\hat{y}_i=0} af_i \tag{4.46}$$

where $a = \log[(1-p)/(p)]$. In practical applications p is a small number, i.e. $p = 0.001$, which implies $a \geq 0$.

On the other hand, Hamming distance objective can be written as

$$\min - \sum_{i:\hat{y}_i=1} f_i + \sum_{i:\hat{y}_i=0} f_i + c_1 \tag{4.47}$$

where $c_1 = \sum_{i:\hat{y}_i=1} 1$.

One can observe that Hamming distance objective is a scaled version of log–likelihood objective by choosing $a = 1$ and adding a constant term c_1. Hence, both objectives have the same optimum solution set. $\square$

Proposition 4.6. *Let* $\mathbf{f}$ *be a feasible integral solution of LPM with objective function value* z*. Then, there is no feasible integral solution of LPM with objective function value in the range* $(z - a, z)$ *with log–likelihood objective.*

Proof. From log–likelihood objective (4.46), we can see that z is an integral multiple of a since $\mathbf{f}$ is integral, i.e. $z = k \cdot a$ where $k \in \mathbb{Z}$. Let $\mathbf{f}'$ be another integral feasible solution of LPM. Then, its objective value z' is also an integral multiple of a, say $z' = k' \cdot a$ and where $k' \in \mathbb{Z}$. The difference among the objectives is $z - z' = (k - k') \cdot a$. From here, we can conclude that the nearest objective function value to z can be either $z' = z + a$ or $z' = z - a$. Hence, there is no feasible integral solution of LPM with objective function value in the range $(z - a, z)$. $\square$

In another words, the minimum difference between two feasible integral solutions is a with log–likelihood objective and 1 with Hamming distance.

Proposition 4.7. *Let* z *be the optimum solution of a RLPM at a branch. Prune the branch if* $z > z_{UB} - a$ *where* z_{UB} *is the best upper bound on the IPM and* a *is the minimum difference between two feasible integral solutions.*

Proof. A branch can be pruned by value dominance if $z > z_{UB}$. Besides, as shown in Proposition 4.6, there cannot be an integral feasible solution in the range $(z_{UB} - a, z_{UB})$. Hence, we can prune the branch if $z > z_{UB} - a$. $\square$

4.2.4. On the Strength of LP Relaxation

We first observe that the objective function of EM minimizes the Hamming distance to the received codeword $\hat{\mathbf{y}}$. Then, the optimum function value of EM is non-negative for all instances, i.e. $z_{EM} \geq 0$. Let $z_{EM} = \min_{\mathbf{f}} z_{EM}(\mathbf{f})$. We have $z_{EM}(\mathbf{f}) = 0$ if $\mathbf{f} = \hat{\mathbf{y}}$ and for any feasible solution $\mathbf{f} \neq \hat{\mathbf{y}}$ the objective function value $z_{EM}(\mathbf{f}) > 0$.

Proposition 4.8. *The optimum objective function value of LEM is 0 for all instances, i.e.* $z_{LEM} = 0$.

Proof. Let $\mathbf{f}$ be a fractional solution of LEM. Then, there is $i \in V$ such that $0 < f_i < 1$. If $\hat{y}_i = 1$, then we will have cost $(1 - f_i) > 0$ and if $\hat{y}_i = 0$, then we will have cost $f_i > 0$ to be added to the objective function. Then, for any fractional solution we have $z_{LEM}(\mathbf{f}) > 0$. In general if $\mathbf{f} \neq \hat{\mathbf{y}}$, then $z_{LEM}(\mathbf{f}) > 0$.

Let $\mathbf{f} = \hat{\mathbf{y}}$, then $\mathbf{f}$ is feasible for LEM since $0 \leq f_i \leq 1 \ \forall i$ and $k_j = \frac{\sum_{i \in V} H_{ij} f_i}{2} \geq 0$ since H_{ji} is a matrix of 0s and 1s. Then, the optimum objective function value $z_{LEM} = 0$ for all $\mathbf{H}$ instances. □

Proposition 4.9. *LPM problem with Hamming distance objective (4.1) has strictly positive optimum objective value, i.e.* $z_{LPM} > 0$, *if received codeword* $\hat{\mathbf{y}}$ *is not a feasible codeword.*

Proof. If received vector $\hat{\mathbf{y}}$ is a feasible codeword, then $\mathbf{f} = \hat{\mathbf{y}}$ be a feasible solution for LPM and it will be optimal. Let $\hat{\mathbf{y}}$ is not a feasible codeword. Then, LPM problem will have a fractional or integral feasible solution $\mathbf{f} \neq \hat{\mathbf{y}}$. This means the Hamming distance objective will be strictly positive for this optimal solution. Hence, $z_{LPM} > 0$, if received codeword $\hat{\mathbf{y}}$ is not a feasible codeword. □

To summarize, linear relaxation of EM formulation gives $z_{LEM} = 0$ for all $\mathbf{H}$ instances. The linear relaxation of IPM problem gives $z_{LPM} = 0$ if the received codeword $\hat{\mathbf{y}}$ is a feasible codeword, otherwise $z_{LPM} > 0$. This means that LPM gives a better lower bound for IPM objective than LEM.

4.2.5. Computational Results

The computations have been carried out on a computer with 2.6 GHz Intel Core i5-3230M processor and 4 GB of RAM working under Windows 10 Professional operating system. We test the perfromance of our branch–and–price (BP) algorithm against CPLEX 12.6.0 using EM formulation. In order to understand the characteristics of the decoding problem, we carry out some pre–computations with 11 LDPC codes. In Table 4.1, we observe that lower bound obtained from LPM formulation is better than LEM.

Table 4.1. LP relaxation and optimal solution values.

Input No	(m, n)	LEM	LPM	z^*
0	(5, 8)	0	2	3
1	(8, 8)	0	2	2
2	(6, 15)	0	3	3
3	(12, 15)	0	3	3
4	(9, 18)	0	2	2
5	(15, 21)	0	3	5
6	(20, 30)	0	2	4
7	(24, 36)	0	1.56	3
8	(30, 40)	0	2	6
9	(36, 48)	0	1.68	5
10	(40, 52)	0	2	6
Average:		0	2.20	3.82

In Table 4.2, CPU seconds consumed by CPLEX and our BP algorithm to find the optimum solution are listed. The results show that our algorithm is not performing better than CPLEX.

Table 4.2. CPU in seconds for CPLEX and BP.

Input No	(m, n)	CPLEX	BP
0	(5, 8)	0.16	0.20
1	(8, 8)	0.20	0.17
2	(6, 15)	0.08	0.20
3	(12, 15)	0.22	0.50
4	(9, 18)	0.36	0.27
5	(15, 21)	0.23	2.15
6	(20, 30)	0.31	8815.70
Average:		0.22	1259.88

The performance of our algorithm can be further visualized in Figure 4.8. In the first subfigure, the upper bound for the current the relaxation problem at the node is given with a dot. As we process the nodes, the best upper bound reaches to the optimum integer solution of value 5 starting form an initial upper bound 9. The second subfigure shows the number of iterations performed at the current node before branching or pruning. It can be seen that we are consuming 28 iterations at the root node before branching. On the average, we are carrying out 2 iterations per node. From these observations, a tight upper bound on the optimum solution will be helpful for early pruning the nodes.

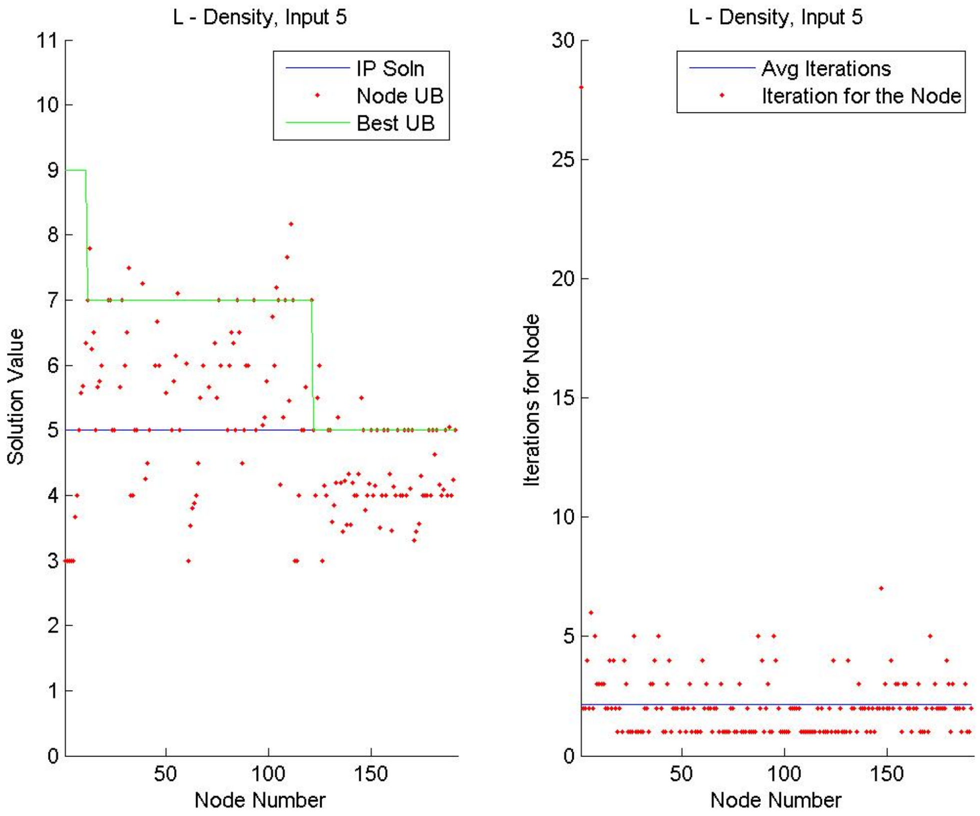


Figure 4.8. Track of iterations for Input 5.

<u>4.2.5.1. Performance of CPLEX.</u> In this section, we test the performance of CPLEX with EM formulation under different **H** instances. We would like to understand the response of CPLEX to the density of **H** matrix and length of the codeword.

H matrix becomes denser as the number of ones in the matrix gets larger. Keeping the (m, n) dimensions of **H** matrix constant, we generate (3, 6), (5, 10), (6, 12), (10, 20), (15, 30), (30, 60)–regular codes. When the matrix dimensions are the same, a (5, 10)–regular code is denser than a (3, 6)–regular code. One expects that solving an instance with a denser **H** matrix is more difficult since the decision variables become more dependent to each other (see Constraints (4.2)).

The length of the codeword, n, also affect the performance of the solver. When **H** matrices are from the same code family, by increasing n means the dimension of the matrix gets larger which adds new constraints to EM. For a (3, 6)–regular code $n = 60$ means $m = 30$ and $n = 120$ means $m = 60$. Here m is both the number of rows of **H** matrix and the number of constraints in EM. We tried 8 different codeword lengths from $n = 60$ to $n = 480$.

For the results in Table 4.3, $\mathbf{0}-$codeword is damaged with an error rate $p = 0.01$ and a received codeword is obtained. As we move to the right on a row, for example $n = 360$, the dimension of **H** matrix is constant but its density increases. Hence, we observe that for $(6, 30, 60)$ code, i.e. $s = 6$ to have $n = s \times 60$ $(360 = 6 \times 60)$, the solution cannot be found due to memory limitations. As we move to down on a column, for example $(s, 30, 60)$, the dimension of **H** increases. We can observe that when the dimension becomes $(s \times 30, s \times 60) = (180, 360)$ for $s = 6$, finding a solution is not possible due to memory.

Table 4.3. Performance of CPLEX under low error rate (in seconds).

	(s, 3, 6)		(s, 5, 10)		(s, 6, 12)		(s, 10, 20)		(s, 15, 30)		(s, 30, 60)	
n	z^*	CPU	z^*	CPU	z^*	CPU	z^*	CPU	z^*	CPU	z^*	CPU
60	3	0,33	3	0,15	3	0,18	3	0,30	3	0,34	1	0,18
120	5	0,32	5	0,13	5	0,22	5	0,24	5	0,38	5	1,77
180	5	0,32	5	0,13	5	0,20	5	0,19	5	0,42	5	1,43
240	6	0,34	6	0,14	6	0,20	6	0,24	6	0,41	6	7,05
300	7	0,36	7	0,38	7	0,23	7	0,23	7	0,44	7	25,65
360	9	0,38	9	0,28	9	0,24	9	0,22	9	0,48		−−
420	10	0,35	10	0,18	10	0,23	10	0,25	10	0,34		−−
480	10	0,36	10	0,17	10	0,22	10	0,26	10	0,29		−−
Avg:		0,34		0,20		0,22		0,24		0,39		7,23

Another performance analysis is carried out with the received codeword is $\hat{\mathbf{y}} =$ 0101...01. For all instances this received codeword is used with approporiate length n. One should change half of the bits of $\mathbf{0}-$codeword in order to obtain $\hat{\mathbf{y}}$. Hence, we are considering a case with high error rate.

The results are given in Table 4.4 below. As we can see from the results that as the error rate increases it is more difficult to find a solution for CPLEX. This analysis shows that CPLEX gets stuck when the density of $\mathbf{H}$ matrix increases and the codeword gets longer.

Table 4.4. Performance of CPLEX under high error rate (in seconds).

	(s, 3, 6)		(s, 5, 10)		(s, 6, 12)		(s, 10, 20)		(s, 15, 30)		(s, 30, 60)	
n	z^*	CPU	z^*	CPU	z^*	CPU	z^*	CPU	z^*	CPU	z^*	CPU
60	8	0,22	6	0,84	6	0,52	6	2,09	4	0,72	0	0,14
120	14	0,37	14	725,84		−−		−−		−−		−−
180	22	4,40		−−		−−		−−		−−		−−
240	30	99,54		−−		−−		−−		−−		−−
300	36	1222,51		−−		−−		−−		−−		−−
360		−−		−−		−−		−−		−−		−−
420		−−		−−		−−		−−		−−		−−
480		−−		−−		−−		−−		−−		−−
Avg:		265,41		363,34		0,52		2,09		0,72		0,14

<u>4.2.5.2. Performance of BP Algorithm.</u> In addition to BP algorithm that is explained in Section 4.2, we add the columns that has zero reduced cost to the RLPM in order to speed up the column generation procedure. We provide a time limit of 10 minutes to BP algorithm.

The results given in Table 4.5 and 4.6 show that the performance of BP is not better than CPLEX in terms of computation time. For some of the instances we give the best solution, LP relaxation lower bound and $\mathbf{0}-$codeword upper bound in Table 4.7. From these results we can see that optimum solution is around the initial lower bound. This means BP devotes most of the time of to find an upper bound.

Then, we expect that providing a tight upper bound can improve the performance of BP algorithm. For this purpose, methods that exist in the literature and the

Table 4.5. Performance of BP under low error rate (in seconds).

	(s, 3, 6)		(s, 5, 10)		(s, 6, 12)		(s, 10, 20)		(s, 15, 30)		(s, 30, 60)	
n	z^*	CPU	z^*	CPU	z^*	CPU	z^*	CPU	z^*	CPU	z^*	CPU
60	3	0,18	3	0,15	3	0,18	3	0,27	3	0,32	1	4,44
120	5	0,16	5	0,18	5	0,24	5	8,65	5	0,91	5	91,45
180	5	0,16	5	0,20	5	0,22	5	20,17	5	276,37	*time*	
240	6	0,17	6	0,20	6	0,26	6	0,43	6	554,08	*time*	
300	7	0,17	7	0,22	7	0,26	7	64,36	*time*		*time*	
360	9	0,16	9	0,19	9	0,31	9	279,82	*time*		*time*	
420	10	0,18	10	0,26	10	0,25	10	488,74	*time*		*time*	
480	10	0,21	10	0,32	10	0,53	10	566,64	*time*		*time*	
Avg:		0,17		0,22		0,28		178,64		207,92		47,95

Table 4.6. Performance of BP under high error rate (in seconds).

	(s, 3, 6)		(s, 5, 10)		(s, 6, 12)		(s, 10, 20)		(s, 15, 30)		(s, 30, 60)	
n	z^*	CPU	z^*	CPU	z^*	CPU	z^*	CPU	z^*	CPU	z^*	CPU
60	8	0,81	6	306,01	*time*		*time*		*time*		*time*	
120	14	49,02	*time*		*time*		*time*		*time*		*time*	
180	*time*		*time*		*time*		*time*		*time*		*time*	
Avg:		24,91		306,01		0		0		0		0

Table 4.7. Initial lower and upper bounds in BP.

(s, k, l)	z^*	$\underline{z}$	$\bar{z}$
$(10, 3, 6)$	8	6,3	30
$(20, 3, 6)$	14	12,89	60
$(30, 3, 6)$	22	19,99	90
$(40, 3, 6)$	30	26	120
$(50, 3, 6)$	36	32,83	150
$(6, 5, 10)$	6	4,44	30
$(12, 5, 10)$	14	8,5	60
$(5, 6, 12)$	6	4,11	30
$(3, 10, 20)$	6	3,02	30

new heuristic methods are tried and their performances are reported in the following sections.

4.3. Feasible Solution Generation Methods

In this section, we list our feasible solution (upper bound) generation techniques. We test their affects on the performance of BP algorithm computationally.

4.3.1. Gallager A and B Algorithms

Gallager A and B algorithms (given in Chapter 3) are hard–decision decoding algorithms [42]. That is the decoded codeword is a sequence of 0s and 1s.

Gallager A and B algorithms are quite similar. For each bit of the received codeword $\hat{\mathbf{y}}$, the algorithm collects the opinions of each check node. If the neighboring check node is unsatisfied, then this is considered as an indication of an error in the corresponding bit. If most of the neighbors of a bit are unsatisfied, we have a strong intuition that the bit is erroneous.

As given in Figure 3.5, Gallager A algorithm prefers to flip the bit that has the maximum number of unsatisfied checks. At each iteration of the algorithm, we flip only one bit which guarantees that the number of unsatisfied checks will decrease at each iteration. In Gallager B algorithm, decrease in the unsatisfied checks at each iteration is not for sure since it applies multi–flip at an iteration.

Stopping criterion can be the number of iterations. The major problem with these algorithms is that they may get stuck when there is a cycle in the LDPC code. When the code gets denser, the probability of observing a cycle in the Tanner graph increases. Since our aim is to solve respectively denser codes, the expectation is that these algorithms performs poorly in finding an upper bound.

For the computational experiments, (3, 6), (5, 10) and (6, 12)–regular codes of different dimensions are used. The codeword 0101 ... 01 is used as the received vector. The results of BP with Gallager A is compared with the ones of CPLEX in Tables 4.8, 4.9 and 4.10.

The best known solution in 10 minutes time limit is reported as z^* for BP. The initial lower bound obtained from LPM model given in $\underline{z}$ column. The initial upper bound obtained from $\mathbf{0}-$codeword is given in column $\bar{z}$. When the values z^*, $\underline{z}$ and $\bar{z}$ are compared, one can see that z^* is closer to the $\underline{z}$. That is most of the time

Table 4.8. Performance of Gallager A for $(s, 3, 6)$ codes (in seconds).

	CPLEX			BP with Gallager A				
s	z^*	CPU	Nodes	z^*	CPU	Nodes	$\underline{z}$	$\bar{z}$
2	0	0,16	0	0	0,13	0	0	6
3	3	0,15	0	3	0,17	2	3	9
4	4	0,13	0	4	0,15	0	4	12
5	3	0,15	0	3	0,16	0	3	15
6	4	0,21	0	4	0,17	0	4	18
7	5	0,24	40	5	0,70	216	5	21
8	6	0,25	393	6	0,54	130	6	24
9	7	0,16	11	7	0,27	26	5,21	27
10	8	0,20	16	8	0,68	140	6,33	30
11	9	0,17	0	9	4,86	1794	7,74	33
12	8	0,20	0	8	3,54	866	6,68	36
13	9	0,21	193	9	6,18	1636	7,66	39
14	10	0,48	390	10	3,60	712	7,35	42
15	11	0,20	0	11	0,37	28	9,36	45
16	12	0,34	1294	12	8,41	1820	10,36	48
17	13	0,71	819	13	19,25	3540	10,29	51
18	12	0,12	0	12	10,46	1500	11,75	54
19	13	0,19	11	13	2,73	252	11,53	57
20	14	0,29	124	14	45,63	5864	12,88	60
21	15	0,48	52	15	114,63	20668	14,08	63
Avg:	8,3	0,3	167,2	8,3	11,1	1959,7	7,3	34,5

is consumed to find an improving upper bound. Total number of nodes used in the branch–and–bound tree for CPLEX and BP are given under the "Nodes" columns. We apply Gallager A algorithm at the end of each node in BP. Then, "Nodes" column represents the number of callings of Gallager A algorithm. It can be observed that Gallager A algorithm is poor to give an improving upper bound.

The performance of Gallager B algortihm is quite similar with Gallager A in terms of finding an improving upper bound.

4.3.2. Belief Propagation Algorithm

Belief Propagation is a soft–decision decoding algorithm [47]. The algorithm calculates the log–likelihood ratios (LLRs) which are used as messages from variable nodes to check nodes. The LLRs can be calculated by equation 4.6.

Table 4.9. Performance of Gallager A for $(s, 5, 10)$ codes (in seconds).

	CPLEX			BP with Gallager A				
s	z^*	CPU	Nodes	z^*	CPU	Nodes	$\underline{z}$	$\bar{z}$
2	2	0,14	0	2	0,18	0	2	10
3	3	0,18	0	3	1,07	74	3	15
4	6	0,29	0	6	36,01	1368	2,26	20
5	5	0,34	1457	5	65,06	1986	5	25
6	6	0,64	885	6	293,22	10366	4,44	30
7	9	2,34	9332	17	*time*	24491	5,59	35
8	8	1,66	1455	28	*time*	33514	5,58	40
9	9	21,70	45240	39	*time*	35879	5,82	45
10	10	25,94	85628	48	*time*	34080	7,27	50
12	14	705,66	1823416	60	*time*	30509	8,5	60
Avg:	7,2	75,9	196741,3	25,8	79,1	17879,7	5,3	36,4

Table 4.10. Performance of Gallager A for $(s, 6, 12)$ codes (in seconds).

	CPLEX			BP with Gallager A				
s	z^*	CPU	Nodes	z^*	CPU	Nodes	$\underline{z}$	$\bar{z}$
2	2	0,22	0	2	1,17	98	2	12
3	4	0,22	307	4	16,33	560	2,62	18
4	4	0,43	762	4	238,01	4316	4	24
5	6	0,59	3326	10	*time*	9666	4,11	30
6	8	11,22	33733	28	*time*	24935	4,72	36
7	8	81,26	270502	42	*time*	20698	4,97	42
8	10	99,33	440923	48	*time*	17970	6,59	48
9	12	7326,56	15428940	54	*time*	15678	6,53	54
Avg:	6,8	939,9	2022311,6	28	85,2	12175,9	4,8	36

The initial message sent from variable k to check j, i.e. $m_{vc_{kj}}$, will be these LLRs. The message sent from check j to variable i, i.e. $m_{cv_{ji}}$, can be calculated with the following formula

$$m_{cv_{ji}} = 2\tanh^{-1}\left(\prod_{k \neq i} \tanh(m_{vc_{kj}}/2)\right). \tag{4.48}$$

As in the case of Gallager, belief propagation (in Figure 4.9) can get stuck when there are cycles in the Tanner graph. Hence, the performance of it is similar with Gallager A and B algorithms which is not tabulated here.

```
Input: An infeasible received vector, $\hat{\mathbf{y}}$
1. Initialize the LLRs using Equation (4.6) and
   messages to check nodes, i.e. $m_{vc_{kj}}$.
2. For each check node calculate messages to variable nodes, i.e. $m_{cv_{ji}}$
   using Equation (4.48).
3. For each variable node calculate the overall LLR and
   make a hard decision for each bit based on the sign of the LLR.
4. If all check nodes are satisfied or iteration limit is reached,
   Then STOP.
5. Else Go to Step 1 with the new LLR and $m_{vc_{kj}}$ values.
6. End If
Output: A feasible decoded codeword, or no solution
```

Figure 4.9. Belief propagation algorithm.

