Optimization–Based Decoding Algorithms for LDPC Convolutional Codes in Communication Systems

In a digital communication system, information is sent from one place to another over a noisy communication channel. It may be possible to detect and correct errors that occur during the transmission if one encodes the original information by adding redundant bits. Low–density parity–check (LDPC) convolutional codes, a member of the LDPC code family, encode the original information to improve error correction capability. In practice these codes are used to decode very long information sequences, where the information arrives in subsequent packets over time, such as video streams. We consider the problem of decoding the received information with minimum error from an optimization point of view and investigate integer programming–based exact and heuristic decoding algorithms for its solution. In particular, we consider relax–and–fix heuristics that decode information in small windows. Computational results indicate that our approaches identify near–optimal solutions significantly faster than a commercial solver in high channel error rates. Our proposed algorithms can find higher quality solutions compared with the state of the art iterative decoding heuristic.

Keywords: Telecommunications, integer programming, relax–and–fix heuristic.

1 Introduction and Literature Review

A digital communication system represents digital information flow from a source to sink over an unreliable environment, such as air or space. Daily communication with digital cellular phones (CDMA, GSM), high speed data modems (V. 32, V. 34), computer networks such as Internet, TV broadcasting or weather forecasting through digital satellites, image and data transmission to a space craft traveling in deep space as in the case of NASA's Pluto mission [1], optical recording in CD-ROMs are some examples of digital communication systems.

Since communication environments are unreliable in nature, errors may be introduced during transmission. In order to minimize the effects of these transmission errors, encoder applies certain techniques known as channel coding to add redundant bits to original information. When information reaches the receiver, decoder makes use of these redundant bits to detect and correct the errors in the received vector to obtain the original information. Work on channel coding, which started in the 1950s, has focused on turbo codes (obtained by parallel concatenation of two convolutional codes with an interleaver) and LDPC codes (described by low–density parity–check matrices).

LDPC codes find wide application areas such as the wireless network standard (IEEE 802.11n), WiMax (IEEE 802.16e) and digital video broadcasting standard (DVB-S2) due to their high error detection and correction capabilities. LDPC code family, first proposed by Gallager in 1962, has sparse parity–check matrix representations [2]. In the following years, LDPC codes were represented by Tanner graphs, which belong to a special type of bipartite graphs that are intensively studied in graph theory [3, 4]. Sparsity property of the parity–check matrix gives rise to the development of iterative message–passing decoding algorithms (such as Belief Propagation (BP)) on Tanner graph with low complexity [5] – [8]. Ease of the application of iterative message–passing decoding algorithms brings the advantage of low decoding latency.

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 [9]. However, these

heuristic decoding algorithms do not guarantee optimality of the 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 [10, 11]. However, the proposed models do not allow decoding in an acceptable amount of time for codes with practical lengths.

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 [12]. Convolutional codes find application areas such as deep–space and satellite communication starting from early 1970s. They can be decoded with Viterbi algorithm, which provides maximum–likelihood decoding by dynamic programming, by dividing the received vector into smaller blocks of bits. Although Viterbi algorithm has a high decoding complexity for long constraint lengths, it can easily implemented on hardware due to its highly repetitive nature [13, 14]. For long block lengths, sequential decoding algorithms such as Fano algorithm [15] and later stack algorithm that is developed by Zigangirov [16] and independently by Jelinek [17] fit well. While Viterbi algorithm finds the best codeword, sequential decoding is suboptimal since it focuses on a certain number of likely codewords [18].

LDPC Convolutional (LDPC–C) codes, introduced by J. Feltström and Zigangirov in 1999, are preferred to LDPC block codes in decoding for the cases where information is obtained continuously. They can be decoded by sliding window decoders which implement iterative decoding algorithms (such as BP) at each window [19]. Although LDPC–C codes provide short–delay and low–complexity in decoding, they are not in communication standards such as WiMax and DVB-S2 yet [20].

In this study, we consider LDPC–C 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 Section 3. Our proposed decoders can be used in a 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.

The rest paper is organized as follows: we define the problem in more detail in the next section. Section 3 explains the proposed decoding techniques. We give the corresponding computational results in Section 4. Some concluding remarks and comments on future work appear in Section 5.

2 Problem Definition

Digital communication systems transmit information from a sender to a receiver over a communication channel. Communication channels are unreliable environments, such as air, that many sender–receiver pairs share. Hence, during transmission some of the transmitted symbols can be lost or their values can change. In coding theory, information is encoded in order to overcome the occurence of such errors during the transmission. Let the information to be sent be represented by a k–bits long sequence $\mathbf{u} = u_1u_2...u_k$ ($u_i \in \{0, 1\}$). In order to test whether the information is sent correctly or not, parity bits are added by the encoder. This is done with a $k \times n$ generator matrix $\mathbf{G}$ through the operation $\mathbf{v} = \mathbf{uG}$ (mod 2). As a result, an n–bits long ($n \geq k$) codeword $\mathbf{v} = v_1v_2...v_n$ ($v_i \in \{0, 1\}$) is obtained. Without loss of generality, we can assume that the first k bits of the codeword are information bits, and the remaining $n - k$ bits are parity bits.

As shown in Figure 1, when the encoded information reaches receiver as an n–bits long vector $\mathbf{r}$, the correctness of the vector is tested at the decoder using the parity bits. If $\mathbf{r}$ is detected to be erroneous, the decoder attempts to determine the locations of the errors and fix them [21, 22]. Hence, the information $\mathbf{u}$ sent from the source is estimated as $\hat{\mathbf{u}}$ at the sink.

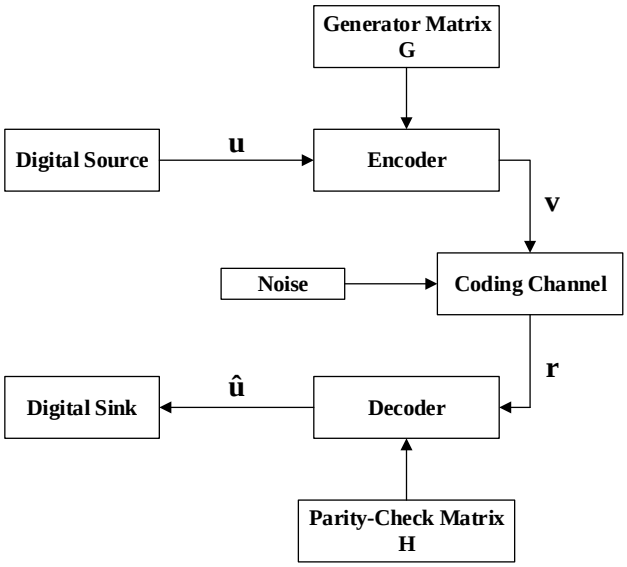

Figure 1: Digital communication system diagram

There are several models used to model the noisy communication channels. In our study, we employ binary symmetric channel (BSC) model for noisy channel. As shown in Figure 2, a transmitted bit is received correctly with probability $1-p$ or an error occurs with probability p [23].

