On the Guaranteed Error-Correction of Decimation-Enhanced Iterative Decoders

Shiva K. Planjery †, David Declercq †, Madiagne Diouf †, and Bane Vasić ‡

† ETIS, ENSEA-Univ. Cergy-Pontoise-CNRS, Cergy-Pontoise, France
‡ Dept. of Electrical and Computer Engineering, University of Arizona, Tucson, AZ, USA

8th International Symposium on Turbo Codes and Iterative Information Processing
Bremen, Germany
August 19, 2014
Outline

1. Importance and Challenge of proving guaranteed error-correction
2. Decimation-enhanced Finite Alphabet Iterative Decoders
3. Analysis to prove guaranteed error-correction
Outline

1. Importance and Challenge of proving guaranteed error-correction
2. Decimation-enhanced Finite Alphabet Iterative Decoders
3. Analysis to prove guaranteed error-correction
Important for several applications and for the BSC, as it governs the slope of the error floor [Ivkovic et al. 2008]

Finite-length analysis: Computation trees [Wiberg 1996], Stopping sets [Di et al. 2002], graph-cover decoding [Vontobel and Koetter]

Dynamics of message-passing still too complex to analyze for guaranteed error-correction.

Only known result with finite length constraints: For column-weight-three codes, $d_v = 3$ of girth $g \geq 10$, Gallager-A/B corrects $t = (g/2 - 1)$ errors in $l = g/2$ iterations, and for $g = 8$, it corrects $t = 3$ errors with additional constraints on the graph [Chilappagari et al. 2010].
Guaranteed Error-Correction

Our approach

- **Finite alphabet iterative decoders (FAIDs):** 3-bit decoders that surpass floating-point BP in the error floor [Planjery et al. 2010].

- **Decimation-enhanced FAIDs:** Decimation was proposed for FAIDs to make them more amenable to analysis [Planjery et al. 2011].

- Decimation involves **fixing the bit values** of certain variable nodes based on the messages passed.

**Our goal:** Present a methodology to analyze FAIDs for guaranteed error-correction capability with the help of decimation as a tool.

- **Sufficient conditions** can be derived on the graph to ensure correction of all error patterns of weight-$t$.

- Test case presented for $t = 4$ and $(d_v = 3, g = 8)$ LDPC codes.
Outline

1. Importance and Challenge of proving guaranteed error-correction
2. Decimation-enhanced Finite Alphabet Iterative Decoders
3. Analysis to prove guaranteed error-correction
Finite Alphabet Iterative Decoder

Definitions

- Messages \( \{m_i\} \) belong to a finite alphabet \( \mathcal{M} \) of cardinality \( N_s \). For this work we use \( N_s = 7 \), which gives

\[
\mathcal{M} = \{-L_3, -L_2, -L_1, 0, +L_1, +L_2, L_3\}
\]

where \( L_k > L_{k-1} \).

- By convention, negative messages correspond to bit value being 1 and positive messages correspond to bit value being 0,

- Message \( L_k \) signifies more confidence in the bit being zero than \( L_{k-1} \)

- The iterative decoder is defined by two Boolean maps: \( \Phi_v(y_i, m_1, \ldots, m_{dv-1}) \) for the variable node and \( \Phi_c(m_1, \ldots, m_{dc-1}) \) for the check node.

- For the BSC, the set of possible decoder inputs also called channel values is \( \mathcal{Y} = \{+C, -C\} \).
Finite Alphabet Iterative Decoder

Definitions

- The **check-node update** is same as the min-sum decoder

\[ m_{d_c} = \Phi_c(m_1, \ldots, m_{d_c - 1}) = \left( \prod_{j=1}^{d_c-1} \text{sgn}(m_j) \right) \min_{j \in \{1, \ldots, d_c - 1\}} (|m_j|) \]

- The **variable node update** is a simple look-up table (for \( d_v = 3 \) codes):