4.3.3. Partial IP Algorithm

The main idea of Partial Integer Programming algorithm in Figure 4.10 is to solve the integer programming problem version of the RLPM at the end of each node of the branch–and–price tree. The model starts with the best known integer solution and try to find a better feasible solution with the generated columns upto that time. In order to save some time the method is applied at the initial nodes of the tree and the application frequency decays to the end of the branch–and–price tree. Again to save time, we generate only one solution that is better than the starting solution. A time limit of 10 minutes for the BP is enforced.

```
Input: A RLPM problem at a node, current best solution $\mathbf{f}$
1. Convert the RLPM to an IP problem.
2. Set a limit on time and the number of solutions generated.
3. Solve the IP problem and update $\mathbf{f}$ if a solution has found.
Output: A better feasible solution, or no solution
```

Figure 4.10. Partial IP algorithm.

The main disadvantage of this method is that we are solving integer models at the nodes. The computation time increases due to these IP models. The results of the computational experiments are given in Tables 4.11, 4.12 and 4.13. The “Impr” column shows the number of solutions that improve the upper bound.

Although the results are not better than CPLEX, we can see that the number of nodes in the branch-and-price tree decreased in (3, 6)–regular codes compared with Gallager A in the expense of time. This means, the algorithm is succesful to find better upper bounds than Gallager A. But this is not sufficient to surpass CPLEX. When the density of the code increases, the performance of Partial IP algorithm decreases both in terms of computational time and number of nodes compared with Gallager A. As a result, this method does not seem to be practical.

Table 4.11. Performance of Partial IP for $(s, 3, 6)$ codes (in seconds).

	CPLEX			BP with Partial IP			
s	z^*	CPU	Nodes	z^*	CPU	Nodes	Impr
2	0	0,16	0	0	0,14	0	0
3	3	0,15	0	3	0,62	6	3
4	4	0,13	0	4	0,17	0	0
5	3	0,15	0	3	0,15	0	0
6	4	0,21	0	4	0,16	0	0
7	5	0,24	40	5	0,94	94	10
8	6	0,25	393	6	1,15	164	9
9	7	0,16	11	7	0,43	20	4
10	8	0,20	16	8	0,43	16	4
11	9	0,17	0	9	1,09	130	2
12	8	0,20	0	8	0,44	6	2
13	9	0,21	193	9	0,34	8	1
14	10	0,48	390	10	3,92	690	5
15	11	0,20	0	11	1,43	28	2
16	12	0,34	1294	12	0,62	20	4
17	13	0,71	819	13	9,52	1114	2
18	12	0,12	0	12	0,37	2	1
19	13	0,19	11	13	2,79	150	3
20	14	0,29	124	14	0,65	4	2
21	15	0,48	52	15	8,68	820	1
Avg:	8,3	0,3	167,2	8,3	1,7	163,6	2,8

4.3.4. Coverage Algorithm

Coverage algorithm in Figure 4.11 works on a coverage problem for a given infeasible vector $\hat{\mathbf{y}}$. Since $\hat{\mathbf{y}}$ is not a feasible codeword, there is a set of check nodes, say C_u, that are not satisfied. Besides, there is a set of variable nodes, say V_c, that are incident to the checks in C_u. This heuristic tries to find the minimum cardinality subset of V_c that cover C_u. The problem can be expressed by a coverage model with binary decision

Table 4.12. Performance of Partial IP for $(s, 5, 10)$ codes (in seconds).

	CPLEX			BP with Partial IP			
s	z^*	CPU	Nodes	z^*	CPU	Nodes	Impr
2	2	0,14	0	2	0,18	0	0
3	3	0,18	0	3	2,97	152	9
4	6	0,29	0	6	40,26	1368	12
5	5	0,34	1457	5	95,12	2164	37
6	6	0,64	885	6	334,23	10366	41
7	9	2,34	9332	17	*time*	21767	48
8	8	1,66	1455	30	*time*	23867	53
9	9	21,70	45240	39	*time*	7941	61
10	10	25,94	85628	50	*time*	33	33
12	14	705,66	1823416	60	*time*	4	4
Avg:	7,2	75,9	196741,3	21,8	94,6	6766,2	29,8

Table 4.13. Performance of Partial IP for $(s, 6, 12)$ codes (in seconds).

	CPLEX			BP with Partial IP			
s	z^*	CPU	Nodes	z^*	CPU	Nodes	Impr
2	2	0,22	0	2	3,64	148	21
3	4	0,22	307	4	18,21	416	31
4	4	0,43	762	4	267,00	4508	35
5	6	0,59	3326	10	*time*	9842	43
6	8	11,22	33733	28	*time*	21782	51
7	8	81,26	270502	42	*time*	12589	59
8	10	99,33	440923	48	*time*	2911	64
9	12	7326,56	15428940	54	*time*	38	38
Avg:	6,8	939,9	2022311,6	24	96,3	6529,2	42,8

variable x_i which takes value 1 if variable node i is selected and 0 otherwise.

Coverage Model (CM):

$$\min \sum_{i \in V_c} x_i \tag{4.49}$$

s.t.

$$\sum_{i \in N(c_j)} x_i \geq 1, \ \forall j \in C_u \tag{4.50}$$

$$x_i \in \{0, 1\}, \ \forall i \in V_c. \tag{4.51}$$

The coverage problem can be solved by a greedy heuristic. We choose the variable node that has the largest number of neighbors, then eliminate the check nodes that are covered by this variable node. For the remaining uncovered check nodes we repeat the greedy procedure.

The set of variable nodes that covers the unsatisfied check nodes can be flipped. We can guarantee that the check nodes in C_u will be satisfied in the next iteration, but it is possible that some of the satisfied checks become unsatisfied. That is we continue with this flipping algorithm based on the coverage problem until we find a feasible solution or terminate due to the iteration limit.

```
Input: An infeasible vector $\hat{\mathbf{y}}$
1. Determine the unsatisfied checks $C_u$ and
   their variable neighbors $V_c$.
2. Solve the CM with greedy heuristic.
3. If all check nodes are satisfied or iteration limit is reached,
   Then STOP.
4. Else Go to Step 1.
5. End If
Output: A feasible solution, or no solution.
```

Figure 4.11. Coverage algorithm.

If we can generate a feasible solution, we add the corresponding columns to our RLPM problem if they are not yet added. We can apply this heuristic to the received codeword and also intermediary infeasible solutions during the branch-and-price algorithm.

We apply this algorithm at the end of each node with a decreasing probability. The results are summarized in Tables 4.14, 4.15 and 4.16. The Coverage algorithm is called Nodes–many time. Among these Feas–many of them give a feasible solution and Impr–many of them improve the best upper bound.

The performance of Coverage algorithm is similar to Gallager A. The Coverage algorithm slightly better than Gallager A in terms of the number of nodes and computational time. However this is not sufficient to perform better than CPLEX.

Table 4.14. Performance of Coverage for $(s, 3, 6)$ codes (in seconds).

	CPLEX			BP with Coverage				
s	z^*	CPU	Nodes	z^*	CPU	Nodes	Impr	Feas
2	0	0,16	0	0	0,13	0	1	1
3	3	0,15	0	3	0,21	2	0	1
4	4	0,13	0	4	0,18	0	0	0
5	3	0,15	0	3	0,17	0	0	0
6	4	0,21	0	4	0,22	0	0	0
7	5	0,24	40	5	0,79	188	2	31
8	6	0,25	393	6	0,69	130	0	6
9	7	0,16	11	7	0,31	26	0	3
10	8	0,20	16	8	0,78	106	2	14
11	9	0,17	0	9	3,83	986	2	28
12	8	0,20	0	8	3,66	720	1	26
13	9	0,21	193	9	6,12	1292	2	19
14	10	0,48	390	10	3,62	514	1	17
15	11	0,20	0	11	0,46	28	0	1
16	12	0,34	1294	12	6,85	1162	4	25
17	13	0,71	819	13	21,69	3540	0	11
18	12	0,12	0	12	11,89	1500	0	5
19	13	0,19	11	13	3,26	252	0	7
20	14	0,29	124	14	50,5	5864	0	6
21	15	0,48	52	15	101,50	14422	1	7
Avg:	8,3	0,3	167,2	8,3	10,8	1536,6	0,8	10,4

4.3.5. Constraint Programming Algorithm

Constraint programming is a well–known method to solve problems [48]. In order to apply constraint programming, a practical representation of the feasible solutions of the problem is found. Then the problem is expressed as a Constraint Satisfaction Problem (CSP). The experiences with this method indicate that if an efficient model can be generated, it is possible to find solutions for combinatorial optimization problems in an acceptable amount of time.

In our case, the sequence of the bits of a codeword, $\mathbf{f} = (f_1, f_2, ..., f_n)$, will be the representation of the solution. Each decision variable, i.e. f_i, can take values from $\{0, 1\}$. The constraints to have a feasible solution are added to the model.

Table 4.15. Performance of Coverage for $(s, 5, 10)$ codes (in seconds).

	CPLEX			BP with Coverage				
s	z^*	CPU	Nodes	z^*	CPU	Nodes	Impr	Feas
2	2	0,14	0	2	0,18	0	0	0
3	3	0,18	0	3	1,13	72	2	8
4	6	0,29	0	6	27,49	1044	1	12
5	5	0,34	1457	5	61,95	1976	0	9
6	6	0,64	885	6	290,00	10366	0	0
7	9	2,34	9332	17	*time*	23727	0	5
8	8	1,66	1455	28	*time*	32621	0	3
9	9	21,70	45240	39	*time*	36097	0	2
10	10	25,94	85628	48	*time*	33433	0	2
12	14	705,66	1823416	60	*time*	31128	0	2
Avg:	7,2	75,9	196741,3	21,4	76,2	17046,4	0,3	4,3

Table 4.16. Performance of Coverage for $(s, 6, 12)$ codes (in seconds).

	CPLEX			BP with Coverage				
s	z^*	CPU	Nodes	z^*	CPU	Nodes	Impr	Feas
2	2	0,22	0	2	1,72	154	3	18
3	4	0,22	307	4	5,25	142	2	12
4	4	0,43	762	4	256,71	4564	1	6
5	6	0,59	3326	10	*time*	9657	0	2
6	8	11,22	33733	28	*time*	24086	0	2
7	8	81,26	270502	42	*time*	20269	0	2
8	10	99,33	440923	48	*time*	17405	0	3
9	12	7326,56	15428940	54	*time*	14816	0	0
Avg:	6,8	939,9	2022311,6	24	87,9	11386,6	0,8	5,6

Constraints 1: For each check j, the neighboring variable nodes should sum to zero in modulo 2

$$\sum_{i \in N(j)} f_i = 0 \pmod 2 \quad \forall j \in C. \tag{4.52}$$

It is known that dummy constraints may improve the performance of constraint programming. Then, the following dummy constraints are added to the model.

Constraints 2: Assume that for a check j the constraint (4.52) is written as $f_2 + f_3 + f_8 + f_{10} = 0 \pmod 2$. Then, we add the following constraints for check j:

$$f_2 \pmod 2 = f_3 + f_8 + f_{10} \pmod 2 \tag{4.53}$$

$$f_2 + f_3 \pmod 2 = f_8 + f_{10} \pmod 2 \tag{4.54}$$

$$f_2 + f_3 + f_8 \pmod 2 = f_{10} \pmod 2 \tag{4.55}$$

When we are at a new branch, we add the branching rule as a constraint to the constraint programming model.

Constraints 3: Assume that we are at a branch $f_i = 0$ for $i \in N_0$ and $f_i = 1$ for $i \in N_1$. Then the branch constraints are

$$f_i = 0 \quad i \in N_0 \tag{4.56}$$

$$f_i = 1 \quad i \in N_1 \tag{4.57}$$

The objective is to minimize the Hamming distance as given in (4.1).

We would like to find a better upper bound at each call of the constraint programming algorithm.

Constraint 4: We force that the objective should be less than the best known upper bound, say $\bar{z}$.

$$-\sum_{i:\hat{y}_i=1} f_i + \sum_{i:\hat{y}_i=0} f_i + c_1 < \bar{z}. \tag{4.58}$$

We utilize Constraint Programming (CP) tool of CPLEX for the implementation. We set a 1 minute time limit for CP and we run it until we find 2 feasible solutions.

Table 4.17. Performance of Constraint for $(s, 3, 6)$ codes (in seconds).

	CPLEX			BP with Constraint				
s	z^*	CPU	Nodes	z^*	CPU	Nodes	Impr	Call
2	0	0,16	0	0	0,38	0	1	1
3	3	0,15	0	3	0,35	0	1	1
4	4	0,13	0	4	0,38	0	1	1
5	3	0,15	0	3	0,37	0	1	1
6	4	0,21	0	4	0,73	0	1	2
7	5	0,24	40	5	27,38	26	3	27
8	6	0,25	393	6	5,99	4	3	5
9	7	0,16	11	7	321,19	22	1	24
10	8	0,20	16	8	*time*	12	1	14
11	9	0,17	0	9	540,76	8	2	10
12	8	0,20	0	10	*time*	13	5	14
13	9	0,21	193	11	*time*	13	4	14
14	10	0,48	390	10	*time*	14	5	15
15	11	0,20	0	11	*time*	14	5	15
16	12	0,34	1294	14	*time*	15	6	16
17	13	0,71	819	15	*time*	16	7	17
18	12	0,12	0	16	*time*	15	7	16
19	13	0,19	11	15	*time*	16	8	17
20	14	0,29	124	18	*time*	14	7	15
21	15	0,48	52	21	*time*	16	9	17
Avg:	8,3	0,3	167,2	9,5	99,7	10,9	3,9	12,1

In order to explore different parts of the solution space at each call of the algorithm, we randomly select the variable to branch in the CP tree. We apply constraint programming when we are at a new branch or we have updated the upper bound. The results are given in Tables 4.17, 4.18 and 4.19. The CP algorithm is tried Call–many times and the upper bound is updated Impr–many times. The results show that this algorithm takes more time for (3, 6)–regular codes but it is more efficient than all other methods in denser codes to find a upper bound in the given time limit. Besides, it uses less nodes to solve the instances for denser codes. Again it is not better than CPLEX.

4.3.6. Simulated Annealing

Simulated annealing is one of the well known metaheuristics in the literature [49]. In our case, we will search around the n bit long received vector $\hat{\mathbf{y}}$. Among n bits of $\hat{\mathbf{y}}$, randomly selected $n/2$ bits are considered to be changed. The value of the bit is

Table 4.18. Performance of Constraint for $(s, 5, 10)$ codes (in seconds).

	CPLEX			BP with Constraint				
s	z^*	CPU	Nodes	z^*	CPU	Nodes	Impr	Call
2	2	0,14	0	2	0,37	0	1	1
3	3	0,18	0	3	0,53	0	1	1
4	6	0,29	0	6	507,41	668	3	669
5	5	0,34	1457	7	*time*	77	3	78
6	6	0,64	885	8	*time*	30	4	31
7	9	2,34	9332	13	*time*	12	4	13
8	8	1,66	1455	14	*time*	12	4	13
9	9	21,70	45240	41	*time*	0	1	1
10	10	25,94	85628	28	*time*	11	6	12
12	14	705,66	1823416	50	*time*	10	4	11
Avg:	7,2	75,9	196741,3	22	169,4	82	3,1	83

Table 4.19. Performance of Constraint for $(s, 6, 12)$ codes (in seconds).

	CPLEX			BP with Constraint				
s	z^*	CPU	Nodes	z^*	CPU	Nodes	Impr	Call
2	2	0,22	0	2	0,76	0	1	1
3	4	0,22	307	4	32,90	26	2	27
4	4	0,43	762	4	377,57	50	4	51
5	6	0,59	3326	10	*time*	33	5	34
6	8	11,22	33733	14	*time*	10	4	11
7	8	81,26	270502	20	*time*	10	5	11
8	10	99,33	440923	38	*time*	9	2	10
9	12	7326,56	15428940	46	*time*	9	2	10
Avg:	6,8	939,9	2022311,6	17,2	137,1	18,4	3,1	19,4

changed with probability 0.50. If the solution is feasible, we move to that solution and update the upper bound if it is improving. If the solution is infeasible, we move to that infeasible solution with a probability.

In Figure 4.12, state s is the current solution and energy e is its objective function value. Initial state is s_0 and the best solution is kept with s_{best} with objective function value e_{best}. The maximum number of iterations is limited with value $k_{max} = 100$. Initial temperature $T = 20$ is halved at every 10 iteration.

For computational experiments we generate (3,6)–regular codes from permutation codes with $n/15$ error bits. The code length changes from $n = 60$ to $n = 480$. BP algorithm has a time limit of 30 minutes. The heuristic is applied at each node of the

```
Input: An infeasible received vector, $\hat{\mathbf{y}}$
Initialize $s \leftarrow s_0, e \leftarrow E(s), s_{best} \leftarrow s, e_{best} \leftarrow e, k = 0$.
While $k < k_{max}$ and $e > e_{max}$
    $T$ = temperature($k/k_{max}$)
    $s_{new} = neighbor(s)$, $e_{new} = E(s_{new})$
    If $P(e, e_{new}, T) > random()$, Then
        $s = s_{new}$, $e = e_{new}$
    End If
    If $e_{new} < e_{best}$, Then
        $s_{best} = s_{new}$, $e_{best} = e_{new}$
    End If
    k = k+1
End While
Output: A feasible codeword, $s_{best}$.
```

Figure 4.12. Simulated annealing algorithm.

branch–and–price tree except the root node.

Table 4.20. Simulated Annealing with Permutation codes under $n/15$ error bits (in seconds).

	(s, 3, 6) CPLEX			(s, 3, 6) BP			
n	z^*	CPU	Nodes	z^*	CPU	CPU Node Heur	Nodes
60	4	0,16	0	4	0,15	0	0
120	8	0,46	198	8	0,25	0	0
180	12	0,22	0	12	0,36	0	0
240	16	0,23	0	16	0,61	0	0
300	20	0,20	0	138	*time*	62,66	25538
360	24	0,30	0	24	1,67	0	0
420	28	0,25	0	28	2,04	0	0
480	32	0,27	0	32	2,16	0	0
Avg:	18	0,26	24,75	32,75	225,91	7,83	3192,25

The results indicate that if the problem is solved at the root node the performance of CPLEX and BP are not very different. For $n = 300$ instance CGA cannot solve it at the root node and terminated with the 30 minutes time limit without finding the optimal solution. Observations reveal that solving subproblems with a parallel computing approach may improve the performace of BP when evaluating the nodes.

4.3.7. G Matrix Applications

A detailed analysis of the solution space of the problem indicates that as the codeword length n increases the number of possible solutions, 2^n, increases exponentially. On the other hand, the number of feasible solutions among them is very few. As an example, for a (3, 6)–regular code when the code length is $n = 30$, only the 0.012% of the solutions is feasible. Besides, for a well designed code it is desired that the distance between the feasible solutions is as large as possible. From this result, one can deduce that the probability to find a feasible solution around $\hat{\mathbf{y}}$ is not high. This also explains the poor performance of the above applied methods.

Another approach is to produce feasible solutions using the generator matrix $\mathbf{G}$. When a parity–check matrix $\mathbf{H}$ is given, carrying out elementary row operations under binary arithmetic, we can have a form $\mathbf{H} = [\mathbf{A}|\mathbf{I_{n-k}}]$ where $\mathbf{A}$ is some $(n-k) \times k$ matrix of 0's and 1's, and $\mathbf{I_{n-k}}$ is $(n-k) \times (n-k)$ identity matrix. Then a $k \times n$ generator matrix $\mathbf{G} = [\mathbf{I_k}|\mathbf{A^T}]$ can be obtained using this $\mathbf{A}$ matrix. Since one can obtain different $\mathbf{A}$ matrices, the generator matrix $\mathbf{G}$ is not unique.

Each of the k rows of $\mathbf{G}$ is a feasible solution, since $\mathbf{G}\mathbf{H}^{\mathrm{T}} = \mathbf{0} \pmod 2$. From here we can see that any binary combination, $\mathbf{u}$, of the rows of $\mathbf{G}$ is also a feasible solution, since $\mathbf{u}\mathbf{G}\mathbf{H}^{\mathrm{T}} = \mathbf{0} \pmod 2$. Moreover, $\mathbf{G}$ is a basis for the solution space of $\mathbf{v}\mathbf{H}^{\mathrm{T}} = \mathbf{0}$ (mod 2). That is any feasible solution can be written as a binary combination of the rows of $\mathbf{G}$.

The decoding problem can be alternatively formulated by using $\mathbf{G}$ matrix. In Best Combination Model (BCM) given below, variable $\mathbf{x}$ represents the binary combination of the rows of $\mathbf{G}$ matrix. The binary representation of this combination can be given by variable $\mathbf{v}$ and we are minimizing the distance of $\mathbf{v}$ to the received vector $\hat{\mathbf{y}}$ in the objective.

Best Combination Model (BCM):

$$\min \sum_{i \in V} \gamma_i v_i \tag{4.59}$$

$$\text{s.t.}$$

$$\mathbf{G}^{\mathrm{T}}\mathbf{x} + \mathbf{v} = 2\mathbf{s} \tag{4.60}$$

$$\mathbf{x} \in \mathbb{B}^k,\ \mathbf{v} \in \mathbb{B}^n, s_i \in \mathbb{Z}^+,\ \forall i \in V. \tag{4.61}$$

This formulation is as hard as the EM for CPLEX. However, we make use of this formulation in order to find an upper bound for our BP algorithm by giving a time limit. The heuristic approaches that are making use of **G** matrix summarized in the following subsections. Three different methods are applied, namely Random Sum Heuristic, Sum Pass Heuristic and Best Combination Heuristic.

4.3.7.1. Random Sum Heuristic. Random sum heuristic in Figure 4.13 randomly combines the rows of generator matrix **G** and produce a new feasible solution. In Figure 4.13, k_{max} represents the maximum number of trials and in our application it is set to $k_{max} = 1000$. In order to speed up the row sums and objective function calculation, *BitArray* data structure is utilized.

```
Input: A generator matrix, $\mathbf{G}$, a received vector $\hat{\mathbf{y}}$
Initialize $\mathbf{z}^* = \infty, \mathbf{y}^*, k_{max}$.
While $k < k_{max}$
    Randomly set $\mathbf{u}_i$ from $\{0, 1\}$ for $i = 1, ..., n$.
    Obtain a feasible solution by $\mathbf{v} = \mathbf{uG}$.
    Calculate the objective function value $\mathbf{z_v}$ of solution $\mathbf{v}$.
    If $\mathbf{z_v} < \mathbf{z}^*$, Then
        $\mathbf{z}^* = \mathbf{z_v}$, $\mathbf{y}^* = \mathbf{v}$
    End If
    k = k+1
End While
Output: A feasible codeword $\mathbf{y}^*$ with objective value $\mathbf{z}^*$.
```

Figure 4.13. Random sum algorithm.

