On Rate-adaptability of Nonbinary LDPC Codes

Demijan Kline*, Jeongseok Ha† and Steven W. McLaughlin*
†School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, USA
Email: {demi, swm}@ece.gatech.edu
‡School of Electrical and Computer Engineering, Information and Communications University
119, Munjiro, Daejeon, 305-732, Korea
Email: jsha@icu.ac.kr

Abstract—This paper examines rate-adaptability of nonbinary LDPC codes at short block lengths over the binary input AWGN channel. Starting from a mother code, higher rate codes are obtained via puncturing, and lower rates are obtained by shortening. We propose algorithms both for puncturing and shortening and design a highly flexible, one encoder/decoder system that supports code rates ranging from 0.1 to 0.9. Our results show that properly designed rate-adaptable nonbinary LDPC codes can significantly outperform their binary counterparts throughout the entire range of code rates.

I. INTRODUCTION

Most research on low-density parity-check (LDPC) codes has been focused on binary LDPC codes. While binary LDPC codes were shown to perform very close to capacity asymptotically [2] [3], it has also been shown that LDPC codes over larger fields can significantly outperform binary LDPC codes at short block lengths [4]. The performance gain over the binary codes, however, comes at a cost in decoding complexity which increases sharply with the size of the field. As a result, nonbinary LDPC codes may not be practical in some applications despite the considerable performance advantage.

Over the last few years, the advent of new decoding algorithms with reduced complexity, e.g. [5] [6] and a better understanding of the design [7] [8] [9] have spurred renewed interest in nonbinary LDPC codes. In this paper we explore their ability to adapt the coding rate to different channel conditions with a single encoder/decoder pair. In particular, we investigate puncturing to achieve higher code rates and shortening to achieve lower code rates.

Li et al. first considered puncturing LDPC codes in [11]. Later, Ha et al. developed framework for analysis of rate-compatible punctured (RCP) binary LDPC codes at both long [12] and short [13] block lengths. Other work followed, such as [14] [15] [17]. In [16] Tian et al. designed a system based on binary LDPC codes that can operate at code rates ranging from 0.1 to 0.9 using a single encoder/decoder pair. Starting from a mother code of rate 0.5 they employed puncturing based on the results in [12] to achieve rates up to 0.9 and they propose a shortening algorithm to lower the rate down to 0.1.

We will show that puncturing and shortening can be successfully applied to nonbinary LDPC codes at short block lengths (we consider lengths in the order of 1000). Our analysis considers only binary extension fields GF(q), such that q = 2^b and b is an integer. Transmission is assumed to be over a binary input additive white Gaussian noise (AWGN) channel, where a fixed-length binary representation is assigned to each field element. We will draw some parallels with the ideas devised for binary LDPC codes and see, that while certain principles conveniently extend to nonbinary LDPC codes, others must be modified for optimized performance. In the binary case the puncturing (or shortening) problem consists of merely choosing the variable nodes to puncture (or shorten), whereas the nonbinary case introduces another degree of freedom: the number of punctured (or shortened) bits per chosen variable node.

We will take a closer look at the design problem and show how rate-adaptive nonbinary codes can be constructed to have good performance. As we will show, if puncturing and shortening locations are selected carefully, nonbinary LDPC codes can considerably outperform their existing binary counterparts.

The subsequent part of this paper is organized as follows. Next section will briefly summarize relevant results from previous work and introduce the required notation. Section III describes puncturing of nonbinary LDPC codes, while section IV focuses on shortening. In section V we present the simulation results and finally, we conclude the paper with a brief summary.

II. BASIC DEFINITIONS AND NOTATION

We consider nonbinary LDPC codes over a binary input AWGN channel. Assume therefore that each field element (sometimes referred to as symbol) has a fixed length binary representation, given by ψ : GF(q) → Ω^b, where Ω = {0, 1} and b = log_q q.

A nonbinary LDPC code over GF(q), by analogy with binary LDPC codes, can be specified by a Tanner graph with the difference that its edges also bear multipliers. These multipliers can be any nonzero GF(q) symbols. They indicate that the corresponding variable node must be multiplied by it before it enters the parity check equation. Let the incidence matrix of a nonbinary LDPC code be a matrix whose (i, j) entry is 1 if the i-th variable node is connected to the j-th check node and 0 otherwise. Note that each symbol in a nonbinary codeword corresponds to exactly one variable node, therefore we take the liberty to use the two words interchangeably.
For a random variable $v$ over $\text{GF}(q)$ define its log-likelihood (LLR) vector as

$$L(v) = [L(v = \alpha_1), \ldots, L(v = \alpha_{q-1})],$$

where

$$L(v = \alpha_i) = \ln \frac{\Pr[v = \alpha_i]}{\Pr[v = 0]}.$$  

We assume a message-passing decoder in the log-likelihood domain introduced by Wyner and et al. [18], where messages exchanged between variable and check nodes are LLR vectors. For given channel observations of the transmitted symbol, the $i$-th component of the initial LLR vector reduces to [18]:

$$\sum_{j:(\psi^{-1}(\alpha_i))_j=1} \frac{2x_j}{\sigma^2},$$

where $(\psi^{-1}(\alpha_i))_j$ denotes the $j$-th bit in the binary representation of $\alpha_i$, $x_j$ is the channel observation of the $j$-th bit, and $\sigma^2$ is the noise variance.

