6. Low Density Parity-Check Codes

Gadiel Seroussi

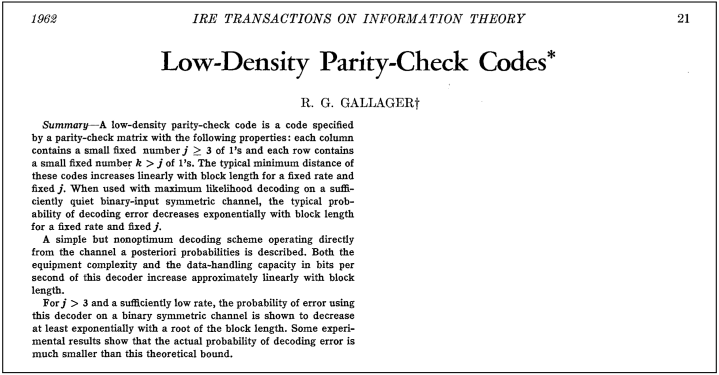
November 6, 2024

Low Density Parity-Check (LDPC) Codes

- A sequence of binary matrices $\{H_{r\times n}\}_{n\geq r\geq 1}$, is said to be of *low density* if the number of 1's in each row and column remains bounded as $r, n \to \infty$. Formally, $\mathrm{wt}\left(H_{r\times n}\right)/n \leq c$ for some constant c and all n.
- $H_{r\times n}$ used as parity-check matrices (PCMs) of linear codes. For "good" codes we will have $r \approx (1-R)n$ for some fixed $R \in (0,1)$.

Low Density Parity-Check (LDPC) Codes

- A sequence of binary matrices $\{H_{r\times n}\}_{n\geq r\geq 1}$, is said to be of *low density* if the number of 1's in each row and column remains bounded as $r, n \to \infty$. Formally, $\text{wt}\left(H_{r\times n}\right)/n \leq c$ for some constant c and all n.
- $H_{r\times n}$ used as parity-check matrices (PCMs) of linear codes. For "good" codes we will have $r \approx (1-R)n$ for some fixed $R \in (0,1)$.



1962 *IRE TRANSACTIONS ON INFORMATION THEORY* 21

Low-Density Parity-Check Codes*

R. G. GALLAGER†

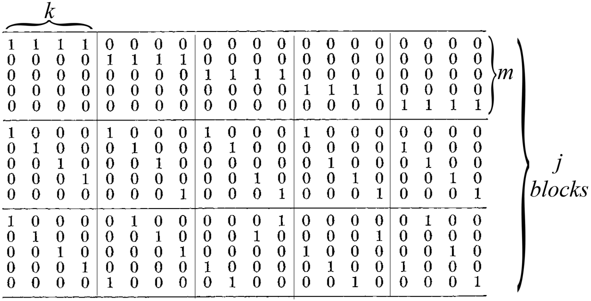
Summary—A low-density parity-check code is a code specified by a parity-check matrix with the following properties: each column contains a small fixed number $j \geq 3$ of 1's and each row contains a small fixed number $k > j$ of 1's. The typical minimum distance of these codes increases linearly with block length for a fixed rate and fixed j. When used with maximum likelihood decoding on a sufficiently quiet binary-input symmetric channel, the typical probability of decoding error decreases exponentially with block length for a fixed rate and fixed j.

A simple but nonoptimum decoding scheme operating directly from the channel a posteriori probabilities is described. Both the equipment complexity and the data-handling capacity in bits per second of this decoder increase approximately linearly with block length.

For $j > 3$ and a sufficiently low rate, the probability of error using this decoder on a binary symmetric channel is shown to decrease at least exponentially with a root of the block length. Some experimental results show that the actual probability of decoding error is much smaller than this theoretical bound.

Gallager's ensemble (1962)

Family of codes $\mathcal{G}(n, j, k)$, $j \leq k$, with parity-check matrix H:
$r \times n$ with j 1's per column, k 1's per row $(n = km,\ r = jm,\ m \geq 1)$.



$$\left.\begin{array}{cccc|cccc|cccc|cccc|cccc}
1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1\\
\hline
1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&0&0&0&0\\
0&1&0&0&0&1&0&0&0&1&0&0&0&0&0&0&1&0&0&0\\
0&0&1&0&0&0&1&0&0&0&0&0&0&1&0&0&0&1&0&0\\
0&0&0&1&0&0&0&0&0&0&1&0&0&0&1&0&0&0&1&0\\
0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1\\
\hline
1&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0\\
0&1&0&0&0&0&1&0&0&0&1&0&0&0&0&1&0&0&0&0\\
0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&1&0\\
0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&1&0&0&0\\
0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1
\end{array}\right.$$

Example of a low-density code matrix; $N = 20, j = 3, k = 4$.

- Last $j - 1$ blocks are random permutations of columns of first: *not systematic*.
- Rows of H not necessarily linearly independent $\Longrightarrow$ redundancy $\leq r$.
- Rate $R \geq \frac{n-r}{n} = 1 - \frac{j}{k}$.

LDPC codes are good

Theorem (Gallager codes approach BSC capacity)

Given a BSC of parameter p, *and a rate* $R < 1 - \mathsf{H}_2(p)$, *there exists an integer* $t(p, R)$ *such that ML decoding of a random Gallager code of rate* R *with LDPC columns of weight* t *achieves vanishing error probability with probability 1.*

LDPC codes are good

Theorem (Gallager codes approach BSC capacity)

Given a BSC of parameter p*, and a rate* $R < 1 - \mathsf{H}_2(p)$*, there exists an integer* $t(p, R)$ *such that ML decoding of a random Gallager code of rate* R *with LDPC columns of weight* t *achieves vanishing error probability with probability 1.*

Theorem (Gallager codes have good distance properties)

Given $\delta < \frac{1}{2}$ *and* R *such that* $R < 1 - \mathsf{H}_2(\delta)$*, there exists an integer* t *such that for sufficiently large* n *there exist Gallager codes of LDPC column weight* t *and parameters* $[n, \geq R, \geq n\delta]$ *(GV bound).*

A bit of history

Partial history, with some major milestones

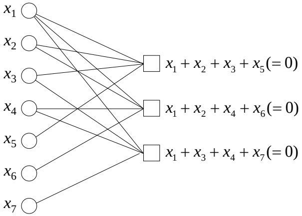
- Gallager [1962, Ph.D. Thesis and paper]
- Russian school: Zyablov, Pinsker, Margulis [1971, 1976, 1982]
- Tanner [1981] reinvention, graph approach, extensions.
- Berrou et al. [1993] Turbo Codes, iterative decoding, approach capacity
- Richardson, Urbanke [1998] Irregular LDPC codes and iterative threshold
- Mac Kay [1999], reinvention, analysis, and extensions
- Luby et al., [1997–] LT codes, Tornado codes, new analysis
- Shokrollahi [2000] Raptor codes
- and much research since then and ongoing ...

LDPC codes have become ubiquitous in many modern applications: 5G, magnetic and SSD storage, etc.

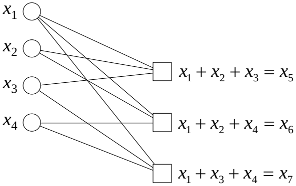
Graph representation of parity check matrix

$$\begin{array}{|ccccccc|} \multicolumn{4}{c}{\overbrace{\text{INFORMATION DIGITS}}} & \multicolumn{3}{c}{\overbrace{\text{CHECK DIGITS}}} \\ x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \\ \hline 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ \hline \end{array} \longleftrightarrow \begin{aligned} x_5 &= x_1 \oplus x_2 \oplus x_3 \\ x_6 &= x_1 \oplus x_2 \oplus x_4 \\ x_7 &= x_1 \oplus x_3 \oplus x_4 \end{aligned}$$

Two representations as a *bipartite graph*



Good for decoding

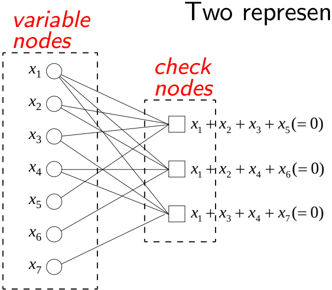


Good for encoding

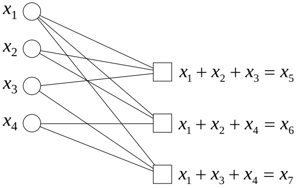
Graph representation of parity check matrix

$$\begin{array}{c} \overbrace{\begin{array}{cccc} x_1 & x_2 & x_3 & x_4 \end{array}}^{\text{INFORMATION DIGITS}} \overbrace{\begin{array}{ccc} x_5 & x_6 & x_7 \end{array}}^{\text{CHECK DIGITS}} \\ \boxed{\begin{array}{ccccccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}} \end{array} \longleftrightarrow \begin{aligned} x_5 &= x_1 \oplus x_2 \oplus x_3 \\ x_6 &= x_1 \oplus x_2 \oplus x_4 \\ x_7 &= x_1 \oplus x_3 \oplus x_4 \end{aligned}$$

Two representations as a *bipartite graph*



Good for decoding



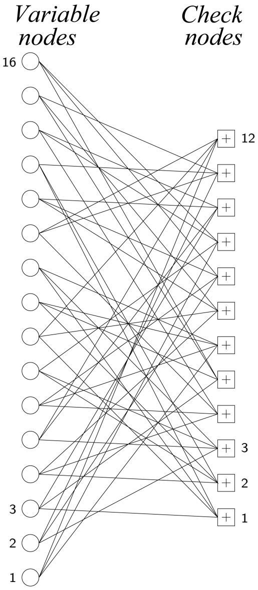
Good for encoding

These are known as *Tanner graphs*

Graph representation of LDPC matrix

Example:

$$H = \begin{bmatrix} 0&0&1&0&0&0&0&1&0&1&0&0&1&0&0&0\\ 0&0&0&1&0&0&1&0&1&0&0&0&1&0&0&0\\ 0&1&0&0&1&0&1&0&0&1&0&0&0&0&0&0\\ 0&0&0&1&0&1&0&0&0&0&1&0&0&1&0&0\\ 0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&1\\ 1&0&0&0&0&1&0&0&1&1&0&0&0&0&0&0\\ 0&0&0&1&0&0&0&1&0&0&0&1&0&0&1&0\\ 0&1&0&0&0&1&0&0&0&0&0&1&0&0&0&1\\ 1&0&0&0&0&0&1&0&0&0&0&0&0&1&0&1\\ 0&0&1&0&1&0&0&0&0&0&0&1&0&1&0&0\\ 0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0\\ 1&0&0&0&1&0&0&1&0&0&1&0&0&0&0&0 \end{bmatrix}$$



Binary code with $n = 16$, $r = 12$.
Each column has weight 3, each row has weight 4.

