LDPC

According to Shannon's coding theorem, to achieve capacity, one needs long codes.

$$P_e^{(n)} \to 0 \quad \text{as} \quad n \to \infty$$

One way to ensure $P_e^{(n)} \to 0$ is to increase dmin. According to Singleton bound, the minimum distance of linear codes

$$\begin{aligned} d_{min} &\leq 1 + (n-k) \\ &= 1 + n - nR \qquad R = \frac{k}{n} \Rightarrow k = nR \\ &= 1 + n(1-R) \end{aligned}$$

Thus, as $n \to \infty$, dmin can increase unboundedly.

As the block length n grows, the decoding complexity increases. Managing complexity as block length increases is the main innovation in LDPC and turbo codes.

Another interesting aspect of Shannon's coding theorem is that for large block lengths (n), almost any code chosen at random is good.

That is one could pick a random mapping from messages to codewords, and get good performance. This aspect was considered "problematic" in the design of practical codes, since randomly chosen codes have little structure. So their decoding is hard, requiring exhaustive search.

However, LDPC and turbo codes turn the conventional wisdom on its head. They are randomly chosen with just enough structure to allow easy decoding.

As we will see, ML decoding of these codes is very hard. But a suboptimal decoder — iterative decoder — comes very close to ML in performance.

Preview of Iterative Decoding

Consider Hamming code (7, 4)

$$G = \left[\begin{array}{ccc|cccc} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right]$$

$$= \left[\begin{array}{c|c} P & I_k \end{array}\right]$$

Encoding

$$\underset{1\times 7}{C} = \underset{1\times 4}{m}\ \underset{4\times 7}{G}$$

The parity check matrix

$$H = \left[\begin{array}{c|c} I_{n-k} & P^T \end{array}\right]$$

$$= \left[\begin{array}{ccc|cccc} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{array}\right]$$

We know that every valid codeword belongs to the null space of the parity check matrix

$\Rightarrow$ $C H^T = \underline{0}$ if and only if C is a valid codeword

Let $c = [c_0\ c_1\ c_2\ c_3\ c_4\ c_5\ c_6]$

Then $cH^T = 0$

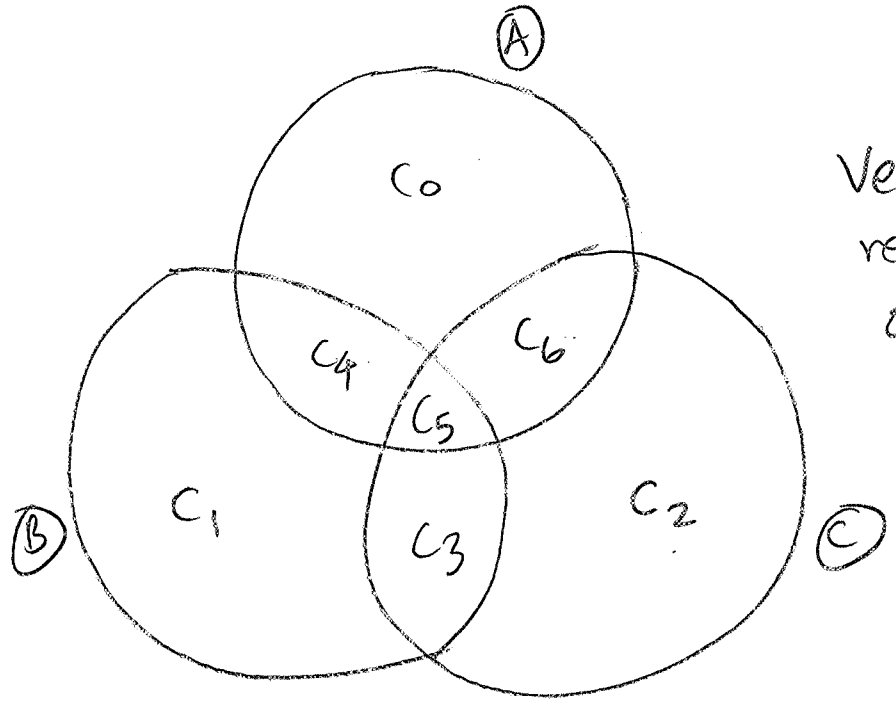
$$\Rightarrow \left.\begin{aligned} c_0 + c_4 + c_5 + c_6 &= 0 \\ c_1 + c_3 + c_4 + c_5 &= 0 \\ c_2 + c_3 + c_5 + c_6 &= 0 \end{aligned}\right\} \circledast$$

Since we are working in GF(2), we can rewrite ⊛ as

$$\left.\begin{aligned} c_0 &= c_4 + c_5 + c_6 \\ c_1 &= c_3 + c_4 + c_5 \\ c_2 &= c_3 + c_5 + c_6 \end{aligned}\right\} \diamondsuit$$

Think of (c_3, c_4, c_5, c_6) as <u>independent</u> (info) variables and (c_0, c_1, c_2) as <u>derived</u> (parity) variables.

