Finding All Small Error-Prone Substructures in LDPC Codes

Chih-Chun Wang, *Member, IEEE*, Sanjeev R. Kulkarni, *Fellow, IEEE*, and H. Vincent Poor, *Fellow, IEEE*

***Abstract*—It is proven in this work that it is NP-complete to exhaustively enumerate small error-prone substructures in arbitrary, finite-length low-density parity-check (LDPC) codes. Two error-prone patterns of interest include stopping sets for binary erasure channels (BECs) and trapping sets for general memoryless symmetric channels. Despite the provable hardness of the problem, this work provides an exhaustive enumeration algorithm that is computationally affordable when applied to codes of practical short lengths $n \approx 500$. By exploiting the sparse connectivity of LDPC codes, the stopping sets of size ≤ 13 and the trapping sets of size ≤ 11 can be exhaustively enumerated. The central theorem behind the proposed algorithm is a new provably tight *upper bound* on the error rates of iterative decoding over BECs. Based on a tree-pruning technique, this upper bound can be iteratively sharpened until its asymptotic order equals that of the error floor. This feature distinguishes the proposed algorithm from existing non-exhaustive ones that correspond to finding *lower bounds* of the error floor. The upper bound also provides a worst case performance guarantee that is crucial to optimizing LDPC codes when the target error rate is beyond the reach of Monte Carlo simulation. Numerical experiments on both randomly and algebraically constructed LDPC codes demonstrate the efficiency of the search algorithm and its significant value for finite-length code optimization.**

***Index Terms*—Branch-and-bound, error floors, exhaustive search, low-density parity-check (LDPC) codes, stopping distance, stopping/trapping sets, support trees.**

I. Introduction

A. Bad Substructures of LDPC Codes

ASSUMING iterative decoding [1], the error floor performance of arbitrary, fixed, finite-length low-density parity-check (LDPC) codes [2], [3] is dominated by the bad substructures residing in the code of interest. For binary erasure channels (BECs), the error-prone patterns have been explicitly characterized and termed the stopping sets [4]. The stopping sets not only determine the error floor performance (of BECs) but also determine the entire bit-error rate (BER) and the frame-error rate (FER) curves of any LDPC code [4], [5]. For general memoryless binary-input/symmetric-output channels, the dominating error-prone patterns are named differently according to the associated analytical techniques; these include trapping sets [6], near-codewords [7], pseudocodewords [8], and instantons [9].

Due to the prohibitive cost of computing the entire stopping set distribution [10] and the even greater expense of exhaustively enumerating all trapping sets, in practice, the performance of an LDPC code is determined in two separate steps. Given one fixed LDPC code, the waterfall threshold of the BER is approximated using density evolution [11], [12] with the corresponding scaling laws [13] taking care of finite-code-length effects, and then pinpointed by Monte Carlo simulation. The error floor is lower-bounded by identifying most of the dominant small stopping sets [14], [6], by importance sampling [15], or by the instanton analysis [9], [16]. For the case of a random finite-length code ensemble (not restricted to one given code), and for the simplest nontrivial setting of BECs, many insightful results have been obtained including the averaged performance [4], the corresponding waterfall scaling law [13], and the ensemble error floor analysis [17], [18]. Other research related to stopping set analysis includes results on the asymptotic stopping set weight spectrum [19], the Hamming distance spectrum [20], [21], and the stopping redundancy [22].

B. Determining the Minimal Distances

The minimal stopping distance is defined as the minimal size of all nonempty stopping sets. As the minimal Hamming distance is related to the maximum *a posteriori* probability (MAP) detector, the minimal stopping distance is the dominating factor determining code performance for iterative decoding over BECs in the low-error-rate regime. Recently, determining the minimal stopping distance for an arbitrary binary linear code has been shown to be an NP-complete problem [23]. Namely, there is little chance that a deterministic algorithm can be devised with polynomial computational complexity with respect to the codeword length n. Although not explicitly stated in [23], it follows that the task of determining the minimal stopping distance for LDPC codes with *sparse* parity matrices is still NP-complete regardless of the sparsity constraint on the codes of interest. By noting that for large LDPC codes, most small stopping sets are also valid codewords due to the combinatorial bias of random construction [18], this NP-completeness result is not totally unexpected since deciding the minimal Hamming distance of any arbitrary linear code is a classic NP-complete problem of information theory [24]–[26].

Manuscript received September 05, 2006; revised February 01, 2008. Current version published April 22, 2009. This work was supported in part by the National Science Foundation under Grants ANI-03-58807 and CNS-06-25657. The material in this paper was presented in part at the IEEE International Symposium on Information Theory (ISIT), Seattle, WA, July 2006, and at the IEEE International Symposium on Information Theory (ISIT), Nice, France, June 2007.

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: chihw@purdue.edu).

S. R. Kulkarni and H. V. Poor are with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA (e-mail: kulkarni@princeton.edu; poor@princeton.edu).

Communicated by T. Richardson, Associate Editor for Coding Theory.

Color versions of Figures 2, 8–13 in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIT.2009.2015993

It is worth pointing out that the NP-completeness result focuses on computer programs that take *any arbitrary* LDPC code as input, and output the corresponding minimal stopping distance. If one is willing to sacrifice the ability to accept *arbitrary* codes as input and limit the codes of interest to have some special structure such as the Hamming codes, the circulant-matrix-based LDPC codes, the convolutional LDPC codes, etc., then the minimal stopping/Hamming distances can be determined efficiently and sometimes analytically through combinatorial or algebraic analyses [27]–[31]. Until the present, there has been no combinatorial or algebraic method for determining the minimal distance of an LDPC code constructed arbitrarily from a given degree distribution, which is arguably the most important class of LDPC codes.

Suppose we are willing to further sacrifice some precision and are interested only in an upper bound on the minimal stopping distance. Since the minimal stopping distance is defined as the minimal size of all nonempty stopping sets, upper bounds on the minimal stopping distance can be obtained by explicitly identifying *any* single stopping set of small size. This task is generally achieved efficiently via the error impulse methods [32], [33]. The efficiency of computing upper bounds motivates the following question: can a nontrivial lower bound on the minimal stopping distance be easily obtained so that the minimal distances can be bracketed efficiently? Unfortunately, few meaningful lower bounds on the minimal stopping distance exist. The lack of progress is essentially due to the aforementioned NP-hardness of the problem.

The challenges of determining the low-error-rate performance of iterative decoding persist for non-erasure channels as well. For non-erasure channels, the minimal trapping distance can be defined analogously as the minimal size of all nonempty trapping sets. In Section III-B, it is proven that the problem of deciding the minimal trapping distance is also NP-complete. In other words, determining the asymptotic low-error-rate performance of the iterative detector (for BECs and for non-erasure channels) is not easier than determining the asymptotic performance of the optimal MAP detector. The NP-hardness of this problem is one of the reasons that all existing finite-code error-floor optimization schemes employ *approximations* or *guided heuristics* as objective functions. Some such examples include the girth of the Tanner graph [34], [35], the Approximate Cycle Extrinsic (ACE) message degree [36], partial stopping set elimination [37], and ensemble-inspired upper bounds [38]. With already significant success based on these indirect metrics, one would expect greater improvement if the minimal stopping/trapping distance can be computed and directly employed during code optimization. One purpose of this paper is to serve as a starting foundation for future research on efficient exhaustive search algorithms for error-prone substructures, which are critical to finite-length code optimization.

Other recent research related to the determination of the minimal Hamming distance, the pseudocodeword weight, and the girth of the Tanner graph can be found in [39]–[44]. Ensemble-based FER bounds for MAP decoders are explored in [45], [46], and the references therein. It is worth noting that deciding the girth of the Tanner graph is *not* an NP-hard problem and can be easily done with complexity $\mathcal{O}(n^2)$.

C. Simple Complexity Comparisons

A simple analysis of the complexity of deciding the minimal stopping/trapping distances (or equivalently, the complexity of exhaustively enumerating small stopping/trapping sets) is discussed as follows. Consider the problem of exhaustively enumerating all minimum stopping sets[1] of size $\leq t$ in a code of length n. A brute-force method must examine all $\sum_{\tau \leq t} \binom{n}{\tau}$ subsets of size $\tau \leq t$. Let d_v and d_c denote the maximal variable and check node degrees in the corresponding Tanner graph. A smarter search achieves $O(n(d_c-1)^{(d_v-1)^t})$ when the girth of the corresponding Tanner graph is larger than $2t$. Namely, we need only to consider the depth-$2t$ support tree rooted at each variable node (see [14] for details). Both the brute-force and the smarter methods show that this problem is *fixed-parameter tractable* with respect to t. On the other hand, when n is fixed, the complexity is bounded by $O(2^n)$ and does not grow with respect to t. So the problem is also fixed-parameter tractable with respect to n. Nonetheless, for practical values of n, t, d_v, and d_c (say $n \geq 500$, $t \geq 10$, $d_v \geq 6$, $d_c \geq 6$) the complexity quickly becomes intractable for any of the above schemes. In this work, instead of improving the asymptotic complexity for large values of n and t, we are interested in developing efficient algorithms that can handle parameters of practical ranges. Further discussion on the NP-completeness is relegated to Section III.

An algorithm that is capable of solving problems with small-to-moderate values of n and t is of practical importance since many applications *use* codes of length $n < 1000$. By taking advantages of the sparse structure of LDPC codes, we are able to enumerate exhaustively all stopping sets of size ≤ 13 for LDPC codes of length $n = 512$ and thus decide the minimal stopping distances if they are ≤ 13. The closed-form complexity of the proposed algorithm is difficult to characterize due to its convoluted nature. For comparison, with $n = 512$ and $t = 13$ the aforementioned brute-force search requires $\binom{512}{13} \approx 2.5 \times 10^{25}$ trials, which demonstrates the efficiency of the proposed scheme. The smarter depth-$2t$ tree approach in [14] results in similar complexity for such values of n and t. The proposed stopping set exhaustive search algorithm will later be generalized for trapping sets as well. All trapping sets of size ≤ 11 can then be exhaustively enumerated for $n = 512$ codes.

D. Main Applications of Our Results

In addition to its computation-theoretic interest, three major applications of the proposed algorithm are listed as follows.

1) For BECs, an exhaustive list of minimum stopping sets can be used to obtain a BER/FER lower bound that is tight in both *order* and *multiplicity*. For non-erasure channels, the exhaustive list of minimum trapping sets can be used to derive lower bounds tightly predicting the error floor performance [6].
2) A central component of the algorithm is a novel BER/FER upper bound for any fixed arbitrary codes over BECs.

[1]To be consistent with graph-theoretic terminology, we will use the terms minimum stopping set and minimal stopping set to denote, respectively, stopping sets that are of minimal size and those which have no sub stopping sets. Further discussion of this terminology is included in Section IV-A.

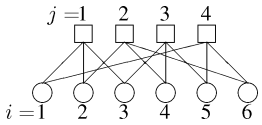


Fig. 1. A simple parity-check code C_1.

Based on a support-tree-pruning technique, this upper bound can be iteratively sharpened until its asymptotic order equals that of the error floor. This thus provides a worst performance guarantee for BECs with arbitrarily low erasure probability.

3) Using the exhaustive list of minimum bad substructures as an objective function, one can optimize the error floor of any arbitrary finite-length code. The error-floor performance can be guaranteed without any combinatorial outlier, a weakness of randomized constructions that was first pointed out in [6]. Further implementation of the code optimization algorithm is discussed in a companion paper [18].

The remainder of this paper is organized as follows. This paper focuses on exhaustive search algorithms for two different types of substructures: the stopping sets and the trapping sets. Basic definitions of these error-prone patterns are given in Section II, and the NP-completeness of exhaustively enumerating minimum trapping sets is proven in Section III. We then derive a tree-based algorithm for computing a BER upper bound for BECs in Section IV, a central component of the proposed exhaustive search algorithm. Inspired by properties of stopping sets and by the corresponding Boolean expression framework, several methods to further improve the efficiency of the algorithm are discussed in Section V. The stopping set exhaustive search algorithm is generalized for trapping sets in Section VI. Section VII contains numerical experiments, including results for the $(23, 12, 7)$ Golay code, the $(155, 64, 20)$ Tanner code [29], the Ramanujan–Margulis ($q = 13$, $p = 5$) code [47], the Margulis code ($p = 7$) [48], and two rate-$1/2$, $n = 576$ lifted LDPC codes from the companion paper [18]. Section VIII concludes this paper.

II. Preliminary and Formulation

A. Code Representation

For fixed $n \in \mathbb{N}$, let $\boldsymbol{x} = (x_1, \ldots, x_n) \in \{0,1\}^n$ denote a transmitted vector, for which n is the codeword length. The receiving signal vector, denoted by $\boldsymbol{y} = (y_1, \ldots, y_n)$, is the image of the transmitted $\boldsymbol{x}$ corrupted by memoryless stationary noise: $\mathsf{P}(\boldsymbol{y}|\boldsymbol{x}) = \prod_{i=1}^{n} \mathsf{P}(y_i|x_i)$. One particular type of channel of interest is the BEC in which we use e to represent the erasure symbol and the erasure probability is $\epsilon = \mathsf{P}(y = e|x)$.

Only linear codes are considered. We use $C = \{\boldsymbol{x}\} \subseteq \{0,1\}^n$ to denote the codebook and use $\boldsymbol{H}$ to denote the corresponding $m \times n$ parity-check matrix $\boldsymbol{H}$. We then have $\boldsymbol{H}\boldsymbol{x}^{\mathrm{T}} = \boldsymbol{0}$ for all $\boldsymbol{x} \in C$ assuming $\boldsymbol{x}$ is a row vector. The $\boldsymbol{H}$ chosen to represent C is not unique. As all our results depend on the choice of $\boldsymbol{H}$, the code C sometimes refers to the corresponding parity-check matrix $\boldsymbol{H}$ rather than to the collection of valid codewords $\{\boldsymbol{x}\}$.

By viewing the matrix $\boldsymbol{H}$ as an incidence matrix of a bipartite graph containing two sets of vertices: the variable nodes $\{v_1, \ldots, v_n\}$ and the check nodes $\{c_1, \ldots, c_m\}$, code C can be expressed in its Tanner graph representation such that $H_{ji} = 1$ if and only if there is an undirected edge connecting (c_j, v_i). For example, a simple parity-check code C_1 with

$$\boldsymbol{H} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

can be mapped bijectively to a bipartite graph as in Fig. 1. The coded bits $\{x_1, \ldots, x_n\}$ and the variable nodes $\{v_1, \ldots, v_n\}$ are generally used interchangeably in the Tanner graph representation. However, in the most stringent definitions, $\{x_1, \ldots, x_n\}$ refers to the "bits" of the codeword vector $\boldsymbol{x}$ satisfying $\boldsymbol{H}\boldsymbol{x}^{\mathrm{T}} = \boldsymbol{0}$ while $\{v_1, \ldots, v_n\}$ refers to the "nodes" in the corresponding Tanner graph.

In this paper, we use the most general definition of LDPC codes, namely, a family of linear codes such that the number of ones in $\boldsymbol{H}$ is of the order of n when n tends to infinity. Although our results are derived with the application to LDPC codes in mind, they hold for arbitrary linear codes as well.

B. Stopping Sets and Trapping Sets

1) Stopping Sets: A stopping set is a subset of $\{v_1, \ldots, v_n\}$ such that the induced subgraph contains no check nodes of degree one. Using code C_1 in Fig. 1 as an example, both $\{v_2, v_3, v_4\}$ and $\{v_1, v_2, v_4, v_5\}$ are stopping sets for code C_1. The minimal stopping distance d_{S} is defined as the minimal size of all nonempty stopping sets. Since the collection of the value-1 bits in any codeword is a stopping set, we have $d_{\mathrm{S}} \leq d_{\mathrm{H}}$, the minimal Hamming distance.

Di *et al.* in [4] showed that for BECs, a belief propagation iterative decoder [49] fails if and only if the set of erased bits contains a stopping set. As a result, for small erasure probability ϵ, the FER performance is determined by the minimal stopping distance d_{S} and the corresponding multiplicity m_{S}, the latter of which is defined as the number of distinct stopping sets of size d_{S}.

2) Trapping Sets: The notion of trapping sets stems from an operational definition in which a trapping set originally refers to a subset of $\{x_1, \ldots, x_n\}$ that is susceptible to errors under iterative decoding. Namely, if the observations on these nodes are misleading, it is "highly unlikely" that the extrinsic messages will correct any of the nodes. The errors are "trapped" within these nodes [6]. It is a concept depending on both the underlying channel model and the decoding algorithm. For example, a stopping set is legitimately a trapping set when the underlying channel model is the BEC and the belief propagation decoder is employed. For non-erasure channels, empirically, almost all trapping sets are *near-codewords* [7]. That is, almost all trapping sets are subsets of $\{v_1, \ldots, v_n\}$ such that the induced subgraph contains only a "limited" number of check nodes of odd degree and is thus "nearly" a codeword. A near-codeword can be categorized by two numbers, the total number of variable nodes involved and the number of check nodes of odd degree in the induced graph. For example, a $(12, 4)$ near-codeword refers to a set of 12 variable nodes for which there are four check nodes of odd degree in the induced subgraph.

To better facilitate our discussion, we redefine the trapping set from a graph-theoretic perspective, which is independent of the underlying channel model and the decoding algorithm:

Definition 1 (k-Out Trapping Sets): A subset of $\{v_1, \ldots, v_n\}$ is a k-out trapping set if in the induced subgraph, there are exactly k check nodes of degree one.

A similar definition is the "elementary trapping set" introduced in [31], which is a special case of the k-out trapping set defined herein.

By the above definition, every stopping set is a 0-out trapping set. $\{v_1, v_2, v_3, v_4\}$ in C_1 of Fig. 1 is a 1-out trapping set. It is worth noting that the definition of a k-out trapping set is slightly different from the definition of near-codewords. The near-codeword focuses on the total number of check nodes of *odd degree*, including those with degree 1 and those with degrees ≥ 3, and emphasizes its link to valid codewords. The k-out trapping set considers a pattern resembling the stopping sets but has k edges connecting to check nodes of *degree* 1. These degree-1 check nodes are able to receive correct extrinsic messages to recover the contaminated bit values. In addition to the k check nodes of degree 1, there might be more check nodes of odd degrees ≥ 3 within a k-out trapping set, which makes the definition different than that of an (s, k) near-codeword. This slight relaxation does not affect the generality of our exhaustive search algorithm for the following two reasons. First, for any set of s variable nodes, an (s, t) near-codeword must, by definition, be a k-out trapping set for some $k \leq t$. As a result, any attempt to enumerate exhaustively all k-out trapping sets of sizes $s \leq s_0$ and $k \leq t_0$ can also serve the purpose of exhaustively enumerating all (s, t) near-codewords for $s \leq s_0$ and $t \leq t_0$. Second, for a randomly constructed code of large length n, almost all small stopping sets are valid codewords and almost all small k-out trapping sets of size s are (s, k) near-codewords due to some combinatorial bias discussed in [18]. In our numerical examples, all minimum k-out trapping sets are (s, k) near-codewords.

One can similarly define the minimal k-out trapping distance $d_{\mathrm{T},k}$ as the minimal size of all nonempty k-out trapping sets.

III. NP-Completeness

A major motivation of this work springs from the fact that the inherent NP-hardness of the minimal stopping/trapping distance problems describes only their asymptotic worst case complexity with large codeword length n, which does not preclude efficient algorithms for shorter codes. However, it is still worth investigating the asymptotic behavior, as it provides valuable insight into the intrinsic hardness of exhaustively enumerating minimum error-prone substructures.

A. Existing Results

The designation NP refers to those problems that are guaranteed to be solvable by a fictional (N)ondeterministic Turing machine within (P)olynomial time with respect to the input size. For example, if the input of the problem is a general linear code, then $\mathcal{O}(n^2)$ bits are required to describe the problem. If the input of the problem is an LDPC code of length n, then in general $\mathcal{O}(n \log(n))$ bits are needed to describe the LDPC code structure. The polynomial termination time is taken with respect to those input sizes (in bits) rather than the codeword length n. More rigorous definitions/discussions of these concepts can be found in [50].

NP-complete problems are the hardest non-deterministically polynomial problems in the sense that the NP = P question is equivalent to asking whether any NP-complete problem can be solved with complexity polynomial in the size of the problem. It is generally believed that solving NP complete problems requires exponential complexity. Some known NP-complete problems in the coding and information theory literature are the following.

1) *Minimal Hamming Distance:* A binary decision problem is a special type of problem in which a yes/no question is proposed and the algorithm/solver outputs 1 or 0 depending on whether the answer to the proposed question is positive or negative. We consider the following decision problem: given as input an arbitrary parity-check matrix $\boldsymbol{H}$ and an integer t, the decision problem MIN-DISTANCE($\boldsymbol{H}, t$) outputs 1 if and only if (iff) $d_{\mathrm{H}} \leq t$, in which d_{H} is the minimal Hamming distance of code C specified by $\boldsymbol{H}$. It has been shown in [24]–[26] that MIN-DISTANCE($\boldsymbol{H}, t$) is NP-complete (with respect to the input size $\mathcal{O}(n^2 + \log(t))$). The question whether $d_{\mathrm{H}} \leq t$ is hard to answer for large $\boldsymbol{H}$ and large t.

