Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2010, Article ID 432495, 17 pages
doi:10.1155/2010/432495

Research Article

The Manifestation of Stopping Sets and Absorbing Sets as Deviations on the Computation Trees of LDPC Codes

Eric Psota[1] and Lance C. Pérez[2]

[1] *University of Nebraska-Lincoln, 329 SEC, Lincoln, NE 68588-0511, USA*
[2] *University of Nebraska-Lincoln, 243N SEC, Lincoln, NE 68588-0511, USA*

Correspondence should be addressed to Eric Psota, epsota24@huskers.unl.edu

Received 16 March 2010; Accepted 10 June 2010

Academic Editor: Christian Schlegel

The error mechanisms of iterative message-passing decoders for low-density parity-check codes are studied. A tutorial review is given of the various graphical structures, including trapping sets, stopping sets, and absorbing sets that are frequently used to characterize the errors observed in simulations of iterative decoding of low-density parity-check codes. The connections between trapping sets and deviations on computation trees are explored in depth using the notion of *problematic* trapping sets in order to bridge the experimental and analytic approaches to these error mechanisms. A new iterative algorithm for finding low-weight problematic trapping sets is presented and shown to be capable of identifying many trapping sets that are frequently observed during iterative decoding of low-density parity-check codes on the additive white Gaussian noise channel. Finally, a new method is given for characterizing the weight of deviations that result from problematic trapping sets.

1. Introduction

Prior to 1993, channel codes were typically designed with the goal of maximizing the minimum distance of the code [1, 2]. The combination of a code with a large minimum distance and a decoder that minimizes the probability of codeword error often resulted in good asymptotic performance. The discovery of turbo codes [3] and the rediscovery of low-density parity-check (LDPC) codes [4] revealed that codes with relatively poor minimum distance properties could achieve near-capacity performance at bit error rates of $P_b < 10^{-6}$. This resulted in a reduced emphasis on maximizing minimum distance when design codes for use on the additive white Gaussian noise (AWGN) channel. As with many other classes of codes, there are no practical bounds for the decoders used for turbo codes and LDPC codes and simulations are required to accurately determine the performance at practical operating points.

With the discovery of turbo codes and the various subsequent iterative decoders, the phenomenon of the error floor has become prominent in practical code design. The term *error floor* refers to the situation where the error rate at the output of the decoder suddenly starts to decrease at a slower rate as a function of increasing signal-to-noise ratio (SNR); that is, the performance curve flattens out. The error floors that occur with iterative decoding of turbo codes and LDPC codes are problematic in practical systems because it is difficult to predict the specific operating point at which they occur, and thus design engineers risk using codes that may have unknown error floors that limit the performance of the system. In the case of iterative decoding of turbo codes, it has been shown that the error floor is usually the result of the overall turbo code having low weight codewords that begin to limit the performance of the code after some SNR is reached [5]. The minimum distance of a turbo code can be increased, and hence the likelihood of an error floor sufficiently mitigated, through the use of various interleaver designs [6, 7].

Low-density parity-check codes with iterative decoding are also known to exhibit error floors [8, 9], albeit at much lower bit error rates than the error floors of turbo codes of similar block length. In many cases, because of the exceptionally low bit error rates at which the error floors of LDPC codes appear, it is not practical to use Monte

Carlo simulations to demonstrate the existence of these error floors. The inability to run conventional computer simulations down to the error floor combined with the lack of practical upper bounds for LDPC codes has inhibited the deployment of LDPC codes in high-throughput applications that require near error-free performance with bit error rates of $P_b < 10^{-15}$.

LDPC codes are most commonly decoded using iterative message-passing decoders such as the min-sum decoder [10] and the sum-product decoder [11] due to their excellent performance and low implementation complexity. Many attempts have been made to estimate the performance of these decoders by characterizing their error mechanisms. Three of the most well-known error mechanisms are stopping sets [12], trapping sets [13], and absorbing sets [8]. Unfortunately, none of these mechanisms leads to strict upper bounds on the performance of iterative decoding over the AWGN channel, and thus they are of limited use in determining error floors. Wiberg showed that deviations on the computation trees of LDPC codes can be used to compute tight upper bounds on the performance of LDPCs with iterative decoders [10]; however, it is computationally intractable to do so after even a small number of decoder iterations. A practical method for determining the error floor of LDPCs with iterative decoding has yet to be discovered.

This paper attempts to make progress on this problem by integrating the precise, but computational intractable, work of Wiberg with the experimental studies of the error mechanisms observed when iteratively decoding LDPCs. The paper begins with a tutorial review of the existing methods for analyzing the performance of iterative message-passing decoders. Then, the notion of a *problematic* trapping set is introduced and its relationship to deviations is examined in detail, with the goal of determining what makes deviations either more or less problematic during iterative message-passing decoding. Finally, an iterative method is given for finding problematic trapping sets using the weights of deviations on the computation trees.

2. Background

The following model for channel coding is used throughout this paper. First, a vector $\mathbf{u} \in \mathbb{F}_2^K$ of K information bits is generated by a binary source. The binary source is assumed to be memoryless, which is often the result of source coding (data compression), and therefore all information sequences in $\mathbb{F}_2^K$ are equally probable. A binary $K \times N$ generator matrix G may be used by the channel encoder to map the information bits $\mathbf{u}$ to a length N codeword $\mathbf{c} \in \mathbb{F}_2^N$, via the mapping $\mathbf{c} = \mathbf{u}G$. Here, it is assumed that the matrix G is full-rank, and thus the rate of the code is $R = K/N$.

Before a codeword $\mathbf{c} \in C$ is transmitted over the channel, it is modulated via the transformation

$$x_i = m(c_i) = 2c_i - 1, \tag{1}$$

for all $i = 0, \ldots, N-1$. The received signal vector $\mathbf{y} \in \mathbb{R}^N$ is given by

$$\mathbf{y} = \mathbf{x} + \mathbf{n}, \tag{2}$$

where $\mathbf{n} \in \mathbb{R}^N$ is the Gaussian noise vector. The log-likelihood ratio (LLR) vector, often used for soft-decision decoding, is given by

$$\lambda_i = \frac{P_{Y|X}(y_i \mid -1)}{P_{Y|X}(y_i \mid 1)}, \tag{3}$$

for all $i = 1, \ldots, N$. This reduces to $\lambda_i = (-2/\sigma^2)y_i$ when $\mathbf{n}$ is a vector of AWGN. An estimate $\hat{\mathbf{c}}$ of the transmitted codeword $\mathbf{c}$ is derived from the received vector $\mathbf{y}$ at the channel decoder. Finally, the information bits $\hat{\mathbf{u}}$ extracted from $\hat{\mathbf{c}}$ are passed to the sink.

As mentioned earlier, each of the information sequences $\mathbf{u} \in \mathbb{F}_2^K$ is equiprobable. Since there is a one-to-one mapping between information sequences and codewords, all codewords in the code C are equiprobable as well. Therefore, $P(\mathbf{c}_i) = P(\mathbf{c}_j)$ for all $\mathbf{c}_i, \mathbf{c}_j \in C$, where $P(\mathbf{c}_i)$ is the probability that codeword $\mathbf{c}_i$ is transmitted. When considering the performance of linear codes, equiprobable codewords allow for the assumption that the all-zeros codeword was transmitted. The all-zeros codeword assumption is used throughout this paper.

From the generator matrix G, it is possible to derive an $(N-K) \times N$ parity-check matrix H for the code. A *parity-check matrix* of a code C is any matrix H, such that $H\mathbf{c}^T = \mathbf{0}$ for all $\mathbf{c} \in C$. LDPC codes are often defined by their parity-check matrix H. In particular, LDPC codes are a class of codes with sparse parity-check matrices. A *sparse* parity-check matrix is any binary matrix that contains more binary 0s than binary 1s. A (d_V, d_F)-*regular* LDPC code is one that has a fixed number d_V of binary 1s in each column of the parity-check matrix and some fixed number d_F of binary 1s in each row of the parity-check matrix. An example of a (2,3)-regular LDPC code of length $N = 6$ and dimension $K = 3$ is given by the parity-check matrix

$$H_{(2,3)} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}. \tag{4}$$

The parity-check matrix of a length N, dimension K code must contain at least $(N-K)$ rows, since the kernel of G has dimension $(N-K)$. However, it is possible for the parity-check matrix to contain more than $(N-K)$ rows. Therefore, the number of rows in the parity-check matrix is denoted by M, where $M \geq (N-K)$.

A *Tanner graph* is a bipartite graphical representation of a low-density parity-check matrix. To construct a Tanner graph from a parity-check matrix, each column i in the parity-check matrix is assigned to a corresponding variable node v_i in the Tanner graph, and each row j is assigned to a corresponding check node f_j in the Tanner graph. The set of all variable nodes is V, and the set of all check nodes is F. There is an edge $e_{i,j}$ between variable node v_i and check node f_j in the Tanner graph if and only if the entry in H at the intersection of the jth row and ith column is a binary 1. The Tanner graph $T = (V \bigcup F, E)$ is thus defined by the set of variable nodes V, the set of check nodes F, and the set of

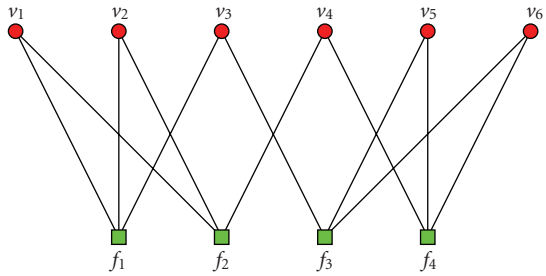


Figure 1: Tanner graph of a $(2,3)$-regular LDPC code.

edges E. The Tanner graph corresponding to the parity-check matrix $H_{(2,3)}$ (given by (4)) is shown in Figure 1.

Note that in the Tanner graphs of irregular LDPC codes variable nodes and check nodes do not all have the same number of incident edges. The number of check nodes that a specific variable node v_i is connected to is denoted by d_{v_i}, and the number of variable nodes that a specific check node f_i is connected to is denoted d_{f_i}.

2.1. Iterative Message-Passing Decoding. The min-sum (MS) and sum-product (SP) decoders are low-complexity, suboptimal iterative decoders that can be used to decode low-density parity-check codes. Given a particular parity-check matrix, the MS decoder operates by passing messages between the check nodes and the variable nodes along the edges of the Tanner graph of the code.

