Design of Low-Density Parity-Check (LDPC) Codes for Deep-Space Applications

K. Andrews,[1] S. Dolinar,[1] D. Divsalar,[1] and J. Thorpe[1]

This article summarizes a comparison of numerous low-density parity-check (LDPC) codes using a set of metrics relevant to deep-space communications. These metrics are the error-correcting capability of the code, the computational complexity of the decoder, the architectural complexities of the encoder and decoder, and the ease of generating a family of structurally related codes at different code rates and block lengths. Based on these metrics, we select a family of LDPC codes that is a good candidate for inclusion in a new Consultative Committee for Space Data Systems (CCSDS) telemetry channel coding standard for deep-space applications.

I. Introduction

The last several years of research have shown that low-density parity-check (LDPC) codes can provide error-correcting performance comparable to, or better than, that of turbo codes [1], with additional benefits. Compared to turbo decoders, LDPC decoders use a structure more suitable for high-speed hardware implementation, and also can require fewer computations per decoded bit. For these reasons, there is interest in selecting a family of LDPC codes for inclusion in a new Consultative Committee for Space Data Systems (CCSDS) telemetry channel coding standard for deep-space applications. This selection is difficult, however, because LDPC codes are an extremely large class of codes, and they vary widely in their characteristics.

In the most general sense, an LDPC code is any linear code with a sparse parity-check matrix. In the research literature, attention generally is restricted to binary codes with block lengths of at least a few thousand information bits, and we stay within these bounds as well. Decoding generally is assumed to be by belief propagation, using a well-established message-passing algorithm that operates on a graph described by the parity-check matrix [2].

There is a vast amount of material published about LDPC codes (cf. [3–5]), so we assume the reader is familiar with their concepts. Without further ado, we define the standard code parameters as follows. Let $\mathcal{C}$ be a binary linear code of rate $R = k/n$, with length n and dimension k. A parity-check matrix H for this code is of size $r \times n$, where $r = n - k$ if the rows are linearly independent, and it has e nonzero entries. This matrix also can be represented as a bipartite graph with variable nodes $\{V_1, \cdots, V_n\}$, constraint nodes $\{C_1, \cdots, C_r\}$, and e edges, where there is an edge connecting C_i to V_j if $H_{i,j} = 1$.

[1] Communications Architectures and Research Section.

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

The degree of a node is the number of edges to which it is attached. If every variable node has degree d_v and every constraint node has degree d_c, the code is called a regular (d_v, d_c) LDPC code, and it has rate $1 - d_v/d_c$. While "sparseness" is a loosely defined criterion for a matrix, most LDPC design algorithms can construct a family of codes of rate R for a variety of lengths n, and within this family, the average degrees $\bar{d}_v = e/n$ and $\bar{d}_c = e/r$ remain constant. Thus $e/(rn)$, the density of ones in H, approaches zero as n is made large.

In Section II, we define a set of metrics relevant to deep-space communications. In Section III, we summarize a variety of LDPC code construction methods. From each of these design methods, a couple representative codes were selected for analysis and computer simulation. Their performance with respect to each of the metrics is given in Table 1 for comparison. In Section IV, we briefly consider which code design methods are most appropriate for deep-space applications, name our current favorite code, and discuss one method to build a family of codes from it.

II. Metrics for Code Comparison

In this section, we define several desirable characteristics of a code and describe metrics for each. Most obviously, a code must provide good power efficiency, requiring a small value of E_b/N_0 to achieve a given error rate, where E_b is the energy per bit and N_0 is the (one-sided) noise power spectral density. Power efficiency is a particularly important metric for deep-space applications because the typical received signal power is extremely small.

The implementation cost of decoders for the Deep Space Network stations is determined by several factors, including the maximum supported data rate and complexity of the required decoder. For the purposes of code design and selection, we have separated the decoder complexity into two metrics: decoder computational complexity and decoder structure. The former simply measures the number of calculations required per decoded bit. The latter indicates the simplicity of organizing and scheduling those calculations.

While an LDPC encoder requires orders of magnitude less computation than the corresponding decoder, the encoder is in the deep-space environment and its implementation cost is also substantial. Because the amount of computation required of the encoder is small, we use a single metric to measure encoder structure.

A. Power Efficiency

The deep-space channel is a nearly perfect additive white Gaussian noise (AWGN) channel (not considering weather effects), so we compare codes based on their error-correcting capability as a function of E_b/N_0. Error correction is most commonly measured by bit-error rate (BER), though in many applications the codeword-error rate (WER) is the more appropriate measure.

As noted, LDPC codes usually are decoded by belief propagation, using a well-established iterative message-passing algorithm operating on the bipartite graph corresponding to the code's defining parity-check matrix. With a sufficient number of iterations, this algorithm typically provides performance near that of the maximum-likelihood decoder, at a much lower complexity. A wide variety of approximated belief propagation algorithms have been proposed, each of which introduces a modest performance loss in exchange for reduced computational complexity. It is anticipated that a standardized CCSDS LDPC code would be decoded by such an algorithm, so we base our code comparisons on results from the belief propagation decoding algorithm.

Figure 1 shows examples of typical WER and BER curves obtained through simulations. As seen in this figure, LDPC codes typically have a threshold E_b/N_0 at which the decoder begins to operate successfully. Both the BER and WER curves rapidly fall several orders of magnitude in the waterfall

Table 1. Comparison of several rate 1/2 codes, all approximately of size ($n = 8000$, $k = 4000$).