For the computational experiments, we generate (5,10)–regular codes from permutation codes with code length from $n = 60$ to $n = 480$. The random sum heuristic is also used to at the beginning of the BP in order to find a tight upper bound.

Table 4.21. Random Sum with Permutation codes under 5 error bits (in seconds).

	(s, 5, 10) CPLEX			(s, 5, 10) BP				
n	z^*	CPU	Nodes	z^*	CPU	CPU Node Heur	Nodes	Initial UB
60	4	0,41	816	4	787,96	126,13	6966	16
120	4	0,02	0	4	1,66	0,06	0	40
180	3	0,02	0	3	3,13	0,13	0	63
240	4	0,02	0	4	5,13	0,24	0	88
300	5	0,03	0	5	7,47	0,36	0	117
360	5	0,03	0	5	8,69	0,49	0	141
420	5	0,05	0	5	11,07	0,63	0	165
480	4	0,04	0	4	15,46	0,89	0	192
Avg:	4,25	0,08	102	4,25	105,07	16,12	870,75	102,75

The CPU time of the BP is increasing as the size of the instance increases when it is solving at the root node. If the problem cannot be solved at the root node the time gets very large compared to the performance of CPLEX. For $n = 60$ case Figure 4.14 shows how the upper and lower bounds are updated during the BP iterations. It is clear from the figure that there is a need for a tight upper bound for early prunning.

4.3.7.2. Best Combination Heuristic. In random sum method, we are randomly generating feasible solutions. However, the main problem is to find nearest feasible solution to the received codeword $\hat{\mathbf{y}}$. We have modeled this problem with BCM formulation. In BCM formulation, it may possible to limit the number of rows that are used in producing new solution, say K, by adding the following constraint:

$$\sum_{j=1}^{k} x_j \leq K. \tag{4.62}$$

In order to understand the affect of K on the objective function value, we carry out analysis with (3,6)–regular codes with different code lengths, i.e. $n = 60$, 120, 180,

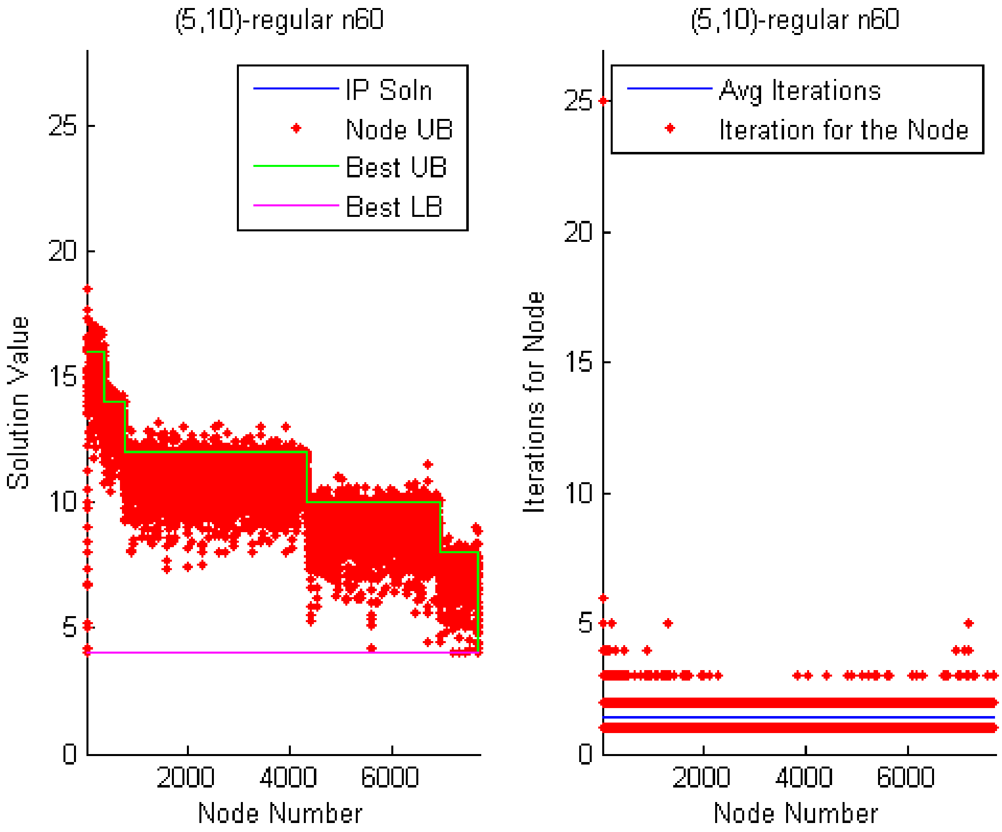


Figure 4.14. (5, 10)–regular $n = 60$ BP iterations.

240. The results are given in Figure 4.15 below. For each instance, we generate 10 different $\mathbf{G}$ matrices and generate a solution pool with the rows of these matrices. In $n = 60$ case, we have 300 many solutions in the pool. Then we try $K = 300, 150$ and 75 values. The results indicate that the limiting value of the K has not a significant affect on the objective function value, since for $n = 60$ the smallest K value is better but for $n = 240$ the largest K value is better.

Then, we have decided not to use constraint (4.62) and use only one generator matrix $\mathbf{G}$, since it is suffient to represent any feasible solution. Best combination heuristic in Figure 4.16 runs with 10 minutes time limit. Computational experiments in Table 4.22 show that the initial upper bound is improved compared with the random sum heuristic for $n = 60$.

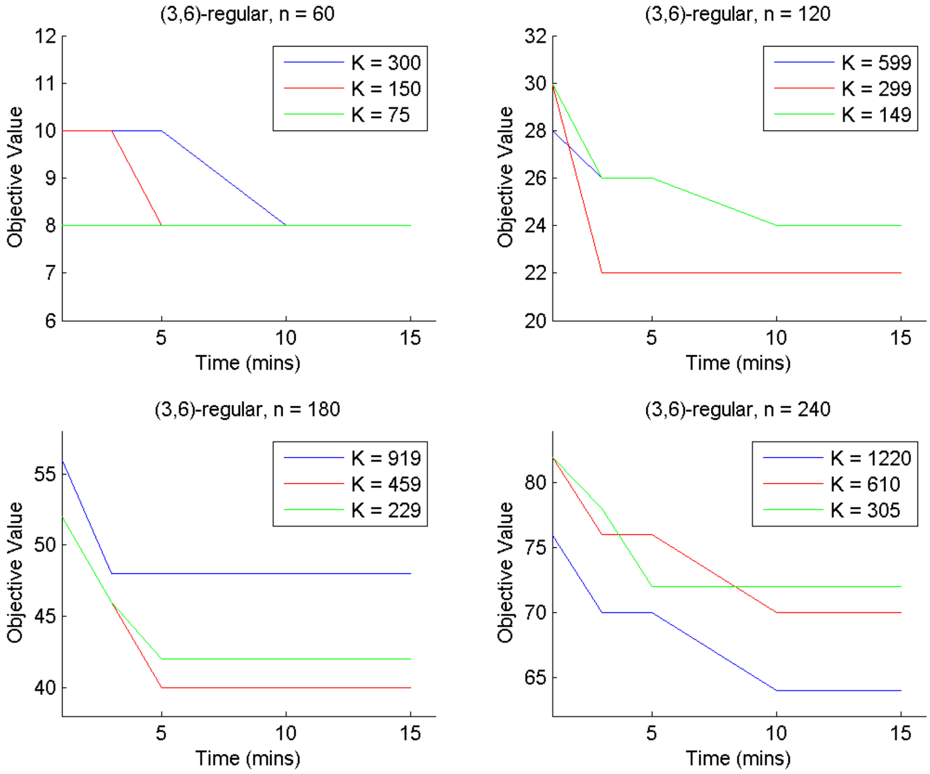


Figure 4.15. *K* value analysis.

```
Input: A generator matrix, $\mathbf{G}$, a received vector $\hat{\mathbf{y}}$
0. Initialize $\mathbf{z}^* = \infty, \mathbf{y}^*, timeLim$.
1. Solve BCM with CPLEX in the given $timeLim$.
   Let $\mathbf{v}$ is the solution with objective value $\mathbf{z_v}$.
2. If $\mathbf{z_v} < \mathbf{z}^*$, Then
3.     $\mathbf{z}^* = \mathbf{z_v}$, $\mathbf{y}^* = \mathbf{v}$
4. End If
Output: A feasible codeword $\mathbf{y}^*$ with objective value $\mathbf{z}^*$.
```

Figure 4.16. Best combination algorithm.

Table 4.22. Best Combination with Permutation codes under 5 error bits (in seconds).

	(s, 5, 10) CPLEX			(s, 5, 10) BP				
n	z^*	CPU	Nodes	z^*	CPU	CPU Node Heur	Nodes	Initial UB
60	4	0,41	816	4	807,34	126,13	6958	8
120	4	0,02	0	4	1,83	0,06	0	40
180	3	0,02	0	3	3,30	0,13	0	63
240	4	0,02	0	4	4,62	0,24	0	88
300	5	0,03	0	5	7,45	0,36	0	117
360	5	0,03	0	5	8,59	0,49	0	141
420	5	0,05	0	5	11,64	0,63	0	165
480	4	0,04	0	4	16,61	0,89	0	192
Avg:	4,25	0,08	102	4,25	107,67	16,12	869,75	101,75

4.3.7.3. Sum Pass Heuristic. Sum pass heuristic in Figure 4.17 starts with an initial feasible solution, takes $\mathbf{0}-$ codeword by default. If the sum of the solution with a row of $\mathbf{G}$ has a better objective function value the next solution is updated with this one. In the next iteration the sum of the solution and another row of $\mathbf{G}$ is taken. The algorithm continues as the objective function value improves. This approach improves the current feasible solution in a greedy fashion. Computational results are similar to the above cases.

```
Input: A generator matrix, $\mathbf{G}$, a feasible codeword $\mathbf{y}$
Initialize $\mathbf{y}_{best} = \mathbf{y}, \mathbf{z}_{best} = \mathbf{z}_{\mathbf{y}}, \mathbf{y}_{temp}, \mathbf{z}_{temp}$, $impr = true$.
While $impr$
    $impr = false$
    ForEach row in $\mathbf{G}$
      $\mathbf{y}_{temp} = \mathbf{y} +$ row of $\mathbf{G}$
      If $\mathbf{z}_{temp} < \mathbf{z}_{best}$, Then
        $impr = true$, $\mathbf{z}_{best} = \mathbf{z}_{temp}$, $\mathbf{y}_{best} = \mathbf{y}_{temp}$.
      End If
      If $impr$, Then $\mathbf{z}_{\mathbf{y}} = \mathbf{z}_{best}, \mathbf{y} = \mathbf{y}_{best}$.
    End ForEach
End While
Output: A feasible codeword $\mathbf{y}$ with objective value $\mathbf{z}_{\mathbf{y}}$.
```

Figure 4.17. Sum pass algorithm.

4.3.8. Diving Heuristic

If the optimum solution cannot be found at the root node of the BP algorithm, it is possible to fix some of the bits of the fractional solution to integer values if the bits are so close to 0 or 1. The problem can be resolved with these fixed bits in a continuous fashion until we end up with an integer feasible solution or an infeasibility. Compared with the simulated annealing results, diving algorithm in Figure 4.18 evaluates more nodes. This may since diving heuristic takes less time compared with the simulated annealing.

```
Input: A fractional solution $\mathbf{y}$ found at the root node
1. While true
2.     If $y_i < 0.01$, Then $y_i = 0$, If $y_i > 0.99$, Then $y_i = 1$.
3.     Solve the model with these bounds with CPLEX.
4.     If feasible, Then
5.       If integer, Then Update $\mathbf{y}^*$, STOP.
6.       Else Update bounds, go to Step 4.
7.       End If
8.     Else (infeasiblity), STOP.
9.     End If
10. End While
Output: A feasible codeword $\mathbf{y}^*$ or infeasibility.
```

Figure 4.18. Diving algorithm.

Table 4.23. Diving with Permutation codes under $n/15$ error bits (in seconds).

	(s, 3, 6) CPLEX			(s, 3, 6) BP			
n	z^*	CPU	Nodes	z^*	CPU	CPU Node Heur	Nodes
60	4	0,16	0	4	0,39	0	0
120	8	0,46	198	8	0,31	0	0
180	12	0,22	0	12	0,41	0	0
240	16	0,23	0	16	0,63	0	0
300	20	0,20	0	138	*time*	0,06	26374
360	24	0,30	0	24	1,94	0	0
420	28	0,25	0	28	2,11	0	0
480	32	0,27	0	32	2,22	0	0
Avg:	18	0,26	24,75	32,75	226,00	0,01	3296,75

5. LDPC CONVOLUTIONAL CODE DECODING

5.1. Introduction

In this part of the thesis, we focus on LDPC convolutional codes. Convolutional codes divide the original information into smaller blocks and decode each block by considering the previous blocks [50]. In the code, the nonzero elements are located on the diagonal as a ribbon and the code has infinite dimension. As given in Figure 5.1 below, a convolutional code consists of m_{s}–many small parity–check matrices at each column, where m_{s} parameter represent the width of the ribbon. The diagonal pattern is obtained by shifting the columns down as the dimension increases.

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_0(1) & & & & \\ \mathbf{H}_1(1) & \mathbf{H}_0(2) & & & \\ \vdots & \mathbf{H}_1(2) & \ddots & & \\ \mathbf{H}_{m_{\mathrm{s}}}(1) & \vdots & \ddots & \mathbf{H}_0(L) & \\ & \mathbf{H}_{m_{\mathrm{s}}}(2) & \ddots & \mathbf{H}_1(L) & \ddots \\ & & & \vdots & \ddots \\ & & & \mathbf{H}_{m_{\mathrm{s}}}(L) & \ddots \end{bmatrix}$$

Figure 5.1. Generic structure of a convolutional code.

These codes find application areas such as satellite communication and video streams where the information is received continuously. Finite dimension convolutional codes, namely spatially–coupled (SC) codes, can be obtained by limiting the dimension of convolutional code by specifying a finite row or column size [51]. In Figure 5.2, an example of finite dimensional (3, 6)–regular LDPC SC code obtained by limiting the row size is given.

The term “(3, 6)–regular” means that there are exactly 3 ones at each column and 6 ones at each row of **H** matrix. This regular structure cannot be seen for the first and the last parts of the code. For example, the number of ones for the first five

$$\mathbf{H} = \begin{bmatrix}
010000000000000000000000000000000000 \\
101000000000000000000000000000000000 \\
010101000000000000000000000000000000 \\
101010010000000000000000000000000000 \\
011001100100000000000000000000000000 \\
100110011001000000000000000000000000 \\
000101101010010000000000000000000000 \\
000010010101101000000000000000000000 \\
000000100110010101000000000000000000 \\
000000001001101010010000000000000000 \\
000000000010011001100100000000000000 \\
000000000000100110011001000000000000 \\
000000000000001011010100010000000000 \\
000000000000000010011001101000000000 \\
000000000000000000100110010101000000 \\
000000000000000000001001101010010000 \\
000000000000000000000010011001100100 \\
000000000000000000000000100110011001
\end{bmatrix}$$

Figure 5.2. A (3, 6)–regular LDPC SC code.

rows of the (3, 6)–regular code in Figure 5.2 is less than 6. Similarly, number of ones is less than 3 in the last nine columns of the code. One can observe the (3, 6)–regular structure for the intermediary rows and columns.

The repeating structure of the convolutional codes allow the application of sliding window decoding approaches which use decoding algorithms such as Viterbi algorithm [11] at each window. Viterbi algorithm finds maximum likelihood estimate of the received vector, but it is based on exhaustive enumeration. Heuristic approaches like Gallager A and B algorithms are easily applicable, however they cannot guarantee that the solution is near optimal. They may even fail to decode if the received vector includes errors.

Our goal in this study is to develop algorithms to decode a finite length received vector encoded with SC codes on BSC. Then we generalize these decoding algorithms to practically infinite length received vectors that are encoded with convolutional codes.

Our proposed decoders can give a near optimal feasible decoding for any real sized received vector in acceptable amount of time.

5.2. SC Code Generation

We implement the SC code generation scheme given in [52] which is also explained in this section. We generate an SC code with the help of a base permutation matrix. As shown in Figure 5.3, by randomly permuting the columns of an $s \times s$ identity matrix $\mathbf{I}_s$, we can obtain a (5, 10)–regular base permutation matrix of dimension $(m, 2m)$ where $m = 5 \times s$. Regularity of the matrix is provided through augmenting identity matrices 10 times at each row and 5 times at each column. In Figure 5.3, $\mathbf{I}_s^i$ represents the ith randomly permuted identity matrix.

$$\mathbf{H}_{base} = \begin{bmatrix} \mathbf{I}_s^1 & \mathbf{I}_s^2 & \mathbf{I}_s^3 & \mathbf{I}_s^4 & \mathbf{I}_s^5 & \mathbf{I}_s^6 & \mathbf{I}_s^7 & \mathbf{I}_s^8 & \mathbf{I}_s^9 & \mathbf{I}_s^{10} \\ \mathbf{I}_s^{11} & \mathbf{I}_s^{12} & \mathbf{I}_s^{13} & \mathbf{I}_s^{14} & \mathbf{I}_s^{15} & \mathbf{I}_s^{16} & \mathbf{I}_s^{17} & \mathbf{I}_s^{18} & \mathbf{I}_s^{19} & \mathbf{I}_s^{20} \\ \mathbf{I}_s^{21} & \mathbf{I}_s^{22} & \mathbf{I}_s^{23} & \mathbf{I}_s^{24} & \mathbf{I}_s^{25} & \mathbf{I}_s^{26} & \mathbf{I}_s^{27} & \mathbf{I}_s^{28} & \mathbf{I}_s^{29} & \mathbf{I}_s^{30} \\ \mathbf{I}_s^{31} & \mathbf{I}_s^{32} & \mathbf{I}_s^{33} & \mathbf{I}_s^{34} & \mathbf{I}_s^{35} & \mathbf{I}_s^{36} & \mathbf{I}_s^{37} & \mathbf{I}_s^{38} & \mathbf{I}_s^{39} & \mathbf{I}_s^{40} \\ \mathbf{I}_s^{41} & \mathbf{I}_s^{42} & \mathbf{I}_s^{43} & \mathbf{I}_s^{44} & \mathbf{I}_s^{45} & \mathbf{I}_s^{46} & \mathbf{I}_s^{47} & \mathbf{I}_s^{48} & \mathbf{I}_s^{49} & \mathbf{I}_s^{50} \end{bmatrix}$$

Figure 5.3. (5, 10)–regular base permutation matrix.

Then, we split the base permutation matrix into two matrices, namely lower triangular **A** and upper triangular **B** as shown in Figure 5.4.

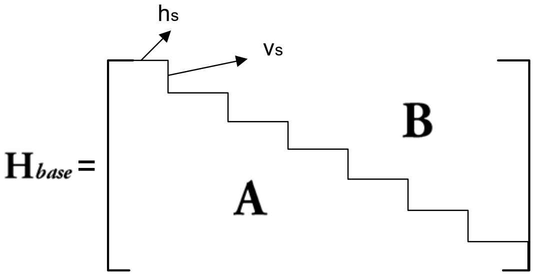


Figure 5.4. **A** and **B** matrices.

We divide $\mathbf{H}_{base}$ with a horizontal step length h_s and a vertical step length v_s. One can observe that when $\mathbf{H}_{base}$ is (5, 10)–regular, its dimension is $(m, 2m)$ for some

m. Then, $r = h_s/v_s = 2$, since there is a number c such that $h_s c = 2m$ and $v_s c = m$.

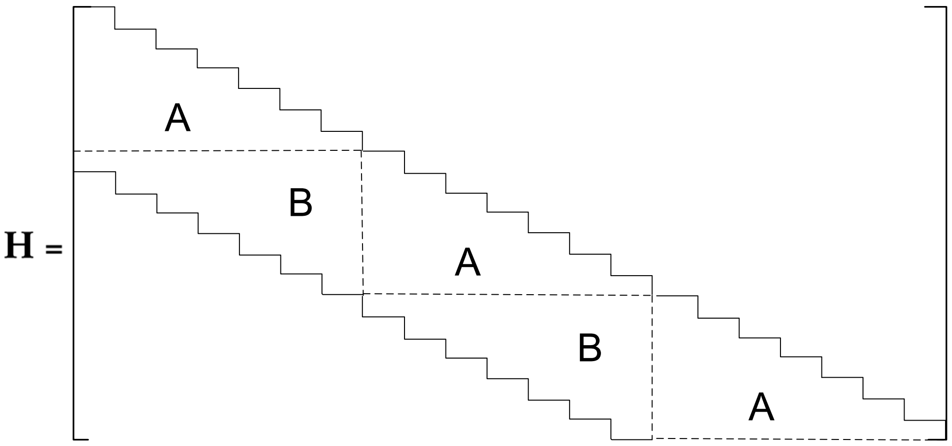


Figure 5.5. (5, 10)–regular SC code.

Then, these **A** and **B** matrices are repeatedly located until the desired SC code size is obtained. After t–many repetitions, SC code has size $(tm, 2tm)$ as shown in Figure 5.5. The ribbon size is $m_s = m + v_s$ for such a code.

5.3. Sliding Window Decoders

Sliding window decoders in practical applications make use of special structure of the convolutional codes [53, 54]. As explained above, convolutional codes have all nonzero entries on a ribbon, with width m_s, that lies on the diagonal. Then, one can consider a window on the convolutional code with height w and decode the received vector partially. Decoding of the received vector proceeds iteratively by sliding the window h_s units horizontally and v_s units vertically.

In sliding window decoders, we can pick window row size $w > m_s$ and column size larger or equal to rw where $r = h_s/v_s$. For the rows of the convolutional code corresponding to the window, all entries in the columns after the window are zero with this window dimension selection.

Figure 5.6 explains the main steps of a generic sliding window decoder. Part of the received vector corresponding to the current window is decoded with an algorithm. Hence, performance of a sliding window decoder depends on how fast and correctly the windows are decoded. As we mention in Chapter 2, Viterbi algorithm adversely affects decoding time when used in window decoding. In the case we implement Gallager A or B algorithm for windows, decoded vector may not close to the original information. We investigate the performance of Gallager A and B algorithms in sliding window decoder with computational experiments in Section 5.8.

```
Input: Received vector $\hat{\mathbf{y}}$, Binary code $\mathbf{H}$
1. Decode the current window with an algorithm.
2. Move the window $h_s$ units horizontally, $v_s$ units vertically.
3. Fix the decoded values of $h_s$–many leaving bits.
4. If all bits decoded, Then STOP, Else go to Step 1.
Output: A decoded codeword.
```

Figure 5.6. Generic sliding window algorithm.

