ECEN 655: Advanced Channel Coding Spring 2014

Lecture 7 — 02/04/14

Prof. Henry Pfister *Scribe: Mengke Lian*

1 4-Cycles in Gallager's Ensemble

What we already know: Belief propagation is exact on tree-structured factor graphs.

A reasonable assumption: For a fixed ensembles of codes, it is better to have fewer cycles.

Gallager's construction: What is the expected number of bits involved in 4-cycles?

1.1 4-Cycles in $(2,k)$-LDPC Code

Consider the number of bits involved in 4-cycles. Since the Gallager ensemble is invariant under column permutation, we have

$$\mathbb{E}\left[\sum_{i=1}^{n}\mathbb{I}\left(\text{bit } i \text{ in 4-cycle}\right)\right] = n\mathbb{P}\left(\text{bit 1 in 4-cycle}\right)$$

This observations allows one to focus on 4-cycles involving the first bit of the code.

For any H from the Gallager ensemble, one can always permute the columns so that the first n/k rows are equal to Q. For example, multiplying H on the right by P_1^T does this because $P_1P_1^T = I$. This holds becaue a permutation preserves all p-norms and hence P_1 must be unitary. We can also permute the rows of the second Q-block (i.e., rows $n/k+1$ to $2n/k$) so that the first row of the 2nd Q-block (i.e., row $n/k+1$) has a one in the first position. Let $\tilde{H} = P_0HP_1^T$ be the parity-check matrix after these permutations (i.e., P_0 defines the necessary row permutation). For a $(2,k)$ code, the first code bit will be involved in a 4-cycle iff row $n/k+1$ of $\tilde{H}$ has a one is column positions $2,3,\ldots,k$.

$$\tilde{H} = \left[\begin{array}{cccccccccccccccc}
1 & 1 & \cdots & 1 & 0 & 0 & \cdots & 0 & \cdots & \cdots & \cdots & \cdots & 0 & 0 & \cdots & 0 \\
0 & 0 & \cdots & 0 & 1 & 1 & \cdots & 1 & \ddots & \cdots & \cdots & \cdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \ddots & \cdots & \cdots & \cdots & \ddots & \cdots & \cdots & \ddots & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ddots & \cdots & \cdots & \cdots & 1 & 1 & \cdots & 1 \\
\hline
1 & a_2 & \cdots & a_k & a_{k+1} & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & a_n \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{array}\right]$$

From the above diagram, we observe that row $n/k+1$ is a binary vector with a one in the first position and $k-1$ ones distributed uniformly across the other positions. The first bit will be

involved in a 4-cycle iff $a_l = 1$ for $l \in \{2, 3, \ldots, k\}$. When one observes a permutation of a binary vector, there are many permutations that are indistinghuishable. The equivalence classes can be identified simply by where the ones are placed in the resulting vector. In this case, there are $\binom{n-1}{k-1}$ ways to choose the $a_2, \ldots, a_n$ values that satisfy the constraints. Of these choices, there are $\binom{n-k}{k-1}$ ways that have $a_2 = a_3 = \cdots = a_k = 0$. Thus, we find that

$$\begin{aligned}\mathbb{P}\left(\text{bit 1 not in 4-cycle}\right) &= \frac{\#\text{ way to choose } k-1 \text{ ones from } n-k \text{ positions}}{\#\text{ way to choose } k-1 \text{ ones from } n-1 \text{ positions}} \\ &= \frac{\binom{n-k}{k-1}}{\binom{n-1}{k-1}} = \prod_{i=0}^{k-2}\left(\frac{n-k-i}{n-1-i}\right) = \prod_{i=0}^{k-2}\left(\frac{1-\frac{k+i}{n}}{1-\frac{i+1}{n}}\right)\end{aligned}$$

Using the Taylor expansion, one can verify that

$$\begin{aligned}\frac{1-ax}{1-bx} &= 1 - ax + bx + \mathrm{O}\big(x^2\big) \\ (1-ax)(1-bx) &= 1 - ax - bx + \mathrm{O}\big(x^2\big).\end{aligned}$$