Before introducing the decoders, some additional notation is necessary. The set of neighbors of check node f_i in the Tanner graph is denoted $N(f_i) = \{v_j \mid h_{i,j} = 1\}$, and similarly the set of neighbors of variable node v_i in the Tanner graph is denoted $N(v_i) = \{f_j \mid h_{j,i} = 1\}$. To denote the set of neighbors of check node f_i excluding variable node v_j, the notation $N(f_i) \setminus v_j$ is used. Similarly, the set of neighbors of variable node v_j excluding check node f_i is denoted $N(v_j) \setminus f_i$. During decoding, messages are passed between neighboring check nodes and variable nodes along the edges of the Tanner graph. Messages from check node f_i to variable node $v_j \in N(f_i)$ are denoted by $m_{f_i \to v_j}$, and messages from variable node v_i to check node $f_j \in N(v_i)$ are denoted $m_{v_i \to f_j}$. Given the transmitted codeword $\mathbf{x}$, the channel output $\mathbf{y}$ available at the receiver, and a maximum number of iterations $\ell_{\max}$, the steps for MS and SP decoding are given in the following algorithm

Algorithm 1 (Min-Sum/Sum-Product Decoding).

Step 1 (Initialization). Set the number of iterations to $\ell = 0$. For all messages $m_{f_i \to v_j}$, set

$$m_{v_i \to f_j} = \lambda_i = \frac{P_{Y|X}(y_i \mid -1)}{P_{Y|X}(y_i \mid 1)} = \frac{-2}{\sigma^2} y_i. \tag{5}$$

Step 2 (Check Node Update). Set $\ell = \ell + 1$. For all messages $m_{v_i \to f_j}$, set

Min-Sum:

$$m_{f_i \to v_j} = \left(\prod_{v_k \in N(f_i) \setminus v_j} \operatorname{sgn}\left(m_{v_k \to f_i}\right) \right) \left(\min_{v_k \in N(f_i) \setminus v_j} \mid m_{v_k \to f_i} \right). \tag{6}$$

Sum-Product:

$$m_{f_i \to v_j} = 2 \cdot \tanh^{-1} \left(\prod_{v_k \in N(f_i) \setminus v_j} \tanh\left(\frac{m_{v_k \to f_i}}{2} \right) \right). \tag{7}$$

Step 3 (Variable Node Update). For all messages $m_{v_i \to f_j}$, set

$$m_{v_i \to f_j} = \lambda_i + \sum_{f_k \in N(v_i) \setminus f_j} m_{f_k \to v_i}. \tag{8}$$

Step 4 (Check Stop Criteria). For all m_{v_i}, set

$$m_{v_i} = \lambda_i + \sum_{f_k \in N(v_i)} m_{f_k \to v_i}. \tag{9}$$

For all $\hat{c}_i$, set

$$\hat{c}_i = \begin{cases} 0 & \text{if } m_{v_i} > 0, \\ 1 & \text{if } m_{v_i} < 0, \end{cases} \tag{10}$$

with $P(\hat{c}_i = 0 \mid m_{v_i} = 0) = P(\hat{c}_i = 1 \mid m_{v_i} = 0) = 0.5$.

If $H\hat{\mathbf{c}}^T = \mathbf{0}$ or $\ell \geq \ell_{\max}$, stop decoding, else return to Step 2.

One of the primary strengths of the min-sum and sum-product decoders is the relatively small number of operations performed during each iteration. During each iteration, the messages $m_{v_i \to f_j}$ and $m_{f_j \to v_i}$ must be computed for each binary 1 in the parity-check matrix. For a (d_V, d_F)-regular LDPC code, there are $(N \times d_V) = (M \times d_F)$ binary 1s in the parity-check matrix. When the degree of the nodes and the number of iterations is fixed, the complexity of MS decoding scales linearly with the length N of the code.

In practice, the min-sum and sum-product decoders do not always output a codeword. It has been shown that when the MS decoder does not output a codeword after a large number (>200) of iterations has been performed, the output often cycles in a repeating sequence of two or more noncodeword outputs [14]. In Sections 2.2 through Section 2.4, three different characterizations are given for the noncodeword outputs of iterative message-passing decoders.

2.2. Stopping Sets. The notion of stopping sets was first introduced by Forney et al. [15] in 2001. Two years later, a formal definition of stopping sets was given by Di et al. [12]. They demonstrated that the bit and word error probabilities of iteratively decoded LDPC codes on the binary erasure channel (BEC) can be determined exactly from the stopping sets of the parity-check matrix.

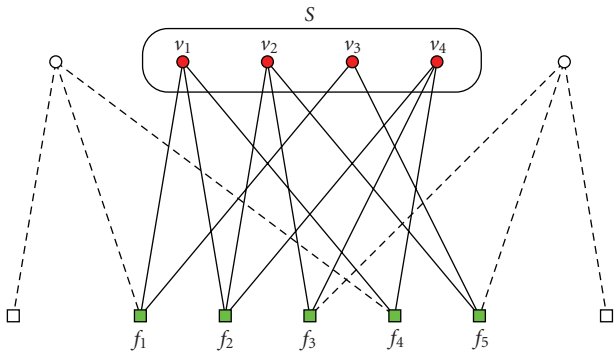


Figure 2: Example of a stopping set in the Tanner graph of an LDPC code.

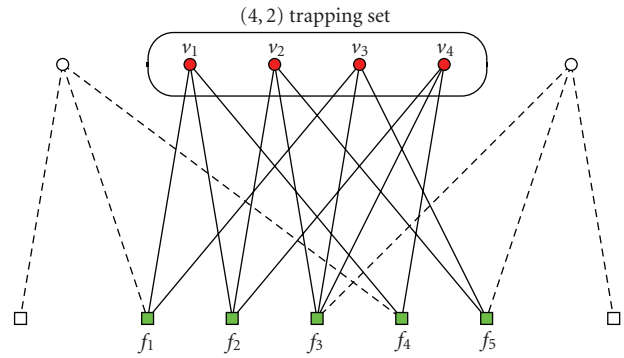


Figure 3: Example of a $(4, 2)$ trapping set in the Tanner graph of an LDPC code.

Definition 1 (stopping sets [12]). A stopping set $\mathcal{S}$ is a subset of the set of variable nodes V, such that any check node connected to a variable node contained in $\mathcal{S}$ is connected to at least two variable nodes in $\mathcal{S}$.

A small example of a stopping set is given in Figure 2. Consider the subset $\mathcal{S} = \{v_1, v_2, v_3, v_4\}$ of the set of variable nodes V. There are five check nodes $\{f_1, f_2, f_3, f_4, f_5\}$ connected to the set $\mathcal{S}$, and each of them is connected to $\mathcal{S}$ at least two times. Note that only f_2 is connected to the set $\mathcal{S}$ an odd number of times; If each of the check nodes is connected to $\mathcal{S}$ an even number of times, $\mathcal{S}$ corresponds to a codeword support set where all bits in $\mathcal{S}$ can be flipped without changing the overall parity of any of the check nodes.

The intuition behind stopping sets begins with an understanding of iterative message-passing decoders. Information given to a specific variable node from a neighboring check node is derived from all other variable nodes connected to that check node. Consider two variable nodes $v_i, v_j \in N(f_k)$, where both variable nodes contain an erasure. In this case, each of the sets $N(f_k) \setminus v_i$ and $N(f_k) \setminus v_j$ contains at least one erasure, thus making it impossible for the check node f_k to determine the parity of either set. For this reason, none of the check nodes connected to a stopping set is capable of resolving erasures, if each variable node contained in the stopping set begins with an erasure from the channel.

Work relating linear programming (LP) pseudocodewords to stopping sets for the binary erasure channel [15], and both the binary symmetric channel (BSC) and the additive white Gaussian noise channel [16], has revealed a relationship between linear programming pseudocodewords and the size of stopping sets. Although stopping sets have a strong relationship with LP pseudocodewords, the performance of neither the MS decoder or the SP decoder on the BSC and AWGN channels can be predicted using stopping sets alone.

2.3. Trapping Sets. Trapping sets, also referred to as near-codewords, were first introduced by MacKay and Postol [13] to provide an explanation for the weaknesses of algebraically constructed low-density parity-check codes. They define trapping sets as follows.

Definition 2 (trapping sets [13]). Consider a length N code with parity-check matrix H, and let $\mathcal{T} \subseteq \{1, \ldots, N\}$ be a set containing $|\mathcal{T}| = t$ coordinates. Consider a length-N binary vector $\mathbf{y}$ with 1s in the coordinates of $\mathcal{T}$ and 0s elsewhere. If the syndrome $\mathbf{s} = H\mathbf{y}$ has Hamming weight w_t, the set $\mathcal{T}$ is referred to as a (t, w_t) trapping set.

Consider the trapping set shown in Figure 3, where the set $\mathcal{T} = \{1, 2, 3, 4\}$ corresponds with a set of variable nodes $\{v_1, v_2, v_3, v_4\}$ in the Tanner graph of the parity-check matrix H. There are four variable nodes in the set, so $t = 4$, and if all variable nodes are set to a binary 1, only check nodes f_2 and f_3 are connected to an odd number of binary 1s, so the syndrome $\mathbf{s}$ has Hamming weight equal to 2. Therefore, according to Definition 2, this set of variable nodes defines a $(4, 2)$ trapping set.

It is important to note that any set of variable nodes can be considered a trapping set defined by some set of parameters, and the significance of trapping sets varies greatly depending on the parameters (t, w_t). In much the same way that low-weight codewords are problematic to decoding, erroneous channel information is more likely to affect the majority of variable nodes in a trapping set which has low-weight t. Richardson [17] shows that trapping sets with small weight t and a small number of unsatisfied check nodes w_t are more likely to cause errors. When a trapping set has small w_t, the extrinsic information being passed into $\mathcal{T}$ can not overcome the intrinsic information reinforced within $\mathcal{T}$.

In [17], trapping sets are examined for different decoders on the binary erasure channel, binary symmetric channel, and the additive white Gaussian noise channel. Whereas stopping sets can be used to precisely determine the probability of error on the BEC, trapping sets appear to cause errors on the AWGN channel. Richardson [17] uses the parameters and multiplicity of various problematic trapping sets to estimate the error floor of LDPC codes at bit error rates where simulations are not feasible. Unfortunately, the somewhat vague definition of problematic trapping sets makes it difficult to use them for performance analysis.

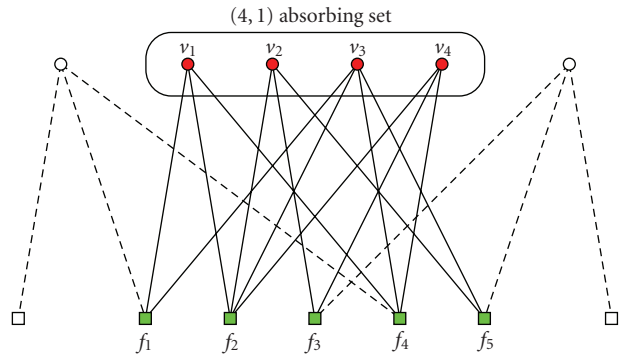


Figure 4: Example of a $(4, 1)$ absorbing set in the Tanner graph of an LDPC code.

