NEW TECHNIQUE FOR IMPROVING PERFORMANCE OF LDPC CODES IN THE PRESENCE OF TRAPPING SETS

Esa Alghonaim, Aiman El-Maleh, M. Adnan Landolsi
esa.alg@gmail.com, *aimane@kfupm.edu.sa*, *andalusi@kfupm.edu.sa*

King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia

Trapping sets are considered the primary factor for degrading the performance of low density parity check (LDPC) codes in the error-floor region. The effect of trapping sets on the performance of an LDPC code becomes worse as the code size decreases. One approach to tackle this problem is to minimize trapping sets during LDPC code design. However, while trapping sets can be reduced, their complete elimination is infeasible due to the presence of cycles in the underlying LDPC code bipartite graph. In this work, we introduce a new technique based on *trapping sets neutralization* to minimize the negative effect of trapping sets under Belief Propagation (BP) decoding. Simulation results for random, progressive edge growth (PEG) and MacKay LDPC codes demonstrate the effectiveness of the proposed technique. The hardware cost of the proposed technique is also shown to be minimal.

Keywords and phrases: LDPC codes, Trapping sets, FER performance, Belief Propagation iterative decoding, Error floor.

1. INTRODUCTION

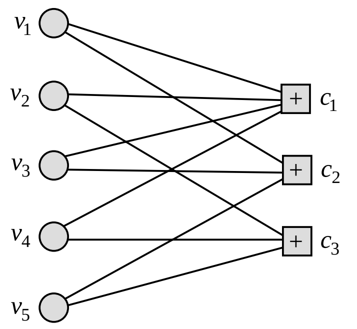
Forward Error Correcting (FEC) codes are an essential component of modern state-of-the-art digital communication and storage systems. Indeed, in many of the recently developed standards, FEC codes play a crucial role for improving the error performance capability of digital transmission over noisy and interference-impaired communication channels.

Low Density Parity Check codes (LDPCs), originally introduced in [1], have recently been undergoing a lot of active research, and are now widely considered to be one of the leading families of FEC codes. LDPC codes demonstrate performance very close to the information-theoretic bounds predicted by Shannon theory, while at the same time having the distinct advantage of low-complexity, near-optimal iterative decoding.

As with other types of codes decoded by iterative decoding algorithms (such as turbo codes), LDPC codes can suffer from the presence of undesirable error floors at increasing SNR levels (although these are found to be relatively lower than the error floors encountered with turbo codes [18]). In the case of LDPC codes, trapping sets [7, 10, 18] have been identified as one of the main factors causing error floors at high SNR values. The analysis of trapping sets and their impact on LDPC codes has been addressed in [3-8]. The main approaches for mitigating the impact of trapping sets on LDPC codes are based on either introducing algorithms to minimize their presence during code design as in [3, 5, 8] or by enhancing decoder performance in the presence of trapping sets as in [4, 6, 7]. The main disadvantage of the first approach, in addition to putting tight constraints on code design, is that trapping sets cannot be totally eliminated at the end due to the "unavoidable" existence of cycles in their underlying bi-partite Tanner graphs especially for relatively short block length codes (which is the focus of this work). In addition, LDPC codes designed to reduce trapping sets may result in large interconnect complexity increasing hardware implementation overhead. The second approach is therefore considered to be more applicable for our purpose, and is the basis of the contributions presented in this paper.

In order to enhance decoder performance in the presence of (unavoidable) trapping sets, an algorithm is introduced in [7] based on flipping the hard decoded bits in trapping sets. First, trapping sets are identified and stored in a look-up table based on BP decoding simulation. Whenever the decoder fails, the decoder uses the look-up table based on the unsatisfied parity checks to determine if a pre-known failure is detected. If a match occurs, the decoder simply flips the hard decision values of trapping bits. This approach suffers from the following disadvantages: (1) the decoder has to exactly specify the trapping sets variable nodes in order to flip them; (2) extra time is needed to search the lookup table for a trapping set; (3) the technique is not amenable to practical hardware implementation.

In [4, 6], the concept of averaging partial results is used to overcome the negative effect of trapping sets in the error floor region. Variable node messages update in the conventional BP decoder are modified in order to make it less sensitive to oscillations in messages received from check nodes. The variable node equation is modified to be the average of current and previous signals values received from check nodes. While this approach is effective in handling oscillating error



$$H = \begin{array}{c|ccccc|c} & v_1 & v_2 & v_3 & v_4 & v_5 & \\ & 1 & 1 & 1 & 1 & 0 & c_1 \\ & 1 & 0 & 1 & 0 & 1 & c_2 \\ & 0 & 1 & 0 & 1 & 1 & c_3 \end{array}$$

Figure 1: The two representations of LDPC codes: graph form and matrix form.

patterns, it does not improve decoder performance in the case of constant error patterns.

In this paper, we propose a novel approach for enhancing decoder performance in presence of trapping sets by introducing a new concept called *trapping sets neutralization*. The effect of a trapping set can be eliminated by setting its variable nodes intrinsic and extrinsic values to zero, i.e. *neutralizing* them. After a trapping set is neutralized, the estimated values of variable nodes are affected only by external messages from nodes outside the trapping set.
Most harmful trapping sets are identified by means of simulation. To be able to neutralize identified trapping sets, a simple algorithm is introduced to store trapping sets configuration information in variable and check nodes.
The remainder of this paper is organized as follows: In Section 2, we give an overview of LDPC codes and BP algorithm. Trapping sets identification and neutralization are introduced in Section 3. Section 4 presents the algorithm of trapping sets neutralization based on learning. Experimental results are given in Section 5. In Section 6, we conclude the paper.

2. OVERVIEW OF LDPC CODES

LDPC codes are a class of linear block codes that use a sparse, random-like parity-check matrix $\boldsymbol{H}$ [1, 2]. An LDPC code defined by the parity check matrix $\boldsymbol{H}$ represents the parity equations in a linear form, where any given codeword $\boldsymbol{u}$ satisfies the set of parity equations such that $\boldsymbol{u} \times \boldsymbol{H} = \boldsymbol{0}$. Each column in the matrix represents a codeword bit while each row represents a parity check equation.
LDPC codes can also be represented by bipartite graphs, usually called Tanner graphs, having two types of nodes: *variable nodes* and *check nodes*, interconnected by edges whenever a given information bit appears in the parity

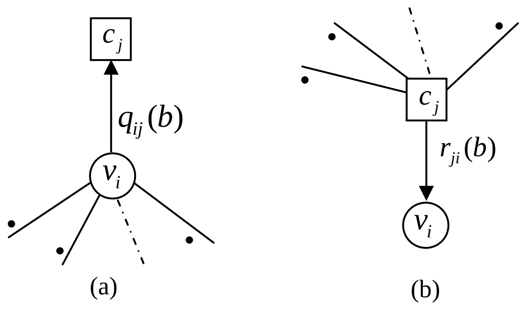


FIGURE 2: (a) Variable-to-check message, (b) Check-to-variable message.

check equation of the corresponding check bit, as shown in Figure 1.
The properties for an (N,K) LDPC code specified by an $M \times N$ $\boldsymbol{H}$ matrix can be summarized as follows:

- *Block size*: number of columns (N) in the $\boldsymbol{H}$ matrix.
- Number of information bits: given by $K = N - M$.
- Rate: the rate of the information bits to the block size. It equals to $1 - \frac{M}{N}$, given that there are no linear dependent rows in the $\boldsymbol{H}$ matrix.
- *Check node degree*: number of 1's in the corresponding row in the $\boldsymbol{H}$ matrix. Degree of a check node c_j is referred as $d(c_j)$.
- *Variable node degree*: number of 1's in the corresponding column in the $\boldsymbol{H}$ matrix. Degree of a variable node v_i is referred as $d(v_i)$.
- *Regularity*: An LDPC code is said to be regular if $d(v_i) = p$ for $1 \le i \le N$ and $d(c_j) = q$ for $1 \le j \le M$. In this case, the code is (p,q) regular LDPC code. Otherwise, the code is considered irregular.
- *Code girth*: the minimum cycle length in the Tanner graph of the code.