Code name	max d_v	max d_c	Protograph name	n	u	r	e	Expansion	Encode Method	Simple Encode	Simple Decode	Ie/k @ WER $= 10^{-4}$	Thresh (dB)	Floor WER	E_b/N_0 @ BER $= 10^{-4}$	E_b/N_0 @ BER $= 10^{-7}$	E_b/N_0 @ WER $= 10^{-3}$	E_b/N_0 @ WER $= 10^{-5}$
Non-protograph code constructions:																		
(3,6) Random	3	6	None						R/U	0	*	84	1.10	$< 10^{-5}$	1.55	1.78	1.59	1.76
(3,6) + PEG	3	6	None						R/U	0	*	88	1.10	$< 10^{-6}$	1.53	1.75	1.55	1.71
Code J	6	5	None						R/U	*	*	149	0.50	10^{-5}	0.98	1.27	0.98	1.18
ARA	5	5	ARA	4	1	3	14	Π ARA	ARA	***	***		0.52	10^{-4}	0.99	1.35	0.99	1.27
ARA-0.33	5	5	ARA-0.33	8	3	7	29	Π ARA	ARA	***	***		0.33	10^{-3}	0.87	1.47	0.87	1.33
Protograph and Progressive Edge Growth code constructions:																		
Summer1 + PEG	9	8	Summer1	8	1	5	29	PEG	R/U	*	***	186	0.28	$> 10^{-3}$	0.86	> 2	0.91	~ 2
D239 + PEG	9	7	D239	6	0	3	21	PEG	IRA	**	***	140	0.66	10^{-5}	1.12	1.40	1.14	1.30
Dpunc + PEG	6	5	Dpunc	6	1	4	19	PEG	IRPA	**	***	176	0.48	10^{-4}	0.95		1.02	
Ex2 + PEG	24	8	E*2	48	0	24	192	PEG	IRA	**	**	178	0.50	10^{-5}	1.00	1.23	1.00	1.23
ARAx2 + PEG	5	5	ARA*2	8	2	6	28	PEG	IRPA	***	***	186	0.52	10^{-5}	0.99	> 1.25	0.98	1.20
ARAA-0.65 + PEG	4	4	ARAA	7	3	5	17	PEG	R/U	**	***	300	0.65	$< 10^{-5}$	1.10	1.32	1.10	1.26
Protograph and circulant Progressive Edge Growth code constructions:																		
Summer1 + cPEG	9	8	Summer1	8	1	5	29	cPEG	R/U	*	****	33.3	0.28	10^{-1}	2.16	4.84	4.06	5.5
ARAx2 + cPEG	5	5	ARA*2	8	2	6	28	cPEG	IRPA	****	****	70.6	0.52	10^{-2}	1.20	2.9	2.11	3.3
ARAx6 + cPEG	5	5	ARA*2*3	24	6	18	84	cPEG	IRPA	****	****	187	0.52	$< 10^{-5}$	0.99	1.25	0.98	1.16
ARAx16 + cPEG	5	5	ARA*2*8	64	16	48	224	cPEG	IRPA	****	****	189	0.52	4×10^{-6}	0.98	1.34	0.98	1.16
Codes for comparison:																		
DVB-like	8	7	DVB	180	0	90	630	PEG	IRA	***	**	140	0.65	$> 10^{-5}$	1.12	1.40	1.14	1.45
Flarion floor			Proprietary										~ 0.65	$< 10^{-8}$	1.13	1.38		
Flarion thresh			Proprietary										~ 0.5	10^{-5}	0.96	1.18		

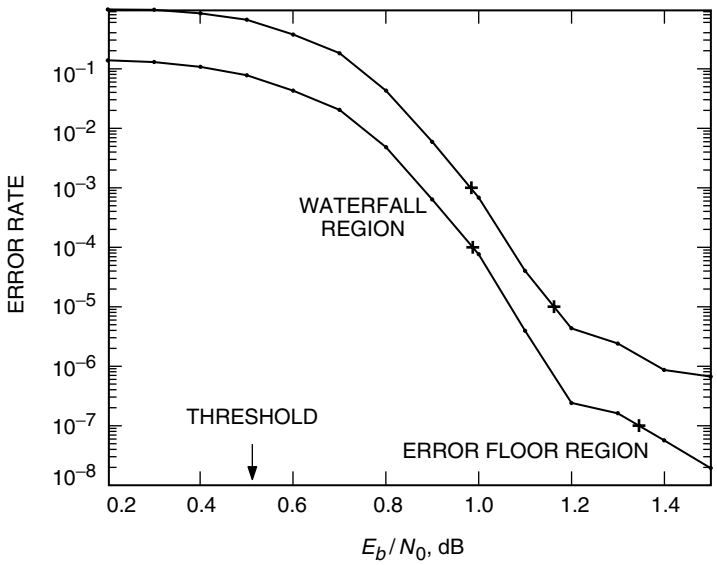

Fig. 1. WER (upper) and BER (lower) curves for the ARAx16+cPEG code.

region as E_b/N_0 is increased a fraction of a decibel above the threshold. Both curves then reach an error floor, below which they fall much more slowly with increasing signal-to-noise ratio (SNR). This means that the BER and WER curves must be characterized by several parameters. At the low SNR end, we have chosen to use the E_b/N_0 required to achieve BER $= 10^{-4}$, and that required for WER $= 10^{-3}$, for these are readily achievable by simulation. At higher SNRs, we use the E_b/N_0 required for BER $= 10^{-7}$ and for WER $= 10^{-5}$, for these values are more appropriate for the transmission of compressed telemetry and are still within the reach of simulation. Figure 1 includes markers showing these E_b/N_0 values, as tabulated for the ARAx16+cPEG code in Table 1.