Note: One can easily show that MIN-DISTANCE($\boldsymbol{H}, t$) is of similar complexity to the problem of directly computing d_{H} given $\boldsymbol{H}$,[2] while the simpler nature of the Yes/No question facilitates the complexity analysis.

2) *Minimal Stopping Distance:* Given as input an arbitrary parity-check matrix $\boldsymbol{H}$ and an integer t, the decision problem MIN-DISTANCE($\boldsymbol{H}, t$) outputs 1 iff $d_{\mathrm{S}} \leq t$, in which d_{S} is the minimal stopping distance of code C specified by $\boldsymbol{H}$. For notational simplicity, we use SD($\boldsymbol{H}, t$) as shorthand. It has been shown in [23] that SD($\boldsymbol{H}, t$) is also an NP-complete problem.

Sometimes a subset of an NP-complete problem, defined by restricting the cases considered, can be polynomially solvable. For example, the SAT problem is polynomially solvable when restricted to 2-SAT. The decision problem SD($\boldsymbol{H}_{\mathrm{sparse}}, t$), where $\boldsymbol{H}_{\mathrm{sparse}}$ is restricted to that of an LDPC code, may have different complexity than the original SD($\boldsymbol{H}, t$) due to the additional sparsity constraint on $\boldsymbol{H}$. Nonetheless, in [23], the NP-completeness of SD($\boldsymbol{H}, t$) is proven by reduction from the classic NP-complete VERTEX-COVER problem to SD($\boldsymbol{H}, t$) in a way that any vertex cover for the original graph $G = (V, E)$ is mapped bijectively to a stopping set for $\boldsymbol{H}$. By carefully inspecting the construction of $\boldsymbol{H}$ from G in [23], we notice that there are $|E| + |V|(|E| + 1)$ rows and $2|E| - 1 + |V||E|$ columns in $\boldsymbol{H}$ and there are $2|E||V| + 3|E|$ nonzero entries in $\boldsymbol{H}$. Therefore, $\boldsymbol{H}$ is a sparse matrix. As a result, the same reduction proposed in [23] can be used to prove that SD($\boldsymbol{H}_{\mathrm{sparse}}, t$) is an NP-complete problem regardless of such a sparsity constraint.

[2]There are instances in which the decision problem is strictly easier than direct computation, for example, the primality test problem versus the factorization problem.

B. Minimal k-Out Trapping Distance

Given as input an arbitrary parity-check matrix $\boldsymbol{H}$ and an integer t, the decision problem k-OUT-TRAPPING-DISTANCE $(\boldsymbol{H}, t)$ outputs 1 iff $d_{\mathrm{T},k} \leq t$, in which $d_{\mathrm{T},k}$ is the minimal k-out trapping distance of the parity-check code C specified by $\boldsymbol{H}$. For notational simplicity, we use k-OTD$(\boldsymbol{H}, t)$ as shorthand. Note that unlike $\boldsymbol{H}$ and t, k is not part of the input. We can then prove the following theorem.

Theorem 1: For any fixed integer k, the decision problem k-OTD$(\boldsymbol{H}, t)$ is NP-complete.

Proof: It is easy to show that k-OTD$(\boldsymbol{H}, t)$ is an NP problem since a nondeterministic Turing machine can choose any one of the nonempty minimum k-out trapping sets and verify whether its size is $\leq t$ in polynomial time with respect to the input size $\mathcal{O}(n^2 + \log(t))$. The NP-completeness proof is based on the polynomial time reduction from SD$(\boldsymbol{H}, t)$ to k-OTD$(\boldsymbol{H}, t)$ as described in Algorithm 1 and illustrated in Fig. 2.

Algorithm 1 Reduction from SD$(\boldsymbol{H}, t)$ to k-OTD$(\boldsymbol{H}, t)$

1: **Input:** the parity-check matrix $\boldsymbol{H}$ and t.

2: Construct the corresponding bipartite graph G from $\boldsymbol{H}$.

3: Construct G' by duplicating $(k+2)$ copies of G, and denote the $(k+2)n$ variable nodes as $\{v_{i,l} : \forall i = 1, \ldots, n, \forall l = 1, \ldots, (k+2)\}$.

4: **for** $i = 1$ to n **do**

5: Form a $(k+2)$-clique among all the variable node copies $\{v_{i,l}\}_l$ by adding $(k+1)(k+2)/2$ new edges.

6: Break each new edge into two by inserting a new check nodes of degree 2 in between.

7: **end for**

8: Denote the final graph as G''.

9: **for** $i = 1$ to n **do**

10: Construct $G^{(i)}$ from G'' by adding k check nodes of degree 1, and connecting them to $v_{i,1}$.

11: Let $\boldsymbol{H}^{(i)}$ denote the parity check matrix of $G^{(i)}$.

12: **end for**

13: **Output:** If k-OTD$(\boldsymbol{H}^{(i)}, t(k+2))$ outputs 1 for any i, then output 1. Otherwise, output 0.

The polynomial running time of Algorithm 1 can be easily proved by inspection. So it remains to prove the correctness of the reduction.

Without loss of generality, suppose $\{v_1, \ldots, v_D\}$ is a stopping set of size D for $\boldsymbol{H}$. Then $\{v_{1,l}, \ldots, v_{D,l} : l = 1, \ldots, k+2\}$ is a k-out trapping set of size $D(k+2)$ for $\boldsymbol{H}^{(1)}$, where the k extrinsic edges are connected to $v_{1,1}$. Therefore, SD$(\boldsymbol{H}, t) = 1$ implies that the output of Algorithm 1 is positive.

Without loss of generality, suppose k-OTD$(\boldsymbol{H}^{(1)}, t(k+2))$ outputs 1, namely, there exists a subset $\boldsymbol{v} \subseteq \{v_{i,l} : i = 1, \ldots, n, l = 1, \ldots, k+2\}$ such that $\boldsymbol{v}$ is a k-out trapping set of size $\leq t(k+2)$ for $\boldsymbol{H}^{(1)}$. We first show that if $v_{i_0,l_0} \in \boldsymbol{v}$ for some i_0 and l_0, then $v_{i_0,l} \in \boldsymbol{v}$ for all $l = 1, \ldots, k+2$. Suppose not. Define the following two nonempty sets: $\boldsymbol{v}_0 = \{v_{i_0,l} : \forall l\} \cap \boldsymbol{v}$ and $\boldsymbol{v}_1 = \{v_{i_0,l} : \forall l\} \backslash \boldsymbol{v}$. Since jointly $\boldsymbol{v}_0$ and $\boldsymbol{v}_1$ form a clique (ignoring the degree-2 check nodes), by the min-cut max-flow theorem, there are $\geq k+1$ edges connecting $\boldsymbol{v}_0$ and $\boldsymbol{v}_1$. All the degree-2 check nodes corresponding to those edges connecting $\boldsymbol{v}_0$ and $\boldsymbol{v}_1$ will be of degree 1 in the subgraph induced by $\boldsymbol{v}$. Therefore, $\boldsymbol{v}$ will induce at least $k+1$ degree-1 check nodes, which contradicts that $\boldsymbol{v}$ is a k-out trapping set. Therefore, $v_{i_0,l_0} \in \boldsymbol{v}$ implies $v_{i_0,l} \in \boldsymbol{v}$ for all $l = 1, \ldots, k+2$.

By the previous analysis, we can safely assume $\boldsymbol{v} = \{v_{i_1,l}, \ldots, v_{i_D,l} : \forall l\}$ for some D. We then show that $\{v_{i_1}, \ldots, v_{i_D}\}$ must be a stopping set for the input $\boldsymbol{H}$. Suppose not, say there are $\kappa \geq 1$ check nodes of degree 1 induced by $\{v_{i_1}, \ldots, v_{i_D}\}$. Since $\boldsymbol{v}$ contains $(k+2)$ parallel copies of $\{v_{i_1}, \ldots, v_{i_D}\}$, there are at least $(k+2)\kappa$ check nodes of degree 1, which contradicts that $\boldsymbol{v}$ is a k-out trapping set. Therefore , $\{v_{i_1}, \ldots, v_{i_D}\}$ must be a stopping set. Since $D(k+2) \leq t(k+2)$, we have $D \leq t$. The output of Algorithm 1 being one implies that SD$(\boldsymbol{H}, t) = 1$. The proof is complete. □

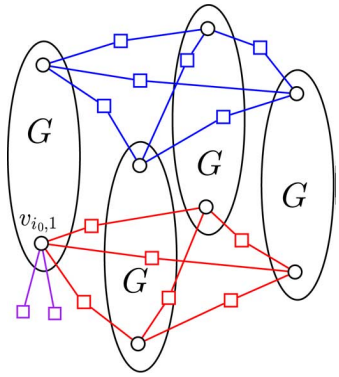


Fig. 2. Illustration of $G^{(i_0)}$ Algorithm 1, in which G is duplicated $k+2$ times $(k = 2)$. Cliques are formed among $\{v_{i,l}\}_l$ and k degree-1 check nodes are added to $v_{i_0,1}$.

Remark: The polynomial time reduction does not assume any underlying structure of the input code. Since for sparse input matrices, SD$(\boldsymbol{H}_{\mathrm{sparse}}, t)$ is still NP-complete, the same reduction in Algorithm 1 shows that even with a sparsity constraint on $\boldsymbol{H}_{\mathrm{sparse}}$, the decision problem k-OTD$(\boldsymbol{H}_{\mathrm{sparse}}, t)$ is still NP-complete.

IV. Searching for the Minimum Stopping Sets

In this section, we focus mainly on exhaustively searching for the minimum stopping sets while the generalization for minimum k-out trapping sets will be addressed in Section VI-D.

The fact that the iterative decoder fails when the erased bits contain a stopping set implies that each stopping set $\boldsymbol{v}_s$ corresponds to a lower bound $\epsilon^{|\boldsymbol{v}_s|}$ on the FER where ϵ is the erasure probability. Exhaustively enumerating stopping sets of sizes smaller than or equal to K is thus equivalent to identifying (or eliminating, if there is no such stopping set)

possible existence of lower bounds of order $\mathcal{O}\left(\epsilon^{|K|}\right)$. On the other hand, an alternative way of eliminating possible FER lower bounds of order $\mathcal{O}\left(\epsilon^{|K|}\right)$ is to construct an upper bound $\mathcal{O}\left(\epsilon^{|K+1|}\right)$ if such an upper bound exists. Sharing the same effects as eliminating possible lower bounds, in the broadest sense, any FER upper bound can be viewed as an exhaustive search algorithm for small stopping sets and *vice versa*. For the following, we construct an algorithm for computing upper bounds on the error probability for arbitrary parity-check codes over BECs, which later leads to an efficient minimum stopping set exhaustive search algorithm in Section V-C.

A. The Boolean Expression Framework

Without loss of generality and for notational simplicity, we assume that the all-zero codeword $\boldsymbol{x} = \mathbf{0}$ is transmitted. With this assumption and for the BEC, a decoding algorithm for bit x_i, $i \in \{1, \ldots, n\}$ is a function outputting either 0 or e, the former of which represents x_i being successfully decoded and the latter of which means the bitwise decoder fails. More explicitly, a decoder for bit x_i is a function $g_i : \{0, e\}^n \mapsto \{0, e\}$ with output $\hat{x}_i = g_i(\boldsymbol{y})$, where $\boldsymbol{y} \in \{0, e\}^n$ is the received signal vector. In this paper, we use the function $f_{i,l}, \forall i \in \{1, \ldots, n\}$, to denote the iterative decoder for bit x_i after l iterations. And we use $f_i = \lim_{l\to\infty} f_{i,l}$ to denote the end result after iterative decoding stops improving with more iterations. If we further relabel the element "e" by "1," f_i and $f_{i,l}$ become Boolean functions.

One advantage of this Boolean expression is that the iterative message map at the variable node becomes "$\bullet$," the binary AND operation, and the iterative message map at the parity-check node becomes "$\vee$," the binary OR operation. If we define the expectation of a Boolean function as its probability of being one, the BER for bit x_i is simply the expectation $p_i = \mathsf{E}\{f_i(\boldsymbol{y})\}$. This Boolean-expression framework was *implicitly considered* in many papers discussing iterative decoders on BECs. For example, [10] and [51] use the "projection algebra" to characterize directly the expectation of the Boolean functions instead of focusing on the underlying Boolean functions. By focusing directly on the Boolean functions, many new useful implications can be derived.

Take the simple parity-check code C_1 in Fig. 1 as an example. Suppose we further use $f_{i\to j,l}$ to represent the message from variable node v_i to check node c_j during the lth iteration. The iterative decoders $f_{2,l}$, $l \in \{0, 1, 2\}$, for bit x_2 then become

$$\begin{aligned} f_{2,0}(\boldsymbol{y}) &= y_2 \\ f_{2,1}(\boldsymbol{y}) &= y_2(y_1 \vee y_3)(y_4 \vee y_6) \\ f_{2,2}(\boldsymbol{y}) &= y_2\left(y_1(f_{5\to4,1} \vee f_{6\to4,1}) \vee y_3(f_{4\to3,1} \vee f_{5\to3,1})\right) \\ &\quad \cdot \left(y_4(f_{3\to3,1} \vee f_{5\to3,1}) \vee y_6(f_{1\to4,1} \vee f_{5\to4,1})\right) \\ &= y_2\left(y_1(y_5 \vee y_6) \vee y_3(y_4 \vee y_5)\right) \\ &\quad \cdot \left(y_4(y_3 \vee y_5) \vee y_6(y_1 \vee y_5)\right). \end{aligned} \tag{1}$$

As the iteration proceeds, we can derive the Boolean expression for all $f_{2,l}(\boldsymbol{y})$, $l \in \mathbb{N}$. The final decoder of bit x_2 is $f_2 := \lim_{l\to\infty} f_{2,l}$, and in this particular example of C_1, one can verify that $f_2(\boldsymbol{y}) = f_{2,2}(\boldsymbol{y})$ for all $\boldsymbol{y} \in \{0,1\}^n$. Although (1) admits a pleasing nested structure, the repeated appearances of many Boolean input y_i's, also known as short "cycles," pose a great challenge to the evaluation of $p_2 = \mathsf{E}\{f_2(\boldsymbol{y})\}$. One solution is to simplify (1) by expanding the nested structure into a sum–product Boolean expression [10]

$$f_2(\boldsymbol{y}) = f_{2,2}(\boldsymbol{y}) = y_1y_2y_6 \vee y_2y_3y_4 \vee y_1y_2y_4y_5 \vee y_2y_3y_5y_6. \tag{2}$$

$\mathsf{E}\{f_2(\boldsymbol{y})\}$ can then be evaluated by the projection algebra in [10] or equivalently by the inclusion–exclusion principle. For this example, we have $p_2(\epsilon) = 2\epsilon^3 + 2\epsilon^4 - 5\epsilon^5 + 2\epsilon^6$, where $\epsilon = \mathsf{E}\{y_i\}, \forall i$, is the erasure probability. For comparison, with $\epsilon = 0.1$, $p_2(\epsilon) = 2.152 \times 10^{-3}$ while the BER predicted by density evolution after two iterations is $\approx 1.4 \times 10^{-4}$. An order-of-magnitude gap is observed between the actual performance and the prediction by density evolution due to the presence of many short cycles.

Following the convention in graph theory, we use "minimum stopping sets" to refer to those stopping sets of minimal size and use "minimal stopping sets" to describe those stopping sets containing no other nonempty stopping set as its proper subset. We then have the following results.

Proposition 1: In the simplified sum–product form [3] of $f_i(\cdot)$, each monomial (product) term, for example $y_1y_2y_4y_5$ in (2), corresponds to a minimal stopping set involving v_i and *vice versa*.

Proof: We first notice that each monomial term must be a stopping set involving v_i since knowing the values of y_j outside the monomial term does not help decode the value of x_i. Moreover, in the simplified sum–product form, each monomial term cannot be absorbed by another monomial term using a subset of y_j. Therefore, if only a subset of y_j of the monomial term is erased, the decoder can always decode the value of x_i, which implies that the monomial term in the simplified form must correspond to a minimal stopping set.

To prove the converse, we notice that if we set the observation values outside a minimal stopping set to be 0 and the observations within the minimal stopping set to be 1, the decoder will fail and the output of f_i will be 1. Therefore, at least one of the monomial terms uses only y_j contained in a subset of the minimal stopping set. The forward direction that we just proved shows that monomial term must be a stopping set. Since a minimal stopping set cannot contain any proper stopping subset, the monomial must correspond to the minimal stopping set. This completes the proof. □

In general, the number of monomial terms in a simplified sum–product form is exponential with respect to n. If only a small collection of the product terms is identified, say $y_1y_2y_6$ and $y_2y_3y_4$, then a lower bound

$$\mathsf{E}\{f_{\mathrm{LB},2}(\boldsymbol{y})\} = 2\epsilon^3 - \epsilon^5 \le \mathsf{E}\{f_2(\boldsymbol{y})\} = p_2(\epsilon), \ \ \forall \epsilon \in [0,1]$$

can be obtained where

$$f_{\mathrm{LB},2}(\boldsymbol{y}) = y_1y_2y_6 \vee y_2y_3y_4 \le f_2(\boldsymbol{y}), \ \forall \boldsymbol{y} \in \{0,1\}^n.$$

In this work, the inequality $a \le b$ between two Boolean variables is defined over their corresponding real values.[4]

[3] A simplified sum–product form contains the minimal number of product terms.

[4] A more formal definition is $(a \le b) \triangleq (\neg a) \vee b$.

In our example, $\mathsf{E}\{f_{\mathrm{LB},2}(\boldsymbol{y})\} = 2\epsilon^3 - \epsilon^5$ is asymptotically tight and determines the error floor. If any one of the minimum stopping sets, which are $y_1y_2y_6$ and $y_2y_3y_4$ in this example, is not identified, then the resulting lower bound will be loose. The major challenge of this approach of constructing lower bounds is thus to ensure that all minimum stopping sets are exhaustively enumerated. Furthermore, even when all minimum stopping sets are exhaustively enumerated, this lower bound is tight only when the erasure probability ϵ is sufficiently small. Currently, whether ϵ is small enough can be determined only by extrapolating the waterfall region, which is subject to doubt since no analytical derivation can be made. A rigorous upper bound thus becomes essential for a deeper understanding of the error-floor behavior.

An upper bound can be constructed by iteratively computing the sum–product form of $f_{2,l+1}(\boldsymbol{y})$ from that of $f_{2,l}(\boldsymbol{y})$. To counteract the exponential[5] growth rate of the number of product terms with respect to l, during each iteration, we can "relax" and "merge" some of the product terms so that the complexity is kept manageable [10]. For example, $f_{2,2}(\boldsymbol{y})$ in (2) can be relaxed and merged as follows:

$$\begin{aligned} f_{2,2}(\boldsymbol{y}) &= y_1y_2y_6 \vee y_2y_3y_5y_6 \vee y_2y_3y_4 \vee y_1y_2y_4y_5 \\ &\le 1y_2y_6 \vee y_21y_6 \vee y_2y_4 \vee 1y_2y_41 \\ &= y_2y_6 \vee y_2y_4 \\ &\triangleq f_{\mathrm{UB},2} \end{aligned}$$

so that the number of product terms is reduced from four to two. This technique is generally very loose since relaxation destroys too much information. To generate tight upper bounds, the number of relaxation steps must be kept minimal, which yields again an exponential growth rate of the number of product terms. As a result, tight upper bounds constructed by this relax-and-merge approach have been reported only for codes of lengths $n \le 20$ [10].

In contrast, we construct an efficient upper bound $\mathrm{UB}_i \ge \mathsf{E}\{f_i(\boldsymbol{y})\}$ by preserving much of the nested structures in (1) instead of focusing on the simplified sum–product form in (2). Tight upper bounds can be obtained for $n = 300$–500, a significant improvement over existing results. Furthermore, the tightness of our bound can be verified with ease, a feature absent in the relax-and-merge approach. Combined with the lower bound $\mathsf{E}\{f_{\mathrm{LB},i}(\boldsymbol{y})\}$, the finite code performance can be upper- and lower-bounded asymptotically tightly.

B. The Iterative Decoding Functions

Some useful notation is as follows. We use $\boldsymbol{y}_h^k$ to represent a partial segment $(y_h, y_{h+1}, \ldots, y_k)$ from y_h to y_k. With this notation, $\boldsymbol{y} = \boldsymbol{y}_1^{i-1}0\boldsymbol{y}_{i+1}^n$ represents a received vector with the ith value y_i being 0.

Definition 2: For any i, y_i is a *determining variable* of $f(\boldsymbol{y})$ if there exist $\boldsymbol{y}_1^{i-1}$ and $\boldsymbol{y}_{i+1}^n$ such that $f(\boldsymbol{y}_1^{i-1}0\boldsymbol{y}_{i+1}^n) \ne f(\boldsymbol{y}_1^{i-1}1\boldsymbol{y}_{i+1}^n)$.