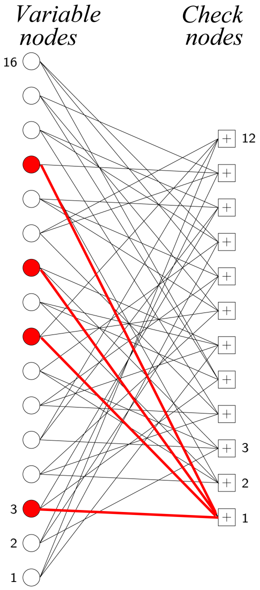
Graph representation of LDPC matrix

Example:

$$H = \begin{bmatrix}
0&0&1&0&0&0&0&1&0&1&0&0&1&0&0&0\\
0&0&0&1&0&0&1&0&1&0&0&0&1&0&0&0\\
0&1&0&0&1&0&1&0&0&1&0&0&0&0&0&0\\
0&0&0&1&0&1&0&0&0&0&1&0&0&1&0&0\\
0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&1\\
1&0&0&0&0&1&0&0&1&1&0&0&0&0&0&0\\
0&0&0&1&0&0&0&1&0&0&0&1&0&0&1&0\\
0&1&0&0&0&1&0&0&0&0&0&1&0&0&0&1\\
1&0&0&0&0&0&1&0&0&0&0&0&0&1&0&1\\
0&0&1&0&1&0&0&0&0&0&0&1&0&1&0&0\\
0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0\\
1&0&0&0&1&0&0&1&0&0&1&0&0&0&0&0
\end{bmatrix}$$

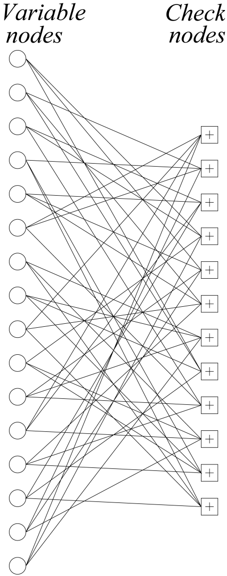
Binary code with $n = 16$, $r = 12$.
Each column has weight 3, each row has weight 4.

Edges corresponding to the first check row.



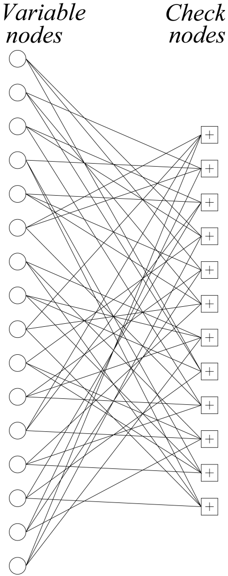
Regular and irregular graphs

- *Regular graph:* All nodes on the left have the same degree (*left-regular*) and all nodes on the right have the same degree (*right-regular*).
  Example: Gallager $\mathcal{G}(n, j, k)$ codes.



Regular and irregular graphs

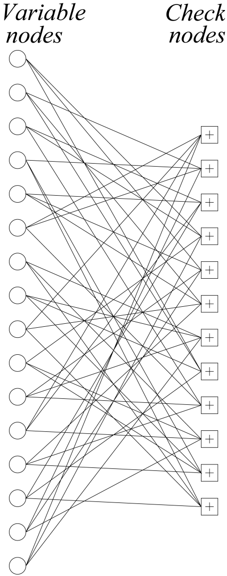
- *Regular graph:* All nodes on the left have the same degree (*left-regular*) and all nodes on the right have the same degree (*right-regular*).
  Example: Gallager $\mathcal{G}(n, j, k)$ codes.
- In general, *irregular* graphs give better performance than regular ones. Some of the best LDPC codes are based on choosing the *distribution of degrees* of the nodes in a clever way.
  Example: MacKay (1999). Random matrix with columns of fixed weight $t \geq 3$ (*right-regular*) and row weight close to uniform within a certain tolerance.



Regular and irregular graphs

- *Regular graph:* All nodes on the left have the same degree (*left-regular*) and all nodes on the right have the same degree (*right-regular*).
  Example: Gallager $\mathcal{G}(n, j, k)$ codes.
- In general, *irregular* graphs give better performance than regular ones. Some of the best LDPC codes are based on choosing the *distribution of degrees* of the nodes in a clever way.
  Example: MacKay (1999). Random matrix with columns of fixed weight $t \geq 3$ (*right-regular*) and row weight close to uniform within a certain tolerance.
- Assume d_v is the average degree of variable nodes, and d_c is the average degree of check nodes. Then

$$nd_v = rd_c \Longrightarrow R \geq 1 - \frac{r}{n} = 1 - \frac{d_v}{d_c}.$$



Iterative Decoding: Bit Flipping (Hard Decision for BSC)

The *bit flipping* scheme is the first of two *iterative algorithms* in Gallager's original paper.

Given: an LDPC matrix H, a limit K on the number of iterations, a *threshold function* $T(k,j)$, $0 \le k < K$, $1 \le j \le n$.

- **Input:** received word $\mathbf{y} = [y_1, y_2, \ldots, y_n]$,
- **Output:** estimated sent codeword $\hat{\mathbf{c}} = [\hat{c}_1, \hat{c}_2, \ldots, \hat{c}_n]$.
  1. Initialization: set $\hat{\mathbf{c}} = \mathbf{y}$, iteration counter $k = 0$.
  2. Compute check digits $\mathbf{s} = [s_1, s_2, \ldots, s_r] = \hat{\mathbf{c}} H^T$.
  3. If $\mathbf{s} = \mathbf{0}$, return $\hat{\mathbf{c}}$ and STOP.
  4. For each code coordinate j, let B_j be the number of *unsatisfied* checks $\hat{c}_j$ participates in.
  5. For each code coordinate j, if $B_j \ge T(k,j)$, *flip* $\hat{c}_j$
  6. Set $k = k + 1$. If $k < K$, go to Step 2.
     Else, return FAIL.

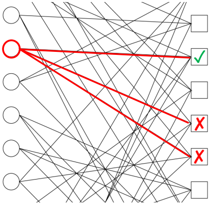
Example threshold function: $T(k,j) = \max_{j'} B_{j'}$ at iteration k.

Iterative Decoding: Bit Flipping (Hard Decision for BSC)

The *bit flipping* scheme is the first of two *iterative algorithms* in Gallager's original paper.

Given: an LDPC matrix H, a limit K on the number of iterations, a *threshold function* $T(k, j)$, $0 \le k < K$, $1 \le j \le n$.

- **Input:** received word $\mathbf{y} = [y_1, y_2, \ldots, y_n]$,
- **Output:** estimated sent codeword $\hat{\mathbf{c}} = [\hat{c}_1, \hat{c}_2, \ldots, \hat{c}_n]$.



① Initialization: set $\hat{\mathbf{c}} = \mathbf{y}$, iteration counter $k = 0$.
② Compute check digits $\mathbf{s} = [s_1, s_2, \ldots, s_r] = \hat{\mathbf{c}}H^T$.
③ If $\mathbf{s} = \mathbf{0}$, return $\hat{\mathbf{c}}$ and STOP.
④ For each code coordinate j, let B_j be the number of *unsatisfied* checks $\hat{c}_j$ participates in.
⑤ For each code coordinate j, if $B_j \ge T(k, j)$, *flip* $\hat{c}_j$
⑥ Set $k = k + 1$. If $k < K$, go to Step 2.
Else, return FAIL.

Example threshold function: $T(k, j) = \max_{j'} B_{j'}$ at iteration k.

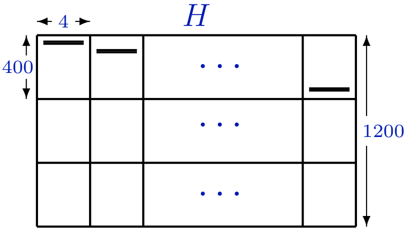
Bit flipping iteration decoding–toy example

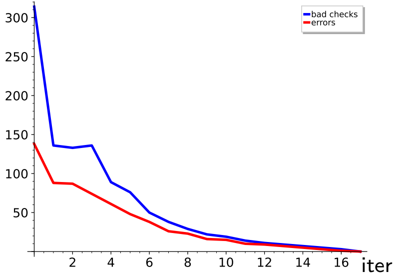
▸ Bit flipping iteration example

Bit Flipping–example with Gallager code

Example: H is a low-density matrix in $\mathcal{G}(1600, 3, 4)$, with $n = 1600$, $r = 1200$. Using $T(k, j) = \max_{j'} B_{j'}$.
Decoding a pattern of 138 binary errors.

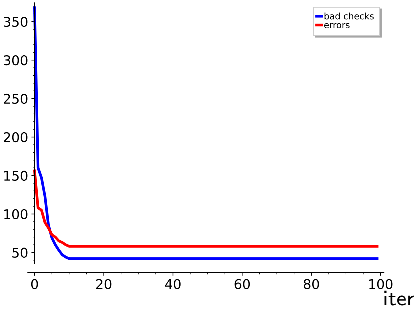
iter	bad checks	errors	$T(k,j)$	flips
0	314	138	3	78
1	136	88	3	1
2	133	87	2	95
3	136	74	3	23
4	89	61	2	47
5	76	48	3	10
6	50	38	2	24
7	38	26	3	3
8	29	23	2	9
9	22	16	3	1
10	19	15	2	5
11	14	10	3	1
12	11	9	2	2
13	9	7	2	2
14	7	5	2	2
15	5	3	2	2
16	3	1	3	1
17	0	0	0	0



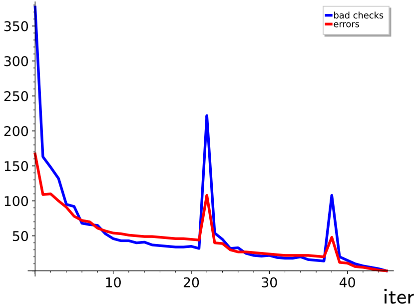


Bit Flipping–example with Gallager code

Example: Same H as before.



Decoding a pattern of 156 errors (fail).

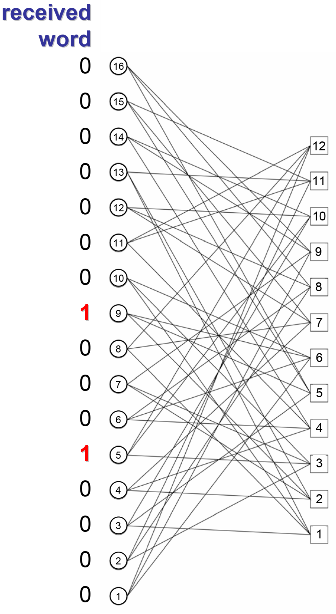


Decoding a pattern of 167 errors (ok).