The iterative message-passing belief propagation algorithm (BP) [1, 2] is commonly used for decoding LDPC codes, and is shown to achieve optimum performance when the underlying code graph is cycle-free. In the following, a brief summary of the BP algorithm is given. Following the notation and terminology used in [16], we define the following:

- u_i : Transmitted bit in a codeword, $u_i \in \{0,1\}$.
- x_i : A transmitted channel symbol, with a value given by: $x_i = \begin{cases} +1 & when \quad u_i = 0 \\ -1 & when \quad u_i = 1 \end{cases}$.
- y_i : A received channel symbol, $y_i = x_i + n_i$, where n_i is zero-mean Additive White Gaussian Noise (AWGN) random variable with variance σ^2.
- For the j^{th} row in an $\boldsymbol{H}$ matrix, the set of column locations having 1's is given by $R_j = \{i : h_{ji} = 1\}$. The

set of column locations having 1's, excluding location i is given by $R_{j\backslash i} = \{i' : h_{ji'} = 1\} \setminus \{i\}$.

- For the i^{th} column in an $\boldsymbol{H}$ matrix, the set of row locations having 1's is given by $C_i = \{j : h_{ji} = 1\}$. The set of row locations having 1's, excluding the location j is given by $C_{i\backslash j} = \{j' : h_{j'i} = 1\} \setminus \{j\}$
- $q_{ij}(b)$: Message (extrinsic information) to be passed from variable node v_i to check node c_j regarding the probability of $u_i = b$, $b \in \{0,1\}$, as shown in Figure 2(a). It equals the probability that $u_i = b$ given extrinsic information from all check nodes, except node c_j.
- $r_{ji}(b)$: Message to be passed from check node c_j to variable node v_i, which is the probability that the j^{th} check equation is satisfied given that bit $u_i = b$ and the other bits have separable (independent) distribution given by $\{q_{ij'}\}_{j' \neq j}$, as shown in Figure 2(b).
- $Q_i(b)$ = the probability that $u_i = b$, $b \in \{0,1\}$
- $L(u_i) \equiv \log \dfrac{\Pr(x_i = +1 \mid y_i)}{\Pr(x_i = -1 \mid y_i)} = \log \dfrac{\Pr(u_i = 0 \mid y_i)}{\Pr(u_i = 1 \mid y_i)}$, $L(u_i)$ is usually referred to as the intrinsic information for node v_i.
- $L(r_{ji}) \equiv \log \dfrac{r_{ji}(0)}{r_{ji}(1)}$ and $L(q_{ij}) \equiv \log \dfrac{q_{ij}(0)}{q_{ij}(1)}$
- $L(Q_i) \equiv \log \dfrac{Q_i(0)}{Q_i(1)}$

The BP algorithm involves one initialization step and three iterative steps as shown below:

Initialization step: Set the initial value of each variable node signal as follows: $L(q_{ij}) \equiv L(u_i) = 2y_i / \sigma^2$, where σ^2 is the variance of noise in the AWGN channel.

Iterative steps: The three iterative steps are as follows:

(I) Update check nodes as follows:

$$L(r_{ji}) = \left(\prod_{i' \in R_{j\backslash i}} \alpha_{i'j} \right) \times \phi \left(\sum_{i' \in R_{j\backslash i}} \phi(\beta_{i'j}) \right) \tag{3}$$

Where $\alpha_{i'j} = sign(L(q_{ij}))$, $\beta_{ij} = \left| L(q_{ij}) \right|$

$$\phi(x) = -\log(\tanh(x/2)) = \log \frac{e^x + 1}{e^x - 1}$$

(II) Update variable nodes as follows:

$$L(q_{ij}) = L(u_i) + \sum_{j' \in C_{i\backslash j}} L(r_{j'i}) \tag{4}$$

(III) Compute estimated variable nodes as follows:

$$L(Q_i) = L(u_i) + \sum_{j \in C_i} L(r_{ji}) \tag{5}$$

Based on $L(Q_i)$, the estimated value of the received bit ($\hat{u}_i$) is given by:

$$\hat{u}_i = \begin{cases} 1 & if \quad L(Q_i) < 0 \\ 0 & else \end{cases} \tag{6}$$

During LDPC decoding, the iterative steps I to III are repeated until one of the following two events occurs:
- The estimated vector $\hat{\boldsymbol{u}} = (\hat{u}_1, ..., \hat{u}_n)$ satisfies the check equations, i.e. $\hat{\boldsymbol{u}} \cdot \boldsymbol{H} = \boldsymbol{0}$.
- Maximum iterations number is reached.

3. TRAPPING SETS

In BP decoding of LDPC codes, dominant decoding failures are, in general, caused by a combination of multiple cycles [10]. In [18], the combination of error bits that leads to a decoder failure are defined as trapping sets. In [7], it is shown that the dominant trapping sets are formed by a combination of short cycles present in the bipartite graph.

In the following, we adopt the terminology and notation related to trapping sets as originally introduced in [6]. Let $\boldsymbol{H}$ be the parity check matrix of (N, K) LDPC code, and let $G(\boldsymbol{H})$ denote its corresponding Tanner graph.

Definition: A (z, w) trapping set T is a set of z variable nodes, for which the subgraph of the z variable nodes and the check nodes that are directly connected to them contains exactly w odd-degree check nodes.

The next example illustrates the behavior of trapping sets and how they are harmful.

Example 1: Consider a regular (N,K) LDPC code with degree (3,6). Figure 3 shows a trapping set T(4,2) in the code graph. Assume that an all-zero codeword ($\boldsymbol{u} = 0$) is sent through an AWGN channel and all bits are received correctly (i.e. have positive intrinsic values) except the 4 bits in the trapping set T(4,2), i.e. $L(u_i)$ <0 for $1 \le i \le 4$ and $L(u_i)$ >0 for $4 < i \le N$. (Assume logic 0 is encoded as +1, while logic 1 is encoded as -1).