2.4. Absorbing Sets. In an attempt to clarify the ambiguity of problematic trapping sets, Zhang et al. introduced the notion of absorbing sets [8]. They define absorbing sets as follows.

Definition 3 (Absorbing Sets [8]). Let $\mathcal{A} \subseteq V$ be a set containing $|\mathcal{A}| = a$ variable nodes. Also, let $O(\mathcal{A}) \subseteq F$ be a set of check nodes such that $|O(\mathcal{A})| = w_a$, and each check node in the set $O(\mathcal{A})$ has an odd number of edges connected to $\mathcal{A}$. If each variable node in $\mathcal{A}$ is connected to strictly more check nodes in $F \setminus O(\mathcal{A})$ than in $O(\mathcal{A})$, the set $\mathcal{A}$ is referred to as a (a, w_a) absorbing set. A *fully absorbing set* also satisfies the condition that each variable node in V is connected to more check nodes in $F \setminus O(\mathcal{A})$ than in $O(\mathcal{A})$.

Note that an (a, w_a) absorbing set is also an (a, w_a) trapping set, but the converse is not always true. Figure 4 shows an example of a $(4, 1)$ absorbing set. The set of variable nodes in this absorbing set is $\mathcal{A} = \{v_1, v_2, v_3, v_4\}$, and the set of unsatisfied check nodes is $O(\mathcal{A}) = \{f_4\}$. The variable node v_2 is not connected to f_4, and the variable nodes v_1, v_3, and v_4 are each connected to f_4 and at least two other check nodes in F. Therefore, each of the variable nodes in $\mathcal{A}$ is connected to more satisfied check nodes than unsatisfied check nodes. Also, note that the $(4, 2)$ trapping set in Figure 3 is not an absorbing set because variable nodes v_2 and v_4 are connected to two unsatisfied check nodes and only one satisfied check node.

Simulations show that the majority of errors encountered in the error floor region during sum-product decoding of the IEEE 802.3 an low-density parity-check code could be attributed to absorbing sets [8, 9]. Although absorbing sets appear to be useful for estimating the performance of iterative message-passing decoding, they do not lead to strict upper bounds. For upper bounds, it is possible to use the concept of deviations on the computation tree.

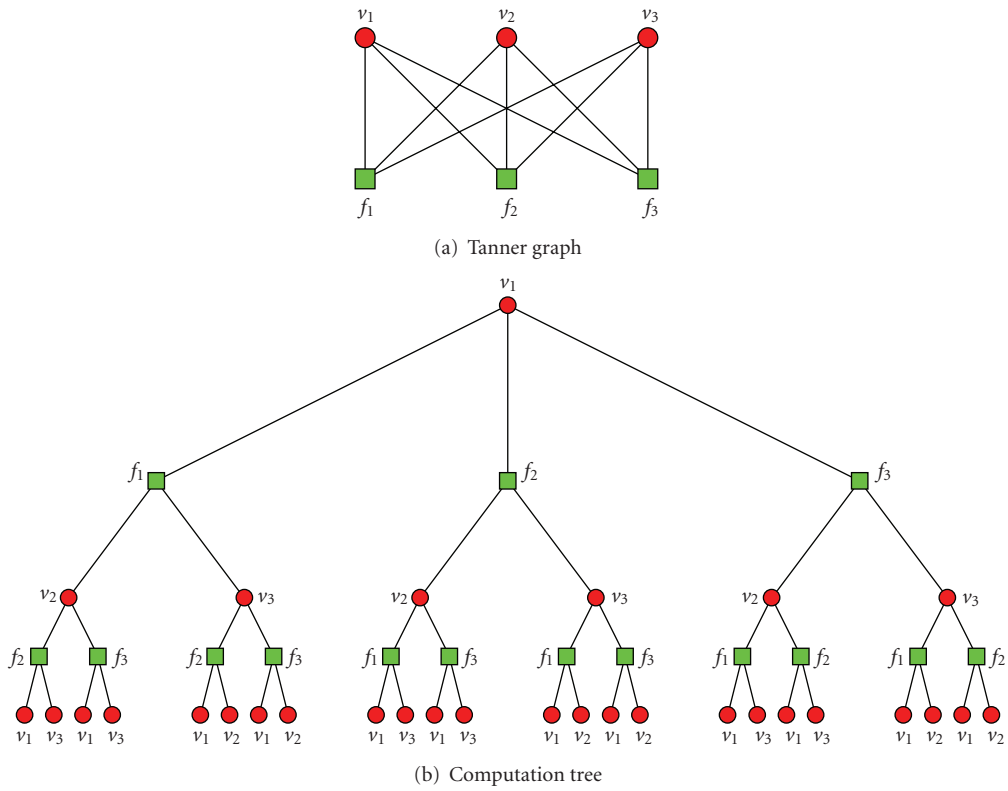
2.5. Computation Trees and Deviations. In his 1996 dissertation, Wiberg [10] presented groundbreaking analytical results with respect to iterative decoding of low-density parity-check codes. He provided extensive analysis of both the MS and SP decoders by introducing a model of iterative decoding known as the computation tree. Wiberg showed that the MS decoder minimizes the probability of word error when decoding a code whose Tanner graph is a tree, while for the same type of code the SP decoder minimizes the probability of bit error.

In addition to introducing computation trees, Wiberg also introduced the concept of deviations. Wiberg proved that deviations on the computation tree with negative cost are required in order for errors to occur during MS and SP decoding. Because of the importance of computation trees and deviations in understanding finite tree-based decoding, they are examined in detail in this section.

Consider a low-density parity-check code represented by a Tanner graph $T = (V \bigcup F, E)$. A computation tree rooted at variable node v_i after ℓ iterations is denoted $R_{v_i}^{(\ell)}$. In order to construct a computation tree from the Tanner graph, a variable node v_i is placed at the top level (root) of a descending tree. To construct the next level in the tree directly below v_i, each of v_i's neighbors in $N(v_i)$ is added to this level and connected to v_i. This process continues level-by-level, where nodes in the previous level are used to determine nodes on next level, while maintaining that each node in the computation tree has the same set of neighbors as its corresponding node in the Tanner graph. For example, if variable node v_j on the last completed level is connected to check node f_k on the level above it, then all check nodes in $N(v_j) \setminus f_k$ must appear on the next level and be connected to v_j, thereby ensuring that v_j is connected to exactly one copy of each check node in $N(v_j)$.

Figure 5 gives an example of a Tanner graph, and its corresponding computation tree rooted at v_1 after two iterations. Nodes at the bottom level of the computation tree are referred to as *leaf nodes*. Notice that the leaf nodes are the only nodes in the computation tree that are not connected to a copy of each of their neighbors in the original Tanner graph.

Computation trees are precise models for analyzing the performance and behavior of min-sum and sum-product decoding for a finite set of iterations. Each of these decoders can be precisely modeled after ℓ iterations by constructing N different computation trees that contain $2\ell+1$ levels of nodes including the root node. The N computation trees are each rooted at a different variable node from the original Tanner graph. Then, for every variable node v_i in each computation tree, the LLR cost γ_i is assigned to that variable node. At this point, MS or SP decoding operations can be performed from the leaf nodes up to the root node. The final cost at each of the root nodes determines the binary estimate of the transmitted codeword computed by the decoder. Because the MS and SP decoders are optimal on Tanner graphs that are trees, the MS and SP decoders are optimal on each of the computation trees derived from the Tanner graph. MS chooses the least cost valid configuration on the tree, where a *valid configuration* refers to any assignment of binary numbers to the variable nodes such that each check node is adjacent to an even number of variable nodes assigned to a binary 1. The SP decoder, on the other hand, chooses the value at the root node that has the highest probability over all valid configurations.



(a) Tanner graph

(b) Computation tree

Figure 5: Computation tree of a simple repetition code after $\ell = 2$ iterations.

Although the computation tree model is precise, after a small number of iterations it becomes impractical to analyze the performance of specific codes by considering all valid configurations on the computation tree. The number of valid configurations on the computation tree can be computed by treating the computation tree as a Tanner graph. In order to define a Tanner graph given the computation tree, treat all check nodes and variable nodes in the computation tree separately. For example, if multiple copies of variable node v_1 are distributed throughout the computation tree, each copy is treated as a distinct variable node. After regarding each variable node in the computation tree as distinct, one can show that each check node on the computation tree corresponds to a linearly independent parity-check equation. If there are $|R_{v_i}^{(\ell)}(V)|$ variable nodes and $|R_{v_i}^{(\ell)}(F)|$ check nodes on a computation tree rooted at variable node v_i after ℓ iterations, then there are a total of $2^{|R_{v_i}^{(\ell)}(V)|-|R_{v_i}^{(\ell)}(F)|}$ valid configurations on the tree. On a (d_V, d_F)-regular LDPC code, the number of variable nodes after ℓ iterations is given by

$$\left|R_{v_i}^{(\ell)}(V)\right| = 1 + \sum_{i=0}^{\ell-1} d_V(d_F-1)\left(((d_V-1)(d_F-1))^i\right), \tag{11}$$

and the number of check nodes is given by

$$\left|R_{v_i}^{(\ell)}(F)\right| = \sum_{i=0}^{\ell-1} d_V\left(((d_V-1)(d_F-1))^i\right). \tag{12}$$

Table 1: The number of nodes and valid configurations on the computation tree of a $(3,6)$-regular LDPC code.

Iterations	Variable Nodes	Check Nodes	Configurations
1	16	3	8192
2	166	33	$\approx 10^{40}$
3	1666	333	$\approx 10^{401}$

To illustrate the growth rate in the number of valid configurations on the computation tree, consider an LDPC code where each variable node has degree $d_V = 3$ and each check node has degree $d_F = 6$. These commonly used code parameters result in what are known as a $(3,6)$-regular LDPC codes. Table 1 shows the number of variable nodes given by

$$\left|R_{v_i}^{(\ell)}(V)\right| = 1 + \sum_{i=0}^{\ell-1} 15\left(10^i\right), \tag{13}$$

the number of checks nodes given by

$$\left|R_{v_i}^{(\ell)}(F)\right| = \sum_{i=0}^{\ell-1} 3\cdot 10^i, \tag{14}$$

and the corresponding number of valid configurations on the computation tree after 1, 2, and 3 iterations. Note that the growth rate is not affected by the block length of the code.

