Finding the Exhaustive List of Small Fully Absorbing Sets and Designing the Corresponding Low Error-Floor Decoder

Gyu Bum Kyung, *Member, IEEE,* and Chih-Chun Wang, *Member, IEEE*

***Abstract*—This work provides an efficient exhaustive search algorithm for finding all small fully absorbing sets (FASs) of any arbitrary low-density parity-check (LDPC) code. The proposed algorithm is based on the branch-&-bound principle for solving NP-complete problems. In particular, given any LDPC code, the problem of finding all FASs of size less than t is formulated as an integer programming problem, for which a new branch-&-bound algorithm is devised with new node selection and tree-trimming mechanisms. The resulting algorithm is capable of finding all FASs of size ≤ 6 for LDPC codes of length ≤ 1000. When limiting the FASs of interest to those with the number of *violated parity-check nodes* ≤ 2, the proposed algorithm is capable of finding all such FASs of size ≤ 13 for LDPC codes of lengths ≤ 1000.**

The resulting exhaustive list of small FASs is then used to devise a new efficient post-processing low-error floor LDPC decoder. The numerical results show that by exploiting the exhaustive list of small FASs, the proposed post-processing decoder can significantly lower the error-floor performance of a given LDPC code. For various example codes of length ≤ 3000, the proposed post-processing decoder lowers the error floor by a couple of orders of magnitude when compared to the standard belief propagation decoder and by an order of magnitude when compared to other existing low error-floor decoders.

***Index Terms*—Branch-&-bound algorithms, error floor, fully absorbing sets (FASs), low-density parity-check (LDPC) codes.**

I. Introduction

Recently, Low-Density Parity-Check (LDPC) codes [1] have received much attention thanks to their near Shannon limit performance and to the ultra-efficient, high-performance belief propagation (BP) decoders. When compared to the prohibitively costly optimal maximum likelihood (ML) decoders, the suboptimality of BP leads to performance degradation both in the *waterfall region* [2] and in the *error-floor region* [3]. Generally, the frame-error-rate (FER) error floor of modern capacity-achieving error control codes (including LDPC codes and turbo codes) is at around 10^{-4} to 10^{-6}, which is detrimental to many important applications that require very low error floor (10^{-12} to 10^{-15}) such as the digital storage devices and optical communications [4]. In the case of turbo codes, the error floor is mainly caused by low-weight codewords. For comparison, the error-floor performance of LDPC codes is related to the error-prone substructures (EPSs) such as stopping sets (SSs) [5], near-codewords [6], trapping sets (TSs) [3], pseudocodewords [7], instantons [8], and absorbing sets (ASs) [9]. Since the EPSs of LDPC codes generally are not valid codewords, the error floor caused by the BP decoder can be lowered by the optimal ML decoder or by an improved BP decoder [10], [11].

The characterization and (exhaustive/inexhaustive) enumeration of EPSs are important for estimating the error floor performance of LDPC codes using importance sampling [3], [12], [13]. Due to the inherent complexity of the exhaustive search problem, several inexhaustive TS search algorithms have been proposed and used to estimate the error-floor performance. In particular, Richardson estimated the error floor of LDPC codes by an inexhaustive list of TSs and by evaluating their contribution to the error floor [3]. Similarly, [14] used FPGA simulation to obtain a partial list of the dominant TSs and they then found all TSs that have the same structure as the dominant TSs by computer search. Reference [15] searched the dominant TSs by applying importance sampling to each 6-bit cycle. Also, [16] obtained the dominant TSs using the tree structure and the error impulse method. Reference [17] found the inexhaustive list of dominant elementary TSs that have the smallest number of unsatisfied check nodes based on the exhaustive list of cycles. Reference [18] proposed an inexhaustive TS enumeration method of TSs by augmenting the Tanner graph.

Finding the minimum distance of an arbitrary linear code is known to be NP-hard [19], [20]. On the other hand, it is worth noting that the NP-hardness is a worst-case study, which may not predict the average cases for the (carefully structured) LDPC codes used in digital communication systems.[1] Moreover, the NP-hardness results focus on the asymptotic regime when the input size $N \to \infty$. It is thus still possible that an algorithm for solving an NP hard problem has exponential complexity but has a very small coefficient, for which the complexity is still very manageable for practical values of N. Unlike other NP-hard problems for which we are interested in as large input sizes as possible, for the case of finding the minimal distance of a given error control code, we generally are interested only in codes of moderate lengths $\leq 10^4$ due to the practical buffer size and delay constraints. As a result, it is of critical value to devise relatively efficient algorithms

This work was supported by NSF Grants CCF-0845968. Part of this paper was presented at the IEEE International Symposium on Information Theory (ISIT), Austin, TX, USA, Jun.13-18, 2010.

G. B. Kyung is with Samsung Advanced Institute of Technology, Samsung Electronics, Yongin, Korea and C.-C. Wang is with the Center for Wireless Systems and Applications (CWSA), the School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907, USA (e-mail: gyubum.kyung@samsung.com and chihw@purdue.edu).

[1]For example, finding the minimum distance of an array-based LDPC code has smaller complexity than that of an arbitrary code [21].

that are capable of finding the minimum distance for codes of moderate length [22].

For LDPC codes, the BP decoder converges to the EPSs instead of the minimum codeword. In this work, we are interested in finding an exhaustive list of all small fully ASs (FASs), which are special types of EPSs first introduced in [9]. Finding all small EPSs is a non-trivial task. For example, the problem of determining the minimum size of the SSs is NP-complete [23] and the NP-hardness is later proven for determining the minimum size of the k-out TSs as well [24]. Reference [25] further generalized the NP-hardness result and showed that even the problem of approximating the minimum size of the "cover SSs" and several different kinds of TSs is also NP-hard. Again, the above NP-hardness results focus on the worst case analysis in the asymptotic regime and thus do not preclude the existence of good algorithms that are capable of exhaustively searching for small EPSs for codes of moderate length (≤ 2000). It is worth noting that although both the problems of finding the minimum Hamming distance and finding the minimum EPS size are NP-hard, for any given linear code, it generally takes longer time to find the minimum Hamming distance than to find the minimum size of EPSs since the size of a minimum EPS is usually much smaller than the size of a minimum codeword. For example, empirically, the minimum Hamming distance of a variable-degree-3 randomly chosen LDPC code of length $\approx 10^3$ is approximately 20 [22], while the minimum size of TSs or SSs of the same code is typically 3 to 5. This fact, together with the recent efficient algorithms of finding the minimum distance for LDPC codes of moderate lengths ≤ 2000 [22], are one of the major motivations of this work.

There are existing algorithms that find the minimal size of any EPSs and/or establish an exhaustive list of small EPSs. Reference [24] proposed the first such exhaustive search algorithm of small SSs and k-out TSs based on tree-pruning techniques. A more efficient exhaustive SS search algorithm was later proposed in [26] based on extended iterative decoding and linear programming techniques. For codes of special structures, exhaustive search may be implemented based on combinatorial methods. For example, for LDPC codes with column degree 3, [27] found all small TSs by counting cycles of minimum length and their possible unions. Other combinatorial exhaustive search methods of ASs for structured LDPC codes can be found in [9], [28].

In parallel with estimating the error floors, several works were proposed to lower the error floor of LDPC codes by improving the suboptimal BP decoder. Reference [29] suggested *averaged BP decoding* to prevent the errors from being trapped during decoding. In contrast with modifying the BP decoder, a different approach to lower the error floor is *post-processing decoding*. Based on a partial TS list, [15] showed that the error floor can be significantly lowered by checking whether the unsatisfied check nodes after BP decoding match that of a TS in the inexhaustive TS list. A bit-guessing approach after decoding failure was developed in [30] to lower the error floor. Reference [10] improved the node selection algorithm of bit guessing and used multi-stage decoding. Reference [11] proposed a bi-mode syndrome-erasure decoder using iterative erasure decoding and a soft erasing technique on the log-likelihood ratio (LLR) messages. Reference [11] also proposed a generalized LDPC decoder to lower the error floor of LDPC codes, which combined several problematic check node processors to a generalized constraint node and applied the BCJR algorithm [31]. Reference [14] used post-processing decoding which lowers the error floor by increasing the reliability of the messages from *unsatisfied* check nodes and decreasing the reliability of the messages from *mis-satisfied* check nodes.

In this work, we first propose an efficient algorithm to find all small FASs for any arbitrary LDPC code that may not have any algebraic structure. We then use the exhaustive list of small FASs to develop a new post-processing decoder that lowers the error floor by a couple of orders of magnitude when compared to the standard BP decoder. Our exhaustive search is based on the branch-&-bound principle, which was previously used in [24] and [26] for exhaustively finding SSs. During the bounding step, we formulate a new integer programming (IP) problem to decide the minimum size of FASs under given constraints and solve the problem using the branch-&-cut algorithm, which can be viewed as a generalization of [22] that finds the minimum Hamming distance using IP solvers. Note that the IP-based approach was also used by [32] to approximate the minimum Hamming distance, which is conceptually quite different from the exhaustive search algorithms in this work and in the existing literature [22], [24], [26]. For numerical evaluations, we applied the proposed algorithms to find the exhaustive lists of small FASs for various benchmark LDPC codes in the literature, which was previously unobtainable by the existing approaches. In all our experiments, our scheme is capable of finding all FASs of size ≤ 6 for LDPC codes of length ≤ 1000. When limiting the FASs of interest to those with the number of *unsatisfied parity-check nodes* ≤ 2, the proposed algorithm is capable of finding all such FASs of size ≤ 13 for LDPC codes of length ≤ 1000.

In the second part of this work, a new post-processing decoder is proposed that takes full advantage of the exhaustive list of small FASs from our search algorithm. In contrast with the existing results based on partial TS lists, the knowledge of the exhaustive list of FASs enables us to better identify the problematic variable and check nodes. A new soft, LLR-based, post-processing decoder is then devised accordingly. Numerical results on the benchmark regular and irregular LDPC codes, including the progressive edge growth codes PEGR504 PEGR1008, PEGI504, and PEGI1008 [33], the MacKay regular codes M816 and M1008, algebraically constructed Tanner (155, 64, 20) code [34], and the Margulis code with $p = 11$ and length 2640 [35], show that for all these codes, the error floor can be lowered by a couple of orders of magnitude when compared to the standard BP decoder, and our post-processing decoder based on the exhaustive small FAS list significantly outperforms the state-of-the-art decoders.