We will represent ◈ on a Venn diagram.



Venn Diagram representation of Hamming code (7,4)

- Each circle corresponds to one parity-check constraint.
- The number of ones in each circle must be even. Why?

Consider Encoding

Let $(c_3, c_4, c_5, c_6) = (0, 1, 0, 1)$

Apply one parity check equation at a time to obtain

$c_0 = 0$

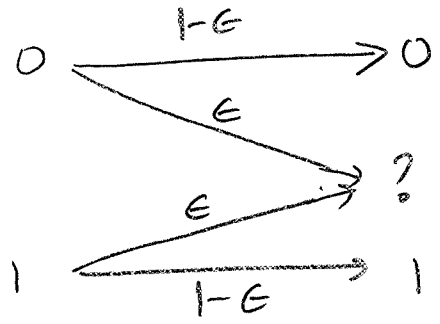
$c_1 = 1$

$c_2 = 1$

$\Rightarrow$ codeword $= (0, 1, 1, 0, 1, 0, 1)$

Now consider decoding

To make life easier, we will consider a simpler channel, binary erasure channel (BEC)

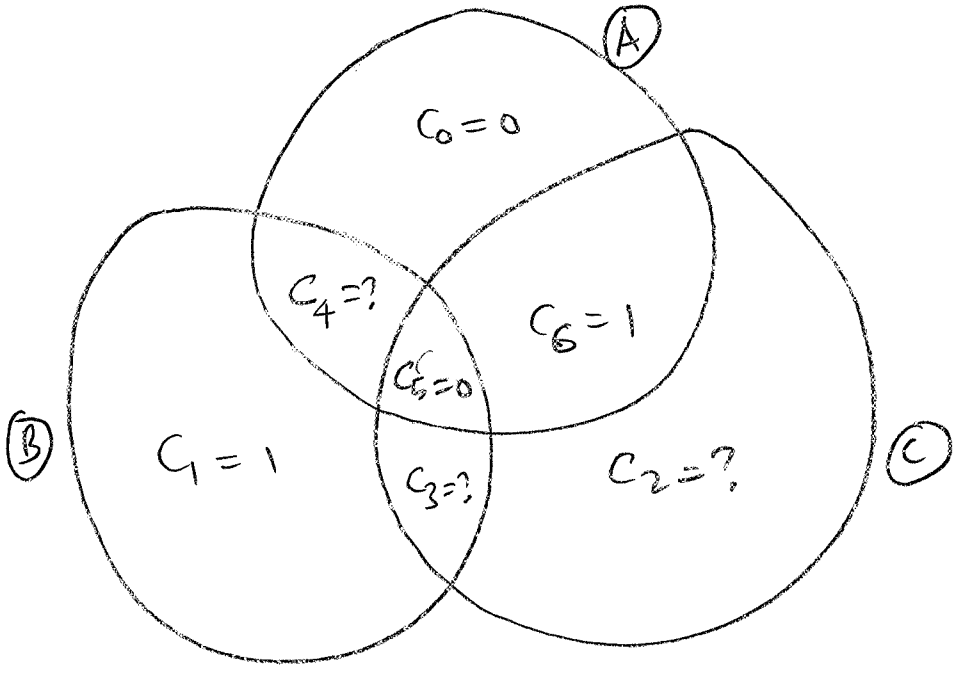


Assume we sent

$$(0, 1, 1, 0, 1, 0, 1)$$
$$c_0\ c_1\ c_2\ c_3\ c_4\ c_5\ c_6$$

and received

$$(0, 1, ?, ?, ?, 0, 1)$$



Start with set (A)

$c_4 = 1$ satisfies the equation

Next move to (B)

$c_3 = 0$ satisfies the equation

Then to (C)