Table 1 illustrates the computational complexity associated with considering each valid configuration on the

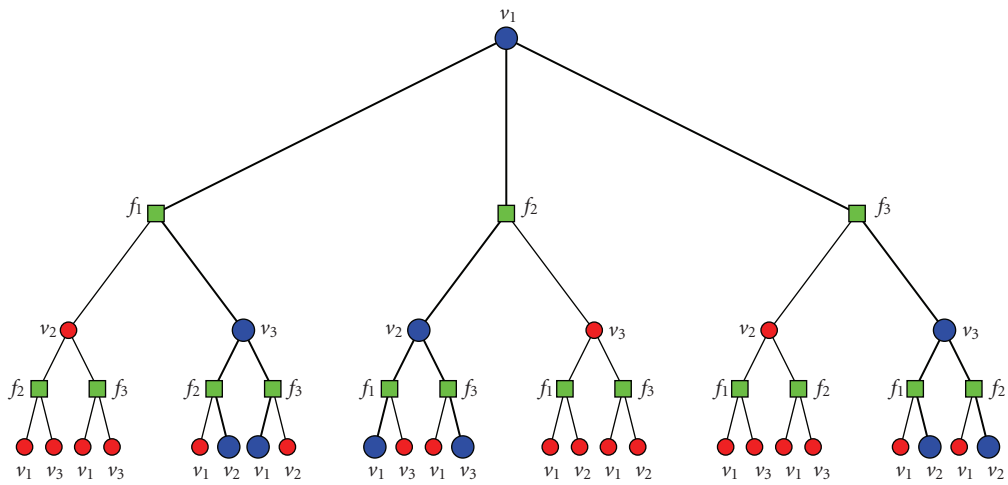


Figure 6: Example of a deviation on the computation tree.

computation tree. In light of this, Wiberg [10] derived a simplified bound on the performance of MS decoding operating on a particular computation tree. In order to obtain this bound, Wiberg introduced the concept of deviations on the computation tree.

Definition 4 (Deviation [10]). A *deviation* is any set of variable nodes on the computation tree satisfying the following three conditions.

(1) Each check node in the computation tree is adjacent to either two or zero variable nodes in the deviation set.

(2) A deviation set contains the root node of the computation tree.

(3) No proper and nonempty subset of variable nodes in the deviation from a valid configuration on the computation tree.

Figure 6 shows an example of a deviation on the computation tree given in Figure 5(b). The larger blue variable nodes are contained in the deviation, whereas the smaller red nodes are not.

Wiberg uses the set of deviations on the computation tree to derive an upper bound on the performance of the min-sum decoder. It is necessary, but not sufficient, for at least one deviation δ in the set of all deviations Δ to have negative cost in order for an error to occur at the root node. The cost of the deviation, denoted by $G(\delta)$, can be found by summing the LLR cost of each of the nodes in the support of the deviation. The cost of a deviation is given by

$$G(\delta) = \sum_{v_i \in \delta} \gamma_i, \tag{15}$$

where copies of $v_i \in \delta$ are counted as many times as they appear in the deviation. A necessary, but not sufficient, condition for an error to occur on the computation tree rooted at variable node v_i is given by [10]

$$\min_{\delta \in \Delta} G(\delta) < 0. \tag{16}$$

Using this condition, a bound can be derived on the probability that the minimum-cost configuration on the computation tree contains a binary 1 at the root node. This bound is

$$\begin{aligned} P(v_i = 1) &\leq P\left(\min_{\delta \in \Delta} G(\delta) < 0\right), \\ &\leq \bigcup_{\delta \in \Delta} P(G(\delta) < 0), \end{aligned} \tag{17}$$

which can be further loosened to

$$P(v_i = 1) \leq \sum_{\delta \in \Delta} P(G(\delta) < 0), \tag{18}$$

by using the union bound.

Wiberg [10] shows that the bound given by (18) can be used to predict the performance of min-sum decoding of infinite-length codes after a specific number of iterations. Wiberg begins by assuming that the computation trees have no repeated nodes. This assumption simplifies the weight enumerators of the deviations for regular LDPC codes. Wiberg also shows that (18) can be used to bound MS decoder performance when there are multiple copies of each variable node in the tree. Thus, in theory, Wiberg's deviation bound can be used to bound the performance of MS decoding of finite length codes. The following proposition shows that the number of deviations grows exponentially with d_V, thus making it computationally intractable to enumerate the deviations even after a small number of iterations.

Proposition 1. *Let $R_{v_i}^{(\ell)}$ be the computation tree of a (d_V, d_F)-regular LDPC code, rooted at variable node v_i after ℓ iterations. Then, the number of deviations that exist on $R_{v_i}^{(\ell)}$ is*

$$(d_F - 1)^{\sum_{i=1}^{\ell} d_V (d_V - 1)^{i-1}}. \tag{19}$$

Table 2: Number of deviations at iterations 1–5 for a $(3,6)$-regular LDPC code.

Iterations	# of deviations
1	125
2	1,953,125
3	4.7684×10^{14}
4	2.8422×10^{31}
5	1.0097×10^{65}

Proof. By the definition of a deviation, we must assign the root node v_i to a binary 1. Each of the d_V check nodes immediately below v_i must assign exactly one of the their $(d_F - 1)$ child variable nodes to a binary 1. Thus, there are a total of $(d_F - 1)^{d_V}$ deviations after one iteration. In addition, there are exactly d_V leaf nodes in the support of each deviation after one iteration.

Each of the previous d_V leaf nodes gets connected to $(d_V - 1)$ check nodes after two iterations. Each of these check nodes assigns one of their $(d_F - 1)$ child variable nodes to a binary 1. Therefore, for each deviation after one iteration there are $(d_F - 1)^{d_V(d_V-1)}$ different deviations after two iterations. This brings the total number of deviations to $(d_F - 1)^{d_V}(d_F - 1)^{d_V(d_V-1)} = (d_F - 1)^{(d_V)^2}$ after two iterations. The total number of leaf nodes in the support of the deviation after two iterations is $d_V(d_V - 1)$.

Following this pattern, the $d_V(d_V - 1)$ variable nodes in support of the deviation after two iterations branchs out to $d_V(d_V - 1)^2$ check nodes. There are $(d_F - 1)^{d_V(d_V-1)^2}$ ways of assigning the leaf nodes to the support of the previous deviation. This brings the total number of deviations to $(d_F - 1)^{(d_V)^2}(d_F - 1)^{d_V(d_V-1)^2} = (d_F - 1)^{(d_V)^3-(d_V)^2+d_V}$ after three iterations.

After ℓ iterations, the $d_V(d_V - 1)^{\ell-2}$ old leaf nodes in the support of the deviation branch out to $d_V(d_V - 1)^{\ell-1}$ new leaf nodes in the support of the deviation. There are $(d_F - 1)^{d_V(d_V-1)^{\ell-1}}$ ways of assigning the support to the previous deviation, and the total number of deviations after ℓ iterations is

$$\prod_{i=1}^{\ell}(d_F - 1)^{d_V(d_V-1)^{i-1}} = (d_F - 1)^{\sum_{i=1}^{\ell} d_V(d_V-1)^{i-1}}. \quad (20)$$

□

The number of deviations on the computation tree of a $(3,6)$-regular low-density parity-check code is given in Table 2 for iterations 1 through 5. Even after only a small number of iterations, it becomes impractical to enumerate each of the deviations in order to compute the upper bound on the probability of bit error of the root variable node of the computation tree.

Using computation trees, Wiberg provided a precise model of the behavior of the min-sum and sum-product decoders. Unfortunately, the size of the computation trees and the number of configurations on them grows too large for practical analysis. Deviations provide a simplified approach to the analysis of computation trees, but the number of deviations also grows exponentially with the number of iterations.

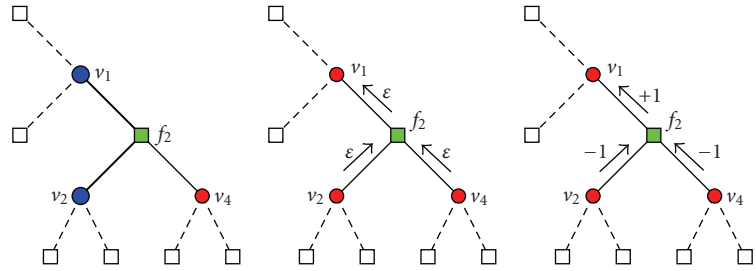
3. Stopping Sets, Absorbing Sets, and Resulting Deviations

Deviations can be used to define a necessary condition for an error to occur during iterative decoding. The condition simply states that there must be at least one deviation with cost less than zero, assuming that the all-zeros codeword was sent. However, this condition says nothing about which deviations are more or less likely to cause errors. What is known is that at high SNRs low-weight, deviations are much more likely to cause errors than high-weight deviations. Thus it is reasonable to expect that low-weight stopping sets and low-weight deviations coincide over the BEC channel, since low-weight stopping sets are precisely the cause of errors for iterative decoding over the BEC [15]. For the same reason, one can expect that trapping sets, or more specifically absorbing sets, coincide with low-weight deviations over the AWGN channel, since they have been frequently observed to cause errors at high SNR during iterative decoding of LDPC codes over the AWGN channel [8, 17]. Connections between stopping/absorbing sets and deviations and their effect on decoding performance are examined in this section.

3.1. Stopping Sets as Deviations. Stopping sets consist of a subset $\mathcal{S}$ of the variable nodes, such that each check node connected to an element in $\mathcal{S}$ is connected at least twice. According to the definition of a deviation given in Definition 4, each check node connected to the variable nodes in a deviation is connected exactly two times. These two properties can be used to study the relationship between stopping sets and their corresponding deviations.

First, consider a computation tree where each of the variable nodes begins with an assignment of a binary 0. Then, assign all copies of variable nodes in $\mathcal{S}$ to a binary 1. If $\mathcal{S}$ does not correspond with a codeword in C, the resulting configuration on the computation tree will not be a valid configuration. For example, consider check node f_2 in Figure 2. Each time, check node f_2 appears in the computation tree, it will be connected to three variable nodes assigned to a binary 1, including the parent variable node and two child variable nodes. If one of the child variable nodes of f_2 along with all of its descendants is set to a binary 0, check node f_2 will be satisfied. If this is done for every unsatisfied check node in the computation tree, a deviation is created that contains only variable nodes in $\mathcal{S}$. Thus, this method allows one to create a deviation from a stopping set.