The remainder of this paper is organized as follows. Notations and definitions are included in Section II. In Section III, we describe an exhaustive search algorithm for all small FASs of any given LDPC code. In Section IV, we use the resulting exhaustive list to design a new post processing decoder that

substantially lowers the error floor of LDPC codes. Section V gives the numerical results for the proposed search algorithm and the performance of the new post processing decoder. Section VI concludes the paper.

II. Notations and Definitions

An LDPC code can be defined by an $m \times n$ sparse parity check matrix $\boldsymbol{H}$ where n is the codeword length and m is the number of parity check equations. For convenience, we denote a code by its $\boldsymbol{H}$ matrix. Also, a code $\boldsymbol{H}$ can be represented by its bipartite Tanner graph which has two sets of nodes: the variable nodes $V = \{v_1, \ldots, v_n\}$ and the check nodes $C = \{c_1, \ldots, c_m\}$. Let $\mathcal{N}(v) \subseteq C$ and $\mathcal{N}(c) \subseteq V$ denote the sets of neighboring nodes of a variable node v and a check node c, respectively. Also, let $d_v = |\mathcal{N}(v)|$ and $d_c = |\mathcal{N}(c)|$ be the corresponding node degrees.

For any $A \subseteq V$, let $E_A \subseteq C$ and $O_A \subseteq C$ be the sets of check nodes that are connected to A for an even number (≥ 0) and an odd number of times, respectively. We also define $E_A(v) = E_A \cap \mathcal{N}(v)$ as the set of check nodes in E_A which are connected to v. Similarly, we define $O_A(v) = O_A \cap \mathcal{N}(v)$. The definitions of the AS and its variants [9] can then be described as follows.

Definition 1 (AS and FAS [9]): A subset $A \subseteq V$ is an AS if $|E_A(v)| > |O_A(v)|$ for all $v \in A$. A subset $A \subseteq V$ is a FAS if $|E_A(v)| > |O_A(v)|$ for all $v \in V$.

That is, a subset $A \subseteq V$ is an AS if for any variable node v in A the number of *satisfied check node neighbors* (those with even degrees in the induced subgraph) is larger than the number of *unsatisfied check node neighbors* (those with odd degrees in the induced subgraph). A subset $A \subseteq V$ is a FAS if the above condition is satisfied for all variable nodes in V. Intuitively, a FAS is a "stationary TS" under the majority-vote-based bit-flipping decoder [36]. That is, if all bits in A are in error, and none of the other bits are, then a majority vote, bit flipping decoder cannot correct any of the erroneous bits in A. Decoding thus fails even if we may still have some unsatisfied check node equations.

An AS A of size s is also called an (s, t) AS if $s = |A|$ and $t = |O_A|$. In [37], it has been shown that empirically, most EPSs contain an AS or a FAS, even when using the standard BP decoder on the additive white Gaussian noise channel (AWGNC) instead of the majority-vote-based bit-flipping decoder on binary symmetric channel. One intuitive explanation is that BP decoding can be viewed as a soft version of the majority-vote-based bit-flipping decoder. Therefore, ASs and FASs are likely to contribute to decoding errors even under BP decoding.

Comparison of ASs (FASs) to other EPSs: The definition of ASs is different from the operational definition of TSs [3], instantons [8], and pseudocodewords [7], and is also different from the graphical definitions of SSs, k-out TSs, and near-codewords. More explicitly, a TS is defined as the set of bits that are not correct after decoding [3]. The instantons are special configurations that contribute most to the error rates [8]. Pseudocodewords are the codewords corresponding to the M-cover of the given code [7]. The graphical definition of ASs is closer to those of SSs, k-out TSs, and near-codewords. That is, a subset $A \subseteq V$ is called a SS if the induced subgraph of A has no degree-1 check nodes [5], which is also the trapping set of BP decoders under binary erasure channels. A subset $A \subseteq V$ is called a k-out TS if the induced subgraph of A has exactly k check nodes of degree one [24]. A SS can be viewed as a 0-out TS. The rationale behind k-out TSs is that check nodes with degree one are able to carry correct extrinsic messages from the outside variable nodes when all bits of A are in error. A subset $A \subseteq V$ is called an (s, t) near-codeword [6] if the induced subgraph of A has exactly s variable nodes, and exactly t check nodes of odd degrees. By definition, one can easily prove that any (s, t) AS is always an (s, t) near-codeword, but not vice versa. One thus can view that an (s, t) AS is a special kind of near-codewords that are likely to be detrimental to BP decoding due to being a TS under the majority-vote-based bit-flipping decoder.

III. A New Exhaustive Search Algorithm for Small FASs

In this section, we propose a new exhaustive search algorithm for all small FASs of LDPC codes. We set up an IP problem as a bounding step of the main branch-&-bound algorithm. In addition, we also provide a few techniques to further improve the efficiency of the algorithm.

A. The Main Branch-&-Bound Algorithm

Let $n \triangleq |V|$ denote the total number of variable nodes. For each n-dimensional vector in $\mathbf{s} \in \{0, 1, *\}^n$, the i-th coordinate being $*$, the "unconstrained symbol", means that the value of the i-th variable node is not fixed to "0" or "1." We sometimes say that the i-th coordinate is an unconstrained position of $\mathbf{s}$ if the value of its i-th coordinate being $*$. We say that a ternary vector $\mathbf{s}_1 = (s_{11}, \cdots, s_{1n}) \in \{0, 1, *\}^n$ is *compatible* to another vector $\mathbf{s}_0 = (s_{01}, \cdots, s_{0n}) \in \{0, 1, *\}^n$ if $s_{1i} = s_{0i}$ for all i satisfying $s_{0i} \neq *$. For example, $\mathbf{s}_1 = (1, 0, 0, 1, 0)$ is compatible to $\mathbf{s}_0 = (*, 0, *, 1, 0)$. Let $\mathsf{Supp}(\mathbf{s})$ be the set of variable nodes corresponding to the 1's in a vector $\mathbf{s}$, which is called the support set of the vector $\mathbf{s}$. Conversely, for any set $A \in V$, $\mathsf{Inc}(A)$ is the corresponding *binary incidence vector*, which has 1's in the positions for all $v \in A$ and 0's in the other positions $V \backslash A$. In addition, we use $\mathcal{A}$ to denote a collection of FASs. The main branch-&-bound algorithm is described as follows. Our search algorithm follows from [26], which proposed an efficient exhaustive SS search algorithm based on extended iterative decoding and linear programming techniques.

Algorithm 1 An exhaustive search algorithm for FASs.

1: **Input**: the parity check matrix $\boldsymbol{H}$, two integers $s_{\max}$, and $t_{\max}$.
2: Set $\mathbf{S} \leftarrow \{(*, *, \cdots, *)\}$ and set $\mathcal{A} \leftarrow \emptyset$.
3: **while** $\mathbf{S} \neq \emptyset$ **do**
4: Take one vector $\mathbf{s}_0$ from $\mathbf{S}$, and let $\mathbf{S} \leftarrow \mathbf{S} \backslash \mathbf{s}_0$.
5: **if** $|\mathsf{Supp}(\mathbf{s}_0)| \leq s_{\max}$ **then**
6: **if** $\mathsf{Supp}(\mathbf{s}_0)$ is a FAS and $|O_{\mathsf{Supp}(\mathbf{s}_0)}| \leq t_{\max}$ **then**
7: $\mathcal{A} \leftarrow \mathcal{A} \cup \{\mathsf{Supp}(\mathbf{s}_0)\}$.
8: **end if**

9: Compute a lower bound b such that $b \leq |A|$ for all $A \subseteq V$ satisfying simultaneously the following three conditions: (i) A is a FAS, (ii) $|O_A| \leq t_{\max}$, and (iii) $\mathsf{Inc}(A)$ is compatible to $\mathbf{s}_0$.
10: **if** $b \leq s_{\max}$ **then**
11: Choose one unconstrained position of $\mathbf{s}_0$ and generate two new vectors $\mathbf{s}_1$ and $\mathbf{s}_2$ by setting the unconstrained position of $\mathbf{s}_0$ to 1 and 0, respectively.
12: $\mathbf{S} \leftarrow \mathbf{S} \cup \{\mathbf{s}_1, \mathbf{s}_2\}$.
13: **end if**
14: **end if**
15: **end while**
16: **Output**: $\mathcal{A}$ is the exhaustive list of all (s,t) FASs with $s \leq s_{\max}$ and $t \leq t_{\max}$.

The intuition behind Algorithm 1 is that we start searching from a vector $\mathbf{s}_0$ in Line 4. Lines 5 to 14 ensure that we search for all FASs that are compatible to the chosen vector $\mathbf{s}_0$. If the condition in Line 5 is satisfied, it means that it is still possible to have some FASs that are compatible to $\mathbf{s}_0$ and the search continues. Otherwise, we discard such $\mathbf{s}_0$ and choose another vector from $\mathbf{S}$. In Line 6, we check whether $\mathbf{s}_0$ itself is a FAS in the search range $(s_{\max}, t_{\max})$. If so, we store $\mathsf{Supp}(\mathbf{s}_0)$ in the output list $\mathcal{A}$. The bounding step in Line 9 is critical to the search efficiency. The reason is that if the lower bound b is strictly larger than the search range $s_{\max}$, then there is no FAS in the search range that is also compatible to $\mathbf{s}_0$. We simply discard such $\mathbf{s}_0$ and move on to the next vector in $\mathbf{S}$. If $b \leq s_{\max}$, we proceed to the branching step in Lines 11 and 12. Namely, we choose one unconstrained position of $\mathbf{s}_0$ and generate two new vectors accordingly and store them back into $\mathbf{S}$. The Algorithm 1 continues until the set $\mathbf{S}$ being empty. The output of the Algorithm 1 is the list of all FASs within the search range $(s_{\max}, t_{\max})$.