Based on equation (5), the estimated value of a variable node is the sum of its intrinsic information and messages received from the neighboring three check nodes. Therefore, the estimation equation for each variable node contains four summation terms: the intrinsic information and three information messages. In this case, the estimated values for v_1 (and v_3) will be incorrect because all of the four summation terms of its estimation equation are negative. For v_2 (and v_4), three out of the four summation terms in its estimation equation have negative values. Therefore, v_2 (and v_4) has high probability to be incorrectly estimated. In this case, the decoder becomes in *trap* and will continue in the trap unless positive

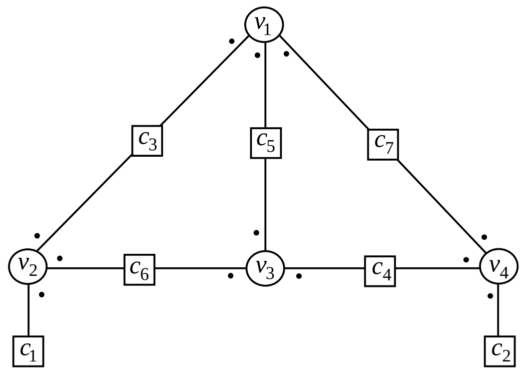


FIGURE 3: Trapping set example of T(4,2)

signals from c_1 and/or c_2 are strong enough to change the polarities of the estimated values of v_2 and/or v_4. This example illustrates a trapping set causing a *constant* error pattern.

As a first step to investigate the effect of trapping sets on LDPC codes performance, extensive simulations for LDPC codes over AWGN channels with various SNR values have been performed. A frame is considered to be in error if the maximum decoding iteration is reached without satisfying the check equations, i.e. the syndrome $\hat{\boldsymbol{u}} \times \boldsymbol{H}$ is non-zero. Error frames are classified based on observing the behavior of the LDPC decoder at each decoding iteration. At the end of each iteration, bits in error are counted. Based on this, error frames are classified into three patterns, described as follows:

I- *Constant error pattern*: where the bit error count becomes constant after only a few decoding iterations.

II- *Oscillating error pattern*: where the bit error count follows a nearly periodic change between maximum and minimum values. An important feature of this error pattern is the high variation in bit error count as a function of decoding iteration number.

III-*Random-like error pattern:* where the bit error count evolution follows a random shape, characterized by low variation range.

Figure 4 shows one example for each of the three error patterns. In a constant error pattern, bit errors count becomes constant after several decoding iterations (10 iterations in the example of Figure 4). In this case, the decoder becomes stuck due to the presence of a tapping set $T(z, w)$ and the number of bits in error equals to z and all check nodes are satisfied except w check nodes.

The major difference between a trapping set $T(z, w)$ causing a constant error pattern and a trapping set $T(e, f)$ causing other patterns is the number of odd-degree check nodes. Based on extensive simulations, it is found that $w \leq f$. This result is interpreted logically as follows: if variable nodes of a trapping set are in error, only odd-degree check nodes are sending correct messages to the

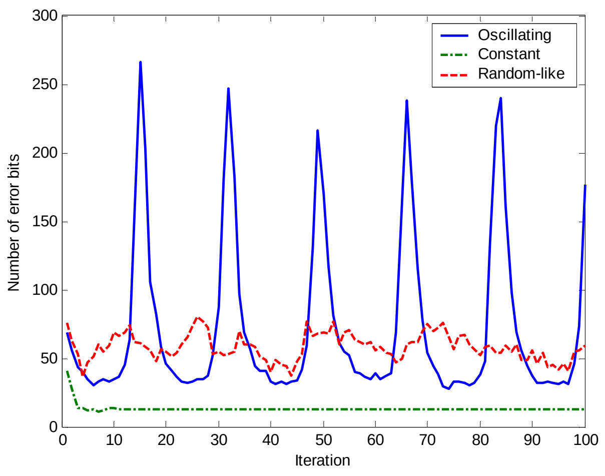


FIGURE 4. Illustration of the three types of error patterns

variable nodes of the trapping set. Therefore, as the number of odd-degree check nodes decreases, the probability of breaking the trap decreases. As an extreme example, a trapping set with no odd-degree check nodes results in a decoder convergence to a codeword other than the transmitted one and thus causes undetected decoder failure.

Table 1 shows examples of percentages of the three error patterns for three LDPC codes based on simulating the codes at error-floor regions. The first LDPC code, HE(1024,512) [15], is constructed to be interconnect efficient for fully parallel hardware implementation. The RND(1024,512) LDPC code is randomly constructed avoiding cycles of size 4. The PEG(100,50) LDPC code is constructed using PEG algorithm [5], which maximizes the size of cycles in the code graph. From Table 1, it is evident that constant error patterns are significant in some LDPC codes including short length codes.

This observation motivates the need for developing a technique for enhancing decoder performance due to trapping sets of constant error patterns type. For trapping sets that cause constant error patterns, when a trap occurs, values of check equations do not change in subsequent iterations. Thus, a decoder trap is detected based on check equations results. The unsatisfied check nodes are used to reach to the trapping set variable nodes.

3.1. BP decoder trapping sets detection

In order to eliminate the effect of trapping sets during the iterations of BP decoder, a mechanism is needed to detect the presence of a trapping set. The proposed trapping sets detection technique is based on monitoring the state

TABLE1: Percentages of error patterns at error-floor region

Code Size	**constant**	**oscillating**	**random-like**
HE(1024,512)	59%	38%	3%
RND(1024,512)	95%	4%	1%
PEG(100,50)	90%	5%	5%

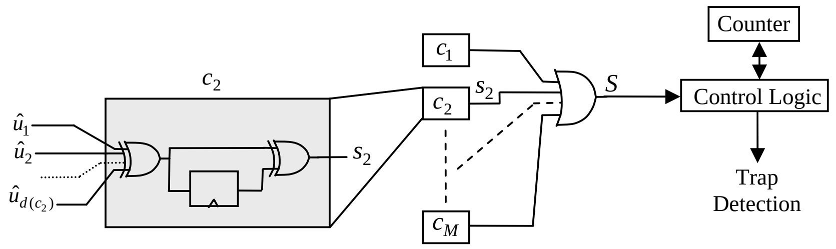


FIGURE 5: Decoder trap detection circuit

of the check equations vector $\hat{\boldsymbol{u}} \times \boldsymbol{H}$. At the end of each decoding iteration, a new value of $\hat{\boldsymbol{u}} \times \boldsymbol{H}$ is computed. If the $\hat{\boldsymbol{u}} \times \boldsymbol{H}$ value is non-zero and remains unchanged (stable) for a pre-determined number of iterations, then a decoder trap is detected. We call this number *the stability parameter* (*d*) and it is normally set to a small value. Based on experimental results, it is found that *d*=3 is a good choice. The implementation of trap detection is similar to the implementation of valid codeword detection with some extra logic in each check node. Figure 5 shows an implementation of trapping sets detection for a decoder with *M* check nodes. The output s_i for a check node c_i is logic zero if the check equation result is equivalent to the check equation result in the previous iteration, i.e. no change in the check equation result. The output *S* is zero if there is no change in all check equations between the current and the previous iteration number.

3.2. Trapping sets neutralization

In this section, we introduce a new technique to overcome the detrimental effect of trapping sets during BP decoding. To overcome the negative impact of a trapping set *T*(*z*,w), the basic idea is to *neutralize* the *z* variable nodes in the trapping set. Neutralizing a variable node involves setting its intrinsic value and extrinsic message values to zero. Specifically, neutralizing a variable node v_i involves the following two steps:

(1) $L(u_i)=0$, (2) $L(q_{ij})=0$, $1 \le j \le d(v_i)$.