Iterative Decoding: Message Passing (Hard Decision)

The *message passing* point of view

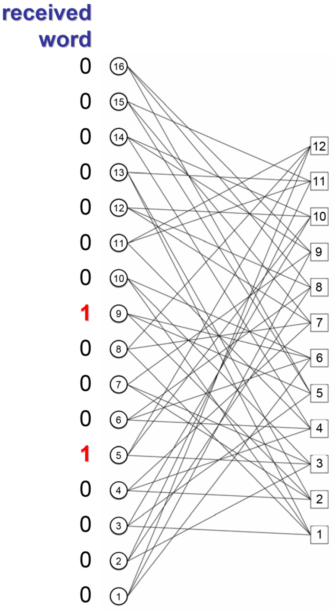
- Decoding algorithm based on *rounds of message passing* between nodes
- *Variable nodes* pass messages to *check nodes*
- *Check nodes* pass messages to *variable nodes*
- Each message is a binary symbol
- Initially, variable nodes store the received symbols



Iterative Decoding: Message Passing (Hard Decision)

The *message passing* point of view

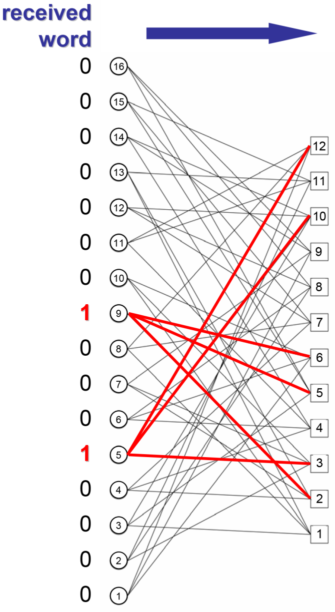
- Decoding algorithm based on *rounds of message passing* between nodes
- *Variable nodes* pass messages to *check nodes*
- *Check nodes* pass messages to *variable nodes*
- Each message is a binary symbol
- Initially, variable nodes store the received symbols
- **Round 0:** Variable nodes pass the received symbols to adjacent check nodes



Iterative Decoding: Message Passing (Hard Decision)

The *message passing* point of view

- Decoding algorithm based on *rounds of message passing* between nodes
- *Variable nodes* pass messages to *check nodes*
- *Check nodes* pass messages to *variable nodes*
- Each message is a binary symbol
- Initially, variable nodes store the received symbols
- **Round 0:** Variable nodes pass the received symbols to adjacent check nodes



Message passing—regular round

Regular round, first half:

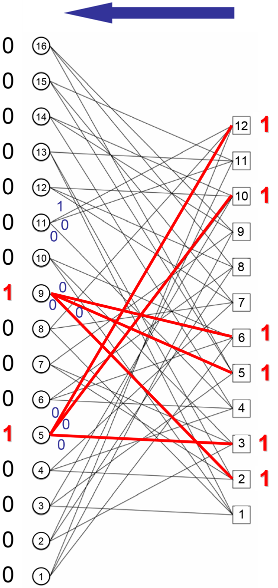
Check nodes to variable nodes

Every check node sends a message to each of its adjacent variable nodes. If the variable nodes adjacent to check node c_i are $v_1, v_2, \ldots, v_\ell$, then the message sent from c_i to v_j is

$$\mu_{i,j} = \sum_{k \in \{1,2,\ldots,\ell\} \setminus \{j\}} m_{k,i} \bmod 2$$

where $m_{k,i}$ is the message sent in the previous round from v_k to c_i.

c_i is telling v_j: "According to my other neighbors, this is the bit value you should have in order to satisfy the check."



Message passing—regular round

Regular round, second half:

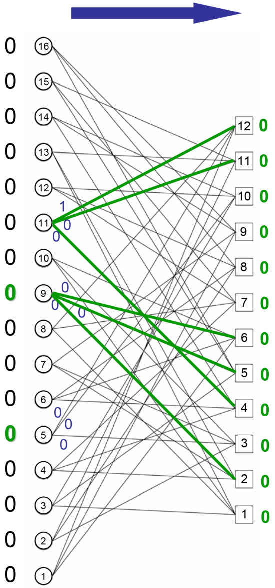
Variable nodes to check nodes (Variant 1)

Every variable node sends a message to each of its adjacent check nodes. If the check nodes adjacent to variable node v_j are $c_1, c_2, \ldots, c_d$, then the message sent from v_j to c_i is

$$m_{ji} = \begin{cases} b & \text{if } \mu_{1,j}=\mu_{2,j}=\ldots\mu_{i-1,j} = \\ & \quad \mu_{i+1,j}\ldots\mu_{d,j}=b \\ y_j & \text{otherwise}\ , \ 1 \le j \le n\,. \end{cases}$$

where $\mu_{s,j}$ is the message sent in the previous round from c_s to v_j.

Either “my *other* neighbors agree this should be my value” or “not enough evidence to change my belief.”



Message passing—regular round

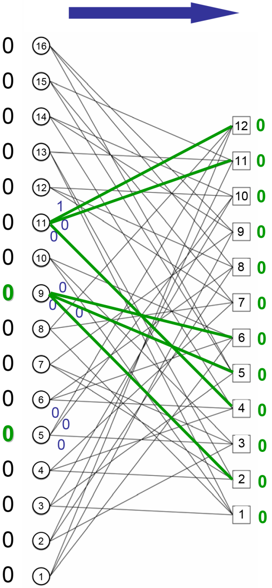
Regular round, second half:

Variable nodes to check nodes (Variant 2)

Instead of requiring $\mu_{1,j}{=}\mu_{2,j}{=}\ldots\mu_{i-1,j}=\mu_{i+1,j}\ldots\mu_{t,j}{=}b$ (all neighbors except possibly c_i sent the same information), a threshold τ (that can vary at each round) is used such that the message sent is b if at least τ neighbors sent the same information. Variant 1 corresponds to $\tau = d-1$ ($d =$ node degree).

Another variant

Variable node takes majority vote, accepts value if there is a winner, or keeps its value otherwise.

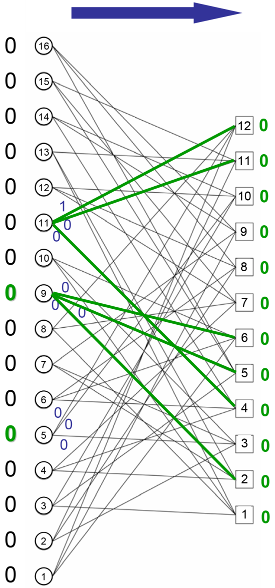


Message passing—regular round

Stopping condition:

All check equations are satisfied

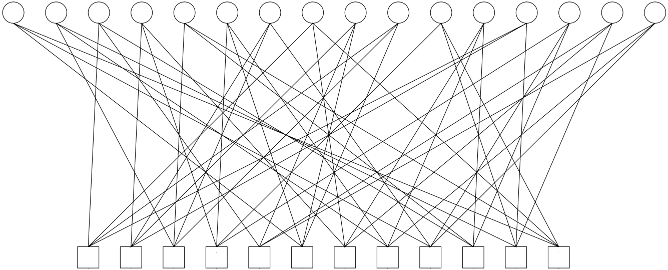
- Subtle point:
  - Even though all checks are satisfied, there might not be a consensus on what the variable values should be.
  - This is because v_j may be telling different c_i's different things. So, v_j may be getting different advice from different c_i's.
  - With n large enough, this will be rare, and in any case a majority decision may be taken, and checked.



Message passing–example

$$H = \begin{bmatrix} 0&0&1&0&0&0&0&1&0&1&0&0&1&0&0&0 \\ 0&0&0&1&0&0&1&0&1&0&0&0&1&0&0&0 \\ 0&1&0&0&1&0&1&0&0&1&0&0&0&0&0&0 \\ 0&0&0&1&0&1&0&0&0&0&1&0&0&1&0&0 \\ 0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&1 \\ 1&0&0&0&0&1&0&0&1&1&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&1&0&0&0&1&0&0&1&0 \\ 0&1&0&0&0&1&0&0&0&0&0&1&0&0&0&1 \\ 1&0&0&0&0&0&1&0&0&0&0&0&0&1&0&1 \\ 0&0&1&0&1&0&0&0&0&0&0&1&0&1&0&0 \\ 0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0 \\ 1&0&0&0&1&0&0&1&0&0&1&0&0&0&0&0 \end{bmatrix}$$

Code $\mathcal{C}$ is $[16, 4, 4]$.



Message passing–example

variable nodes

```
 j :   checks
 0 : [5, 8, 11]
 1 : [2, 7, 10]
 2 : [0, 4, 9]
 3 : [1, 3, 6]
 4 : [2, 9, 11]
 5 : [3, 5, 7]
 6 : [1, 2, 8]
 7 : [0, 6, 11]
 8 : [1, 4, 5]
 9 : [0, 2, 5]
10 : [3, 10, 11]
11 : [6, 7, 9]
12 : [0, 1, 10]
13 : [3, 8, 9]
14 : [4, 6, 10]
15 : [4, 7, 8]
```

check nodes

```
 i :   variables
 0 : [2, 7, 9, 12]
 1 : [3, 6, 8, 12]
 2 : [1, 4, 6, 9]
 3 : [3, 5, 10, 13]
 4 : [2, 8, 14, 15]
 5 : [0, 5, 8, 9]
 6 : [3, 7, 11, 14]
 7 : [1, 5, 11, 15]
 8 : [0, 6, 13, 15]
 9 : [2, 4, 11, 13]
10 : [1, 10, 12, 14]
11 : [0, 4, 7, 10]
```

Message passing–example

variable nodes

```
y    j :    checks
0    0 : [5, 8, 11]
0    1 : [2, 7, 10]
0    2 : [0, 4, 9]
0    3 : [1, 3, 6]
1    4 : [2, 9, 11]
0    5 : [3, 5, 7]
0    6 : [1, 2, 8]
0    7 : [0, 6, 11]
1    8 : [1, 4, 5]
0    9 : [0, 2, 5]
0   10 : [3, 10, 11]
0   11 : [6, 7, 9]
0   12 : [0, 1, 10]
0   13 : [3, 8, 9]
0   14 : [4, 6, 10]
0   15 : [4, 7, 8]
```

check nodes

```
     i :   variables
0    0 : [2, 7, 9, 12]
0    1 : [3, 6, 8, 12]
0    2 : [1, 4, 6, 9]
0    3 : [3, 5, 10, 13]
0    4 : [2, 8, 14, 15]
0    5 : [0, 5, 8, 9]
0    6 : [3, 7, 11, 14]
0    7 : [1, 5, 11, 15]
0    8 : [0, 6, 13, 15]
0    9 : [2, 4, 11, 13]
0   10 : [1, 10, 12, 14]
0   11 : [0, 4, 7, 10]
```

Message passing–example

Round 0

variable nodes

```
y   j :   checks
0   0 : [5, 8, 11]
0   1 : [2, 7, 10]
0   2 : [0, 4, 9]
0   3 : [1, 3, 6]
1   4 : [2, 9, 11]
0   5 : [3, 5, 7]
0   6 : [1, 2, 8]
0   7 : [0, 6, 11]
1   8 : [1, 4, 5]
0   9 : [0, 2, 5]
0  10 : [3, 10, 11]
0  11 : [6, 7, 9]
0  12 : [0, 1, 10]
0  13 : [3, 8, 9]
0  14 : [4, 6, 10]
0  15 : [4, 7, 8]
```

check nodes

```
    i :   variables
0   0 : [2, 7, 9, 12]
1   1 : [3, 6, 8, 12]
1   2 : [1, 4, 6, 9]
0   3 : [3, 5, 10, 13]
1   4 : [2, 8, 14, 15]
1   5 : [0, 5, 8, 9]
0   6 : [3, 7, 11, 14]
0   7 : [1, 5, 11, 15]
0   8 : [0, 6, 13, 15]
1   9 : [2, 4, 11, 13]
0  10 : [1, 10, 12, 14]
1  11 : [0, 4, 7, 10]
```