B. The Bounding Step in Line 9

A good lower bound in Line 9 helps the program to quickly eliminate unnecessary branches and thus speeds up the exhaustive search. Given a to-be-searched vector $\mathbf{s}_0$, we use an IP solver to find a lower bound b in Line 9. In our IP problem, the integer variables are denoted by x_{v_i}, w_{c_j}, and z_{c_j} where the subscripts are the variable nodes $v_i \in V$ and check nodes $c_j \in C$. For notational simplicity, we use x_i, w_j, and z_j as shorthand instead of x_{v_i}, w_{c_j}, and z_{c_j} when there is no ambiguity.

$$\text{Minimize} \quad \sum_{v_i \in V} x_i \tag{1}$$

$$\text{subject to} \quad \sum_{v_i \in \mathcal{N}(c_j)} x_i = 2w_j + z_j \text{ for all } c_j \in C \tag{2}$$

$$\sum_{c_j \in \mathcal{N}(v_i)} z_j \leq \left\lfloor \frac{d_{v_i} - 1}{2} \right\rfloor \text{ for all } v_i \in V \tag{3}$$

$$\sum_{v_i \in V} x_i \geq 1 \tag{4}$$

$$\sum_{c_j \in C} z_j \leq t_{\max} \tag{5}$$

for all the unconstrained positions of $\mathbf{s}_0$, $x_i \in \{0,1\}$ (6)

for all other x_i, set the value to that of the i-th coordinate of $\mathbf{s}_0$ (7)

w_j is a non-negative integer for all $c_j \in C$ (8)

$z_j \in \{0,1\}$ for all $c_j \in C$ (9)

In this IP problem, x_i decides whether v_i is part of a FAS. Equation (2) uses z_j to capture the resulted parity of c_j. Equation (3) follows from the definition of FASs. Equation (4) is used to eliminate the size-0 FAS. Equation (5) is used to decide the search range by limiting the number of the check codes of odd degree in a FAS. The objective function (1) minimizes the size of the FAS satisfying the given constraints. With the given constraint vector $\mathbf{s}_0$, (7) hardwires the corresponding positions of x_i to be 0 or 1 depending on the locations of 0's and 1's in $\mathbf{s}_0$.

C. Implementation

We solve the IP program in the previous subsection by CPLEX 10.2 [38], which is a commercial optimization software that uses LP relaxation and the branch-&-cut[2] principle to solve the IP or mixed IP problems. In our implementation, we define the "branch node limit" and "time limit" to regulate the number of LP relaxation involved. That is, if the number of branching nodes exceeds our predefined node limit or if the elapsed time exceeds the predefined time limit, we stop the branch-&-cut process of the current IP problem, use the current objective value of the LP relaxation as our lower bound b, and proceed to Line 10. The rationale behind introducing the node and time limits is that computing the exact optimal value of the IP problem in (1) to (5) generally takes too much time. Since we are interested only in a lower bound, any objective value of the corresponding relaxed LP problem is sufficient as a loose lower bound. The node and time limits provide a tradeoff between the time it takes to find the lower bound and how tight the lower bound is. In our implementation, we use 5–20 as the branch node limit and 1–5 seconds as the time limit.

As with any branch-&-bound algorithm, the efficiency depends dramatically on the branching policy in Line 11, which chooses an unconstrained location to perform the branching step. We use the following branching policy. During the LP relaxation, the values of x_i are chosen from the interval $[0,1]$ rather than from $\{0,1\}$. For any *soft* x_i values found in the LP relaxation, let z'_j be the distance of $\mathsf{sum}_j \triangleq \sum_{v_i \in \mathcal{N}(c_j)} x_i$ to the closest even number, i.e. $z'_j \triangleq |\mathsf{sum}_j - 2\lfloor 0.5 + 0.5\mathsf{sum}_j \rfloor|$. For any given $V_1 \subseteq V$, let $\mathcal{N}(V_1) \triangleq \bigcup_{v_i \in V_1} \mathcal{N}(v_i)$ be the check nodes adjacent to V_1. Similarly, for any $C_1 \subseteq C$, let $\mathcal{N}(C_1) \triangleq \bigcup_{c_j \in C_1} \mathcal{N}(c_j)$ be the variable nodes adjacent to C_1. For any given vector $\mathbf{s}_0$, let B be the set of variable nodes corresponding to the unconstrained positions in $\mathbf{s}_0$. Given $\mathbf{s}_0$, we compute ξ_k as follows for all its *unconstrained neighbor*

[2]To further improve the algorithm, we add additional user-defined *cuts*, which are first proposed in [22] and greatly reduce the number of branching nodes to prevent the out-of-memory problem.

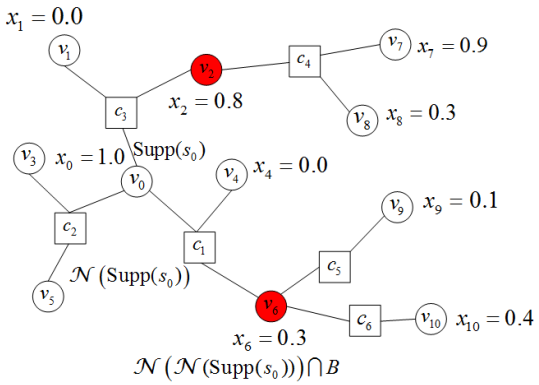


Fig. 1. Illustration of the branching policy.

$v_k \in \mathcal{N}(\mathcal{N}(\mathsf{Supp}(\mathbf{s}_0))) \cap B$:

$$\xi_k \triangleq \min\left(\max\left(\sum_{c_j\in\mathcal{N}(v_k)} z'_{j,x_k=0} - \left\lfloor\frac{d_{v_k}-1}{2}\right\rfloor, 0\right), \max\left(\sum_{c_j\in\mathcal{N}(v_k)} z'_{j,x_k=1} - \left\lfloor\frac{d_{v_k}-1}{2}\right\rfloor, 0\right)\right), \tag{10}$$

where $z'_{j,x_k=0}$ and $z'_{j,x_k=1}$ are the z'_j values computed when hardwiring the x_k value to 0 and 1, respectively. The larger value of ξ_k means that it is more likely that using x_k as a branching choice is going to violate (3) so that the IP solver can quickly refine the solution. Among all $v_k \in \mathcal{N}(\mathcal{N}(\mathsf{Supp}(\mathbf{s}_0))) \cap B$, we choose the one with the largest ξ_k as the branching position in Line 11. To illustrate, consider a simple example in Fig. 1. There are 11 variable nodes v_0 to v_{10}. Consider a to-be-searched vector $\mathbf{s}_0 = (1, 0, *, 0, 0, 0, *, *, *, *, *)$ and the variable values x_0 to x_{10} of the bounding step (based on LP relaxation) are also provided in Fig. 1. Since $\mathsf{Supp}(\mathbf{s}_0) = \{v_0\}$, the unconstrained positions are $B = \{v_2, v_6, v_7, v_8, v_9, v_{10}\}$, $\mathcal{N}(\mathsf{Supp}(\mathbf{s}_0)) = \{c_1, c_2, c_3\}$, and $\mathcal{N}(\mathcal{N}(\mathsf{Supp}(\mathbf{s}_0))) = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}$. As a result, the metric ξ_k is computed for variable nodes in $\mathcal{N}(\mathcal{N}(\mathsf{Supp}(\mathbf{s}_0))) \cap B = \{v_2, v_6\}$, which are the red circles in Fig. 1. For example, the computation of ξ_k for v_6 ($k = 6$) is as follows. We first compute $z'_{j,x_k=0}$ and $z'_{j,x_k=1}$ for c_1, c_5, and c_6, the check node neighbors of v_6. In particular, for c_1, $z'_{1,x_6=0} = x_0 + x_4 + 0 = 1.0 + 0.0 + 0.0 = 1.0$ and $z'_{1,x_6=1} = 2 - (1.0 + 0.0 + 1) = 0$. Similarly, we can compute the $(z'_{j,x_k=0}, z'_{j,x_k=1})$ value for other check nodes c_5 and c_6, and we have $(z'_{5,x_6=0}, z'_{5,x_6=1}) = (0.1, 0.9)$ and $(z'_{6,x_6=0}, z'_{6,x_6=1}) = (0.4, 0.6)$. Thus,

$$\xi_6 = \min\left(\max\left(1.0 + 0.1 + 0.4 - 1, 0\right), \max\left(0.0 + 0.9 + 0.6 - 1, 0\right)\right) = 0.5.$$

For v_2, we compute $(z'_{3,x_2=0}, z'_{3,x_2=1}) = (1, 0)$ and $(z'_{4,x_2=0}, z'_{4,x_2=1}) = (0.8, 0.2)$ for its neighbors c_3 and c_4, and obtain $\xi_2 = \min\left(\max\left(1.0 + 0.8, 0\right), \max\left(0 + 0.2, 0\right)\right) = 0.2$. Therefore, we choose x_6, which has the largest ξ_k, as the next branching node.

For comparison, in our implementation, if we choose the branch location randomly, it takes 45 hours 43 minutes to find all FASs with $s_{\max} = t_{\max} = 5$ of the PEGR504 code [39]. However, it only takes 6 hours 55 minutes using the branching policy described in this subsection.

Now we discuss the complexity of the proposed search algorithm. The complexity greatly depends on the search limit $(s_{\max}, t_{\max})$. In Table I, we provide the execution time in seconds to find $(s_{\max}, t_{\max})$ FASs for M816 using our proposed algorithm.

D. Other Potential Methods for Further Enhancing the Efficiency

In this subsection, we list two other methods that may further improve the efficiency of the branch-&-bound search algorithm. Both of them have a similar effect as that of the node/time limit, which tradeoffs the tightness of the lower bound with faster execution time per branch.