In our approach, we solve each window with EM formulation that is written for the decision variables and constraints within the window. At each iteration, h_s–many bits and v_s–many constraints leave the window. Exiting bits are decoded in the previous window and can be fixed to their decoded values in the proceeding iterations. The decoded bits will affect the upcoming bits by appearing as a constant in the constraints (4.2). Our sliding window decoding algorithm has main steps that are given in Figure 5.7.

```
Input: Received vector $\hat{\mathbf{y}}$, Binary code $\mathbf{H}$
1. Solve EM for the current window.
2. Move the window $h_s$ units horizontally, $v_s$ units vertically.
3. Fix the decoded values of $h_s$–many leaving bits.
4. Update constraints (4.2) with the fixed bits.
5. If all bits decoded, Then STOP, Else go to Step 1.
Output: A decoded codeword.
```

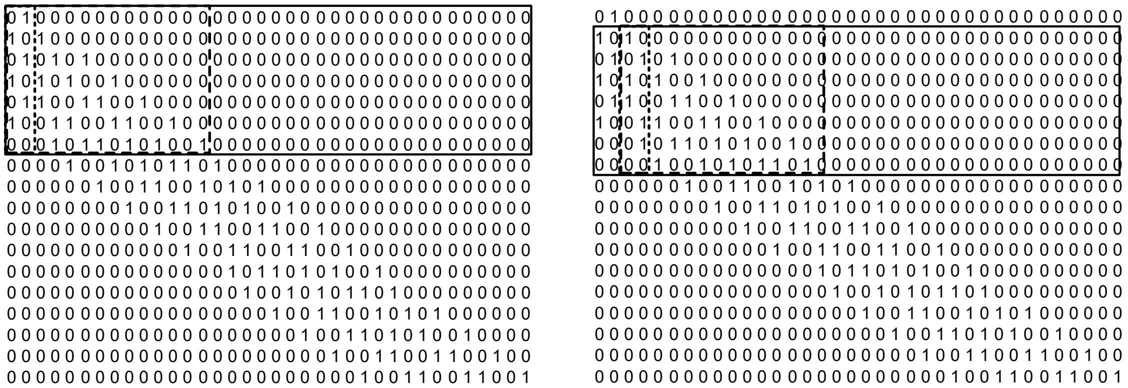
Figure 5.7. Sliding window algorithm.

It is possible to apply different strategies in window dimension selection and window solution generation. This gives rise to our four different sliding window decoders, i.e. complete window, finite window and repeating windows decoders for SC codes and a convolutional code decoder, that are explained in the next sections.

5.4. Complete Window (CW) Decoder

Complete window (CW) decoder requires that binary code has finite dimension. Hence, it is applicable for SC–Convolutional codes. In CW, the window height is w and width is n (the length of the received vector $\hat{\mathbf{y}}$). This means in a window we have w–many constraints and n–many bits as f_i decision variables.

We consider two diffrent ways in window decoding. In the first approach, i.e. Some Binary CW (SBCW), we restrict the first undecoded h_s bits of the window to be binary and relax the bits coming after those as continuous variables. As an example, when we solve the first window of the code in Figure 5.8, first h_s bits (corresponding to the dotted rectangle) are binary and we relax all the remaining bits as continuous. When we move to the next window by shifting the window v_s units down, first h_s bits have been fixed to their decoded values, the next h_s bits are set to be binary and the bits coming after are continuous variables. The decoder proceeds in this fashion.



```
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 1 1 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 1 1 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 1 0 1 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 1 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 1 1 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 1 1 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 1 0 1 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 1 0 1 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 1 1 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 1 1 0 0 1
```

```
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 1 1 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 1 1 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 1 0 1 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 1 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 1 1 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 1 1 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 1 0 1 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 1 0 1 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 1 1 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 1 1 0 0 1
```

Figure 5.8. Sliding window in CW decoder.

One can see that the dashed rectangle in Figure 5.8 covers all nonzero entries in the window. From this observation as a second approach, i.e. All Binary CW (ABCW), we consider to force the first undecoded (rw)–many bits (corresponding to the dashed rectangle) of the window to be binary and the ones after these are continuous. As we move to the next window, h_s–many bits are fixed and the dashed rectangle shift to right h_s units. Moving from one window to the other requires removing first v_s–many constraints and including new v_s–many constraints.

The method of fixing some of the decision variables and relaxing some others is known as Relax–and–Fix heuristic in the literature [55, 56]. In general, fixing the values of the variables may lead to infeasibility in the next iterations. However, we do not observe such a situation in our computational experiments when we pick the window that is sufficiently large to cover all nonzero entries for the undecoded bits in the corresponding rows. We can observe that a window of size $w \times (rw)$ (dashed rectangle) can cover the undecoded nonzero entries.

5.5. Finite Window (FW) Decoder

In finite window (FW) decoder, we have smaller window of size $w \times (rw)$. That is we have w–many constraints and (rw)–many f_i decision variables. At each iteration, after solving EM model for the window, we fix first h_s–many bits and slide the window. In Some Binary FW (SBFW) decoder, we restrict first h_s–many bits to be binary and relax the rest as continuous. For All Binary FW (ABFW) method, all (rw)–many bits are binary variables.

The window position can be seen in Figure 5.9 as the window slides. The previous decoded bits appear as a constant in constraints (4.2) of EM formulation for the current window. In FW, we store only one window model. This means we are storing w–many constraints and (rw)–many f_i decision variables in the memory at a time.

As we move from one window to the other, we remove h_s–many decision variables and introduce h_s–many new ones. Also, we remove v_s–many constraints and add v_s–

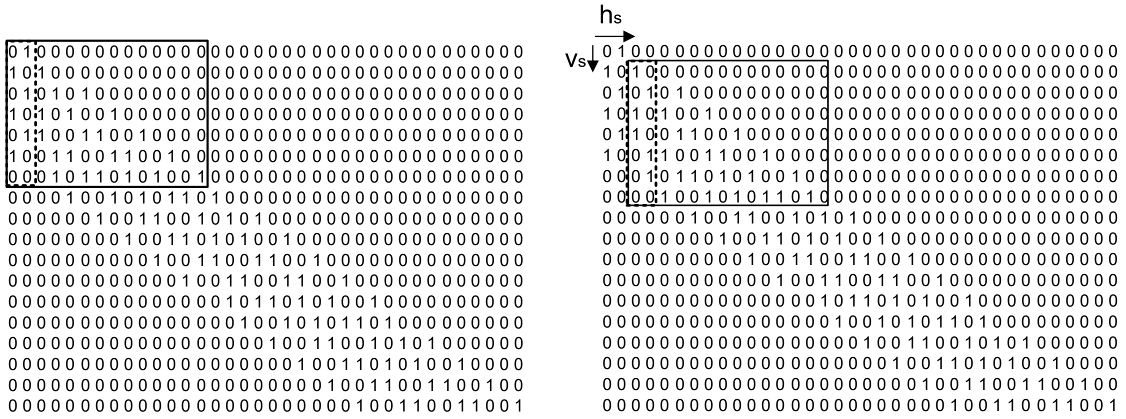


Figure 5.9. Sliding window in FW decoder.

many new constraints.

5.6. Repeating Windows (RW) Decoder

As explained in Section 5.2, an SC code is obtained by repetitively locating **A** and **B** matrices. As can be seen in Figure 5.10, a window will come out again after m–iterations, where m is the number of rows in $\mathbf{H}_{base}$.

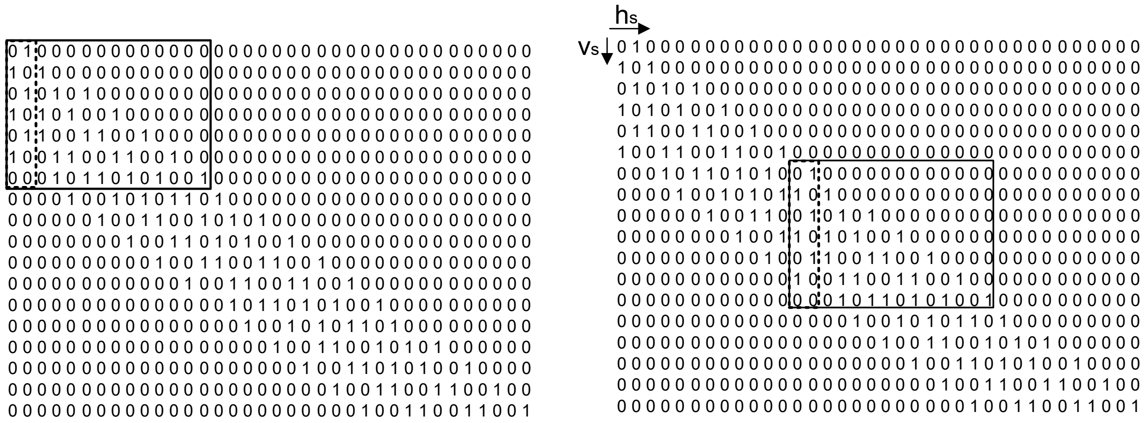


Figure 5.10. Sliding window in RW decoder.

This means that there are m–many different windows. However, the first and the $(m+1)$st windows still differ from each other in terms of their EM formulation. That is the constant term in constraints (4.2) and the objective function coefficients change but the coefficients of the decision variables stay same. Hence, we store m–many window models and when its turn comes we solve the window after updating the constant term and the objective function.

Assuming that a window is of size $w \times (rw)$, having m–many window models requires to store (mw)–many constraints and (mrw)–many f_i decision variables in the memory. However, we do not need to add or remove constraints and decision variables. FW decoder has the burden of add/remove operations and the advantage of low memory usage. On the other hand, RW decoder directly calls the window models on the expense of memory.

In Some Binary RW (SBRW) only first h_s–many bits are binary, whereas All Binary RW (ABRW) has all (rw)–many bits as binary variables.

5.7. Convolutional Code (CC) Decoder

The decoders CW, FW and RW assume that we are given a finite dimensional code that can be represented by a **H** matrix. Hence, they are applicable for SC codes. However, as explained before, convolutional codes are practically infinite dimensional codes and cannot be represented by a compact **H** matrix on computer. On the other hand, they are generated from **A** and **B** matrices. Therefore, we can store a part of convolutional code as given in Figure 5.11 that includes the required information.

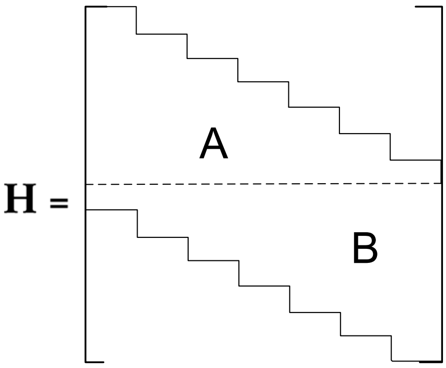


Figure 5.11. A part of convolutional code.

With this part of the convolutional code, we can represent the (i, j)th entry of the convolutional code with a function. Hence, we can represent the current window model using this small matrix. This allows the application of FW and RW decoders to convolutional codes. Note that our CW decoder is not applicable to CC, since it takes into account all bits of the received vector.

5.8. Computational Results

The computations have been carried out on a computer with 2.6 GHz Intel Core i5-3230M processor and 4 GB of RAM working under Windows 10 Professional.

In our computational experiments, we evaluate the performance of our sliding window decoders. In our decoders, the number of the constraints and decision variables in EM formulation limited with the size of the window. We make use of CPLEX 12.6.0 to solve EM for the current window (see Step 1 of Figure 5.7). We compare the performance of our sliding window decoders with Exact Model Decoder (EMD). In EMD, EM formulation includes all constraints and decision variables corresponding to the SC code. That is, for a SC code of size $(n/2, n)$ we have $n/2-$many constraints (4.2) and $n-$many f_i decision variables in EM. We again utilize CPLEX for solving EM of EMD.

Table 5.1. List of computational parameters.

Parameters	
n	1200, 3600, 6000, 8400, 12000
p	0.02 (low), 0.05 (high)
m	150
w	$m + 1$ (small), $\frac{3m}{2} + 1$ (large)
h_s	2
v_s	1

A summary of the parameters that are used in the computational experiments are given in Table 5.1. We generate a base permutation matrix of size $(m, 2m) = (150, 300)$. We obtain a (5, 10)–regular SC code **H** of desired dimensions from this base permutation matrix. In our experiments, we consider four different code length, i.e. $n = 1200, 3600, 6000, 8400$ for SC codes. In order to test the algorithms for convolutional codes, we consider a larger code length $n = 12000$. For each code length n, we experiment 10 random instances and report the average values. We investigate two levels of error rate, i.e. low error $p = 0.02$ and high error $p = 0.05$. There are two alternatives for the window sizes, namely small window $w = m + 1$ and large window $w = \frac{3m}{2} + 1$.

In our sliding window algorithms, we solve the window models with CPLEX within 1 minute time limit. On the other hand, we set a time limit of 4000 seconds to EMD for solving a SC code instance. Since we are testing a larger code length, i.e. $n = 12000$, for convolutional codes, we set a time limit of 5000 seconds to EMD to find a solution.

Table 5.2. Performance of EMD with $p = 0.02$ and 0.05.

p	0.02				0.05			
n	z	CPU	Gap (%)	# OPT	z	CPU	Gap (%)	# OPT
1200	23.9	0.16	0	10	56.3	221.42	0	10
3600	72.7	0.23	0	10	736.0	1797.72	35.44	6
6000	121.0	0.32	0	10	1358.3	3890.06	43.37	4
8400	169.9	0.54	0	10	3623.7	4049.98	80.45	1
12000	238.6	0.85	0	10	4300.6	4457.34	70.84	2

Table 5.2 gives the performance of EMD under low and high error rates. The column "z" shows the objection function value of the best known solution found within the time limitation. "CPU" is the computational time in terms of seconds. "Gap (%)" is the relative difference between the best lower and upper bounds. "# OPT" is the number of instances that are solved to optimality among 10 trials. The first four rows in Table 5.2 are average results for SC codes. The last row is the average result for convolutional code. As the error rate increases, EMD has difficulty in finding optimal solutions. A similar pattern is observed when the length of the received vector n increases. That is, the optimality gap increases when the code gets longer as expected.

5.8.1. SC Code Results

In this section, we discuss the results of the computational experiments of $n = 1200, 3600, 6000, 8400$ for error probabilities 0.02 and 0.05 and two levels of window size, i.e., small and large.

Table 5.3. Performance of SBCW.

	w	small				large			
p	n	z	CPU	Gap (%)	# SOLVED	z	CPU	Gap (%)	# SOLVED
0.02	1200	23.9	5.48	0	10	23.9	6.31	0	10
	3600	72.7	31.89	0	10	72.7	39.05	0	10
	6000	121.0	83.77	0	10	121.0	97.54	0	10
	8400	169.9	168.97	0	10	169.9	187.91	0	10
0.05	1200	56.3	26.70	0	10	107.3	334.28	9.03	10
	3600	181.8	224.30	2.83	10	1052.1	1056.68	53.46	10
	6000	564.2	1165.43	14.92	10	2504.2	1419.37	80.25	10
	8400	2243.75	2188.73	48.27	10	4016.5	1346.33	89.48	10

Tables 5.3 and 5.4 summarize the results for CW decoder explained in Section 5.4. “Gap (%)” column represents the percent difference from the best known lower bound found by CPLEX while obtaining the results in Table 5.2. “# SOLVED” column shows the number of instances that can be decoded by the method.

When $p = 0.02$, CW decoder can find optimal solutions as EMD in Table 5.2. However, CW completes decoding in longer time for both SB and AB variants and both window sizes. This is since solving EM model with CPLEX (in EMD) under low error probability is easy and decoding in small windows takes longer time in CW.

Table 5.4. Performance of ABCW.

	w	small				large			
p	n	z	CPU	Gap (%)	# SOLVED	z	CPU	Gap (%)	# SOLVED
0.02	1200	23.9	6.33	0	10	23.9	7.65	0	10
	3600	72.7	34.05	0	10	72.7	43.86	0	10
	6000	121.0	88.51	0	10	121.0	107.59	0	10
	8400	169.9	177.23	0	10	169.9	201.58	0	10
0.05	1200	58.3	17.85	2.59	10	56.3	57.01	0	10
	3600	181.8	67.58	2.83	10	614.9	494.62	26.82	10
	6000	392.9	537.07	16.81	10	1279.3	941.14	36.05	10
	8400	533.5	642.54	17.50	10	3119.4	761.03	67.13	10

When the error probability increases to 0.05 and window size is small, we can see that CW finds better feasible solutions in shorter time than EMD (in Table 5.2) for SB and AB variants. As the window size gets larger, only AB alternative gives better gap and time values compared with EMD.

In general, with high error probability AB takes shorter time and obtains better gaps than SB (see results for $p = 0.05$ in Tables 5.3 and 5.4, Tables 5.5 and 5.6, Tables 5.7 and 5.8). Note that this is somewhat counter intuitive since the number of binary variables in AB variant is larger than SB. However, note that AB has the advantage of being able to use the integral solution of the previous window as a starting solution of the new window. Hence, AB has more time to find a better solution in the current window within the time limit compared with SB.

When $p = 0.05$, the performance of CW deteriorates as the window size gets larger. Solving a larger model in a window decreases the quality of the solution obtained within the time limit. Size of the window model also depends on the length of the received vector n. Hence, the gap values increase as n increases.

Table 5.5. Performance of SBFW.

	w	small				large			
p	n	z	CPU	Gap (%)	# SOLVED	z	CPU	Gap (%)	# SOLVED
0.02	1200	23.9	10.91	0	10	23.9	11.90	0	10
	3600	72.7	24.25	0	10	72.7	47.47	0	10
	6000	121.0	41.98	0	10	121.0	74.04	0	10
	8400	169.9	62.48	0	10	169.9	129.44	0	10
0.05	1200	56.3	15.07	0	10	56.3	364.67	0	10
	3600	196.6	348.29	5.89	10	177.0	3581.46	0.78	10
	6000	353.9	973.76	12.89	10	300.3	6889.76	0.86	10
	8400	629.8	3561.43	26.65	10	427.0	11445.70	1.03	10

Results given in Tables 5.5 and 5.6 show that FW (see Section 5.5) can find optimal solution in all cases when $p = 0.02$. With this error probability, FW needs more time to find the optimal solution for SB and AB alternatives when the window size gets larger. The computational times are larger than EMD for both alternatives.

Table 5.6. Performance of ABFW.

w		small				large			
p	n	z	CPU	Gap (%)	# SOLVED	z	CPU	Gap (%)	# SOLVED
0.02	1200	23.9	6.59	0	10	23.9	15.80	0	10
	3600	72.7	22.95	0	10	72.7	65.66	0	10
	6000	121.0	39.22	0	10	121.0	114.59	0	10
	8400	169.9	56.45	0	10	169.9	165.71	0	10
0.05	1200	58.3	21.48	2.59	10	56.3	72.78	0	10
	3600	214.6	387.94	10.97	10	177.0	999.93	0.78	10
	6000	368.1	653.55	16.05	10	300.3	2087.41	0.86	10
	8400	617.8	1792.25	25.96	10	427.0	3061.82	1.03	10

However, as error probability gets higher, FW can find better solutions than EMD in shorter time for SB and AB methods. AB method is faster than SB, since it can make use of integral solution found in the previous window. Note that a similar pattern also appears in CW as discussed before.

FW takes more time than CW for both SB and AB alternatives, since it needs to add and remove variables while moving to the next window position. On the other hand, the size of the window model is independent from code length n, hence we can find better solutions within the time limit. As a result, the gap values are better than CW decoder.

We also observe that, at $p = 0.05$ increasing the window size improves the gap values in contrast to CW decoder. In FW decoder, although the window size does not depend on n, gap values still depend on n due to error accumulation during the iterations. That is, if a window is not decoded optimally, this near optimal window solution will propagate to the upcoming window decodings. As the code length n gets larger, this effect becomes more apparent and the gap values increases. If the window size is larger, then we are considering more information during the window decoding, which improves the gap values. This effect is explained graphically in Figure 5.12.

From Tables 5.7 and 5.8, we can see that RW cannot complete decoding at all cases. RW decoder stores m–CPLEX models in memory and CPLEX needs additional memory for branch–and–bound tree while solving the window model. Hence, when the

Table 5.7. Performance of SBRW.

	w	small				large			
p	n	z	CPU	Gap (%)	# SOLVED	z	CPU	Gap (%)	# SOLVED
0.02	1200	23.9	6.52	0	10	23.9	55.84	0	10
	3600	72.7	24.42	0	10	72.7	258.60	0	10
	6000	168.9	766.64	11.17	10	122.1	410.82	0	9
	8400	234.2	1088.18	7.86	10	168.8	438.66	0	7
0.05	1200	75.2	330.79	18.42	10	109.0	6542.20	35.10	10
	3600	269.9	2094.05	27.82	10	480.5	74868.29	62.15	5
	6000	628.5	7330.18	50.52	10	–	–	–	0
	8400	999.0	13140.1	57.29	10	–	–	–	0

window size gets larger, we see that memory is not sufficient to complete the iterations for some instances.

Table 5.8. Performance of ABRW.

	w	small				large			
p	n	z	CPU	Gap (%)	# SOLVED	z	CPU	Gap (%)	# SOLVED
0.02	1200	23.9	6.25	0	10	23.9	8.77	0	10
	3600	72.7	24.78	0	10	72.7	38.88	0	10
	6000	121.0	42.41	0	10	121.0	67.53	0	10
	8400	169.9	61.40	0	10	169.9	98.08	0	10
0.05	1200	56.3	10.24	0	10	56.3	155.94	0	10
	3600	217.6	674.08	11.35	10	175.9	1498.80	0.68	8
	6000	369.1	845.82	16.20	10	300.3	5067.35	0.86	10
	8400	616.2	2806.38	25.81	10	432.5	9445.67	1.23	8

Comparison of Tables 5.6 and 5.8 shows that ABFW and ABRW methods give similar gap values as expected. However, ABRW method requires more time to manage window models. As the window size gets larger, the computational time of ABRW is even worse than EMD (in Table 5.2) with high error probability.

5.8.2. Convolutional Code Results

We also investigate the performance of FW and RW decoders for very large code length. For this purpose we take $n = 12000$ and consider high ($p = 0.05$) error probability, small and large window sizes. CW method is inapplicable in practice for very large code lengths, since it includes all the bits of the codeword as a decision variable to the window model. Performance of EMD for $n = 12000$ is given in the last row of Table 5.2.

Table 5.9. Performances of FW and RW decoders.

	w	small				large			
		z	CPU	Gap (%)	# SOLVED	z	CPU	Gap (%)	# SOLVED
FW	SB	926.6	5796.79	30.93	10	597.6	14749.46	0.77	10
	AB	990.2	3686.91	34.03	10	597.6	3999.73	0.77	10
RW	SB	1434.1	20013.34	58.21	10	–	–	–	0
	AB	907.0	4537.75	27.74	10	596.4	5428.56	0.45	5

Table 5.9 summarizes the average results of 10 instances for FW and RW decoders with SB and AB alternatives. When we have small window size, all methods can decode the received vector. Among all, ABFW completed decoding within shortest time.

When the window size gets larger, RW decoder cannot solve all instances due to memory limit. On the other hand, FW decoder can solve all instances with better gap values compared with the small window size. ABFW takes less time by making use of integral starting solution advantage over SBFW. Moreover, compared with the EMD (last row of Table 5.2), ABFW finds near optimal solutions in shorter time for all instances. However, EMD can solve only 2 instances to optimality. For the 5 instances that ABRW can decode, ABFW and ABRW get the same objective values. For these cases, ABFW is faster than ABRW as expected.

Considering the computational results for convolutional codes, we can see that ABFW is the best alternative for decoding process in terms of both time and solution quality. We further evaluate the performances of the methods by analyzing their decoding errors with respect to the original vector as given in Figure 5.12.