Using the method previously described, a deviation can be constructed using only the variable nodes contained in a stopping set. This is illustrated in Figure 7(a), where a portion of the deviation defined by the set $\mathcal{S}$ from Figure 2 is given. Since only one of the child variable nodes can be included in the deviation, it is sufficient to randomly include v_2 and exclude v_4, since both nodes are included in $\mathcal{S}$. The effect of this deviation is now examined over the BEC and



(a) Deviation (b) Costs on the BEC (c) Costs on the BSC

FIGURE 7: A small portion of a deviation on a computation tree illustrating the impact of stopping set deviations on the BEC and BSC channels.

the BSC. Stopping sets are known to cause errors over the BEC. The reason for this is illustrated by Figure 7(b). When each variable node in $\mathcal{S}$ is received as an erasure ε, it only takes one erasure below f_2 to cause an erasure message to be sent from f_2 to v_1. However, even though two erasures are connected below f_2 in Figure 7(b), the same erasure message is sent up to v_1. This example illustrates that a deviation containing all erasures is sufficient for an erasure output at the root node. Thus there is a strong connection between stopping sets and deviations over the BEC, since both cause errors using iterative decoding and deviations can be created from stopping sets.

The reason stopping sets will not cause errors as frequently over the BSC is illustrated in Figure 7(c). A deviation containing only the variable nodes in a stopping set exists on a computation tree rooted at any one of the variable nodes in $\mathcal{S}$ regardless of the channel. However, the impact of deviations is not the same across all channels. For example, if two messages of -1 (representing binary 1s) are being sent from v_2 and v_4, the message sent from f_2 up to v_1 is a $+1$ (representing a binary 0). This example shows that while a deviation created from a stopping set is sufficient to cause erasure outputs from the decoder over the BEC, this same deviation may not cause errors over the BSC.

From Figure 7, it is clear that deviations created from stopping sets have a different effect on iterative decoding over the binary erasure channel and the binary symmetric channel. While it is not as clear how deviations created from stopping sets will effect decoding over the AWGN channel, some similarities can be drawn between the BEC and the AWGN. In terms of channel LLR costs, an erasure over the BEC behaves like a LLR cost of zero over the AWGN. A real-valued LLR interpretation of the BEC channel can be created using real-valued costs of $-1.0, +1.0$, and 0.0 to represent a binary 1, 0, and an erasure ε, respectively. Binary information is transmitted as $x = -1.0$ and $x = +1.0$ over the AWGN channel, and the probability $P(y = +1.0 \mid x = -1.0) = P(y = -1.0 \mid x = +1.0) < P(y = 0.0 \mid x = -1.0) = P(y = 0.0 \mid x = +1.0)$, regardless of the channel SNR. Thus, it is reasonable to suspect that the AWGN channel behaves more like the BEC than the BSC, especially at high channel SNR.

3.2. Absorbing Sets as Deviations. Absorbing sets project to deviations on the computation tree in a different way than stopping sets. Since each check node connected to a stopping set $\mathcal{S}$ is connected at least twice, a deviation can easily be defined using only nodes in $\mathcal{S}$ on any computation tree rooted at a node in $\mathcal{S}$. Unlike stopping sets, absorbing sets can have only a single connection to a check node. When an absorbing set has a single connection to a check node, it is not possible to form a deviation on any computation tree using only the variable nodes in $\mathcal{A}$, unless there is a stopping set $\mathcal{S} \subseteq \mathcal{A}$. If $\mathcal{S} \subseteq \mathcal{A}$, a deviation can be formed on the computation tree by simply avoiding variable nodes in $\mathcal{A}$ that are not in $\mathcal{S}$.

For an absorbing set $\mathcal{A}$ that does not contain a stopping set, it is of interest to know how the absorbing set manifests itself as a deviation on the computation tree rooted at one of the variable nodes in $\mathcal{A}$. This manifestation takes the form of a deviation with as many variable nodes in $\mathcal{A}$ as possible. Because it is known that each variable node is connected to strictly more satisfied check nodes than unsatisfied check nodes, it is possible to compute a bound on the number of variable nodes in a deviation δ that are contained in $\mathcal{A}$ for regular LDPC codes.

Consider an absorbing set $\mathcal{A}$ on the Tanner graph of a (d_V, d_F)-regular low-density parity-check code. Let each variable node $v_i \in \mathcal{A}$ be connected to at least $d_{\mathcal{A}} > d_V/2$ satisfied check nodes. A computation tree rooted at a variable node $v_i \in \mathcal{A}$ after ℓ iterations is given by $R^{\ell}_{v_i}$. A deviation on this computation tree can be constructed by selecting the nodes in the deviation level-by-level. The deviation construction begins by including the root node v_i at level $\ell = 0$. At the next level, one variable node connected to each of the check nodes in $N(v_i)$ must be included in δ. When possible, variable nodes in $\mathcal{A}$ will always be included in δ. Therefore, after $\ell = 1$ there are at least $1 + d_{\mathcal{A}}$ variable nodes in δ that are also in $\mathcal{A}$, and at most $d_V - d_{\mathcal{A}}$ variable nodes in δ that are not in $\mathcal{A}$. Continuing to level $\ell = 2$ in the computation tree, each of the $d_{\mathcal{A}}$ variable nodes in $\mathcal{A}$ at level $\ell = 1$ in δ connects to $d_{\mathcal{A}} - 1$ new variable nodes in $\mathcal{A}$ and $d_V - d_{\mathcal{A}}$ new variable nodes not in $\mathcal{A}$. Each of the $d_V - d_{\mathcal{A}}$ variable nodes at level $\ell = 1$ in δ that are not in $\mathcal{A}$ connects to $d_V - 1$ variable nodes that are not it $\mathcal{A}$. After $\ell = 2$, the number of variable nodes in δ that are also in $\mathcal{A}$ is

$$|\delta_{\mathcal{A}}| \geq 1 + d_{\mathcal{A}} \sum_{i=1}^{\ell} (d_{\mathcal{A}} - 1)^{i-1}. \tag{21}$$

Similarly, the total number of variable nodes in δ is

$$|\delta| = 1 + d_V \sum_{i=1}^{\ell} (d_V - 1)^{i-1}. \tag{22}$$

Therefore, the number of variable nodes in δ that are not in $\mathcal{A}$ is $|\delta| - |\delta_{\mathcal{A}}|$.

It is clear from (21) and (22) that the lower bound on the portion of variable nodes in $\mathcal{A}$ within the deviation δ given by $|\delta_{\mathcal{A}}|/|\delta|$ approaches zero as ℓ approaches infinity. Thus, the bound does not appear to be an accurate method for calculating the true portion. The bound given by (21) is computed under the worst-case scenario that, after a deviation reaches a check node with only one connection to the set $\mathcal{A}$, the descendants of that check node contained in the deviation do not contain any more variable nodes in $\mathcal{A}$. On a connected Tanner graph with no nodes of degree one and $d_{\mathcal{A}} \neq d_V$, this assumption is most likely never true, since nodes in $\mathcal{A}$ will eventually (with increasing ℓ) be included in the descendants of the failed check node, and consequently the variable nodes will also be included in any deviation which is a manifestation of $\mathcal{A}$.

For finite-length, (d_V, d_F)-regular low-density parity check codes, it is possible to determine the exact number of variable nodes $|\delta_{\mathcal{A}}|$ after a given number of iterations ℓ. In order to find $|\delta_{\mathcal{A}}|$, each variable node $v_i \in \mathcal{A}$ is assigned a LLR cost of $\lambda_i = 0.0$ and each variable node $v_j \in V \setminus \mathcal{A}$ is assigned a LLR cost of $\lambda_j = +1.0$. Then, using the resulting LLR cost vector, MS decoding is performed for ℓ iterations. The final cost m_i for any variable node $v_i \in \mathcal{A}$, is the number of nodes in the minimum-cost deviation on the computation tree rooted at v_i after ℓ iterations. Deviations for (d_V, d_F)-regular LDPC codes contain a fixed number of variable nodes determined by (22), and the only way to reduce the cost of the deviation is to include variable nodes from the set $\mathcal{A}$. Thus, the minimum-cost deviation on the computation tree of a (d_V, d_F)-regular LDPC code will include the maximum number of variable nodes in $\mathcal{A}$ that is possible, and the MS decoder will output the cost of this deviation. It is important to note that the cost of the deviation returned by the MS decoder corresponds to the number of variable nodes in the deviation that are not in $\mathcal{A}$. Therefore, in order to determine the number of variable nodes in the deviation that are in $\mathcal{A}$, it is necessary to subtract this MS decoder cost from the result given by (22).

Example 1. Consider the length $N = 20$, dimension $K = 10$, $(3, 6)$-regular low-density parity-check code defined by the parity check matrix

$$H = \begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0
\end{bmatrix}. \tag{23}$$

A $(3, 1)$ fully absorbing set is defined by the set of variable nodes $\mathcal{A}_1 = \{v_3, v_7, v_{16}\}$, where columns 1 through 20 of H corresponds to the set of variable nodes $\{v_0, v_1, \ldots, v_{19}\}$. After 50 iterations, each of the deviations on each of the N computation trees contains $|\delta| \approx 3.378 \times 10^{15}$ variable nodes, as computed by (22). After setting the LLR costs for variable nodes in $\mathcal{A}_1$ to $\lambda = 0.0$, and all other LLR cost to $\lambda = +1.0$, the output of the MS decoder after 50 iterations for variable node v_7 is a cost of $m_{v_7} = 3.044 \times 10^{14}$. Therefore, the minimum-weight deviation contains $3.073 \times 10^{15} = 3.378 \times 10^{15} - 3.044 \times 10^{14}$ variable nodes also contained in the absorbing set $\mathcal{A}_1$. Thus, a deviation can be formed on the computation tree with 91% of its variable nodes coming from $\mathcal{A}_1$. In contrast, the bound given by (21) for $(3, 6)$-regular LDPC codes after 50 iterations with $d_{\mathcal{A}} = 2$ only guarantees that 101 variable nodes from $\mathcal{A}_1$ will be included in the minimum weight deviation. This huge disparity between the bound and the actual result given by MS decoding reveals the effects of the assumptions used to derive (21). It is worth noting that the proportion of variable nodes in $\mathcal{A}_1$ in the minimum weight deviation after 51 and 52 iterations were also 91%, so this proportion appears to stabilize after a sufficient number of iterations.