1) LP relaxation for the integer variables: It is worth noting when using the LP relaxation to solve the IP problem, there is no need to identify the optimal integer solution as we are only interested in a lower bound b as in Line 9. Therefore instead of feeding CPLEX with an IP problem in (1) to (5), we can directly feed CPLEX with a LP relaxation or a mixed IP problem to compute a loose lower bound.
2) Use only a subset of check node equations: To further enhance the efficiency of computing a lower bound b, we can further relax the constraints (2) of the IP problem. Namely, instead of finding a solution satisfying (2) for all c_j, we find a relaxed solution that satisfies (2) only for those c_j that is within 3–5 hops[3] away from a variable node in $\mathsf{Supp}(\mathbf{s}_0)$. This approach was first used in [26] for finding SSs. However, for the case of finding FASs, considering a partial list of c_j improves the speed of the bounding step in Line 9, but also results in a much looser lower bound b, which increases the total number of branching steps in Line 11. In our implementation, we notice that this partial list relaxation is more suitable when finding FASs of very small sizes $s_{\max} \leq 5$. When targeting moderate-sized FASs $s_{\max} \geq 6$, it is more efficient to use the full list of constraints.

E. Comparison with the existing works

References [24] and [26] focus on the exhaustive search algorithms for SSs and TSs. They have similar structures as our work in the sense that all three works are based on the branch & bound algorithm. On the other hand, this work is the first to focus on the FAS and we will later show how to use the exhaustive list to design the corresponding low error-floor post-processing decoder. In [24], for codes of lengths $n \simeq 500$, SSs of size ≤ 13 and TSs of size ≤ 11 can be exhaustively enumerated. In [26], for Tanner155 all SSs of size at most 18 can be enumerated in about 1 minute and SSs of size$\leq$ 23

[3]Any parity check node c is always an odd number of hops away from a variable node since the underlying graph is bipartite.

	$s_{\max}=3$	$s_{\max}=4$	$s_{\max}=5$	$s_{\max}=6$	$s_{\max}=7$	$s_{\max}=8$	$s_{\max}=9$	$s_{\max}=10$	$s_{\max}=11$
$t_{\max}=1$	66		73		78		2340		11640
$t_{\max}=2$		89		562		13680		96400	
$t_{\max}=3$	135		8062		32340				
$t_{\max}=4$		7223		108944					

TABLE I
THE TIME IN SECONDS TO FIND $(s_{\max}, t_{\max})$ FASS FOR M816 USING THE PROPOSED ALGORITHM.

can be enumerated for codes of lengths $n \simeq 500$. Also, for structured LDPC codes like IEEE 802.16e LDPC codes the algorithm is able to find SSs of size≤ 28 even for an LDPC code of length $n = 2304$. Our algorithm is capable of finding all FASs of size ≤ 6 for LDPC codes of length ≤ 1000. When limiting the FASs of interest to those with the number of *violated parity-check nodes* ≤ 2, the proposed algorithm is capable of finding all such FASs of size ≤ 13 for LDPC codes of lengths ≤ 1000.

In the existing literature, [27] is the only other method that finds the exhaustive list of small FASs. Since non-exhaustive search algorithms are generally much faster than an exhaustive search scheme, we limit the following complexity comparison to the results in [27]. The approach of [27] is drastically different from the proposed work in this paper. More explicitly, [27] is based on counting cycles to search for AS or FASs. By focusing exclusively on regular LDPC codes with column weight three and by taking the advantage of the corresponding topological structures, the exhaustive search algorithm in [27] is several orders faster than the proposed work in this paper, which tradeoffs the computation efficiency with the types of LDPC codes over which the FASs can be exhaustively searched (our work is capable of searching any given LDPC code, not limited to column-weight-3 codes). For example, [27] reported that the time to exhaustively search the FASs of size ≤ 5 for codes of lengths ≤ 1000 was about five minutes. For comparison, our algorithm can find FASs with $(s_{\max}=5, t_{\max}=3)$ in roughly 2.5 hours for M816 (see Table I for detailed data). Whether the results in [27] can be generalized for regular LDPC codes with column weight four and irregular LDPC codes remains an open problem. Recently, another work finding TSs of LDPC codes has been proposed in [17] and its extended version [40]. The algorithm is able to find dominant TS more efficiently by using other EPSs like cycles as inputs. For example, TSs with $(s_{\max}=13, t_{\max}=3)$ can be enumerated for PEGR504 in 10 minutes, but it remains an open question whether the TS list generated by [17] is inexhaustive or not.

IV. LOW ERROR-FLOOR POST-PROCESSING DECODING

In this section, we propose a new post-processing decoder using the exhaustive list of FASs obtained from the algorithm described in Section III, which lowers the error floor under the AWGNC. Let $m^{(l)}_{v_i\to c_j}$ be the LLR message from a variable node v_i to a check node c_j under a standard BP decoder, which is computed by [41].

$$m^{(l)}_{v_i\to c_j} = m^{(0)}_{v_i} + \sum_{c_k\in\mathcal{N}(v_i)\backslash c_j} m^{(l-1)}_{c_k\to v_i}, \tag{11}$$

where $m^{(0)}_{v_i} \triangleq \frac{2}{\sigma^2}y_i$ is the initial LLR message, y_i is the received symbol, σ^2 is the noise variance, and l denotes the number of decoding iterations. Similarly, let $m^{(l)}_{c_j\to v_i}$ be the LLR message from a check node c_j to a variable node v_i under BP decoding, which is computed by

$$m^{(l)}_{c_j\to v_i} = 2\tanh^{-1}\left(\prod_{v_k\in\mathcal{N}(c_j)\backslash v_i}\tanh\left(\frac{m^{(l)}_{v_k\to c_j}}{2}\right)\right). \tag{12}$$

Also, let $m^{(l)}_{v_i}$ be the LLR message of v_i after l decoding iterations, which is computed by

$$m^{(l)}_{v_i} = m^{(0)}_{v_i} + \sum_{c_k\in\mathcal{N}(v_i)} m^{(l-1)}_{c_k\to v_i}. \tag{13}$$

Let $L^{(0)} = (m^{(0)}_{v_1}, \cdots, m^{(0)}_{v_n})$ and $L^{(l)} = (m^{(l)}_{v_1}, \cdots, m^{(l)}_{v_n})$ denote the initial LLR vector and the final LLR vector after l iterations of BP, respectively. Let $Y^{(0)}$ and $Y^{(l)}$ denote the hard decisions based on the signs of $L^{(0)}$ and $L^{(l)}$, respectively.

The newly available knowledge about the exhaustive FAS list is critical to the design of our post-processing decoder. More explicitly, suppose that the BP decoder indeed converges to a small FAS. The receiver can only observe which parity-check equation is not satisfied but does not know the detailed positions of the error bits. However, since the decoder has an exhaustive list of small FASs, the decoder can check each FAS one by one and see whether any of the small FAS has the same pattern of "unsatisfied" check nodes that matches the observed pattern of "unsatisfied" parity-check equations. Since the list is exhaustive, each small FAS in the list will lead to a unique matching pattern. Then the decoder can simply flip the decoded bit values in those positions and we can decode the true codeword. The error floor can thus be lowered. Unfortunately, this simple *compare-&-flip decoding* does not work for the AWGNC, as the errors are likely caused by some harmful combination of LLR values rather than by explicit bit flipping. Moreover, for AWGNCs, the number of unsatisfied check nodes of $Y^{(0)}$ and $Y^{(l)}$ is usually much larger than that of any single small FAS. Namely, for AWGNCs the BP decoder generally converges to a union of multiple small FASs, instead of a single small FAS. In summary, the post processing decoder needs to take into account the soft, LLR-based nature of AWGNCs, and consider the union of multiple small FASs.

The main idea of our post-processing decoder is to generate a new LLR vector L_{new} based on the exhaustive list of small FASs, $L^{(0)}$, $L^{(l)}$, $Y^{(0)}$, and $Y^{(l)}$. The new LLR vector L_{new} is then used as new initial LLR messages to rerun the BP

decoder. More explicitly, we find a set of suitable FASs in a soft way, take into account the unions of multiple FASs, and then compute L_{new} accordingly. The construction of L_{new} can be fine tuned by changing the values of the parameters[4] $(\beta_{\text{th}}, d, \alpha, \Delta_\alpha, l')$. The detailed description of the algorithm is as follows.

Algorithm 2 A post-processing decoding algorithm for the AWGNC.

1: **Input**: the parity check matrix $\boldsymbol{H}$, the exhaustive list of FASs $\mathcal{A}$, $(L^{(0)}, L^{(l)})$, $(Y^{(0)}, Y^{(l)})$, and the tuning parameters $(\beta_{\text{th}}, d, \alpha, \Delta_\alpha, l')$
2: **if** $Y^{(l)}$, the output of BP, is not a codeword **then**
3: Set the binary vector $U \leftarrow Y^{(0)}$.
4: Run BP for 1 iteration based on $L^{(0)}$ and denote the resulting check-to-variable message as $m^{(1)}_{c_j \to v_i}$.
5: Calculate the *soft parity message* m_{c_j} for all $c_j \in C$ after the 1-iteration BP decoding.
6: For each $A \in \mathcal{A}$, compute

$$\beta_A = \sum_{c_j \in O_A} \left(1 + 1_{\{c_j \in O_{\mathsf{Supp}\left(Y^{(0)}\right)} \cap O_{\mathsf{Supp}\left(Y^{(l)}\right)}\}}\right) m_{c_j}, \tag{14}$$

where $1_{\{\cdot\}}$ is the indicator function.
7: Sort $\mathcal{A}$ according to β_A in the ascending order, that is, $\beta_{A_1} \le \beta_{A_2} \le \cdots$. And set $k \leftarrow 1$
8: **while** $\beta_{A_k} < \beta_{\text{th}}$ **do**
9: Set the LLR vector $L_{\text{new}} \leftarrow L^{(0)}$.
10: Set $A \leftarrow \bigcup_{i=k}^{k+d-1} A_i$, $E \leftarrow \bigcup_{i=k}^{k+d-1} E_{A_i}$, and $O = \bigcup_{i=k}^{k+d-1} O_{A_i}$.
11: Compute the following message scaling function $f(c_j, v_i)$ for all edges (c_j, v_i). In the case that $v_i \in A_{k+h}$ for some $h \in \{0, \cdots, d-1\}$, we set $f(c_j, v_i) = \alpha - h\Delta_\alpha$ if $c_j \in O \cap O_{\mathsf{Supp}(U)}$; and set $f(c_j, v_i) = \frac{1}{\alpha - h\Delta_\alpha}$ if $c_j \in E \backslash O_{\mathsf{Supp}(U)}$. Otherwise, set $f(c_j, v_i) = 1$.
12: Denote the i-th component of L_{new} by m_{v_i}. For those $v_i \in A$, replace the corresponding m_{v_i} by $m^{(0)}_{v_i} + \sum_{c_j \in \mathcal{N}(v_i)} f(c_j, v_i) m^{(1)}_{c_j \to v_i}$.
13: Using the computed L_{new} vector as the initial LLR vector, run BP decoding for l' iterations and let $\bar{Y}$ be the binary decision after l' iterations.
14: If $\bar{Y}$ is a valid codeword, then **RETURN** $\bar{Y}$ as the final output.
15: $k \leftarrow k + 1$.
16: **end while**
17: **end if**
18: **RETURN** $Y^{(l)}$.