In this figure, the average decoding errors of the 10 instances for code length $n = 12000$ with error probability $p = 0.05$ are given. We divide n into 120 sections each include 100 bits. For each section, average errors from the original code vector is plotted. When the window size is small, average error gets larger as the iterations proceeds. That is when we make error in decoding in early steps of the decoding process, this error will increase the probability that we are decoding erroneously in the upcoming windows. On the other hand, when the window size gets larger, we have

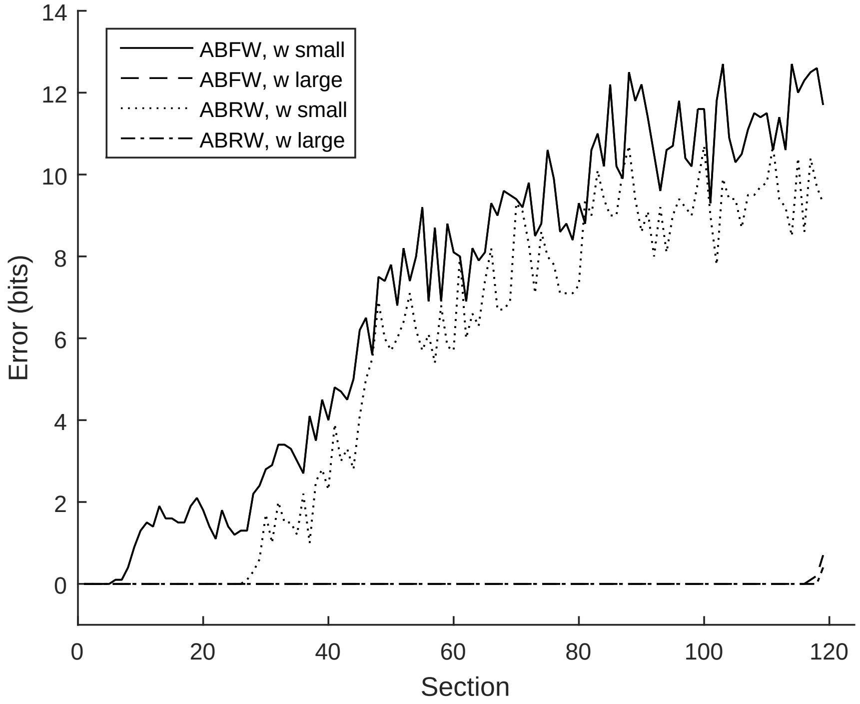


Figure 5.12. Error accumulation in decoding.

more information about the convolutional code, which decreases the error accumulation during the iterations. However, taking a large window size requires more decoding time. As a result, one should take into account the trade off between computational time and the solution quality when deciding on the window size.

The performance of decoding algorithms are interpreted with Bit Error Rate (BER) in telecommunications literature. BER is the percentage of the decoded bits that are different than the original vector [44].

$$BER = \frac{\sum_{i=0}^{n} \mid y_i^o - y_i^d \mid}{n} \times 100 \tag{5.1}$$

BER can be calculated with the formula given in equation (5.1), where $\mathbf{y^o}$ is the original and $\mathbf{y^d}$ is the decoded codeword.

Table 5.10. BER of Sliding Window Decoders.

w	ABFW	ABRW
small	6.918	5.402
large	0.008	0.003

We can calculate the BER values for our decoding algorithms using the data of Figure 5.12. The BER results given in Table 5.10 show that the error correction capability increases when we have larger window. For example, among 100 bits of the codeword that is decoded by ABFW method, approximately 7 bits (% 6.918) are different from the original codeword when window size is small. As the window size gets larger, this difference drops to 8 bits among 100,000 bits (% 0.008).

In our final experiment, our goal is to compare our proposed decoding algorithms with two commonly used algorithms. In practical applications, decoding of a received vector is done with iterative algorithms. Among these Gallager A and B algorithms are popular due to their ease of application [42, 57]. The performance of our proposed decoding algorithm (ABFW) can be tested against a sliding window decoder that uses Gallager A or B algorithm for decoding windows (see Figure 5.6).

Gallager A and B algorithms are quite similar. For each bit of the received codeword $\hat{\mathbf{y}}$, the algorithm collects messages, which are the values of the parity–check equations, from each check node. If the neighboring check node is unsatisfied, then this is considered as an indication of an error in the corresponding bit. If most of the neighbors of a bit are unsatisfied, we have a strong intuition that the bit is erroneous. Let d_i be the number of neighbors of variable node i in the Tanner graph of the code.

As given in Algorithm 3.5, Gallager A algorithm prefers to flip the bit that has the maximum number of unsatisfied checks. At each iteration of the algorithm, we flip only one bit which guarantees that the number of unsatisfied check nodes will decrease at each iteration. Gallager B algorithm decides whether to flip or not each bit at an iteration. For each bit, Gallager B flips the bit if the number of unsatisfied check nodes is larger than the satisfied ones. In Gallager B algorithm, decrease in the unsatisfied check nodes at each iteration is not for sure since it applies multi–flip at an iteration.

We apply Gallager algorithm at each window of the sliding window algorithm instead of solving window model with CPLEX. A known problem with these algorithms is that they may get stuck when there is a cycle in the LDPC code [58]. In such a case, the algorithm may terminate with no conclusion. To avoid such a situation, we take the stopping criterion as the number of iterations and bound it with value 100. Note that this may result in ending with an infeasible solution when the algorithm terminates.

Table 5.11. Performance of Gallager A.

	w	small					large				
p	n	z	CPU	Gap (%)	# FEAS	BER	z	CPU	Gap (%)	# FEAS	BER
0.02	1200	159.2	10.69	84.99	0	11.94	234.1	20.79	89.79	0	18.23
	3600	213.1	49.32	65.99	0	4.33	276.9	99.38	73.81	0	5.89
	6000	268.3	99.14	54.96	0	2.81	338.2	192.15	64.27	0	3.74
	8400	323.9	159.23	47.63	0	2.21	386.4	299.31	56.12	0	2.72
	12000	395.1	261.76	39.65	0	1.67	451.5	506.70	47.22	0	1.85
0.05	1200	191.3	11.44	70.62	0	15.43	259.1	20.74	78.31	0	19.7
	3600	348.9	52.08	49.66	0	10.57	387.5	99.06	54.69	0	10.13
	6000	518.7	102.85	42.59	0	9.96	539.3	192.02	44.71	0	8.78
	8400	684.8	163.59	38.83	0	9.64	684.8	299.04	38.74	0	7.99
	12000	917.7	252.25	35.35	0	9.22	879.1	485.53	32.39	0	7.11

Table 5.11 shows the average of 10 instances with Gallager A algorithm when it is applied in the windows of sliding window decoder. Gallager A algorithm cannot find a feasible solution for any of the cases, as given in "# FEAS" column. That is the decoded vector does not satisfy the equality $\mathbf{vH}^{\mathrm{T}} = \mathbf{0}$ (mod 2). Besides, decoded vectors are far away from the best known lower bounds (found by CPLEX while obtaining the results in Table 5.2) which can be seen from the "Gap (%)" column.

"BER" column shows the percent difference from the original codeword. When the values compared with the ones in Table 5.10 for $n = 12000$ and $p = 0.05$, our proposed ABFW algorithm provides significantly higher quality solutions compared to Gallager A.

Table 5.12. Performance of Gallager B.

	w	small					large				
p	n	z	CPU	Gap (%)	# FEAS	BER	z	CPU	Gap (%)	# FEAS	BER
0.02	1200	174.2	10.75	86.15	0	13.55	469.7	21.19	94.69	0	38.67
	3600	781.1	50.46	85.63	0	20.96	1645.2	100.27	95.53	0	45.51
	6000	1854.7	93.83	92.02	0	30.49	2846.3	193.40	95.74	0	47.34
	8400	3037.8	149.39	94.11	0	35.85	4056.3	301.41	95.81	0	48.19
	12000	4816.4	250.38	94.95	0	39.95	5865.5	489.15	95.93	0	48.83
0.05	1200	519.4	11.63	89.01	0	42.87	591.4	21.11	90.48	0	49.29
	3600	1704.5	47.87	89.69	0	47.22	1791.3	100.43	90.19	0	49.69
	6000	2889.0	94.72	89.69	0	48.09	2983.8	194.52	90.02	0	49.88
	8400	4073.6	150.02	89.71	0	48.49	4184.2	302.27	89.99	0	50.03
	12000	5847.9	251.29	89.85	0	48.71	5973.1	489.25	90.07	0	49.96

As summarized Table 5.12, BER values are high since on the contrary to Gallager A algorithm, Gallager B does not guarantee to decrease the error as its iterations proceed. That is error accumulation effect appears in BER results more dramatically for Gallager B. Both Gallager A and B algorithms are faster than ABFW method. However, their solutions are usually not feasible and are distant from the best known lower bound.

These results indicate that ABFW is a strong candidate for decoding problem in communication systems. Gallager A and B algorithms give quick but poor quality solutions. These algorithms may be practical for TV broadcasting and video streams since fast decoding is crucial for these applications. On the other hand, as in the case of NASA's Mission Pluto, we may have some received information that cannot be reobtained from the source. For such cases high solution quality is the key issue instead of decoding speed. Hence, ABFW method is more practical for these kind of communication systems.

6. LDPC CODE DESIGN WITHOUT SMALL CYCLES

6.1. Introduction

In this chapter, we explain the details of our branch–and–cut algorithm to design LDPC codes without small cycles. In practical applications, iterative decoding algorithms, such as Gallager A, are applied. The performance of iterative algorithms are adversely affected with the existance of small cycles in Tanner graph.

An example of Gallager A algorithm on a Tanner graph is demostrated in Figure 6.1. For the Tanner graph given in Figure 6.1a, we can see that vector $\mathbf{v} = (0\ 0\ 0\ 1\ 1)$ is a codeword since it satisfies all parity–check equations. Assuming that we received vector $\mathbf{r} = (0\ 0\ 0\ 0\ 1)$, Gallager A algorithm sends these bits to the check nodes in order to evaluate the parity–check equations as in Figure 6.1a. We can see that second and third parity–check equations are unsatisfied. Then, each check node sends the information of whether it is satisfied (S) or unsatisfied (U) to its neigboring variable nodes.

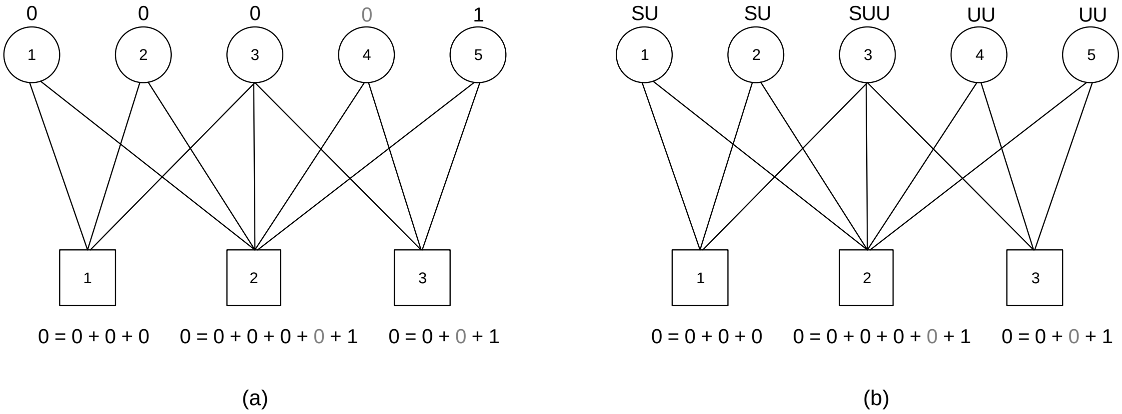


Figure 6.1. Message–passing among variable and check nodes.

For Gallager A algorithm, v_3, v_4 and v_5 are candidate bits to be flipped since $u_i = 2$ for these bits and $u_i > d_i/2$. Since algorithm picks only one bit at each iteration, let us flip v_3 to 1 as shown in Figure 6.2a.

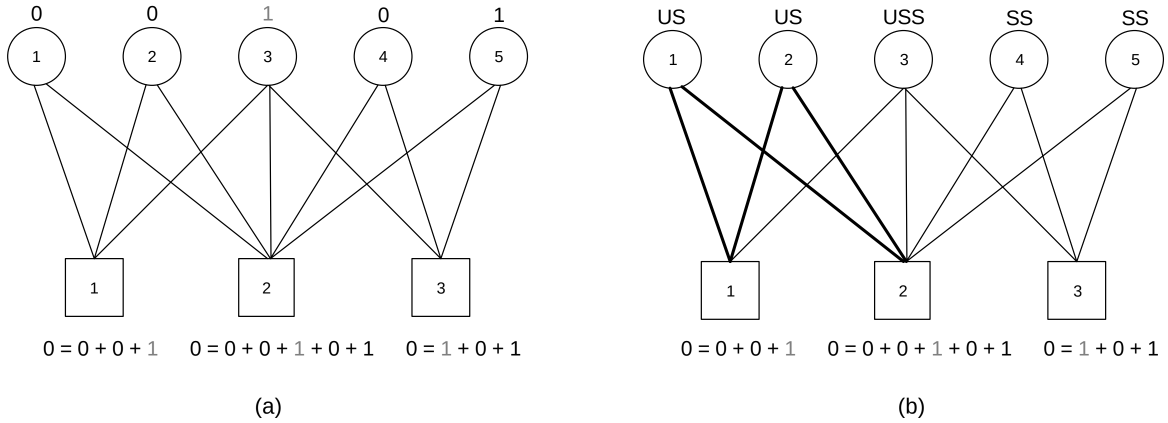


Figure 6.2. An iteration of Gallager A algorithm.

The resulting vector (0 0 1 0 1) satisfies c_2 and c_3 while violating c_1. However, the algorithm terminates with this vector since none of the bits satisfy $u_i > d_i/2$ condition after check nodes pass their information to variable nodes in Figure 6.2b. Hence, vector $\mathbf{r} = (0\ 0\ 1\ 0\ 1)$ is an infeasible stationary point for Gallager A algorithm.

Existance of small cycles (such as $(v_1,\ c_1,\ v_2,\ c_2)$ in Figure 6.2b) is known to cause decoding failures [59]. The size of the smallest cycle is known as the *girth* of the graph [60]. In this part of the thesis, we will focus on designing LDPC code whose Tanner graph does not contain small cycles. In particular, we aim to construct a Tanner graph with a given target girth value.

6.2. Mathematical Formulations

In our Girth Feasibility Model (GFM), our aim is to generate **H** matrix of dimension (m, n), where $m = n - k$, with a girth no smaller than a given value T. In GFM model given below, X_{ji} variable represents the (j, i) entry of the **H** matrix, dv_i is the degree of variable node i, dc_j is the degree of check node j. Constraints (6.2) and (6.3) allow to generate an irregular code with given degree values. As a special case, picking $dv_i = J$ for all i and $dc_j = K$ for all j, one can obtain a $(J, K)-$regular **H** matrix.

We introduce cycle breaking constraints (6.4) for the cycles with size less than target girth T. In GFM, the objective is a constant since target girth T is a given value. Hence, any feasible solution of the model will be optimal.

Girth Feasibility Model (GFM):

$$\max\ T \tag{6.1}$$

$$\text{s.t.:}\ \sum_{j=1}^{m} X_{ji} = dv_i,\ i = 1, ..., n \tag{6.2}$$

$$\sum_{i=1}^{n} X_{ji} = dc_j,\ j = 1, ..., m \tag{6.3}$$

$$\sum_{(j,i)\in C} X_{ji} \leq |C| - 1,\ \forall C \textit{ cycle with } |C| < T \tag{6.4}$$

$$X_{ji} \in \{0, 1\},\ \ j = 1, ..., m,\ i = 1, ..., n. \tag{6.5}$$

An alternative modeling approach is to assume dv_i and dc_j as the target degrees of v_i and c_j, respectively. In Minimum Degree Deviation Model (MDD), the objective is to minimize the degree deviations dv_i^s of v_i and dc_j^s of c_j from target values.

Minimum Degree Deviation Model (MDD):

$$\min\ \sum_{i=1}^{n} dv_i^s + \sum_{j=1}^{m} dc_j^s \tag{6.6}$$

$$\text{s.t.:}\ \sum_{j=1}^{m} X_{ji} + dv_i^s = dv_i,\ i = 1, ..., n \tag{6.7}$$

$$\sum_{i=1}^{n} X_{ji} + dc_j^s = dc_j,\ j = 1, ..., m \tag{6.8}$$

$$(6.4) - (6.5) \tag{6.9}$$

$$dv_i^s, dc_j^s \geq 0,\ \ j = 1, ..., m,\ i = 1, ..., n. \tag{6.10}$$

One can observe that MDD is always feasible since $X_{ji} = 0$ for all (j, i), $dv_i^s = dv_i$ for all i and $dc_j^s = dc_j$ for all j is a trivial solution. Moreover, if optimum objective function value of MDD is zero, which means constraints (6.2) and (6.3) are satisfied without deviation, we get a feasible (optimum) solution of GFM. In the following proposition, we introduce some necessary conditions on (m, n) dimensions of $\mathbf{H}$ matrix.

Proposition 6.1. *For a (J, K)–regular $\mathbf{H}$ matrix having girth T and dimension (m, n)*

(1) $n = 2m$ *if* $K = 2J$,
(2) $n \geq 1 + \sum_{l=2}^{(T-2)/2} J(K-1)[(J-1)(K-1)]^{l-2}$.

Proof. For a $(J, 2J)$–regular $\mathbf{H}$ matrix, each variable node has J neighbors and each check node has $2J$ neighbors in Tanner graph. Hence, total variable degree should be equal to total check degree in a bipartite graph, $nJ = m(2J) \implies n = 2m$.

Moreover, there cannot be cycles in Tanner graph of size less than or equal to $T - 2$ since its girth is T. A cycle of size $T - 2$ includes $(T - 2)/2$ variable nodes. As given in Figure 6.3, in order not to have a cycle of size less than or equal to $T - 2$, each spanning tree with depth $(T - 2)/2$ emanating from a variable node should include distinct variable nodes.

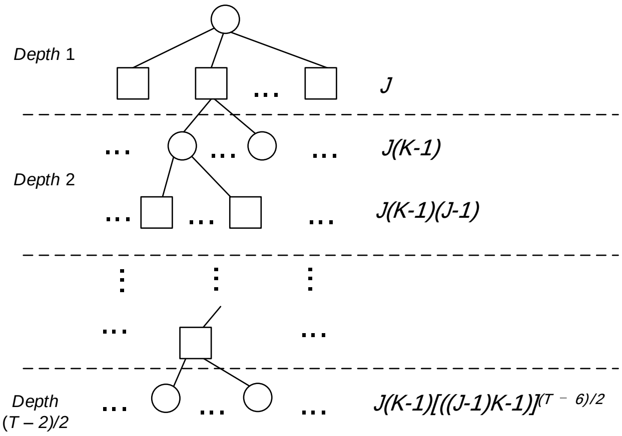


Figure 6.3. Number of spanned nodes in a depth–$(T - 2)/2$ tree.

Since **H** matrix is (J, K)–regular, we can reach J check nodes from a variable node and K variable nodes from a check node. At depth $l \geq 2$, we can visit $J(K-1)[(J-1)(K-1)]^{l-2}$ variable nodes. That is, we need at least $1 + \sum_{l=2}^{(T-2)/2} J(K-1)[(J-1)(K-1)]^{l-2}$ variable nodes to generate a (J, K)–regular **H** matrix with girth T. $\square$

From Proposition 6.1, we deduce that for a given (m, n) dimension, existance of a (J, K)–regular **H** matrix having girth T is not guaranteed. As an example, there is no (3, 6)–regular **H** matrix with $n < 1 + 15 \sum_{l=2}^{3} 10^{l-2} = 166$ and girth $T = 8$. That is GFM can be infeasible depending on the value of target girth T. Hence, in our study we work with MDD model. Since there can be exponential number of cycles in a Tanner graph, we can have exponential number of constraints (6.4) in MDD model. In order to obtain a solution in acceptable amount of time, we add the constraints (6.4) in a cutting–plane fashion to MDD. This gives rise to our branch–and–cut algorithm explained in the next section.

6.3. Branch–and–Cut Algorithm

The main steps of our Branch–and–Cut (BC) approach are listed in Figure 6.4. In BC approach, we are given a target girth value T and the dimension of **H** matrix as (m, n). We initialize our algorithm by relaxing constraints (6.4) from MDD, to obtain relaxed model MDD^r.

We can find either an integral or a fractional solution after solving the relaxed MDD. In the case we find an integral solution, we test its feasibility with respect to the relaxed constraints (6.4) with Figure 6.5. The integral solution is seperated from the solution space by adding required constraints from (6.4) if solution is not feasible. Similarly, we try to seperate a fractional solution from the solution space with Figure 6.7, in order to strengthen the linear relaxation of MDD.

```
Input: Target girth value $T$, $(m, n)$
0. Obtain MDD$^r$ by removing constraints (6.4) from MDD,
   add MDD$^r$ to list $\mathcal{L}$, set $x^* = null$ and $z^* = \infty$.
1. While list $\mathcal{L}$ is not empty
2.     Select and remove a problem from $\mathcal{L}$.
3.     Solve LP relaxation of the problem.
4.     If solution infeasible, Then prune the branch and go to Step 1.
5.     Else let the current solution be $x$ with objective value $z$.
6.     End If
7.     If $z \geq z^*$, Then prune the branch and go to Step 1.
8.     If $x$ is an integer solution,
           If Figure 6.5 finds cycles smaller than $T$,
           Then add cuts (6.4) and go to Step 3.
           Else set $z^* \leftarrow z$, $x^* \leftarrow x$.
           End If
9.     Else If Figure 6.7 generates any cuts,
               Then add cuts (6.4) and go to Step 3.
10.    Else branch to partition the problem into subproblems.
               Add these problems to $\mathcal{L}$ and go to Step 1.
11.    End If
12. End While
Output: H matrix with girth at least $T$.
```

Figure 6.4. Branch–and–Cut algorithm.

In integral solution seperation problem, we find all cycles in Tanner graph whose length is less than T with a depth–first–search algorithm using Figure 6.5. In Figure 6.6, we explain Figure 6.5 with $T = 6$ on Tanner graph given in Figure 6.1. In Figure 6.6a, the search algorithm starts with v_1 at level 0, i.e. $l = 0$, and it is labeled. We label c_1 at $l = 1$, v_2 at $l = 2$ and c_2 at $l = 3$ since they are the first untracked neighbors of their predecessors. At $l = 4$, we visit v_1 but it is labeled. This means we have a cycle of length–4 consisting of nodes stored in *nodeTrack* array and we add this cycle to $\mathcal{C}$ set which keeps all cycles whose length is less than T in current integral **H** matrix.

In Figure 6.6b, we consider other untracked neighbors of c_2 at level 4. After observing that none of v_3, v_4 and v_5 form a cycle, we unlabel them and return to level 3. At $l = 3$, we see that there are no other untracked neighbors of c_2 and backtrack to level 2. In Figure 6.6c, we see v_3 is untracked and we label it at $l = 2$. We label c_2 at $l = 3$ and v_1 at $l = 4$. Hence, we found another cycle of length–4 and add this to set $\mathcal{C}$.

```
Input: A solution of MDD$^r$ with integral $X_{ji}$ values, $T$ target girth
Let set of cycles $\mathcal{C} = \emptyset$ and $nodeTrack$ be an array
For Each variable node $i$, let $l = 1$
    While $l > 0$, Do set $nodeTrack[0] = i$ and label node $i$
        For Each level $l$ from 1 to $T - 2$
            Set $nodeTrack[l]$ to first untracked neighbor of $nodeTrack[l-1]$
            If $nodeTrack[l]$ is labeled, Then a cycle of size $l$ is added to $\mathcal{C}$
                unlabel $nodeTrack[l]$ and
                go to next untracked neighbor of $nodeTrack[l-1]$
                If no such neighbor, Then $l \leftarrow l - 1$
            Else label $nodeTrack[l]$ and $l \leftarrow l + 1$, End If
        For Each
    End While
End For Each
Output: Set of cycles $\mathcal{C}$.
```

Figure 6.5. Integral solution seperation algorithm.