Definition 3 (Iterative Decoding Functions): A Boolean function $f(\boldsymbol{y})$ is an *iterative decoding* function if $f(\boldsymbol{y})$ can be expressed using only $y_1, \ldots, y_n$, binary AND "$\bullet$" and binary OR "$\vee$" operations without the binary NOT "$\neg$" operation. By the distributiveness of the AND and OR operations, an iterative decoding function can always be simplified as a sum of monomials without using NOT operations.

For example: $y_1 \vee y_2(y_1 \vee y_3)$ is an iterative decoding function while the XOR function $y_1(\neg y_2) \vee (\neg y_1)y_2$ is not. All $f_i(\boldsymbol{y})$, $f_{i,l}(\boldsymbol{y})$, and $f_{i\to j,l}(\boldsymbol{y})$ discussed in the previous section are iterative decoding functions. Before introducing a simple upper bound, we first prove the following two properties of iterative decoding functions.

Proposition 2 (Monotonicity): Suppose $f(\boldsymbol{y})$ is an iterative decoding function. Then $f(\boldsymbol{y})$ is monotonically increasing with respect to each input variable y_i. That is

$$\forall i, \forall \boldsymbol{y}_1^{i-1}, \forall \boldsymbol{y}_{i+1}^n, \quad f(\boldsymbol{y}_1^{i-1}0\boldsymbol{y}_{i+1}^n) \le f(\boldsymbol{y}_1^{i-1}1\boldsymbol{y}_{i+1}^n).$$

Proof: Since each variable node corresponds to an AND and each check node corresponds to an OR, it can be shown that any iterative decoding function can be converted to a parity-check code detection problem assuming iterative decoders. One important property of iterative decoding over BECs is that regardless of whether the underlying graph is cyclic or not, the decoding is monotonic. Namely, knowing one more uncorrupted observation y_i can only improve the decoding result even for codes with cycles. *Proposition 2* is simply a restatement of the monotonicity argument. This property, however, differentiates the iterative decoding functions $f(\boldsymbol{y})$ from arbitrary Boolean functions. □

Proposition 3 (Positive Correlation): The correlation between two iterative decoding functions $f(\boldsymbol{y})$ and $g(\boldsymbol{y})$ is always nonnegative, i.e., $\mathsf{E}\{f(\boldsymbol{y})g(\boldsymbol{y})\} \ge \mathsf{E}\{f(\boldsymbol{y})\}\mathsf{E}\{g(\boldsymbol{y})\}$.

Proof: We prove this result by induction on the number of common *determining variables* of $f(\boldsymbol{y})$ and $g(\boldsymbol{y})$. When $f(\boldsymbol{y})$ and $g(\boldsymbol{y})$ share no common *determining variable*, then $f(\boldsymbol{y})$ and $g(\boldsymbol{y})$ are independent and $\mathsf{E}\{f(\boldsymbol{y})g(\boldsymbol{y})\} = \mathsf{E}\{f(\boldsymbol{y})\}\mathsf{E}\{g(\boldsymbol{y})\}$.

Suppose the same inequality holds for $f(\boldsymbol{y})$ and $g(\boldsymbol{y})$ sharing k determining variables. Consider the case in which $f(\boldsymbol{y})$ and $g(\boldsymbol{y})$ share $k+1$ determining variables. Without loss of generality, let y_1 be one of the common determining variables. Given $y_1 = 0$ (or $y_1 = 1$), the conditional $f(\boldsymbol{y})$ and $g(\boldsymbol{y})$ share only k common determining variables. We then have

$$\begin{aligned} &\mathsf{E}\{f(\boldsymbol{y})g(\boldsymbol{y})\} \\ &\quad = \mathsf{P}(y_1 = 1)\mathsf{E}\{f(\boldsymbol{y})g(\boldsymbol{y})|y_1 = 1\} \\ &\qquad + \mathsf{P}(y_1 = 0)\mathsf{E}\{f(\boldsymbol{y})g(\boldsymbol{y})|y_1 = 0\} \\ &\quad \ge \mathsf{P}(y_1 = 1)\mathsf{E}\{f(\boldsymbol{y})|y_1 = 1\}\mathsf{E}\{g(\boldsymbol{y})|y_1 = 1\} \\ &\qquad + \mathsf{P}(y_1 = 0)\mathsf{E}\{f(\boldsymbol{y})|y_1 = 0\}\mathsf{E}\{g(\boldsymbol{y})|y_1 = 0\} \end{aligned} \quad (3)$$

in which the last inequality follows from induction. On the other hand

$$\begin{aligned} &\mathsf{E}\{f(\boldsymbol{y})\}\mathsf{E}\{g(\boldsymbol{y})\} \\ &= (\mathsf{P}(y_1=1)\mathsf{E}\{f(\boldsymbol{y})|y_1=1\} + \mathsf{P}(y_1=0)\mathsf{E}\{f(\boldsymbol{y})|y_1=0\}) \\ &\quad \cdot (\mathsf{P}(y_1=1)\mathsf{E}\{g(\boldsymbol{y})|y_1=1\} + \mathsf{P}(y_1=0)\mathsf{E}\{g(\boldsymbol{y})|y_1=0\}). \end{aligned} \quad (4)$$

[5]The number of product terms is generally $\mathcal{O}(((d_v-1)(d_c-1))^l)$, where d_v and d_c are the maximal variable and check node degrees.

Subtracting (4) from (3), we have

$$\begin{aligned}(3)-(4) \geq & \mathsf{P}(y_1=1)\mathsf{P}(y_1=0) \\ & \cdot(\mathsf{E}\{f(\boldsymbol{y})|y_1=1\}-\mathsf{E}\{f(\boldsymbol{y})|y_1=0\}) \\ & \cdot(\mathsf{E}\{g(\boldsymbol{y})|y_1=1\}-\mathsf{E}\{g(\boldsymbol{y})|y_1=0\}).\end{aligned}$$

By *Proposition 2*, the right-hand side of the above inequality is always nonnegative. It is thus proved that $\mathsf{E}\{f(\boldsymbol{y})g(\boldsymbol{y})\} \geq \mathsf{E}\{f(\boldsymbol{y})\}\mathsf{E}\{g(\boldsymbol{y})\}$ for $f(\boldsymbol{y})$ and $g(\boldsymbol{y})$ sharing $k+1$ common determining variables. By induction, *Proposition 3* is proved. □

C. A Simple Upper Bound Based on the Decoding Tree

Let $g(\boldsymbol{y})$ and $h(\boldsymbol{y})$ be two iterative decoding functions. Consider the simple variable node message map $f_v = g \cdot h$ and the simple check node message map $f_c = g \vee h$. The three simplest cases are discussed below and three rules are introduced to cope with these scenarios.

Rule 0–Density-Evolution-Like Evaluation:

If $g(\boldsymbol{y})$ and $h(\boldsymbol{y})$ share no determining variables, then $g(\boldsymbol{y})$ and $h(\boldsymbol{y})$ are independent and

$$\begin{aligned}\mathsf{E}\{f_v(\boldsymbol{y})\} &= \mathsf{E}\{g(\boldsymbol{y})h(\boldsymbol{y})\} = \mathsf{E}\{g(\boldsymbol{y})\}\mathsf{E}\{h(\boldsymbol{y})\} \\ \mathsf{E}\{f_c(\boldsymbol{y})\} &= \mathsf{E}\{g(\boldsymbol{y}) \vee h(\boldsymbol{y})\} \\ &= \mathsf{E}\{g(\boldsymbol{y})\}+\mathsf{E}\{h(\boldsymbol{y})\}-\mathsf{E}\{g(\boldsymbol{y})\}\mathsf{E}\{h(\boldsymbol{y})\}.\end{aligned}$$

Namely, when the inputs are independent, the BER of interest can be iteratively computed by the above formulas. Nonetheless, a more interesting question is for the cases in which we have dependent $g(\boldsymbol{y})$ and $h(\boldsymbol{y})$. Can we still use this density-evolution-like (DE-like) evaluation as an upper bound? This question is answered in Rules 1 and 2.

Rule 1–A Simple Relaxation:

Suppose $g(\boldsymbol{y})$ and $h(\boldsymbol{y})$ share at least one determining variable, i.e., there is at least one repeated node in the input arguments of $g(\boldsymbol{y})$ and $h(\boldsymbol{y})$. By the inclusion–exclusion principle and by *Proposition 3*, we have

$$\begin{aligned}\mathsf{E}\{f_c(\boldsymbol{y})\} &= \mathsf{E}\{g(\boldsymbol{y}) \vee h(\boldsymbol{y})\} \\ &= \mathsf{E}\{g(\boldsymbol{y})\}+\mathsf{E}\{h(\boldsymbol{y})\}-\mathsf{E}\{g(\boldsymbol{y})h(\boldsymbol{y})\} \\ &\leq \mathsf{E}\{g(\boldsymbol{y})\}+\mathsf{E}\{h(\boldsymbol{y})\}-\mathsf{E}\{g(\boldsymbol{y})\}\mathsf{E}\{h(\boldsymbol{y})\}. \quad (5)\end{aligned}$$

The above rule suggests that when the incoming messages $g(\boldsymbol{y})$ and $h(\boldsymbol{y})$ of a check node are dependent, the error probability of the outgoing message can be upper-bounded by *blindly* assuming the incoming messages are independent and by invoking Rule 0, the DE-like evaluation. Furthermore, the upper bound computed by Rule 1 has the same asymptotic order as the target error probability, as can be easily seen in (5). The multiplicity of the upper bound may be different from that of the error probability. Due to the random-like interconnection within the code graph, for most cases, $g(y)$ and $h(y)$ are "nearly independent" and the multiplicity loss is not significant. The realization of Rule 1 is illustrated in Fig. 3, in which we assume that y_1 is the only common determining variable.

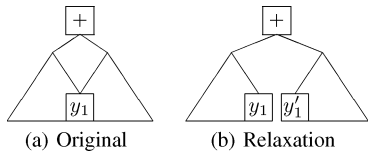


Fig. 3. Rule 1: A simple relaxation for check nodes.

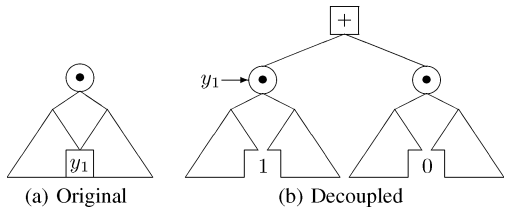


Fig. 4. Rule 2: A pivoting rule for variable nodes.

Rule 2–The Pivoting Rule:

Consider the simplest case in which $g(\boldsymbol{y})$ and $h(\boldsymbol{y})$ share one and only one common determining variable y_1. By simple Boolean algebra, we can upper-bound $f_v(\boldsymbol{y}) = g(\boldsymbol{y}) \cdot h(\boldsymbol{y})$ as follows:

$$g(\boldsymbol{y}) \cdot h(\boldsymbol{y}) \leq (g(0\boldsymbol{y}_2^n) \cdot h(0\boldsymbol{y}_2^n)) \vee (y_1 g(1\boldsymbol{y}_2^n) \cdot h(1\boldsymbol{y}_2^n)).$$

It can be proved by *Proposition 2* that the above inequality is actually an equality and we have

$$f_v(\boldsymbol{y}) = (g(0\boldsymbol{y}_2^n)h(0\boldsymbol{y}_2^n)) \vee (y_1 g(1\boldsymbol{y}_2^n)h(1\boldsymbol{y}_2^n)). \quad (6)$$

The realization of the above equation is demonstrated in Fig. 4. Once the tree in Fig. 4(a) is transformed to Fig. 4(b), messages entering the two variable nodes in Fig. 4(b) become independent since with the value of y_1 fixed, the conditional functions $g(y_1\boldsymbol{y}_2^n)$ and $h(y_1\boldsymbol{y}_2^n)$ share no common determining variables and are independent of each other. The expectation of the variable node AND operation can be *exactly* evaluated by Rule 0. Although the two messages entering the top check node in Fig. 4(b) are still dependent, the expectation at the check node OR operation can be *upper-bounded* by reapplying Rule 1. The final upper bound thus becomes

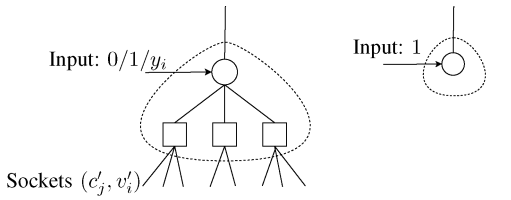
$$\begin{aligned}\mathsf{E}\{f_v(\boldsymbol{y})\} \leq & \mathsf{E}\{f_v(0\boldsymbol{y}_2^n)\}+\mathsf{E}\{y_1\}\mathsf{E}\{f_v(1\boldsymbol{y}_2^n)\} \\ & -\mathsf{E}\{f_v(0\boldsymbol{y}_2^n)\}\mathsf{E}\{y_1\}\mathsf{E}\{f_v(1\boldsymbol{y}_2^n)\} \quad (7)\end{aligned}$$

where for any deterministic $y_1 \in \{0,1\}$

$$\mathsf{E}\{f_v(y_1\boldsymbol{y}_2^n)\} = \mathsf{E}\{g(y_1\boldsymbol{y}_2^n)\}\mathsf{E}\{h(y_1\boldsymbol{y}_2^n)\}.$$

In summary, after performing the pivoting rule as described by (6) and in Fig. 4, the expectation $\mathsf{E}\{f_v(\boldsymbol{y})\}$ can also be upper-bounded by the DE-like evaluation.

Remark 1: The pivoting rule (6) gives an equality and thus does not incur any performance loss. Any potential looseness of this upper bound (7) results from the application of Rule 1.



(a) A normal (c_j, v_i) leaf unit. (b) A dummy leaf unit.

Fig. 5. The (c_j, v_i) and the dummy leaf units.

Rule 2 thus preserves the asymptotic order of $\mathsf{E}\{f_v(\boldsymbol{y})\}$ as does Rule 1. Only the multiplicity term might be loose.

Remark 2: Although Rules 0 to 2 hold for iterative decoding functions in sum–product form as well, it is more efficient to represent an iterative decoding function in its original tree form since the latter admits efficient computation of the DE-like evaluation. For general iterative decoding functions, the DE-like evaluation can only be used as "an approximation" of the finite code performance without concrete justification. Rules 1 and 2 provide a mechanism for converting the original tree to a suitable format such that the direct DE-like evaluation gives us a rigorous upper bound on the finite code length BER.

D. The Algorithm

As a simple (easy to evaluate) but asymptotically tight upper bound, Rules 0 to 2 are designed to upper-bound the expectation of single operations, either an AND or an OR operation, with the incoming messages $g(\boldsymbol{y})$ and $h(\boldsymbol{y})$ sharing at most one common determining variable. Thus, Rules 0 to 2 are not directly applicable to real decoding problems for parity-check codes, which involve multiple concatenated operations and multiple common determining variables. Nonetheless, once carefully concatenated, Rules 0 to 2 can construct a BER upper bound UB_i for an infinite-sized decoding tree with many repeated nodes while still preserving most of the nested structure.

Before the detailed description of the concatenation of these three rules, we define a (c_j, v_i) leaf unit as in Fig. 5(a). To be more explicit, a (c_j, v_i) leaf unit contains one variable node and $\deg(v_i) - 1$ check nodes. Within the (c_j, v_i) leaf unit, there are $\deg(v_i) - 1$ edges connecting the variable node and the check nodes. There is one upstream edge entering the variable node and $\sum_{c'_j \in \partial v_i \backslash c_j} (\deg(c'_j) - 1)$ downstream edges leaving the check nodes, where ∂v_i denotes the set of neighbors of v_i. Each downstream edge is assigned an open socket labeled (c'_j, v'_i) such that $v'_i \in \partial c'_j \backslash v_i$. Additional leaf units can be concatenated through those sockets during tree construction. Once there is one leaf unit connected to a parent leaf unit through a particular edge, the corresponding socket is then considered closed. There is one input entering the variable node, which can be 0 or 1 or an active observation y_i. It is very important to view a leaf unit as a purely graphical structure and not associating any labels to the variable/check nodes. Only the sockets and an active observation are labeled by (c'_j, v'_i) and by y_i in a leaf unit.

We also define a $(\emptyset, v_i)$ leaf unit that is very similar to a (c_j, v_i) leaf unit except that there is no upstream edge entering the variable node and there are $\deg(v_i)$ interior check nodes and $\sum_{c'_j \in \partial v_i} (\deg(c'_j) - 1)$ downstream edges. The $(\emptyset, v_i)$ leaf unit will be used exclusively for the root of a decoding tree, for which there is no upstream edge. A dummy leaf unit contains only one variable node and one input that is hardwired to 1 as illustrated in Fig. 5(b).

The detailed concatenation of Rules 0 to 2 is provided in Algorithm 2.

Algorithm 2 A tree-based method for upper bounding p_i, the BER of bit x_i.

1: **Initialization:** Let $\mathcal{T}$ be a tree containing an $(\emptyset, v_i)$ leaf unit, of which all children sockets are open and the variable node has an active input y_i.

2: **repeat**

3: Locate an open socket, say a (c_j, v_k)-socket. Append to the socket a (c_j, v_k) leaf unit with an active input y_k. Use v to denote the variable node in the newly added leaf unit.

4: **if** there exists at least another variable node v' in $\mathcal{T}$ that also has active input observation y_k **then**

5: Let V' denote the collection of all such v' in $\mathcal{T}$ that have active input observation y_k.

6: Let $\mathsf{yca}(v, V')$ denote the youngest common ancestor[6] between v and all $v' \in V'$. Use v^* to denote the node in V' such that the youngest common ancestor $\mathsf{yca}(v, v^*)$ of v and v^* is $\mathsf{yca}(v, V')$.

7: **if** $\mathsf{yca}(v, V')$ is a variable node **then**

8: As suggested by Rule 2, a pivoting construction involving tree duplication is initiated. Consider the iterative decoding function $f(\boldsymbol{y})$ for the subtree rooted at $\mathsf{yca}(v, V')$. The two incoming messages along the branches corresponding to v and v^*, respectively, are denoted as $g(\boldsymbol{y})$ and $h(\boldsymbol{y})$. The shared determining variable is y_k. The pivoting rule with respect to $f(\boldsymbol{y})$, $g(\boldsymbol{y})$, $h(\boldsymbol{y})$, and y_k is illustrated in Fig. 6(a),(b).

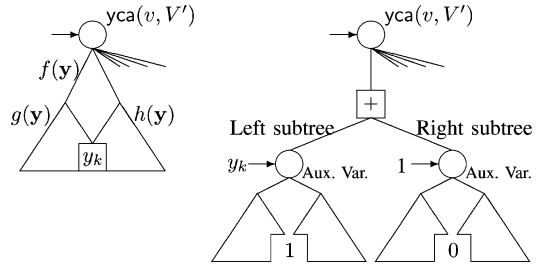
9: As illustrated, the subtrees correspond to $g(\boldsymbol{y})$ and $h(\boldsymbol{y})$ are duplicated and two new variable nodes and one new check node are added between $\mathsf{yca}(v, V')$ and the duplicated subtrees.

10: Replace all active y_k observations in the duplicated left and right subtrees either by 1 or by 0 depending on whether the active observation is in the left subtree or in the right subtree.

11: The root of the left subtree, which is an auxiliary variable node, is assigned an active input y_k while the root of the right subtree has its input hardwired to 1. (See Fig. 6(b).)

---

[6]For any two nodes v_1 and v_2 in a tree, their youngest common ancestor $\mathsf{yca}(v_1, v_2)$ can be uniquely defined by finding all the common ancestors of this pair of nodes and then selecting the youngest of them. For any existing variable node v' with active observation y_k, we can uniquely determine the $\mathsf{yca}(v, v')$ between the variable node v in the newly added leaf unit and the existing node v'. In Line 6 of Algorithm 2, $\mathsf{yca}(v, V')$ refers to the youngest node among $\{\mathsf{yca}(v, v') : v' \in V'\}$. Note that $\mathsf{yca}(v, V')$ is unique but v^* may not be. Algorithm 2 works for any choice of v^*.



(a) Original (b) After pivoting

Fig. 6. The pivoting rule, Rule 2, in a real decoding tree. For the most general case, yca(v, V') can have ≥ 2 children and the children not involved during pivoting are represented by lines entering yca(v, V') with different slopes. The function $f(\boldsymbol{y})$ focuses mainly on the AND operation on the two incoming messages $g(\boldsymbol{y})$ and $h(\boldsymbol{y})$ with respect to the branches corresponding to v and v^*.

12: **end if**

13: **end if**

14: **until** the size of $\mathcal{T}$ exceeds the preset limit.

15: Append all remaining open sockets by dummy leaf units (Fig. 5(b)).

16: UB_i can be computed by blindly assuming all incoming messages in $\mathcal{T}$ are independent and applying Rule 0 iteratively.