Hence, for a $(2, k)$-LDPC code, the probability that bit 1 is involved in a 4-cycle is given by

$$\begin{aligned}\mathbb{P}\left(\text{bit 1 in 4-cycle}\right) &= 1 - \mathbb{P}\left(\text{bit 1 not in 4-cycle}\right) \\ &= 1 - \prod_{i=0}^{k-2}\left(\frac{1-\frac{k+i}{n}}{1-\frac{i+1}{n}}\right) \\ &= 1 - \prod_{i=0}^{k-2}\left(1 - \frac{k+1}{n} + \mathrm{O}\big(n^{-2}\big)\right) \\ &= \sum_{i=0}^{k-2}\left(\frac{k-1}{n} + \mathrm{O}\big(n^{-2}\big)\right) \\ &= \frac{(k-1)^2}{n} + \mathrm{O}\big(n^{-2}\big).\end{aligned}$$

Hence, we find that the expected number of bits involved in 4-cycles is

$$n\mathbb{P}\left(\text{bit 1 in 4-cycle}\right) = n\left(\frac{(k-1)^2}{n} + \mathrm{O}\big(n^{-2}\big)\right) \to (k-1)^2.$$

Since the number converges to a constant, this gives a hint that a clever choice of permutations might avoid all 4-cycles.

1.2 4-Cycles in (j, k)-LDPC Code

By induction, we can compute $\mathbb{P}\left(\text{bit 1 in 4-cycle}\right)$ in (j, k)-LDPC Code:

$$\mathbb{P}\left(\text{bit 1 in 4-cycle}\right) = 1 - \mathbb{P}\left(\text{bit 1 not in 4-cycle}\right) = 1 - \prod_{i=1}^{j-1}\frac{\binom{n-i(k-1)-1}{k-1}}{\binom{n-1}{k-1}}$$

This is again based on counting the equivalence classes of permutations that avoid 4-cycles for the first bit. Using the same approach as above, one can verify that

$$\lim_{n\to\infty} \mathbb{E}\left[\sum_{i=1}^{n} \mathbb{I}\left(\text{bit } i \text{ in 4-cycle}\right)\right] = (k-1)^2 \binom{j}{2}.$$

One can see this heuristically by noticing that there are $\binom{j}{2}$ ways to choose two sub-blocks of the parity-check matrix and form a $(2, k)$ LDPC code. Each of these will contribute $(k-1)^2$ "involved bits" on average.

1.3 Finding 4-Cycles

Let H be the parity-check matrix for LDPC code with its i-th column h_i. Then

$$\text{dot}(h_i, h_j) = \#\text{ overlapping ones in } h_i \text{ and } h_j$$

If `H` is a sparse matrix in Matlab, then one can compute

`Z = H' * H - diag ( sum(H,2) )` matrix containing # overlapping ones between columns of `H`

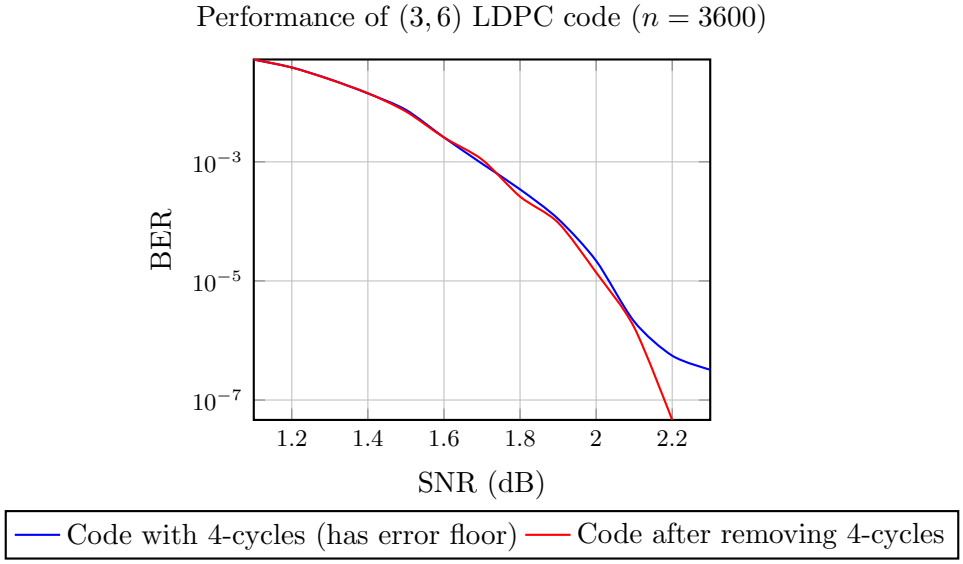
`[i,j,v] = find(Z>=2)` gives all pairs of columns that contain 4-cycles