The detailed explanation of the above algorithm is as follows. Line 2 focuses only on the case that the BP decoder fails to generate a valid codeword, in which we perform post-processing decoding. Lines 3 and 9 state that we use the initial messages $Y^{(0)}$ and $L^{(0)}$ to initialize the U and the L_{new} vectors. Line 4 prepares some $m^{(1)}_{c_j \to v_i}$ values that will be used later in the algorithm. To compute the soft parity message in Line 5, we first recall that for the standard BP, the check node message is computed by (12). The soft-parity message m_{c_j} in Line 5 is computed similarly by considering all neighbors of c_j:

$$m_{c_j} = 2 \tanh^{-1}\left(\prod_{v_k \in \mathcal{N}(c_j)} \tanh\left(\frac{m^{(1)}_{v_k \to c_j}}{2}\right)\right). \tag{15}$$

In Line 6, we sum up the m_{c_j} values for $c_j \in O_A$. The more negative the sum is, the more likely that those odd parity c_j's are caused by errors in the FAS A. To emphasize the repeated observations from both $Y^{(0)}$ and $Y^{(l)}$, those $c_j \in O_{\mathsf{Supp}(Y^{(0)})} \cap O_{\mathsf{Supp}(Y^{(l)})}$ are emphasized by assigning a weight of 2 instead of 1 when computing β_A in (14). This β_A is then used to sort the exhaustive FAS list $\mathcal{A}$. A_1, having the smallest (the most negative) β_{A_1}, is then the most likely FAS candidate in our post-processing decoder. We then search over a sliding window of d A_i's, from A_k to A_{k+d-1}, for $k = 1, 2, \cdots$. To limit the number of trials, we require $\beta_{A_k} < \beta_{\text{th}}$ to avoid searching over too many FASs. For each window, we use an algorithm that is an improved version of the increase and decrease (ID) algorithm proposed in [14]. The main idea of the ID algorithm is to increase the check-to-variable (c_j-to-v_i) LLR that contains the correcting power (emitting from an unsatisfied check node and entering an erroneous bit) while decreasing the c_j-to-v_i LLR that emits from a mis-satisfied check node (those connected to a non-zero but even number of erroneous bits) and enters an erroneous bit v_i. However, without the knowledge of the exhaustive FAS list, the existing method in [14] has only the knowledge of which c_j is unsatisfied but does not know which $v_i \in \mathcal{N}(c_j)$ is actually in error and needs a stronger correcting power. The method in [14] also does not know which check node is mis-satisfied. In contrast, our scheme takes advantage of both the c_j and the v_i information from our exhaustive FAS list. More explicitly, if $v_i \in A_{k+h}$ and $c_j \in O \cap O_{\mathsf{Supp}(U)}$, then it is likely that c_j is indeed an unsatisfied check node and we would like to use a scaling factor $f(c_j, v_i) > 1$ in order to correct the erroneous bit v_i. Similarly, if $v_i \in A_{k+h}$ and $c_j \in E \backslash O_{\mathsf{Supp}(U)}$, then it is likely that c_j is a "mis-satisfied" check node and we would like to use a scaling factor $f(c_j, v_i) < 1$ to weaken its impact on the erroneous bit v_i. In Line 11, our scaling factor further takes into account the effect of the h value. Namely, we want to impose greater changes on those $f(c_j, v_i)$ with $v_i \in A_{k+h}$ for smaller h. (Recall that with the sorting step Line 7 the A_k is a more likely candidate than A_{k+d-1}). If v_i is in several A_{k+h}'s with different h's, then we use the smallest h for computing $f(c_j, v_i)$. Line 12 replaces part of the L_{new} vector by the newly increased/decreased messages. Line 15 means that if no correct codeword can be found by processing A_k to A_{k+d-1}, we repeat the same procedure but now focus on processing A_{k+1} to A_{k+d}.

The following simple example illustrates the proposed post-processing decoder. Suppose $d = 1$, $\alpha = 1.5$, $\Delta_\alpha = 0.1$, $\beta_{\text{th}} = -0.5$, and $l' = 5$ and we consider a small code of 5 variable nodes and 4 check nodes as depicted in Fig. 2. We also assume the initial LLR messages $L^{(0)}$ are -1.2, -0.5, 1.0, -0.4, and 0.6, and assume after l iterations the final LLR messages $L^{(l)}$ are 2.4, 4.2, 1.2, 2.2, and -2.6, which are listed

[4]One such choice is $\beta_{\text{th}} = 3$, $d = 4$, $\alpha = 1.5$, $\Delta_\alpha = 0.1$, and $l' = 5$.

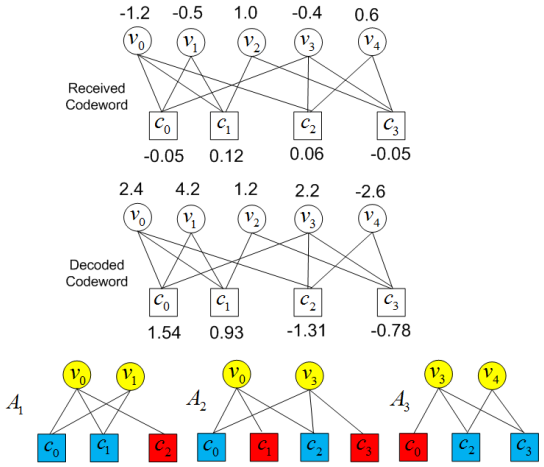


Fig. 2. An example of new post-processing decoding.

on top of each variable node. The corresponding soft parity messages are computed by (15) and listed under each check node. We first use Algorithm 1 to compute an exhaustive list of FASs with the search range being $(s_{\max}, t_{\max}) = (2, 2)$. There are exactly three FASs $\mathcal{A} = \{A_1, A_2, A_3\}$ within the search range and they are depicted in Fig. 2. For each FAS A_i, $i \in \{1, 2, 3\}$, we use red and blue check nodes to represent the check nodes in O_{A_i} and E_{A_i}, respectively. We then calculate β_{A_i} by (14). For example, for the first FAS A_1, since $O_{A_1} = \{c_2\}$, $\mathsf{Supp}(Y^{(0)}) = \{v_0, v_1, v_3\}$, $O_{\mathsf{Supp}(Y^{(0)})} = \{c_0, c_3\}$, $\mathsf{Supp}(Y^{(l)}) = \{v_4\}$, and $O_{\mathsf{Supp}(Y^{(l)})} = \{c_2, c_3\}$, we have $\beta_{A_1} = m_{c_2} = 0.06$. Similarly, we can compute $\beta_{A_2} = m_{c_1} + 2 * m_{c_3} = -0.05 + 2 * -0.05 = -0.15$, and $\beta_{A_3} = m_{c_0} = -0.05$. Since the smaller β_{A_i} (more negative) indicates that FAS A_i is likely to be the error pattern, we sort these three FASs according to β_{A_i} and select A_2 as our first candidate in this example. We then adjust the LLR values of edges from the check nodes using the message scaling function $f(c_j, v_i)$ and $\alpha = 1.5$ as described in Line 11. For example, we compute $f(c_3, v_3) = \alpha - h\Delta_\alpha = 1.5 - 0 \times 0.1 = 1.5$ since $c_3 \in O \cap O_{\mathsf{Supp}(Y^{(0)})}$ and $v_3 \in A_2$. As a result we "strengthen" the LLR message $m^{(1)}_{c_3 \to v_3}$ by the factor $f(c_3, v_3) = 1.5$ in order to increase the correcting power from the unsatisfied check node c_3. Similarly, since $c_2 \in E \backslash O_{\mathsf{Supp}(Y^{(0)})}$, we compute $f(c_2, v_3) = \frac{1}{\alpha - h\Delta_\alpha} = 1/1.5$. We then "weaken" the LLR message $m^{(1)}_{c_2 \to v_3}$ by the factor $f(c_2, v_3) = 1/1.5$ in order to decrease the wrong LLR from the mis-satisfied check node c_2. After all check-to-variable messages are updated, we apply additional $l' = 5$ iterations of BP decoding and see whether the new BP iteration can converge to a true codeword.

Note that the proposed algorithm is based on the initial LLR $L^{(0)}$ and the initial decision $Y^{(0)}$ (through the initialization of L_{new} and U in Lines 9 and 3, respectively). One can easily extend the proposed algorithm by considering the final LLR $L^{(l)}$ and the final converged decision $Y^{(l)}$ as well. That is, we replace Line 3 by $U \leftarrow Y^{(l)}$, replace Line 9 by $L \leftarrow L^{(l)}$, delete Line 4, and compute Lines 5 and 12 directly from the $m^{(l)}_{c_j \to v_i}$ messages after l BP iterations. In our implementation, we concatenate these two algorithms (one based on $L^{(0)}$ and one based on $L^{(l)}$, respectively) in sequence, i.e., we run the $L^{(0)}$-based scheme first and if it succeeds, then stop. If it does not succeed, we then continue to run the $L^{(l)}$-based scheme. This concatenation further enhances the performance as the usage of $L^{(l)}$ is able to catch some errors that are not decodable by the original $L^{(0)}$-based scheme. In our experiments, the performance of the algorithm is quite robust and any parameters in the range: $\beta_{\text{th}} \in [0, 10]$, $d \in [1, 10]$, $\alpha \in [1.5, 2]$, $\Delta_\alpha \in [0, 0.2]$, $l' = [1, 20]$, lead to good error-floor performance.