The evaluation of UB_i in Algorithm 2 is taken iteratively based on Rule 0. For the case in which an observation y is hardwired to zero/one, we substitute $\mathsf{E}\{y\} = 0$ or $\mathsf{E}\{y\} = 1$ into the formula of the variable node function $\mathsf{E}\{f_v(\boldsymbol{y})\}$. Note: the formulas in Rule 0 are written in terms of expectations. When Rule 0 is used for computing the upper bounds, one should follow the corresponding arithmetic operations but should not associate the computed quantities as expectations any more. A more accurate description for the iterative upper bound computation in Rule 0 should be

$$\begin{aligned}\text{UB}_{\mathsf{E}\{f_v(\boldsymbol{y})\}} &= \text{UB}_{\mathsf{E}\{g(\boldsymbol{y})\}}\text{UB}_{\mathsf{E}\{h(\boldsymbol{y})\}} \\ \text{UB}_{\mathsf{E}\{f_c(\boldsymbol{y})\}} &= \text{UB}_{\mathsf{E}\{g(\boldsymbol{y})\}} + \text{UB}_{\mathsf{E}\{h(\boldsymbol{y})\}} \\ &\quad - \text{UB}_{\mathsf{E}\{g(\boldsymbol{y})\}}\text{UB}_{\mathsf{E}\{h(\boldsymbol{y})\}}\end{aligned}$$

instead of using the expectation operator. We then have the following theorem:

Theorem 2: Algorithm 2 generates a BER upper bound $\text{UB}_i \geq p_i \triangleq \mathsf{E}\{f_i(\boldsymbol{y})\}$.

The proof of *Theorem 2* involves the graph-theoretic properties of yca(v, V') defined in Line 6 of Algorithm 2 and is deferred to Appendix I in the interest of streamlining the discussion.

Let us denote by $f_{\mathcal{T},t}(\boldsymbol{y})$ the corresponding iterative decoding function based on the tree $\mathcal{T}$ assuming t iterations of the REPEAT-UNTIL loop in Line 2 are performed. Then, the detailed behavior of Algorithm 2 can be further described by the following theorem.

Theorem 3 (A Narrowing Search): For all $t \in \mathbb{N}$ and for all $\boldsymbol{y} \in \{0,1\}^n$, we have

$$f_i(\boldsymbol{y}) \leq f_{\mathcal{T},t+1}(\boldsymbol{y}) \leq f_{\mathcal{T},t}(\boldsymbol{y}).$$

In other words, for any set $\boldsymbol{v} \subseteq \{v_1, \ldots, v_n\}$, let $\boldsymbol{y}_{\boldsymbol{v}} = (y_1, \ldots, y_n) \in \{0,1\}^n$ be the corresponding binary vector such that $\forall i \in \{1, \ldots, n\}$, $y_i = 1$ if $v_i \in \boldsymbol{v}$ and $y_i = 0$, otherwise. Then for any stopping set $\boldsymbol{v}_s$, $f_{\mathcal{T},t}(\boldsymbol{y}_{\boldsymbol{v}_s}) = 1$. There may be a $\boldsymbol{v}$ that is not a stopping set but that still satisfies $f_{\mathcal{T},t}(\boldsymbol{y}_{\boldsymbol{v}}) = 1$. Theorem 3 says that there may be some $\boldsymbol{v}$ such that $f_i(\boldsymbol{y}_{\boldsymbol{v}})$ is strictly less than $f_{\mathcal{T},t}(\boldsymbol{y}_{\boldsymbol{v}})$. But as iteration proceeds, the number of such $\boldsymbol{v}$ becomes smaller and smaller. Thus, the collection of "possibly bad $\boldsymbol{v}$" characterized by $\mathcal{T}$ becomes smaller and smaller.

The proof of *Theorem 3* is based on an incremental tree-revealing argument presented in Appendix II.

We conclude this section by listing some other properties of Algorithm 2 as follows.

- The only computationally expensive step is when Rule 2 is invoked, which, in the worst case, may double the tree size and thus reduce the efficiency of Algorithm 2. Some tree-pruning rules will be introduced later in Section V-A, which alleviate significantly the computational cost associated to Rule 2.
- Once the tree construction is completed, evaluating UB_i for any $\epsilon \in [0,1]$ can be achieved efficiently with complexity $\mathcal{O}(|\mathcal{T}|\log(|\mathcal{T}|))$, where $|\mathcal{T}|$ is the size of $\mathcal{T}$.
- The preset size limit of $\mathcal{T}$ provides a tradeoff between computational resources and the tightness of the resulting UB_i. Since Algorithm 2 generates a rigorous upper bound, one can terminate the program early even before the tightest results are obtained, as long as the intermediate results have met the evaluation/design requirements.

V. Performance/Efficiency Related Issues

In this section, different schemes to further improve the efficiency/performance of UB_i or to confirm the tightness of UB_i will be discussed.

A. Pruning the Tree

To control the growth rate of the size of $\mathcal{T}$, we introduce the following two lossless pruning rules, which also play an important role when proving the optimality of Algorithm 2.

Rule 3–The Equation $0 \cdot f(\boldsymbol{y}) = 0$:

We first note that $0 \cdot f(\boldsymbol{y}) = 0$ for any $f(\boldsymbol{y})$. Therefore, if a variable node v has input 0, then we can completely discard all other messages entering v. Or equivalently, we can prune all subtrees rooted at v and the size of $\mathcal{T}$ will be reduced. Rule 3 can be applied to the right subtree in Fig. 6(b) every time pivoting is performed, since Line 10 in Algorithm 2 replaces many active inputs by fixed zeros.

Rule 4–Degenerate Pivoting:

We first note that the following equation holds for all iterative decoding functions $g(\boldsymbol{y})$:

$$y_k \cdot g(\boldsymbol{y}_1^{k-1} y_k \boldsymbol{y}_{k+1}^n) = y_k \cdot g(\boldsymbol{y}_1^{k-1} 1 \boldsymbol{y}_{k+1}^n) \tag{8}$$

for all $\boldsymbol{y} \in \{0,1\}^n$.

For the variable node v in the newly added leaf unit with active input y_k, define yca(v, V') and v^* as in Algorithm 2. Motivated by (8), if yca$(v, V') = v^*$, then no pivoting is necessary and we can directly replace all active y_k inputs in the subtree

rooted at v^* by 1's, as suggested by the right-hand side of (8). Rule 4 can be viewed as a degenerate case of Rule 2 in which $h(\boldsymbol{y}) = y_k$.

Whether $\mathsf{yca}(v, V') = v^*$ should be checked every time pivoting is about to be performed. If $\mathsf{yca}(v, V') = v^*$, then we can use Rule 4 instead of pivoting, and the computational cost is reduced.

B. The Socket-Finding Module

The tightness of UB_i in Algorithm 2 depends heavily on the socket-finding (SF) module invoked in Line 3. A properly designed SF module generates an upper bound UB_i of which the asymptotic order is $+1$ to $+3$ better than that of a random SF module. The intuition behind a good SF module is as follows. Theorem 3 ensures that the support tree $\mathcal{T}$ captures all stopping sets plus some non-stopping sets. Adding new leaf-units enlarges the support tree $\mathcal{T}$, which consequently places more constraints on $f_{\mathcal{T},t}(\boldsymbol{y})$ and potentially may eliminate some non-stopping sets. Nonetheless, the newly added constraint may be redundant to the existing constraints captured by $\mathcal{T}$, namely, it is possible that $f_{\mathcal{T},t+1}(\boldsymbol{y}) = f_{\mathcal{T},t}(\boldsymbol{y}), \forall \boldsymbol{y}$ and no non-stopping set is eliminated. The goal of a good SF module is thus to ensure that the constraint resulted by a new leaf unit *eliminates* at least one non-stopping set such that $\exists \boldsymbol{y}$ such that $f_{\mathcal{T},t+1}(\boldsymbol{y}) < f_{\mathcal{T},t}(\boldsymbol{y})$.

The ultimate benefit of an optimal SF module is stated in the following theorem.

Theorem 4 (The Optimal SF Module): With an optimal SF module, the simple pruning rules in Section V-A, and a sufficiently large amount of memory, there exists $t_0 < \infty$ such that

$$f_i(\boldsymbol{y}) = f_{\mathcal{T},t_0}(\boldsymbol{y}),\ \forall \boldsymbol{y} \in \{0,1\}^n.$$

In words, the output tree $\mathcal{T}$ is able to implicitly characterize *all* stopping sets containing x_i within a finite number of steps. We then have the following corollary.

Corollary 1 (Order Tightness of Algorithm 2): With an optimal SF module and the pruning rules, the UB_i computed by Algorithm 2 is tight in terms of the asymptotic order if sufficient computational resources are provided. Namely, there exists $C < \infty$ such that $\frac{\mathrm{UB}_i(\epsilon)}{p_i(\epsilon)} < C$ for all $\epsilon \in (0, 1]$.

Proof: This is a direct result of *Theorem 4* and the order preserving property of Rule 1, the only relaxation rule. In particular, the proof follows from *Proposition 5* in Appendix I. □

It is worth emphasizing that if a suboptimal SF module is used or the pruning rules are not employed, $f_{\mathcal{T},t}(\boldsymbol{y})$ may be strictly bounded away from $f_i(\boldsymbol{y})$ for some $\boldsymbol{y}$ even when $t \to \infty$. The proof of *Theorem 4* is provided in Appendix III.

For all our numerical experiments, we use an efficient approximation of the optimal SF module motivated by the proof of *Theorem 4*.

C. Confirming the Tightness of UB_i and Constructing an Exhaustive List of Minimum Stopping Sets

Two important features of the proposed algorithm are the easy confirmation of the tightness of UB_i and a byproduct tight lower bound LB_i, which together with the tight UB_i brackets the error-floor performance of a given parity-check code. In this subsection, we will discuss how to construct an exhaustive list of minimum stopping sets containing x_i from Algorithm 2.

To this end, after t iterations of Algorithm 2, we first enumerate exhaustively all the potentially bad sets $\boldsymbol{v} \subseteq \{v_1, \ldots, v_n\}$ of minimal size such that $f_{\mathcal{T},t}(\boldsymbol{y}_{\boldsymbol{v}}) = 1$. Denote the minimal size as $w_{\mathcal{T}}$ and denote the exhaustive collection of those $\boldsymbol{v}$ by $\boldsymbol{V}_{\min}$. We then have the following corollaries.

Corollary 2 (Tightness Confirmation): If there exists $\boldsymbol{v} \in \boldsymbol{V}_{\min}$ that is a stopping set, then UB_i is tight in terms of the asymptotic order. Otherwise, UB_i is loose.

Corollary 3 (Stopping Set Exhaustive Enuemration): Let $\boldsymbol{V}_{\min,\mathrm{SS}} \subseteq \boldsymbol{V}_{\min}$ denote the collection of all $\boldsymbol{v} \in \boldsymbol{V}_{\min}$ that are also stopping sets. If $\boldsymbol{V}_{\min,\mathrm{SS}}$ is not empty, then $\boldsymbol{V}_{\min,\mathrm{SS}}$ is an exhaustive list of all minimum stopping sets containing bit x_i. If $\boldsymbol{V}_{\min,\mathrm{SS}} = \emptyset$, then there is no stopping set of size $\leq w_{\mathcal{T}}$.

Corollary 4 (The Tight Lower Bound): A nonempty $\boldsymbol{V}_{\min,\mathrm{SS}}$ can be used to derive a lower bound $\mathsf{E}\{f_{\mathrm{LB},i}(\boldsymbol{y})\}$ that is guaranteed to be tight in both the asymptotic order and the multiplicity as described in Section IV-A.

Proofs of Corollaries 2 to 4: We first notice that $w_{\mathcal{T}}$ is the asymptotic order of both $\mathsf{E}\{f_{\mathcal{T},t}(\boldsymbol{y})\}$ and the upper bound UB_i due to the order-preserving property of Rule 1. Suppose there exists $\boldsymbol{v} \in \boldsymbol{V}_{\min}$ that is a stopping set. By *Theorem 3*, no stopping set containing x_i has weight less than $w_{\mathcal{T}}$ and the asymptotic order of $p_i = \mathsf{E}\{f_i(\boldsymbol{y})\}$ is thus $w_{\mathcal{T}}$. UB_i therefore has the same asymptotic order as that of $\mathsf{E}\{f_i(\boldsymbol{y})\}$. For the other case in which no element of $\boldsymbol{V}_{\min}$ is a stopping set, by *Theorem 3*, any stopping set containing x_i must have weight strictly larger than $w_{\mathcal{T}}$ and therefore the asymptotic order of $\mathsf{E}\{f_i(\boldsymbol{y})\}$ must be strictly larger than $w_{\mathcal{T}}$. UB_i is loose and Corollary 2 is proved.

Theorem 3 also guarantees that the minimal stopping distance d_{S} is $\geq w_{\mathcal{T}}$. If there exists $\boldsymbol{v} \in \boldsymbol{V}_{\min}$ that is a stopping set, then $d_{\mathrm{S}} = w_{\mathcal{T}}$. Theorem 3 then implies that all minimum stopping sets must be in $\boldsymbol{V}_{\min}$. Therefore, $\boldsymbol{V}_{\min,\mathrm{SS}}$ is an exhaustive list of minimum stopping sets. Corollary 3 is proved.

Corollary 4 is a straightforward result of Corollary 3 . □

The exhaustive list $\boldsymbol{V}_{\min,\mathrm{SS}}$ gives us a richer understanding of the code behavior than the Yes/No answer to the decision problem $\mathrm{SD}(\boldsymbol{H}, t)$ in Section III-B.

VI. Further Generalizations

In the previous sections, a tree-based algorithm for upper-bounding BER performance and exhaustively enumerating minimum stopping sets is provided. To further demonstrate the versatility of the proposed algorithm, we discuss the following generalizations, including the minimum k-out trapping set exhaustive search algorithms.

A. Applications on Shortened and Punctured Codes

Algorithm 2 can be easily generalized for codes with punctured or shortened bits as follows.

If we take a closer look at the puncture operation on a particular bit x_k, it is equivalent to deterministically assigning "erasure" to y_k. That is, it becomes an erasure channel with $y_k = 1$ with probability one. Similarly, shortening a bit x_k is equivalent to deterministically assigning value 0 to the observation y_k. Based on this observation, Algorithm 2 can be applied to codes with punctured and shortened bits by replacing all active y_k inputs with fixed 0/1 values depending on whether bit x_k is shortened or punctured. The ability to analyze shortened/punctured codes makes the algorithm directly applicable to irregular/regular repeat–accumulate (RA) codes without modifications.

Not only can the algorithm handle the punctured and the shortened bits, the upper bound UB_i can be computed for the setting of parallel independent channels, in which different bits x_k may experience different erasure probabilities $\epsilon_k = \mathsf{E}\{y_k\}$. No modification of Algorithm 2 is necessary since Algorithm 2 does not assume a common erasure probability for different bits x_k. The unequal erasure probabilities affect the output UB_i only during the evaluation stage (Line 16 of Algorithm 2).

B. The Branch-and-Bound Approach

The upper bound UB_i computed by Algorithm 2 also suits naturally the classic branch-and-bound approach widely used in the computer science community for solving NP-hard problems with practical input parameters. A detailed description of exploiting Algorithm 2 as part of a branch-and-bound method is as follows.

We first notice that the expectation $\mathsf{E}\{f_i(\boldsymbol{y})\}$ can be further decomposed as

$$\mathsf{E}\{f_i(\boldsymbol{y})\} = \sum_{j=1}^{M} \mathsf{P}(\mathcal{A}_k)\mathsf{E}\{f_i(\boldsymbol{y})|\boldsymbol{y} \in \mathcal{A}_k\}$$

where the $\mathcal{A}_k$'s are M events partitioning the sample space $\{0,1\}^n$. For example, we can define a collection of nonuniform $\mathcal{A}_k$'s by

$$\begin{aligned} \mathcal{A}_1 &= \{\boldsymbol{y} \in \{0,1\}^n : y_{10} = 0\} \\ \mathcal{A}_2 &= \{\boldsymbol{y} \in \{0,1\}^n : y_{10} = 1, y_{47} = 0\} \\ \text{and} \quad \mathcal{A}_3 &= \{\boldsymbol{y} \in \{0,1\}^n : y_{10} = 1, y_{47} = 1\}. \end{aligned}$$

Since for any k, $f_i(\boldsymbol{y})|_{\boldsymbol{y}\in\mathcal{A}_k}$ is simply another finite code with many punctured and shortened bits, Algorithm 2 can be applied to each $f_i(\boldsymbol{y})|_{\boldsymbol{y}\in A_k}$, respectively, and different bounds $\mathrm{UB}_{i,k} \geq \mathsf{E}\{f_i(\boldsymbol{y})|\boldsymbol{y} \in A_k\}$ will be obtained. An overall, composite upper bound can be obtained by combining the upper bounds for different partitioning events

$$\text{C- UB}_i = \sum_{k=1}^{M} \mathsf{P}(\mathcal{A}_k)\mathrm{UB}_{i,k} \geq \mathsf{E}\{f_i(\boldsymbol{y})\} = p_i.$$

One might also view the above method as a divide-and-conquer approach.

In addition to upper-bounding p_i, this branch-and-bound approach can also be used to search exhaustively for minimum stopping sets after the following slight modification. Consider the event $\mathcal{A}_2$ as an example. Applying Algorithm 2 to $f_i(\boldsymbol{y})|_{\boldsymbol{y}\in\mathcal{A}_2}$ returns an exhaustive list of the minimum stopping sets of the corresponding punctured/shortened code in which x_{10} is punctured while x_{47} is shortened. Any minimum stopping set $\boldsymbol{x}$, however, will not include x_{10} since $y_{10} = 1$ is considered to be a fixed, punctured code structure instead of a channel observation. By taking the union, $\{x_{10}\} \cup \boldsymbol{x}$ becomes a minimum stopping set of the original code $f_i(\boldsymbol{y})$ with $\boldsymbol{y}$ being confined in $\mathcal{A}_2$

$$f_i(\boldsymbol{y}_{\{x_{10}\}\cup\boldsymbol{x}}) = 1 \quad \text{and} \quad \boldsymbol{y}_{\{x_{10}\}\cup\boldsymbol{x}} \in \mathcal{A}_2.$$

In summary, with the above slight modification, the branch-and-bound approach is able to search exhaustively through the partitioned space $\bigcup_{k=1}^{M} \mathcal{A}_k$ one event at a time. The reduced sample space in each event $\mathcal{A}_k$ makes the branch-and-bound approach much more efficient than the original algorithm alone.

With a properly chosen nonuniform partition $\{\mathcal{A}_k\}$ of 50–20,000 events, the branch-and-bound approach generally results in upper bounds of the asymptotic order $+1$ to $+3$ higher than the original UB_i, and is capable of exhaustively enumerating small stopping sets of sizes $+1$ to $+3$ larger than the capability of the original algorithm. With the improved efficiency, we are able to determine the minimal stopping distance and enumerate exhaustively all minimum stopping sets for many rate-$1/2$ LDPC codes of practical length $n \approx 500$.

C. BER Versus FER

Thus far, all our discussions have been based on the perspective of each individual bit. We either construct an upper bound on the BER p_i, or enumerate exhaustively all minimum stopping sets involving bit x_i. Upper bounds for the average BER can be easily obtained by taking averages over upper bounds for individual bits. An equally interesting problem is to upper bound the FER. In this subsection, we briefly discuss how to upper-bound the FER or to enumerate exhaustively the minimum stopping sets from the frame perspective.

We again rely on the Boolean expression framework. By noticing that the frame error detector is simply the binary OR of all individual bit detectors, we have $f_{\mathrm{FER}}(\boldsymbol{y}) = \bigvee_{i=1}^{n} f_i(\boldsymbol{y})$, which is yet another iterative decoding function. Since Algorithm 2 is applicable to all iterative decoding functions, the upper bound $\mathrm{UB}_{\mathrm{FER}}$ and the exhaustive list of minimum stopping sets can be obtained by directly applying Algorithm 2 on $f_{\mathrm{FER}}(\boldsymbol{y})$. A graphical interpretation of $f_{\mathrm{FER}}(\boldsymbol{y})$ can be obtained by introducing an auxiliary variable and check node pair (x_0, y_0) such that the new variable node x_0 is punctured and the new check node y_0 is connected to all $n+1$ variable nodes from x_0 to x_n. The FER of the original code now equals the BER p_0 of variable node x_0 and can be upper-bounded by Algorithm 2. An efficient and straightforward partition $\{\mathcal{A}_k\}_{k\in\{1,\ldots,n+1\}}$ for the branch-and-bound approach is

$$\begin{aligned} \forall k \in \{1,\ldots,n\},\ \mathcal{A}_k &:= \{\boldsymbol{y} : \boldsymbol{y}_1^{k-1} = \boldsymbol{0}, y_k = 1\}, \\ \text{and} \quad \mathcal{A}_{n+1} &:= \{\boldsymbol{y} : \boldsymbol{y} = \boldsymbol{0}\}. \end{aligned} \tag{9}$$

All our FER results are based on the above partition or on a finer partition derived from the above basic partition.

Remark: Since the FER depends only on the worst bit performance, it is generally easier to construct tight $\mathrm{UB}_{\mathrm{FER}}$ for the FER than for the BER. Although it is computationally more challenging to find the bit-wise upper bounds/exhaustive lists of

minimum stopping sets, the bit-wise information provides detailed performance prediction for each individual bit and is of great importance during code optimization and analysis [37], [18].