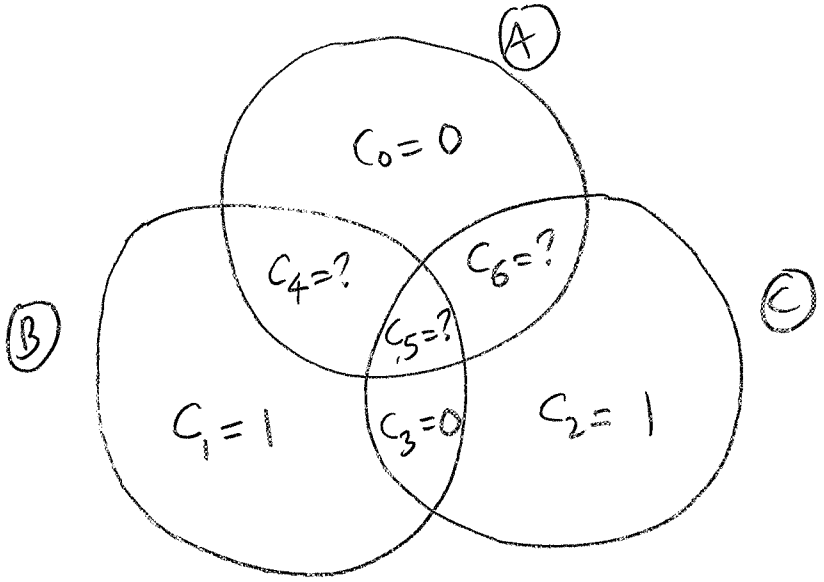
$c_2 = 1$ satisfies the equation

Decoded codeword = $(0, 1, 1, 0, 1, 0, 1)$

(valid codeword)

The most interesting aspect of this decoder is that it does only local processing. We fixed one bit at a time without any consideration to other equations.

Unfortunately, this local processing does not work all the time.



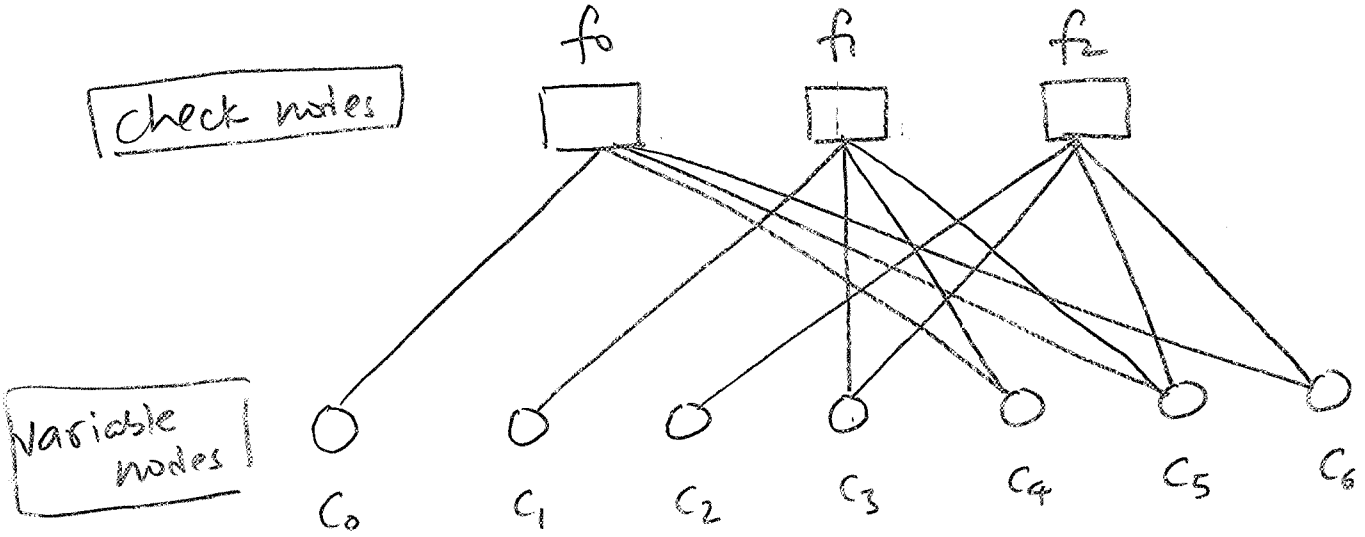
Let the received vector be $(0, 1, 1, 0, ?, ?, ?)$

Local processing fails to resolve any ambiguity, none of the parity check equations by themselves help.

However an exhaustive search will show that $(0, 1, 1, 0, 1, 0, 1)$ is a possible codeword.

In practice, for long codewords, the problem situations do not occur very often. But we will use another representation.

Tanner graph



$$H = \left[\begin{array}{ccc|cccc} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{array}\right]$$

Tanner graphs are graphical representation for LDPC codes. They provide a complete representation of the code. They also help in convenient description of the decoding algorithm.

Tanner graphs are <u>bipartite graphs</u>. That means that the nodes of the graph can be separated into two distinctive sets and edges are only connecting nodes of two different types.