Using this same method on the $(3, 3)$ fully absorbing set $\mathcal{A}_2 = \{v_0, v_7, v_{18}\}$, it is found that the minimum weight deviation after 50 iterations contains 2.696×10^{15} copies of the variable nodes in $\mathcal{A}_2$, equivalent to 78% of the total number of variable nodes in the deviation. By comparing the properties of the minimum-weight deviations resulting from $\mathcal{A}_1$ and $\mathcal{A}_2$, one might expect that $\mathcal{A}_1$ is more likely to cause errors than $\mathcal{A}_2$. This is because 91% of the cost of the deviation resulting from $\mathcal{A}_1$ is determined by the variable nodes in $\mathcal{A}_1$, compared to only 78% for $\mathcal{A}_2$. Simulation results in Section 4 show that $\mathcal{A}_1$ causes errors much more frequently than $\mathcal{A}_2$.

for $i_{\text{fixed}} = 1, \ldots, N$
  Set $m_{\min} = \infty$.
  Set $\chi = \{i_{\text{fixed}}\}$.
  while $m_{\min} > 0.0$
    for $i = 1, \ldots, N$
    -Set $\chi = \chi \bigcup \{i\}$.
    -Set $\lambda_k = 0.0$ for all $k \in \chi$.
    -Set $\lambda_k = 1.0$ for all $k \in V \setminus \chi$.
    -Perform MS Decoding for ℓ iterations.
      if $\min_{j=1,\ldots,N} m_{v_j} < m_{\min}$
        -Set $m_{\min} = \min_{j=1,\ldots,N} m_{v_j}$.
        -Set $j_{\min} = \arg\min_{j=1,\ldots,N} m_{v_j}$.
      end
      -Set $\chi = \chi \bigcap (\{V \setminus \{i\}\} \bigcup \{i_{\text{fixed}}\})$.
      end
    -Set $\chi = \chi \bigcup \{j_{\min}\}$.
    -Create a binary vector $\mathbf{v}$ with $v_k = 1$ if $k \in \chi$, and $v_k = 0$ if $k \in V \setminus \chi$.
    -Compute the integar syndrome $\mathbf{s}_{\text{int}} = H\mathbf{v}^T$.
    -Compute the binary syndrome $\mathbf{s}_{\text{bin}} = H\mathbf{v}^T$ with Hamming weight w_s.
    -Compute the integar vector $\mathbf{z} = H^T \mathbf{s}_{\text{bin}}$
    if $\min_{k=1,\ldots,M} s_{\text{int},k} \geq 2$
      χ is a $(|\chi|, w_s)$ Stopping Set.
    end
    χ is a $(|\chi|, w_s)$ Trapping Set.
    if $z_k < \left\lfloor \frac{d_{v_k}}{2} \right\rfloor$ for all $k \in \chi$
      χ is a $(|\chi|, w_s)$ Absorbing Set.
    end
    if $z_k < \left\lfloor \frac{d_{v_k}}{2} \right\rfloor$ for all $k = 1, \ldots, N$
      χ is a $(|\chi|, w_s)$ Fully Absorbing Set.
    end
  end
end

Algorithm 1: Iterative problematic trapping set finder.

4. Finding Problematic Trapping Sets

Any set of nodes can be interpreted as a trapping set, including stopping sets, absorbing sets, and fully absorbing sets. This is because trapping sets are only defined by the number of variable nodes in the set and the number of failed check nodes. In order to simplify analysis, the trapping sets studied in this paper are restricted to the study of problematic trapping sets.

Definition 5 (Problematic Trapping Set). A *problematic trapping set* is a trapping set such that the number of failed check nodes connected to the trapping set is less than or equal to the number variable nodes contained in the trapping set.

Because trapping sets with small weight and a small number of failed check nodes are often the cause of errors at high SNR [17], it is unlikely that the restriction to problematic trapping sets will eliminate error patterns of interest. In Section 3, it was shown that MS decoding can be used to determine the proportion of variable nodes that are both inside and outside an absorbing set. The same idea is used in this section to find problematic trapping sets using MS decoding. The iterative method given by Algorithm 1 operates by forcing the trapping set to contain one variable node, and then adding more variable nodes one-by-one that decrease the cost of the minimum-cost deviation the most.

Once Algorithm 1 reaches cost $m_{\min} = 0.0$, it is possible to discover more problematic trapping sets by removing nodes one-by-one from the set $\mathcal{X}$ that result in the smallest increase in the cost $m_{\min}$. In order to examine the efficacy of Algorithm 1, the length $N = 20$, dimension $K = 10$, $(3, 6)$-regular low-density parity-check code given in Example 1 was used. This code was chosen because the Hamming weight of all of its codewords could be enumerated, and thus the minimum distance of the code could be easily computed.

Table 3: Stopping/trapping/absorbing sets of a length $N = 20$, dimension $K = 10$, $(3,6)$-regular LDPC code with weight of 3 or less found using Algorithm 1, and the number of times they were observed after 1000 iterations of SP decoding at SNR = 8.0 dB.

Set	Size	Dev. %	Stop.	Abs.	Full Abs.	Observed
$\{v_0, v_{17}\}$	$(2,2)$	74%		X		16
$\{v_1, v_{19}\}$	$(2,2)$	73%		X		7
$\{v_2, v_{14}\}$	$(2,2)$	71%		X		4
$\{v_3, v_{16}\}$	$(2,2)$	72%		X		7
$\{v_4, v_{17}\}$	$(2,2)$	72%		X		28
$\{v_6, v_{15}\}$	$(2,2)$	76%		X		0
$\{v_7, v_{17}\}$	$(2,2)$	73%		X		6
$\{v_8, v_{11}\}$	$(2,2)$	75%		X		21
$\{v_9, v_{14}\}$	$(2,2)$	73%		X		19
$\{v_{10}, v_{18}\}$	$(2,2)$	73%		X		4
$\{v_{11}, v_{19}\}$	$(2,2)$	76%		X		28
$\{v_6, v_{12}\}$	$(2,2)$	76%		X		0
$\{v_6, v_{12}, v_{15}\}$	$(3,1)$	100%	X	X	X	908
$\{v_0, v_{10}, v_{17}\}$	$(3,1)$	91%		X	X	9
$\{v_1, v_4, v_{19}\}$	$(3,1)$	90%		X	X	20
$\{v_2, v_5, v_{14}\}$	$(3,1)$	89%		X	X	6
$\{v_3, v_7, v_{16}\}$	$(3,1)$	91%		X	X	26
$\{v_4, v_{14}, v_{17}\}$	$(3,1)$	90%		X	X	35
$\{v_6, v_7, v_{17}\}$	$(3,1)$	92%		X	X	0
$\{v_8, v_{11}, v_{18}\}$	$(3,1)$	91%		X	X	28
$\{v_9, v_{14}, v_{19}\}$	$(3,1)$	90%		X	X	20
$\{v_7, v_{10}, v_{18}\}$	$(3,1)$	91%		X	X	16
$\{v_{11}, v_{15}, v_{19}\}$	$(3,1)$	91%		X	X	17
$\{v_0, v_7, v_{18}\}$	$(3,3)$	80%		X		0
$\{v_1, v_{14}, v_{19}\}$	$(3,3)$	64%				0
$\{v_2, v_5, v_{19}\}$	$(3,3)$	79%		X		0
$\{v_0, v_4, v_{14}\}$	$(3,3)$	79%		X		0
$\{v_5, v_6, v_{15}\}$	$(3,3)$	86%				0
$\{v_7, v_{12}, v_{15}\}$	$(3,3)$	86%				0
$\{v_2, v_9, v_{12}\}$	$(3,3)$	78%		X		0
$\{v_6, v_{11}, v_{12}\}$	$(3,3)$	85%				0
$\{v_{12}, v_{13}, v_{15}\}$	$(3,3)$	85%				0
$\{v_{13}, v_{14}, v_{19}\}$	$(3,3)$	57%		X		0
$\{v_7, v_{17}, v_{18}\}$	$(3,3)$	85%				0
Other						275

Applying Alogorithm 1 to this code resulted in a total of 37 trapping sets with parameters shown in Table 3. Note that the proportion of nodes in each set that are included in the minimum weight deviation is given by "Dev. %". This code was found to have minimum distance equal to 4, so only stopping/trapping/absorbing sets of weight less than or equal to 3 are tabulated.

In order to determine how effective Algorithm 1 is at locating problematic trapping sets, the same length $N = 20$, dimension $K = 10$, $(3,6)$-regular low-density parity-check code was simulated using sum-product decoding over the additive white Gaussian noise channel with an SNR of $E_b/N_0 = 8.0$ dB. A total of 1500 noncodeword outputs with weight less than or equal to 3 were observed during SP decoding after 1000 iterations. It is important to note that the noncodewords outputs were not simply the last quantized output given after 1000 iterations. The output of SP decoding typically changes after each iteration when it does not converge to a codeword. For this reason, the noncodeword outputs were computed by averaging the cost m_{v_i} for each variable node v_i from $i = 1, \ldots, N$ over the last 200 iterations to compute a final output cost. This is similar to the method used in [14] for characterizing the changing outputs of the MS decoder.

Table 3 shows the number of observed output errors, and compares them to the problematic trapping sets found using Algorithm 1. Approximately 82% of the observed errors corresponded to one of the problematic trapping sets found using Algorithm 1. Also, the number of times a particular problematic trapping set is observed indicates how

problematic the set is to the SP decoder. The single most problematic set, resulting in over 60% of the output errors, was the $(3, 1)$ set that satisfies the definitions of a stopping set, absorbing set, and fully absorbing set. Errors falling into the "Other" category were highly variable, and no specific output pattern in this set accounted for more than 6 of the total observed errors.

The problematic trapping sets with the highest proportion of nodes within their corresponding deviation were the $(3, 1)$ trapping sets. Not surprisingly, the $(3, 1)$ stopping set has the highest proportion of variable nodes in its deviation. While the proportions were noticeably different between different-sized sets, the difference was minimal within sets of the same size. Furthermore, it is difficult to make any connections between proportions within sets of the same size and their corresponding probability of causing an error. One possible reason for this might be the overlap between the different problematic trapping sets, and between the problematic trapping sets and codewords. For example, the reason that the $(2, 2)$ fully absorbing set $\{v_6, v_{12}\}$ did not appear in the simulations might be because two of its variable nodes overlap with the exceptionally problematic $(3, 1)$ stopping set, and thus any significant channel noise received by variable nodes v_6 and v_{12} may be highly likely to cause the $(3, 1)$ stopping set to be output by the decoder.

It is worth noting that the average value of received information within the absorbing sets of weight less than or equal to three was $y = 0.102115$. Recall that a binary 0 is modulated to $x = -1.0$, so the mean value of the noise within the absorbing sets was +1.102115. This cost further justifies the earlier assertion that the AWGN channel behaves more like the BEC channel at high SNR than the BSC channel, since an erasure over the BEC can be interpreted as noise of +1.0 and a bit flip over the BSC can be interpreted as noise of +2.0.