The neutralization concept is illustrated by the following example.

Example 2: For the trapping set *T*(4,2) in Example 1, it has been shown that when all code bits are received correctly except *T*(4,2) bits, the decoder fails to correct the codeword resulting in an error pattern of constant type.

Now, consider neutralizing the trapping set variable nodes by setting its intrinsic and extrinsic values to *zero*. After neutralization, the decoder converges to a valid codeword within two iterations, as follows: In the first iteration after neutralization, for v_2 and v_4, two extrinsic messages become positive due to positive messages from nodes c_1 and c_2, which shifts estimated values of v_2 and v_4 to the positive correct values. For nodes v_1 and v_3, all extrinsic values are zeros and their estimated values remain zero. In the second iteration after neutralization, for v_1 and v_3, two extrinsic messages become positive due to positive extrinsic messages from nodes v_2 and v_4, which shifts estimated values of v_1 and v_3 to the positive correct values.

The proposed neutralization technique has three important characteristics: (1) It is not necessary to exactly determine the variable nodes in a trapping set, such as the trapping set bits flipping technique used in [7]. In the previous example, if only 3 out of the 4 trapping sets variables are neutralized, the decoder will still be able to recover from the trap. (2) If some nodes outside a trapping set are neutralized (due to inexact identification of the trapping set), their extrinsic messages are expected to quickly recover their estimation function to correct values due to correct messages from neighbouring nodes. This is because most of the extrinsic messages are correct in the error-floor regions. (3) Neutralization is performed during BP decoding iterations as soon as a trapping set is detected, which makes the decoder able to converge to a valid codeword within the allowed maximum number of iterations.

As an example, for the near-constant error pattern in Figure 4, a trap occurs at iteration 10 and is detected at iteration 13 (assuming *d*=3). In this case, the decoder has a plenty of time to neutralize the trapping set before reaching the maximum 100 iterations. In general, based on our simulations, a decoder trap is detected during early decoding iterations.

4. BP DECODER WITH TRAPPING SETS NEUTRALIZATION BASED ON LEARNING

In this section, we introduce an algorithm to correct constant error pattern types (causing error floors) associated with LDPC BP decoding. The proposed algorithm involves two parts: (1) a pre-processing phase called: *learning* phase and (2) actual decoding phase. The learning phase is an off-line computation process in which trapping sets are identified. Then, variable and check nodes are configured according to the identified trapping sets. In the actual decoding phase, the proposed decoder runs as a standard BP decoder with the ability to detect and neutralize trapping sets using variable and check nodes configuration information obtained during the learning phase. When a trapping set is detected, the decoder stops running BP iterations and switches to a neutralization process, in which the detected trapping set is neutralized. Upon completion of the neutralization process, the decoder resumes to normal running of BP iterations. The neutralization process involves forwarding messages between the trapping sets check and variable nodes.

Before proceeding with the details of the proposed decoder, we give an example on how variable and check nodes are configured during the learning phase and how this configuration is used to neutralize a trapping set during actual decoding.

Example 3: Given the trapping set $T(4,2)$ of the previous example, we show the following: (a) how the nodes of this trapping set are configured, (b) how the neutralization process is performed during the actual decoding phase.

(a) In the learning phase, the trapping set nodes $\{c_1, c_2, c_3, c_4, c_5, c_6, c_7, v_1, v_2, v_3, v_4\}$ are configured for neutralization. First, a tree is built corresponding to the trapping set starting with odd-degree check nodes as the first level of the tree, as shown in Figure 6. The reason for starting from odd-degree check nodes is because they are the only gates leading to a trapping set when the decoder is in a trap. When the decoder is stuck due to a trapping set, all check nodes are satisfied except the odd-degree check nodes of the trapping set. Therefore, odd-degree check nodes in trapping sets are the keys for the neutralization process.

Degree one check nodes in the trapping set (c_1 and c_2 in this example) are configured to initiate messages to their neighboring variable nodes requesting them to perform neutralization. We call these messages: *neutralization initiation messages*. In Figure 6, arrows pointing out from a node indicate that the node is configured to forward a neutralization message to its neighbor. The task of neutralization message forwarding in a trapping set is to send neutralization message to every variable node in the trapping set. In our example, c_1 and c_2 are configured for neutralization message initiation, while v_2, c_3 and c_6 are configured for neutralization messages forwarding. This configuration is enough to forward neutralization messages to all variable nodes in the trapping set. Another possible configuration is: c_1 and c_2 are configured for neutralization message initiation while v_4, c_4 and c_7 are configured for neutralization messages forwarding. Thus, in general, there is no need to configure all trapping set nodes.

(b) Now, assume that the proposed decoder is running BP iterations and falls in a trap due to $T(4,2)$. Next, we show how the pre-configured nodes are able to neutralize the trapping set $T(4,2)$ in this example. First, the decoder detects a trap event and then it stops running BP iterations and switches to a neutralization process. The decoder runs the neutralization process for a fixed number of cycles and then resumes running the BP

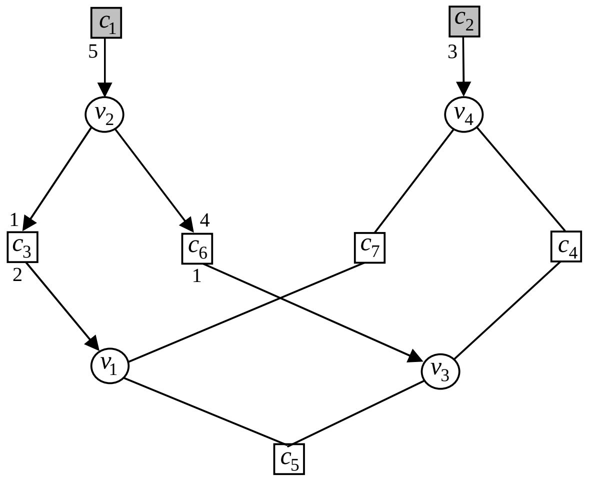


FIGURE 6: Tree structure for the trapping set $T(4,2)$

iterations. In the first cycle of the neutralization process, the unsatisfied check nodes initiate a neutralization message according to the configuration stored in them during the learning phase. Because the decoder failure is due to $T(4,2)$, all check nodes in the decoder are satisfied except the two check nodes c_1 and c_2. Therefore, only c_1 and c_2 initiate neutralization messages to nodes v_2 and v_4, respectively. In the second neutralization cycle, variable nodes v_2 and v_4 receive neutralization messages, perform neutralization and v_2 forwards the neutralization message to c_3 and c_6. In the third neutralization cycle, c_3 and c_6 receive and forward the neutralization messages to v_1 and v_3, respectively, which in turn perform neutralization but do not forward neutralization messages. After that, no message forwarding is possible until neutralization cycles end. After the neutralization process, the decoder resumes running BP iterations. The proposed decoder converges to a valid codeword within two iterations after resuming running BP iterations, as previously shown in Example 2.