**Table**: \( \Phi_v(+C, m_1, m_2) \) of a 7-level FAID for a node \( v_i \) with \( d_v = 3 \) and \( y_i = +C \)

<table>
<thead>
<tr>
<th>( m_1 / m_2 )</th>
<th>(-L_3)</th>
<th>(-L_2)</th>
<th>(-L_1)</th>
<th>0</th>
<th>(+L_1)</th>
<th>(+L_2)</th>
<th>(+L_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-L_3)</td>
<td>(-L_3)</td>
<td>(-L_3)</td>
<td>(-L_2)</td>
<td>(-L_1)</td>
<td>(-L_1)</td>
<td>(-L_1)</td>
<td>(+L_1)</td>
</tr>
<tr>
<td>(-L_2)</td>
<td>(-L_3)</td>
<td>(-L_1)</td>
<td>(-L_1)</td>
<td>0</td>
<td>(+L_1)</td>
<td>(+L_1)</td>
<td>(+L_3)</td>
</tr>
<tr>
<td>(-L_1)</td>
<td>(-L_2)</td>
<td>(-L_1)</td>
<td>0</td>
<td>0</td>
<td>(+L_1)</td>
<td>(+L_2)</td>
<td>(+L_3)</td>
</tr>
<tr>
<td>0</td>
<td>(-L_1)</td>
<td>0</td>
<td>0</td>
<td>(+L_1)</td>
<td>(+L_2)</td>
<td>(+L_3)</td>
<td>(+L_3)</td>
</tr>
<tr>
<td>(+L_1)</td>
<td>(-L_1)</td>
<td>(+L_1)</td>
<td>(+L_1)</td>
<td>(+L_2)</td>
<td>(+L_2)</td>
<td>(+L_3)</td>
<td>(+L_3)</td>
</tr>
<tr>
<td>(+L_2)</td>
<td>(-L_1)</td>
<td>(+L_2)</td>
<td>(+L_2)</td>
<td>(+L_3)</td>
<td>(+L_3)</td>
<td>(+L_3)</td>
<td>(+L_3)</td>
</tr>
<tr>
<td>(+L_3)</td>
<td>(+L_1)</td>
<td>(+L_3)</td>
<td>(+L_3)</td>
<td>(+L_3)</td>
<td>(+L_3)</td>
<td>(+L_3)</td>
<td>(+L_3)</td>
</tr>
</tbody>
</table>
Decimation-enhanced FAID
Incorporating decimation at the variable node update

- **Decimation**: Freeze the bit value of a variable node based on its incoming messages at the end of the $l^{th}$ iteration.

- A decimated node $v_i$ will always send the strongest possible messages $\in \{-L_3, +L_3\}$.

- A decimation rule $\beta(C, m_1, m_2, m_3) \rightarrow \{-1, 0, 1\}$ is used to decide whether a variable node should be decimated and what value it should be decimated to.

- $\gamma_i$ is the output of $\beta$ when applied to a node $v_i$.
  - $\gamma_i = +1 \Rightarrow v_i$ is decimated to zero
  - $\gamma_i = -1 \Rightarrow v_i$ is decimated to one
  - $\gamma_i = 0 \Rightarrow v_i$ is not decimated.

- **Decimation round**: After $l$ iterations, nodes are being decimated using $\beta$, and the decoder is re-initialized.
Decimation-enhanced FAID

Description of $\beta$

- Defined by the set $\Xi$ which consists of all unordered triples $(m_1, m_2, m_3) \in \mathcal{M}^3$ that decimate a node to zero.