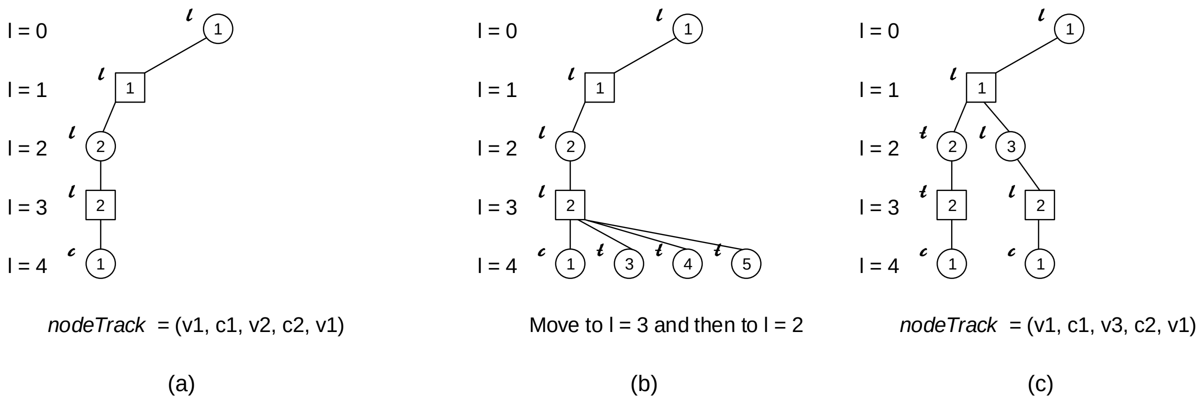


Figure 6.6. Depth–first–search in integral solution seperation.

The solution time to find an optimal solution of MDD can be improved by reducing the solution space using cuts for fractional solutions. In such a case, we have fractional X_{ji} values in Tanner graph. We consider to find maximum average cost cycle in Tanner graph with X_{ji} as cost values. If this cycle violates constraints (6.4) and its length is less than T, then we can add the corresponding violated constraint.

Minimum mean cost cycle is a well known network problem in literature and there is a polynomial time solution algorithm for directed graphs [61]. The problem simply aims to find a smallest mean cost $\sum_{(j,i)\in C} X_{ji}/|C|$ cycle among all directed cycles in graph. However, we cannot implement this algorithm directly, since Tanner graph is undirected. For the solution we can repeatedly implement a negative cycle detection algorithm embedded within a binary search algorithm. Bellman–Ford algorithm can detect negative cycles while searching 1–to–many shortest paths for directed graphs. Bellman–Ford algorithm is also applicable for undirected graphs, if for an edge (j, i) the algorithm updates distance label of node i when it is not the predecessor of node j [61]. If the algorithm detects a negative cycle, we can track the predeccessor list to form the cycle.

In fractional solution seperation problem, we use undirected Bellman–Ford algorithm to detect negative cycles in a binary search method. We first set edge costs as $-X_{ji}$ to turn our maximization problem to minimization. Let μ represent an estimation on the minimum mean cycle cost, and μ^* denote the (unknown) optimal value of μ. Then, given a μ value, we update the edge costs to $(-X_{ji} - \mu)$ and check for existance of a negative cycle. If we start with a μ which is an upper bound for μ^*, we can face with one of these cases in binary search for minimum mean cost μ^*.

Case 1: G has a negative cycle C. In this case, $\sum_{(j,i)\in C}(-X_{ji} - \mu) < 0$. This means,

$$\mu > -\frac{\sum_{(j,i)\in C} X_{ji}}{|C|} > \mu^*. \tag{6.11}$$

Hence, μ is a strict upper bound on μ^*. We can update μ as $\mu = -\frac{\sum_{(j,i)\in C} X_{ji}}{|C|}$ in the next iteration.

Case 2: G has a zero–cost cycle C^*. In this case, $\sum_{(j,i)\in C^*}(-X_{ji} - \mu) = 0$. This means,

$$\mu = -\frac{\sum_{(j,i)\in C^*} X_{ji}}{|C^*|} = \mu^*. \tag{6.12}$$

Hence, $\mu = \mu^*$ and C^* is a minimum mean cost cycle.

```
Input: A solution of $MDD^r$ with fractional $X_{ji}$ values, $T$ target girth
Let $\mu = 0$, set cost of edge $(j,i)$ as $(-X_{ji} - \mu)$
While we can detect negative cycle $C$ with undirected Bellman–Ford
    If $|C| < T$ and $C$ is violating (6.4), Then add corresponding cut (6.4)
    Update $\mu \leftarrow -\frac{\sum_{(j,i)\in C} X_{ji}}{|C|}$
End While
Output: Cuts added to $MDD^r$ model.
```

Figure 6.7. Fractional solution seperation algorithm.

Fractional solution seperation algorithm is summarized in Figure 6.7. We set initial $\mu = 0$, since it is an upper bound on μ^*. If we can find a negative cycle with size $|C| < T$, we can add a cut to MDD if it is violated. This means, C is a cycle with $\sum_{(j,i)\in C} X_{ji} > |C| - 1$. We continue updating μ values until we find a minimum mean cycle. Negative cycles found are added as cuts if they violate constraints (6.4).

6.4. Improvements on Branch–and–Cut Algorithm

In this section we propose some improvements on the BC algorithm given in the previous section. We first observe that the solution space of MDD includes symmetric solutions. Hence, we consider a variable fixing approach to decrease the adverse effect of symmetry. Secondly, we introduce some valid inequalities to improve the linear relaxation of MDD. Finally, we utilize an algorithm from telecommunications literature, i.e. progressive edge growth (PEG), to provide an initial solution to BC algorithm.

6.4.1. Symmetry in MDD Solution Space

In combinatorial optimization problems such as scheduling, symmetry among the solutions is an important issue which directly affects the performance of applied solution methods [62, 63]. We observe that feasible region of MDD contains symmetric solutions. That is, for a Tanner graph there can be isomorphic representations by permuting the variable and check nodes. As an example, the variable nodes are in the order of $\{v_1, v_2, v_3, v_4\}$ in Figure 6.8a and the names of v_2 and v_4 are switched in Figure 6.8b.

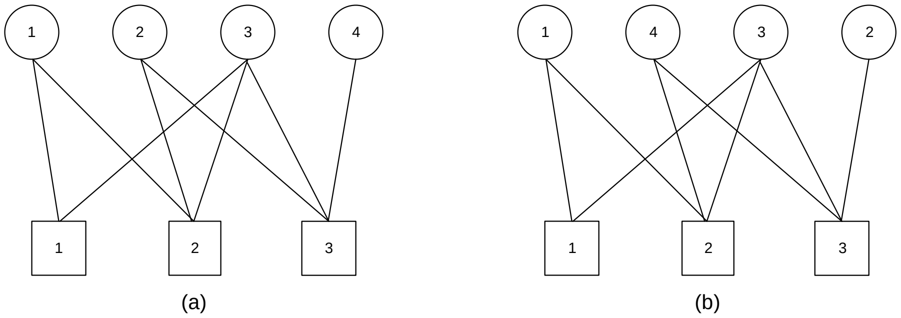


Figure 6.8. Symmetry in MDD solution space.

In Figure 6.9, $\mathbf{H_1}$ and $\mathbf{H_2}$ are the parity–check matrices for Tanner graphs in Figures 6.8a and 6.8b, respectively. We see that although Tanner graphs are isomorphic, their $\mathbf{H}$ matrix representations are not identical. In MDD solution space $\mathbf{H_1}$ and $\mathbf{H_2}$ are considered as two different solutions, which increases the complexity of the solution algorithm.

$$\mathbf{H_1} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \qquad \mathbf{H_2} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

Figure 6.9. Parity–check matrices for Tanner graphs in Figure 6.8.

We can calculate the number of symmetric solutions for a Tanner graph as $(n!)(m!)$, since we can permute n variable nodes as $(n!)$ and m check nodes as $(m!)$ different ways.

6.4.2. Symmetry Breaking with Variable Fixing

In literature, ordering the decision variables, adding symmetry–breaking cuts to the formulation and reformulating the problem are some of the techniques to eliminate symmetric solutions from the feasible region [62, 63]. In our case, we propose a fixing scheme for nonzero X_{ji} entries of **H** matrix that breaks symmetry and does not form any cycles in Tanner graph.

In our variable fixing method (given as Figure 6.10) we consider (J, K)–regular **H** matrices and two modes, i.e. *basic* and *extended.* In *basic* mode, we fix first K entries in the first row to 1 and first J entries in the first column to 1. The remaining entries in first row and column are set to 0 since constraints (6.2) for $i = 1$ and constraints (6.3) for $j = 1$ are satisfied. We illustrate *basic* and *extended* modes in Figure 6.11 for (3, 6)–regular codes of dimension $(30, 60)$ below. Bold entries in Figure 6.11 are fixed with *basic* mode.

```
Input: $(m, n)$ dimension, $(J, K)$ values, mode type
0. Let $r_{cr} = \lfloor (n-1)/(K-1) \rfloor$ and $c_{cr} = \lfloor (m-1)/(J-1) \rfloor$
   Set $X_{1i} = 0$, $i = 1, ..., n$, $X_{j1} = 0$, $j = 1, ..., m$
   If $mode = extended$
      For $j = 2, ..., r_{cr}, i = 1, ..., n$, set $X_{ji} = 0$
      For $j = r_{cr} + 1, ..., m, i = 2, ..., c_{cr}$, set $X_{ji} = 0$
   End If
1. Set $X_{1i} = 1, i = 1, ..., K$ and $X_{j1} = 1, j = 1, ..., J$
2. If $mode = extended$
3.     For $j = 2, ..., r_{cr} + 1, i = 1, ..., K - 1$,
4.         If $1 + (j-1)(K-1) + i \leq n$, Then set $X_{j,1+(j-1)(K-1)+i} = 1$.
5.     End For
6.     For $j = 1, ..., J - 1, i = 2, ..., c_{cr} + 1$,
7.         If $1 + i(J-1) + j \leq m$, Then set $X_{1+i(J-1)+j,i} = 1$.
8.     End For
9. End If
Output: Some $X_{ji}$ values are fixed.
```

Figure 6.10. Variable fixing algorithm.

In *extended* mode, we extend variable fixing further as (m, n) dimension of **H** matrix allows. In Figure 6.11, the labels on the rows and colums show the sum of the values in that row and column, respectively. We observe that for $r_{cr} = \lfloor (n-1)/(K-1) \rfloor$ many row sums are equal to 6 and $c_{cr} = \lfloor (m-1)/(J-1) \rfloor$ many column sums are equal to 3. Hence, for c_{cr}–columns constraints (6.2) and for r_{cr}–rows constraints (6.3) are satisfied. We remain with a rectangle of size $(m - r_{cr}) \times (n - c_{cr})$, which includes unfixed X_{ji} variables shown as dots.

$$
\begin{array}{ccl|l}
 & & & c_{cr} \\
 & & 33333333333333 & 2111111111111111111111111111111111111111111111 \\
 & 6 & \mathbf{11111100000000} & \mathbf{0000000000000000000000000000000000000000000000} \\
 & 6 & \mathbf{1}0000011111000 & 0000000000000000000000000000000000000000000000 \\
 & 6 & \mathbf{1}0000000000111 & 1100000000000000000000000000000000000000000000 \\
 & 6 & \mathbf{0}1000000000000 & 0011111000000000000000000000000000000000000000 \\
 & 6 & \mathbf{0}1000000000000 & 0000000111110000000000000000000000000000000000 \\
 & 6 & \mathbf{0}0100000000000 & 0000000000001111100000000000000000000000000000 \\
 & 6 & \mathbf{0}0100000000000 & 0000000000000000011111000000000000000000000000 \\
 & 6 & \mathbf{0}0010000000000 & 0000000000000000000000111110000000000000000000 \\
 & 6 & \mathbf{0}0010000000000 & 0000000000000000000000000001111100000000000000 \\
 & 6 & \mathbf{0}0001000000000 & 0000000000000000000000000000000011111000000000 \\
r_{cr} & 6 & \mathbf{0}0001000000000 & \underline{0000000000000000000000000000000000000111110000} \\
 & 5 & \mathbf{0}0000100000000 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots1111 \\
 & 1 & \mathbf{0}0000100000000 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
 & 1 & \mathbf{0}0000010000000 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
 & 1 & \mathbf{0}0000010000000 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
 & 1 & \mathbf{0}0000001000000 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
 & 1 & \mathbf{0}0000001000000 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
 & 1 & \mathbf{0}0000000100000 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
 & 1 & \mathbf{0}0000000100000 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
 & 1 & \mathbf{0}0000000010000 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
 & 1 & \mathbf{0}0000000010000 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
 & 1 & \mathbf{0}0000000001000 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
 & 1 & \mathbf{0}0000000001000 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
 & 1 & \mathbf{0}0000000000100 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
 & 1 & \mathbf{0}0000000000100 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
 & 1 & \mathbf{0}0000000000010 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
 & 1 & \mathbf{0}0000000000010 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
 & 1 & \mathbf{0}0000000000001 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
 & 1 & \mathbf{0}0000000000001 & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
 & 1 & \mathbf{0}0000000000000 & 1\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots
\end{array}
$$

Figure 6.11. Variable fixing on a (3, 6)–regular **H** matrix of dimension (30, 60).

Some characteristics of cycles in a Tanner graph can be visualized by considering the Tanner graph given in Figure 6.8a and corresponding parity–check matrix $\mathbf{H_1}$ in Figure 6.9. It can be seen that $C_1 = (v_1, c_1, v_3, c_2)$ and $C_2 = (c_1, v_1, c_2, v_2, c_3, v_3)$ are two cycles in Tanner graph in Figure 6.8a. Figures 6.12a and 6.12b visualize cycles C_1 and C_2 on $\mathbf{H_1}$, respectively.

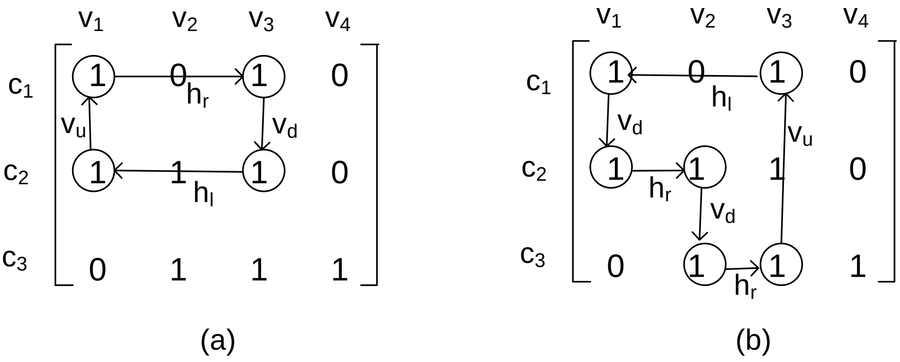


Figure 6.12. Cycles C_1 and C_2 on $\mathbf{H_1}$.

We can observe that a cycle is an alternating sequence of horizontal and vertical movements between cells having value 1. In particular, cycle C_1 is a sequence of horizontal right (h_r), vertical down (v_d), horizontal left (h_l) and vertical up (v_u) movements. Similarly, cycle C_2 can be expressed with sequence $(v_d, h_r, v_d, h_r, v_u, h_l)$. Moreover, we deduce that there needs to be at least one from each of h_u, h_d, v_u and v_d movements in a cycle.

Proposition 6.2. *Variable fixing on* **H** *matrix with extended mode does not form any cycles in Tanner graph.*

Proof. Assume we apply variable fixing with *extended* mode and consider cells whose X_{ji} values have been fixed to 1. There are four cases to have an alternating sequence among variable and check nodes as given in Figures 6.13 and 6.14.

In Figure 6.13a, the sequence of case 1 is $(v_d, h_r, v_d, h_r, ...)$ and in Figure 6.13b for case 2, we have sequence $(h_r, v_d, h_r, v_d, ...)$, which do not include v_u and h_l movements. Hence, there cannot be any cycles in these cases.

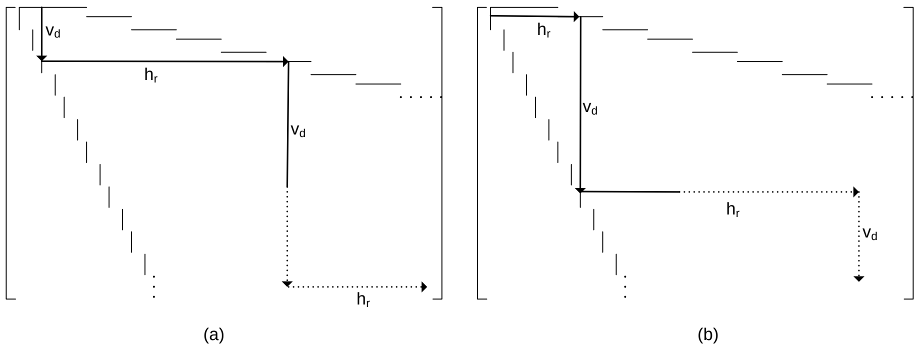


Figure 6.13. Alternating variable and check nodes, cases 1 and 2.

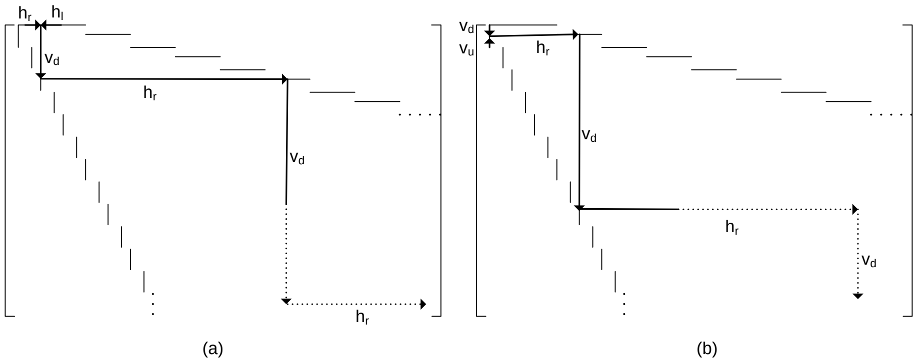


Figure 6.14. Alternating variable and check nodes, cases 3 and 4.

In Figure 6.14a (case 3), we have two options to start, i.e. h_r or h_l movements. Then the sequence will be (h_r or $h_l, v_d, h_r, v_d, h_r, ...$), which does not include v_u movement. In Figure 6.14b (case 4), v_d or v_u are candidates to begin the sequence. In this case, the sequence will be (v_d or $v_u, h_r, v_d, h_r, v_d, ...$) which does not include h_l movement. Hence, there are no cycles in these cases either. □

We can use the partial solution obtained with Figure 6.10 to generate a feasible solution of MDD. Since partial solution does not include any cycles (see Proposition 6.2), setting the nonfixed entries to zero gives a feasible solution (an upper bound).

6.4.3. Valid Inequalities for Cycle Regions

After applying extended fixing, MDD problem reduces to locating ones in reduced rectangle of size $(m-r_{cr})\times(n-c_{cr})$. That is problem size reduced by $\left(1-\frac{(m-r_{cr})\times(n-c_{cr})}{m\times n}\right)$ $\times 100\%$. We can further improve the performance of BC algorithm by introducing valid inequalities that help to break the symmetry in reduced problem assuming code is (J,K)–regular. We first observe that when we are given a dimension (m,n), the reduced rectangle always appears in between the two extending 1–blocks as given in Figure 6.15.

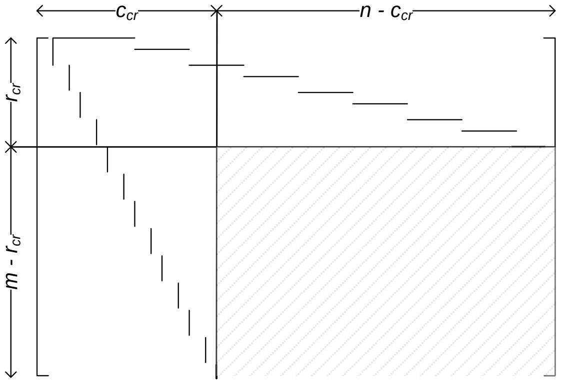


Figure 6.15. Reduced rectangle when (m,n) is given.

MDD model locates ones in reduced rectangle to minimize the degree deviation without creating cycles of size smaller than girth T. Hence, we can generate valid inequalities that eliminate cycles with size smaller than T in reduced rectangle. For this purpose as given in Figure 6.16, we first divide the region between the extending 1–blocks into smaller rectangles, i.e. subblocks, having $(J-1)(K-1)$ rows and $(K-1)$ columns since we assume a (J,K)–regular code.

We investigate the size of a cycle that will be formed when we locate only a single 1 in a subblock and categorize the subblock according to this size. For example in Figure 6.16, we observe that cycle size is common for all entries in the subblock and we classify the subblocks as Cycle–4, Cycle–6, Cycle–8 and Cycle–10 regions. Moreover, we observe that these cycle regions have repeating pattern due to (J,K)–regularity.

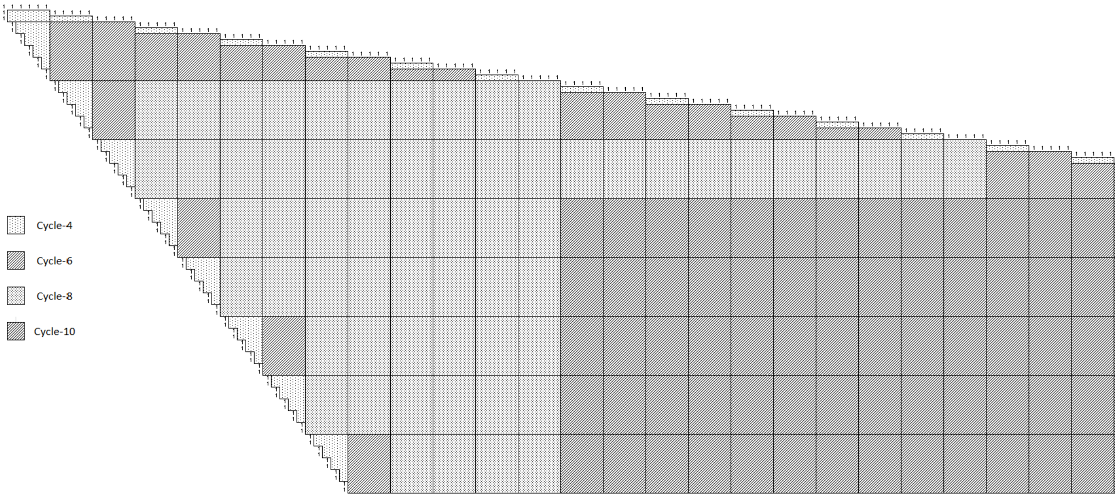


Figure 6.16. Subblocks and cycle regions with $J = 3$ and $K = 6$.