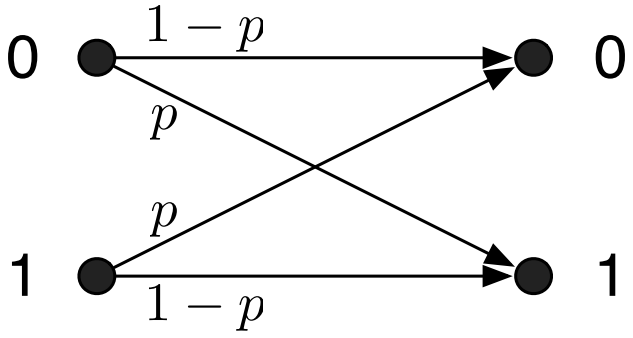

Figure 2: Binary symmetric channel

In BSC, the received vector $\mathbf{r}$ includes both correct and incorrect bits. Although we do not know which bits are erroneously received, flipping the bit fixes the error when the error location is known. Hence, the aim of the decoder is to determine the error locations in BSC.

As explained above, original information $\mathbf{u}$ is encoded with $k \times n$ generator matrix $\mathbf{G}$. Received vector $\mathbf{r}$ is decoded with a parity–check matrix $\mathbf{H}$ of dimension $(n-k)\times n$. $(J, K)-$regular

LDPC codes are member of linear block codes that can be represented by a parity–check matrix $\mathbf{H}$ with $J-$many ones at each column and $K-$many ones at each row. An example of a parity–check matrix from $(3,6)-$regular LDPC code family is given in Figure 3.

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

Figure 3: A parity–check matrix from $(3,6)-$regular LDPC code family

Vectors $\mathbf{v}$ that satisfy the equation $\mathbf{vH}^{\mathrm{T}} = \mathbf{0}$ (mod 2) are codewords. For any original information $\mathbf{u}$, encoded vector $\mathbf{v} = \mathbf{uG}$ (mod 2) is a codeword. The channel decoder concludes that whether the received vector $\mathbf{r}$ has changed or not by checking the value of expression $\mathbf{rH}^{\mathrm{T}}$ is equal to vector $\mathbf{0}$ in (mod 2) or not.

LDPC codes can also be represented using Tanner graphs [8]. On one side of this bipartite graph, there are n variable nodes standing for n codeword symbols of the code and on the other side of the bipartite graph there are $(n-k)$ check nodes corresponding to $(n-k)$ parity–check equations defined by each row of the $\mathbf{H}$ matrix. Here, $\mathbf{H}$ matrix is the bi–adjacency matrix of Tanner graph. This representation of LDPC codes brings the advantage of applying the iterative decoding and other decoding algorithms easily. Figure 4 shows Tanner graph representation of $\mathbf{H}$ matrix defined in Figure 3.

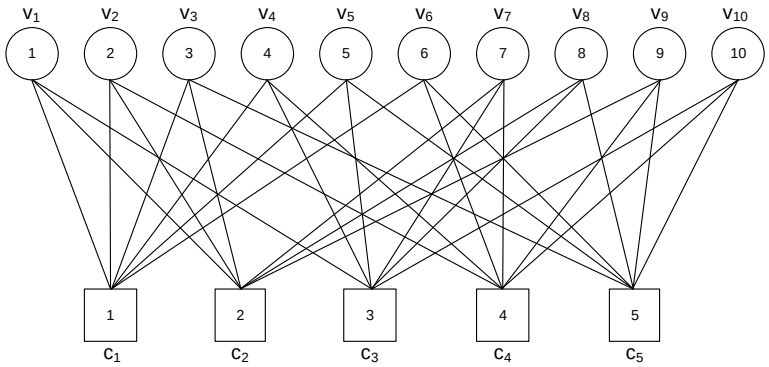

Figure 4: Tanner graph representation of the parity–check matrix given in Figure 3

In our study, we focus on LDPC–C codes. LDPC–C codes divide the original information into smaller blocks and decode each block by considering the previous blocks [19]. 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, an LDPC–C code consists of m_s–many small parity–check matrices at each column, where m_s parameter represents the width of the ribbon. The diagonal pattern is obtained by shifting the columns down as the dimension increases.

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

Figure 5: Generic structure of an LDPC–C code

These codes find application areas such as satellite communication and video streams where the information is received continuously. Finite dimension LDPC–C codes, namely terminated LDPC–C codes, can be obtained by limiting the dimension of LDPC–C code by specifying a finite row or column size [24]. In Figure 6, we give an example (3, 6)–regular terminated LDPC–C code of dimensions $(24, 36)$ obtained by limiting the number of columns to 36.

One can observe that the number of ones in the first five and the last six rows of the code in Figure 6 is less than 6. Note that (3, 6)–regularity holds for the intermediary rows.

The repeating structure of LDPC–C codes allow the application of sliding window decoding approaches which use iterative decoding algorithms (such as BP) at each window [24]. Although iterative decoding algorithms are easily applicable, 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 with terminated LDPC–C codes on BSC. Then we generalize these decoding algorithms to decode practically infinite length received vectors with LDPC–C codes. Our proposed decoders can

$$
\mathbf{H} = \left[\begin{array}{l}
010000000000000000000000000000000000 \\
101000000000000000000000000000000000 \\
010101000000000000000000000000000000 \\
101010010000000000000000000000000000 \\
011001100100000000000000000000000000 \\
100110011001000000000000000000000000 \\
000101101010010000000000000000000000 \\
000010010101101000000000000000000000 \\
000000100110010101000000000000000000 \\
000000001001101010010000000000000000 \\
000000000010011001100100000000000000 \\
000000000000100110011001000000000000 \\
000000000000000101101010010000000000 \\
000000000000000010010101101000000000 \\
000000000000000000100110010101000000 \\
000000000000000000001001101010010000 \\
000000000000000000000010011001100100 \\
000000000000000000000000100110011001 \\
000000000000000000000000000101101010 \\
000000000000000000000000000010010101 \\
000000000000000000000000000000100110 \\
000000000000000000000000000000001001 \\
000000000000000000000000000000000010 \\
000000000000000000000000000000000000
\end{array}\right]
$$

Figure 6: A (3, 6)–regular terminated LDPC–C code

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

3 Solution Methods

We propose three different sliding window decoders for terminated LDPC–C codes and a sliding window decoder for LDPC–C codes. The terminology used in this paper is summarized in Table 1.

3.1 Mathematical Formulation

The decoding problem of a terminated LDPC–C code can be represented with Exact Model (EM) which is given in [25]. 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_{ij} is the $(i, j)-$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.

Table 1: List of symbols

	Parameters
k	length of the original information
$\mathbf{G}$	generator matrix
$\mathbf{H}$	parity-check matrix
$\hat{\mathbf{y}}$	received vector
n	length of the encoded information, # of columns in $\mathbf{H}$
p	error probability in BSC
m	# of rows in base permutation matrix
m_s	width of the ribbon of an LDPC–C code
C	set of check nodes
V	set of variable nodes
w	height of the window
h_s	horizontal step size
v_s	vertical step size
r	h_s/v_s ratio
	Decision Variables
f_i	ith bit of the decoded vector
k_j	an auxiliary integer variable

Exact Model:

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

s.t.