D. Exhaustively Enumerating Minimum k-Out Trapping Sets

The Algorithm:

We are now ready to generalize the bit-wise minimum stopping set exhaustive search algorithm for minimum k-out trapping sets in a frame-wise perspective. We start by converting our bit-wise stopping set exhaustive search algorithm to its frame-wise counterpart. Namely, using the procedures described in the previous sections, one is able to obtain an efficient algorithm taking inputs $\boldsymbol{H}$, t, and a list of punctured bits $\boldsymbol{x}_p = \{x_p\}$, and outputting $(\boldsymbol{X}_{\min}, w_{\min})$. If the minimal size of the stopping sets is within the searchable range t, $\boldsymbol{X}_{\min}$ is an exhaustive list of the frame-wise minimum stopping sets and $w_{\min}$ is the corresponding minimal size. If the minimal stopping distance is beyond the searchable range t, then $\boldsymbol{X}_{\min} = \emptyset$ and $w_{\min} = t$. We denote this algorithm as the minimum stopping set exhaustive search SSES$(\boldsymbol{H}, t, \boldsymbol{x}_p)$. The minimum k-out trapping set exhaustive search algorithm is then described in Algorithm 3.

Algorithm 3 A minimum k-out trapping set exhaustive search algorithm based on SSES$(\boldsymbol{H}, t, \boldsymbol{x}_p)$

1: **Input:** $\boldsymbol{H}$ and t.

2: **Initialization:** $\boldsymbol{X}_{\min} \leftarrow \emptyset$ and $w_{\min} \leftarrow t$.

3: **repeat**

4: Based on the Tanner graph, select k edges connecting to k distinct check nodes and denote them as $\{(x_{i_1}, c_{j_1}), \ldots, (x_{i_k}, c_{j_k})\}$.

5: **if** there is no edge between $x_{i_{k'}}$ and $c_{j_{k''}}$ for any $k' \neq k''$, **then**

6: Construct a new $\boldsymbol{H}'$ by removing all columns i in $\boldsymbol{H}$ that satisfy both 1) $i \neq i_a, \forall a = 1, \ldots, k$, and 2) $\exists a \in \{1, \ldots, k\}$ such that $H_{j_a i} = 1$.

7: Construct a new $\boldsymbol{H}''$ by removing rows $j_a, \forall a = 1, \ldots, k$ from $\boldsymbol{H}'$.

8: Let the punctured bits be $\boldsymbol{x}_p \leftarrow \{x_{i_a} : \forall a = 1, \ldots, k\}$

9: $(\boldsymbol{X}_{\text{temp}}, w_{\text{temp}}) \longleftarrow$ SSES$(\boldsymbol{H}'', t, \boldsymbol{x}_p)$.

10: **if** $w_{\text{temp}} + k < w_{\min}$ **then**

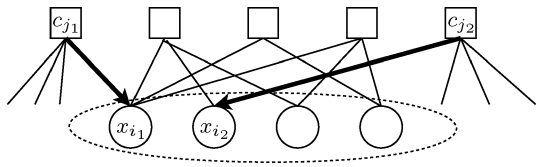
11: $\boldsymbol{X}_{\min} \leftarrow \boldsymbol{X}_{\text{temp}} \circ \boldsymbol{x}_p$† and $w_{\min} \leftarrow w_{\text{temp}} + k$.

12: **else if** $w_{\text{temp}} + k = w_{\min}$ **then**

13: $\boldsymbol{X}_{\min} \leftarrow \boldsymbol{X}_{\min} \cup (\boldsymbol{X}_{\text{temp}} \circ \boldsymbol{x}_p)$. †

14: **end if**

15: **end if**



Stopping Set for $\mathbf{H} \Longleftrightarrow$ k-out Trapping Set for $\mathbf{H}''$

Fig. 7. Illustration of Algorithm 3, in which $k = 2$. The two corresponding extrinsic edges (x_{i_1}, c_{j_1}) and (x_{i_2}, c_{j_2}) are illustrated.

16: **until** all possible selections of k distinct edges are exhausted.

17: **Output:** $\boldsymbol{X}_{\min}$ and $w_{\min}$.

†: The "$\circ$" operation is defined as follows. Any element in $\boldsymbol{X} \circ \boldsymbol{x}_p$ is of the form $\boldsymbol{x} \cup \boldsymbol{x}_p$ for some $\boldsymbol{x} \in \boldsymbol{X}$.

Theorem 5: The output of Algorithm 3 is an exhaustive list of minimum k-out trapping sets if the minimal k-out trapping distance is $\leq t$.

Proof: The **if** condition in Line 5 ensures that the k edges chosen in Line 4 are legitimate choices of the k "extrinsic" edges in any k-out trapping sets since each c_{j_a} is connected to one and only one x_{j_a}.

Line 6 ensures that all variable nodes connected to those c_{j_a} (excluding the selected x_{i_a}) are hardwired to 0. Therefore, we can simply remove those corresponding columns while we search for the minimum stopping sets of the remaining graph. Line 7 further strips away those extrinsic edges (x_{i_a}, c_{j_a}). Puncturing in Line 8 forces the SSES algorithm to focus exclusively on minimum stopping sets that contain $\boldsymbol{x}_p$. Combining the above modifications, any minimum stopping set in $\boldsymbol{H}''$ corresponds to a minimum k-out trapping set in $\boldsymbol{H}$. By exhaustively considering all ways of choosing the k extrinsic edges, the proof is complete. See Fig. 7 for illustration of this behavior. □

With a fixed search range t, the complexity of Algorithm 3, although being polynomial with respect to n, grows on the order of $\mathcal{O}\left(n^k\right)$, which makes the above algorithm less appealing for cases of large k. Using the efficient SSES$(\boldsymbol{H}, t, \boldsymbol{x}_p)$ proposed in this paper, the proposed algorithm is capable of identifying minimum k-out trapping sets for short practical codes $n \approx 500$ with $k = 1, 2$.

On the other hand, for commonly used LDPC codes with many low-degree variable nodes, the dominating error patterns are generally those k-out trapping sets with small $k \leq 4$, as noted in [6]. In the same paper, it has been shown that for a rate $51/64$, $n = 4096$ almost-regular code, more than 55% of the FER is contributed by the minimum k-out trapping sets with $k = 0, 1, 2$, equivalent to the $(10, 0), (9, 1), (8, 2)$ near-codewords respectively [6, Fig. 5]. Additional numerical examples will be reported in Section VII, including the exhaustive enumeration of all minimum 2-out trapping sets of size 8 for the $(155, 64, 20)$ Tanner code [29].

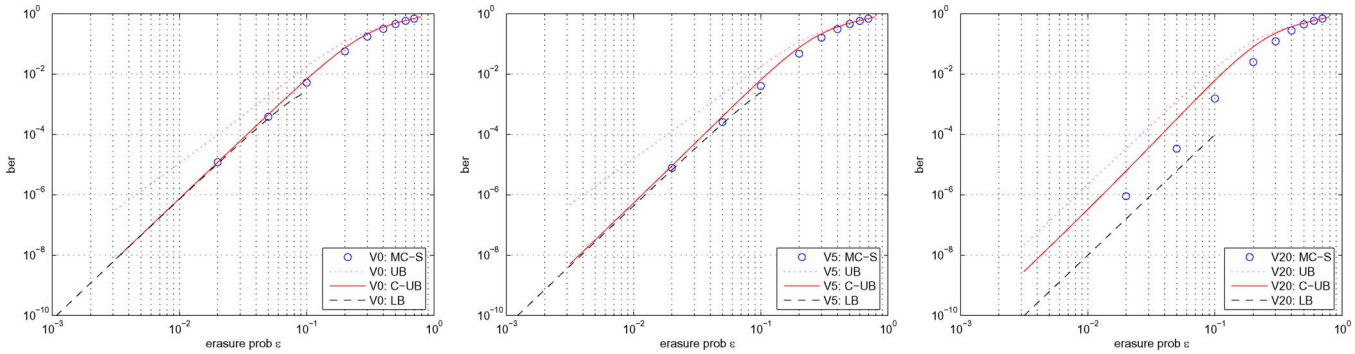


Fig. 8. Comparisons among the upper bound (UB), the composite upper bound (C-UB), the Monte Carlo simulation (MC-S), and the byproduct tight lower bound (LB) for bits 0, 5, and 20 of the $(23, 12)$ binary Golay code. The corresponding asymptotic (order, multiplicity) pairs obtained by UB, C-UB, and the actual BER is are V0: $\{(3,10),(4,75),(4,75)\}$, V5: $\{(3,15),(4,50),(4,45)\}$, and V20: $\{(4,221),(4,27),(4,1)\}$.

Some Remarks:

Two final remarks on the efficiency and algorithmic issues are as follows.

Remark 1: The core of our results is the bit-wise Algorithm 2. The efficiency of this bit-based algorithm can be further improved for codes with special automorphism structures, including the Margulis codes, the Ramanujan–Margulis codes, the lifted LDPC codes based on permutation matrices, etc. Taking the cyclically lifted LDPC codes with lifting factor L as an example, the complexity of Algorithm 3 for k-out trapping sets can be reduced by a factor of L, which makes our algorithm especially suitable for the protograph codes based on small base codes with high lifting factor L[52].

Remark 2: For large k, exhaustively enumerating minimum k-out trapping sets becomes tricky especially when the minimal variable node degree is $\leq k$. If a single variable node v is of degree $\deg(v) \leq k$, then $\{v\}$ itself is the minimum k-out trapping set and would thus dominate the outputs of Algorithm 3 and make the results useless. For example, searching for minimum 2-out trapping sets of an irregular code with degree-2 nodes always returns trapping sets containing single degree-2 nodes. A simple remedy is to carefully select the $\{(x_{i_1}, c_{j_1}), \ldots, (x_{i_k}, c_{j_k})\}$ of interest in Line 4 of Algorithm 3 and preclude uninteresting combinations. A more effective method would require unambiguous definitions of "nontrivial" k-out trapping sets and is currently under investigation.

VII. Numerical Experiments

The numerical experiments will be divided into four categories: i) non-sparse codes, ii) LDPC codes with random construction, iii) error-floor optimized LDPC codes, and iv) algebraically constructed codes. We use SSES and k-TSES as shorthand for the minimum stopping set exhaustive search and the minimum k-out trapping set exhaustive search algorithms discussed in the previous sections.

A. Non-Sparse Codes

1) The $(23, 12)$ Binary Golay Code: The standard parity-check matrix of the Golay code is described by $\boldsymbol{H} = [\boldsymbol{H}'\ \boldsymbol{I}]$, where $\boldsymbol{I}$ is an 11×11 unity matrix and $\boldsymbol{H}'$ is as follows:

$$\boldsymbol{H} = [\boldsymbol{H}'\ \boldsymbol{I}], \quad \boldsymbol{H}' = \begin{pmatrix} 100111000111 \\ 101011011001 \\ 101101101010 \\ 101110110100 \\ 110011101100 \\ 110101110001 \\ 110110011010 \\ 111001010110 \\ 111010100011 \\ 111100001101 \\ 011111111111 \end{pmatrix}.$$

The minimal stopping distance of the Golay code is 4 and all 130 minimal stopping sets can be found by the proposed SSES in 2 seconds. All bits are involved in at least one stopping set of size 4 and thus their BERs are of similar magnitudes. Fig. 8 compares the upper bound (UB), the composite upper bound (C-UB) determined by the branch-and-bound method in Section VI-B, the Monte Carlo simulation (MC-S), and the tight lower bound (LB), for bits 0, 5, and 20. As illustrated, C-UB and LB tightly bracket the MC-S results, which shows that the UB and C-UB are capable of decoupling even *non-sparse* Tanner graphs with plenty of cycles. However, the non-sparsity slows down SSES considerably, compared to sparse LDPC codes of similar sizes, due to the higher frequency of invoking the pivoting rule. Nonetheless, the proposed SSES still demonstrates superior performance when compared to the naive brute-force search. For comparison, the extremely short codeword length of the Golay code makes it still possible to use the naive brute-force search, which requires $\binom{23}{4} = 490,314$ trials.

B. LDPC Codes With Random Construction

1) A $(3,6)$ LDPC Code With $n = 50$: A $(3,6)$ LDPC code with $n = 50$ is randomly generated, and the UB, the C-UB, the MC-S, and the tight LB are performed on bits 0, 26, and 19, as plotted in Fig. 9, and the statistics of all 50 bits are provided in Table I(a). The UB is tight in the asymptotic order for *all* bits while for 34 bits, the UB is also tight in multiplicity. Among the 16 bits for which UB is not tight in multiplicity, 11 bits are within a factor of three of the actual multiplicity, which is

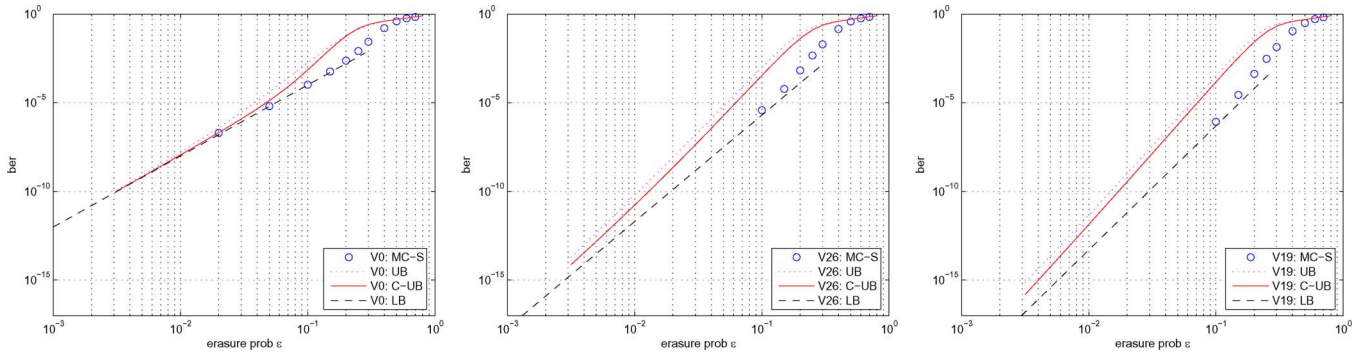


Fig. 9. Comparisons among the UB, the C-UB, the MC-S, and the LB for bits 0, 26, and 19 of a randomly generated $(3,6)$ LDPC code with $n = 50$. The asymptotic (order, multiplicity) pairs of the UB, the C-UB, and the actual BER are V0: $\{(4,1),(4,1),(4,1)\}$, V26: $\{(6,2),(6,4),(6,2)\}$, and V19: $\{(7,135),(7,10),(7,5)\}$.

TABLE I

PERFORMANCE STATISTICS FOR COMPLETE RANDOMLY CONSTRUCTED $(3,6)$ LDPC CODES: "Num. bits" IS THE NUMBER OF BITS WITH THE SPECIFIED ASYMPTOTIC ORDER. "order*" IS THE NUMBER OF BITS WITH UBS TIGHT ONLY IN THE ORDER. "+multi*" IS THE NUMBER OF BITS WITH UBS TIGHT BOTH IN THE ORDER AND IN THE MULTIPLICITY. "order >" IS THE NUMBER OF BITS WITH A UB OF THE SPECIFIED ORDER WHILE NO BRACKETING LOWER BOUND CAN BE ESTABLISHED

(a) $n = 50$

Order	3	4	5	6	7
Num. bits	3	11	10	20	6
order*			3	8	5
+ multi*	3	11	7	12	1

(b) $n = 72$

Order	2	4	5	6	7	8
Num. bits	4	4	5	28	28	3
order*			1	11	26	1
+ multi*	4	4	4	17	2	

(c) $n = 144$

Order	2	5	7	8	9
order*	4	3	7	27	2
order>			6	90	5

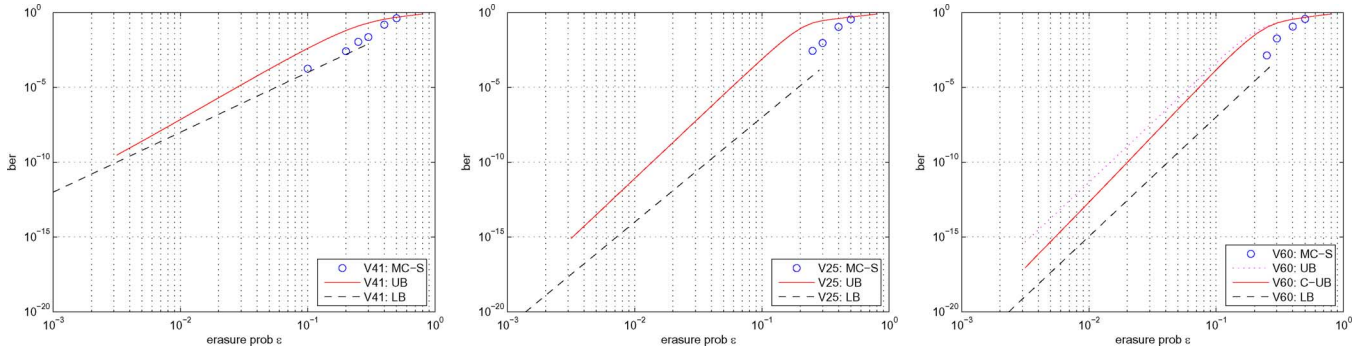


Fig. 10. Comparisons among the UB, the C-UB, the MC-S, and the LB for bits 41, 25, and 60 of a randomly generated $(3,6)$ LDPC code with $n = 72$. The asymptotic (order, multiplicity) pairs of the UB, the C-UB, and the actual BER are V41: $\{(4,1),-,(4,1)\}$, V25: $\{(7,1),-,(7,1)\}$, and V60: $\{(7,19),(8,431),(8,11)\}$.

obtained from the bit-wise exhaustive list of minimum stopping sets obtained from the SSES.

In contrast with the Golay code example, the tight performance can be attributed to the sparse connectivity of the corresponding Tanner graph. The smaller sizes of variable and check node degrees result in a considerably smaller decoding tree and fewer chances of invoking the pivoting rule, both of which favor significantly the SSES algorithm. As can be seen in Fig. 9(right), the composite approach C-UB possesses the greatest advantage over simple UBs that are not tight in multiplicity. The C-UB and the LB again tightly bracket the asymptotic performance.

2) A $(3,6)$ LDPC Code With $n = 72$: The UB, the C-UB, the MC-S, and the tight LB are applied to bits 41, 25, and 60, as plotted in Fig. 10 and the complete statistics of all 72 bits are included in Table I(b). Almost all asymptotic orders and most multiplicities can be captured by the UB with only two exception bits. Both of the exception bits are of order 8, which is computed by the branch-and-bound SSES for each bit respectively.

3) A $(3,6)$ LDPC Codes With $n = 144$: Complete statistics of all 144 bits are presented in Table I(c), and we start to see many examples (101 out of 144 bits) in which the simple UB is not able to capture the asymptotic order and we have to resort to the C-UB for tighter results. It is worth noting that even the simple UB is able to identify some bits with BER of order 9, which requires $\binom{143}{8} \approx 3.55 \times 10^{12}$ trials if a naive brute-force search is employed.[7]

C. The Error-Floor Optimized LDPC Codes

For the randomly constructed LDPC codes in the previous examples, the *frame-wise* minimal stopping distances range from 2 to 4, which does not present any challenge for the proposed frame-wise SSES. That is the reason why we deliberately omitted the discussion of the frame-wise minimum stopping set

[7]For the bit-wise brute-force search, since the target bit is always included, we need only to search through $\binom{143}{8}$ possibilities instead of $\binom{144}{9}$.

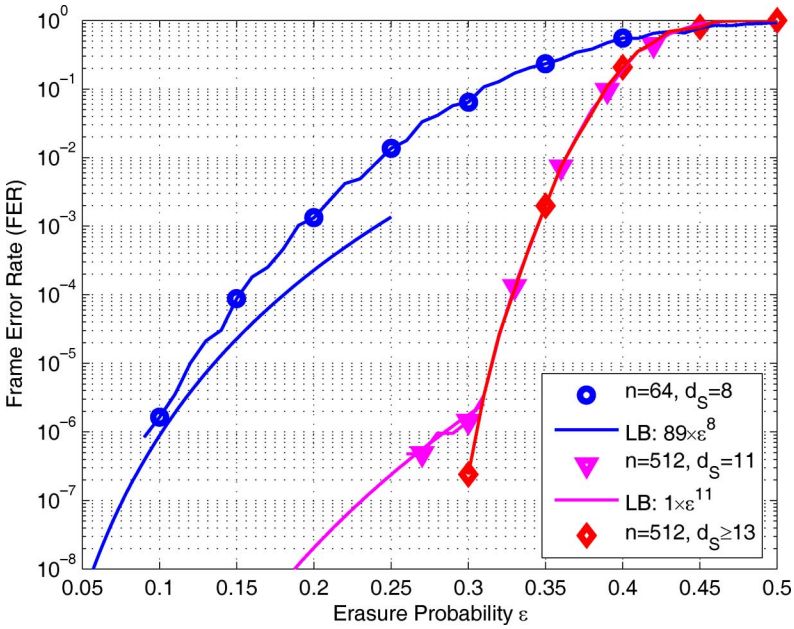


Fig. 11. Comparisons among three different regular $(3, 6)$ codes in the following order. A short $n = 64$ code with minimal stopping distance d_{S} optimized to 8 and multiplicity 89 confirmed by stopping set exhaustive search. A randomly chosen $n = 512$ code with $d_{\mathrm{S}} = 11$ and multiplicity 1 determined by the stopping set exhaustive search algorithm. An optimized $n = 512$ code with $d_{\mathrm{S}} \geq 13$. For the last code, one stopping set of size 24 is identified during the Monte Carlo simulation.

exhaustive enumeration in the previous examples and focused mainly on using the SSES as a bit-wise exhaustive search or a bit-wise upper bound. For the following, we consider randomly constructed regular and irregular codes that have been optimized for the error-floor performance by some other algorithms so that their frame-wise minimal stopping distances are between 8 and 13. The SSES will provide exhaustive lists of minimum stopping sets. k-TSES will also be applied to different codes of length ≈ 500. This range of codeword lengths is large enough to be considered practical and is beyond the reach of any naive brute-force search algorithm. On the other hand, it is still short enough so that its minimum stopping/trapping sets can be exhaustively enumerated by the proposed SSES and k-TSES algorithms.