Message passing–example

Round 1a

variable nodes

```
y    j :    checks
0    0 : [5, 8, 11]
0    1 : [2, 7, 10]
0    2 : [0, 4, 9]
0    3 : [1, 3, 6]
1    4 : [2, 9, 11]
0    5 : [3, 5, 7]
0    6 : [1, 2, 8]
0    7 : [0, 6, 11]
1    8 : [1, 4, 5]
0    9 : [0, 2, 5]
0   10 : [3, 10, 11]
0   11 : [6, 7, 9]
0   12 : [0, 1, 10]
0   13 : [3, 8, 9]
0   14 : [4, 6, 10]
0   15 : [4, 7, 8]
```

check nodes

```
     i :    variables
0    0 : [2, 7, 9, 12]
1    1 : [3, 6, 8, 12]
1    2 : [1, 4, 6, 9]
0    3 : [3, 5, 10, 13]
1    4 : [2, 8, 14, 15]
1    5 : [0, 5, 8, 9]
0    6 : [3, 7, 11, 14]
0    7 : [1, 5, 11, 15]
0    8 : [0, 6, 13, 15]
1    9 : [2, 4, 11, 13]
0   10 : [1, 10, 12, 14]
1   11 : [0, 4, 7, 10]
```

Message passing–example

Round 1b

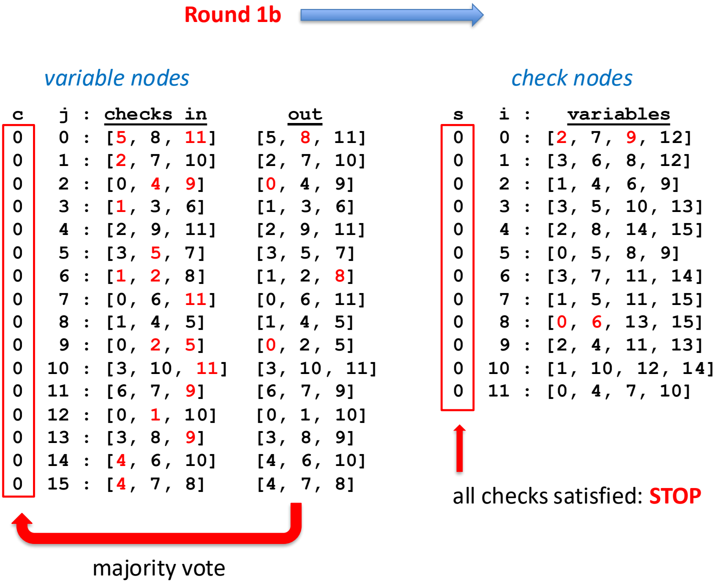
variable nodes

```
y   c    j : checks in      out
0   0    0 : [5, 8, 11]    [5, 8, 11]
0   0    1 : [2, 7, 10]    [2, 7, 10]
0   0    2 : [0, 4, 9]     [0, 4, 9]
0   0    3 : [1, 3, 6]     [1, 3, 6]
1   0    4 : [2, 9, 11]    [2, 9, 11]
0   0    5 : [3, 5, 7]     [3, 5, 7]
0   0    6 : [1, 2, 8]     [1, 2, 8]
0   0    7 : [0, 6, 11]    [0, 6, 11]
1   0    8 : [1, 4, 5]     [1, 4, 5]
0   0    9 : [0, 2, 5]     [0, 2, 5]
0   0   10 : [3, 10, 11]   [3, 10, 11]
0   0   11 : [6, 7, 9]     [6, 7, 9]
0   0   12 : [0, 1, 10]    [0, 1, 10]
0   0   13 : [3, 8, 9]     [3, 8, 9]
0   0   14 : [4, 6, 10]    [4, 6, 10]
0   0   15 : [4, 7, 8]     [4, 7, 8]
```

check nodes

```
     i :   variables
0    0 : [2, 7, 9, 12]
0    1 : [3, 6, 8, 12]
0    2 : [1, 4, 6, 9]
0    3 : [3, 5, 10, 13]
0    4 : [2, 8, 14, 15]
0    5 : [0, 5, 8, 9]
0    6 : [3, 7, 11, 14]
0    7 : [1, 5, 11, 15]
0    8 : [0, 6, 13, 15]
0    9 : [2, 4, 11, 13]
0   10 : [1, 10, 12, 14]
0   11 : [0, 4, 7, 10]
```

Message passing–example



Round 1b

variable nodes

c	j	checks in	out
0	0	[5, 8, 11]	[5, 8, 11]
0	1	[2, 7, 10]	[2, 7, 10]
0	2	[0, 4, 9]	[0, 4, 9]
0	3	[1, 3, 6]	[1, 3, 6]
0	4	[2, 9, 11]	[2, 9, 11]
0	5	[3, 5, 7]	[3, 5, 7]
0	6	[1, 2, 8]	[1, 2, 8]
0	7	[0, 6, 11]	[0, 6, 11]
0	8	[1, 4, 5]	[1, 4, 5]
0	9	[0, 2, 5]	[0, 2, 5]
0	10	[3, 10, 11]	[3, 10, 11]
0	11	[6, 7, 9]	[6, 7, 9]
0	12	[0, 1, 10]	[0, 1, 10]
0	13	[3, 8, 9]	[3, 8, 9]
0	14	[4, 6, 10]	[4, 6, 10]
0	15	[4, 7, 8]	[4, 7, 8]

majority vote

check nodes

s	i	variables
0	0	[2, 7, 9, 12]
0	1	[3, 6, 8, 12]
0	2	[1, 4, 6, 9]
0	3	[3, 5, 10, 13]
0	4	[2, 8, 14, 15]
0	5	[0, 5, 8, 9]
0	6	[3, 7, 11, 14]
0	7	[1, 5, 11, 15]
0	8	[0, 6, 13, 15]
0	9	[2, 4, 11, 13]
0	10	[1, 10, 12, 14]
0	11	[0, 4, 7, 10]

all checks satisfied: **STOP**

Message passing–example 2

variable nodes

```
y   j :   checks
0   0 : [5, 8, 11]
0   1 : [2, 7, 10]
0   2 : [0, 4, 9]
0   3 : [1, 3, 6]
0   4 : [2, 9, 11]
0   5 : [3, 5, 7]
0   6 : [1, 2, 8]
1   7 : [0, 6, 11]
0   8 : [1, 4, 5]
0   9 : [0, 2, 5]
0  10 : [3, 10, 11]
0  11 : [6, 7, 9]
0  12 : [0, 1, 10]
0  13 : [3, 8, 9]
0  14 : [4, 6, 10]
1  15 : [4, 7, 8]
```

check nodes

```
    i :   variables
0   0 : [2, 7, 9, 12]
0   1 : [3, 6, 8, 12]
0   2 : [1, 4, 6, 9]
0   3 : [3, 5, 10, 13]
0   4 : [2, 8, 14, 15]
0   5 : [0, 5, 8, 9]
0   6 : [3, 7, 11, 14]
0   7 : [1, 5, 11, 15]
0   8 : [0, 6, 13, 15]
0   9 : [2, 4, 11, 13]
0  10 : [1, 10, 12, 14]
0  11 : [0, 4, 7, 10]
```

Message passing–example 2

Round 0

variable nodes

```
y    j :   checks
0    0 : [5, 8, 11]
0    1 : [2, 7, 10]
0    2 : [0, 4, 9]
0    3 : [1, 3, 6]
0    4 : [2, 9, 11]
0    5 : [3, 5, 7]
0    6 : [1, 2, 8]
1    7 : [0, 6, 11]
0    8 : [1, 4, 5]
0    9 : [0, 2, 5]
0   10 : [3, 10, 11]
0   11 : [6, 7, 9]
0   12 : [0, 1, 10]
0   13 : [3, 8, 9]
0   14 : [4, 6, 10]
1   15 : [4, 7, 8]
```

check nodes

```
     i :   variables
1    0 : [2, 7, 9, 12]
0    1 : [3, 6, 8, 12]
0    2 : [1, 4, 6, 9]
0    3 : [3, 5, 10, 13]
1    4 : [2, 8, 14, 15]
0    5 : [0, 5, 8, 9]
1    6 : [3, 7, 11, 14]
1    7 : [1, 5, 11, 15]
1    8 : [0, 6, 13, 15]
0    9 : [2, 4, 11, 13]
0   10 : [1, 10, 12, 14]
1   11 : [0, 4, 7, 10]
```

Message passing–example 2

Round 1a ←

variable nodes

```
y    j :   checks
0    0 : [5, 8, 11]
0    1 : [2, 7, 10]
0    2 : [0, 4, 9]
0    3 : [1, 3, 6]
0    4 : [2, 9, 11]
0    5 : [3, 5, 7]
0    6 : [1, 2, 8]
1    7 : [0, 6, 11]
0    8 : [1, 4, 5]
0    9 : [0, 2, 5]
0   10 : [3, 10, 11]
0   11 : [6, 7, 9]
0   12 : [0, 1, 10]
0   13 : [3, 8, 9]
0   14 : [4, 6, 10]
1   15 : [4, 7, 8]
```

check nodes

```
     i :   variables
1    0 : [2, 7, 9, 12]
0    1 : [3, 6, 8, 12]
0    2 : [1, 4, 6, 9]
0    3 : [3, 5, 10, 13]
1    4 : [2, 8, 14, 15]
0    5 : [0, 5, 8, 9]
1    6 : [3, 7, 11, 14]
1    7 : [1, 5, 11, 15]
1    8 : [0, 6, 13, 15]
0    9 : [2, 4, 11, 13]
0   10 : [1, 10, 12, 14]
1   11 : [0, 4, 7, 10]
```