Experimentally, we observe that most codes with n of at least a few thousand have a sharp transition between the waterfall and error floor portions of their BER and WER curves. For reasons that are not theoretically understood, there appears to be a trade-off between low thresholds and low error floors. This means that an LDPC code is best suited for use near the transition point in its performance curve, so its coordinates are of particular interest. For some codes, the transition is rounded enough that the height of this corner is not strictly quantitative, but the room for interpretation is generally much less than an order of magnitude in WER.

A code's threshold is a somewhat vague parameter when determined by simulation. In its place, a precise threshold is found by the code analysis tool of density evolution [6]. While this metric is based on an approximation that only predicts code performance, it is generally accurate to about a tenth of a decibel or better and is often computed early in a code design. These characteristics make it a valuable parameter for code comparison.

Density evolution is a method to determine the SNR at which belief propagation decoding of an LDPC code becomes effective. It does so by starting with the channel output probability densities, and propagating these densities through the iterative message-passing algorithm approximately as belief propagation would do. While intractably complex when done exactly according to belief propagation because of dependence between the random variables, it becomes practical if all messages arriving at each node in the graph are assumed to be independent. This approximation becomes progressively better as the sizes of the loops in the graph become large, which often occurs when an LDPC design algorithm is used to design progressively larger codes. Even with the assumption of message independence, tracking complete probability densities is complex and is often approximated. Of the several methods known [7–9], we have chosen to use Chung's Reciprocal Channel Approximation [9].

B. Decoder Computational Complexity

The computational complexity involved in decoding an LDPC code determines the resources needed for a decoder that runs at a given speed. While computational complexity is best measured in different ways for different decoder implementation technologies, such as software, field programmable gate arrays (FPGAs), or application-specific integrated circuits (ASICs), we aim to choose a metric that is independent of implementation. The belief propagation algorithm consists of the computation at each graph node of output messages as functions of the input messages. This computation is repeated for each edge in the graph, for as many iterations as needed. This observation motivates our definition of computational complexity:

$$\text{Complexity} = Ie/k$$

in units of message updates per decoded bit, where I is the average number of iterations performed, e is the number of edges in the graph, and k is the dimension of the code.

Certainly the difficulty of each message update computation depends upon the number of input variables, i.e., the number of other edges connected to the graph node. This effect is not included in the measure of computational complexity because it proves to be small in many decoder implementations. For example, a software decoder that updates outgoing message v_j at a variable node of degree d_v by addition of log-likelihood ratios (LLRs) must evaluate

$$v_j = \sum_{i \neq j} u_i, \qquad i, j \in \{1, \cdots, d_v\}$$

where u_i are the incoming messages to the node. If the software actually computes these terms by

$$s = \sum_{i=1}^{d_v} u_i, \qquad v_j = s - u_j, j \in \{1, \cdots, d_v\}$$

then the number of additions and subtractions required is $2d_v - 1$, which is nearly proportional to d_v. A similar argument holds at the check nodes, making the total computation per iteration nearly proportional to e.

A hardware decoder may perform all computations at a node, or set of nodes, simultaneously. Its speed may be limited by the memory accesses required to read and write the edge message values. This complexity is exactly proportional to e.

C. Decoder Structure

In addition to the number of computations required by the decoder, the scheduling of those computations is also an important code selection criterion. Particularly desirable characteristics become intimately dependent on details of the implementation, but some salient features are likely to be important. A code with large node degrees can be expected to be more difficult to decode than one with small degrees, due perhaps to the depth of logic required for the computations or to memory access scheduling. A code with well-organized edges may permit simpler memory structures, and address generation may be possible using counters or simple combinatorial circuits rather than large lookup tables. We have chosen to use a semi-quantitative ranking that ranges from zero to four stars, one for each of the following criteria:

(1) The maximum node degrees satisfy $d_v < 10$, $d_c < 10$, reducing logic depth.

(2) The code possesses a protograph structure [10], simplifying parallel hardware decoding.

(3) The protograph has fewer than 300 edges, reducing descriptional complexity.

(4) The code uses circulant permutations, simplifying address generation.

D. Encoder Structure

For LDPC codes, as for all practical codes, encoder complexity is small compared to decoder complexity. For data return from deep space, it remains a concern because the encoder must tolerate the hostile space environment with limited resources available, while the decoder is on Earth.

While an LDPC code is technically nothing more than a parity-check matrix, a standardized code must primarily consist of an encoder definition. Before 2001, it had been accepted that, except in special cases, a sparse H necessitated a dense generator matrix G, computed by a process akin to matrix inversion. Encoding message vector m would require computing mG, of computational complexity quadratic in k. Richardson and Urbanke [11] showed that, in general, encoding complexity is nearly linear in k, being quadratic only in a small term g that grows much more slowly than k. For many LDPC code families, $g = 0$, so complexity is truly linear. The encoder's computational complexity is thus reduced to an issue of a proportionality constant that is highly dependent on implementation technology.

The difficulty of building an encoder is determined primarily by its descriptional complexity, i.e., a measure of the amount of memory required to store the code description. In general, Richardson and Urbanke's encoding method depends on sparse matrices that have no internal structure, and the locations of all the nonzero bits must be enumerated.

For the LDPC codes of greatest interest, particularly well structured encoders are possible. Irregular repeat and accumulate (IRA) codes [12,13] are one class that has attracted attention, and, as described by their name, encoding consists of sparse matrix multiplication followed by cumulative modulo-2 summation. Quasi-cyclic codes [14] can use encoders built from shift registers. Some LDPC code families are built from circulants [15] to lower the descriptional complexity. Divsalar et al. have also proposed accumulate-repeat-accumulate (ARA) codes [16] and accumulate-repeat-accumulate-accumulate (ARAA) codes [17] that permit particularly simple encoders of a rather different style.