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

$$f_i \in \{0, 1\}, \ \forall i \in V, \tag{3}$$

$$k_j \geq 0, \ k_j \in \mathbb{Z}, \ \forall j \in C. \tag{4}$$

Constraints (2) guarantee that the decoded vector $\mathbf{f}$ satisfies the equality $\mathbf{f}\mathbf{H}^{\mathrm{T}} = \mathbf{0} \pmod 2$. The objective (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 (3) and (4) set the binary and integrality restrictions on decision variables **f** and **k**, respectively.

Since EM is an integer programming formulation, it is not practical to obtain an optimal decoding using commercial solver for real–sized (approximately $n = 4000$) terminated LDPC–C codes. This can be seen from the computational experiments in Section 4. Instead, we will look at terminated LDPC–C code in small windows and solve limited models at each window.

Note that LDPC–C code decoding problem cannot be represented as a compact mathematical formulation since this would require infinite number of decision variables f_i and constraints.

3.2 LDPC–C Code Generation

We generate a terminated LDPC–C code with the help of a (5, 10)–regular base permutation matrix $\mathbf{H}_{base}$ with dimensions $(m, 2m)$ for some m. We utilize the algorithm described in [26] to obtain a (5, 10)–regular $\mathbf{H}_{base}$ that does not include length–4 cycles in its Tanner graph. Then, we split $\mathbf{H}_{base}$ matrix into two matrices, namely lower triangular **A** and upper triangular **B** as shown in Figure 7.

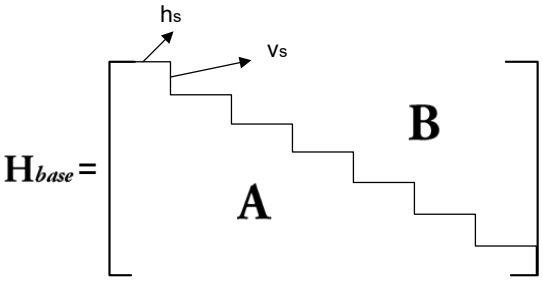

Figure 7: **A** and **B** matrices

We divide $\mathbf{H}_{base}$ with a horizontal step length h_s and a vertical step length v_s. We have $r = h_s/v_s = 2$, since there is a number c such that $h_s c = 2m$ and $v_s c = m$. These **A** and **B** matrices are repeatedly located until the desired terminated LDPC–C code size is obtained. After t–many repetitions, the size of the terminated LDPC–C code is $(m(t+1), 2mt)$ as shown in Figure 8. The ribbon size is $m_s = m$ for such a code.

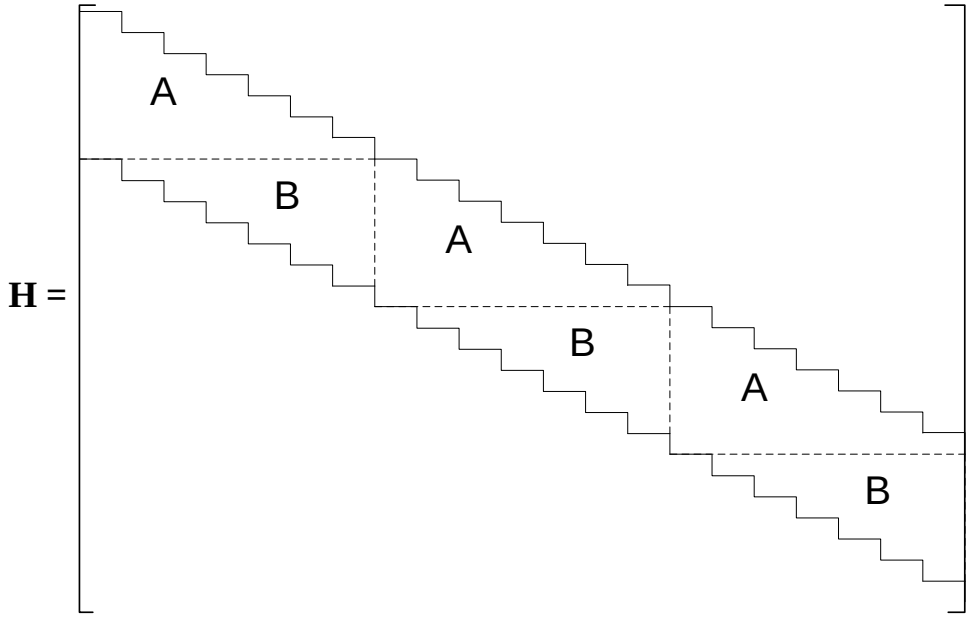

Figure 8: (5, 10)–regular terminated LDPC–C code

3.3 Sliding Window Decoders

In practical applications, special structure of the LDPC–C codes allows to decode these codes with sliding window decoders [27, 28]. As explained in Section 2, LDPC–C 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 LDPC–C 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 LDPC–C code corresponding to the window, all entries in the columns after the window are zero with this window dimension selection.

Algorithm 1 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. Sliding window decoders that are considered in the literature often implement BP algorithm to decode the windows. We investigate the performance of BP algorithm in sliding window decoder via computational experiments in Section 4.

Algorithm 1: (Generic Sliding Window)

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 vector/codeword

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 (2). Our sliding window decoding algorithm has main steps that are given in Algorithm 2.

Algorithm 2: (Sliding Window)

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 (2) with the fixed bits
5. **If** all bits decoded, **Then** STOP, **Else** go to Step 1

Output: A decoded vector/codeword

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 terminated LDPC–C codes and an LDPC–C code decoder, that are explained in the next sections.

3.3.1 Complete Window (CW) Decoder

Complete window (CW) decoder requires that binary code has finite dimension. Hence, it is applicable only for terminated LDPC–C 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 9, 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.