Message passing–example 2

Round 1b

variable nodes

```
y    j : checks in       out
0    0 : [5, 8, 11]     [5, 8, 11]
0    1 : [2, 7, 10]     [2, 7, 10]
0    2 : [0, 4, 9]      [0, 4, 9]
0    3 : [1, 3, 6]      [1, 3, 6]
0    4 : [2, 9, 11]     [2, 9, 11]
0    5 : [3, 5, 7]      [3, 5, 7]
0    6 : [1, 2, 8]      [1, 2, 8]
1    7 : [0, 6, 11]     [0, 6, 11]
0    8 : [1, 4, 5]      [1, 4, 5]
0    9 : [0, 2, 5]      [0, 2, 5]
0   10 : [3, 10, 11]    [3, 10, 11]
0   11 : [6, 7, 9]      [6, 7, 9]
0   12 : [0, 1, 10]     [0, 1, 10]
0   13 : [3, 8, 9]      [3, 8, 9]
0   14 : [4, 6, 10]     [4, 6, 10]
1   15 : [4, 7, 8]      [4, 7, 8]
```

check nodes

```
     i :   variables
0    0 : [2, 7, 9, 12]
0    1 : [3, 6, 8, 12]
0    2 : [1, 4, 6, 9]
0    3 : [3, 5, 10, 13]
0    4 : [2, 8, 14, 15]
1    5 : [0, 5, 8, 9]
0    6 : [3, 7, 11, 14]
0    7 : [1, 5, 11, 15]
0    8 : [0, 6, 13, 15]
0    9 : [2, 4, 11, 13]
1   10 : [1, 10, 12, 14]
0   11 : [0, 4, 7, 10]
```

Message passing–example 2

Round 2a

variable nodes

```
y    j : checks in     out
0    0 : [5, 8, 11]    [5, 8, 11]
0    1 : [2, 7, 10]    [2, 7, 10]
0    2 : [0, 4, 9]     [0, 4, 9]
0    3 : [1, 3, 6]     [1, 3, 6]
0    4 : [2, 9, 11]    [2, 9, 11]
0    5 : [3, 5, 7]     [3, 5, 7]
0    6 : [1, 2, 8]     [1, 2, 8]
1    7 : [0, 6, 11]    [0, 6, 11]
0    8 : [1, 4, 5]     [1, 4, 5]
0    9 : [0, 2, 5]     [0, 2, 5]
0   10 : [3, 10, 11]   [3, 10, 11]
0   11 : [6, 7, 9]     [6, 7, 9]
0   12 : [0, 1, 10]    [0, 1, 10]
0   13 : [3, 8, 9]     [3, 8, 9]
0   14 : [4, 6, 10]    [4, 6, 10]
1   15 : [4, 7, 8]     [4, 7, 8]
```

check nodes

```
     i :   variables
0    0 : [2, 7, 9, 12]
0    1 : [3, 6, 8, 12]
0    2 : [1, 4, 6, 9]
0    3 : [3, 5, 10, 13]
0    4 : [2, 8, 14, 15]
1    5 : [0, 5, 8, 9]
0    6 : [3, 7, 11, 14]
0    7 : [1, 5, 11, 15]
0    8 : [0, 6, 13, 15]
0    9 : [2, 4, 11, 13]
1   10 : [1, 10, 12, 14]
0   11 : [0, 4, 7, 10]
```

Message passing–example 2

Round 2b →

variable nodes

```
y    j : checks in     out
0    0 : [5, 8, 11]    [5, 8, 11]
0    1 : [2, 7, 10]    [2, 7, 10]
0    2 : [0, 4, 9]     [0, 4, 9]
0    3 : [1, 3, 6]     [1, 3, 6]
0    4 : [2, 9, 11]    [2, 9, 11]
0    5 : [3, 5, 7]     [3, 5, 7]
0    6 : [1, 2, 8]     [1, 2, 8]
1    7 : [0, 6, 11]    [0, 6, 11]
0    8 : [1, 4, 5]     [1, 4, 5]
0    9 : [0, 2, 5]     [0, 2, 5]
0   10 : [3, 10, 11]   [3, 10, 11]
0   11 : [6, 7, 9]     [6, 7, 9]
0   12 : [0, 1, 10]    [0, 1, 10]
0   13 : [3, 8, 9]     [3, 8, 9]
0   14 : [4, 6, 10]    [4, 6, 10]
1   15 : [4, 7, 8]     [4, 7, 8]
```

check nodes

```
     i :   variables
0    0 : [2, 7, 9, 12]
0    1 : [3, 6, 8, 12]
0    2 : [1, 4, 6, 9]
0    3 : [3, 5, 10, 13]
0    4 : [2, 8, 14, 15]
0    5 : [0, 5, 8, 9]
0    6 : [3, 7, 11, 14]
0    7 : [1, 5, 11, 15]
0    8 : [0, 6, 13, 15]
0    9 : [2, 4, 11, 13]
0   10 : [1, 10, 12, 14]
0   11 : [0, 4, 7, 10]
```

Message passing–example 2

Round 2b

variable nodes

```
c     j : checks in       out
0     0 : [5, 8, 11]     [5, 8, 11]
0     1 : [2, 7, 10]     [2, 7, 10]
0     2 : [0, 4, 9]      [0, 4, 9]
0     3 : [1, 3, 6]      [1, 3, 6]
0     4 : [2, 9, 11]     [2, 9, 11]
0     5 : [3, 5, 7]      [3, 5, 7]
0     6 : [1, 2, 8]      [1, 2, 8]
0     7 : [0, 6, 11]     [0, 6, 11]
0     8 : [1, 4, 5]      [1, 4, 5]
0     9 : [0, 2, 5]      [0, 2, 5]
0    10 : [3, 10, 11]    [3, 10, 11]
0    11 : [6, 7, 9]      [6, 7, 9]
0    12 : [0, 1, 10]     [0, 1, 10]
0    13 : [3, 8, 9]      [3, 8, 9]
0    14 : [4, 6, 10]     [4, 6, 10]
0    15 : [4, 7, 8]      [4, 7, 8]
```

unanimity

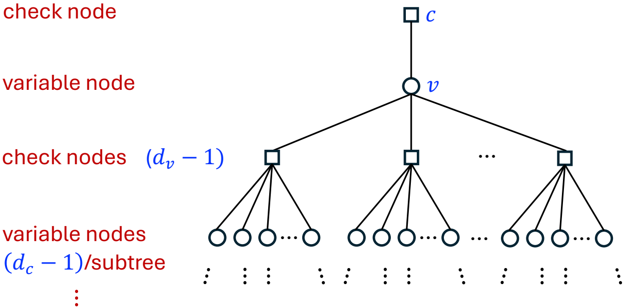
check nodes

```
      i :   variables
0     0 : [2, 7, 9, 12]
0     1 : [3, 6, 8, 12]
0     2 : [1, 4, 6, 9]
0     3 : [3, 5, 10, 13]
0     4 : [2, 8, 14, 15]
0     5 : [0, 5, 8, 9]
0     6 : [3, 7, 11, 14]
0     7 : [1, 5, 11, 15]
0     8 : [0, 6, 13, 15]
0     9 : [2, 4, 11, 13]
0    10 : [1, 10, 12, 14]
0    11 : [0, 4, 7, 10]
```

all checks satisfied: **STOP**

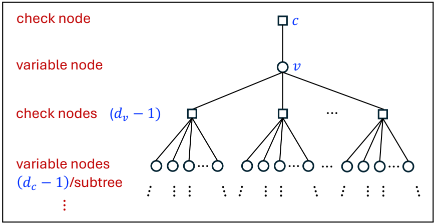
Why (when) does iterative decoding work? [Gallager'62]

Local neighborhood "tree" of an edge (v, c).



- Assume a BSC with $P(\text{bit error}) = p$.
- Let p_i be the probability of a message m_{vc} being *wrong* at iteration i, with $p_0 = p$.
- We derive an expression for p_{i+1}.

Why (when) does iterative decoding work?



- Consider a neighbor c' of v, $c' \neq c$. Check node c' sends the *correct* value to v if an *even* number of neighbors of c' (excluding v) sent c' the wrong value. So,

$$P(\mu_{c'v} \text{ good}) = \sum_{\substack{\ell \text{ even} \\ 0 \leq \ell < d_c}} \binom{d_c - 1}{\ell} p_i^{\ell} (1 - p_i)^{d_c - 1 - \ell} = \frac{1 + (1 - 2p_i)^{d_c - 1}}{2} .$$

  - Hence, the probability that v was (initially) received in error, and sent *incorrectly* in round $i{+}1$ is

$$p_0 \left(1 - \left[\frac{1 + (1 - 2p_i)^{d_c - 1}}{2} \right]^{d_v - 1} \right) .$$

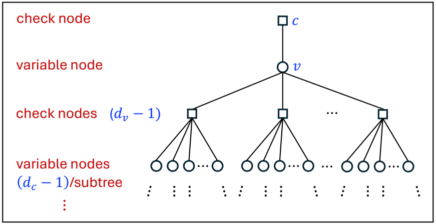
  - Similarly, the probability that v was received correctly, but sent *incorrectly* in round $i{+}1$ is

$$(1 - p_0) \left[\frac{1 - (1 - 2p_i)^{d_c - 1}}{2} \right]^{d_v - 1} .$$

Why (when) does iterative decoding work?

- We get the following recursion

$$p_{i+1} = p_0 - p_0 \left[\frac{1+(1-2p_i)^{d_c-1}}{2}\right]^{d_v-1} + (1-p_0)\left[\frac{1-(1-2p_i)^{d_c-1}}{2}\right]^{d_v-1}.$$



check node

variable node

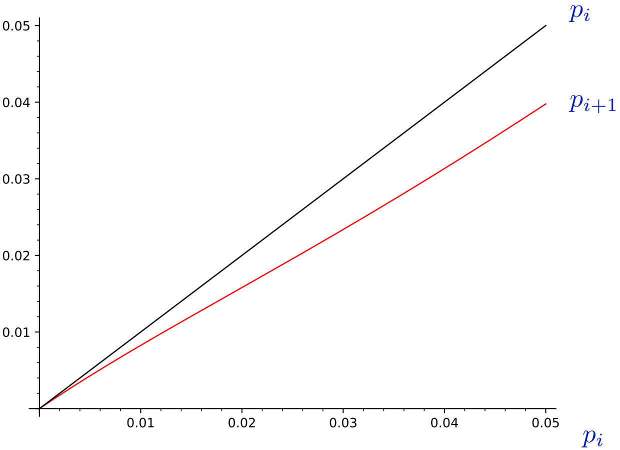
check nodes $(d_v - 1)$

variable nodes $(d_c - 1)$/subtree

  - If p_0 is such that $p_{i+1} < p_i$, then the bit error probability *strictly decreases* at each iteration. In fact, a more detailed analysis of the recursion proves that there exists a threshold p_0^* such that $p_i \xrightarrow{i} 0$ for all $p_0 < p_0^*$.
  - *But there are strong conditional independence assumptions in this calculation!*
  - These assumptions hold if the neighborhood *is indeed a tree* over the number of iterations run.
  - We need bipartite graphs of *large girth* (*girth*: length of smallest loop in the graph). For that, we need *large* n.

Why (when) does iterative decoding work?