The discussion above identifies some desirable attributes of an LDPC code from the perspective of encoder complexity, but fails to give a quantitative metric. In its place, we are left with a semi-quantitative score ranging from zero to four stars, based upon the following criteria:

(1) $g < 10$, so encoding complexity is nearly linear in the code block length.

(2) The code uses circulant permutations, simplifying address generation.

(3) An encoder can be built from simple components such as accumulators.

(4) An encoder requires $\leq$2 accumulators and no sparse matrix multiplication.

III. Code Designs

In this section, a variety of LDPC construction techniques are outlined. We start with the "classic" regular LDPC codes first introduced by Gallager. We identify a weakness in the construction, propose a new construction method to address that weakness, and continue in this fashion. With each successive construction method, some representative code designs are selected and characterized. In general, each code design method is superior to the preceding ones, although there is a tendency for uncontrolled

characteristics to get worse, resulting in trade-offs among the attributes of a code. For example, as a code design method is modified to lower the threshold, the error floor tends to rise.

Each of the characterized codes is listed in Table 1, showing the performance measured by the metrics developed in the previous section. To facilitate comparison, every code has rate 1/2 and is approximately of size $(n = 8000, k = 4000)$, encoding about 4000 information bits into about 8000 code symbols.

A. Regular LDPC Codes

The first LDPC codes introduced by Gallager [18] were regular (3,6) codes with rate 1/2. Within the node degree constraints, the parity-check matrix is selected randomly. Performance results for a typical code constructed in this way are shown in the first line of Table 1.

This code scores rather poorly in most categories, but it serves as a well-studied reference point. In particular, the code's threshold of 1.10 dB, shared by all regular (3,6) codes, sets an upper bound on "interesting" thresholds for LDPC codes. The lower bound is set by the capacity of the binary input AWGN channel at 0.187 dB.

This code's graph contains many small loops, including loops of length 4. The belief propagation decoding algorithm is known to be optimal [19] for loop-free graphs and to be suboptimal [20] in the presence of more than one loop, because incoming messages at a node are not independent. It is generally accepted that belief propagation decoding performs poorly in the presence of many small loops. It also is known that while $(d_v = 3, d_c = 6)$ is the optimal choice among regular rate 1/2 codes, appropriate irregular degree distributions perform better [7].

B. Progressive Edge Growth

The weaknesses in the Gallager code caused by the presence of many small loops can be addressed by designing H rather than choosing it randomly. Some researchers use algebraic constructions to control the girth of the code's graph [14,15]; others use computer searches.

Progressive edge growth (PEG) [21] is one computer search algorithm for designing H with given degree constraints that attempts to prevent small loops in the graph. The PEG algorithm is initialized with a trivial graph consisting of n variable nodes, r constraint nodes, and no edges. Each node is given the appropriate number of "sockets" where edges will be inserted. PEG then proceeds by placing edges one at a time, each one connecting the variable node and check node sockets that maximize the minimum loop length of the resulting graph (or random selections are made in the case of ties). This greedy algorithm has weaknesses (early choices made by the algorithm can force poor results for reasons not apparent when the choice is made, and minimum loop length is a poor proxy for overall code performance), and many variations on the theme have been used. A regular (3,6) code built using PEG is shown in the second line of Table 1.

C. Multi-Edge-Type Codes

The use of irregular node degree distributions can lower a code's threshold, and good degree distributions are derived in [1]. Simulations show that codes constructed by randomly connecting nodes with the appropriate degree distribution often have high error floors and perform rather badly. Design methods to avoid short loops improve these codes, but not markedly. This problem is solved by dividing the nodes into categories, and applying constraints on connections between nodes in the various categories.

Multi-edge-type codes [22,23] developed by Richardson led the way in this research. To construct such a code, one begins with a collection of variable nodes and check nodes, each with unconnected edge sockets to indicate its target node degree. Each empty socket is then assigned a "color" to designate its edge type; there must be the same number of variable node sockets and constraint node sockets of each

color. Finally, edges are drawn between the sockets of the variable nodes and check nodes, assigned to connect matching colors. These edges could be assigned randomly, or better performance is achieved by using the PEG algorithm described above, under the additional color constraint.

The structure of one multi-edge-type code, named Code J, is shown in Fig. 2. This code has 9615 variable nodes (3750 of degree two, 3192 of degree three, 1423 of degree six, and 1250 of degree one), and the degree-six variable nodes are punctured, giving a code of length $n = 8192$. The 4788 degree-5 and 731 degree-4 check nodes constrain the code to dimension $k = 4096$. The graph edges are divided into five types, and their connections are described by five permutations designed using the PEG algorithm. Performance of this code is given in the third line of Table 1. The use of this structure reduces the code's threshold to 0.50 dB, about 0.6 dB better than the Gallager codes. This comes at a cost of introducing an error floor around WER $= 10^{-5}$ that was absent in the Gallager constructions.

As illustrated by this code, the use of punctured variable nodes can yield significant improvement in decoding threshold and provides additional freedom in code design. There is typically a cost in decoder complexity, however, because punctured codes generally use more edges per information bit than unpunctured codes.

D. Codes Built From Accumulators and Other Simple Elements

A rather different code construction method expands upon ideas from serial concatenated convolutional codes (SCCCs) [24] and irregular repeat accumulate (IRA) codes. Encoders for good codes can be built from accumulators (the $1/(1 + D)$ convolutional code), permuters, symbol repeaters, and periodic puncturers. The encoder for an accumulate-repeat-accumulate (ARA) code [16] is shown in Fig. 3(a). This code has a threshold of 0.52 dB, and there are variants with thresholds both higher and lower.