Algorithm 1 was able to find 82% of the most problematic errors with weight less than $d_{\min}$. In order to test the algorithm on an LDPC code with longer block length, a length $N = 200$, $K = 100$, $(3, 6)$-regular LDPC code was used. The resulting output of Algorithm 1 was 1019 absorbing sets, 941 fully absorbing sets, and 1 stopping set, and the sizes of the sets ranged from 3 to 9. Only sets containing less than 10 variable nodes were considered problematic, since the code is known to contain a codeword of weight 10. Simulations were performed using SP decoding at SNR = 5.0 dB, at which the bit error rate of the code is $P_b = 4.8 \times 10^{-9}$. Overall, 40 noncodeword errors were observed, of which 20 were error patterns of weight less than 10. Of these, all were absorbing sets and 19 were fully absorbing sets. Unlike the results given for the length $N = 20$ code, only two of the 20 absorbing sets was found by Algorithm 1. However, the two that were found were the two smallest absorbing sets, including $(5, 3)$ and $(6, 4)$ fully absorbing sets.

As expected, the number of trapping sets grows very large when increasing the size and dimension of the code. Algorithm 1 is capable of locating the majority of problematic trapping sets for a small length $N = 20$ LDPC code, but for the larger length $N = 200$ LDPC code it was only able to identify the two smallest sets observed during simulations. This is likely due to the fact that the code had not yet reached its error floor, as evidenced by the fact that half of the observed error patterns had weight greater than or equal to the weight of a known codeword. Because error floors occur at such low bit error rates for large LDPC codes, it is difficult to observe problematic trapping sets using simulations. Thus, the effectiveness of Algorithm 1 at identifying problematic trapping sets remains unknown for large codes with error floors beyond the reach of simulations.

5. The Weight of Deviations Induced by Problematic Trapping Sets

In [17], Richardson characterizes trapping sets by their size and the number of associated failed check nodes. To find the impact of the trapping sets with respect to probability of error, Richardson uses simulations that force the noise in the trapping set and push the received information away from modulated 0s and towards modulated 1s. The result is an estimate of the probability of error caused by trapping sets at high SNR. In this section, a new method is used to examine the probability of error associated with trapping sets. Instead of using simulations to estimate the probability of error, deviations induced by the trapping set are created to analyze the probability of error. Since bounds on the probability of error can be derived from deviations, if one could prove that minimum-weight deviations were induced by problematic trappings sets and then computed the weights of the deviations, it may be possible determine the probability of error associated with trapping sets without having to rely on simulations.

It may seem surprising that almost all problematic trapping sets listed in Table 3 result in a deviation where the variable nodes within the trappings set make up the majority of the variable nodes within that deviation. For example, consider the fully absorbing set given by the nodes $\{v_{11}, v_{19}\}$. While there are only two variable nodes contained in the absorbing set, they make up 76% of the variable nodes in a deviation that exists on the computation tree rooted at either variable node within the set. This implies that there might be a way of cleverly designing deviations which contain a disproportionately large number of certain variable nodes. A deviation rooted at variable node v_{19} after 4 iterations is shown in Figure 8. This deviation was designed to include more copies of variable nodes v_{11} and v_{19} than other variable nodes. Overall, the number of copies of each variable node in the deviation is $\#v_{11} = 16$, $\#v_{19} = 13$, $\#v_6 = 9$, $\#v_{12} = 5$, and $\#v_{15} = 3$. Although the overall configuration contains 5 different variable nodes, almost 2/3 of them are copies of v_{11} or v_{19}. Now, consider the subgraph of the Tanner graph defined by these 5 variable nodes, shown in Figure 9. This subgraph defines a $(5, 1)$ stopping set. It was shown in Section 3.1 that, because it is a stopping set, the deviation in Figure 8 could continue to grow indefinitely without the need for variable nodes outside the stopping set.

To construct the deviation in Figure 8, decisions were made at check nodes f_0 and f_6. Those decisions are expressed in the directed bipartite graph shown in Figure 10. The number of decisions for each check node is equivalent to the

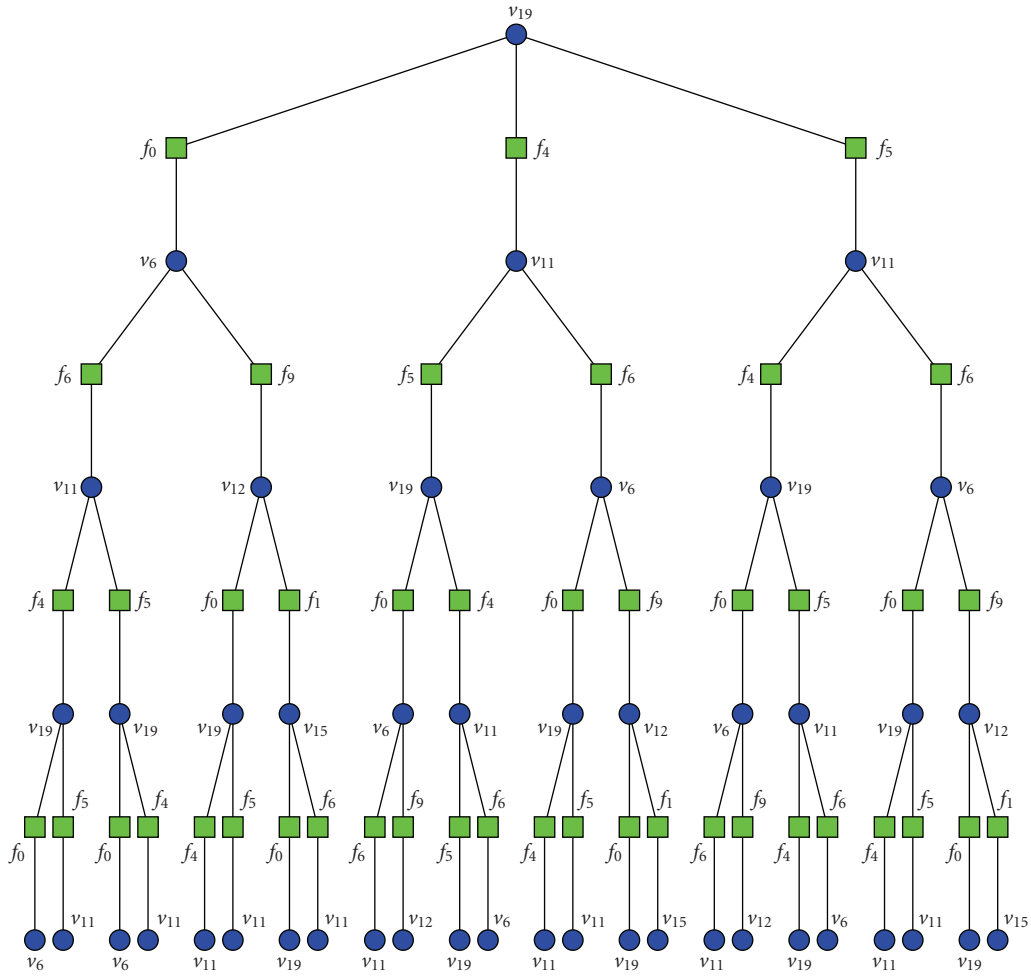


Figure 8: Deviation designed to maximize the number of copies of v_{11} and v_{19}.

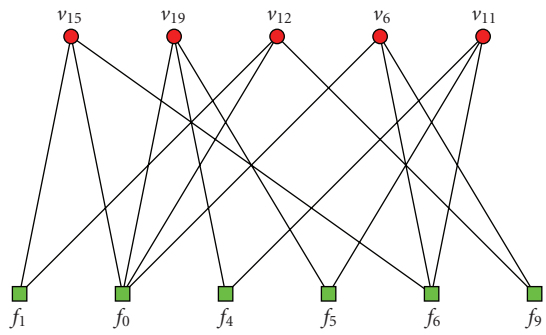


Figure 9: Subgraph of the Tanner graph.

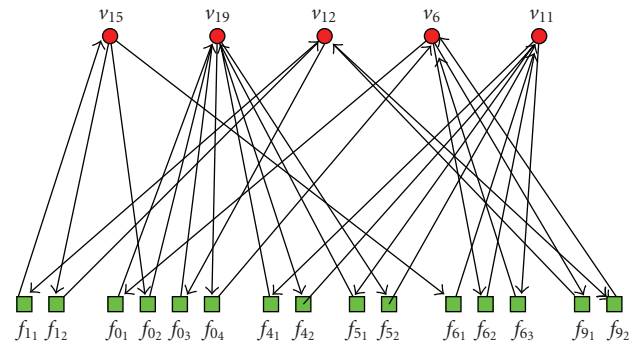


Figure 10: Subgraph of the Tanner graph with repeated check nodes and directed edges.

number of edges incident to the check node. Check nodes f_1, f_4, f_5, and f_9 each have two copies in the directed graph to preserve the fact that they have bidirectional edges. However, check nodes f_0 and f_6 do not simply have bidirectional edges connecting them to each of their incident variable nodes, as demonstrated by the deviation in Figure 8. This comes from the fact that check nodes with degree higher than 2 are still only connected to 2 variable nodes within the deviation. Thus, at each check node with degree greater than 2, a decision is made as to which variable nodes it includes in the deviation.

Using the adjacency matrix of the directed bipartite graph, it is possible to compute the number of copies, or the multiplicity, of each node at each level in the deviation.

Table 4: The multiplicity of variable nodes contained in the deviation created using the directed graph given in Figure 10.

i	v_6	v_{11}	v_{12}	v_{15}	v_{19}
0	0	0	0	0	1
2	1	2	0	0	0
4	2	1	1	0	2
6	2	2	2	1	5
8	4	11	2	2	5
20	326	468	158	78	506
60	3.4×10^8	4.9×10^8	1.7×10^8	8.5×10^7	5.3×10^8
100	3.6×10^{14}	5.2×10^{14}	1.8×10^{14}	8.9×10^{13}	5.5×10^{14}
Total	7.1×10^{14}	1.0×10^{15}	3.6×10^{14}	1.8×10^{14}	1.1×10^{15}

The method for computing the multiplicity of nodes in the deviation is similar to the method given in [18] for computing the multiplicity of nodes on computation trees. For the directed Tanner graph given in Figure 10, the adjacency matrix is given by

$$A = \begin{bmatrix} & & 0&0&0&1&0&0&0&0&0&0&0&0&1&0&1 \\ & & 0&0&0&0&0&0&0&1&0&1&1&1&0&0&0 \\ & \mathbf{0} & 0&0&0&0&0&1&0&0&0&0&0&0&0&1&0 \\ & & 0&0&0&0&1&0&0&0&0&0&0&0&0&0&0 \\ & & 1&1&1&0&0&0&1&0&1&0&0&0&0&0&0 \\ 1&0&0&0&0 & & & & & & & & & & \\ 0&0&0&1&0 & & & & & & & & & & \\ 0&0&1&0&0 & & & & & & & & & & \\ 0&0&0&0&1 & & & & & & & & & & \\ 0&0&1&0&0 & & & & & & & & & & \\ 0&0&0&1&0 & & & & & & & & & & \\ 0&1&0&0&0 & & & & & & & & & & \\ 0&0&0&0&1 & & & \mathbf{0} & & & & & & & \\ 0&1&0&0&0 & & & & & & & & & & \\ 0&0&0&0&1 & & & & & & & & & & \\ 0&0&0&1&0 & & & & & & & & & & \\ 1&0&0&0&0 & & & & & & & & & & \\ 0&1&0&0&0 & & & & & & & & & & \\ 1&0&0&0&0 & & & & & & & & & & \\ 0&0&1&0&0 & & & & & & & & & & \end{bmatrix}. \tag{24}$$