Before discussing the neutralization algorithm, a description for the configuration parameters used in variable and check nodes is given followed by an illustrative example. Each variable node v_i is assigned a bit γ_i and each check node c_j is assigned a bit β_j^q and a word α_j^q for each of its links q. The following is a description for these parameters:

γ_i: Message forwarding configuration bit assigned for a variable node v_i. When a variable node v_i receives a neutralization message, it acts as follows: If $\gamma_i = 1$ then v_i forwards the received neutralization message to all neighboring check nodes except the one that sent the message; otherwise it does not forward the received message.

TABLE 2: Nodes configuration for *T*(4,2).

Configuration	Meaning
$\beta_1^5 = 1$	c_1 initiates a message through link 5 (i.e. initiates message to v_2)
$\beta_2^3 = 1$	c_2 initiates a message through link 3 (i.e. initiates message to v_4)
$\gamma_2 = 1$	v_2 forwards incoming messages to all neighbors
$\alpha_3^2 = (000001)_2$	c_3 forwards incoming messages from link 1 to link 2 (i.e. from v_2 to v_1)
$\alpha_6^1 = (001000)_2$	c_6 forwards incoming messages from link 4 to link 1 (i.e. from v_2 to v_3)

β_j^q: Message initiation configuration bit assigned for a link indexed q in a check node c_j, where $1 \le q \le d(c_j)$.

α_j^q: Message forwarding configuration word assigned for a link indexed q in a check node c_j, where $1 \le q \le d(c_j)$. The size of α_j^q in bits equals to $d(c_j)$. If a check node c_j has to forward a neutralization message received at link indexed p through a link indexed q, then α_j^q is configured by setting the bit number p to 1, i.e. the bit $\alpha_j^q(p)$ is set to 1. For example, if a degree 6 check node c_j has to forward a neutralization message received at the link indexed 2 through the link indexed 3, α_j^3 is configured to $(000010)_2$, i.e. $\alpha_j^3(2) = 1$.

The following example illustrates variable and check nodes configuration values for a given trapping set.

Example 4: Assume that the trapping set *T*(4,2) in Figure 6 is identified in a regular (3,6) LDPC code. Check nodes links indices are indicated on the links; for example, in c_1, (c_1, v_2) link has index 5. The configuration for this trapping set is shown in Table 2.

Algorithm-1 lists the proposed trapping set neutralization algorithm. Since the decoder does not know how many cycles are needed to neutralize a trapping set, it performs neutralization and message forwarding cycles for a pre-set number (nt_cycles). For example, two neutralization cycles are needed to neutralize the trapping set shown in Figure 6. The number of neutralization cycles is pre-set during the learning phase to the maximum number of neutralization cycles required for all trapping sets. Based on simulation results, it is found that a small number of neutralization cycles are often required. For example, 5 neutralization cycles are found sufficient to neutralize trapping sets of 20 variable nodes.

Algorithm-1: Trapping sets neutralization algorithm.

```
Inputs: LDPC code,
($\gamma_i$, $\beta_j^q$, $\alpha_j^q$) : nodes configuration,
Result of the check equation in each check node $c_j$,
$nt\_cycles$ : number of neutralization cycles
Output: Some variable nodes are neutralized

1. For each check node $c_j$ with unsatisfied equation do
     for $1 \le q \le d(c_j)$, if $\beta_j^q = 1$ then initiate
     a neutralization message through link $q$
2. $l = 1$    // Current number of neutralization cycle
3. While $l \le nt\_cycles$ do
     For each variable node $v_i$ that received a
     neutralization message do the following:
      - perform node neutralization on $v_i$
      - if $\gamma_i = 1$ then forward the message to all neighbors
     For every check node $c_j$ that received a
     neutralization message through link $p$ do the
     following:
      - for $1 \le q \le d(c_j)$, if the bit $\alpha_j^q(p)$ is set then
          forward the message through link $q$
     $l = l + 1$
```

4.1. Trapping sets learning phase

The trapping sets learning phase involves two steps. First, the trapping sets of a given LDPC code are identified. Then, variable and check nodes are configured based on the identified trapping sets.

4.1.1. Trapping sets identification

Trapping sets can be identified based on two approaches: (1) By performing decoding simulations and observing decoder failures [14], (2) By using graph search methods [7]. The first approach is adopted in this work as it provides information on the frequency of occurrence of each trapping set, considered as its weight. This weight is computed based on how many decoder failures occur due to that trapping set, and is used to measure its negative impact compared to other trapping sets. The priority of configuring nodes for a trapping set is assigned according to its weight; more harmful trapping sets are given higher configuration priority.

Algorithm-2 lists the proposed trapping sets identification algorithm. Decoding simulations of an all-zeros codeword with AWGN are performed until a decoder failure is observed. Then, the received frame y that caused the decoding failure is identified and decoding iterations are redone while observing trap detection indicator. If a trap is not detected, then decoding simulations are continued searching for another decoder failure. However, if a trap is detected, then the trapping

Algorithm-2: Trapping sets identification algorithm.

```
Inputs: LDPC code ,
 $no\_failures$ : Number of processed decoder failures
Output: $TS\_List$

1. $TS\_List = \varnothing$ , $failures = 0$
2. While $failures \le no\_failures$ do
     $\boldsymbol{u} = \boldsymbol{0}$ , $\boldsymbol{x} = \boldsymbol{+1}$ , $\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{n}$ // transmit a codeword
     Decode $y$ using standard BP decoder.
     If $\hat{\boldsymbol{u}} \cdot \boldsymbol{H} = \boldsymbol{0}$ then goto 2 // Valid codeword
     $failures = failures + 1$
     Re-decode $\boldsymbol{y}$ observing trap detection indicator
     If a decoder trap is not detected then goto 2
     $TS$ = List of variable nodes $v_i$ in error ($\hat{u}_i$ =1) and
     unsatisfied check nodes.
     If $TS \in TS\_List$ then increment $TS$ weight
     Else add $TS$ to $TS\_List$ and set its weight to 1
```

set TS is identified as follows. First, the unsatisfied check nodes are considered the odd-degree check nodes in the trapping set TS while the variable nodes with hard decision errors ($\hat{u}_i$ =1) are considered the variable nodes of the trapping set. Finally, if the identified trapping set TS is already in the trapping sets list, TS_List , then its weight is incremented by one; otherwise the identified trapping set is added to the trapping sets list, TS_List , and its weight is set to one.

4.1.2. Nodes configuration

The second step in the trapping sets learning phase is to configure variable and check nodes in order for the decoder to be able to neutralize identified trapping sets during decoding iterations.

Before discussing the configuration algorithm, we discuss the case when two trapping sets have common nodes and its impact on the neutralization process. Then, we propose a solution to overcome this problem. This is illustrated through the following example.