Performances of a couple ARA codes are shown in the fourth and fifth lines of Table 1. These codes use a highly structured "block interleaver" for convenience (denoted as the "Π ARA" expansion in the table); it is probable that a well-chosen permutation would lower the observed error floor.

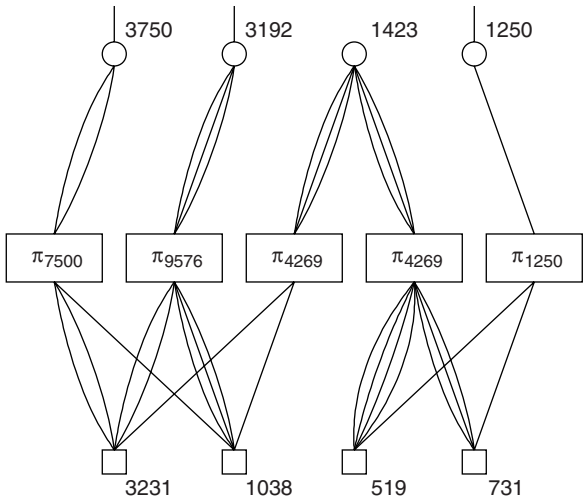

Fig. 2. Multi-edge-type structure of Code J, with threshold 0.5 dB.

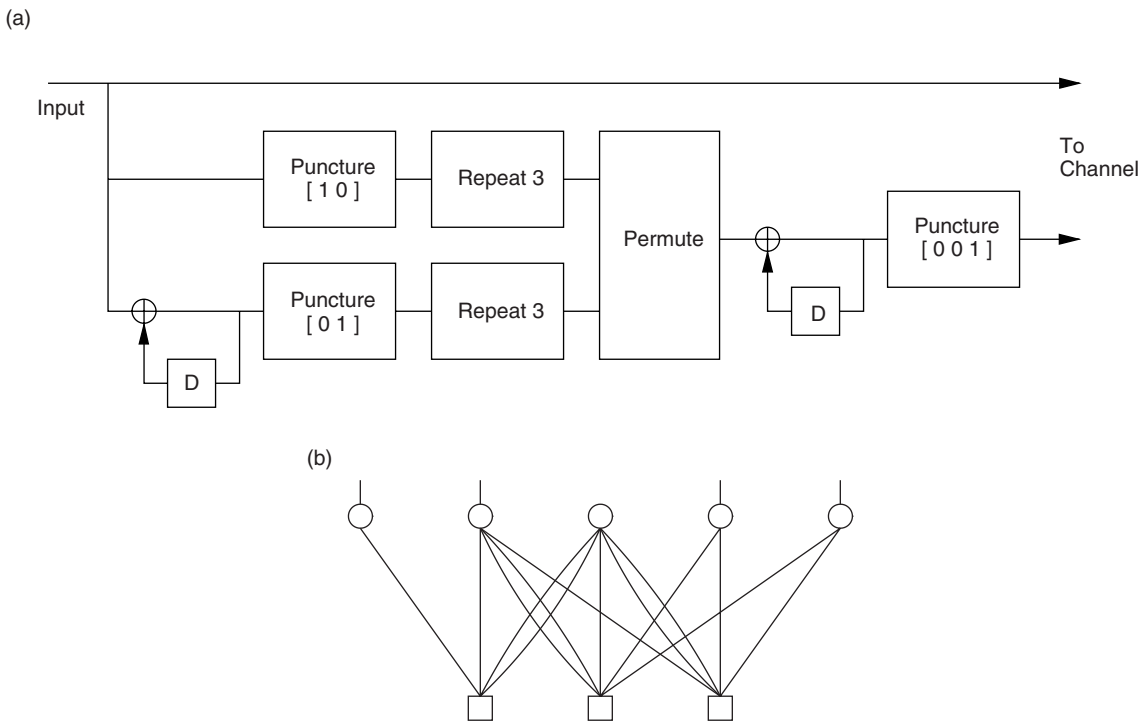

Fig. 3. ARA (a) encoder and (b) protograph.

E. Protograph Codes

Protograph codes [10] combine the ideas of multi-edge-type codes with the particular goal of low structural complexity for hardware decoders. This code construction begins by designing a small bipartite graph, with n_p variable nodes, r_p constraint nodes, and e_p edges, with the code rate, node degree distributions, and threshold desired for the final code. Various methods of protograph design are available; we have had successes with manual selection guided by experience and heuristics, and with simulated annealing that seeks to minimize threshold within constraints on protograph size. Each node then is replicated T times, and each edge is replaced by a "bundle" of edges, connecting the corresponding copies of variable nodes and constraint nodes. Finally, each bundle is severed and reconnected in some permuted way, thus interconnecting the copies of the original protograph. These permutations can again be selected by a PEG algorithm. The result is a code of size $(n = Tn_p, k = n - r = Tn_p - Tr_p)$.

Hardware decoders for protograph LDPC codes can be built to process a complete copy of the protograph in each clock cycle [25], thus processing e_p edges per cycle, and requiring T cycles per half-iteration. For this reason, the size of the protograph should be tailored to match the size of the available hardware. The protograph construction procedure can be applied recursively, thus building a full graph in two or more stages. This method can provide a single code that can be decoded with several different levels of parallelism.

Many protograph codes have been designed and tested; a few are shown in Fig. 4, with performance results included in Table 1. The table shows that these codes range from D239+PEG with a threshold of 0.66 dB and an error floor at WER $= 10^{-5}$, to Summer1+PEG with a remarkably low threshold of 0.28 dB and an error floor above WER $= 10^{-3}$. This trade-off between threshold and error floor is a recurring theme in LDPC code design, although whether this trade-off is inevitable remains an open question in the research community.