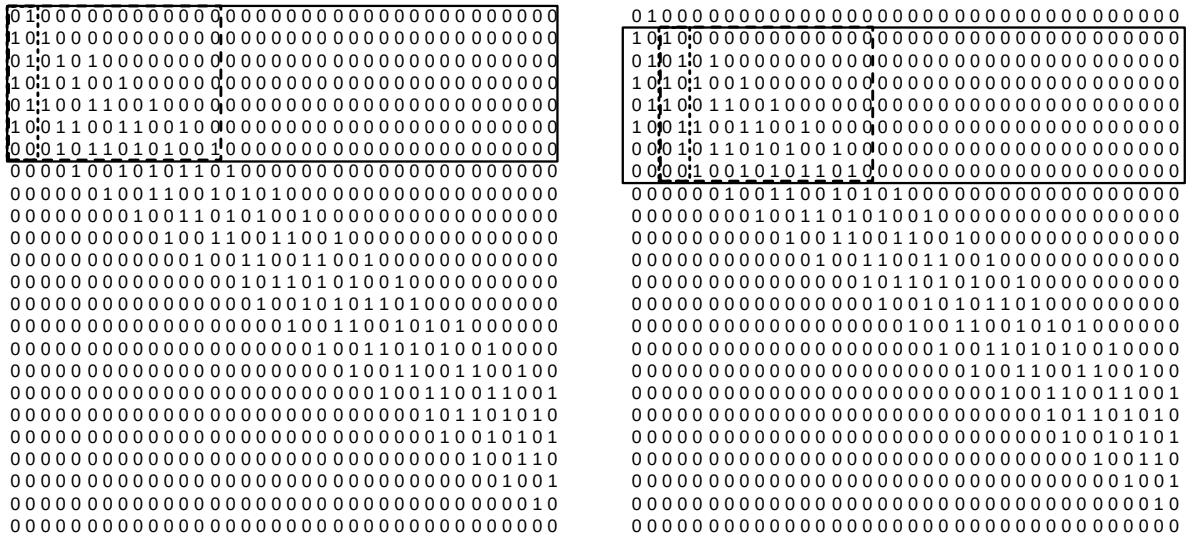

Figure 9: Sliding window in CW decoder

One can see that the dashed rectangle in Figure 9 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 [29, 30]. 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.

3.3.2 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.

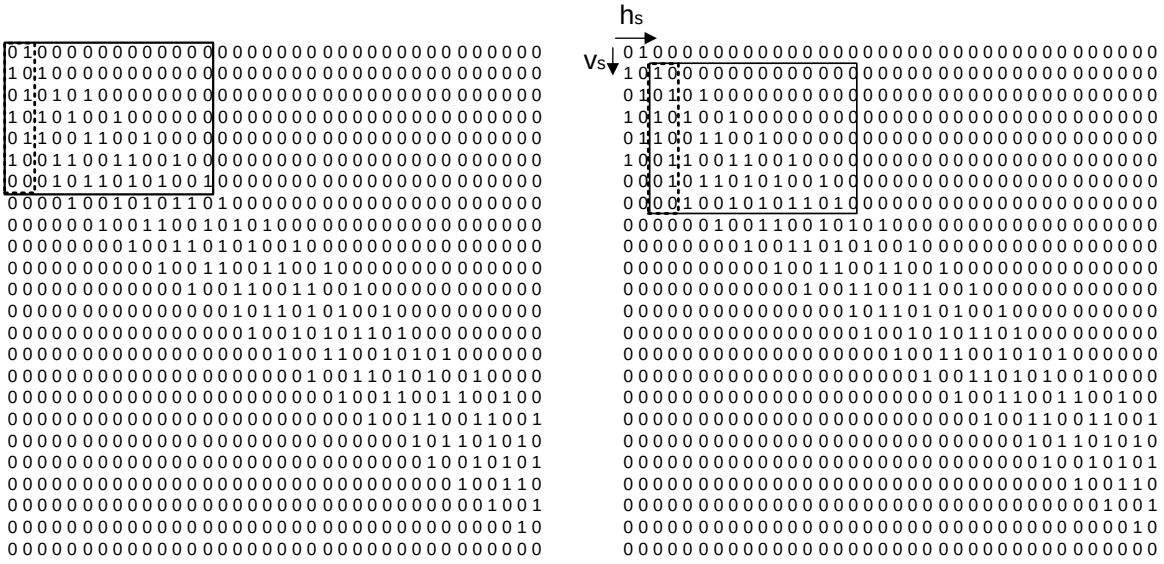

Figure 10: Sliding window in FW decoder

The window position can be seen in Figure 10 as the window slides. The previous decoded bits appear as a constant in constraints (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–many new constraints.

3.3.3 Repeating Windows (RW) Decoder

As explained in Section 3.2, a terminated LDPC–C code is obtained by repetitively locating **A** and **B** matrices. As can be seen in Figure 11, a window will come out again after m–iterations, where m is the number of rows in $\mathbf{H}_{base}$.

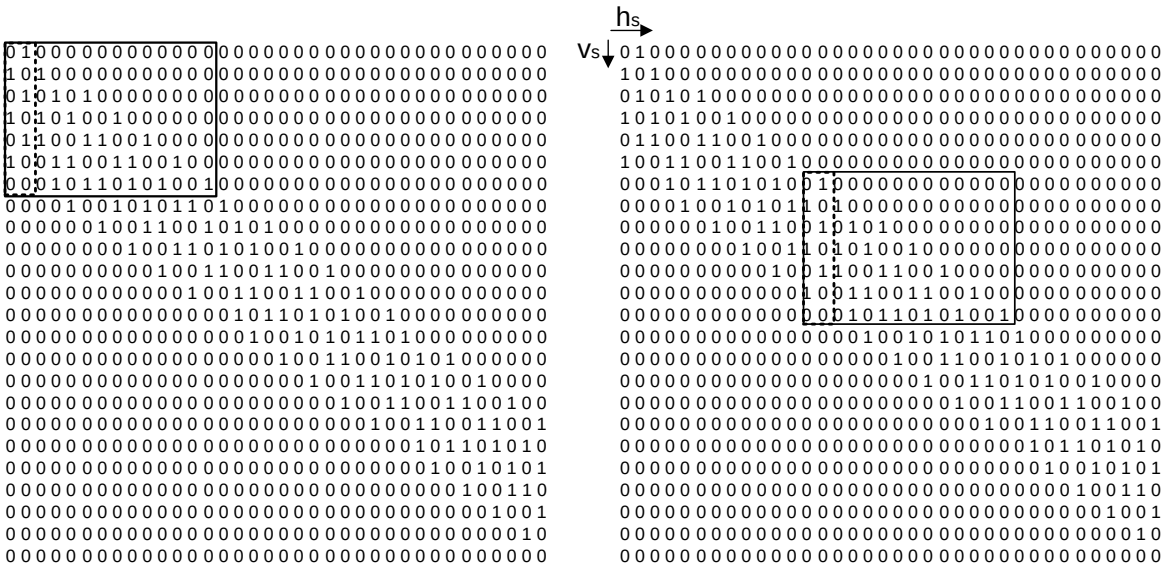

Figure 11: Sliding window in RW decoder

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 (2) and the objective function coefficients change but the coefficients of the decision variables stay the 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 at 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.

3.3.4 LDPC 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 $\mathbf{H}$ matrix. Hence, they are applicable for terminated LDPC–C codes. However, as explained in Section 2, LDPC–C codes are practically infinite dimensional codes and cannot be represented by a compact $\mathbf{H}$ matrix on the computer. On the other hand, we can store a part of the LDPC–C code as given in Figure 12, and represent the (i, j)th entry of the code with a function.