Example 5: Figure 7 shows partial nodes of two trapping sets TS_1 and TS_2 in a regular (3,6) LDPC code. $\{v_1, v_3, v_5\} \in TS_1$ and $\{v_2, v_3, v_4\} \in TS_2$. v_3 is a common node between TS_1 and TS_2 . Configuration values after configuring nodes for TS_1 and TS_2 are as follows:

$\alpha_1^3 = (000011)_2$ (Link 3 in c_1 forwards messages received from link 1 or link 2)

γ_3 =1 (v_3 forward messages to neighbors)

$\alpha_2^2 = (000001)_2$ (Link 2 in c_2 forwards messages received from link 1)

$\alpha_3^2 = (000001)_2$ (Link 2 in c_3 forwards messages received from link 1)

Therefore, when the decoder performs a neutralization process due to TS_1, node v_4 will be neutralized although

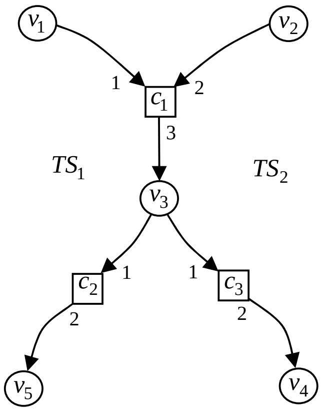


FIGURE 7: Example of common nodes between two trapping sets

it is not a subset of TS_1. Similarly, performing neutralization process due to TS_2 causes node v_5 (which is not a subset of TS_2) to be neutralized. Fortunately, as mentioned in Sec 3.1, when the decoder is in a trap due to a trapping set TS , the decoder converges to a valid codeword even if some variable nodes outside TS have been unnecessarily neutralized. However, based on simulation results, neutralizing a large number of variable nodes other than the desired nodes leads to a decoder failure.

Having introduced the trapping sets common nodes problem, we next show the proposed solution for this problem. Define ω_j for each trapping set TS_j as follows:

ω_j : Ratio of neutralized variable nodes outside the set TS_j to the total number of variable nodes (*N*).

Define T as the maximum allowed value for ω_j . The proposed solution is as follows: after configuring a trapping set TS_k , we compute ω_j for $1 \le j \le k$. If $\omega_j \le T$ for $1 \le j \le k$ then we accept the new configuration, otherwise, TS_k is rejected and the configuration is restored to its state before configuring TS_k .

Algorithm-3 lists nodes configuration algorithm. Initially, configurations of all variable and check nodes are set to zero, step 1. This means that no node is allowed to initiate or forward a neutralization message. Sorting in step 2 is important to give more harmful trapping sets (with greater weight) configuration priority over less harmful trapping sets. Step 4 processes trapping sets in TS_List one by one. For each trapping set TS_k , update nodes configuration by setting nodes configuration parameters (γ_i , β_j^q , α_j^q) related to variable and check nodes in TS_k . Then, for each previously configured trapping set TS_j , $1 \le j \le k$, we

Algorithm-3: Nodes configuration algorithm.

```
Inputs: $TS\_List$, LDPC code of size $(N,K)$
Outputs: $\gamma_i$ , $1 \le i \le N$
         $\beta_j^q$, $\alpha_j^q$, $1 \le j \le N-K$, $1 \le q \le d(c_j)$

1. $\gamma_i = 0$ for $1 \le i \le N$
   $\beta_j^q = 0$ and $\alpha_j^q = 0$, for $1 \le j \le N-K$ and
   $1 \le q \le d(c_j)$
2. Sort $TS\_List$ according to trapping sets weights in a
   descending order.
3. $k = 1$
4. While ( $k \le$ size of $TS\_List$ ) do
   Update configuration so that it includes $TS_k$
   Compute $\omega_j$ for $1 \le j \le k$
   If $\omega_j \le T$ for $1 \le j \le k$ then
       accept configuration update
   Else reject $TS_k$ and reject configuration update
   $k = k+1$
```

compute ω_j. The parameter ω_j for a trapping set TS_j is computed as follows: check equations for all check nodes of the decoder are set as satisfied (i.e. assigned zero values) except odd-degree check nodes in TS_j, and then a neutralization process is performed as in Algorithm-1. The actual number of neutralized variable nodes outside the trapping set variable nodes is divided by *N* (code size) to get ω_j. If the ω_j parameter for all previously configured trapping sets is less than or equal to the threshold *T*, then the new configuration is accepted, otherwise TS_k is rejected (ignored) and nodes configuration is restored to the state before the last update.

4.2. The proposed learning-based decoder

The algorithm of the proposed learning-based decoder is listed in Algorithm-4. The algorithm is similar to the conventional BP decoding algorithm with the addition of trapping sets detection and neutralization. Note that if a trapping set is not detected during decoding iterations, then the proposed algorithm becomes identical to the conventional BP decoder. After each decoding iteration, the trap detection flag is checked, step 6. If a trap is detected, then normal decoding iterations is paused, the decoder performs a neutralization process based on Algorithm-1, the iteration number is increased by the number of neutralization cycles to compensate for the time spent in the neutralization process, and finally the decoder resumes conventional BP iterations. In step 7, before performing a neutralization process, the decoder checks nt_done to make sure that no neutralization process has been performed in the previous iterations. This condition is present to guarantee that the decoder

Algorithm-4: The proposed learning-based decoder.

```
Inputs: LDPC code,
Nodes configuration ( $\gamma_i$ , $\beta_j^q$, $\alpha_j^q$ ),
data received from channel ,
$max\_iter$ :maximum iterations,
$nt\_cycles$ : number of neutralization cycles
Output: decoded codeword

1. $iter = 0$,  $nt\_done = 0$
2. $iter = iter + 1$
3. Run a normal BP decoding iteration.
4. If $\hat{\boldsymbol{u}} \cdot \boldsymbol{H} = \boldsymbol{0}$ then stop    // valid codeword
5. If $iter = max\_iter$ then stop  // decoder failure
6. If decoder trap is not detected then goto step 2
7. If  ( $iter + nt\_cycles < max\_iter$ ) and
   ( $nt\_done$ =0) then do:
   - Perform neutralization // Algorithm-1
   - $iter = iter + nt\_cycles$
   - $nt\_done$ =1
8. Goto step 2
```

will not keep running into the same trap and perform the neutralization process repeatedly. This may happen when a trap is redetected before the decoder is able to get out of it. Upon trap detection and before deciding to perform a neutralization process, the decoder must check another condition. It must ensure that the decoding iterations left before reaching maximum iterations is enough to perform a neutralization process, step 7. For example, consider a decoder with 64 maximum decoding iterations and 5 neutralization cycles. If a trapping set is detected at iteration number 62, the decoder will not have enough time to complete neutralization process.

4.3. Hardware cost

The hardware cost for the proposed algorithm is considered low. For trapping sets storage, we need to assign one bit for each variable node (message forwarding bit). For each check node c_i, we need to assign one bit for message initiating and one word of size $d(c_i)$ for message forwarding. Fortunately, the communication links needed to forward neutralization messages between check and variable nodes of the trapping sets already exist as part of the BP decoding. Therefore, no extra hardware cost is added for the communication between trapping sets nodes. What is needed is a simple control logic to decide to perform message initiation and forwarding based on the stored forwarding information. The decoder trap detection, shown in Figure 5, is implemented as a logic tree similar to the tree of the valid codeword detection implementation. The cost is low, as it mainly consists of a simple logic circuit within the check nodes, in the addition to an OR gate tree combining logic outputs from check nodes. Using a simple multiplexer, valid code word detection logic and trap detection logic can share most of their components. It is worth emphasizing that it is not necessary to store configuration information for all variable and check nodes. Only a subset included in the learned trapping sets are used, which further reduces the required overhead.