V. Numerical Results

A. Results of the Exhaustive FAS Search Algorithm

We report the results of applying the proposed algorithms to three classes of codes, randomly generated regular codes, randomly generated irregular codes, and codes generated by algebraic methods.

1) Randomly Constructed Regular LDPC Codes: We apply Algorithm 1 to four randomly generated regular $(3, 6)$ LDPC codes from [39]. PEGR504 and PEGR1008 are of length 504 and 1008 and are constructed by the PEG algorithm [33]. M816 and M1008 are regular $(3, 6)$ LDPC codes of lengths 816 and 1008. The results are listed in Table II. The first column of Table II specifies the code of interest, the second column specifies the search range of the FASs, and the following columns provide detailed information of the FAS list. For example, our search algorithm shows that for PEGR504, there are only two types of FASs satisfying $s \leq 14$ and $t = 1$. They are the $(9, 1)$ and the $(13, 1)$ FASs. There is no (11,1), no (12,1), and no (14,1) FAS in PEGR504. For these codes, our algorithm is capable of finding all small FASs when setting the search range $(s_{\max}, t_{\max})$ to $(6, 6)$. Moreover, if we restrict ourselves to search for a smaller $t_{\max} = 2$, Algorithm 1 finds all[5] FASs of size ≤ 13. For example, Algorithm 1 shows that there are exactly 14 different types of FAS in PEGR504 with $s + t \leq 13$ (See the first seven rows of the super-row corresponding to PEGR504 in Table II). If we further assume that the most detrimental FASs have only ≤ 2 unsatisfied check nodes, then Algorithm 1 shows that with a cycle-avoiding construction, PEGR504 has only $1 + 1 + 4 + 6 + 26 = 38$ such FASs and the minimal size of them is $s = 8$ with $t = 2$.

Without taking advantage of any special structures of regular (3,6) codes, Algorithm 1 is actually computationally slower than the degree-3-centric result in [27] when searching for small FASs with $s_{\max} \leq 5$. On the other hand, our scheme can effectively find much larger FASs such as $(10, 2)$ FASs (see Table II) for which there are no analytical exhaustive search results in the existing literature [27]. It is worth noting that as an offline analysis tool, searching the FASs is generally a one-time task. Using an Intel Core2 2.4GHz CPU with 2GB memory, the running time to generate the FAS tables for these four codes ranges from 5 to 48 hours, which is within a reasonable time frame for most code design and

[5]It can be proven that there are no$(s, 2)$ FASs when s is an odd number for the regular LDPC codes with column weight 3.

Code	(s,t) range	(s,t)	Num	(s,t)	Num	(s,t)	Num	(s,t)	Num	(s,t)	Num
PEGR 504	$(\leq 13, = 0)$	No FASs									
	$(\leq 14, = 1)$	$(9,1)$	1	$(13,1)$	1						
	$(\leq 13, = 2)$	$(8,2)$	4	$(10,2)$	6	$(12,2)$	26				
	$(\leq 10, = 3)$	$(5,3)$	14	$(7,3)$	47	$(9,3)$	146				
	$(\leq 9, = 4)$	$(4,4)$	760	$(6,4)$	849	$(8,4)$	2270				
	$(\leq 8, = 5)$	$(5,5)$	10156	$(7,5)$	22430						
	$(\leq 7, = 6)$	$(6,6)$	66352								
PEGR 1008	$(\leq 13, \leq 2)$	No FASs									
	$(\leq 8, = 3)$	No FASs									
	$(\leq 7, = 4)$	$(4,4)$	2	$(6,4)$	1						
	$(\leq 6, = 5)$	$(5,5)$	11236								
	$(\leq 7, = 6)$	$(6,6)$	85845								
M816	$(\leq 13, = 0)$	No FASs									
	$(\leq 14, = 1)$	No FASs									
	$(\leq 13, = 2)$	$(4,2)$	3	$(6,2)$	2	$(8,2)$	1	$(10,2)$	15	$(12,2)$	15
	$(\leq 8, = 3)$	$(3,3)$	126	$(5,3)$	86	$(7,3)$	108				
	$(\leq 7, = 4)$	$(4,4)$	1350	$(6,4)$	2236						
	$(\leq 6, = 5)$	$(5,5)$	7286								
	$(\leq 7, = 6)$	$(6,6)$	68124								
M1008	$(\leq 13, = 0)$	No FASs									
	$(\leq 14, = 1)$	No FASs									
	$(\leq 13, = 2)$	$(4,2)$	6	$(6,2)$	5	$(8,2)$	3	$(10,2)$	4	$(12,2)$	6
	$(\leq 8, = 3)$	$(3,3)$	153	$(5,3)$	8388	$(7,3)$	111				
	$(\leq 7, = 4)$	$(4,4)$	1130	$(6,4)$	1668						
	$(\leq 6, = 5)$	$(5,5)$	8388								
	$(\leq 7, = 6)$	$(6,6)$	72486								
Tanner 155	$(\leq 15, = 0)$	No FASs									
	$(\leq 16, = 1)$	No FASs									
	$(\leq 15, = 2)$	$(8,2)$	465	$(10,2)$	1395	$(12,2)$	930	$(14,2)$	5115		
	$(\leq 12, = 3)$	$(5,3)$	155	$(9,3)$	930	$(11,3)$	5270				
	$(\leq 11, = 4)$	$(8,4)$	1395	$(10,4)$	17670						
	$(\leq 10, = 5)$	$(5,5)$	1860	$(7,5)$	6975	$(9,5)$	33945				
	$(\leq 9, = 6)$	$(6,6)$	13330	8	72850						
	$(\leq 8, = 7)$	$(7,7)$	53475								
Margulis 2640	$(\leq 11, \leq 2)$	No FASs									
	$(\leq 8, = 3)$	No FASs									
	$(\leq 7, = 4)$	$(4,4)$	1320								
	$(\leq 6, = 5)$	$(5,5)$	11088								
PEGI 504	$(\leq 11, = 0)$	No FASs									
	$(\leq 11, = 1)$	$(7,1)$	2	$(8,1)$	8	$(9,1)$	16	$(10,1)$	22	$(11,1)$	39
	$(\leq 8, = 2)$	$(2,2)$	230	$(3,2)$	219	$(4,2)$	208	$(5,2)$	198	$(6,2)$	205
		$(7,2)$	271	$(8,2)$	458						
	$(\leq 5, = 3)$	$(3,3)$	1517	$(4,3)$	3867	$(5,3)$	6888				
	$(\leq 4, = 4)$	$(4,4)$	26927								
	$(\leq 4, = 5)$	$(4,5)$	3322								
PEGI 1008	$(\leq 10, = 0)$	No FASs									
	$(\leq 10, = 1)$	$(7,1)$	1	$(8,1)$	4	$(9,1)$	8	$(10,1)$	14		
	$(\leq 8, = 2)$	$(2,2)$	458	$(3,2)$	439	$(4,2)$	420	$(5,2)$	404	$(6,2)$	387
		$(7,2)$	403	$(8,2)$	519						
	$(\leq 5, = 3)$	$(3,3)$	3041	$(4,3)$	7737	$(5,3)$	13843				
	$(\leq 4, = 4)$	$(4,4)$	105897								

TABLE II

THE NUMBER OF (s,t) FASS FOR DIFFERENT LDPC CODES. IN EACH SUPER COLUMN, THE VECTOR IN THE FIRST SUB-COLUMN REPRESENTS THE (s,t) VALUE SATISFYING THE CONDITION SPECIFIED IN THE SECOND COLUMN FROM THE LEFT AND THE SECOND SUBCOLUMN CORRESPONDS TO THE NUMBER OF SUCH FASS.

analysis purposes. See Table I for detailed computation time comparison.

2) Algebraically Constructed LDPC Codes: We also apply Algorithm 1 to the algebraically constructed (155, 64, 20) Tanner code (Tanner155) [34] and Margulis2640 of $p = 11$ and length 2640 [35]. Our algorithm exhaustively find all FASs with the search range $(s_{\max}, t_{\max})$ set to $(7,7)$ and to $(5,5)$, respectively. See Table II. In addition, if we use a smaller $t_{\max} = 2$, we find all FASs of size ≤ 15 for Tanner155 and size ≤ 11 for Margulis2640. In [24], the exhaustive list of minimal 2-out TSs is found for Tanner155, which is identical to the exhaustive list of $(8,2)$ FASs obtained by our algorithm. Our algorithm also exhaustively finds FASs of larger (s,t) values that were previously not found in [24]. For example, there are 930 $(12,2)$ FASs in Tanner155 and all $(12,2)$ FASs can be generated from the following representatives.

0 37 52 57 60 61 72 73 81 93 136 153,
0 37 43 57 60 61 64 72 89 93 136 145,
0 37 43 52 57 61 64 81 86 93 145 153,
0 24 54 57 61 66 80 93 113 122 130 139,
0 4 41 57 61 63 86 97 104 118 127 141,
0 4 8 18 20 41 57 61 67 97 104 129,

$$0\ 7\ 9\ 24\ 28\ 54\ 57\ 61\ 87\ 93\ 113\ 149,$$
$$\text{and}\ \ 0\ 11\ 37\ 48\ 61\ 69\ 78\ 93\ 97\ 105\ 127\ 150$$

by the automorphisms discussed in [34]. For the Margulis 2640 code, Algorithm 1 shows that there is no FAS of types $(6,4)$, $(7,3)$, $(8,2)$, $(9,1)$, $(10,2)$, and $(11,1)$.