The codes discussed herein are optimized by the code annealing method, presented in a companion paper [18], that uses the exhaustive lists of minimum stopping/trapping sets generated by SSES and k-TSES as the objective functions for error-floor optimization. For all the Monte Carlo simulation results, 100 frame error events are observed for each simulation point.

Regular $(3, 6)$ LDPC Codes on BECs:

Experiment 1: An Optimized $(3, 6)$ LDPC Code With $n = 64$: Using the exhaustive list as the objective function, its minimal stopping distance has been optimized to 8. Even for codes of this small size 64, a brute-force search requires $\binom{64}{8} \approx 4.43 \times 10^9$ computations. All 89 minimum stopping sets are identified by SSES, and the exhaustive list provides a tight lower bound $89 \times \epsilon^8$ for the FER. The corresponding Monte Carlo simulation results and the tight lower bound are plotted in Fig. 11. For comparison, the girth for any $(3, 6)$ code of length $n = 64$ is upper-bounded by 6, counting both variable and check nodes, while the minimum stopping set of this code consists of eight variable nodes. This is a concrete example showing the limitations of using girth as an objective function. Among all $(3, 6)$ codes with $n = 64$, many codes are "girth-optimal" but only a tiny fraction of them has $d_{\mathrm{S}} \geq 8$. Even with a limited attainable girth size (three variable nodes), one can still construct codes with much larger minimal stopping distances once we directly use the minimal stopping distance itself as the objective function.

Experiment 2: A Randomly Chosen $(3, 6)$ LDPC Code With $n = 512$: It is well known that regular $(3, 6)$ codes generally have better error-floor performance with random construction than irregular codes with many degree-2 variable nodes. We arbitrarily choose one realization from the random code ensemble. This $n = 512\,(3, 6)$ code has minimal stopping distance $d_{\mathrm{S}} = 11$ with multiplicity 1. The SSES is able to identify the only one minimum stopping set, and the resulting tight lower bound coincides with Monte Carlo simulation when $\epsilon \leq 0.3$; see Fig. 11. Using Monte Carlo simulation, one can also identify the same minimum stopping set, but cannot claim with 100% certainty that there is no other stopping set of size 11, which is a major distinction between the proposed SSES algorithm and any randomized enumeration algorithm, e.g., the error impulse methods.

Experiment 3: An Optimized $(3, 6)$ LDPC Code With $n = 512$: An optimized $(3, 6)$ code with $n = 512$ is constructed by using the exhaustive list of minimum stopping sets as the objective function. After optimization, all stopping sets of size ≤ 12 are removed. The minimal stopping distance must be ≥ 13. One stopping set of size 24 is observed during the Monte Carlo simulation, which gives us an upper bound on the minimal stopping distance. As can be seen in Fig. 11, for this optimized code with minimal stopping distance ≥ 13,

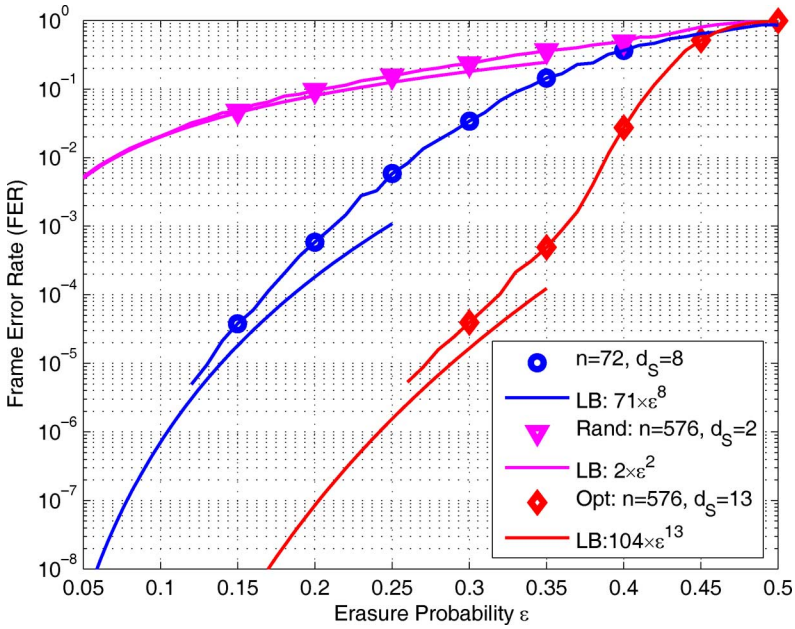


Fig. 12. Comparisons among three different irregular $(\lambda_1(x), \rho_1(x))$ codes with their minimal stopping distances d_{S} and multiplicities m_{S} determined by the minimum stopping set exhaustive search algorithm. One of these is a short $n = 72$ code with d_{S} optimized to 8 and $m_{\mathrm{S}} = 71$. A second, a randomly chosen $n = 576$ code, denoted by "Rand," with $d_{\mathrm{S}} = 2$ and $m_{\mathrm{S}} = 2$, has the poorest FER performance due to the existence of small stopping sets. In contrast with "Rand," the error floor of an optimized $n = 576$ code with $d_{\mathrm{S}} = 13$ and $m_{\mathrm{S}} = 104$, denoted by "Opt," is improved over its random counterpart. Tight lower bounds derived from the exhaustive lists of minimum stopping sets are plotted for comparison.

no error floor can be observed by software simulations until FER $= 10^{-7}$.

Irregular LDPC Codes on BECs:

A more interesting subject is irregular LDPC codes with degree distributions optimized for threshold performance [11], which turns out to be the most computationally friendly instance for the proposed SSES algorithm. For rate-$1/2$ irregular codes of length 512, the searchable range t can be extended to 13, which is $+2$ better than its regular code counterpart. All irregular codes considered in this subsection have the following variable and check node degree distributions [11] optimized for threshold performance:

$$\lambda_1(x) = 0.41667x + 0.16667x^2 + 0.41667x^5$$
$$\text{and} \quad \rho_1(x) = x^5,$$

which has asymptotic erasure probability threshold $\epsilon^* \approx 0.4775$ [53]. Codes of different codeword lengths and construction methods are discussed as follows and their results are illustrated in Fig. 12. For comparison, to enumerate exhaustively all stopping sets of size 13 for an $n = 576$ code requires $\binom{576}{13} \approx 1.08 \times 10^{26}$ trials if a naive brute-force approach is adopted.

Experiment 4: An Optimized $n = 72$ Irregular LDPC Code: Using the above (λ_1, ρ_1) degree distributions, an irregular code is constructed with optimized minimal stopping distance $d_{\mathrm{S}} = 8$ as described in [18], and all of the 71 minimum stopping sets can be identified. Both codes in Experiments 1 and 4 are of minimal stopping distance 8 and of similar sizes, which shows that although there are inherently many "bad" irregular codes, the error floor of "good" irregular codes is still comparable to that of good regular codes, provided proper optimization is performed using the exhaustive list of minimum stopping sets as the objective functions.

Experiment 5: An Optimized $n = 576$ Irregular LDPC Code: Using the same (λ_1, ρ_1) degree distributions with $n = 576$, one can construct an irregular code with minimal stopping distance 13 [18]. All 104 minimum stopping sets have been identified by the SSES algorithm and a tight lower bound is provided. For comparison, we also plot the FER curve of a typical irregular code of the same degree distributions (λ_1, ρ_1). At $\epsilon = 0.3$, an improvement by a factor $10,000$ is seen when compared to the typical code performance.

Irregular LDPC Codes on Gaussian Channels:

We focus our trapping set discussions on binary-input additive white Gaussian noise channels (BiAWGNCs). Namely, the observation vector $\boldsymbol{y} = (\mathbf{1} - 2\boldsymbol{x}) + \sigma\boldsymbol{n}$, where $\mathbf{1}$ is an all-one vector, $\boldsymbol{n}$ is a standard Gaussian vector with unity covariance matrix, and $1/\sigma^2$ is the signal to noise (power) ratio. Only irregular LDPC codes will be considered and their common degree distributions, optimized for threshold performance, are as follows:

$$\lambda_2(x) = 0.31961x + 0.27603x^2 + 0.01453x^5 + 0.38983x^6$$
$$\text{and} \quad \rho_2(x) = 0.50847x^5 + 0.49153x^6$$

with an asymptotic threshold $\sigma^* \approx 0.937$[53]. All the minimum stopping sets or k-out trapping sets are identified by the SSES and k-TSES discussed previously. The corresponding Monte Carlo simulation results are in Fig. 13.

Experiment 6: "Rand" is an $n = 512$ code arbitrarily chosen from the (λ_2, ρ_2) irregular code ensemble with an additional

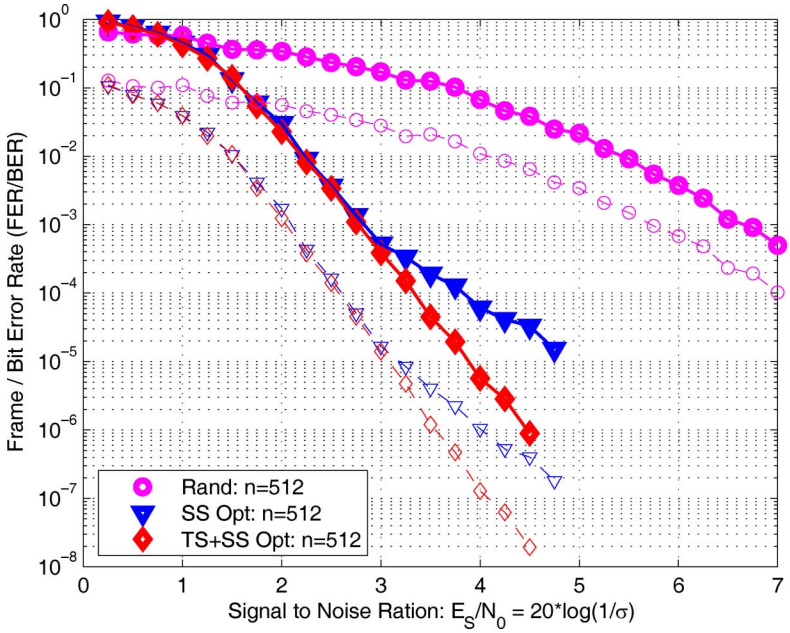


Fig. 13. Monte Carlo simulations for irregular $(\lambda_2(x), \rho_2(x))$, $n = 512$ codes with different constructions: "Rand," "SS Opt," and "TS+SS Opt" as described in Experiments 6, 7, and 8, respectively. Their corresponding (min_size,multiplicity) of the k-out trapping sets for $k = 0, 1$, respectively, are: "Rand" $(2, 1), (2, 8)$; "SS Opt" $(13, 40), (5, 4)$; "TS+SS Opt" $(11, 12), (10, 24)$. Eighty decoding iterations are performed and 100 error events are observed for each simulation point.

constraint that there are no parallel edges in the corresponding bipartite Tanner graph. Without any graph optimization, the performance is not impressive for irregular codes of this short length. To further pinpoint the cause of the bad performance, its minimum k-out trapping sets are exhaustively enumerated for $k = 0, 1$ (the 0-out trapping sets correspond to the stopping sets), and their (min_size,multiplicity) pairs are $(2, 1)$ and $(2, 8)$, respectively. Its short, minimal trapping distances explain the poor performance of this code.

Experiment 7: "SS Opt" is an $n = 512$ code with degree distributions (λ_2, ρ_2) but optimized with respect to the minimal stopping distance by the code annealing method in [18]. Its corresponding (min_size,multiplicity) of the k-out trapping sets for $k = 0, 1$ are $(13, 40)$ and $(5, 4)$, respectively. This example confirms the phenomenon that codes with error floor performance optimized for BECs also exhibit good error floor performance for other channels as well since the minimal 1-out trapping distance has been improved to 5 as a byproduct of optimizing the minimal stopping distance.

Experiment 8: "TS+SS Opt" is an $n = 512$ code with degree distributions (λ_2, ρ_2) but optimized with respect to both the minimal 1-out trapping distance and the minimal stopping distance (the minimal 0-out trapping distance) by the code annealing method. Its corresponding (min_size,multiplicity) of the k-out trapping sets for $k = 0, 1$ are $(11, 12)$ and $(10, 24)$, respectively. Namely, for any set of contaminated bits of size ≤ 9, there are at least two extrinsic messages that might break this error pattern. The BEC performance of "TS+SS Opt" is not as good as "SS Opt," since the minimal stopping distance of the former is 11 as compared to 13. However, the minimal 1-out trapping distance optimization further brings down the error floor for the non-erasure BiAWGNCs. No error floor is observed until FER $= 10^{-6}$.

Remark: When applied to codes of larger sizes, e.g., $n \geq 1000$, the searchable range t of the SSES and k-TSES also increases when compared to $n \approx 500$ codes due to the larger girth. Nevertheless, the minimal stopping/trapping distances for longer codes generally grow faster than the searchable range t and fewer tight results can be obtained for the error-floor optimized regular/irregular codes of length $n \geq 1000$.

D. Algebraically Constructed Codes

We apply the exhaustive search algorithms to the following well-studied codes: the $(155, 64, 20)$ Tanner code [29], the Ramanujan–Margulis $(2184, 1092)$ code with $q = 13$ and $p = 5$ [47], and the $(672, 336)$ Margulis code with $p = 7$ [48]. Since all these codes are regular codes of minimal variable node degree 3, the 2-out trapping set exhaustive search is performed in addition to the stopping set and the 1-out trapping set exhaustive search.

1) The $(155, 64, 20)$ Tanner Code: The minimal Hamming distance of this code is known to be 20, which is computed by MAGMA, a software suite, taking advantage of the corresponding algebraic structure [54]. One can easily locate a stopping set of size 18 by Monte Carlo simulation, which serves as an upper bound on the minimal stopping distance. The SSES and k-TSES are then performed for different types of error-prone patterns of this code.

For stopping sets (0-out trapping sets), all candidates of size ≤ 12 have been exhaustively examined and none of them is a stopping set, which results in a lower bound of 13 on the minimal stopping distance. For 1-out trapping sets, all candidates of size ≤ 11 have been exhaustively examined and none of them is a 1-out trapping set, which results in a lower bound of 12 on the minimal 1-out trapping distance. For 2-out trapping sets, we are able to enumerate exhaustively all 465 minimum 2-out trapping sets, which are of size 8. All 465 minimum 2-out trapping sets

can be obtained from the following five representatives by the automorphisms discussed in [29]:

$$7, 17, 19, 33, 66, 76, 128, 140$$
$$7, 31, 33, 37, 44, 65, 100, 120$$
$$1, 19, 63, 66, 105, 118, 121, 140$$
$$44, 61, 65, 73, 87, 98, 137, 146$$
and $$31, 32, 37, 94, 100, 142, 147, 148.$$

Recently, the instanton analysis in [9] and [16] identified some dominating error patterns, called *instantons*. Each of the reported instantons contains one minimum 2-out trapping set as a substructure.[8] These results again confirm that the dominating error patterns are generally k-out trapping sets with small k.

2) The Ramanujan–Margulis $(2184, 1092)$ *Code w.* $q = 13$, $p = 5$*:* MacKay *et al.* [7] first pointed out there is a codeword of Hamming weight 14 for the Ramanujan–Margulis $(2184, 1092)$ code with $q = 13$ and $p = 5$. By the automorphism of the code, there are at least 1092 codewords of such weight. Using a refinement of the error impulse method, Hu *et al.* [33] also identified the same 1092 codewords for the Ramanujan–Margulis $(2184, 1092)$ code. The remaining question is whether there is any other codeword of equal or smaller Hamming weight. There are at least two approaches to answering this question: one is to use the algebraic structure of the Ramanujan–Margulis code and mathematically prove that there is no other codeword of equal or smaller weight. The second approach is by exhaustive search over all possible codewords of smaller weights.

Our SSES algorithm shows that there are "only" 1092 stopping sets of size 14 and there is no stopping set of smaller size. Since any codeword must be a stopping set, the minimal Hamming distance of the Ramanujan–Margulis $(2184, 1092)$ code must be 14 and its multiplicity must be 1092. What MacKay *et al.* and Hu *et al.* found are indeed the minimum codewords.

We also exhaustively search for this code's k-out trapping sets for $k = 1, 2$, respectively. All candidates for 1-out trapping sets of size ≤ 12 and all candidates for 2-out trapping sets of sixe ≤ 9 have been examined and none of them is a trapping set, which results in lower bounds of 13 and 10 on the minimal 1-out and 2-out trapping distances, respectively. The lack of trapping sets of small sizes explains why the dominating error event of the Ramanujan–Margulis $(2184, 1092)$ code concerns the small codewords instead of the small near-codewords.

3) The $(672, 336)$ *Margulis Code With* $p = 7$ *[48]:* It is known that this code has a codeword of length 16 [9]. The SSES and k-TSES are again performed to search for the error-prone patterns of this code.

For stopping sets, the SSES shows that there is no stopping set of size ≤ 13. For k-out trapping sets, there is no 1-out trapping set of size ≤ 12 and there is no 2-out trapping set of size ≤ 9. Since no exhaustive list of any type of minimum error-prone patterns can be obtained, our results serve only as lower bounds on the sizes of minimum error-prone patterns. For comparison, all stopping sets observed by the Monte Carlo simulation are of size ≥ 16 and no tighter upper bound can be obtained for the minimal stopping distance. Combined with our results, the minimal stopping and the Hamming distances of the $(672, 336)$ Margulis code are between 14 and 16.

[8] In [9] and [16], the constituent permutation matrices are cyclically shifted to the right, which results in different indexing than the original construction in which the permutation matrices are cyclically shifted to the left [29]. An index remapping is necessary for direct comparison.

TABLE II
SUMMARY OF BOUNDING THE MINIMAL HAMMING, STOPPING, 1-OUT TRAPPING AND 2-OUT TRAPPING DISTANCES FOR THE TANNER $(155, 64, 20)$ CODE, THE RAMANUJAN–MARGULIS $(2184, 1092)$ CODE WITH $q = 13$ AND $p = 5$, AND THE MARGULIS $(672, 236)$ CODE WITH $p = 7$. THE PAIR (d, m) REPRESENTS THE MINIMAL DISTANCE d (OR THE RANGE OF THE MINIMAL DISTANCE) AND HOW MANY CODEWORDS/STOPPING SETS/TRAPPING SETS ARE OF WEIGHT d. UPPER BOUNDS ARE OBTAINED FROM ENUMERATIONS, WHILE LOWER BOUNDS ARE FROM THE EXHAUSTIVE SEARCH ALGORITHM PROVIDED HEREIN. † IS OBTAINED USING COMPUTER SEARCH VIA MAGMA [29]

	Tanner(155,64)	R-M(2184,1092)	Marg(672,336)
Hamming Dist.			
by enum.	$(20, ?)^{\dagger}$	$(\leq 14, ?)$	$(\leq 16, ?)$
by exhaust'n	$(\geq 13, ?)$	$(14, 1092)$	$(\geq 14, ?)$
Stopping Dist.			
by enum.	$(\leq 18, ?)$	$(\leq 14, ?)$	$(\leq 16, ?)$
by exhaust'n	$(\geq 13, ?)$	$(14, 1092)$	$(\geq 14, ?)$
1-Out Trap. Dist.			
by exhaust'n	$(\geq 12, ?)$	$(\geq 13, ?)$	$(\geq 13, ?)$
2-Out Trap. Dist.			
by exhaust'n	$(8, 465)$	$(\geq 10, ?)$	$(\geq 10, ?)$

Results for the three algebraically constructed codes are summarized in Table II.

Note: The SSES and k-TSES are directly applied to these codes without taking advantage of their algebraic structures except for the automorphism as discussed in the final remarks of Section VI-D.