1.4 Removing 4-Cycles

$$H = \left[\begin{array}{cccccccccccccccc}
\boxed{1} & 1 & 1 & \boxed{1} & & & & \cdots & \cdots & \cdots & \cdots & & & & & \\
 & & & & 1 & 1 & 1 & 1 & \cdots & \cdots & \cdots & \cdots & & & & \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & & & & & & & & \cdots & \cdots & \cdots & \cdots & 1 & 1 & 1 & 1 \\
\hline
\boxed{1} & 0 & 0 & \boxed{1} & 1 & & & \cdots & \cdots & \cdots & \cdots & 1 & \boxed{0} & & & \\
 & 1 & & & & 1 & & \cdots & \cdots & \cdots & \cdots & & 1 & & & \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & & & 1 & & & 1 & \cdots & \cdots & \cdots & \cdots & & 0 & & 1 &
\end{array}\right]$$

Randomly swap bad (sub-)column with any other (sub-)column in same block to remove 4-cycles. If the block length is sufficiently large, this will reduce the number of 4-cycles on average and eventually remove al 4-cycles.

The following simulation results for a $(3, 6)$ Gallager code with $n = 3600$ shows that effect of removing 4-cycles is larger in the error floor than in the waterfall region.



Note: While the above figure was generated by an actual experiment, the reader is warned that there can be significant variation in the performance of randomly chosen codes with and without 4-cycles.

1.5 Matlab Code for Finding & Removing 4-Cycles

```
function H = sampleGallager(j,k,n)
% Sample a random parity-check matrix from Gallager's (j,k)-regular
% ensemble
% Input:
% j - The degree of every variable node
% k - The degree of every check node
% n - The length of LDPC code
% Output:
% H - The parity-check matrix of the code
    col = zeros(j*n,1);
    row = zeros(j*n,1);
    for i = 0:j-1
        col((i*n+1):((i+1)*n)) = 1:n;
        row((i*n+1):((i+1)*n)) = i*n/k + ceil(randperm(n)/k);
    end
    H = sparse(row,col,1,j*n/k,n);
end

function H = remove_4Cycle(H)
% G random parity-check matrix in Gallager's (j,k)-regular ensemble, and
% try to remove all 4-Cycle
% Input:
% H - The parity-check matrix before removing 4-Cycle
% Output:
```

```
% H - The parity-check matrix after removing 4-Cycle
    n = size(H,2);  k = sum(H(1,:));    max_iter = 3000;     iter = 0;
    while iter < max_iter
        % Find 4-Cycles
        Z = H'*H - diag(sum(H,1));
        [col1,col2] = find(triu(Z)>=2);
        num_cycle = length(col1);
        % Check if all 4-Cycles are removed
        if num_cycle == 0
            fprintf('All 4-cycles have been removed after %d iteration\n', iter)
            break;
        end
        for i = 1:length(col1)
            % Find first same element
            index = find(H(:,col1(i)).*H(:,col2(i))==1,1,'first');
            if isempty(index) continue;    end
            % Find which level is the element in
            level = ceil(index/(n/k));
            % Compute row index of that level
            rows = ( (level-1) * n/k + 1 ) : ( level * n/k );
            % Randomly choose another sub-column in that level
            c = randsample( find(H(index,:) == 0), 1);
            % Swap the first sub-column and the random one
            H(rows,[col1(i) c]) = H(rows,[c col1(i)]);
        end
        iter = iter + 1;
    end

    if iter == max_iter
        [col1,col2] = find(triu(H'*H - diag(sum(H,1)))>=2);
        num_cycle = length(col1);
        fprintf('There are %d 4-cycles remaining after %d iteration\n', num_cycle, max_iter)
    end
end
```

2 LLR Messages in Decoding

The simplest approach is to use uniform quantization over some bounded region. In particular, we can irestrict range of LLRs to $[-L_{max}, L_{max}]$ and use a quantization step size of $\Delta = \frac{2L_{max}}{2^{\#\text{bits}}-1}$. This leads to symmetric 0-centered quantization.