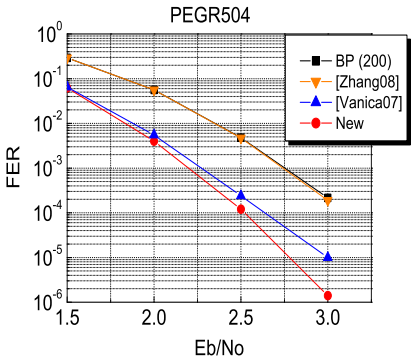
3) Randomly Constructed Irregular LDPC Codes: The proposed algorithm can also be applied to irregular LDPC codes. Nonetheless, for irregular LDPC codes, there exist usually a lot of variable nodes of degree 2. By the definition of FASs, if a degree-2 node v is in a FAS, then v must satisfy $|O_A(v)| < 0.5d_v = 1$, which implies $|O_A(v)| = 0$. This constraint limits the FAS search to finding sets of erroneous variable nodes for which no degree-2 node can be connected to any unsatisfied check node. Such a condition is generally too restrictive, and the resulting FAS search algorithm will automatically exclude too many meaningful error-prone patterns. The following lemma illustrates the restrictiveness of the original FAS definition when applied to irregular LDPC codes that have a substantial number of degree-2 nodes.

Lemma 1: If $\bigcup_{v\in V_2}\mathcal{N}(v) = C$ where V_2 is the set of variable nodes of degree 2, then there is no (s,t) FAS for any $t > 0$. That is, all FASs are of type $(s,0)$ and are actually codewords. The problem of finding the minimum FASs is thus equivalent to that of finding the minimum codewords.

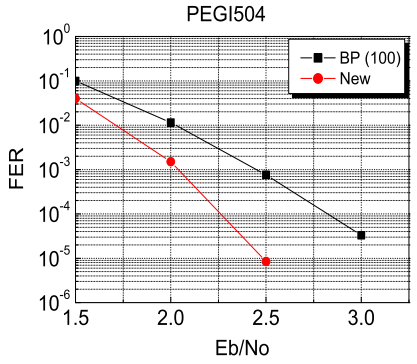
The proof of Lemma 1 is relegated to Appendix A. Note that the condition $\bigcup_{v\in V_2}\mathcal{N}(v) = C$ is satisfied for all dual-diagonal LDPC codes [42], for which Lemma 1 shows that the traditional definition of FASs [9] is too restrictive, and the only EPS that can be a FAS must also be a codeword. To resolve this restriction, for irregular LDPC codes, we relax the definition of a FAS to $|E_A(v)| \geq |O_A(v)|$ for all $v \in V$ of degree 2 and keep the original constraint $|E_A(v)| > |O_A(v)|$ for all v of degree ≥ 3. By modifying (3) of our lower bounding IP problem accordingly, we can apply Algorithm 1 to irregular codes as well and exhaustively identify the FASs using the new relaxed definition. We apply Algorithm 1 to PEGI504 and PEGI1008, two randomly constructed irregular LDPC codes in [39]. Table II lists all the corresponding FASs satisfying simultaneously $s + t \leq 8$ and $s \geq t$.

B. Post-Processing Decoding

In Fig. 3, we plot the FER performance of our new post-processing decoder for PEGR504 and PEGI504 over the AWGNC. We use $l = 200$ and 100 for PEGR504 and PEGI504 respectively, and use the following parameters $\beta_{\text{th}} = 3$, $d = 4$, $\alpha = 1.5$, $\Delta_\alpha = 0.1$ and $l' = 20$. We use all FASs listed in Table II as the input $\mathcal{A}$. For comparison, we also plot the best existing result on PEGR504 [10], denoted by [Varnica07]. As seen in Fig. 3, Algorithm 2 outperforms the bit-guessing scheme in [10] and lowers the error floor of the classic BP decoding by one to two orders of magnitude for both regular and irregular PEG-type LDPC codes. Performance gap between normal BP decoding and our post-processing decoding using FASs is wider in the high signal-to-noise ratio (SNR) regime, as seen in Fig. 3 and the later Figs. 4 and 5. Note that in the high-SNR regime, the number of additional post-processing iterations is generally much smaller because



(a) PEGR504



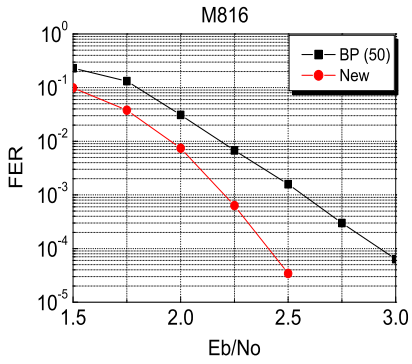
(b) PEGI504

Fig. 3. The FER curves of the proposed decoder compared to the decoders in [10], [14]. 'BP (l)' is the performance of standard BP decoding with l iterations for both codes. '[Zhang08]' is the performance of [14] and '[Varnica07]' is the performance of [10] on PEGR504. The results of our proposed decoder are denoted by 'New'.

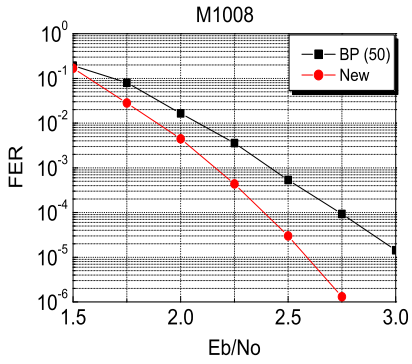
the error pattern in the high-SNR regime is usually a small FAS (or very close to a small FAS). Therefore, Algorithm 2 works very efficiently.

The existing ID algorithm [14] is not very effective without the knowledge of the exhaustive list of small FASs. As can be seen in Fig. 3, the performance improvement of the approach in [14] is almost non-existent, which is in sharp contrast to the algorithm proposed in this work. When applying [14] to PEGR504, we notice that the algorithm in [14] only works when the error of the final decoding is precisely a small FAS. However, this happens only for regular LDPC codes and in the very high SNR regime. For irregular codes, BP decoding does not converge to small FASs even in high SNR region. As a result, even for regular (2048,1723) code, the error-floor improvement [14] is negligible for moderate-to-high SNR (with FER between 10^{-4} to 10^{-7}) and is significant only for very high SNR (FER$\approx 10^{-12}$). For comparison, Algorithm 2 works for both regular and irregular codes over all the entire SNR range. It is worth mentioning that when the final decoding error is precisely a small FAS, we can simply use our exhaustive list to match and flip the erroneous bits.

In Fig. 4, we also show the FER performance of our new



(a) M816



(b) M1008

Fig. 4. The FER curves of the proposed decoder. The result of our proposed decoder is denoted by 'New' and normal BP decoding with 50 iterations is denoted by 'BP (50)'.

post-processing decoder for M816 and M1008 for $l = 50$ iterations. In Fig. 5, we apply Algorithm 2 to Margulis2640, and compare the performance with that of the three existing post-processing decoders in [11]. For this code, we use $l = 50$ iterations of BP decoding and use $\beta_{\text{th}} = 3$, $d = 5$, $\alpha = 1.5$, $\Delta_\alpha = 0.1$, and $l' = 20$ in Algorithm 2. We employ all FASs of Margulis2640 with $s_{\max} = t_{\max} = 5$. The performance of the proposed decoder outperforms the state-of-the-art bi-mode and generalized-LDPC decoders in [11]. It is worth mentioning that with the sorting step in Line 7 of Algorithm 2, oftentimes we do not need to go through the entire FAS list since the complexity of Algorithm 2 is proportional to the number of FASs used, which means that the complexity is kept low. For example, in our experiments, the average number of FASs used in Algorithm 2 is 13.56 at 2.3dB for Margulis2640. Half of the time, the number of FASs used by Algorithm 2 is only 1, which shows the efficiency of the proposed decoder when compared to the results in [11]. Moreover, this is a post-processing decoding which imposes zero overhead when BP is successful. When the original BP is not successful (usually FER$\leq 10^{-5}$ for the high SNR regime of interest), the post-processing decoder imposes additional complexity that is proportional to the number of FASs. In sum, the average complexity is nearly identical to the original BP.

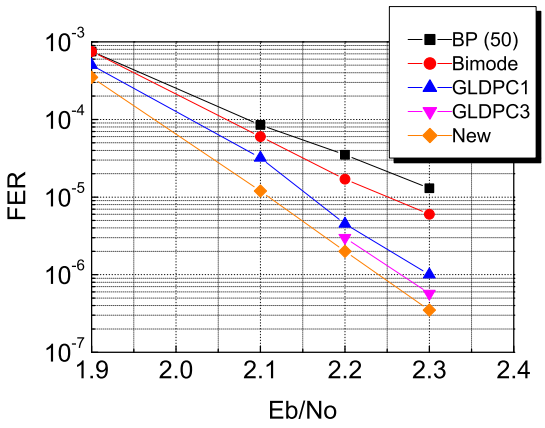


Fig. 5. The FER curves of the proposed decoder for Margulis2640 compared to the decoders in [11]. We use 'GLDPC1' and 'GLDPC3' as shorthand for the two 'generalized' decoders in [11]. The results of our proposed decoder are denoted by 'New'.

For example, if we use the original BP with $l = 50$ iterations, we have FER$= 1.3 \times 10^{-5}$ at 2.3dB for Margulis2640. In our experiment, there is approximately 1 decoding error in 76800 frames. The total number of regular BP iterations for the 76799 error-free frames is roughly 532908 (assuming the BP stops when all parity-check equations are satisfied). For the single erroneous frame, the number of iterations is calculated by $l +$ (the average number of FASs used) $\times\, l' = 50 + 13.56 \times 20 = 322$, which is negligible compared to the normal BP iterations 532908. Note that the decoder in [10] is also a post-processing decoder and the average complexity should be comparable to our proposed algorithm. On the other hand, the approach in [11] modifies the decoder and the resulting decoding algorithm has almost twice the average complexity as analyzed in [11]. The complexity cost of the the bi-mode decoder in [11] is low, but the corresponding performance improvement is also insignificant, as can be seen in Fig. 5.

VI. Conclusions

We have proposed an efficient algorithm to find all small FASs of any given LDPC code using a branch-&-bound algorithm. The bounding step of the exhaustive search algorithm is formulated by a new integer programming problem and several techniques are devised to further enhance its efficiency. We have then used the resulting exhaustive list to design a new post-processing decoder that substantially lowers the error floor of any LDPC code. By taking advantage of the exhaustive FAS list, the post-processing decoder can be made more effective and results on example LDPC codes show that our approach lowers the error floor by a couple of orders of magnitude when compared to the standard BP decoder and outperforms the existing state-of-the-art post-processing decoders.