5. EXPERIMENTAL RESULTS

In order to demonstrate the effectiveness of the proposed technique, extensive simulations have been performed on several LDPC code types and sizes over BPSK modulated AWGN channel. The maximum number of iterations is set to 64. Due to the required CPU-intensive simulations, especially at high SNR, a parallel computing simulation platform was developed to run the LDPC decoding simulations on 170 nodes on a departmental LAN network [17].

The following is a brief description for the LDPC codes used in the simulation:

- HE(1024,512): A near regular LDPC code of size (1024,512) constructed to be interconnect efficient for fully parallel hardware implementation [15].
- RND(1024,512): Regular (3,6) LDPC code of size (1024,512) randomly generated with the avoidance of cycles of size 4.
- PEG(1024,512): Irregular LDPC code of size (1024,512) generated by PEG algorithm [5]. This algorithm maximizes graph cycles and implicitly minimizes trapping sets of constant type.
- PEG(100,50): Similar to the previous code, but its size is (100,50).
- MacKay(204,102): A regular LDPC code of size (204,102) on MacKay's website [13] labelled as: 204.33.484.txt

In each of the five codes, we compare performance results for the proposed algorithm with conventional BP decoding and the average decoding algorithm proposed in [6]. The average decoding algorithm is a modified version of the BP algorithm in which messages are averaged over several decoding iterations in order to prevent sudden magnitude changes in the values of variable nodes messages. We also add another curve showing the performance of the proposed algorithm on top of average decoding algorithm. Using the proposed algorithm on top of averaging algorithm is identical to the proposed algorithm listed in Algorithm-4, except that in step 3 average decoding algorithm iteration is taking place instead of normal BP decoding iteration. In the learning phase of each LDPC code, we set trapping sets detection parameter (*d*) to 3 and we set the threshold value (*T*) to 10%.

Figure 8 shows the performance results for RND(1024,512). It is evident that the performance of the proposed learning-based algorithm outperforms that of the average decoder in the error-floor region. At low

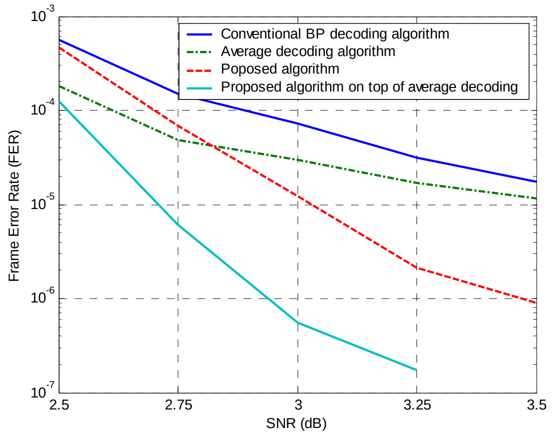


FIGURE 8: Performance results for RND(1024,512) LDPC code

SNR region, average decoding algorithm is better than the proposed algorithm. The reason is due to the few occurrences of constant trapping sets in the low SNR region. As SNR increases, constant error frames increase until they become dominant in error-floor region. The proposed algorithm on top of average decoding shows the best results in all SNR regions. This is because it combines the advantages of the two algorithms: learning-based and average decoding as it improves both constant and non-constant type of patterns.

Figures 9 and 10 show the performance results for the two LDPC codes, PEG(100,50) and PEG(1024,512), respectively. While there is significant improvement for the proposed algorithm in PEG(100,50), there is almost no improvement in PEG(1024,512). The low improvement gain in PEG(1024,512) is due to the low percentage (not more than 8%) of trapping sets that cause constant error patterns. However, it is hard to implement PEG(1024,512) codes using fully-parallel architectures. As can be seen from the PEG code construction algorithm [5], when a new connection is to be added to a variable node, the selected check node for connection is the one in the farthest level of the tree originated from the variable node. This results in interconnections even denser than pure random construction methods.

Figure 11 shows the performance for an interconnect efficient LDPC code, HE(1024,512) [15], that has been implemented in a fully parallel hardware architecture. This LDPC code is designed to have a balance between decoder throughput and error performance. The figure shows that the best performance is obtained using the proposed algorithm on top of the average decoding algorithm. The performance at 3.25B is not drawn due to the excessive simulation time needed at this point.
Based on the results of all simulated codes, it is clearly demonstrated that the application of the proposed algorithm on top of average decoding achieves significant performance improvements in comparison with conventional LDPC decoding. In particular, one can observe that performance improvements are highlighted

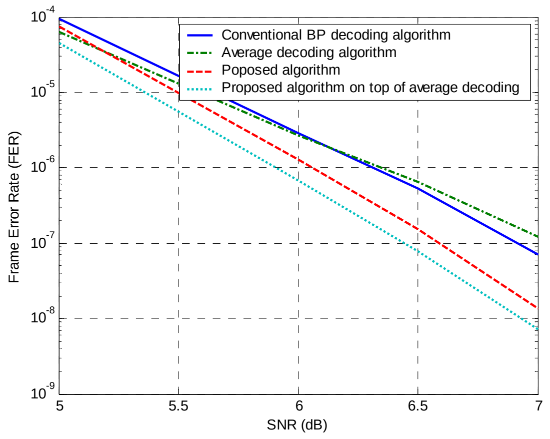


FIGURE 9: Performance results for PEG(100,50) LDPC code

for LDPC codes with relatively low performance using conventional LDPC decoder. This allows LDPC code design techniques to relax some of the design constraints and focus on reducing hardware complexity such as creating interconnect-efficient codes.

Table 3 lists part of the trapping sets that are identified during the learning phase of the HE(1024,512) LDPC code. The complete number of identified trapping sets is 55. One may note that trapping sets with the highest weights have small number of variable and odd-degree check nodes. Table 4 shows the number of identified trapping sets and percentage of check and variable nodes configured to perform neutralization messages forwarding. It is clear that only a subset of the variable and check nodes are configured, which further decreases hardware cost.

6. CONCLUSION

In this paper, we have introduced a new technique to enhance the performance of LDPC decoders especially in

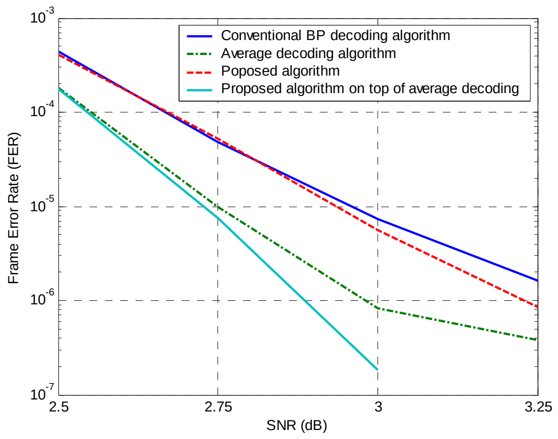


FIGURE 11: Performance results for HE(1024,512) LDPC code

the error floor regions. This technique is based on identifying trapping sets of constant error pattern and reducing their negative impact by neutralizing them. The proposed technique, in addition to enhancing performance, has simple hardware architecture with reasonable overhead. Based on extensive simulations on different LDPC code designs and sizes, it is shown that the proposed technique achieves significant performance improvements for: (1) short LDPC codes, (2) LDPC codes designed under additional constraints such as interconnect-efficient codes. It is also demonstrated that the application of the proposed technique on top of average decoding achieves significant performance improvements over conventional LDPC decoding for all of the investigated codes. This makes LDPC codes even more attractive for adoption in various applications and enables the design of codes that optimize hardware implementation without compromising the required performance.