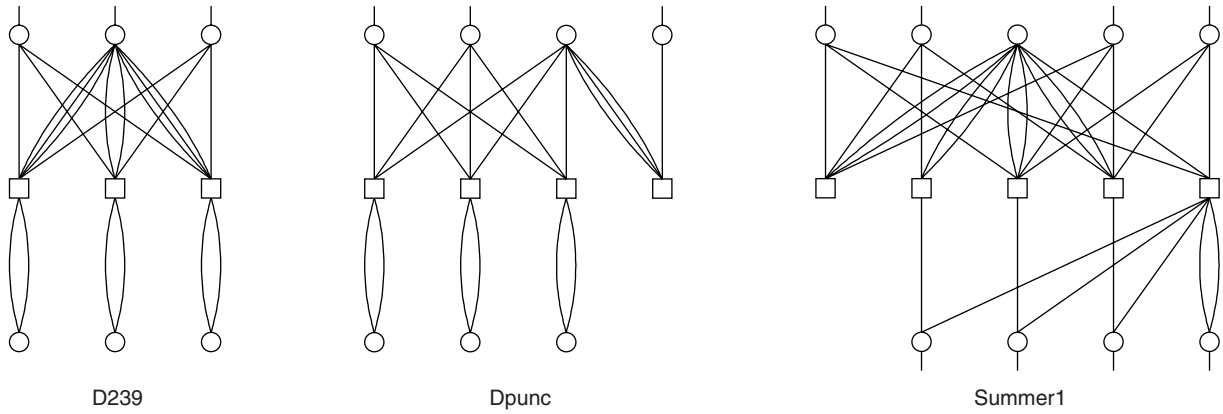

Fig. 4. A few good protographs.

With the addition of one graph edge per accumulator to close the chains into "tail-biting" loops, the ARA codes and variants described in the previous section become protograph codes. The ARA code with the encoder shown in Fig. 3(a) has the protograph shown in Fig. 3(b). Rather than using the block interleaver previously described, other interleavers can be (implicitly) defined by using a PEG expansion of the protograph.

F. Circulant PEG

The protograph codes described above have some structure, but their descriptional complexity remains high because the permutations used in each of the bundles of edges must be stored in memory at every encoder and decoder, and in a standardization document. Instead of constructing random-looking permutations with a PEG algorithm, one can use algorithmically described permutations. A simple algorithm can greatly reduce the descriptional complexity at a cost in design freedom. If particular simple permutations do not generate short loops or other structural defects, the resulting code can be very good; otherwise the defects are replicated many times and the results are disastrous.

One particularly simple set of permutations is the set of cyclic shifts, often called circulants [21,26]. A protograph code can be expanded using circulants rather than permutations generated by PEG, thus reducing the descriptional complexity of the code from approximately $e_p \log_2 (T!)$ bits to $e_p \log_2 T$ bits. The design of circulant protograph codes begins with the selection of a protograph, as described above. Then circulants are chosen for the edge bundles one at a time, once again by a variant of a PEG algorithm.

A small protograph has few edges, and circulant expansion is sufficiently restrictive that it is often impossible to avoid low-weight codewords. One solution is to expand the protograph to the full code in two or three stages, using the circulant PEG algorithm recursively. When the protograph expansion factor T is small, such as in the first stage of a multi-stage protograph expansion, we have sometimes found it advantageous to select the circulants by hand.

Some resulting codes shown in Table 1 are Summer1+cPEG, ARAx2+cPEG, and ARAx6+cPEG. The thresholds for these codes match those for the non-circulant versions of course, and an additional star is garnered for simplicity of both the encoder and decoder.

The code Summer1+cPEG was expanded in a single stage using circulants of size $T = 1024$. Independent of the circulants chosen, this expansion inevitably yields a code with T codewords of weight 9, and so performance is poor. The code ARAx2+cPEG was expanded in two stages, first by a factor of two to separate the parallel edges in the protograph, and then by a factor of 1024 to the full code size. The construction inevitably yields a code with T codewords of weight 12, also giving poor performance. The code ARAx6+cPEG was expanded in three stages, first by $T = 2$ as before, then by $T = 3$, and finally by

$T = 341$. Its minimum distance is large (the smallest weight codeword we have found has weight 31), and the code's performance is similar to that of the code ARAx2+PEG. The code ARAx16+cPEG also was expanded in three stages, by $T = 2$, $T = 8$, and $T = 128$, yielding a code of size $(n = 8192, k = 4096)$. Because the number of information bits in this code is a power of two, data handling by a microprocessor may be simplified.

G. Encoder Implementations

All the codes described can be systematically encoded using Richardson and Urbanke's near-linear-time encoding algorithm [11], and in some cases, the algorithm becomes particularly simple, as mentioned in Section II.D. Their algorithm puts as large a portion of H as possible into lower triangular form through row and column permutations. In general, g rows cannot be triangularized, and the algorithm requires inversion of a $g \times g$ matrix, and multiplications by several matrices with a dimension g. For special cases where $g = 0$, their encoding algorithm reduces to a sparse matrix multiplication, followed by back-substitution (due to the triangular portion of H).