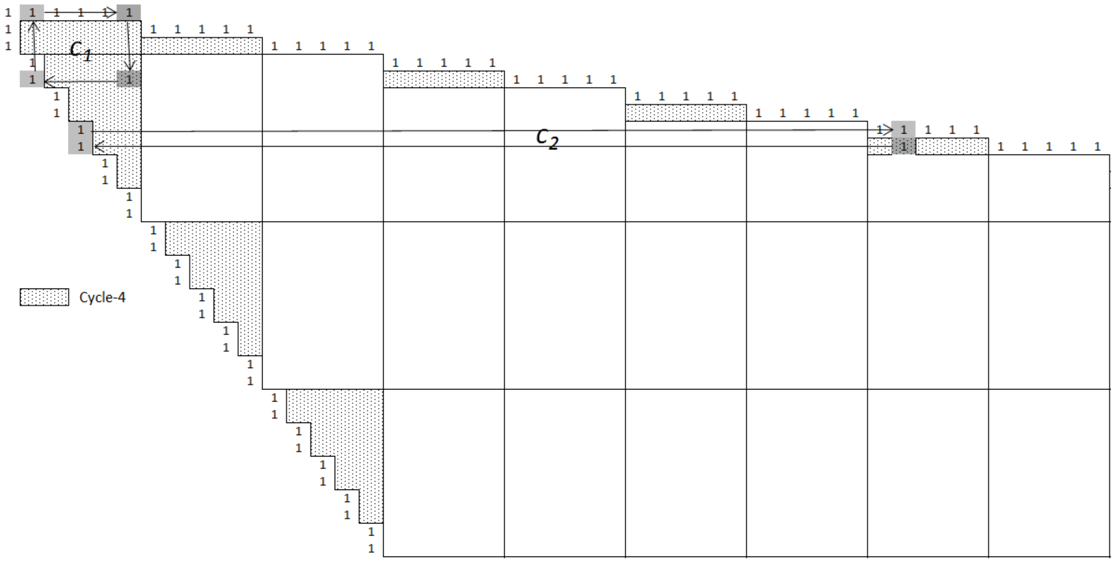


Figure 6.17. Cycle–4 regions with $J = 3$ and $K = 6$.

In particular, when there is a 1 in a Cycle–4 region (dotted area), we have a cycle of length 4 as in the case of cycles C_1 and C_2 in Figure 6.17. Besides, Cycle–4 regions repeats both horizontally and vertically.

Similar horizontal and vertical repeating patterns can be seen for Cycle–6 and Cycle–8 regions in Figure 6.18. Making use of the patterns, one can express the cycle region of an entry (j, i) as a function. We introduce valid inequalities for MDD based on the cycle region information of the entries in the reduced rectangle.

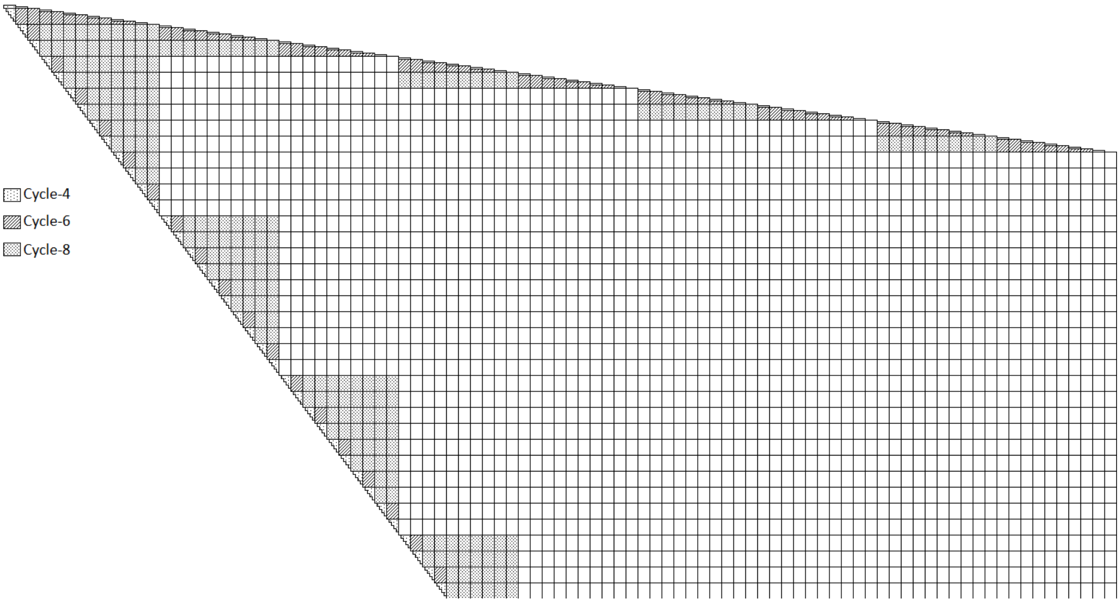


Figure 6.18. Cycle–4, Cycle–6 and Cycle–8 regions with $J = 3$ and $K = 6$.

Proposition 6.3. *Let (j, i) be an entry in the reduced rectangle, i.e. $j \in \{m-r_{cr}, ..., m\}$ and $i \in \{n - c_{cr}, ..., n\}$ and let cycleRegion(j, i) represent the cycle region of the entry. Let S denote the number of subblocks that intersects with the reduced rectangle and let B_s, $s \in \{1, ..., S\}$ represent the set of (j, i) entries in subblock s.*

(1) If cycleRegion$(j, i) < T$, then constraint

$$X_{ji} = 0 \tag{6.13}$$

is valid.

(2) If $T = 8$ and $(j, i) \in B_s$ with cycleRegion(j, i) = 8 or 10, then constraints

$$\sum_{j=1}^{J-1} \sum_{((k-1)(J-1)+j,i) \in B_s} X_{(k-1)(J-1)+j,i} \leq 1, \quad k \in \{1, ..., K-1\} \tag{6.14}$$

are valid.

(3) If $T = 10$ and $(j, i) \in B_s$ with cycleRegion$(j, i) = 10$, then constraint

$$\sum_{(j,i)\in B_s} X_{ji} \leq 1 \tag{6.15}$$

is valid.

Proof. Let us consider each item seperately.

(1) There cannot be cycles of size smaller than girth T. If $X_{ji} = 1$, then we have a cycle of size $cycleRegion(j,i) < T$, which is not desired. Hence, $X_{ji} = 0$ in this case.

(2) If $T = 8$, then there should not be any cycles of size 6. Let us consider a subblock with cycle region 8 or 10, which is subdivided into $K-1$ equal pieces each includes $J-1$ rows. In Figure 6.19, we give an example for Cycle–8 subblock with $J = 3$ and $K = 6$ where we have $(K-1) = 5$ subpieces each having $(J-1) = 2$ rows. As seen in figure, a cycle of size 6 forms when there is more than one nonzero entry in a subpiece.

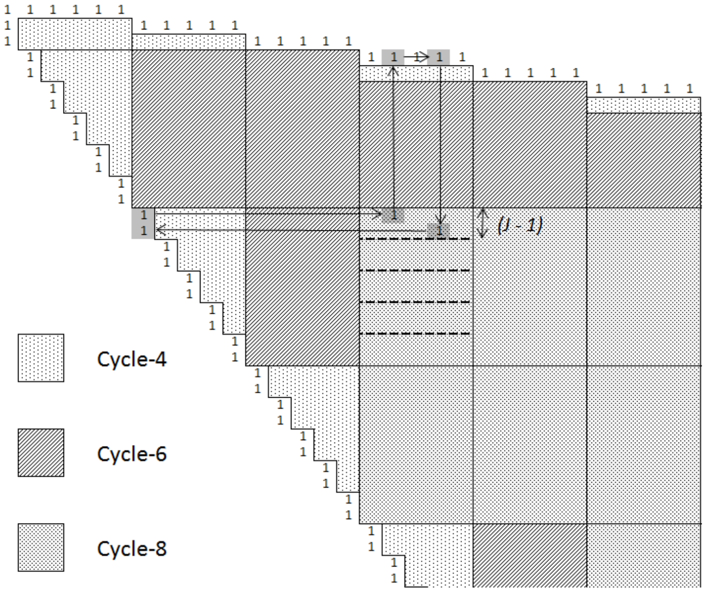


Figure 6.19. A cycle of size 6 on Cycle–8 region with $J = 3$ and $K = 6$.

A similar case appears for Cycle–10 subblocks. Hence, constraints (6.14) are valid, since they force to have at most one nonzero entry in each subpiece when cycle region of the subblock is either 8 or 10.

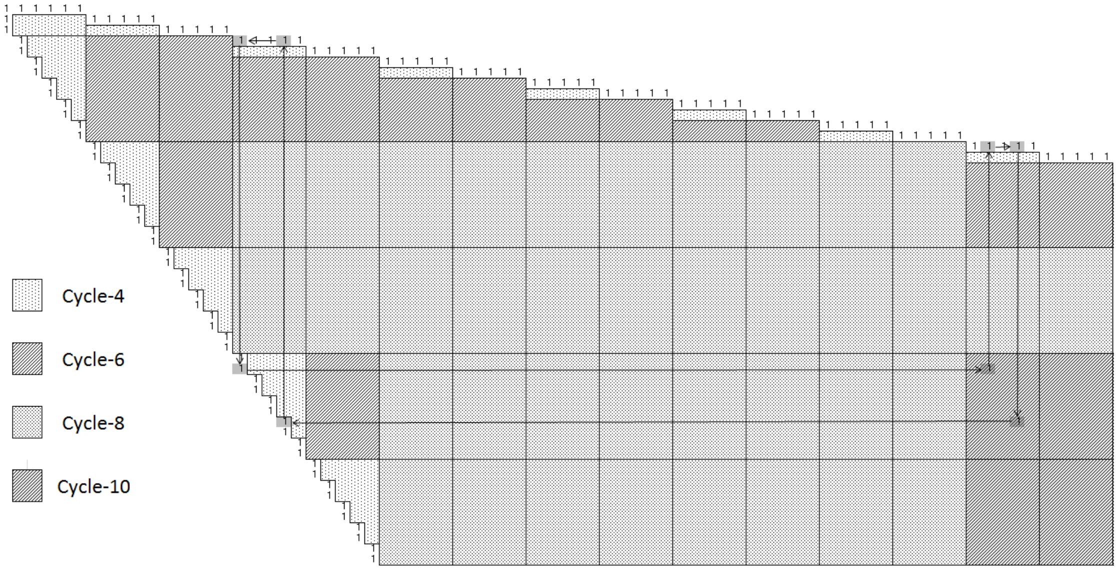


Figure 6.20. A cycle of size 8 on Cycle–10 region with $J = 3$ and $K = 6$.

(3) A cycle of size 8 is not allowed when $T = 10$. However, when there is more than one nonzero entry in a subblock with cycle region 10, there is a cycle of size 8 as given in Figure 6.20. Constraint (6.15) is valid since it bounds the number of nonzero entries from above with 1. □

As discussed in Section 6.4.1, a Tanner graph can be alternatively represented by reordering its variable and check nodes. In Proposition 6.5, we show that any (J, K)–regular $\mathbf{H}$ matrix of dimension (m, n) that has sufficiently large girth T can be expressed as in Figure 6.21 by reordering the rows and columns. Before, we have Proposition 6.4 for $(J, K)-$regular codes using the relationships $J < K$ and $n > K$ which are valid in practical applications.

Proposition 6.4. *For a $(J, K)-$regular code of dimension (m, n), we have $r_{cr} \leq c_{cr}$ where $r_{cr} = \lfloor (n-1)/(K-1) \rfloor$ and $c_{cr} = \lfloor (m-1)/(J-1) \rfloor$ as in Figure 6.10, Step 0.*

Proof. Let $\frac{J}{K} = a \in (0, 1)$, then we have $mK = nJ \implies m = na$. Then we can write, $\frac{m-1}{J-1} = \frac{na-1}{Ka-1} = \frac{a(n-1)+a-1}{a(K-1)+a-1} > \frac{n-1}{K-1}$ since $a < 1$. From here we obtain $\lfloor \frac{n-1}{K-1} \rfloor \leq \lfloor \frac{m-1}{J-1} \rfloor \implies r_{cr} \leq c_{cr}$. □

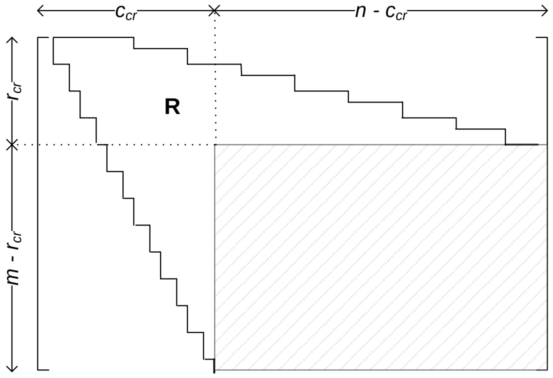


Figure 6.21. Reordered (J, K)–regular **H** matrix with girth $T > t$.

Proposition 6.5. *Let* **H** *be a* (J, K)*–regular parity–check matrix of dimension* (m, n)*. Let* r_{cr} *and* c_{cr} *be defined as in Proposition 6.4. Let* $t = \max_{(j,i)\in R}\{cycleRegion(j, i)\}$ *where* R *is the region between the two extending 1–blocks and outside the reduced rectangle as in Figure 6.21. Then, nonzero entries of* **H** *can be represented as two extending 1–blocks as in Figure 6.21 by reordering its rows and columns if it has a girth* $T > t$*. Remaining nonzero entries are in the reduced rectangle.*

Proof. Let **H** be (J, K)–regular matrix of dimension (m, n) with girth $T > t$. Let us apply the reordering algorithm in Figure 6.22 on **H**.

```
Input: H, $(m, n)$ dimension, $(J, K)$ values, $T$ value
1. Pick row 1, reorder columns such that all ones are in first $K$ columns.
   Pick column 1, reorder rows such that all ones are in first $J$ rows.
2. For $s \in \{2, ..., r_{cr}\}$
3.     Pick row $s$, reorder columns such that $(K - 1)$ ones are in first available columns.
       Pick column $s$, reorder rows such that $(J - 1)$ ones are in first available rows.
4. End For
5. For $s \in \{r_{cr} + 1, ..., c_{cr}\}$
6.     Pick column $s$, reorder rows such that $(J - 1)$ ones are in first available rows.
7. End For
Output: Reordered H matrix.
```

Figure 6.22. Reordering algorithm.

At Step 1 of Figure 6.22, J many ones are located in first column. For second row, i.e. $s = 2$, first available $(K-1)$ columns to locate ones are columns $(K+1, ..., 2K-1)$, since otherwise a cycle with size less than T exists. Similarly for second column, i.e. $s = 2$, first available $(J-1)$ rows are $(J+1, ..., 2J-1)$ without creating a cycle. The algorithm continues in this fashion for r_{cr} rows and columns. Since, in Proposition 6.4 we see that $r_{cr} \leq c_{cr}$, we continue to locate ones for remaining $(c_{cr} - r_{cr})$ many columns. $\square$

Proposition 6.6. *Let z^* be the optimum objective value of MDD and z_f^* be the optimum objective value of MDD when variables are fixed with extended mode. Let t be defined as in Proposition 6.5. Assume there exists a (J, K)–regular code with dimension (m, n), then*

(1) $0 = z^* = z_f^*$ *if* $T > t$,
(2) $0 = z^* \leq z_f^*$ *if* $T \leq t$.

Proof. For any dimension (m, n), we have $z^* \leq z_f^*$ since we fix some X_{ji} variables in *extended* mode. If there exists a (J, K)–regular code, then there is optimal solution with objective value $z^* = 0$. We know from Proposition 6.5 when $T > t$, a (J, K)–regular code can be expressed as in Figure 6.21, which coincides with the case in *extended* mode. Hence, we have $z_f^* = z^* = 0$.

In MDD if *cycleRegion*$(j, i) \geq T$, then X_{ji} can be nonzero without harming the girth T. When $T \leq t$, there are $(j, i) \in R$ in Figure 6.21 with *cycleRegion*$(j, i) \geq T$ and they are fixed to zero since we fix all entries in region R to zero in *extended* mode. Then, we have $0 = z^* \leq z_f^*$ in this case. $\square$

6.4.4. Progressive Edge Growth (PEG) Algorithm

The last improvement for our BC algorithm is to introduce a starting solution for initial upper bound. For this purpose, we adapt an existing algorithm from the literature known as Progressive Edge Growth (PEG) algorithm [64]. We modify this algorithm for our problem by starting PEG from partial initial solution generated by our fixing algorithm given in Figure 6.10. We also update PEG such that the generated solution has girth at least T.

```
Input: $(m, n)$ dimension, $\mathbf{dv}$ and $\mathbf{dc}$ vectors, $T$ value
0. Initialize $\mathbf{X} \leftarrow \mathbf{0}$, $\mathbf{dv^c} \leftarrow \mathbf{0}$, $\mathbf{dv^s} \leftarrow \mathbf{dv}$ and $\mathbf{dc^s} \leftarrow \mathbf{dc}$, $\mathcal{I} \leftarrow \mathbf{0}$
1. Apply Figure 6.10 and update slacks
   $dv_i^s \leftarrow dv_i^s - \sum_j X_{ji}$ for all $i$ and $dc_j^s \leftarrow dc_j^s - \sum_i X_{ji}$ for all $j$
   and current degrees $dv_i^c \leftarrow \sum_j X_{ji}$ for all $i$
2. For $i \in \{1, ..., n\}$ set $\mathcal{I} \leftarrow \mathbf{0}$
3.     For $k \in \{0, ..., dv_i^c\}$
4.         If $k = 0$, Then set $X_{ji} = 1$ for $j = \text{argmax}_j\{dc_j^s\}$
5.         Else apply BFS from $v_i$ to span check nodes, let tree has depth $l$
6.             If $2l \geq T$ or $|\mathcal{N}_i^l| \leq m$, let $\mathcal{I}$ is incidence vector for $\mathcal{N}_i^l$
                   set $X_{ji} = 1$ for $j = \text{argmax}_j\{(1 - \mathcal{I}_{c_j})dc_j^s\}$
7.         End If
8.         Update $dv_i^c$, $dv_i^s$, $dc_j^s$ as in Step 1
9.     End For
10. End For
Output: An initial solution for MDD.
```

Figure 6.23. Modified PEG algorithm.

In Figure 6.23, $\mathbf{dv}$ and $\mathbf{dc}$ are target degree vectors for variable and check nodes, respectively. Let deviation from target degrees for variable and check nodes be given by slack vectors $\mathbf{dv^s}$ and $\mathbf{dc^s}$, and current degrees of variable nodes be listed in vector $\mathbf{dv^c}$. Moreover, $\mathcal{N}_i^l$ represents the set of all check nodes that can be reached from v_i with a tree of depth l. Hence, set $\mathcal{N}_i^l \setminus \mathcal{N}_i^{l-1}$ collects check nodes that are reached at the lth step from v_i for the first time. We can represent the check nodes in set $\mathcal{N}_i^l$ with an incidence vector $\mathcal{I}$ as $\mathcal{I}_{c_j} = 1$ if $c_j \in \mathcal{N}_i^l$ and zero otherwise.

Starting from the solution provided by Figure 6.10, PEG adds an edge (j, i), i.e. $X_{ji} = 1$, if this edge does not form a cycle ($|\mathcal{N}_i^l| \leq m$) or the size of the cycle created is greater or equal to T (Step 6). For edge assignment, the algorithm picks c_j having the maximum slack value dc_j^s to in order to fit the target degree dc_j. The generated solution is feasible for MDD since it has girth at least T.

6.5. Computational Results

The computations have been carried out on a computer with 2.0 GHz Intel Xeon E5–2620 processor and 46 GB of RAM working under Windows Server 2012 R2 operating system. In computational experiments, we use CPLEX 12.6.2 to test the performance of BC method and evaluate how different improvement strategies on BC given in Section 6.4 affect the results. We implement all algorithms in C++ programming language. We summarize the solution methods in Table 6.1.

Table 6.1. Summary of solution methods.

Method	Mode	Valid Inequalities	PEG
BC_0	–	–	–
BC_1	*basic*	–	–
BC_2	*extended*	–	–
BC_3	*extended*	√	–
BC_4	*extended*	√	√

In BC_0, we apply BC method in Figure 6.4 without any improvement technique. Figure 6.4 includes Figures 6.5 and 6.7 to seperate integral and fractional solutions, respectively. In CPLEX, we implement Figure 6.5 using *LazyCutCallback* and Figure 6.7 with *UserCutCallback* routines. We utilize default branching settings of CPLEX. In BC_1 method, we apply Figure 6.10 to fix first row and column of **H** matrix with *basic* mode. In BC_2 method, Figure 6.10 is implemented with *extended* mode to fix r_{cr} rows and c_{cr} columns (see Section 6.4.2). In BC_3 method, we apply fixing with *extended* mode and use valid inequalities explained in Section 6.4.3. Finally in BC_4 method, we provide initial solution with modified PEG (Figure 6.23) under *extended* mode and use valid inequalities in Section 6.4.3. We list the parameters used in computational experiments in Table 6.2.

Table 6.3 shows the computational results for method BC_0 with respect to different parameters. Column "z" is the objective function value of MDD found by CPLEX within 3600 seconds time limit. Best known lower bound found by CPLEX in the time

Table 6.2. List of computational parameters.

Parameters	
(J, K)	(3, 6)– regular codes
(m, n)	(10, 20), (15, 30), (20, 40), (30, 60),
	(40, 80), (100, 200), (150, 300), (250, 500), (500, 1000)
T	6, 8, 10
Time Limit	3600 secs

limit is given in column "z_l". For each of the methods, we have an initial feasible solution (an upper bound) with objective value z_u^i. In BC$_0$ method, $\mathbf{H} = \mathbf{0}$ is a trivial solution providing an initial upper bound. In methods from BC$_1$ to BC$_4$ initial feasible solution is obtained from variable fixing (see Section 6.4.2) or initial heuristic (PEG) (see Section 6.4.4). Computational time in seconds is given with column "CPU (secs)" and percentage difference among z_l and z is under column "Gap (%)". In column "Lazy" we show number of cuts added to MDD using Figure 6.5, whereas column "User" is the number of cuts added to MDD with Figure 6.7.

As explained in Section 6.2, we have a (J, K)–regular code if $z_l = z = 0$. We can conclude that it is not possible to have a (J, K)–regular code with given (m, n) and girth T when we have $z \geq z_l > 0$ (see Proposition 6.1). In Table 6.3, we can see that BC$_0$ can find (3, 6)–regular code almost all instances when $T = 6$. As T and n increase, BC$_0$ method cannot improve initial upper bound z_u^i. For $T = 8$ and $T = 10$, we observe that the number of lazy and user cuts added to MDD gets smaller as n gets larger. This is since adding a cut takes more time as n increases, which causes the algorithm to generate fewer cuts within the given time limit.

Table 6.4 shows our computational results for BC$_1$ and BC$_2$. We have better initial upper bound (z_u^i) values compared to BC$_0$ when we implement variable fixing with *basic* mode in BC$_1$ and we can improve z_u^i values more in BC$_2$ with *extended* mode. We observe that $z_l = 1$ for $T = 6$ and $n = 20$ in BC$_1$, which means it is not possible to have a (3, 6)–regular code for this dimension.

Table 6.3. Computational results for BC_0.