VIII. CONCLUSION AND FUTURE RESEARCH

In this paper, the problem of exhaustively enumerating minimum error-prone patterns for arbitrary finite-length LDPC codes has been discussed, and efficient exhaustive search algorithms have been proposed for both the minimum stopping sets and the minimum trapping sets for general channels. The algorithm, based on the decoding tree of the iterative decoder, is equivalent to a narrowing search of minimum error-prone patterns. The optimality of the algorithms has been proven and several improvements have been made to further improve the efficiency, including a composite branch-and-bound approach. All the proofs and derivations are based on the proposed Boolean expression framework for iterative decoding functions over binary erasure channels. Extensive numerical experiments have been conducted on different codes, including randomly constructed codes, error-floor optimized codes, and algebraically constructed codes. For rate-$1/2$ codes of short practical lengths $n \approx 500$, both the minimum stopping sets and the minimum trapping sets can be enumerated exhaustively. The exhaustive list of error-prone patterns finds applications in code behavior analysis, upper- and lower-bounding code performance, and error-floor optimization.

The NP-completeness of the minimum trapping set exhaustive search problem has also been established, which shows the inherent hardness of the problem. The NP-completeness statement is genuinely an asymptotic worst case analysis and implies that there is little chance that an efficient algorithm for exhaustively enumerating minimum stopping/trapping sets exists for

long LDPC codes. However, it does not preclude the existence of efficient algorithms for codes of short lengths. The algorithms discussed in this work serve as the first successful step for codes of short practical lengths.

We conclude this paper by providing a short, noncomprehensive list of potential improvements.

1) Several parts of the existing algorithms can be further improved, including but not limited to a better design of the socket-finding module and the partitioning events for the composite branch-and-bound approach. The effects of different partitioning events on the efficiency of the algorithms can easily be as large as a factor of 100. With a considerable amount of design freedom over the partitioning events, we believe the full power of the branch-and-bound approach has not been realized. One possible approach is to use the information from the Monte Carlo simulation as a design guideline for partitioning events.
2) Better implementations with different data structures, machine-optimized software codes, or even a hardware implementation of the algorithms could improve the applicable range of the proposed algorithms to $n \approx 1000$ or larger, which will have even broader impact on finite length LDPC code analysis and design.
3) The minimum trapping sets are the error-floor-determining factor for general channels, and the sizes of interest are generally less than 15, as opposed to the minimum stopping distances of interest, which are generally 15–30. The above reasons make the exhaustive search for trapping sets an especially appealing problem due to its practical importance and its much smaller search range. In this work, the minimum trapping set exhaustive search problem has been solved by the reduction to minimum stopping set exhaustive enumeration. Directly designing algorithms for the minimum trapping set exhaustive search algorithm would further enhance its performance and extend its applications to longer codes.

APPENDIX I
A PROOF OF THE CORRECTNESS OF ALGORITHM 2

The proof of the correctness of Algorithm 2 consists of the following two steps. For any $\mathcal{T}$ constructed by Algorithm 2 after t iterations, let $f_{\mathcal{T},t}(\boldsymbol{y})$ denote the corresponding iterative decoding function (after appending the dummy leaf units). We will show that i) $\mathsf{E}\{f_{\mathcal{T},t}(\boldsymbol{y})\}$ is no larger than the UB_i computed by Algorithm 2, and ii) the decoding function $f_i(\boldsymbol{y}) \leq f_{\mathcal{T},t}(\boldsymbol{y})$ for all $\boldsymbol{y}$. Since ii) is a restatement of *Theorem 3*, we will prove i) in this appendix by the following two propositions and leave ii) to Appendix II.

Proposition 4: In the tree $\mathcal{T}$ generated by Algorithm 2, all messages entering the same variable nodes must be independent. Namely, if $g(\boldsymbol{y})$ and $h(\boldsymbol{y})$ enter the same variable node, then they share no common determining variable.

Proof: A stronger statement of the above proposition is that the youngest common ancestor of any two variable nodes with the same active inputs must be a check node. A simple proof that this new statement implies the proposition is as follows. Suppose there exist two distinct, dependent messages $g(\boldsymbol{y})$ and $h(\boldsymbol{y})$ entering the same variable node v. Then $g(\boldsymbol{y})$ and $h(\boldsymbol{y})$ share at least one common determining variable, denoted by y_k, and there must be at least one variable node with active observation y_k in each of the subtrees of $g(\boldsymbol{y})$ and $h(\boldsymbol{y})$. Pick one such variable node from each subtree. The youngest common ancestor of the selected pair of variable nodes is thus v, which contradicts the assumption that the youngest common ancestor of any pair of variable nodes with the same active observation is always a check node.

We then prove by induction that all youngest common ancestors, if they exist, are check nodes. During the initialization, there is only one root variable node with active observation y_i in $\mathcal{T}$, and the above statement obviously holds. Suppose the statement holds after t iterations of the REPEAT-UNTIL loop in Line 2. For the $(t+1)$th iteration of adding a (c_j, v_k) leaf unit into $\mathcal{T}$, we use the same notation as in Algorithm 2 for which v^* denotes the existing variable node with active input y_k and jointly v (the variable node in the newly added (c_j, v_k) leaf unit) and v^* have a common youngest ancestor $\mathsf{yca}(v, v^*)$. Consider two separate cases: $\mathsf{yca}(v, v^*)$ is a check node and $\mathsf{yca}(v, v^*)$ is a variable node. For the case in which $\mathsf{yca}(v, v^*)$ is a check node, no pivoting is involved so one needs only to show that after adding v there is no other variable node v'' with active observation y_k such that the youngest common ancestor of v and v'' is a variable node. We prove this statement again by contradiction. Suppose there exists such a v''. Since $\mathsf{yca}(v, v^*)$ is the youngest of all common ancestors and is a check node, $\mathsf{yca}(v, v^*)$ must be a strict descendent of $\mathsf{yca}(v, v'')$. Therefore, $\mathsf{yca}(v^*, v'')$ must equal $\mathsf{yca}(v, v'')$. The above implies that $\mathsf{yca}(v^*, v'')$ is a variable node, which contradicts the induction assumption that after t iterations all youngest common ancestors of any pair of variable nodes with the same active observation must be check nodes.

Before proving the case in which $\mathsf{yca}(v, v^*)$ is a variable node, we first show that if $\mathsf{yca}(v, v^*)$ is a variable node, then before pivoting, there are exactly two messages $g(\boldsymbol{y})$ and $h(\boldsymbol{y})$ entering $\mathsf{yca}(v, v^*)$ such that they share a single common determining variable y_k. A short argument is as follows. Since $\mathsf{yca}(v, v^*)$ is a youngest common ancestor of nodes v and v^*, there must be two *interior-node-disjoint paths* from $\mathsf{yca}(v, v^*)$ to v and v^*, which implies that v and v^* must contribute to different messages entering $\mathsf{yca}(v, v^*)$. Therefore, there are at least two incoming messages having y_k as their determining variables. Suppose there is another message $g'(\boldsymbol{y})$ entering $\mathsf{yca}(v, v^*)$ which also has y_k as its determining variable. Then there must be a variable node v'' with active y_k in the subtree corresponding to $g'(\boldsymbol{y})$. Then $\mathsf{yca}(v^*, v'')$ must equal $\mathsf{yca}(v, v^*)$, which contradicts the induction assumption $\mathsf{yca}(v^*, v'')$ being a check node. Moreover, the same argument also shows that all variable nodes with active input y_k and in the subtree rooted at $\mathsf{yca}(v, v^*)$ must be within the subtrees corresponding to the two messages $g(\boldsymbol{y})$ and $h(\boldsymbol{y})$.

To show that after pivoting (i.e., when $\mathsf{yca}(v, v^*)$ is a variable node) the same statement holds, we first note that after pivoting, all variable nodes with active input y_k are replaced either by 0 or by 1 with the only exception being the root of the left subtree; see Fig. 6 for illustration. Suppose after pivoting there exist two variable nodes v' and v'' with active input y_k

and the youngest common ancestor $\mathsf{yca}(v', v'')$ being a variable node. Since the tree structure outside the two duplicated subtrees remains unchanged, one of v' and v'' must be the root of the left subtree, say v', and the other node v'' must not be in the left and right subtrees of Fig. 6. Otherwise, the induction assumption does not hold. Since v'' cannot be a strict descendent of $\mathsf{yca}(v, v^*)$ as $\mathsf{yca}(v, v^*)$ must carry only two incoming messages, $\mathsf{yca}(v', v'')$ must be a strict ancestor of $\mathsf{yca}(v, v^*)$. The facts that after pivoting, v'' is not a descendent of $\mathsf{yca}(v, v^*)$ and that $\mathsf{yca}(v', v'')$ is a strict ancestor of $\mathsf{yca}(v, v^*)$ imply that before pivoting, the youngest common ancestor of nodes v^* and v'' is also $\mathsf{yca}(v', v'')$, which contradicts the assumption of induction.

Another possibility is that after pivoting, there exist two active nodes v' and v'', both of which are accepting the same free observation y_χ, $\chi \neq k$, and the youngest common ancestor of $\mathsf{yca}(v', v'')$ is a variable node. We consider the following four cases concerning the tree configuration after pivoting: (a) neither v' nor v'' is a descendent of $\mathsf{yca}(v, v^*)$, (b) one of them, say v', is a descendent of $\mathsf{yca}(v, v^*)$ while the other is not, (c) both of them are in the same left (or right) subtree, and (d) v' and v'' are in the left and the right subtrees, respectively. For case (a), we first note that the pivoting rule affects only the descendents of $\mathsf{yca}(v, v^*)$. Therefore, v' and v'' are intact during pivoting and their youngest common ancestor must be a check node due to the induction assumption. For case (b), the youngest common ancestor of v' and v'' after pivoting must be an ancestor of $\mathsf{yca}(v, v^*)$ since v' is a descendent of $\mathsf{yca}(v, v^*)$ but v'' is not. Since v' must be obtained from a variable node with active y_χ input during the tree-duplicating step of pivoting, we use v''' to denote the corresponding variable node of v' before pivoting. Then the $\mathsf{yca}(v', v'')$ (after pivoting) is also the youngest common ancestor of (v''', v'') (before pivoting). Therefore $\mathsf{yca}(v', v'')$ must be a check node. For case (c), since v' and v'' are in the same left (or right) subtree, their youngest common ancestor must also be in the same left (or right) subtree. As a result, the relative locations among v' and v'', and their youngest common ancestor must be preserved during pivoting. By the induction assumption, $\mathsf{yca}(v', v'')$ must be a check node as well. For case (d), the youngest common ancestor of v' and v'' is the check node of degree 2 taking the two subtrees as its children. For all cases (a)–(d), the youngest common ancestor between v' and v'' must be a check node.

From the above reasoning, after $(t+1)$ iterations, all youngest common ancestors of any pair of variable nodes with the same active observation must be check nodes. *Proposition 4* is thus proved by induction. □

Proposition 5: The UB_i computed in Algorithm 2 is an upper bound on $\mathsf{E}\{f_{\mathcal{T},t}(\boldsymbol{y})\}$. Furthermore, UB_i is of the same asymptotic order as that of $\mathsf{E}\{f_{\mathcal{T},t}(\boldsymbol{y})\}$.

Proof: By *Proposition 4*, all messages entering the variable nodes are independent, and one can thus use Rule 0 to compute the exact expectation of the outgoing message of a variable node. On the other hand, Rule 1 gives us an upper bound on the expectation of the outgoing message of a check node, namely

$$\begin{aligned}\mathsf{E}\{f_c(\boldsymbol{y})\} &\leq \mathrm{UB}_{\mathsf{E}\{f_c(\boldsymbol{y})\}} \\ &= \mathsf{E}\{g(\boldsymbol{y})\} + \mathsf{E}\{h(\boldsymbol{y})\} - \mathsf{E}\{g(\boldsymbol{y})\}\mathsf{E}\{h(\boldsymbol{y})\}.\end{aligned}$$

Using Rule 0 for $\mathsf{E}\{f_v(\boldsymbol{y})\}$ and Rule 1 for $\mathsf{E}\{f_c(\boldsymbol{y})\}$, we are able to compute their upper bounds assuming we know the expectations of all input messages, which are unfortunately not available and are the quantities to be bounded. Fortunately, both Rule 0 and Rule 1 are monotonically increasing functions with respect to the input expectations. We can thus substitute all input expectations with their upper bounds and obtain the iterative upper bounding formula

$$\begin{aligned}\mathrm{UB}_{\mathsf{E}\{f_v(\boldsymbol{y})\}} &= \mathrm{UB}_{\mathsf{E}\{g(\boldsymbol{y})\}}\mathrm{UB}_{\mathsf{E}\{h(\boldsymbol{y})\}} \\ \mathrm{UB}_{\mathsf{E}\{f_c(\boldsymbol{y})\}} &= \mathrm{UB}_{\mathsf{E}\{g(\boldsymbol{y})\}} + \mathrm{UB}_{\mathsf{E}\{h(\boldsymbol{y})\}} \\ &\quad - \mathrm{UB}_{\mathsf{E}\{g(\boldsymbol{y})\}}\mathrm{UB}_{\mathsf{E}\{h(\boldsymbol{y})\}}\end{aligned}$$

which is as if we were computing the expectation value by blindly assuming all messages were independent. Since the only inequality involved is from Rule 1, which induces no order loss, UB_i is an upper bound on $\mathsf{E}\{f_{\mathcal{T},t}(\boldsymbol{y})\}$ tight in asymptotic order. □

Appendix II
The Proof of *Theorem 3*

The proof of Theorem 3 is based on a tree-revealing argument, the description of which needs the following lemma and notation.

For any infinite-sized tree $\mathcal{T}$ corresponding to an iterative decoding function $f_{\mathcal{T}}(\boldsymbol{y})$, a *localized* subtree of $\mathcal{T}$ is defined as a subtree $\mathcal{U}$ sharing the same root as $\mathcal{T}$. The function $f_{\mathcal{U}}(\boldsymbol{y})$ is computed by appending any open sockets of $\mathcal{U}$ by dummy leaf units. Since for BECs, revealing more symbols always improves the performance, we have the following self-explanatory lemma.

Lemma 1: If $\mathcal{U}$ is a localized subtree of $\mathcal{T}$, then $f_{\mathcal{U}}(\boldsymbol{y}) \geq f_{\mathcal{T}}(\boldsymbol{y})$ for all $\boldsymbol{y} \in \{0,1\}^n$.

Proof of Theorem 3: We use $\mathcal{T}_t$ to denote the finite tree $\mathcal{T}$ of Algorithm 2 after t iterations of the REPEAT-UNTIL loop. We will sequentially construct a series of infinite-sized trees $\overline{\mathcal{T}}_t$ such that each $\mathcal{T}_t$ is a localized subtree of $\overline{\mathcal{T}}_t$ for all t. (Any infinite-sized trees are denoted with an overline.)

The first entry of the series of infinite-sized trees is defined as $\overline{\mathcal{T}}_0 \triangleq \overline{\mathcal{T}}_{v_i}$, where $\overline{\mathcal{T}}_{v_i}$ denotes the infinite-sized tree corresponding to the iterative decoder $f_i(\boldsymbol{y})$. By noting that $\mathcal{T}_0$, containing a $(\emptyset, v_i)$ leaf unit, is a localized subtree of $\overline{\mathcal{T}}_0$ and by *Lemma 1*, we have

$$\begin{aligned}f_{\mathcal{T},0}(\boldsymbol{y}) &= f_{\mathcal{T}_0}(\boldsymbol{y}) \\ &\geq f_{\overline{\mathcal{T}}_0}(\boldsymbol{y}) = f_{\overline{\mathcal{T}}_{v_i}}(\boldsymbol{y}) = f_i(\boldsymbol{y}), \forall \boldsymbol{y} \in \{0,1\}^n.\end{aligned}$$

We now consider the situation after the first iteration, which naturally consists of two cases corresponding to whether pivoting is performed or not during the first iteration. For the simpler case, in which $\mathsf{yca}(v, V')$ is a check node and no pivoting is performed, define $\overline{\mathcal{T}}_1 = \overline{\mathcal{T}}_0$ to be the second entry of the series of infinite-sized trees. By observing that $\mathcal{T}_1$ is obtained from $\mathcal{T}_0$ by adding another leaf unit that was previously in $\overline{\mathcal{T}}_0$ but not in $\mathcal{T}_0$, we have that $\mathcal{T}_1$ is a localized subtree of $\overline{\mathcal{T}}_1 = \overline{\mathcal{T}}_0$, and $\mathcal{T}_0$ is a localized subtree of $\mathcal{T}_1$. Therefore

$$\begin{aligned}f_{\mathcal{T},0}(\boldsymbol{y}) &= f_{\mathcal{T}_0}(\boldsymbol{y}) \\ &\geq f_{\mathcal{T},1}(\boldsymbol{y}) = f_{\mathcal{T}_1}(\boldsymbol{y}) \\ &\geq f_{\overline{\mathcal{T}}_1}(\boldsymbol{y}) = f_{\overline{\mathcal{T}}_0}(\boldsymbol{y}) = f_i(\boldsymbol{y}), \ \ \forall \boldsymbol{y} \in \{0,1\}^n. \qquad (10)\end{aligned}$$

The more interesting case is when pivoting is performed. Let $\mathcal{T}_{0.5}$ denote the tree after adding another leaf unit but before pivoting. By Lemma 1, we have $f_{\mathcal{T},0}(\boldsymbol{y}) = f_{\mathcal{T}_0}(\boldsymbol{y}) \geq f_{\mathcal{T}_{0.5}}(\boldsymbol{y})$. Since pivoting only transforms the original tree $\mathcal{T}_{0.5}$ into its equivalent form $\mathcal{T}_1$, we have $f_{\mathcal{T},1}(\boldsymbol{y}) = f_{\mathcal{T}_1}(\boldsymbol{y}) = f_{\mathcal{T}_{0.5}}(\boldsymbol{y})$ for all possible $\boldsymbol{y}$. Furthermore, the same pivoting operation can be applied to the infinite-sized $\overline{\mathcal{T}}_0$. When applying pivoting to $\overline{\mathcal{T}}_0$, we limit the replacement of active inputs in Line 10 to those variable nodes in $\overline{\mathcal{T}}_0$ that are also in $\mathcal{T}_1$. Use $\overline{\mathcal{T}}_1$ to denote the end result after pivoting. By the same reason that pivoting does not change the function, we will have $f_{\overline{\mathcal{T}}_1}(\boldsymbol{y}) = f_{\overline{\mathcal{T}}_0}(\boldsymbol{y})$. By noting that $\mathcal{T}_1$ is again a localized subtree of $\overline{\mathcal{T}}_1$, (10) holds for the case in which pivoting is performed as well. The above completes the proof of the induction from $t = 0$ to $t = 1$. The induction from tth iteration to $(t+1)$th iteration follows analogously. The proof of *Theorem 3* is complete. □

APPENDIX III
PROOF OF *Theorem 4*

Theorem 4 is the culmination of all analyses of Algorithm 2, the proof of which needs the following concepts/notation in addition to those established in Appendices I and II.

The Transcribed Tree $\mathcal{T}$:

Following the same notation as in Appendix II, let $\overline{\mathcal{T}}_{v_i}$ denote the infinite-sized tree corresponding to the iterative decoder $f_i(\boldsymbol{y})$. Denote a series of localized finite subtrees of $\overline{\mathcal{T}}_{v_i}$ by $\{\mathcal{T}_{v_i,0}, \ldots, \mathcal{T}_{v_i,s}, \ldots\}$ such that $\mathcal{T}_{v_i,0}$ contains only one $(\emptyset, v_i)$ leaf unit. $\mathcal{T}_{v_i,s+1}$ can be iteratively constructed from $\mathcal{T}_{v_i,s}$ by adding a single leaf unit to $\mathcal{T}_{v_i,s}$. Note: Unlike Algorithm 2, no pivoting is performed this time. $\mathcal{T}_{v_i,s}$ simply includes more and more leaf units of $\overline{\mathcal{T}}_{v_i}$ as s becomes large.

Again let $\mathcal{T}_t$ denote the tree resulting from Algorithm 2 after t iterations of the REPEAT-UNTIL loop in Line 2. We then have the following definition.

Definition 4: For any t and s, if the corresponding Boolean functions $f_{\mathcal{T}_t}(\boldsymbol{y})$ and $f_{\mathcal{T}_{v_i,s}}(\boldsymbol{y})$ are identical for trees $\mathcal{T}_t$ and $\mathcal{T}_{v_i,s}$, then we say the $\mathcal{T}_t$ tree, resulted from Algorithm 2, is *transcribed* from the $\mathcal{T}_{v_i,s}$ tree.

Namely, $\mathcal{T}_t$ is of a different structure than $\mathcal{T}_{v_i,s}$ but is equivalent to $\mathcal{T}_{v_i,s}$ from the output value perspective. We say that Algorithm 2 transcribes $\mathcal{T}_t$ from $\mathcal{T}_{v_i,s}$.

The Balanced Growing SF Module:

A balanced growing socket finding (SF) module is defined as follows.