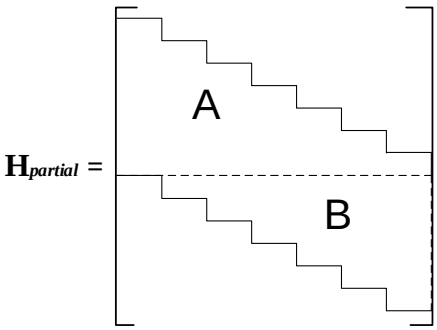

Figure 12: A part of LDPC–C code

Hence, we can represent the current window model using $\mathbf{H}_{partial}$ matrix. This allows the application of FW and RW decoders to LDPC–C codes. Note that our CW decoder is not applicable to CC, since it takes into account all bits of the received vector.

4 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 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 12.8.0 to solve EM for the current window (see Step 1 of Algorithm 2). 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 terminated LDPC–C code. That is, for a terminated LDPC–C code of size $(n/2, n)$ we have $n/2-$many constraints (2) and $n-$many f_i decision variables in EM. We again utilize CPLEX for solving EM of EMD.

Table 2: Summary of methods

Method	# CPLEX Models	Move to Next Window	Window Size	# Vars k_j	# Vars f_i		# Vars int./binary/cont.
CW	1	Delete/add v_s–const.	$w \times n$	w	n	SB	$w/h_s/(n-h_s)$
		Update const. (2)				AB	$w/rw/(n-rw)$
FW	1	Delete/add v_s–const., h_s–vars	$w \times rw$	w	rw	SB	$w/h_s/(rw-h_s)$
		Update const. (2)				AB	$w/rw/0$
RW	m	Update obj. func. coeffs	$w \times rw$	mw	mrw	SB	$mw/mh_s/m(rw-h_s)$
		Update const. (2)				AB	$mw/mrw/0$
EMD	1	—	$(n/2) \times n$	$(n/2)$	n	—	$(n/2)/n/0$

We summarize the solution methods in Table 2. "# CPLEX Models" gives the number of CPLEX models stored in the memory. One needs to carry out operations given in "Move to Next Window" column when sliding the window of size "Window Size". In "# Vars" columns, we list the number of k_j and f_i decision variables and also give the number of integer, binary and continuous decision variables stored in the memory for SB and AB approaches of the methods. CC decoder is not listed in Table 2, since it is the application of FW and RW decoders to practically infinite dimensional codes.

Table 3: 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 3. We generate a base permutation matrix of size $(m, 2m) = (150, 300)$. We obtain a (5, 10)–regular terminated LDPC–C code $\mathbf{H}$ 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 terminated LDPC–C codes. In order to test the algorithms for LDPC–C 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 terminated LDPC–C code instance. Since we are testing a larger code length, i.e., $n = 12000$, for LDPC–C codes, we set a time limit of 5000 seconds to EMD to find a solution.

Table 4: Performance of EMD with $p = 0.02$ and 0.05

p	0.02					0.05					# Vars
n	z	CPU	Gap (%)	BER	# OPT	z	CPU	Gap (%)	BER	# OPT	int./binary
1200	24.3	0.15	0	0	10	57.3	113.65	0	0	10	600/1200
3600	73.4	0.46	0	0	10	647.9	2773.12	51.33	0.1719	4	1800/3600
6000	121.4	0.60	0	0	10	1225.6	2802.49	58.31	0.2043	3	3000/6000
8400	170.2	0.81	0	0	10	1532.1	3268.40	56.84	0.1860	2	4200/8400
12000	239.1	1.11	0	0	10	3457.1	4005.35	68.45	0.2886	2	6000/12000

Table 4 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 performance of a decoding algorithm is interpreted with Bit Error Rate (BER) in telecommunications literature. As given in Equation (5), BER is the rate of the decoded bits that are different than the original codeword [23]. Here, $\mathbf{y^o}$ is the original codeword, and $\mathbf{y^d}$ is the decoded vector.

$$BER = \frac{\sum_{i=0}^{n} \mid y_i^o - y_i^d \mid}{n} \tag{5}$$

"BER" column shows the observed BER values. The first four rows in Table 4 are the average results for terminated LDPC–C codes. The last row is the average result for LDPC–C 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 and BER increase when the code gets longer as expected. The last column "# Vars" shows the number of integer and binary decision variables in EMD with respect to the code length n. We calculate these values with the formula given in Table 2 as an indication of the decoder's complexity.

4.1 Terminated LDPC–C Code Results

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

Table 5: Complexity summary of CW, FW and RW decoders

w	small			large		
	CW	FW	RW	CW	FW	RW
	int./binary	int./binary	int./binary	int./binary	int./binary	int./binary
SB	151/2	151/2	22650/300	226/2	226/2	33900/300
AB	151/302	151/302	22650/45300	226/452	226/452	33900/67800

Since EM is an integer programming formulation, we can express the complexity of the decoders with the number of integer and binary decision variables. In Table 5, we list the number of decision variables in the window models of CW, FW and RW decoders using formulas in Table 2. We consider their SB and AB variants for small and large window sizes. We observe that, on the contrary of EMD, the number of integer and binary variables in the window model is independent of the code length n for these decoders. RW has the largest number of decision variables, since it stores m window models in the memory at a time.

Table 6: Performances of SBCW and ABCW

		w	small					large				
	p	n	z	CPU	Gap (%)	BER	# SOLVED	z	CPU	Gap (%)	BER	# SOLVED
SB	0.02	1200	24.3	6.84	0	0	10	24.3	8.92	0	0	10
		3600	73.4	31.52	0	0	10	73.4	39.57	0	0	10
		6000	121.4	87.81	0	0	10	121.4	98.83	0	0	10
		8400	170.2	156.51	0	0	10	170.2	180.15	0	0	10
	0.05	1200	61.2	25.72	5.51	0.0089	10	57.3	320.36	0	0	10
		3600	209.8	178.81	29.02	0.0162	10	178.9	1010.71	17.32	0	10
		6000	379.1	731.05	28.28	0.0225	10	524.4	1308.12	49.12	0.0584	10
		8400	653.4	2225.35	40.99	0.0442	10	853	1422.66	54.85	0.0748	10
AB	0.02	1200	24.3	10.43	0	0	10	24.3	10.70	0	0	10
		3600	73.4	37.94	0	0	10	73.4	55.38	0	0	10
		6000	121.4	89.92	0	0	10	121.4	102.17	0	0	10
		8400	170.2	175.67	0	0	10	170.2	190.47	0	0	10
	0.05	1200	57.3	15.82	0	0	10	57.3	76.39	0	0	10
		3600	182.2	72.32	18.87	0.0019	10	178.9	499.87	17.32	0	10
		6000	310.2	445.90	13.72	0.0029	10	323.1	831.27	16.37	0.0053	10
		8400	448.3	629.04	13.30	0.0055	10	522.9	1020.06	25.56	0.0169	10

Table 6 summarizes the results for CW decoder explained in Section 3.3.1. "Gap (%)" column represents the percent difference from the best known lower bound found by CPLEX while obtaining the results in Table 4. "# SOLVED" column shows the number of instances that can be decoded by the method.

When $p = 0.02$, CW decoder can find optimal solutions (Gap (%) = 0) which are original codewords (BER = 0) as EMD in Table 4. 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.