					CPU	Gap	# Cuts	
T	n	z_l	z	z_u^i	(secs)	(%)	Lazy	User
6	20	0	20	120	*time*	100	7399	0
	30	0	0	180	13.80	0	5784	0
	40	0	0	240	0.39	0	331	0
	60	0	0	360	0.45	0	184	0
	80	0	0	480	0.41	0	94	0
	200	0	0	1200	1.06	0	238	0
	300	0	0	1800	2.62	0	165	0
	500	0	0	3000	4.72	0	114	0
	1000	0	0	6000	32.71	0	111	0
8	20	0	62	120	*time*	100	51759	19192
	30	0	86	180	*time*	100	138018	9890
	40	0	240	240	*time*	100	196066	4452
	60	0	360	360	*time*	100	285614	2683
	80	0	480	480	*time*	100	328598	2055
	200	0	1200	1200	*time*	100	404838	736
	300	0	1800	1800	*time*	100	327245	261
	500	0	3000	3000	*time*	100	207064	61
	1000	0	0	6000	905.21	0	2458	2
10	20	0	62	120	*time*	100	171969	31649
	30	0	164	180	*time*	100	393619	7676
	40	0	240	240	*time*	100	410765	5554
	60	0	360	360	*time*	100	554898	3740
	80	0	480	480	*time*	100	496226	2465
	200	0	1200	1200	*time*	100	67718	406
	300	0	1800	1800	*time*	100	22282	88
	500	0	3000	3000	*time*	100	11548	10
	1000	0	6000	6000	*time*	100	87546	65

Table 6.4. Computational results for BC$_1$ and BC$_2$.

		BC$_1$							BC$_2$						
					CPU	Gap	# Cuts					CPU	Gap	# Cuts	
T	n	z_l	z	z_u^i	(secs)	(%)	Lazy	User	z_l	z	z_u^i	(secs)	(%)	Lazy	User
6	20	1	20	104	*time*	95	3804	0	12	20	62	*time*	40	246	0
	30	0	0	164	23.11	0	7016	0	0	0	92	0.10	0	2532	0
	40	0	0	224	0.39	0	420	0	0	0	122	0.12	0	160	0
	60	0	0	344	0.37	0	124	0	0	0	182	0.20	0	148	0
	80	0	0	464	0.56	0	125	0	0	0	242	0.23	0	146	0
	200	0	0	1184	1.43	0	108	0	0	0	602	0.48	0	109	0
	300	0	0	1784	2.31	0	87	0	0	0	902	1.11	0	167	0
	500	0	0	2984	4.73	0	94	0	0	0	1502	2.44	0	225	0
	1000	0	0	5984	49.23	0	110	0	0	0	3002	21.83	0	165	0
8	20	0	44	104	*time*	100	19099	16644	42	42	62	0.08	0	0	0
	30	0	74	164	*time*	100	73701	8222	64	64	92	0.33	0	244	0
	40	0	92	224	*time*	100	131947	4385	56	84	122	*time*	32	2660	68
	60	0	344	344	*time*	100	225388	1903	12	80	182	*time*	85	25418	0
	80	0	464	464	*time*	100	240048	1703	0	242	242	*time*	100	61703	0
	200	0	1184	1184	*time*	100	407426	895	0	602	602	*time*	100	229615	0
	300	0	1784	1784	*time*	100	331382	487	0	902	902	*time*	100	292952	0
	500	0	2984	2984	*time*	100	216118	124	0	0	1502	1633.83	0	148866	0
	1000	0	0	5984	454.20	0	1386	6	0	0	3002	449.31	0	1263	0
10	20	0	58	104	*time*	100	57480	80057	54	54	62	0.09	0	0	0
	30	0	164	164	*time*	100	242023	16891	92	92	92	0.09	0	0	0
	40	0	224	224	*time*	100	342790	8174	122	122	122	0.11	0	0	0
	60	0	344	344	*time*	100	290718	3953	182	182	182	0.14	0	0	0
	80	0	464	464	*time*	100	471767	5285	236	236	242	142.56	0	3850	42
	200	0	1184	1184	*time*	100	51505	675	66	602	602	*time*	89	310451	1
	300	0	1784	1784	*time*	100	20565	135	0	902	902	*time*	100	461039	0
	500	0	2984	2984	*time*	100	9568	60	0	1502	1502	*time*	100	467420	0
	1000	0	5984	5984	*time*	100	90273	91	0	3002	3002	*time*	100	110798	0

In Table 6.4, we observe that we can solve more instances to optimality, i.e. Gap (%) value is zero, with BC$_2$ method. There are instances such as $T = 10$ and $n = 80$ that we have $z_l = z > 0$, for which we can say that the best possible code includes $z/2 = 236/2 = 118$ fewer ones than a (3, 6)–regular code (having $X_{j,i} = 1$ improves MDD objective by 2).

Comparing Table 6.4 and 6.5, we can see that z_u^i values for BC$_2$ and BC$_3$ are the same since we apply *extended* mode for both. On the other hand, we have better z_u^i values in BC$_4$ since we apply Figure 6.23 to generate an initial feasible solution (see Section 6.4.4). Results show that z values get better, the number of cuts added to MDD gets smaller and computational time improves on average as we have tighter initial solutions.

Among the methods from BC$_0$ to BC$_4$, we can see that BC$_4$ uses fewer cuts on the average and solves more instances to optimality (19 instances out of 27 instances). Besides, BC$_4$ provides an evidence that there cannot be a (J, K)–regular code (when

Table 6.5. Computational results for BC$_3$ and BC$_4$.

		BC$_3$							BC$_4$						
					CPU	Gap	# Cuts					CPU	Gap	# Cuts	
T	n	z_l	z	z_u^i	(secs)	(%)	Lazy	User	z_l	z	z_u^i	(secs)	(%)	Lazy	User
6	20	12	20	62	*time*	40	260	0	13.9	20	26	*time*	37	238	0
	30	0	0	92	0.15	0	1784	0	0	0	8	0.22	0	2522	0
	40	0	0	122	0.14	0	160	0	0	0	2	0.36	0	441	0
	60	0	0	182	0.20	0	160	0	0	0	2	0.16	0	154	0
	80	0	0	242	0.24	0	148	0	0	0	2	0.33	0	184	0
	200	0	0	602	0.55	0	109	0	0	0	4	0.56	0	104	0
	300	0	0	902	1.02	0	167	0	0	0	2	1.11	0	167	0
	500	0	0	1502	3.33	0	225	0	0	0	2	3.05	0	207	0
	1000	0	0	3002	39.79	0	170	0	0	0	4	29.84	0	174	0
8	20	42	42	62	0.12	0	0	0	42	42	62	0.13	0	0	0
	30	64	64	92	0.16	0	0	0	64	64	86	0.13	0	0	0
	40	84	84	122	7.89	0	473	0	84	84	86	2.59	0	367	0
	60	28	64	182	*time*	56	55860	0	28	60	66	*time*	53	58432	0
	80	8	242	242	*time*	97	95449	0	8	38	38	*time*	87	83615	0
	200	0	0	602	2181.18	0	154415	0	0	0	16	1893.82	0	166949	0
	300	0	902	902	*time*	100	280596	0	0	10	10	*time*	100	284583	0
	500	0	0	1502	614.80	0	33635	0	0	0	10	1414.95	0	71447	0
	1000	0	0	3002	324.91	0	587	0	0	0	12	384.75	0	866	0
10	20	54	54	62	0.10	0	0	0	54	54	62	0.13	0	0	0
	30	92	92	92	0.09	0	0	0	92	92	92	0.11	0	0	0
	40	122	122	122	0.11	0	0	0	122	122	122	0.17	0	0	0
	60	182	182	182	0.11	0	0	0	182	182	182	0.13	0	0	0
	80	236	236	242	0.18	0	1	0	236	236	236	0.17	0	1	0
	200	260	602	602	*time*	57	100732	4	260	314	314	*time*	17	78306	16
	300	104	902	902	*time*	88	273318	0	104	274	274	*time*	62	335686	0
	500	0	1502	1502	*time*	100	170322	0	0	174	174	*time*	100	165584	0
	1000	0	3002	3002	*time*	100	52500	0	0	60	60	*time*	100	47637	0

$z_l > 0$) for 13 instances within given time limit. Taking into account that code design problem is an offline problem, one can implement BC$_4$ method to construct a (J, K)–regular code providing sufficiently large time.

7. CONCLUSIONS

In this thesis, we first consider to design decoders with high error correction capability in Chapter 4. In particular, we consider a Branch–and–Price (BP) algorithm for LDPC decoding in Section 4.2. We explain a method to repair infeasibilities at the nodes of BP algorithm. Besides, we implement different techniques to generate an upper bound for early prunning the branch–and–bound tree of BP algorithm. Computational experiments show that our BP algorithm is not as fast as CPLEX running on EM formulation in LDPC decoding. Some future research on generating tight upper bound for feasible codeword is necessary in order to obtain better BP performance.

In Chapter 5 we proposed optimization–based sliding window decoders for SC codes, namely complete window (CW), finite window (FW), repeating windows (RW) decoders. We explained how one can utilize these algorithms to practically decode infinite dimensional convolutional codes and introduce convolutional code (CC) decoder. The computational results indicate that within the given time limit sliding window decoders find better feasible solutions in shorter time compared with exact model decoder (EMD). For each proposed decoder, we implement some binary (SB) and all binary (AB) variants. Among the sliding window decoders, AB approach is better than SB due to starting solution advantage.

For the decoding of convolutional codes, our proposed ABFW algorithm is the best among all methods in terms of both computational time and solution quality. One can obtain better solutions by increasing the window size in the expense of computational time.

Although, RW approach reveals worse performance than FW method, it can still be a nice candidate to decode time invariant convolutional codes where all windows are same. In such a case, one needs to store a single window model instead of m. This can decrease the memory usage and improve the computational time.

Gallager A and B algorithms are popular in practical applications. Compared with ABFW approach, these algorithms give poor quality solution in shorter time. Our proposed algorithm ABFW can contribute to the communication system reliability by providing near optimal decoded codewords. It is applicable in settings such as deep space communications where obtaining a high–quality decoding within reasonable amount of time is crucial.

In Chapter 6, we consider LDPC code design problem and provide an MIP formulation for girth feasibility problem. For the solution of problem, we propose a branch–and–cut (BC) method. We analyze structural properties of the problem and improve our BC algorithm by using techniques such as variable fixing, adding valid inequalities to model and providing an initial solution using a heuristic. Computational experiments indicate that each of these techniques improve BC one step further. Among all, the method which combines all of these strategies, i.e. method BC_4, can solve largest number of instances to optimality and gives smallest gap values on average in acceptable amount of time. One important gain of the method is that it can provide an evidence whether there can be a (J, K)–regular code or not.

In this study, our focus has been on (J, K)–regular codes. In telecommunication applications irregular LDPC codes are also utilized. Hence, extending these techniques to irregular LDPC codes can be a track of future research. Spatially–coupled (SC) LDPC codes are another code family which become popular due to their channel capacity approaching error correction capability. Design of SC LDPC codes without small cycles will be a valuable contribution to the future communication standards.

REFERENCES

1. Vacchione, J. D., *et al.*, “Telecommunications antennas for the Juno mission to Jupiter”, in *Proc. 2012 IEEE Aerospace Conf.*, pp. 1–16.

2. DeBoy, C. C. *et al.*, “The RF Telecommunications System for the New Horizons Mission to Pluto”, in *Proc. 2004 IEEE Aerospace Conf.*, pp. 1463–1478.

3. Gallager, R. G., “Low-density parity-check codes,” *IRE Transactions on Information Theory*, vol 8, no. 1, pp. 21–28, January 1962.

4. Tanner, R. M., “A recursive approach to low complexity codes,” *IEEE Transactions on Information Theory*, vol IT-27, no. 5, pp. 533–547, September 1981.

5. Zhang, J., and M. P. C., Fossorier, “Shuffled iterative decoding,” *IEEE Trans. on Commun.*, vol 53, no. 2, pp. 209–213, 2005.

6. Chen, J., A., Dholakia, E., Eleftheriou, M. P. C., Fossorier, and X. Y., Hu, “Reduced–complexity decoding of LDPC codes,” *IEEE Trans. on Commun.*, vol 53, no. 8, pp. 1288–1299, 2005.

7. Berlekamp, E. R., R. J., McEliece, and H. C. A., van Tilborg, “On the inherent intractability of certain coding problems,” *IEEE Trans. Inf. Theory*, vol. 24, pp. 384–386, May 1978.

8. Feldman, J., and D. R., Karger, “Decoding turbo-like codes via linear programming,” *Journal of Computer and System Sciences*, vol 68, pp. 733-752, 2004.

9. Feldman, J., M. J., Wainwright, and D. R., Karger, “Using linear programming to decode binary linear codes,” *IEEE Transactions on Information Theory*, vol 51, no. 3, pp. 954–972, March 2005.

10. Elias, P., “Coding for Noisy Channels”, MIT Res. Lab. of Electronics, Cambridge, MA, IRE Convention Rec., Part 4, pp. 37–46, 1955.

11. Viterbi, A. J., “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm”, *IEEE Trans. on Information Theory*, vol. 13, no. 2, pp. 260–269, April 1967.

12. Brice, P., W. Jiang, and G. Wan, “A Cluster-Based Context-Tree Model for Multivariate Data Streams with Applications to Anomaly Detection”, *INFORMS J. on Computing*, vol. 23, no. 3, pp. 364–376, September 2010.

13. Fano, R. M., “A heuristic discussion of probabilistic decoding,” *IEEE Trans. Inform. Theory*, vol. IT-9, no. 2, pp. 64–73, April 1963.

14. Sh. Zigangirov, K., “Some sequential decoding procedures,” *Probl. Peredachi Inf.*, 2, pp. 13–25, 1966.

15. Jelinek, F., “A fast sequential decoding algorithm using a stack,” *IBM J. Res. and Dev.*, 13, pp. 675–685, November 1969.

16. Han, Y. S., and P.-N., Chen, “Sequential decoding of convolutional codes” in *Wiley Encyclopedia of Telecommunications*, Wiley, 2003.

17. Bocharova, I. E., B. D. Kudryashov, and R. Johannesson, “LDPC Convolutional Codes versus QC LDPC Block Codes in Communication Standard Scenarios”, in *2014 IEEE Int. Symp. on Information Theory*, pp. 2774–2778.

18. Keha, A. B., and T. M., Duman, “Minimum distance computation of LDPC codes using branch and cut algorithm,” *IEEE Transactions on Communications*, vol 58, no. 4, pp. 1072–1079, 2010.

19. McGowan, J. A., and R. C., Williamson, “Loop removal from LDPC codes,” *IEEE Inf. Theory Workshop*, pp. 1–4, 2003.

20. Hu, X. Y., E., Eleftheriou, and D. M., Arnold, “Regular and irregular progressive edge-growth Tanner graphs,” *IEEE Trans. on Inf. Theory*, vol 51, pp. 386–398, 2005.

21. Compello, J., and D. S., Modha, “Extended bit–filling and LDPC code design,” *Proc. of the IEEE Globecom Conf.*, vol 2, pp. 985–989, 25–29 November 2001.

22. Dinoi, L., F., Scottile, and S., Benedetto, “Design of variable–rate irregular LDPC codes with low error floor,” *IEEE Int. Conf. on Commun.*, vol 1, pp. 647–651, 16–20 May 2005.

23. Poulliat, C., M., Fossorier, and D., Declercq, “Design of regular (2, d_c)–LDPC codes over GF(q) using their binary images,” *IEEE Trans. on Commun.*, vol 56, no. 10, pp. 1626–1635, October 2008.

24. Lu, J., and J. M. F., Moura, “TS–LDPC codes: Turbo–structured codes with large girth,” *IEEE Trans. on Inf. Theory*, vol 53, no. 3, pp. 1080–1094, March 2007.

25. Chen, H., and Z., Cao, “A modified PEG algorithm for construction of LDPC codes with strictly concentrated check–node degree distributions,” *Wireless Commun. and Networking Conf.*, pp. 564–568, 11–15 March 2007.

26. Healy, C. T., and R. C., de Lamare, “Decoder–optimised progressive edge growth algorithms for the design of LDPC codes with low error floors,” *IEEE Commun. Lett.*, vol 16, no. 6, pp. 889–892, June 2012.

27. Divsalar, D., S., Dolinar, and C., Jones, “Low–rate LDPC codes with simple protograph structure,” *Int. Symp. on Inf. Theory*, pp. 1622–1626, 4–9 September 2005.

28. El-Khamy, M., J., Hou, and N., Bhushan, “Design of rate–compatible structured LDPC codes for hybrid ARQ applications,” *IEEE J. on Selected Areas in Commun.*, vol 27, no. 6, pp. 965–973, August 2009.

29. Etesami, S. J., and W., Henkel, “A protograph construction for LDPC unequal error protection codes,” Bremen Jacobs University, August 2009.

30. Bonello, N., S., Chen, and L., Hanzo, “Construction of regular quasi–cyclic protograph LDPC codes based on Vandermonde matrices,” *IEEE Trans. on Vehicular Technology*, vol 57, no. 4, pp. 2583–2588, July 2008.

31. Myung, S., K., Yang, and J., Kim, “Quasi–cyclic LDPC codes for fast encoding,” *IEEE Trans. on Inf. Theory*, vol 51, no. 8, pp. 2894–2901, August 2005.

32. Li, Z., and B. V. K. V., Kumar, “A class of good quasi–cyclic low–density parity check codes based on progressive edge growth graph,” *Conf. Record of the Thirty–Eight Asilomar Conf. on Signals, Systems and Computers*, vol 2, pp. 1990–1994, 7–10 November 2004.

33. Prompakdee, P., W., Phakphisut, and P., Supnithi, “Quasi cyclic–LDPC codes based on PEG algorithm with maximized girth property,” *2011 Int. Symp. on Intelligent Signal Processing and Commun. Syst. (ISPACS)*, pp. 1–4, 7–9 December 2011.

34. Venkiah, A., D., Declercq, and C., Poulliat, “Design of cages with a randomized progressive edge–growth algorithm,” *IEEE Commun. Lett.*, vol 12, no. 4, pp. 301–303, April 2008.

35. Xiao, H., and A. H., Banihashemi, “Improved progressive–edge–growth (PEG) construction of irregular LDPC codes,” *IEEE Commun. Lett.*, vol 8, no. 12, pp. 715–717, December 2004.

36. Myung, S., K., Yang, and J., Kim, “Lifting methods for quasi–cyclic LDPC codes,” *IEEE Commun. Lett.*, vol 10, no. 6, pp. 489–491, June 2006.

37. Liu, Z., and D. A., Pados, “LDPC codes from generalized polygons,” *IEEE Trans. on Inf. Theory*, vol 51, no. 11, pp. 3890–3898, November 2005.

38. Yedidia, J., and Y., Wang, “Method for determining quasi–cyclic low–density parity–check code, and system for encoding data based on quasi–cyclic low–density parity–check code”, WO Patent 2013047258 A1, 4 April 2013.

39. Wang, Y., S. C., Draper, and J. S., Yedidia, “Hierarchical and high–girth QC LDPC codes,” *IEEE Trans. on Inf. Theory*, vol 59, no. 7, pp. 4553–4582, July 2013.

40. Psota, E., and L. C., Pérez, “Iterative construction of regular LDPC codes from independent tree–based minimum distance bounds,” *IEEE Commun. Lett.*, vol 15, no. 3, pp. 334–336, March 2011.

41. Ryan, W., and S., Lin, “Low–density parity–check codes,” in *Channel Codes: Classical and Modern.* 1st Ed. New York: Cambridge Univ. Press, 2009.

42. Leiner, B. M. J., “LDPC codes - a brief tutorial,” *Wien Technical University*, 2005.

43. Shokrollahi, A., “LDPC codes: an introduction,” *Digital Fountain, Inc.*, 2003.

44. MacKay, D. J. C., *Information theory, inference, and learning algorithms.* Cambridge, United Kingdom: Cambridge Univ. Press, 2003.

45. Feldman, J., T., Malkin, R. A., Servedio, C., Stein, and M. J., Wainwright, “LP decoding corrects a constant fraction of errors,” *IEEE Transactions on Information Theory*, vol. 53, No. 1, 2007.

46. Karmarkar, N., “A new polynomial-time algorithm for linear programming,” *Combinatorica*, vol. 4, no. 4, pp. 373-395, 1984.

47. Balatsoukas–Stimming, A., “Belief Propagation and LDPC Code Design (A Review),” *Technical University of Crete*, 2011.

48. Smith, B. M., “Modeling”, *Handbook of Constraint Programming*, Eds. F., Rossi, P. , van Beek, and T., Walsh, Elsevier, 2006.

49. Kirkpatrick, S., C. D., Gelatt, M. P., Vecchi, “Optimization by simulated annealing”, *Science*, New Series, vol. 220, no. 4598, pp. 671–680, 1983.

50. Jimenez-Feltström, A., and K., Sh. Zigangirov, “Time-varying periodic convolutional codes with low-density parity-check matrix,” *IEEE Trans. on Information Theory*, vol. 45, pp. 2181–2191, 1999.

51. Kudekar, S., T. J., Richardson, and R. L., Urbanke, “Threshold saturation via spatial coupling: why convolutional LDPC ensembles perform so well over the BEC,” *IEEE Trans. on Information Theory*, vol 57, no. 2, pp. 803–834, February 2011.

52. Ashrafi, R., and A. E., Pusane, “Spatially-coupled communication system for the correlated erasure channel,” *IET Communications*, vol. 7, no. 8, pp. 755–765, May 2013.

53. Lentmaier, M., A., Sridharan, D. J., Costello, Jr., and K., Sh. Zigangirov, “Iterative decoding threshold analysis for LDPC convolutional codes,” *IEEE Trans. on Information Theory*, vol. 56, pp. 5274–5289, 2010.

54. Costello, Jr., D. J., A. E., Pusane, S., Bates, K., Sh. Zigangirov, “A comparison between LDPC block and convolutional codes,” *Proc. Inf. Theory and Applications Workshop*, San Diego, CA, USA, 2006.

55. Wolsey, L. A., *Integer programming.* New Jersey: Wiley, 1998, pp. 215–216.

56. Ferreira, D., R. Morabito, and S., Rangel, “Relax and fix heuristics to solve one-stage one-machine lot-scheduling models for small-scale soft drink plants,” *Computers and Operations Research*, vol 37, no. 4, pp. 684–691, April 2010.

57. Moallemi, C. C., and B., Van Roy, “Resource Allocation via Message Passing,” *INFORMS J. on Computing*, vol 23, no. 2, pp. 205–219, July 2010.

58. Sarıduman, A., A. E., Pusane, and Z. C., Taşkın, “An integer programming-based

search technique for error-prone structures of LDPC codes," *AEU - Int. J. of Electron. and Commun.*, vol. 68, no. 11, pp. 1097-1105, November 2014.

59. Richardson, T., "Error floors for LDPC codes," *Proc. Allerton Conferance on Communication Control and Computing*, vol 41, no. 3, pp. 1426–1435, September 2003.

60. Diestel, R., *Graph Theory.* 4th ed. Berlin, Germany: Springer-Verlag, June 2010.

61. Ahuja, R. K., T. L., Magnanti, and J. B., Orlin, *Network Flows, Theory, Algorithms and Applications.* 1st ed. New Jersey, USA: Prentice Hall, 1993.

62. Walsh, T., "General Symmetry Breaking Constraints," in *Principles and Practice of Constraint Programming*, Nantes, France: Springer, 2006, vol 4204, pp. 650–664.

63. Sherali, H. D., and J. C., Smith, "Improving discrete model representations via symmetry considerations," *Management Science*, vol 47, no. 10, pp. 1396–1407, 2001.

64. Hu, X. Y., E., Eleftheriou, and D. M., Arnold, "Progressive edge-growth Tanner graphs," *Proc. IEEE Global Telecommunications Conf.*, vol 2, pp. 995–1001, 2001.