Table: Set $\Xi$ consisting of all message triples such that $\beta(C, m_1, m_2, m_3) = 1$

<table>
<thead>
<tr>
<th>$m_1$</th>
<th>$m_2$</th>
<th>$m_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_3$</td>
<td>$L_3$</td>
<td>$L_3$</td>
</tr>
<tr>
<td>$L_3$</td>
<td>$L_3$</td>
<td>$L_2$</td>
</tr>
<tr>
<td>$L_3$</td>
<td>$L_3$</td>
<td>$L_1$</td>
</tr>
<tr>
<td>$L_3$</td>
<td>$L_3$</td>
<td>0</td>
</tr>
<tr>
<td>$L_3$</td>
<td>$L_3$</td>
<td>$-L_1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$m_1$</th>
<th>$m_2$</th>
<th>$m_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_3$</td>
<td>$L_2$</td>
<td>$L_2$</td>
</tr>
<tr>
<td>$L_3$</td>
<td>$L_2$</td>
<td>$L_1$</td>
</tr>
<tr>
<td>$L_3$</td>
<td>$L_2$</td>
<td>0</td>
</tr>
<tr>
<td>$L_3$</td>
<td>$L_2$</td>
<td>$-L_1$</td>
</tr>
<tr>
<td>$L_3$</td>
<td>$L_1$</td>
<td>$L_1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$m_1$</th>
<th>$m_2$</th>
<th>$m_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_3$</td>
<td>$L_1$</td>
<td>0</td>
</tr>
<tr>
<td>$L_3$</td>
<td>$L_1$</td>
<td>$-L_1$</td>
</tr>
<tr>
<td>$L_3$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$L_2$</td>
<td>$L_2$</td>
<td>$L_2$</td>
</tr>
<tr>
<td>$L_2$</td>
<td>$L_2$</td>
<td>$L_1$</td>
</tr>
</tbody>
</table>

- The cardinality of $\Xi$ determines how aggressive or conservative the rule $\beta$ is:
Decimation-enhanced FAID
Decoding algorithm used for the Analysis

1. Initially all messages are set to zero.
2. Run the decoder as a typical FAID for $l = 3$ iterations using $\Phi_v$ and $\Phi_c$.
3. At the end of $l = 3$ iterations, perform a decimation round using $\beta$ (first decimation round).
4. Restart decoder, and run for one iteration. Then perform decimation on the non-decimated nodes.
5. If additional nodes get decimated, go back to Step 4. Else proceed,
6. Continue the decoder using maps $\Phi_v$ and $\Phi_c$ for remainder of iterations on the residual graph.

In Step 6, the residual graph is the graph induced by the remaining non-decimated nodes.
1. Importance and Challenge of proving guaranteed error-correction

2. Decimation-enhanced Finite Alphabet Iterative Decoders

3. Analysis to prove guaranteed error-correction
Assumptions for Analysis

- We consider \((d_v = 3, g = 8)\) LDPC codes, \(t = 4\) error patterns and \(N_s = 7\) DFAID.
- Constraints on the DFAID map:
  \[\Phi_v(C, 0, 0) = L_1, \Phi_v(C, L_1, L_1) = L_2, \text{ and } \Phi_v(C, L_2, L_2) = L_3.\]

Our approach:

- Expand the neighborhood of error patterns of weight-\(t\) into a computation tree.
- Analysis will mainly involve examining whether a node is decimated or not through their computation tree.
- For a given error pattern, sufficient conditions in the form of additional constraints on the code structure will be derived.

Three main steps for analysis:

1. **Step 1:** Analyzing the decimation of error nodes, and proving that no error node gets decimated (necessary condition).
2. **Step 2:** Analyzing the decimation of correct nodes in the neighborhood of the error pattern, to determine the residual graph.
3. **Step 3:** Proving that the error pattern is corrected in \(l\) iterations.
Lemma

If $\beta(C, L_1, L_1, L_1) = 0$ and no error node gets decimated during first decimation round, then no error node will get decimated in any subsequent decimation round.

- Due to the above lemma, it suffices to prove that no error node gets decimated at the end of third iteration (first decimation round).

- Due to the monotonicity property of $\beta$, we need to only examine the worst-case messages an error node receives, i.e. the most negative messages it can receive.

- If worst-case message triple is not included in the set $\Xi$, then the node does not get decimated.
Step 1: Analyzing Decimation of Error Nodes

Example: 4-error pattern on 8-cycle

\(H\) denotes the graph induced by error nodes (graph of 8-cycle).