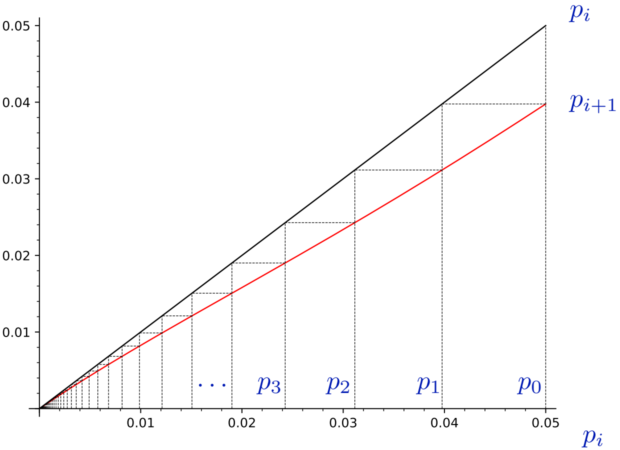
Example: $d_c = 11$, $d_v = 7$: p_{i+1} vs p_i for $p_0 = 0.05$.



(We have $p_0^* \approx 0.0556$ in this case.)

Why (when) does iterative decoding work?

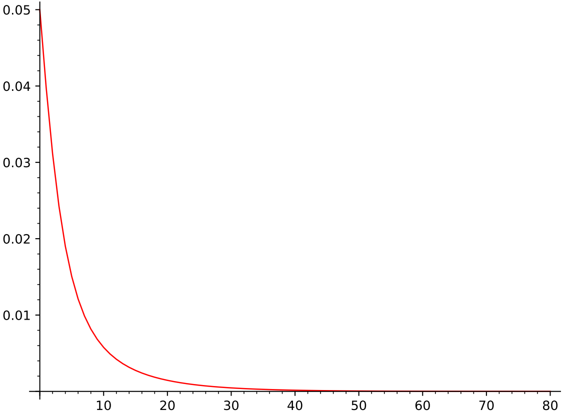
Example: $d_c = 11$, $d_v = 7$: p_{i+1} vs p_i for $p_0 = 0.05$.



(We have $p_0^* \approx 0.0556$ in this case.)

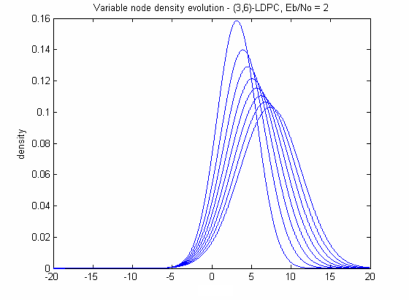
Why (when) does iterative decoding work?

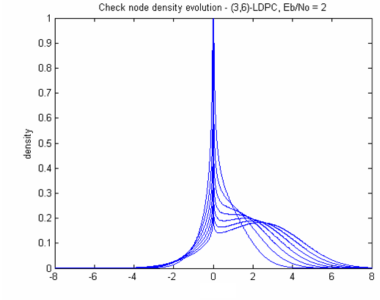
Example: $d_c = 11$, $d_v = 7$: p_i vs i for $p_0 = 0.05$.



Iterative Decoding for the BSC—Soft Decision

- In *soft decision decoding*, the received data is given in the form of a vector of *probabilities* $(p_1\, p_2\, \ldots\, p_n)$ such that $p_i = P(x_i = 1 \,|\, y_i)$.
- The goal of the iterative decoder is to improve these estimates so that eventually we can get an estimate $\hat{x}_i$ of x_i with $P(\hat{x}_i = x_i \,|\, y_1, y_2, \ldots, y_n)$ approaching one (*density evolution*).





- Density evolution calculations are very difficult in general, but explicit recursions can be set up for some interesting cases under some independence assumptions.

Belief propagation

- The tool used to iteratively improve the symbol probability estimates is *belief propagation*. This has several closely related interpretations, names, and representations, e.g. *sum-products* algorithm, *Bayesian networks*, *generalized ditributive law*, etc.
- We work with *log-likelihood ratios*

$$m_{v_i} = \log \frac{P(v_i = 0 \,|\, \hat{v}_i)}{P(v_i = 1 \,|\, \hat{v}_i)}\,.$$

In our problem, the x_i's play the role of the v_i's, and the initial estimates $\hat{v}_i$ are the received symbols y_i.

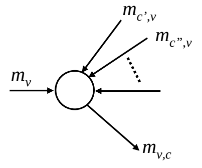
- *Messages* passed through the graph effect log-likelihood updates.

Belief propagation

- The message passed from a message node v to a check node c is the probability that v has a certain value given the observed value of that message node, and all the values communicated to v in the prior round from check nodes incident to v *other than* c.
- The message passed from c to v is the probability that v has a certain value given all the messages passed to c in the previous round from message nodes *other than* v.

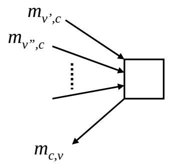
Likelihood updates

From variables to checks



$$m_{v,c}^{(\ell)} = \begin{cases} m_v & \ell = 0, \\ m_v + \sum_{c' \in C_v \setminus \{c\}} m_{c',v}^{(\ell-1)}, & \ell > 0\,. \end{cases}$$

From checks to variables



$$m_{c,v}^{(\ell)} = \log \frac{1 + \prod_{v' \in V_c \setminus \{v\}} \tanh(m_{v',c}^{(\ell)}/2)}{1 - \prod_{v' \in V_c \setminus \{v\}} \tanh(m_{v',c}^{(\ell)}/2)}$$

When does iterative decoding work?

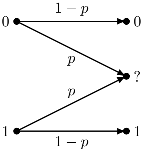
- As in the hard decision case, the soft decision belief propagation iteration relies on the conditional independence of the messages passed in the process of updating the variable nodes.
- Same *large girth* requirements for the code graph.
- Let $\mathcal{S}(e, d)$ denote the local neighborhood to depth d of edge e. Let L denote any fixed number of rounds of the iterative decoding.

Key observation

In a random bipartite graph, for any given L, $\mathcal{S}(e, 2L)$ is cycle-free with probability $\to 1$ as $n \to \infty$.

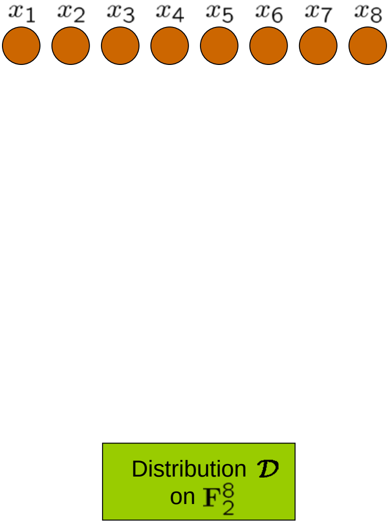
- Even for finite n, if the number of iterations is not too large (as is the case in practice), the independence assumption holds with high probability.
- There exist construction techniques that produce graphs of large girth.

LDPC codes for the binary erasure channel (BEC)



- The BEC is a good model for packet losses in networks
- A class of LDPC codes, developed in the early 2000s and often referred to as *fountain codes* (incl. Tornado, LT, and Raptor codes), targets the BEC and has many interesting features
  - Codes are *rateless*: a potentially endless stream of encoded symbols is sent by the sender. The receiver collects enough of them until it can decode (then may ask sender to stop).
  - In *multicast* situations, different receivers may read different lengths of the encoded stream.
  - The codes can be encoded and decoded in very low complexity (linear or almost linear in the length of the encoded block).
  - The codes are *random*; they are generated *on the fly* by the sender, and also by the receiver based on common randomness.
  - The best codes in the class approach the capacity of the BEC even if the channel parameter is unknown.

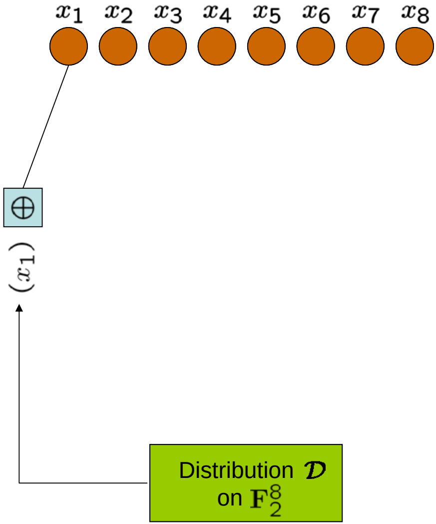
LT codes (Luby 2002)—Encoding example



Distribution $\mathcal{D}$ on $\mathbf{F}_2^8$

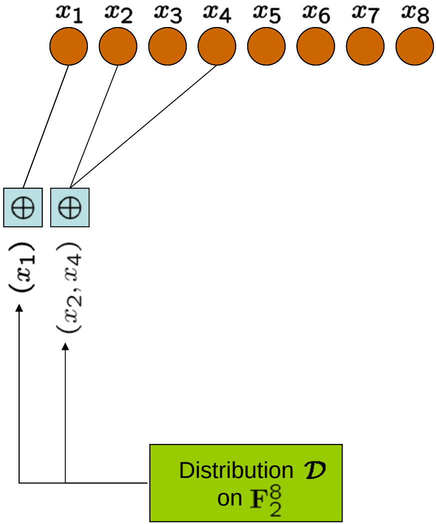
[Example: A. Shokrollahi]

LT codes (Luby 2002)—Encoding example



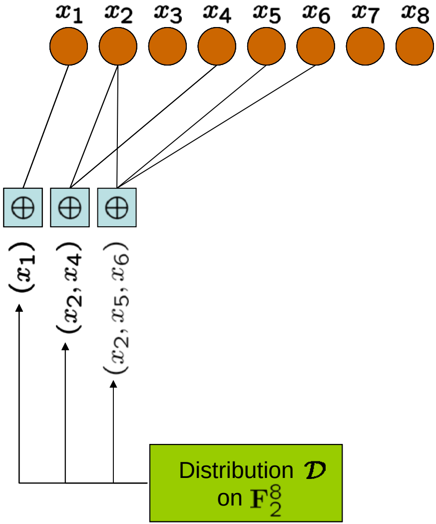
[Example: A. Shokrollahi]

LT codes (Luby 2002)—Encoding example



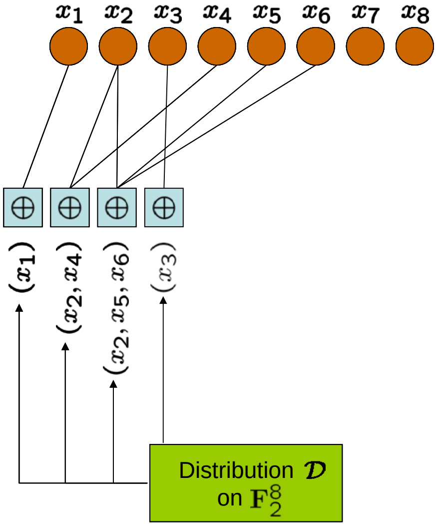
[Example: A. Shokrollahi]

LT codes (Luby 2002)—Encoding example



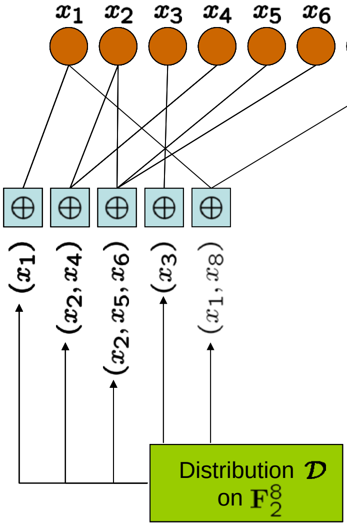
[Example: A. Shokrollahi]