APPENDIX A
A PROOF OF LEMMA 1

Proof: We need to show $\mathsf{Inc}(A)$ is always a codeword, i.e., $|O_A| = 0$ for all FASs A. Suppose not. Then there exists a FAS A with $|O_A| > 0$. Choose arbitrarily a $c' \in O_A$. Since $\bigcup_{v \in V_2} \mathcal{N}(v) = C$, there exists at least one variable node $v' \in V_2$ that is connected to c'. Thus $|O_A(v')| \geq 1$, which in turn implies that $|O_A(v')| \geq |E_A(v')|$ because $|O_A(v')| + |E_A(v')| = 2$. However, it contradicts the definition of a FAS that $|O_A(v)| < |E_A(v)|$ for all $v \in V$. By contradiction, the proof is complete. ■

REFERENCES

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

[2] D. Burshtein and G. Miller, "An efficient maximum-likelihood decoding of LDPC codes over the binary erasure channel," *IEEE Trans. Inform. Theory*, vol. 50, no. 11, pp. 2837–2844, Nov. 2004.

[3] T. Richardson, "Error floors of LDPC codes," in *Proc. 41st Annu. Allerton Conf. on Commun. Contr. and Computing*. Monticello, IL, Oct. 2003, pp. 1427–1435.

[4] I. B. Djordjevic, S. Sankaranarayanan, S. K. Chilappagari, and B. Vasic, "Low-density parity-check codes for 40-Gb/s optical transmission systems," *IEEE Jour. Select. Topics in Quan. Elec.*, vol. 12, no. 4, pp. 555–562, Jul./Aug. 2006.

[5] C. Di, D. Proietti, E. Telatar, T. Richardson, and R. Urbanke, "Finite length analysis of low-density parity-check codes," *IEEE Trans. Inform. Theory*, vol. 48, no. 6, pp. 1570–1579, Jun. 2002.

[6] D. MacKay and M. Postol, "Weakness of Margulis and Ramanujan-Margulis low-density parity-check codes," *Electronic Notes in Theoretical Computer Science*, vol. 74, 2003.

[7] P. Vontobel and R. Koetter, "Graph-cover decoding and finite-length analysis of message-passing iterative decoding of LDPC codes," Dec. 2005, preprint-arXiv:cs.IT/0512078.

[8] M. Stepanov and M. Chertkov, "Instanton analysis of low-density parity-check codes in the error-floor regime," in *Proc. IEEE Int'l. Symp. Inform. Theory*. Seattle, WA, Jul. 2006, pp. 552–556.

[9] L. Dolecek, Z. Zhang, V. Anantharam, M. Wainwright, and B. Nikolic, "Analysis of absorbing sets and fully absorbing sets of array-based LDPC codes," *IEEE Trans. Inform. Theory*, vol. 56, no. 1, pp. 181–201, Jan. 2010.

[10] N. Varnica, M. P. C. Fossorier, and A. Kavcic, "Augmented belief propagation decoding of low-density parity-check codes," *IEEE Trans. Commun.*, vol. 55, no. 7, pp. 1308–1317, Jul. 2007.

[11] Y. Han and W. E. Ryan, "Low-floor decoders for LDPC codes," *IEEE Trans. Commun.*, vol. 57, no. 6, pp. 1663–1673, Jun. 2009.

[12] E. Cavus, C. L. Haymes, and B. Daneshrad, "Low BER performance estimation of LDPC codes via application of importance sampling to trapping sets," *IEEE Trans. Commun.*, vol. 57, no. 7, pp. 1886–1888, Jul. 2009.

[13] L. Dolecek, P. Lee, Z. Zhang, V. Anantharam, B. Nikolic, and M. Wainwright, "Predicting error floors of structured LDPC codes: deterministic bounds and estimates," *IEEE Jour. Select. Areas in Commun.*, vol. 27, no. 6, pp. 908–917, Aug. 2009.

[14] Z. Zhang, L. Dolecek, B. Nikolic, V. Anantharam, and M. Wainwright, "Lowering LDPC error floors by postprocessing," in *Proc. IEEE Glob. Telecom. Conf.* Cannes, France, Dec. 2008, pp. 1–6.

[15] E. Cavus and B. Daneshrad, "A performance improvement and error floor avoidance technique for belief propagation decoding of LDPC codes," in *Proc. IEEE Int'l. Symp. on Pers., Indoor and Mobile Radio Comm.* Berlin, Germany, Sept. 2005, pp. 2386–2390.

[16] C. A. Cole, S. G. Wilson, E. K. Hall, and T. R. Giallorenzi, "A general method for finding low error rates of LDPC codes," Feb. 2008, preprint-arXiv:cs.IT/0605051v1.

[17] M. K. Dehkordi and A. H. Banihashemi, "An efficient algorithm for finding dominant trapping sets of LDPC codes," in *Proc. 6th Int'l. Symp. Turbo Codes and and Iterative Inform. Processing*. Brest, France, Sept. 2010, pp. 444–448.

[18] S. Abu-Surra, D. DeClercq, D. Divsalar, and W. E. Ryan, "Trapping set enumerators for specific LDPC codes," in *Proc. Inform. Theory and App. Workshop*. La Jolla, CA, USA, Jan. 2010.

[19] E. R. Berlekamp, R. J. McEliece, and H. C. A. van Tilborg, "On the inherent intractability of certain coding problems," *IEEE Trans. Inform. Theory*, vol. 24, no. 3, pp. 384–386, May 1978.

[20] A. Vardy, "The intractability of computing the minimum distance of a code," *IEEE Trans. Inform. Theory*, vol. 43, no. 6, pp. 1757–1766, Nov. 1997.

[21] K. Yang and T. Helleseth, "On the minimum distance of array codes as LDPC codes," *IEEE Trans. Inform. Theory*, vol. 49, no. 12, pp. 3268–3271, Dec. 2003.

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

[23] K. Krishnan and P. Shankar, "On the complexity of finding stopping set size in Tanner graphs," in *Proc. 40th Conf. Inform. Sciences and Systems*. Princeton, NJ, Mar. 2006.

[24] C.-C. Wang, S. R. Kulkarni, and H. V. Poor, "Finding all small error-prone substructures in LDPC codes," *IEEE Trans. Inform. Theory*, vol. 55, no. 5, pp. 1976–1998, May 2009.

[25] A. McGregor and O. Milenkovic, "On the hardness of approximating stopping and trapping sets," *IEEE Trans. Inform. Theory*, vol. 56, no. 4, pp. 1640–1650, Apr. 2010.

[26] E. Rosnes and Ø. Ytrehus, "An efficient algorithm to find all small-size stopping sets of low-density parity-check matrices," *IEEE Trans. Inform. Theory*, vol. 55, no. 9, pp. 4167–4178, Sept. 2009.

[27] S. K. Chilappagari, S. Sankaranarayanan, and B. Vasic, "Error floors of LDPC codes on the binary symmetric channel," in *Proc. IEEE Int'l. Conf. on Commun.* Istanbul, Turkey, Jun. 2006, pp. 1089–1094.

[28] C. Schlegel and S. Zhang, "On the dynamics of the error floor behavior in regular LDPC codes," in *Proc. IEEE Inform. Theory Workshop*. Taormina, Sicily, Italy, Oct. 2009, pp. 243–247.

[29] S. Ländner and O. Milenkovic, "Algorithmic and combinatorial analysis of trapping sets in structured LDPC codes," in *Proc. 2005 Int'l. Conf. Wireless Networks, Commun., Mobile Comp.*, Jun. 2005, pp. 630–635.

[30] H. Pishro-Nik and F. Fekri, "Improved decoding algorithms for low-density parity-check codes," in *Proc. 3rd Int'l. Symp. Turbo Codes and Related Topics*. Brest, France, Aug. 2003.

[31] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. 20, no. 2, pp. 284–287, Mar. 1974.

[32] X. Y. Hu, M. P. C. Fossorier, and E. Eleftheriou, "On the computation of the minimum distance of low-density parity-check codes," in *Proc. IEEE Int'l. Conf. on Commun.* Paris, France, Jun. 2004, pp. 767–771.

[33] X. Hu, M. Fossorier, and E. Eleftheriou, "Regular and irregular progressive edge-growth Tanner graph," *IEEE Trans. Inform. Theory*, vol. 51, no. 1, pp. 386–398, Jan. 2005.

[34] R. Tanner, D. Sridhara, A. Sridharan, T. Fuja, and D. Costello, "LDPC block and convolutional codes based on circulant matrices," *IEEE Trans. Inform. Theory*, vol. 50, no. 12, pp. 2966–2984, Dec. 2004.

[35] G. A. Margulis, "Explicit constructions of graphs without short cycles," *Combinatorica*, vol. 2, no. 1, pp. 71–78, 1982.

[36] D. Burshtein, "On the error correction of regular LDPC codes using the flipping algorithm," *IEEE Trans. Inform. Theory*, vol. 54, no. 2, pp. 517–530, Feb. 2008.

[37] Z. Zhang, L. Dolecek, B. Nikolic, V. Anantharam, and M. Wainwright, "Investigation of error floors of structured low-density parity-check codes via hardware simulation," in *Proc. IEEE Glob. Telecom. Conf.* San Francisco, CA, Oct.-Nov. 2006.

[38] *ILOG CPLEX 10.2 User's Manual*. Sunnyvale, CA: ILOG SA, 2007.

[39] D. J. C. MacKay, "Encyclopedia of spase graph codes," [Online]. Available: http://www.inference.phy.cam.ac.uk/mackay/codes/data.html.

[40] M. Karimi and A. H. Banihashemi, "An efficient algorithm for finding dominant trapping sets of LDPC codes," Aug. 2011, preprint-arXiv:1108.4478v1.

[41] T. J. Richardson and R. L. Urbanke, "The capacity of low-density parity-check codes under message-passing decoding," *IEEE Trans. Inform. Theory*, vol. 47, no. 2, pp. 599–618, Feb. 2001.

[42] H. Jin, A. Khandekar, and R. McEliece, "Irregular repeat-accumulate codes," in *Proc. 2nd Int'l. Symp. Turbo Codes and Related Topics*. Brest, France, Sept. 2000, pp. 1–8.