- \(\bullet\) - error nodes, \(\bigcirc\) - correct nodes outside \(H\)
Step 1: Analyzing Decimation of Error Nodes

Example: 4-error pattern on 8-cycle

\( H \) denotes the graph induced by error nodes (graph of 8-cycle).

- error nodes, \( \circ \) - correct nodes outside \( H \)

Assume that this correct node sends worst possible message in third iteration.

Assume that this correct node sends worst possible message in third iteration.
Step 1: Analyzing Decimation of Error Nodes

Example: 4-error pattern on 8-cycle

$H$ denotes the graph induced by error nodes (graph of 8-cycle).

- error nodes, ○ - correct nodes outside $H$

These checks could be either in $H$ or outside $H$
**Step 1: Analyzing Decimation of Error Nodes**

Example: 4-error pattern on 8-cycle

\( H \) denotes the graph induced by error nodes (graph of 8-cycle).

- error nodes, \( \bigcirc \) - correct nodes outside \( H \)

To obtain most negative messages, they must be the degree-1 checks of \( H \).
Step 1: Analyzing Decimation of Error Nodes

Example: 4-error pattern on 8-cycle

Due to girth-8 constraint, $v_6$ and $v_8$ each can be connected to at most one other degree-1 check in $H$. Since $\beta(C, L_1, L_1, L_1) = 0 \iff \beta(-C, -L_1, -L_1, -L_1) = 0$, the error node $v_1$ will not get decimated.
Step 1: Analyzing Decimation of Error Nodes
Repeat analysis for all possible 4-error patterns

Lemma

If the graph $G$ of a column-weight-three code has girth-8, then for any 4-error pattern, no error node will get decimated during decoding by the 7-level DFAID.

This systematic approach can also be useful for designing the most aggressive decimation rule $\beta$ that ensures that no error node is decimated.
**Principle:** we expand further the computation tree under additional graphical constraints: *girth, absence of a particular sub-graph.*

The following verifications are performed on the computation tree:

- We examine whether any correct nodes in the neighborhood of $H$ can get decimated in the first decimation round.

- For correct nodes **that are decimated in the first round**, we examine other correct nodes that they shares a check node with by graph expansion.

**Result:** at the end of the iterative procedure, the nodes that do not get decimated at any decimation round constitute the **residual graph**.
Step 3: Guaranteed Error-Correction

Additional constraints on the neighborhood of 8-cycle containing the 4-error pattern

The above set of graphs $G$ are the forbidden graphs on the neighborhood of the 8-cycle in graph $G$ that were enforced to ensure that all correct nodes outside the 8-cycle are decimated.

$G(a, b)$ denotes a forbidden graph with $a$ variable nodes and $b$ check nodes.

These graphs constitute additional constraints and form sufficient conditions to ensure guaranteed correction of the 4-error pattern.
Step 3: Guaranteed Error-Correction

**Theorem**

*If the graph $G$ of a column-weight-three code has girth-8, and does not contain any subgraphs belonging to $\mathcal{G}$, then the 7-level DFAID requires at most two decimations, and $l = 3$ iterations to correct the 4-error patterns whose induced subgraph forms an 8-cycle.*

- We also repeat the same procedure for any 4-error pattern that does not form an 8-cycle (but the 8-cycle case is the worst case).
- Note that the analysis also provides the number of decimations rounds required to reach the residual graph.
- Remark: Gallager-A/B cannot correct the 4-error pattern
Conclusions

1. We presented a systematic approach for analyzing FAIDs for guaranteed error-correction.

2. The use of decimation makes FAIDs more amenable to analysis as we are dealing with depth-3 pruned computation trees.

3. Necessary conditions on the decoder (update rules) are derived to ensure that no error node can be decimated.

4. Sufficient conditions on the graph constraints can be derived to ensure decimation of correct nodes outside the residual graph.

5. The approach is not restricted to a particular code, except that it has to have $(d_v = 3, g = 8)$.

6. The sufficient conditions on the sub-graphs are too strong for efficient code design. Relax these conditions by adapting the decimation rule $\beta$?