While the implementation is more complex, non-uniform quantization allows one to achieve very low error rates with only a few bits of storage. For example, see `http://arxiv.org/abs/1212.6465`.

2.1 Simple Update Rules

Bit node:

$$\hat{L}_{i\to a} = \tilde{L}_i + \sum_{b\in F(i)\setminus a} L_{b\to i}$$

Saturate $\hat{L}$ to $\in [-L_{max}, L_{max}]$ (e.g., a value of $L_{max} = 15$ is sufficient in many cases).

Check node:

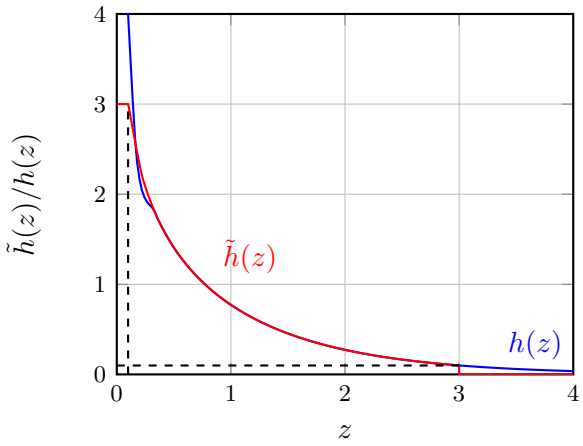
$$\hat{L}_{a\to i} = \tilde{h}\left(\sum_{j\in V(a)\setminus i} \tilde{h}\left(|L_{i\to a}|\right)\right) \prod_{j\in V(a)\setminus i} \operatorname{sgn}\left(L_{i\to a}\right)$$

where

$$h(z) = -\ln\left(\tanh\left(\frac{|z|}{2}\right)\right)$$

$$\tilde{h}(z) = \begin{cases} L_{max}, & z < h^{-1}(L_{max}) = h(L_{max}) \\ 0, & z > L_{max} \\ h(z), & \text{otherwise} \end{cases}$$

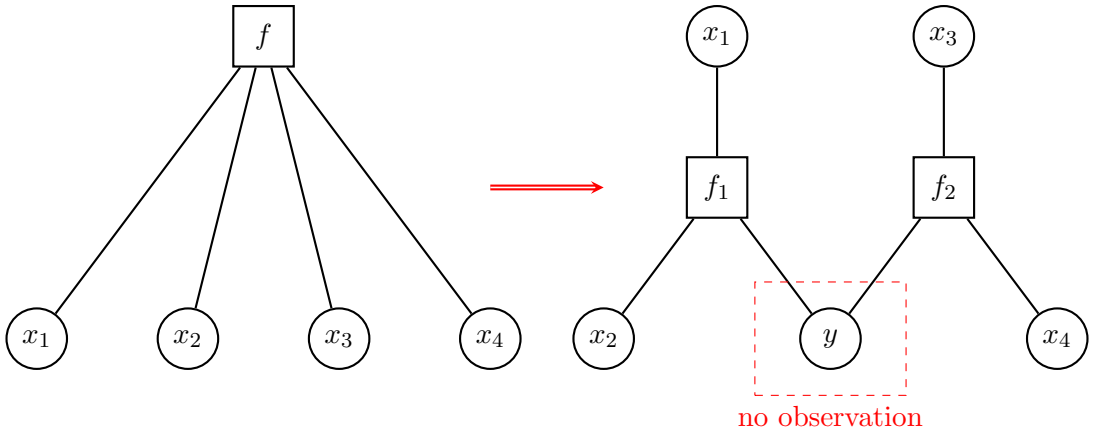
This function is plotted for the case of $L_{max} = 3$ where $h(L_{max}) \approx 0.099$.