Suppose now some symbols in a codeword are punctured. We define the puncturing pattern to be the set of all bits that are set to be punctured. If at least one bit of a symbol is punctured, we say that symbol is in the puncturing pattern. A symbol can either be punctured entirely, meaning that all bits in its binary representation are punctured, or partially, if some of the bits in its binary representation remain unpunctured. If all symbols in a puncturing pattern are punctured entirely, we say the code is punctured symbolwise; otherwise, it is punctured bitwise. The exact same notation is analogously defined for shortening.

We define the notion of symbol recovery analogously, but slightly differently than we did in [13]. Under assumption that all symbols are punctured entirely, we say a punctured symbol $v$ is $k$ step recoverable ($k$-SR), if it receives a first LLR vector with no zero components from one of its neighboring check nodes in the $k$-th iteration. We also say its level of recoverability of $v$ is $k$. The maximum level of recoverability in a puncturing pattern equals the highest level of recoverability of a punctured variable node in that pattern.

Note here the assumption that all symbols are punctured entirely. For example, if a set $P$ of symbols in a codeword is punctured partially, a symbol $v \in P$ is $k$-SR if in the case when all symbols in $P$ are punctured entirely, $v$ receives a first LLR vector with no zero components in the $k$-th iteration.

### Table I

<table>
<thead>
<tr>
<th># bits/symbol</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td># zero comp. in the init. LLR vector</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>15</td>
<td>31</td>
<td>63</td>
</tr>
</tbody>
</table>

In $\text{GF}(64)$, for instance, puncturing 5 bits in a symbol rather than all 6 reduces the number of zero components in the initial LLR vector from 63 to 31. On the other hand, reducing the number of punctured bits per symbol from 2 to 1 only slightly reduces the number of zero components. By decreasing the number of punctured bits per variable node, we decrease the initial uncertainty about the variable node in the decoder. Yet, that also requires an increased number of partially punctured variable nodes to achieve a certain code rate. Clearly, there is a tradeoff between the overall number of punctured variable nodes (entirely or partially) and the initial uncertainty of the punctured variable nodes in the decoder.

Our observations in numerous experiments indicate that puncturing many bits in symbols with high levels of recoverability has increasingly negative effects on performance. Thus, puncturing many bits in symbols with high levels of recoverability should be avoided if possible. Furthermore, the observations suggest that at intermediate rates (where we have the flexibility), bitwise puncturing should be employed rather than symbolwise puncturing.

We propose a design of puncturing patterns using the following two-step process.

**Step 1: Symbol Grouping**

We want to ensure that punctured symbols can be recovered and that they are recovered as early as possible. Toward this end we first apply the grouping algorithm from [13] on the Tanner graph corresponding to the incidence matrix of a given nonbinary LDPC code. The algorithm starts by searching for 1-SR nodes and it tries to maximize their number. When no more 1-SR nodes can be found, it proceeds with 2-SR nodes, etc., until every variable node is set to be either punctured or unpunctured. The information obtained from this algorithm will indicate which variable nodes are allowed to be punctured (entirely or partially). We can hence say that Step 1 reduces the search space from all bits in a codeword to exclusively those that pertain to the chosen variable nodes.

**Step 2: Bitwise Spreading at Intermediate Rates**

As we mentioned in the above discussion, puncturing variable nodes entirely leads to a high degree of uncertainty in the decoder and should therefore be avoided if possible. At the highest code rate we do not have much flexibility as we have to puncture all variable nodes chosen in Step 1 entirely. However, at intermediate rates we propose switching to bitwise puncturing. Puncturing many bits per symbol should be performed exclusively on variable nodes with low levels of recoverability, preferably 1. At the same time, puncturing should be spread uniformly over all variable nodes of the same...
level of recoverability. This way the number of punctured bits per variable node at a given level is minimized.

If the punctured patterns are required to be rate-compatible, some additional care is needed at intermediate rates, but generally this is done fairly easily.