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 4) for SB and AB variants. As the window size gets larger, AB alternative gives better gap and time values compared with EMD. We observe for $n = 3600$ with large window that the original codewords (BER = 0) are not the nearest codewords to the received vectors (Gap (%) = 17.32), since the error probability is high ($p = 0.05$).

In general, with high error probability AB takes shorter time and obtains better gaps than SB (see results for $p = 0.05$ in Table 6, Table 7 and Table 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 7: Performances of SBFW and ABFW

		w	small					large				
	p	n	z	CPU	Gap (%)	BER	# SOLVED	z	CPU	Gap (%)	BER	# SOLVED
SB	0.02	1200	24.3	9.96	0	0	10	24.3	13.68	0	0	10
		3600	73.4	22.34	0	0	10	73.4	36.70	0	0	10
		6000	121.4	35.59	0	0	10	121.4	68.81	0	0	10
		8400	170.2	54.84	0	0	10	170.2	102.00	0	0	10
	0.05	1200	57.3	18.80	0	0	10	57.3	249.32	0	0	10
		3600	187	227.97	20.42	0.0047	10	178.9	2805.17	17.32	0	10
		6000	309.9	444.06	13.67	0.0028	10	302.4	4461.94	12.02	0.0002	10
		8400	445	2994.19	12.69	0.0048	10	438.4	8742.12	11.38	0.0025	10
AB	0.02	1200	24.3	10.48	0	0	10	24.3	11.23	0	0	10
		3600	73.4	28.00	0	0	10	73.4	38.09	0	0	10
		6000	121.4	46.95	0	0	10	121.4	62.18	0	0	10
		8400	170.2	57.75	0	0	10	170.2	91.84	0	0	10
	0.05	1200	57.3	14.09	0	0	10	57.3	109.47	0	0	10
		3600	181.2	47.52	18.45	0.0013	10	178.9	339.36	17.32	0	10
		6000	302.9	85.08	12.17	0.0004	10	301.7	1690.17	11.84	0	10
		8400	433.4	273.61	10.82	0.0016	10	425.9	1673.26	9.39	0	10

Results given in Table 7 shows that FW (see Section 3.3.2) can find optimal solution (original codeword) 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.

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 and BER 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

and BER values still depend on n due to error accumulation during the iterations. That is, if a window is not decoded as the original, this erroneous window solution will propagate to the upcoming window decodings. As the code length n gets larger, this effect becomes more apparent and BER values increase. If the window size is larger, then we are considering more information during the window decoding, which improves the gap and BER values. This effect is explained graphically in Figure 13.

Table 8: Performances of SBRW and ABRW

		w	small					large				
	p	n	z	CPU	Gap (%)	BER	# SOLVED	z	CPU	Gap (%)	BER	# SOLVED
SB	0.02	1200	24.3	10.71	0	0	10	24.3	53.48	0	0	10
		3600	73.4	33.71	0	0	10	73.4	289.14	0	0	10
		6000	170.5	345.47	12.71	0.0107	10	121.4	420.09	0	0	10
		8400	195.4	1255.38	10.00	0.0115	10	172.1	586.85	0	0	7
	0.05	1200	61.7	358.64	6.17	0.0092	10	65.5	48528.36	9.49	0.0118	10
		3600	217.4	2005.05	30.13	0.0222	10	217	10281.91	36.39	0.0215	6
		6000	369.1	6835.73	26.97	0.0228	10	—	—	—	—	0
		8400	516.2	12518.18	24.38	0.0222	10	—	—	—	—	0
AB	0.02	1200	24.3	10.64	0	0	10	24.3	13.36	0	0	10
		3600	73.4	33.07	0	0	10	73.4	40.93	0	0	10
		6000	121.4	49.99	0	0	10	121.4	69.90	0	0	10
		8400	170.2	64.83	0	0	10	170.2	134.22	0	0	10
	0.05	1200	57.3	13.41	0	0	10	57.3	247.41	0	0	10
		3600	181.6	114.39	18.62	0.0015	10	178.9	748.96	17.32	0	10
		6000	304.0	175.26	12.45	0.0008	10	301.7	3356.17	11.84	0	10
		8400	434.7	381.82	11.07	0.0020	10	425.9	3759.18	9.39	0	10

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

Comparison of Tables 7 and 8 shows that ABFW and ABRW methods give similar gap and BER 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 3) with high error probability.

4.2 LDPC–C Code Results

We also investigate the performance of FW and RW decoders for very large code length. For this purpose we take $n = 12000$, consider low ($p = 0.02$) and high ($p = 0.05$) error probabilities, small and large window sizes. CW method is inapplicable in practice for very large code lengths,

since the window model includes all bits of the vector as decision variables. Performance of EMD for $n = 12000$ is given in the last row of Table 4.

Table 9: Performances of FW and RW decoders with $n = 12000$

		w	small					large				
		p	z	CPU	Gap (%)	BER	# SOLVED	z	CPU	Gap (%)	BER	# SOLVED
FW	SB	0.02	239.1	80.75	0	0	10	239.1	297.30	0	0	10
		0.05	616.5	5361.52	11.07	0.0034	10	599.0	13457.22	8.66	0	10
	AB	0.02	239.1	90.33	0	0	10	239.1	133.12	0	0	10
		0.05	607.7	902.82	9.93	0.0023	10	599.0	2664.98	8.66	0	10
RW	SB	0.02	265.2	6382.53	8.17	0.0028	10	—	—	—	—	0
		0.05	716.6	19769.51	23.58	0.0206	10	—	—	—	—	0
	AB	0.02	239.1	96.00	0	0	10	239.1	147.21	0	0	10
		0.05	609.2	3041.51	10.18	0.0026	10	599.0	4313.20	8.66	0	10

Table 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 the 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 4), ABFW finds near optimal solutions in shorter time for all instances. However, EMD can solve only 2 instances to optimality when $p = 0.05$. ABFW is faster than ABRW as expected. Considering the computational results for LDPC–C 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 ABFW and ABRW methods by analyzing their decoding errors with respect to the original vector, i.e., BER values. In Figure 13, we give the detailed plots for the average decoding errors of 10 instances with $n = 12000$, $p = 0.05$, small and large window sizes that are reported in Table 9. We divide n into 120 sections each including 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 more information about the LDPC–C 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.