LT codes (Luby 2002)—Encoding example



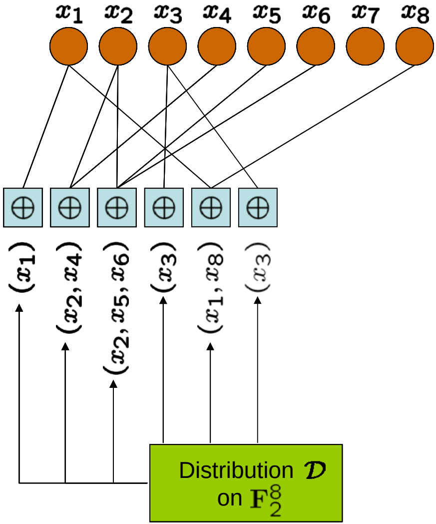
[Example: A. Shokrollahi]

LT codes (Luby 2002)—Encoding example



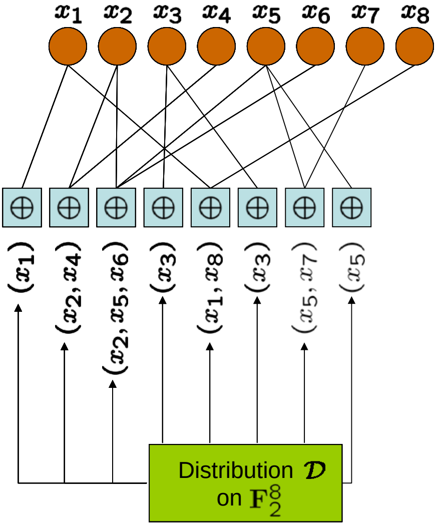
[Example: A. Shokrollahi]

LT codes (Luby 2002)—Encoding example



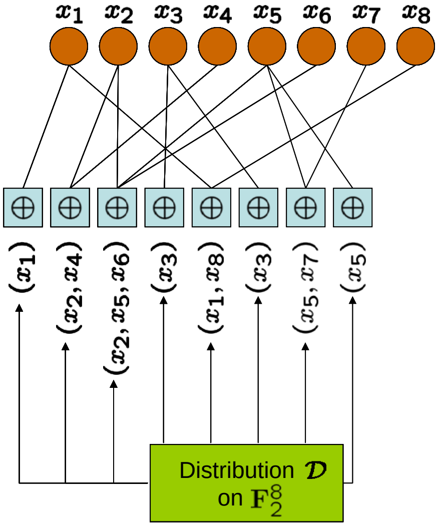
[Example: A. Shokrollahi]

LT codes (Luby 2002)—Encoding example



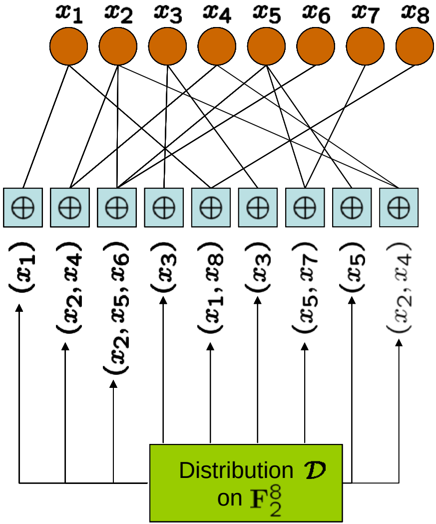
[Example: A. Shokrollahi]

LT codes (Luby 2002)—Encoding example



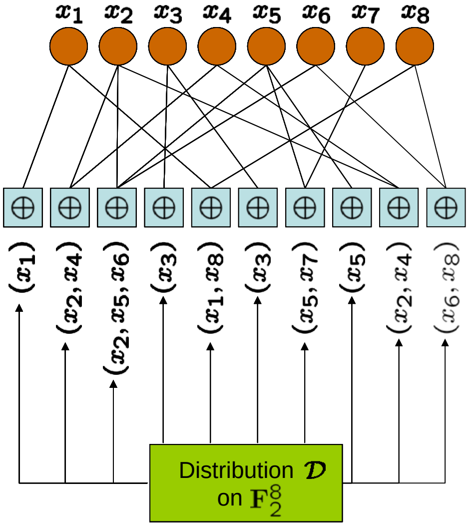
[Example: A. Shokrollahi]

LT codes (Luby 2002)—Encoding example



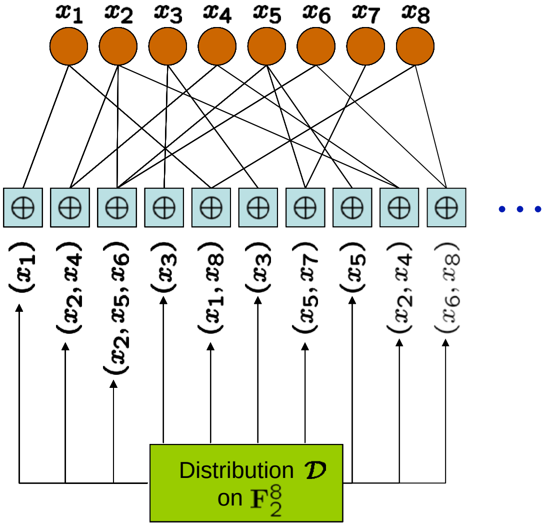
[Example: A. Shokrollahi]

LT codes (Luby 2002)—Encoding example



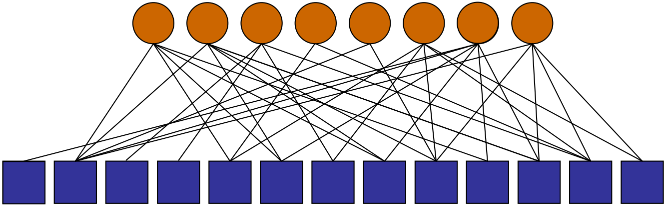
[Example: A. Shokrollahi]

LT codes (Luby 2002)—Encoding example



[Example: A. Shokrollahi]

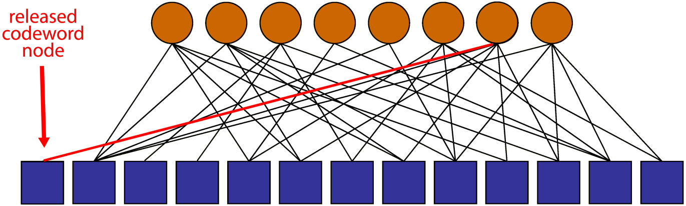
LT codes— Decoding example



1. Find codeword node c of (reduced) degree 1
2. Assign value of c to the variable node v connected to it
3. Subtract the value of v from all the codeword nodes connected to it
4. Remove all edges incident on v
5. If there are message symbols that have not been decoded, go to Step 1

[Example: A. Shokrollahi]

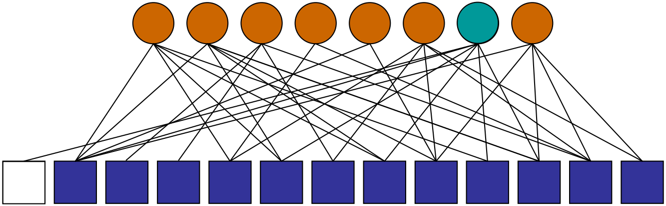
LT codes— Decoding example



1. *Find codeword node c of reduced degree 1* (*released*)
2. Assign value of c to the variable node v connected to it
3. Subtract the value of v from all the codeword nodes connected to it
4. Remove all edges incident on v
5. If there are message symbols that have not been decoded, go to Step 1

[Example: A. Shokrollahi]

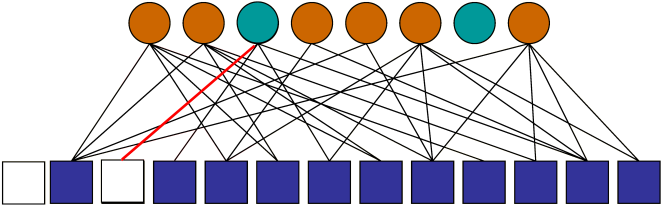
LT codes— Decoding example



1. Find codeword node c of reduced degree 1 (*released*)
2. Assign value of c to the variable node v connected to it
3. Subtract the value of v from all the codeword nodes connected to it
4. Remove all edges incident on v
5. If there are message symbols that have not been decoded, go to Step 1

[Example: A. Shokrollahi]

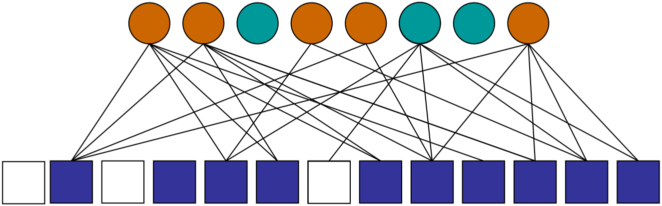
LT codes— Decoding example



1. Find codeword node c of reduced degree 1 (*released*)
2. Assign value of c to the variable node v connected to it
3. Subtract the value of v from all the codeword nodes connected to it
4. Remove all edges incident on v
5. If there are message symbols that have not been decoded, go to Step 1

[Example: A. Shokrollahi]

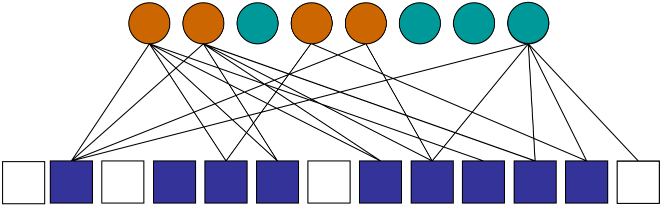
LT codes— Decoding example



1. Find codeword node c of reduced degree 1 (*released*)
2. Assign value of c to the variable node v connected to it
3. Subtract the value of v from all the codeword nodes connected to it
4. Remove all edges incident on v
5. If there are message symbols that have not been decoded, go to Step 1

[Example: A. Shokrollahi]

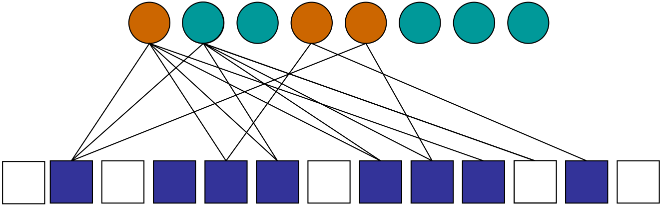
LT codes— Decoding example



1. Find codeword node c of reduced degree 1 (*released*)
2. Assign value of c to the variable node v connected to it
3. Subtract the value of v from all the codeword nodes connected to it
4. Remove all edges incident on v
5. If there are message symbols that have not been decoded, go to Step 1