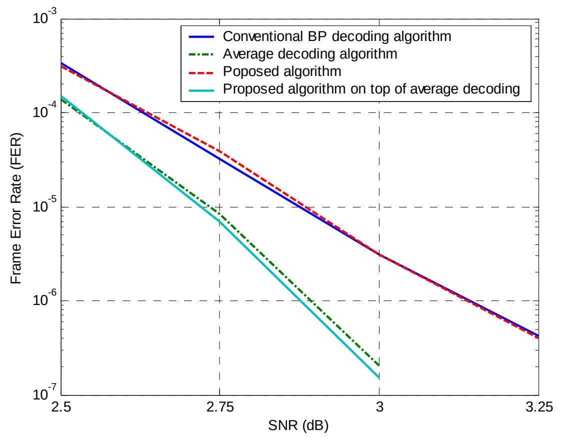


FIGURE 10: Performance results for PEG(1024,512) LDPC code

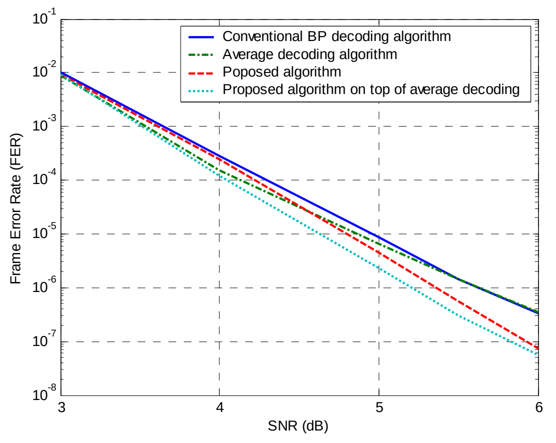


FIGURE 12: Performance results for MacKay(204,102) LDPC code

TABLE 3: Results after the learning phase of HE(1024,512) LDPC code

i	TS_i size	TS_i weight	ω_j
1	(8,2)	106	0%
2	(8,2)	49	0%
3	(12,2)	13	0%
4	(10,3)	9	1%
5	(8,3)	8	2%
6	(10,2)	7	0%
7	(7,3)	5	2%
8	(7,3)	5	3%
9	(7,3)	4	0%
10	(15,2)	3	0%

TABLE 4: Identified trapping sets and configuration percentages for different LDPC codes

CODE	#TS	%V	%C
HE(1024,512)	55	27.15%	13.46%
RND(1024,512)	50	18.46%	9.9%
PEG(1024,512)	8	6.74%	3.42%
PEG(100,50)	57	60%	31.67%
MacKay(204,102)	40	50%	27.94%

ACKNOWLEDGMENT

The authors would like to thank King Fahd University of Petroleum & Minerals for supporting this work under Project no. IN070376.

REFERENCES

[1] R. G. Gallager, “Low Density Parity-Check Codes”. MIT Press, Cambridge, MA, 1963.

[2] D.J.C. MacKay, “Good error-correcting codes based on very sparse matrices,” *IEEE Trans. Inform. Theory*, vol.45, pp.399-431, March 1999.

[3] Tao Tian, Christopher R. Jones, John D. Villasenor, and Richard D. Wesel, “Selective Avoidance of Cycles in Irregular LDPC Code Construction,” *IEEE Transactions on Communications* , Volume 52, Issue 8, Aug. 2004 Page(s):1242 - 1247

[4] S. Gounai, T. Ohtsuki, and T. Kaneko, ”Modified Belief Propagation Decoding Algorithm for Low-Density Parity Check Code Based on Oscillation,” *in Proc. IEEE Vehic. Tech. Conf. Spring 2006*. Vol. 3, pp. 1467-1471.

[5] X.-Y. Hu, E. Eleftheriou, and D.-M. Arnold, “Progressive edge-growth Tanner graphs,” in *Proc. IEEE GLOBECOM 2001*, San Antonio, TX, Nov. 2001, pp. 995–1001.

[6] Stefan Landner and Olgica Milenkovic, “Algorithmic and combinatorial analysis of trapping sets in structured LDPC codes”, *International Conference on Wireless Networks, 2005, Communications and Mobile Computing,* Volume: 1, page(s): 630-635 vol.1

[7] Cavus, E.; Daneshrad, B, “A performance improvement and error floor avoidance technique for belief propagation decoding of LDPC codes,” *IEEE 16th International Symposium on Personal, Indoor and Mobile Radio Communications*, 2005.Volume 4, 11-14 Sept. 2005 Page(s):2386 - 2390 Vol. 4

[8] G. Richter and A. Hof, “On a Construction Method of Irregular LDPC Codes Without Small Stopping Sets,”, *IEEE International Conference on Communications*, Volume 3, June 2006 Page(s):1119 – 1124.

[9] William Ryan, “A Low-Density Parity-Check Code Tutorial, Part II - The Iterative Decoder”, ECE dept. The University of Arizona, April 2002.

[10] T. Tian, C. Jones, J. D. Villasenor, and R. D. Wesel, “Construction of irregular LDPC codes with low error floors," *Proc. IEEE International Conference on Communications*, vol. 5, pp. 3125-3129, May 2003.

[11] Aiman El-Maleh, Basil Arkasosy, M. Adnan Al-Andalusi, “Interconnect-Efficient LDPC Code Design,” *18th IEEE Int. Conf. on Microlelectronics*, pp. 127-130 , Dec. 2006.

[12] IEEE 802.16e. Air interface for fixed and mobile broadband wirelessaccess systems. IEEE P802.16e/D12 Draft, Oct 2005.

[13] D.C. Mackay codes, http://www.inference.phy.cam.ac.uk/mackay/codes.

[14] T. Richardson, "Error Floors of LDPC Codes," *Proc. 41th Annuial Allerton Conf: on Communication, Computing and Control*, Monticello, IL, Oct 2003.

[15] M. Mohiyuddin, A. Prakash, A. Aziz and W. Wolf, “Synthesizing Interconnect Efficient Low Density Parity Check Codes,” *Design Automation Conf.*, pp. 488-491, 2004.

[16] William Ryan, “A Low-Density Parity-Check Code Tutorial, Part II - The Iterative Decoder”, ECE dept. The University of Arizona, April 2002.

[17] Esa Alghonaim, Aiman El-Maleh and Adnan Al-Andalusi,” Parallel computing platform for evaluating LDPC codes performance,” accepted to *IEEE International Conference on Signal Processing and Communications* (ICSPC 2007).

[18] T. Richardson, "Error Floors of LDPC Codes," Proc. 41th Annuial Allerton Conf: on Communication, Computing and Control, Monticello, IL, Oct 2003.