Definition 5 (The Balanced Growing SF Module): With the assumption that Algorithm 2 is combined with the pruning rules in Section V-A, an SF module is balanced growing with respect to the series $\{\mathcal{T}_{v_i,0}, \ldots, \mathcal{T}_{v_i,s}, \ldots\}$ if there exists an increasing sequence of time instants $t_1 < t_2 < \cdots < t_s < \cdots$, such that $\mathcal{T}_{t_s}$ is a transcribed tree of $\mathcal{T}_{v_i,s}$ for all $s \in \mathbb{N}$.

Proposition 6: The balanced growing SF module exists for any sequence $\{\mathcal{T}_{v_i,0}, \ldots, \mathcal{T}_{v_i,s}, \ldots\}$.

Proof: This proposition will be proved by explicit construction.

When $s = 0$, simply set $t_0 = 0$. Since $\mathcal{T}_{t_0}$ contains only a $(\emptyset, v_i)$ leaf unit and is identical to $\mathcal{T}_{v_i,0}$, $\mathcal{T}_{t_0}$ is a transcribed tree of $\mathcal{T}_{v_i,0}$. *Proposition 6* holds for $s = 0$.

Suppose *Proposition 6* holds for the general sth entry, i.e., $\mathcal{T}_{t_s}$ is a transcribed tree of $\mathcal{T}_{v_i,s}$. Suppose further that $\mathcal{T}_{v_i,s+1}$ is obtained from $\mathcal{T}_{v_i,s}$ by adding a (c_j, v_k) leaf unit. Using the notation of Appendix II, let $\overline{\mathcal{T}}_{t_s}$ denote the infinite-sized tree after t_s iterations, of which $\mathcal{T}_{t_s}$ is a localized subtree.

We first note that the pivoting operation involves duplicating trees, which results in a one-to-many relationship between nodes in the original tree and nodes in the pivoted tree. Due to the fact that the pivoting operation may be performed many times before arriving at $\mathcal{T}_{t_s}$, there may be more than one leaf unit in $\overline{\mathcal{T}}_{t_s}$ but not in $\mathcal{T}_{t_s}$ that correspond to the (c_j, v_k) leaf unit that will be added to $\mathcal{T}_{v_i,s}$. All these leaf units are adjacent to $\mathcal{T}_{t_s}$ and are candidates for the next to-be-added leaf unit at the $(t_s + 1)$th iteration of Algorithm 2. We denote these leaf units by $(c_j, v_k)^{(1)}, (c_j, v_k)^{(2)}, \ldots, (c_j, v_k)^{(w)}$. If all these leaf units can be added to $\mathcal{T}_{t_s}$ successfully, we will obtain a $\mathcal{T}_{t_{s+1}}$ that is a transcribed tree of $\mathcal{T}_{v_i,s+1}$, which completes the proof of Proposition 6 . So the remaining question is how to add all of them successfully to $\mathcal{T}_{t_s}$.

For the (t_s+1)th iteration, we design the SF module in Line 3 to choose $(c_j, v_k)^{(1)}$. If the resulting $\mathsf{yca}(v, V')$ is a check node, no pivoting will be performed and $(c_j, v_k)^{(1)}$ is added to $\mathcal{T}_{t_s}$ directly without any unexpected side effects. $(w-1)$ leaf units $(c_j, v_k)^{(2)}, \ldots, (c_j, v_k)^{(w)}$ remain. For the case in which the resulting $\mathsf{yca}(v, V')$ is a variable node, pivoting will be performed and subtrees will be duplicated. Some leaf units in the previous remaining list $(c_j, v_k)^{(2)}, \ldots, (c_j, v_k)^{(w)}$ are duplicated as well, and the overall effect may be a longer list of remaining leaf units $(c_j, v_k)^{(2)}, \ldots, (c_j, v_k)^{(w)}, (c_j, v_k)^{(w+1)}, \ldots, (c_j, v_k)^{(w+w_d)}$. We use the last w_d nodes to denote the duplicated leaf units resulting from pivoting. This node duplication effect presents the major challenge of adding all $\{(c_j, v_k)^{(w)}\}$ into $\mathcal{T}_{t_s}$. The design goal is to have an SF module that does not result in an ever-increasing list of $\{(c_j, v_k)^{(w)}\}$.

Without loss of generality, we assume that $(c_j, v_k)^{(2)}, \ldots, (c_j, v_k)^{(1+w_d)}$ are the w_d leaf units in $\overline{\mathcal{T}}_{t_s}$ that will be duplicated by the left and the right subtrees in Fig. 6(b). After pivoting, the original leaf units are duplicated and have identical copies in the left and the right subtrees. We use the first w_d leaf units $(c_j, v_k)^{(2)}, \ldots, (c_j, v_k)^{(1+w_d)}$ to represent the corresponding leaf units in the right subtree, and use $(c_j, v_k)^{(w+1)}, \ldots, (c_j, v_k)^{(w+w_d)}$ for the corresponding leaf units in the left subtree. For the $(t_s + 2)$th iteration, the SF module locates $(c_j, v_k)^{(w+1)}$ as the next to-be-added leaf. Since the root of the left subtree, denoted by v, has an active observation y_k, the $\mathsf{yca}(v_k^{(w+1)}, V')$ resulted from adding $(c_j, v_k)^{(w+1)}$ is simply v. By invoking Rule 4 instead of Rule 2, no pivoting is necessary and we can directly include $(c_j, v_k)^{(w+1)}$ by replacing its active input y_k by 1. By adding the remaining $(c_j, v_k)^{(w+2)}, \ldots, (c_j, v_k)^{(w+w_d)}$ into $\mathcal{T}_{t_s+1}$ sequentially, and noting that only Rule 4 will be invoked, we are able to shortened the list of the remaining leaf units to

$(c_j, v_k)^{(2)}, \ldots, (c_j, v_k)^{(w)}$. In summary, the end result of the above construction is that regardless of whether $\mathsf{yca}(v_k^{(1)}, V')$ is a variable node or not, this SF module is able to successfully add leaf unit $(c_j, v_k)^{(1)}$ into $\mathcal{T}_{t_s}$ without introducing any new to-be-added leaf units. Repeating the same procedure for $v_k^{(2)}, \ldots, v_k^{(w)}$, we have constructed an SF module that generates $\mathcal{T}_{t_{s+1}}$, a transcribed tree of $\mathcal{T}_{v_i,s}$. The SF module under this construction is thus balanced growing. The proof is complete. □

Proof of Theorem 4: With the concept of balanced-growing SF modules, the proof of *Theorem 4* becomes straightforward. We construct a series of finite trees $\{\mathcal{T}_{v_i,0}, \ldots, \mathcal{T}_{v_i,s}, \ldots\}$ by the breadth-first search. Its corresponding balanced growing SF module is then an optimal SF module satisfying *Theorem 4*, the justification of which is as follows.

Let s_0 denote the first s such that the breadth-first search has visited all nodes of depth $2n$. Since the iterative decoder stops after at most n rounds of message exchanging, the finite tree $\mathcal{T}_{v_i,s_0}$ is equivalent to the infinite tree $\overline{\mathcal{T}}_{v_i}$ in terms of function outputs. The balanced growing SF module guarantees that after a finite time $t_0 \triangleq t_{s_0}$, $\mathcal{T}_{t_0}$ is a transcribed tree of $\mathcal{T}_{v_i,s_0}$. Therefore, $f_{\mathcal{T}_{t_0}}(\boldsymbol{y}) = f_{\mathcal{T}_{v_i,s_0}}(\boldsymbol{y}) = f_{\overline{\mathcal{T}}_{v_i}}(\boldsymbol{y}) = f_i(\boldsymbol{y})$ for all $\boldsymbol{y}$. The proof is complete. □

REFERENCES

[1] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Network of Plausible Inference*. San Mateo, CA: Morgan Kaufmann, 1988.

[2] R. Gallager, *Low-Density Parity-Check Codes*. Cambridge, MA: MIT Press, 1963, vol. 21, Research Monograph Series.

[3] D. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. Inf. Theory*, vol. 45, no. 2, pp. 399–431, Mar. 1999.

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

[5] J. Zhang and A. Orlitsky, "Finite length analysis of large- and irregular-left degree ldpc codes over erasure channels," in *Proc. IEEE Int. Symp. Information Theory*, Lausanne, Switzerland, Jun./Jul. 2002, p. 3.

[6] T. Richardson, "Error floors of LDPC codes," in *Proc. 41st Annu. Allerton Conf. Communications, Control, and Computing*, Monticello, IL, Oct. 2003.

[7] D. MacKay and M. Postol, "Weakness of Margulis and Ramanujan-Margulis low-density parity check codes," *Electron. Notes Theor. Comp. Sci.*, vol. 74, 2003.

[8] P. Vontobel and R. Koetter, "Graph-cover decoding and finite-length analysis of message-passing iterative decoding of LDPC codes," *IEEE Trans. Inf. Theory*, preprint–arXiv:cs.IT/0512078, to be published.

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

[10] J. Yedidia, E. Sudderth, and J.-P. Bouchaud, Projection Algebra Analysis of Error Correcting Codes Mitsubishi Electric Res. Labs., Cambridge, MA, Tech. Rep. TR2001–35, 2001.

[11] T. Richardson and R. Urbanke, "The capacity of low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 599–618, Feb. 2001.

[12] C.-C. Wang, S. R. Kulkarni, and H. V. Poor, "Density evolution for asymmetric memoryless channels," *IEEE Trans. Inf. Theory*, vol. 51, no. 12, pp. 4216–4236, Dec. 2005.

[13] A. Amraoui, R. Urbanke, A. Montanari, and T. Richardson, "Further results on finite-length scaling for iteratively decoded LDPC ensembles," in *Proc. IEEE Int. . Symp. Information Theory*, Chicago, IL, Jun./Jul. 2004, p. 103.

[14] C. Cole, S. Wilson, E. Hall, and T. Giallorenzi, "Analysis and design of moderate length regular LDPC codes with low error floors," in *Proc. 40th Conf. Information Sciences and Systems*, Princeton, NJ, Mar. 2006.

[15] R. Holzlöhner, A. Mahadevan, C. Menyuk, J. Morris, and J. Zweck, "Evaluation of the very low BER of FEC codes using dual adaptive importance sampling," *IEEE Commun. Lett.*, vol. 9, no. 2, pp. 163–165, Feb. 2005.

[16] M. Stepanov, V. Chernyak, M. Chertkov, and B. Vasić, "Diagnosis of weaknesses in modern error correction codes: A physics approach," *Phys. Rev. Lett.*, vol. 95, Nov. 2005.

[17] T. Richardson, M. Shokrollahi, and R. Urbanke, "Finite-length analysis of various low-density parity-check ensembles for the binary erasure channel," in *Proc. IEEE Int. Symp. Information Theory*, Lausanne, Switzerland, Jun./Jul. 2002, p. 1.

[18] C.-C. Wang, "Code annealing and the suppressing effect of the cyclically lifted LDPC code ensemble," in *Proc. 2006 IEEE Information Theory Workshop*, Chengdu, China, Oct. 2006, pp. 194–198.

[19] A. Orlitsky, K. Viswanathan, and J. Zhang, "Stopping set distribution of LDPC code ensembles," *IEEE Trans. Inf. Theory*, vol. 51, no. 3, pp. 929–953, Mar. 2005.

[20] S. Litsyn and V. Shevelev, "On ensembles of low-density parity-check codes: Asymptotic distance distributions," *IEEE Trans. Inf. Theory*, vol. 48, no. 4, pp. 887–908, Apr. 2002.

[21] S. Litsyn and V. Shevelev, "Distance distributions in ensembles of irregular low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 49, no. 12, pp. 3140–3159, Dec. 2003.

[22] M. Schwartz and A. Vardy, "On the stopping distance and the stopping redundancy of codes," *IEEE Trans. Inf. Theory*, vol. 52, no. 3, pp. 922–932, Mar. 2006.

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

[24] E. Berlekamp, R. McEliece, and H. van Tilborg, "On the inherent intractability of certain coding problems," *IEEE Trans. Inf. Theory*, vol. IT-24, no. 2, pp. 384–386, Mar. 1978.

[25] I. Dumer, D. Micciancio, and M. Sudan, "Hardness of approximating the minimum distance of a linear code," *IEEE Trans. Inf. Theory*, vol. 49, no. 1, pp. 22–37, Jan. 2003.

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

[27] K. Abdel-Ghaffar and J. Weber, "Stopping set enumerators of full-rank parity-check matrices of Hamming codes," in *Proc. IEEE Int. Symp. Information Theory*, Seattle, WA, Jul. 2006, pp. 1544–1548.

[28] M. Fossorier, "Quasi-cyclic low-density parity-check codes from circulant permutation matrices," *IEEE Trans. Inf. Theory*, vol. 50, no. 8, pp. 1788–1793, Aug. 2000.

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

[30] R. Smarandache, A. Pusane, P. Vontobel, and D. Costello, Jr., "Pseudo-codewords in LDPC convolutional codes," in *Proc. IEEE Int. Symp. Information Theory*, Seattle, WA, Jul. 2006, pp. 1364–1368.

[31] S. Ländner and O. Milenkovic, "Algorithmic and combinatorial analysis of trapping sets in structured LDPC codes," in *Proc. 2005 Int. Conf. Wireless Networks, Communications, and Mobile Computing*, Maui, HI, Jun. 2005, pp. 630–635.

[32] C. Berrou and S. Vaton, "Computing the minimum distance of linear codes by the error impulse method," in *Proc. IEEE Int. Symp. Information Theory*, Luasanne, Switzerland, Jun./Jul. 2002, p. 5.

[33] X. Hu, M. Fossorier, and E. Eleftheriou, "On the computation of the minimum distance of low-density parity-check codes," in *Proc. IEEE Int. Conf. Communications*, Paris, France, 2004, pp. 767–771.

[34] Y. Mao and A. Banihashemi, "A heuristic search for good low-density parity-check codes at short block lengths," in *Proc. IEEE Int'. Conf. Communications*, Helsinki, Finland, Jun. 2001, pp. 41–44.

[35] X. Hu, E. Eleftheriou, and D. Arnold, "Regular and irregular progressive edge-growth Tanner graphs," *IEEE Trans. Inf. Theory*, vol. 51, no. 1, pp. 386–398, Jan. 2005.

[36] T. Tian, C. Jones, J. Villasenor, and R. Wesel, "Selective avoidance of cycles in irregular LDPC code construction," *IEEE Trans. Commun.*, vol. 52, no. 8, pp. 1242–1247, Aug. 2004.

[37] A. Ramamoorthy and R. Wesel, "Construction of short block length irregular low-density parity-check codes," in *Proc. IEEE Int. Conf. Communications*, Jun. 2004, vol. 1, pp. 410–414.

[38] E. Sharon and S. Litsyn, "A method for constructing LDPC codes with low error floor," in *Proc. IEEE Int. Symp. Information Theory*, Seattle, WA, Jul. 2006, pp. 2569–2573.

[39] J. Tillich and G. Zémor, "On the minimum distance of structured LDPC codes with two variable nodes of degree 2 per parity-check equation," in *Proc. IEEE Int. Symp. Information Theory*, Seattle, WA, Jul. 2006, pp. 1549–1553.

[40] P. Vontobel and R. Koetter, "Lower bounds on the minimum pseudo-weight of linear codes," in *Proc. IEEE Int. Symp. Information Theory*, Chicago, IL, Jun./Jul. 2004, p. 70.

[41] X. Wu, X. You, and C. Zhao, "An efficient girth-locating algorithm for quasi-cyclic LDPC codes," in *Proc. IEEE Int. Symp. Information Theory*, Seattle, WA, Jul. 2006, pp. 817–820.
[42] T. Halford, K. Chugg, and A. Grant, "Which codes have 4-cycle-free Tanner graphs," in *Proc. IEEE Int. Symp. Information Theory*, Seattle, WA, Jul. 2006, pp. 871–875.
[43] T. Etzion, A. Trachtenberg, and A. Vardy, "Which codes have cycle-free Tanner graphs," *IEEE Trans. Inf. Theory*, vol. 45, no. 5, pp. 2173–2181, Sep. 1999.
[44] S. Hoory, "The size of bipartite graphs with a given girth," *J. Comb. Theory. Ser. B*, vol. 86, no. 2, pp. 215–220, 2002.
[45] M. Twitto, I. Sason, and S. Shamai (Shitz), "Tightened upper bounds on the ML decoding error probability of binary linear codes," *IEEE Trans. Inf. Theory*, to be published.
[46] S. Shamai (Shitz) and I. Sason, "Variations on the Gallager bounds, connections, and applications," *IEEE Trans. Inf. Theory*, vol. 48, no. 12, pp. 3029–3051, Dec. 2002.
[47] J. Rosenthal and P. Vontobel, "Constructions of LDPC codes using Ramanujan graphs and ideas from Margulis," in *Proc. 38th Annu. Allerton Conf. Communications, Control and Computing*, Monticello, IL, 2000, pp. 248–257.
[48] G. A. Margulis, "Explict constructions of graphs without short cycles," *Combinatorica*, vol. 2, no. 1, pp. 71–78, 1982.
[49] F. Kschischang, B. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 498–519, Feb. 2001.
[50] M. Sipser, *Introduction to the Theory of Computation*, 2nd ed. Boston, MA: Course Technology, 2005.
[51] J. Yedidia and J.-P. Bouchaud, Renormalization Group Approach to Error-Correcting Codes Mitsubishi Electric Research Laboratories, Cambridge, MA, Tech. Rep. TR2001–19, 2001.
[52] D. Divsalar, "Ensemble weight enumerators for protograph LDPC codes," in *Proc. IEEE Int. Symp. Information Theory*, Seattle, WA, Jul. 2006, pp. 1554–1558.
[53] [Online]. Available: http://lthcwww.epfl.ch/research/
[54] [Online]. Available: http://magma.maths.usyd.edu.au/

Chih-Chun Wang (M'06) received the B.E. degree in electrical engineering from National Taiwan University, Taipei, Taiwan in 1999 and the M.S. and Ph.D. degrees in electrical engineering from Princeton University, Princeton, NJ, in 2002 and 2005, respectively.

He worked at the Comtrend Corporation, Taipei, Taiwan, as a Design Engineer in 2000 and spent the summer of 2004 with Flarion Technologies, Bedminster, NJ. In 2005, he held a postdoctoral position in the Electrical Engineering Department, Princeton University. His current research interests are in the graph-theoretic and algorithmic analysis of iterative decoding and of network coding. His other research interests fall in the general areas of optimal control information theory, detection theory, coding theory, iterative decoding algorithms, and network coding.

Dr. Wang received the National Science Foundation Faculty Early Career Development (CAREER) Award in 2009.

Sanjeev R. Kulkarni (M'91–SM'96–F'04) received the B.S. degree in mathematics, the B.S. degree in electrical engineering, the M.S.degree in mathematics from Clarkson University, Potsdam, NY, in 1983, 1984, and 1985, respectively, the M.S. degree in electrical engineering from Stanford University, Stanford, CA, in 1985, and the Ph.D. degree in electrical engineering from the Massachusetts Institute of Technology (MIT), Cambridge, in 1991.

From 1985 to 1991, he was a Member of the Technical Staff at MIT Lincoln Laboratory, Lexington, MA. Since 1991, he has been with Princeton University, Princeton, NJ, where he is currently Professor of Electrical Engineering, and an affiliated faculty member in the Department of Operations Research and Financial Engineering and the Department of Philosophy. He spent January 1996 as a research fellow at the Australian National University, Canberra; 1998 with Susquehanna International Group, and Summer 2001 with Flarion Technologies. His research interests include statistical pattern recognition, nonparametric statistics, learning and adaptive systems, information theory, wireless networks, and image/video processing.

Prof. Kulkarni received an ARO Young Investigator Award in 1992, an NSF Young Investigator Award in 1994, and several teaching awards at Princeton University. He has served as an Associate Editor for the IEEE TRANSACTIONS ON INFORMATION THEORY.

H. Vincent Poor (S'72–M'77–SM'82–F'87) received the Ph.D. degree in electrical engineering and computer science from Princeton University, Princeton, NJ, in 1977.

From 1977 until 1990, he was on the faculty of the University of Illinois at Urbana-Champaign. Since 1990, he has been on the faculty at Princeton University, where he is the Dean of Engineering and Applied Science, and the Michael Henry Strater University Professor of Electrical Engineering. His research interests are in the areas of stochastic analysis, statistical signal processing and their applications in wireless networks, and related fields. Among his publications in these areas are the recent books *MIMO Wireless Communications* (Cambridge University Press, 2007), coauthored with Ezio Biglieri, *et al.*, and *Quickest Detection* (Cambridge University Press, 2009), coauthored with Olympia Hadjiliadis.

Dr. Poor is a member of the National Academy of Engineering, a Fellow of the American Academy of Arts and Sciences, and a former Guggenheim Fellow. He is also a Fellow of the Institute of Mathematical Statistics, the Optical Society of America, and other organizations. In 1990, he served as President of the IEEE Information Theory Society, and in 2004–2007 as the Editor-in-Chief of these TRANSACTIONS. He was the recipient of the 2005 IEEE Education Medal. Recent recognition of his work includes the 2007 IEEE Marconi Prize Paper Award, the 2007 Technical Achievement Award of the IEEE Signal Processing Society, and the 2008 Aaron D. Wyner Distinguished Service Award of the IEEE Information Theory Society.