[Example: A. Shokrollahi]

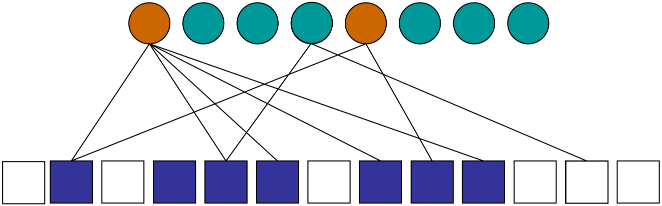
LT codes— Decoding example



1. Find codeword node c of reduced degree 1 (*released*)
2. Assign value of c to the variable node v connected to it
3. Subtract the value of v from all the codeword nodes connected to it
4. Remove all edges incident on v
5. If there are message symbols that have not been decoded, go to Step 1

[Example: A. Shokrollahi]

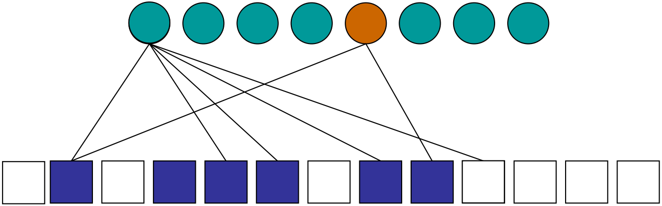
LT codes— Decoding example



1. Find codeword node c of reduced degree 1 (*released*)
2. Assign value of c to the variable node v connected to it
3. Subtract the value of v from all the codeword nodes connected to it
4. Remove all edges incident on v
5. If there are message symbols that have not been decoded, go to Step 1

[Example: A. Shokrollahi]

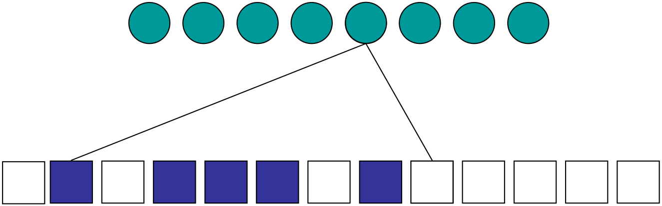
LT codes— Decoding example



1. Find codeword node c of reduced degree 1 (*released*)
2. Assign value of c to the variable node v connected to it
3. Subtract the value of v from all the codeword nodes connected to it
4. Remove all edges incident on v
5. If there are message symbols that have not been decoded, go to Step 1

[Example: A. Shokrollahi]

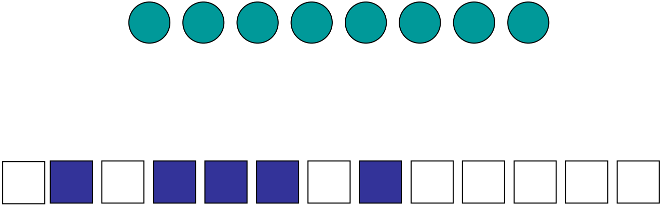
LT codes— Decoding example



1. Find codeword node c of reduced degree 1 (*released*)
2. Assign value of c to the variable node v connected to it
3. Subtract the value of v from all the codeword nodes connected to it
4. Remove all edges incident on v
5. If there are message symbols that have not been decoded, go to Step 1

[Example: A. Shokrollahi]

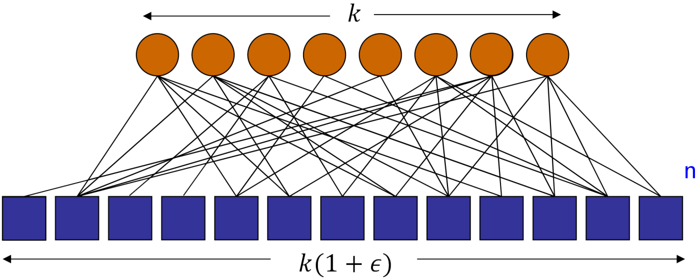
LT codes— Decoding example



1. Find codeword node c of reduced degree 1 (*released*)
2. Assign value of c to the variable node v connected to it
3. Subtract the value of v from all the codeword nodes connected to it
4. Remove all edges incident on v
5. If there are message symbols that have not been decoded, go to Step 1

[Example: A. Shokrollahi]

LT codes—goals



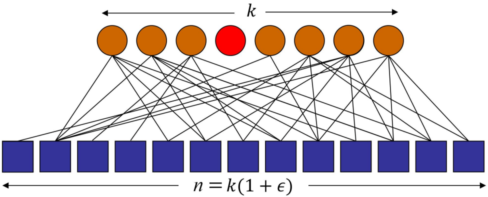
non-erased symbols

Given k message symbols encoded with an LT code, we want

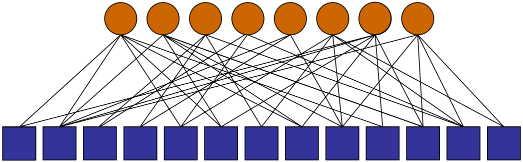
- To be able to recover the message from $(1+\epsilon)k$ non-erased code symbols, for a vanishingly small ϵ, with very high probability.
- To do so with linear (or close to linear) complexity.

LT codes—challenges

Bad things that may happen:



- An *uncovered* input symbol: an input symbol that was not hit by any of the linear combinations drawn with distribution $\mathcal{D}$.



- No codeword node is released—decoding is stuck.

LT Codes–degree distribution

- The distribution of check equation weights is crucial for correct decoding, for both of the challenges listed.
- LT codes draw check equations according to a distribution $\mathcal{D}$ on F_2^k
  - A distribution $\{\Omega_w \,|\, 1 \le w \le k\}$ on the Hamming weights
  - Uniform distribution for a given weight
  - Probability of a coefficient vector $v \in F_2^k$:

$$\mathsf{Prob}_{\mathcal{D}}(v) = \frac{\Omega_w}{\binom{k}{w}}, \quad w = \mathsf{wt}(v)$$

- The parameters of the code are $(k, \Omega(x))$,

$$\Omega(x) = \Omega_1 x + \Omega_2 x^2 + \cdots + \Omega_k x^k$$

Average weight (degree) of a code symbol

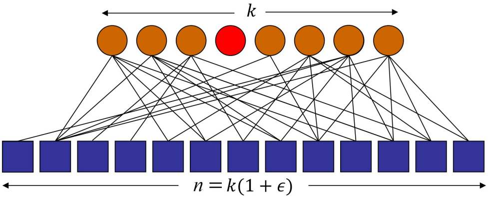
$$\Omega(x) = \sum_{w=1}^{k} \Omega_w x^w, \quad \Omega_w = \mathsf{Prob}(\mathsf{wt} = w)$$

$$E[w] = \sum_{w=1}^{k} w\,\Omega_w = \Omega'(1)$$

Average weight (degree) of a code symbol

$$\Omega(x) = \sum_{w=1}^{k} \Omega_w x^w, \quad \Omega_w = \mathsf{Prob}(\mathsf{wt} = w)$$

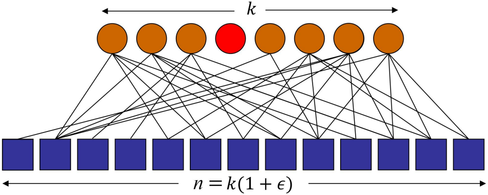
$$E[w] = \sum_{w=1}^{k} w\,\Omega_w = \Omega'(1)$$



Average weight (degree) of a code symbol

$$\Omega(x) = \sum_{w=1}^{k} \Omega_w x^w, \quad \Omega_w = \text{Prob}(\text{wt} = w)$$

$$E[w] = \sum_{w=1}^{k} w\,\Omega_w = \Omega'(1)$$



Prob(fail) $\geq$ Prob(message symbol not covered)

$$= \left(\sum_{w=1}^{k} \Omega_w \frac{\binom{k-1}{w}}{\binom{k}{w}} \right)^n = \left(\sum_{w=1}^{k} \Omega_w \left(1 - \frac{w}{k}\right) \right)^n = \left(1 - \frac{\Omega'(1)}{k}\right)^{k(1+\epsilon)}$$

$$\approx e^{-\Omega'(1)(1+\epsilon)} \overset{\text{want}}{\leq} \frac{1}{k^c} \Rightarrow \Omega'(1) \geq \frac{c \ln k}{1+\epsilon}$$

average degree must be at least logarithmic

LT—Ensuring enough symbols are released (w.h.p.)

We are interested in the probability that an output symbol of initial degree w is released at step $i+1$, when the $i+1$st input symbol is recovered.

This is the probability that the symbol has *exactly* one neighbor among the $k-i-1$ input symbols that are not yet recovered, and that not all the remaining $w-1$ neighbors are among the i already recovered. It can be shown that under the assumptions on the distribution $\mathcal{D}$,

$$P(\text{output symbol “released” at step } i{+}1 \mid \deg \text{ is } w) =$$

$$w\left(1-\frac{i+1}{k}\right)\left(\left(\frac{i+1}{k}\right)^{w-1}-\left(\frac{i}{k}\right)^{w-1}\right).$$

$$P(\text{output symbol “released” at step } i{+}1) =$$

$$\left(1-\frac{i+1}{k}\right)\left(\Omega'((i+1)/k)-\Omega'(i/k)\right).$$

LT—Ensuring enough symbols are released (w.h.p.)

$$P(\text{output symbol “released” at step } i{+}1) =$$

$$\left(1 - \frac{i+1}{k}\right) \left(\Omega'((i+1)/k) - \Omega'(i/k)\right) .$$

For n symbols, number released $\approx$

$$\frac{n}{k} \left(1 - \frac{i+1}{k}\right) \Omega''(i/k) \overset{\text{want}}{\geq} 1$$

with $n \approx k$:

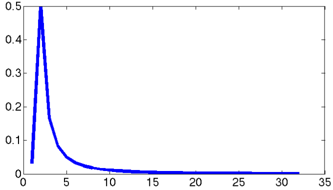
$$\Omega(x) = \Omega_1 + \sum_{i \geq 2} \frac{x^i}{i(i-1)} \quad \textit{soliton distribution}$$

LT—Ensuring enough symbols are released (w.h.p.)

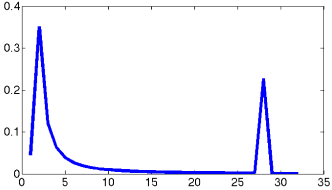
"Soliton" distribution

$$\Omega_i = \begin{cases} \frac{1}{k} & i = 1\,, \\ \frac{1}{i(i-1)} & 2 \le i \le k\,. \end{cases}$$

With the soliton distribution, the *expected* number of output nodes released at each step is *exactly* one. This makes it a poor choice in practice, as even a minimal deviation from the expected behavior will get the process stuck. More robust solutions have been developed.



Soliton $k = 32$



Robust soliton
$k = 32, M = 28, \delta = 10^{-4}$