IV. SHORTENING NONBINARY LDPC CODES

In the previous section we showed how higher rate codes can be obtained from a given mother code. Now we proceed to show how the rate can be lowered by shortening information bits. Shortening is a process where a subset of information bits is set to a value which is known to both encoder and decoder, while the number of parity checks remains unchanged. The number of transmitted information bits is thus reduced and, in effect, the code rate becomes lower. Let \( K \) be the number of variable nodes carrying information symbols, \( N \) the number of all variable nodes, \( K_a \) the number of shortened bits, \( R_s \) the code rate after shortening, and \( b \) the number of bits/symbol. We then have the following relations:

\[
R_s = \frac{K \cdot b - K_a}{N \cdot b - K_a},
\]

\[
K_a = \frac{b(K - N \cdot R_s)}{1 - R_s}.
\]

Tian et al. considered shortening binary LDPC codes in [16]. They observe that, in the binary case, shortening a bit can be viewed as deleting the corresponding column from a parity check matrix, or equivalently, as deleting the corresponding variable node and its edges from the Tanner graph. Therefore, they propose shortening bits in such a way that the resulting Tanner graph exhibits an optimized degree distribution.

Quite analogously to puncturing, when we move from binary codes to nonbinary LDPC codes, the shortening problem becomes one of not only choosing which variable node to shorten, but also how many bits per variable node should be shortened. Using the same arguments as with puncturing, we claim that bitwise shortening yields better performance as symbolwise shortening. In this case, shortened bits should be spread over all variable nodes carrying information bits such that the number of shortened bits per information variable node is as uniform as possible. This simple process is captured by the following algorithm.

1) Assume \( K_a \) bits need to be shortened;
2) \( a = \lfloor K_a/K \rfloor, i_1 = K_a - K \cdot a, i_2 = K_a - i_1; \)
3) Randomly choose \( i_1 \) information variable nodes and shorten \( a + 1 \) bits on each of them;
4) Shorten \( a \) bits on the remaining \( i_2 \) information variable nodes;

V. SIMULATION RESULTS

To verify the effectiveness of the proposed solution, we design a system that supports code rates ranging from 0.1 to 0.9, following the example of [16]. For a representative of nonbinary LDPC codes we choose a regular (2, 4) mother code over \( \text{GF}(64) \) with a block length of 142. Since each symbol in \( \text{GF}(64) \) is represented by 6 bits, one unpunctured codeword requires 852 channel uses on a binary input channel. As its binary competitor, we choose an irregular binary LDPC code from [16] with a block length of 852 and the degree distribution:

\[
\begin{align*}
\lambda(x) & = 0.25105x + 0.30938x^2 + 0.00104x^3 + 0.43853x^4, \\
\rho(x) & = 0.63676x^5 + 0.36324x^7.
\end{align*}
\]

Both parity check matrices were generated using the Progressive Edge Growth (PEG) algorithm [7]. The edge multipliers for the nonbinary code were chosen randomly from all nonzero \( \text{GF}(64) \) elements for each edge. The number of maximum iterations was set to 50 in all experiments.

First, we focus on increasing the rate by means of puncturing. We created a puncturing pattern that achieves the code rate 0.9 for both codes using the algorithm from [13]. The group distribution is shown in Table II.

<table>
<thead>
<tr>
<th>Level of Recoverability</th>
<th>Binary</th>
<th>GF(64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>852</td>
<td>142</td>
</tr>
</tbody>
</table>

In order to evaluate the validity of our puncturing approach we design four different puncturing patterns at the code rate 0.8 and evaluate their performance. First, we choose punctured bits completely at random. Second, we puncture the code symbolwise according to [13]. Third, we randomly select the pattern such that each punctured bit must belong to one of the variable node selected in Step 1, and the fourth pattern was obtained according to the proposed algorithm in Section III.

Figure 1 shows the results of this comparison. Clearly, puncturing bits randomly is not a good strategy. It yields very poor performance and at high rates carelessly selected puncturing patterns can result in unrecoverable symbols. Symbolwise punctured code according to [13] ensures that all punctured bits are recovered quickly and the performance is significantly improved. Random bitwise puncturing after Step 1 actually performs worse than symbolwise puncturing. However, by systematically reducing the number of punctured bits per symbol and therewith the uncertainty in the decoder, as proposed in Step 2, the performance can be further improved by 0.3 dB as compared to the algorithm in [13] (this comparison together with all others that follow are taken at the bit error rate of 10^{-5}). The design approach proposed in Section III is therefore a valid solution for the puncturing problem that yields good performance.
We further compare the performance of the proposed puncturing algorithm with symbolwise puncturing from [13] at all considered intermediate code rates: 0.6, 0.7, and 0.8. At the highest considered rate of 0.9 all symbols must be punctured entirely, therefore both approaches are equal. The bitwise patterns for each rate are given in Table III.