Of the codes discussed, only the ARA and ARAA codes have $g = 0$, but many have $g = 1$. This occurs when the code contains at least r variable nodes of degree ≤ 2, with the degree 2 nodes connected in a single loop. This is a rather common occurrence because the ideal degree distributions for rate 1/2 codes include nearly this many degree 2 variable nodes [1], and the PEG algorithm tends to place them in a single loop, either when used with individual edges or with circulants. By deleting a single edge to break this loop, g becomes zero. Moreover, the lower-triangular portion of H can be arranged to contain ones on the primary and first sub-diagonals and to be zero elsewhere. This means the back-substitution stage of Richardson and Urbanke's encoding algorithm can be implemented with the $1/(1 + D)$ feedback shift register, commonly called an accumulator. In this way, Richardson and Urbanke's general encoding method reduces to irregular repeat and accumulate (IRA) encoding, as developed by Jin [12], Ryan [13], and others.

Some of the codes above can be modified to become IRA codes by deletion of a single edge in the graph. This has a negligible impact on code performance, and the irregularity due to this deleted edge can be accommodated readily in the decoder design. Thus, this is a natural improvement to codes D239, Dpunc, and Ex2, and each of these has been granted an additional star for low encoder complexity for possessing this characteristic.

In Table 1, an encoding method is listed for each code. When no simpler method is available, the general Richardson–Urbanke method is listed as "R/U." Some codes permit an irregular repeat and partial accumulate (IRPA) encoder, a mild generalization of the irregular repeat and accumulate (IRA) encoding method mentioned above. The accumulate-repeat-accumulate (ARA) codes are named for one of their possible encoders; IRPA encoders can also be used for these codes. The protograph-plus-circulant construction creates codes with "quasi-cyclic" parity check matrices. In some cases, these codes have quasi-cyclic generator matrices [27]. Even though these matrices are dense in general, they also permit low-complexity encoders based on hardware shift registers.

IV. Code Selection

For comparison, a few codes designed by other organizations are listed in the last three lines of Table 1. The Digital Video Broadcast working group has chosen a structured IRA code that is very similar to a protograph-plus-circulant construction. This code uses a very large protograph and has an unimpressive threshold, but scores well in other categories. Performance of a reduced-size version of this code is shown in Table 1. Flarion Technologies, Inc., is a leader in LDPC code design; performance data from two of their codes are shown at the bottom of the table. The first is designed to have a particularly low error floor; the second is designed for a good threshold. While the code constructions are proprietary, there are good reasons to believe they are extremely similar to protograph codes expanded with circulants.

Reviewing the table, we see that the multi-edge-type and protograph codes have thresholds 0.5 to 0.8 dB better than the regular (3,6) codes. Within the class of protograph codes, the circulant constructions generally have the lowest encoder and decoder complexities. The location of the error floor varies widely among codes built with the protograph-plus-circulant construction method, and the best ones found to date belong to the ARAx6+cPEG and ARAx16+cPEG codes. In exchange for the improvements in encoder and decoder descriptional complexity and in threshold, there is a modest cost in decoder computational complexity.

Selection of the best code for an application depends upon the relative value of the different metrics to that application. The determination of these relative values is beyond the scope of the code design problem. Based upon past experience with error-correcting codes for the deep-space link, we currently believe that an ARA+cPEG code is an attractive choice.

Code characteristics change with rate and block length, so a family of codes should be standardized to support a variety of applications. It is desirable that these LDPC codes have similar structures, potentially to simplify the implementation of a single encoder or decoder that must support several members of the family.

One way to define a family of codes with different rates is to use a set of nested protographs. Starting with the ARA protograph shown in Fig. 3(b) and adding degree-3 variable nodes as shown in Fig. 5, we have a family of protographs of rate $(x+1)/(x+2)$ for $x \geq 0$. Each protograph in this family is within 0.3 dB of the binary-input channel capacity at its rate [16], and each contains the protographs of lower rates as subgraphs. Codes of different block lengths can be formed by using circulants of different sizes.

V. Conclusions

Design and selection of an LDPC code for standardization is difficult because there are a large number of desirable characteristics that must be traded against one another, and the search space is tremendous.

Of the codes investigated to date, the ARA protograph expanded with circulants is a leading contender. While some code design issues remain, most notably in selecting the number of stages in which to expand the protograph and the sizes of those circulant expansions, many strengths are clear. Decoder complexity is relatively low, requiring an average of 27 iterations at WER $= 10^{-4}$ for 189 edge message updates per

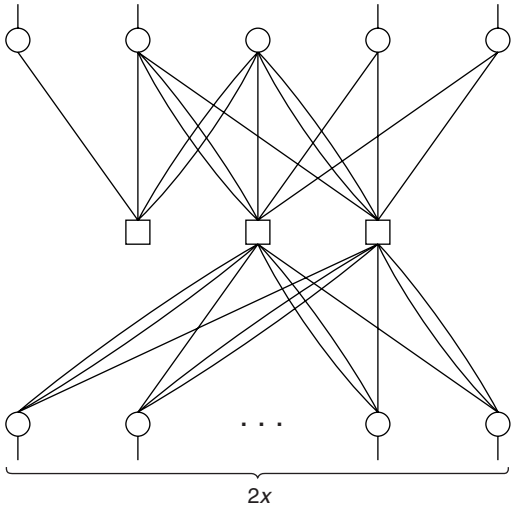

Fig. 5. A family of ARA protographs, of rate $(x+1)/(x+2)$ for $x \geq 0$.

decoded bit. Decoder implementation is particularly simple since these codes use a small protograph, with maximum variable and check node degrees of only five. The circulant expansion technique provides low descriptional complexity at both the encoder and decoder. Encoding can be accomplished in any of three ways: using the accumulate-repeat-accumulate block diagram shown in Fig. 3, by sparse matrix multiplication and accumulation, or with shift registers to implement a quasi-cyclic generator. The code threshold is at 0.516 dB, only 0.329 dB from channel capacity. The ARA protograph is also one member of a family of protographs describing codes of rate $(x + 1)/(x + 2)$, all of which perform within a few tenths of a decibel of channel capacity. The selection of a few of these ARA protographs, combined with circulants of the appropriate sizes, can generate an attractive family of codes with a variety of code rates and block lengths.