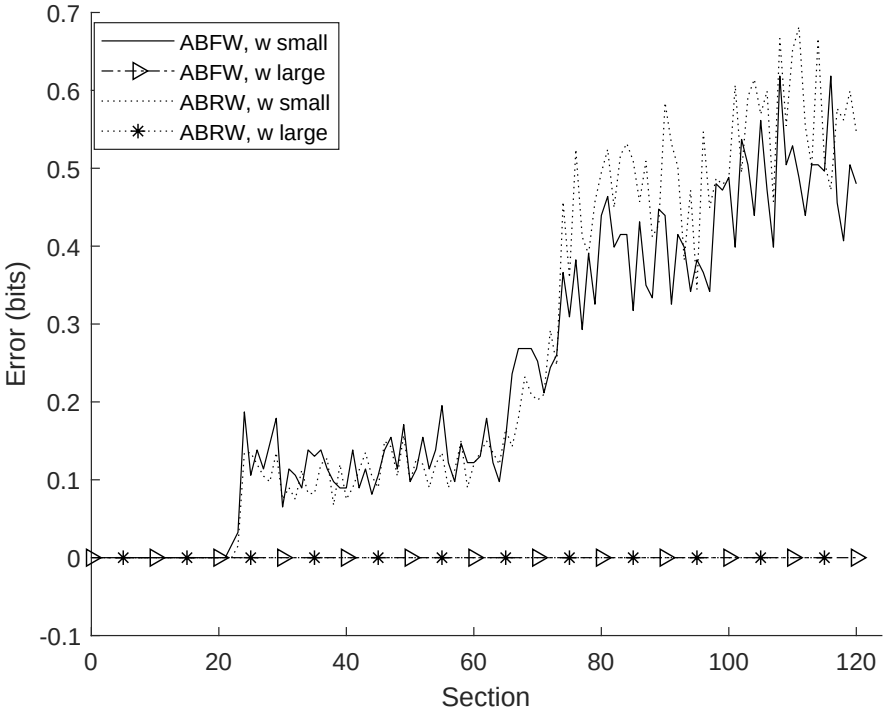

Figure 13: Error accumulation in decoding

As we report in Table 9, for small window size and $p = 0.05$, the average BER values of ABFW and ABRW methods that we plot in Figure 13 are 0.0023 and 0.0026, respectively. The error correction capability increases with larger window size, and the BER values drop to zero for both of the methods. For example, among 10,000 bits of the received vector that is decoded by ABFW method, on the average 23 bits (BER = 0.0023) are different from the original codeword when window size is small. As the window size gets larger, ABFW decoder can obtain the original codeword (BER = 0).

In practical applications, decoding of a received vector is done with iterative algorithms. BP is commonly implemented in today's communication systems. The performance of our proposed decoding algorithm (ABFW) can be tested against a sliding window decoder that uses the state of the art decoder BP for decoding windows (see Algorithm 1).

BP calculates a log–likelihood ratio (LLR) for each bit i, which is used as the message of the variable node i to the check nodes. LLRs can be calculated with Equation (6), where the

received value $\hat{y}_i$ of bit i is decoded as f_i.

$$LLR_i = log(\frac{Pr(\hat{y}_i|f_i=0)}{Pr(\hat{y}_i|f_i=1)}) \tag{6}$$

LLR_i is the initial message of the variable node i to the check node j, i.e. $m_{vc_{ij}}$. The message sent from the check node j to the variable node i, i.e. $m_{cv_{ji}}$, can be calculated with the following formula

$$m_{cv_{ji}} = 2\tanh^{-1}\left(\prod_{k\neq i}\tanh(m_{vc_{kj}}/2)\right). \tag{7}$$

The variable node i updates LLR_i as the summation of the received $m_{cv_{ji}}$ messages from its neigboring check nodes. Then, a decoded vector $\mathbf{f}$ is obtained by decoding $f_i = 0$ if $LLR_i < 0$, and $f_i = 1$ if $LLR_i > 0$. The message passing among the variable and check nodes continues until a codeword is obtained, i.e., $\mathbf{fH}^T = \mathbf{0}$ (mod 2), or a stopping condition is reached. The main steps of BP are given in Algorithm 3.

Algorithm 3: (Belief Propagation)

Input: Received vector, $\hat{\mathbf{y}}$

1. Calculate LLRs with Equation (6),
   and initialize the messages to the check nodes, i.e. $m_{vc_{ij}}$.
2. Calculate the messages to the variable nodes with Equation (7), i.e. $m_{cv_{ji}}$.
3. **For Each** variable node i
   Update LLR_i value with $m_{cv_{ji}}$ messages.
4. **If** $LLR_i < 0$, **Then** $f_i = 0$. **Else** $f_i = 1$.
5. **End For Each**
6. **If** all check nodes are satisfied or iteration limit is reached, **Then** STOP.
7. **Else** update $m_{vc_{ij}}$ messages to the check nodes, and go to Step 2.
8. **End If**

Output: A decoded vector/codeword

We apply BP 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 [33]. In such a case, the algorithm may terminate

with no conclusion. To avoid such a situation, we bound the maximum number of iterations by 500. Note that this may result in ending with an infeasible vector when the algorithm terminates. As noted in the literature, BP algorithm reveals better BER performance when the window size is $w > 5m$ [34]. Thus, in our computational experiments we pick the window size as $w = 6m + 1$ for BP.

Table 10 shows the average of 10 instances with BP algorithm when it is applied in the windows of sliding window decoder. As given in "# FEAS" column, BP decodes all received vectors to the original codewords (BER = 0) which are the nearest codewords (Gap = 0) when $p = 0.02$. BP cannot find a feasible solution for some of the cases when $p = 0.05$. That is, the decoded vector does not satisfy the equality $\mathbf{vH}^{\mathrm{T}} = \mathbf{0} \pmod 2$. Compared to the direct solution of EM model (in Table 4) BP has better BER and gap figures, which shows that the generated vectors are closer to the received vectors and the original codewords. On the other hand, our ABFW decoder (in Table 7) reveals better BER and gap performances than BP at the expense of CPU time.

Table 10: Performance of BP

p	0.02					0.05				
n	z	CPU	Gap (%)	BER	# FEAS	z	CPU	Gap (%)	BER	# FEAS
1200	24.3	13.10	0	0	10	57.3	12.70	0.00	0	10
3600	73.4	39.77	0	0	10	181.5	36.42	18.46	0.0014	4
6000	121.4	59.95	0	0	10	308.7	73.47	13.85	0.0021	1
8400	170.2	95.54	0	0	10	442.1	152.89	12.79	0.0033	0
12000	239.1	134.24	0	0	10	614.9	327.92	11.05	0.0035	0

In our final experiment, we carry out a simulation of the BER performances of our ABFW decoder, and the state of the art decoder BP. In our simulation, given in Figure 14, we test the performance of the decoders with five different error probabilities, i.e., $p = 10^{-1}$, $10^{-1.2}$, $10^{-1.4}$, $10^{-1.6}$, $10^{-1.8}$. In the simulations, we generate LDPC–C codes from a $\mathbf{H}_{base}$ with dimensions $(150, 300)$, and without length–4 cycles in its Tanner graph (see Section 3.2). We randomly generate a received vector and implement a decoder (ABFW or BP) until we observe 50 window decodings different than the original codeword. We perform the simulation for ABFW decoder with "Small" and "Large" window sizes. We pick the window size for BP decoder as $w = 6m + 1$. As an example, when $p = 10^{-1.4}$, BER of "ABFW Large" decoder is less than 10^{-4} as plotted in Figure 14.