The columns and rows of A correspond, in the same order, to the nodes v_6, v_{11}, v_{12}, v_{15}, v_{19}, f_{0_1}, f_{0_2}, f_{0_3}, f_{0_4}, f_{1_1}, f_{1_2}, f_{4_1}, f_{4_2}, f_{5_1}, f_{5_2}, f_{6_1}, f_{6_2}, f_{6_3}, f_{9_1}, and f_{9_2}. Beginning with a vector $\mathbf{m}_0$ which has a 1 in the position of the root node and 0s elsewhere, the number of nodes on each level i of the deviation can be calculated recursively using

$$\mathbf{m}_i = \begin{cases} A\mathbf{m}_{i-1}, & \text{if } i \text{ is even}, \\ A\mathbf{m}_{i-1} - B\mathbf{m}_{i-2}, & \text{if } i \text{ is odd}, \end{cases} \tag{25}$$

where B is used to subtract the parent check nodes at each check node level in the deviation. The matrix B for the directed Tanner graph in Figure 10 is given by

$$B = \begin{bmatrix} & & & & & & & & & & & & & & \\ & \mathbf{0} & & & & & & & & & & & & & \\ & & & & & & & & & & & & & & \\ & & & & & & & & & & & & & & \\ & & 0&0&0&1 & & & & & & & & & \\ & & 0&0&0&0 & & & & & & & & & \\ & & 0&0&0&0 & & & & & & & & & \\ & & 1&1&1&0 & & & & & & & & & \\ & & & & 0&1 & & & & & & & & & \\ & & & & 1&0 & & & & & & & & & \\ & & & & & & 0&1 & & & & & & & \\ & & & & & & 1&0 & & & & & & & \\ & & & & & & & & 0&1 & & & & & \\ & & & & & & & & 1&0 & & & & & \\ & & & & & & & & & & 0&0&0 & & \\ & & & & & & & & & & 0&0&1 & & \\ & & & & & & & & & & 1&1&0 & & \\ & & & & & & & & & & & & & 0&1 \\ & & & & & & & & & & & & & 1&0 \end{bmatrix}. \tag{26}$$

Using the recursive formula given by (25), the number of variable nodes at different levels in the computation tree is given in Table 4. The deviation at level $i = 100$ exists on the computation tree rooted at variable node v_{19} after $\ell = 50$ iterations. This deviation contains a total of 3.38×10^{15} variable nodes, over 63% of which are copies of v_{11} and v_{19}. Using a similar method to that given in [19], the effective weight of the deviation can be calculated. First, let a_{v_i} be the number of copies of variable node v_i within the deviation, and let b be the total number of variable nodes in the deviation. The cost of the deviation is modeled by the normal distribution

$$\mathcal{N}\left(\sum_{v_i \in \mathcal{S}} \frac{a_{v_i}}{b}, \sum_{v_i \in \mathcal{S}} \left(\frac{a_{v_i}}{b}\sigma\right)^2\right), \tag{27}$$

where σ^2 is the variance of the AWGN noise. This random variable can be rescaled resulting in the distribution

$$\mathcal{N}\left(\left(\frac{\sum_{v_i \in \mathcal{S}} a_{v_i}}{\sqrt{\sum_{v_i \in \mathcal{S}} \left(a_{v_i}\right)^2}}\right)^2, \left(\frac{\sum_{v_i \in \mathcal{S}} a_{v_i}}{\sqrt{\sum_{v_i \in \mathcal{S}} \left(a_{v_i}\right)^2}}\right)^2 \sigma^2\right), \tag{28}$$

where the mean

$$\left(\frac{\sum_{v_i \in \mathcal{S}} a_{v_i}}{\sqrt{\sum_{v_i \in \mathcal{S}} \left(a_{v_i}\right)^2}}\right)^2, \tag{29}$$

is the weight of the deviation. Note that if all a_{v_i} were equal, the weight of the set $\mathcal{S}$ would be equal to the number of nodes it contains, which is consistent with the notion of Hamming weight when $\mathcal{S}$ is equal to a codeword. From Table 4, the weight of the deviation created from the directed Tanner graph in Figure 10 after $\ell = 50$ iterations is 3.8784. This is less than the Hamming weight of the minimum distance codeword in the code, which has weight 4.0. Thus, the minimum weight of deviations on the computation tree rooted at v_{19} is probably less than the minimum distance of the code.

6. Conclusion

Practical methods for predicting and understanding the performance of low-density parity-check codes with iterative decoders are needed in order to avoid the use of codes with error floors. Trapping sets, which include absorbing sets and stopping sets, provide insight into the error mechanisms of iterative decoders but are too imprecise to be used to make design decisions with respect to error floors. Deviations on computation trees are precise and can be used to compute strict upper bounds on the performance of MS and SP decoding, but computing these bounds quickly becomes computationally intractable. The paper examined the connections between trapping sets and their corresponding deviations through the notion of problematic trapping sets in an attempt to find a practical and precise method for predicting the performance of LDPC codes with iterative decoding.

It was shown that the variable nodes in a stopping set can be used to define a deviation, while trapping sets and absorbing sets only define a deviation if a subset of their variable nodes forms a stopping set. When trapping sets and absorbing sets do not include a stopping set, it is necessary to include additional variable nodes in order to construct a corresponding deviation. The number and proportion of variable nodes outside the set that are needed to construct the deviation can be found experimentally using a modification of the MS decoder. This modified MS algorithm leads to an iterative method for identifying low-weight problematic trapping sets in an LDPC code. Simulation results demonstrate that this method is capable of finding many of the low-weight trapping sets that determine the performance of LDPC codes at moderate SNRs. The efficacy of this algorithm is limited by computational constraints.

Finally, an analytical approach for determining the weight of deviations induced by trapping sets on the computation tree was introduced. This approach involves determining the minimum-weight stopping set that contains a given trapping set, and then determining a directed Tanner graph from the stopping set that favors certain variable nodes within the trapping set. It was then shown that the effective weight of the deviation can be found using a recursive method for computing the multiplicity of variable nodes within the deviation, In one example, it was proven that a deviation exists on the computation tree with weight less than the Hamming weight of the code. This result shows that trapping sets probably result in a necessary condition for an error to occur during iterative decoding, and in certain cases, this condition is satisfied with probability higher than the probability of an ML codeword error.

Acknowledgments

This paper was funded in part by AFOSR Contract FA9550-06-1-0375 and Department of Education Grant no. P200A070344.

References

[1] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *SIAM Journal on Applied Mathematics*, vol. 8, pp. 300–304, 1960.

[2] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, pp. 260–269, 1967.

[3] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit errorcorrecting coding and decoding," in *Proceedings of the IEEE International Conference on Communications*, pp. 1064–1070, Geneva, Switzerland, 1993.

[4] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *Electronics Letters*, vol. 32, no. 18, pp. 1645–1646, 1996.

[5] L. C. Pérez, J. Seghers, and D. J. Costello Jr., "A distance spectrum interpretation of turbo codes," *IEEE Transactions on Information Theory*, vol. 42, no. 6, pp. 1698–1709, 1996.

[6] O. Y. Takeshita and D. J. Costello Jr., "New deterministic interleaver designs for turbo codes," *IEEE Transactions on Information Theory*, vol. 46, no. 6, pp. 1988–2006, 2000.

[7] J. Sun and O. Y. Takeshita, "Interleavers for turbo codes using permutation polynomials over integer rings," *IEEE Transactions on Information Theory*, vol. 51, no. 1, pp. 101–119, 2005.

[8] Z. Zhang, L. Dolecek, B. Nikolić, V. Anantharam, and M. J. Wainwright, "Design of ldpc decoders for improved low error rate performance: quantization and algorithm choices," *IEEE Transactions on Communications*, vol. 57, no. 11, pp. 1–12, 2009.

[9] C. Schlegel and S. Zhang, "On the dynamics of the error floor behavior in (regular) ldpc codes," submitted to *IEEE Transactions on Information Theory*.

[10] N. Wiberg, *Codes and decoding on general graphs*, Ph.D. thesis, Linköping University, Linköping, Sweden, 1996.

[11] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufmann, San Mateo, Calif, USA, 1988.

[12] C. Di, D. Proietti, I. E. Telatar, T. J. Richardson, and R. L. Urbanke, "Finite-length analysis of low-density parity-check codes on the binary erasure channel," *IEEE Transactions on Information Theory*, vol. 48, no. 6, pp. 1570–1579, 2002.

[13] D. J. C. MacKay and M. S. Postol, "Weaknesses of Margulis and Ramanujan-Margulis low-density parity-check codes," *Electronic Notes in Theoretical Computer Science*, vol. 74, pp. 99–106, 2003.

[14] N. Axvig, D. Dreher, K. Morrison, E. Psota, L. C. Pérez, and J. L. Walker, "Average min-sum decoding of LDPC codes," in *Proceedings of the 5th International Symposium on Turbo Codes and Related Topics (TURBOCODING '08)*, pp. 356–361, September 2008.

[15] G. D. Forney Jr., R. Koetter, F. R. Kschischang, and A. Reznik, "On the effective weights of pseudocodewords for codes defined on graphs with cycles," in *Codes, Systems, and Graphical Models (Minneapolis, MN, 1999)*, vol. 123 of *IMA Volumes in Mathematics and Its Applications*, pp. 101–112, Springer, New York, NY, USA, 2001.

[16] C. Kelley, D. Sridhara, J. Xu, and J. Rosenthal, "Pseudocodeword weights and stopping sets," in *Proceedings of the IEEE International Symposium on Information Theory*, p. 150, Chicago, Ill, USA, June-July 2004.

[17] T. Richardson, "Error floors of LDPC codes," in *Proceedings of the 41st Allerton Conference on Communications, Control, and Computing*, Monticello, Ill, USA, October 2003.

[18] B. J. Frey, R. Koetter, and A. Vardy, "Signal-space characterization of iterative decoding," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 766–781, 2001.

[19] E. Psota and L. C. Pérez, "LDPC decoding and code design on extrinsic trees," in *Proceedings of the IEEE International Symposium on Information Theory (ISIT '09)*, pp. 2161–2165, June-July 2009.