References

[1] T. Richardson, M. A. Shokrollahi, and R. Urbanke, "Design of Capacity-Approaching Irregular Low-Density Parity-Check Codes," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 619–637, February 2001.

[2] F. Kschischang, B. Frey, and H.-A. Loeliger, "Factor Graphs and the Sum-Product Algorithm," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 498–519, February 2001.

[3] *IEEE Transactions on Information Theory*, Special Issue: Codes on Graphs and Iterative Algorithms, vol. 47, February 2001.

[4] *IEEE Communications Magazine*, Feature Topic: Capacity Approaching Codes, Iterative Decoding Algorithms, and their Applications, vol. 41, August 2003.

[5] *Proceedings of the 2004 IEEE International Symposium on Information Theory*, Chicago, Illinois, June 2004.

[6] T. Richardson and R. Urbanke, "The Capacity of Low-Density Parity-Check Codes Under Message-Passing Decoding," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 599–618, February 2001.

[7] S.-Y. Chung, T. Richardson, and R. Urbanke, "Analysis of Sum-Product Decoding of Low-Density Parity-Check codes Using a Gaussian Approximation," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 657–670, February 2001.

[8] S. ten Brink, "Convergence of Iterative Decoding," *Electronics Letters*, vol. 35, no. 13, pp. 806–808, May 1999.

[9] S. Y. Chung, *On the Construction of Some Capacity-Approaching Coding Schemes*, Ph.D. Thesis, Massachusetts Institute of Technology, September 2000.

[10] J. Thorpe, "Low-Density Parity-Check (LDPC) Codes Constructed from Protographs," *The Interplanetary Network Progress Report 42-154, April–June 2003*, Jet Propulsion Laboratory, Pasadena, California, pp. 1–7, August 15, 2003. http://ipnpr.jpl.nasa.gov/tmo/progress_report/42-154/154C.pdf

[11] T. Richardson and R. Urbanke, "Efficient Encoding of Low-Density Parity-Check Codes," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 638–656, February 2001.

[12] H. Jin, A. Khandekar, and R. McEliece, "Irregular Repeat-Accumulate Codes," *Proceedings of 2nd Intl. Symposium on Turbo Codes and Related Topics*, Brest, France, pp. 1–8, September 2000.

[13] M. Yang, Y. Li, and W. Ryan, "Design of Efficiently Encodable Moderate-Length High-Rate Irregular LDPC Codes," *Proceedings of the 41st Allerton Conference*, October 2002.

[14] Y. Kou, S. Lin, and M. Fossorier, "Low-Density Parity-Check Codes Based on Finite Geometries: A Rediscovery and New Results," *IEEE Transactions on Information Theory*, vol. 47, no. 7, pp. 2711–2736, November 2001.

[15] A. Sridharan, D. Costello, and R. Tanner, "A Construction for Low Density Parity Check Convolutional Codes Based on Quasi-Cyclic Block Codes," *IEEE International Symposium on Information Theory*, Lausanne, Switzerland, p. 481, June 2002.

[16] A. Abbasfar, D. Divsalar, and K. Yao, "Accumulate Repeat Accumulate Codes," *IEEE International Symposium on Information Theory*, Chicago, Illinois, June 2004.

[17] D. Divsalar, S. Dolinar, and J. Thorpe, "Accumulate-Repeat-Accumulate-Accumulate-Codes," *IEEE Vehicular Technology Conference*, Los Angeles, California, September 2004.

[18] R. G. Gallager, "Low Density Parity Check Codes," *IRE Transactions on Information Theory*, vol. IT-8, pp. 21–28, 1962.

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

[20] Y. Weiss and W. Freeman, "Correctness of Belief Propagation in Gaussian Graphical Models of Arbitrary Topology," *Neural Computation*, vol. 13, no. 10, pp. 2173–2200, 2001.

[21] X.-Y. Hu, E. Eleftheriou, and D.-M. Arnold, "Progressive Edge-Growth Tanner Graphs," *IEEE Globecom Conference 2001*, vol. 2, pp. 995–1001, November 2001.

[22] T. Richardson, "Multi-Edge Type LDPC Codes," presented at the Workshop Honoring Prof. McEliece on his 60th Birthday, California Institute of Technology, Pasadena, California, May 24–25, 2002.

[23] T. Richardson, "Multi-Edge Type LDPC Codes," *IEEE International Symposium on Information Theory*, Recent Results Session, Lausanne, Switzerland, 2002.

[24] D. Divsalar, S. Dolinar, and F. Pollara, "Improving Turbo-Like Codes using Iterative Decoder Analysis," *IEEE International Symposium on Information Theory*, Washington, D.C., p. 100, June 2001.

[25] J. Lee, B. Lee, J. Thorpe, S. Dolinar, J. Jamkins, and K. Andrews, "A Scaleable Architecture of a Structured LDPC Decoder," IEEE International Symposium on Information Theory, Chicago, Illinois, June 2004.

[26] Y. Kou, H. Tang, S. Lin, and K. Abdel-Ghaffar, "On Circulant Low Density Parity Check Codes," *IEEE International Symposium on Information Theory*, p. 200, June 2002.

[27] S. Lin, "Quasi-Cyclic LDPC Codes," CCSDS Working Group White Paper, October 2003.