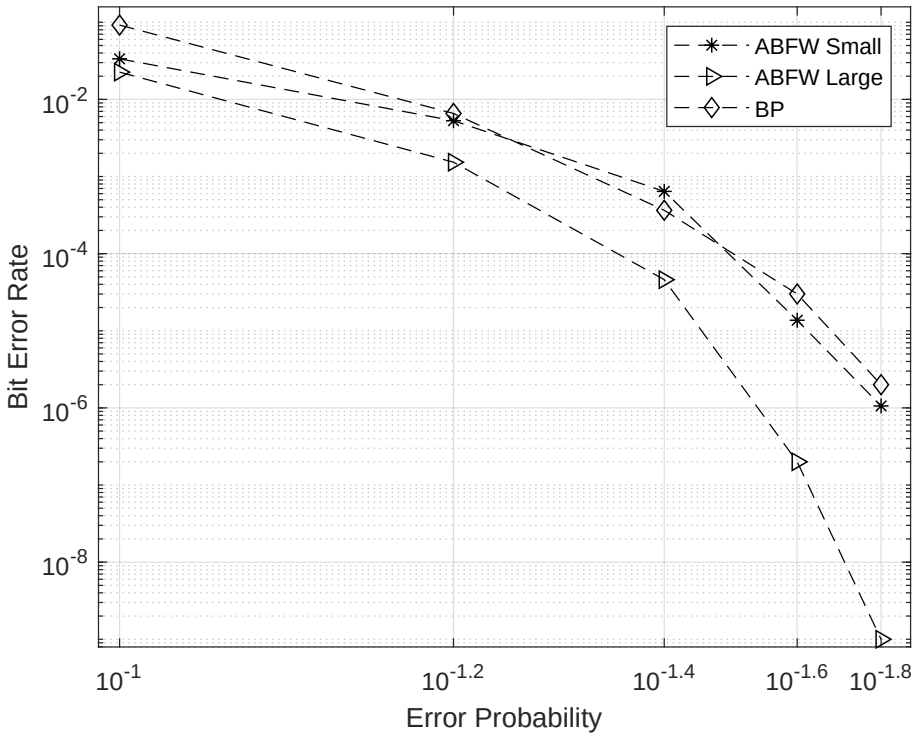

Figure 14: BER simulations of ABFW and BP

In Figure 14, we observe that increasing the window size improves BER of ABFW decoder by eliminating error accumulation (see Figure 13). "ABFW Small" and BP decoders perform closer BER values especially for $p \leq 10^{-1.2}$. As the error probability decreases, the BER performance of "ABFW Large" decoder improves significantly, which indicates there is no lower bound on its BER. We observe that BP can correct the errors as well as "ABFW Small" decoder, whereas "ABFW Large" outperforms BP in terms of BER at the expense of decoding time.

These results indicate that ABFW with window size $w > \frac{3m}{2}$ is a strong candidate for the decoding problem in communication systems. BP algorithm can quickly obtain vectors, which are not always feasible. On the other hand, ABFW generates near optimal codewords in acceptable amount of time by finding feasible codewords at each window. BP is practical for TV broadcasting and video streams, since fast decoding is crucial for these applications. However, 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.

5 Conclusions

We proposed optimization–based sliding window decoders for terminated LDPC–C 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 LDPC–C codes and introduce LDPC 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 at 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 LDPC–C codes where all windows are the 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.

Belief Propagation (BP) is popular in practical applications. Compared with ABFW approach, BP gives 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.

Acknowledgements

This research has been supported by the Turkish Scientific and Technological Research Council with grant no 113M499.

References

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

[2] R. G., Gallager, "Low-density parity-check codes," *IRE Trans. on Information Theory*, vol. 8, no. 1, pp. 21–28, January 1962.

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

[4] J. P., van den Berg, and A. J. R. M., Gademann, "Optimal routing in an automated storage/retrieval system with dedicated storage," *IIE Transactions*, vol 31, pp. 407–415, 1999.

[5] V., Guruswami, "Iterative decoding of low–density parity–check codes", *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, Issue 90, October 2006.

[6] F. R., Kschischang, B. J. Frey, and H. A., Loeliger, "Factor graphs and the sum–product algorithm," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 498–519, February 2001.

[7] J. S., Yedidia, W. T., Freeman, and Y., Weiss, "Understanding belief propagation and its generalizations," Mitsubishi Electric Res. Labs, Cambridge, Tech. Rep., TR2001-22, November 2001.

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

[9] E. R., Berlekamp, 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.

[10] J., Feldman, and D. R., Karger, "Decoding turbo-like codes via linear programming," *J. Computer and System Sciences*, vol 68, pp. 733-752, 2004.

[11] J., Feldman, M. J., Wainwright, and D. R., Karger, "Using linear programming to decode binary linear codes," *IEEE Trans. on Information Theory*, vol 51, no. 3, pp. 954–972, March 2005.

[12] P., Elias, "Coding for Noisy Channels", MIT Res. Lab. of Electronics, Cambridge, MA, IRE Convention Rec., Part 4, pp. 37–46, 1955.

[13] A. J., Viterbi, "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.

[14] P., Brice, 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.

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

[16] K., Sh. Zigangirov, "Some sequential decoding procedures," *Probl. Peredachi Inf.*, 2, pp. 13–25, 1966.

[17] F., Jelinek, "A fast sequential decoding algorithm using a stack," *IBM J. Res. and Dev.*, 13, pp. 675–685, November 1969.

[18] Y. S., Han, and P.-N., Chen, "Sequential decoding of convolutional codes" in *Wiley Encyclopedia of Telecommunications*, Wiley, 2003.

[19] A., Jimenez-Feltström 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.

[20] I. E. Bocharova, 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.

[21] T. K., Moon, *Error correction coding: mathematical methods and algorithms.* New Jersey: Wiley, 2005.

[22] I., Ben-Gal, Y. T., Herer, and T., Raz, "Self-correcting inspection procedure under inspection errors", *IIE Transactions*, vol. 34, pp. 529–540, 2002.

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

[24] S., Kudekar, 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.

[25] A. B., Keha, and T. M., Duman, "Minimum distance computation of LDPC codes using branch and cut algorithm," *IEEE Trans. on Communications*, vol 58, no. 4, pp. 1072–1079, 2010.

[26] B., Kabakulak, Z. C., Taşkın, A. E., Pusane, "A branch-and-cut algorithm to design LDPC codes without small cycles in communication systems," *http://arxiv.org/abs/1709.09936*, 2017.

[27] M., Lentmaier, 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.

[28] D. J., Costello, Jr., 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.

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

[30] N. K., Freeman,J., Mittenthal, and S. H., Melouk, "Parallel-machine scheduling to minimize overtime and waste costs," *IIE Transactions*, vol 46, pp. 601–618, 2014.

[31] B. M. J., Leiner, "LDPC codes - a brief tutorial," *Wien Technical University*, 2005.

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

[33] A., Sarıduman, 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.

[34] M., Battaglioni, M., Baldi, E., Paolini, "Complexity-constrained spatially coupled LDPC codes based on protographs," *Proc. International Symposium on Wireless Communication Systems*, Bologna, Italy, 2017.