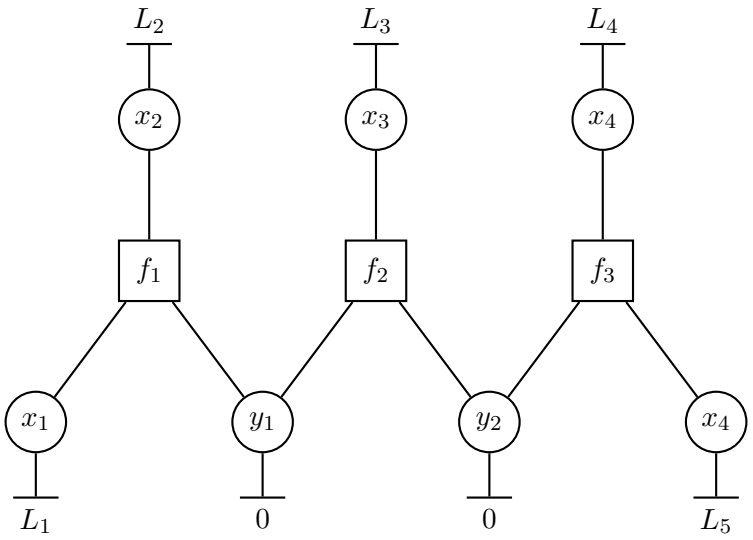


Note: While this approach to the check update rule is simple and can be made to work well in the waterfall region, the dynamic range of the $h(\cdot)$ function can be problematic when trying to achieve low error floors.

2.2 Forward-Backward Approach to Check-Node Rule

To use non-uniform quantization or improve numerical stability, one can use a forward-backward approach to the check-node update. This approach is based on factoring a degree-d check node into a chain of degree-3 check nodes.





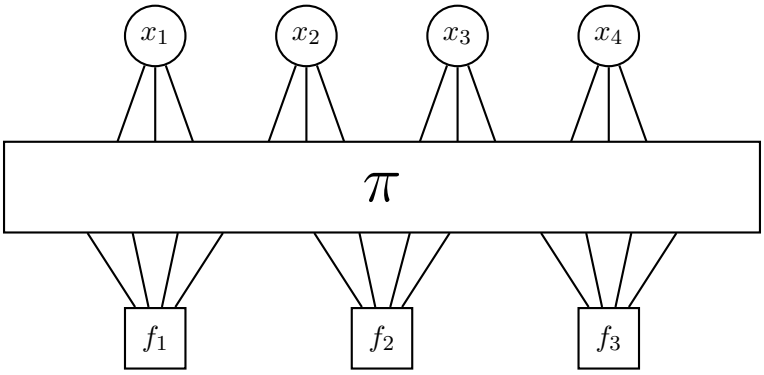
Forward-Backward Approach Using this approach, one can have arbitrary non-uniform quantization by using a lookup table for

$$a \boxplus b \triangleq 2 \tanh^{-1}\left(\tanh\left(\frac{a}{2}\right)\tanh\left(\frac{b}{2}\right)\right)$$

In the example, one can compute the output LLR L_3' associated with the input LLR L_3 using

$$\begin{aligned} L_{y_1} &= L_1 \boxplus L_2 \\ L_{y_2} &= L_4 \boxplus L_5 \\ L_3' &= L_{y_1} \boxplus L_{y_2} \end{aligned}$$

2.3 Data Storage



(1) Assume messages from the checks are stored in order by check and then edge number (i.e., the 1st edge of the 1st check followed by 2nd edge of 1st check, etc...)

(a) Let $c[i]$ be the i-th edge message in this order

(b) We will refer this edge ordering as the *natural order* for the checks

(c) Initialize all check messages to 0 (i.e., $c[i] = 0, \forall i \in 1, \ldots, |E|$)

(2) Assume messages from the bits are stored in order by bit and then edge number (i.e., the 1st edge of the 1st bit followed by 2nd edge of 1st bit, etc...)

(a) Let $b[i]$ be the i-th edge message in this order

(b) Likewise, this edge ordering is called the *natural order* for the checks

(3) Decoding proceeds as follows:

(a) Reorder check messages into natural order for bit messages: $b[i] = c[\pi(i)], \forall i \in 1, \ldots, |E|$

(b) Do bit message updates on $b[i]$ (note: all operations are memory local)

(c) Reorder bit messages into natural order check bit messages: $c[\pi(i)] = b[i], \forall i \in 1, \ldots, |E|$

(d) Do check message updates on $c[i]$ (note: all operations are memory local)

(4) Other advantages

(a) For a fixed bit and check node degrees, one can generate a random code simply by choosing π to be a uniform random permutation.