**TABLE III**  
**THE BREAKDOWN OF BITWISE PUNCTURED PATTERNS FOR THE GF(64) LDPC CODE.** *The given numbers (except for the bottom row) indicate the number of variable nodes.*

<table>
<thead>
<tr>
<th>Level</th>
<th># of punct. bits</th>
<th>0.6</th>
<th>0.7</th>
<th>0.8</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-SR</td>
<td>6 bits</td>
<td>48</td>
<td>42</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2-SR</td>
<td>6 bits</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 bits</td>
<td></td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3-SR</td>
<td>6 bits</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all punct. bits</td>
<td>144</td>
<td>246</td>
<td>318</td>
<td>384</td>
<td></td>
</tr>
</tbody>
</table>

Notice at rate 0.6, for instance, how instead of puncturing only 24 variable nodes entirely, the bitwise pattern was spread over all 1-SR variable nodes. As the required number of punctured bits is small, puncturing variable nodes of higher levels of recoverability was not necessary. At the rates 0.7 and 0.8, the number of required punctured bits was high enough that we had to partially puncture 2-SR variable nodes, while at the highest rate of 0.9 we had to use symbolwise puncturing.

Figure 2 shows the simulation results. Notice that the proposed puncturing consistently outperforms symbolwise puncturing from [13]. The performance gain ranges between 0.3 dB and 0.5 dB.

We also compare how our system compares to an equivalent binary system at rates 0.5–0.9. The puncturing patterns for the binary code were chosen according to [13]. As can be seen from Figure 3, nonbinary codes clearly outperform binary codes and the performance gain steadily increases with the rate. At the highest rate of 0.9 it equals 1.5 dB.

**Fig. 1.** Performance comparison between four differently chosen puncturing patterns at rate 0.8. The star markers correspond to randomly selected puncturing pattern, triangle markers represent symbolwise puncturing according to [13], square markers represent random bitwise puncturing after Step 1 and circle markers represent the proposed puncturing.

**Fig. 2.** Performance comparison between the proposed puncturing and symbolwise puncturing from [13] at rates 0.6, 0.7 and 0.8. Filled circles represent proposed puncturing, while unfilled circles represent symbolwise puncturing.

**Fig. 3.** Performance comparison between rate-adaptive nonbinary and binary codes at rates 0.5, 0.6, 0.7, 0.8, and 0.9. Filled circles represent nonbinary codes, and unfilled circles represent binary codes.

**TABLE IV**  
**CODING GAINS OF PROPOSED NONBINARY LDPC CODES OVER THEIR BINARY COUNTERPARTS.**

<table>
<thead>
<tr>
<th>code rate</th>
<th>0.5</th>
<th>0.6</th>
<th>0.7</th>
<th>0.8</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>gain over binary</td>
<td>0.3 dB</td>
<td>0.4 dB</td>
<td>0.5 dB</td>
<td>0.8 dB</td>
<td>1.5 dB</td>
</tr>
</tbody>
</table>

Next, we turn to shortening. Following the same approach, we first verify the validity of our shortening algorithm. On the one hand we shorten the code as proposed in Section IV, and on the other we shorten the code symbolwise, such that the
check node degree distribution stays as uniform as possible. The shortening patterns for all considered rates are shown in Table V.

<table>
<thead>
<tr>
<th>codes</th>
<th># of short. bits</th>
<th>0.1</th>
<th>0.2</th>
<th>0.3</th>
<th>0.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>6 bits</td>
<td>63</td>
<td>53</td>
<td>40</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>5 bits</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 bits</td>
<td>34</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 bits</td>
<td>37</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 bits</td>
<td>27</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 bit</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all short. bits</td>
<td>378</td>
<td>318</td>
<td>240</td>
<td>138</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 4.** Performance comparison between the proposed shortening and symbolwise shortening at rates 0.1–0.4. Filled circles represent proposed shortening, while unfilled circles represent symbolwise shortening.

According to the results in Figure 4 both approaches perform similarly at rates 0.3 and 0.4, while at the lower rates the proposed approach outperforms symbolwise shortening. The gap at rate 0.1 is close to 1 dB.

Finally we check how shortened nonbinary codes compare against their binary counterparts. Toward that end we perform shortening on the binary mother code as proposed in [16] to obtain rates 0.4, 0.3, 0.2, and 0.1. The performance results in Figure 5 confirm that shortened nonbinary codes significantly outperform binary codes and the gap increases with the lower rate. At the lowest considered rate of 0.1 the performance gap is around 2 dB.

**VI. CONCLUSION**

We considered puncturing and shortening of nonbinary LDPC codes at short block lengths. In contrast to binary codes, which offer only one degree of freedom: the choice of variable nodes to be punctured/shortened, nonbinary codes introduce another one: the number of bits that should be punctured/shortened per chosen variable node.

**REFERENCES**