Tanner graphs representing parity check matrices have $(n-k)$ check nodes = # of parity bits
n variable nodes = # of coded bits

Message passing algorithm

(1) In the first step, all variable nodes send their bit to check nodes. (That's all they know)

$0 \longrightarrow f_0$
$1 \longrightarrow f_1$
$? \longrightarrow f_2$
$? \longrightarrow f_1, f_2$
$? \longrightarrow f_0, f_1$
$0 \longrightarrow f_0, f_1, f_2$
$1 \longrightarrow f_0, f_2$

(2) Each check node computes the parity assuming other received bits are correct and then sends that parity to that variable node. If it cannot compute anything it sends ?

resolved at c_4

$f_0 \longrightarrow c_0 = ?,$ $\boxed{c_4 = 1}$ $c_5 = ?,\ c_6 = ?$

$f_1 \longrightarrow c_1 = ?,\ c_3 = ?,\ c_4 = ?,\ c_5 = ?$

$f_2 \longrightarrow c_2 = ?,\ c_3 = ?,\ c_5 = ?,\ c_6 = ?$

(3) If a bit is resolved from ? to 0 or 1, flip it.
Repeat step (1), (2) till all variable node are
0 or 1.

FACTOR GRAPHS

In this part, we will formalize the local processing algorithm using the machinery of factor graphs and message passing over factor graphs.

Factor graphs are simply a recursive use of distributive law and allow efficient computation of marginals of multivariate functions.

Distributive Law

Let $\mathbb{F}$ be a field, and let $a, b, c \in \mathbb{F}$.

By the distributive law

$$ab + ac = a(b+c)$$

We will use this simple fact to reduce computational complexity of evaluations like

$$\sum_{i,j} a_i b_j = \left(\sum_i a_i\right)\left(\sum_j b_j\right).$$

Example :

$$f(x_1, x_2, x_3, x_4, x_5, x_6) = f_1(x_1, x_2, x_3)\, f_2(x_1, x_4, x_6)\, f_3(x_4)\, f_4(x_4, x_5)$$

We are interested in computing the marginal of f with respect to x_1.

$$f(x_1) = \sum_{x_2, x_3, x_4, x_5, x_6} f(x_1, x_2, x_3, x_4, x_5, x_6) = \sum_{\sim x_1} f(x_1, x_2, x_3, x_4, x_5, x_6)$$

$\sum_{\sim x_1}$: summation over all other variables except x_1

Assume that all variables take values in a finite alphabet, call it $\mathcal{X}$.

Determining $f(x_1)$ $\forall$ x_1 by brute force requires $\Theta(|\mathcal{X}|^6)$ operations. We can do better by using the factorization.

$$f(x_1) = \left[\sum_{x_2, x_3} f_1(x_1, x_2, x_3)\right] \left[\sum_{x_4} f_3(x_4) \left(\sum_{x_6} f_2(x_1, x_4, x_6)\right)\left(\sum_{x_5} f_4(x_4, x_5)\right)\right]$$

Factor 1 | Factor 2

Fix x_1

The evaluation of the first factor can be accomplished with $\Theta(|X|^2)$ operations.

The second factor depends only on x_4, x_5 and x_6.

For each value of x_4 (and fixed x_1), determine $\sum_{x_5} f_4(x_4, x_5)$ and $\sum_{x_6} f_2(x_1, x_4, x_6)$. Multiply by $f_3(x_4)$ and sum over x_4.

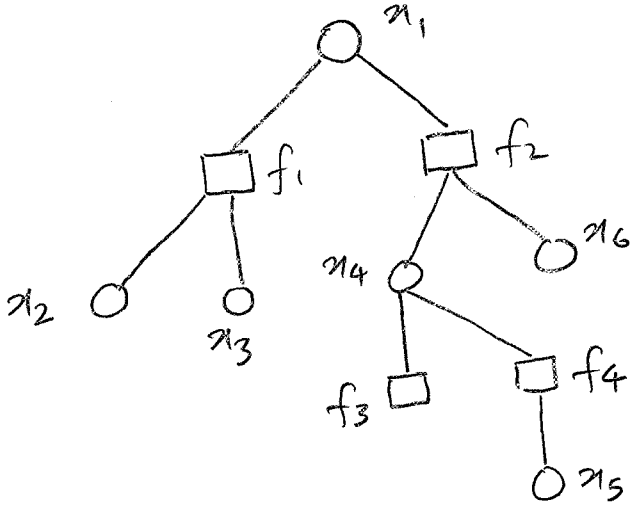
↳ takes $\Theta(|X|^2)$ operations.

Now there are $|X|$ values of x_1, the overall task has complexity $\Theta(|X|^3)$.

Graphical Representation of Factorization

1) For each variable, draw a variable node (circle).
2) For each factor, draw a factor node (square).
3) Connect a variable node to a factor node by an edge iff the corresponding variable appears in this factor.

Factor graph is bipartite.



For our problem, the factor graph is a bipartite tree.

$\Rightarrow$ There are no cycles in the graph.

$\Rightarrow$ The marginal can be computed efficiently by message-passing

Example (Special case: Tanner Graph)

$$H = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \end{matrix} \\ \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

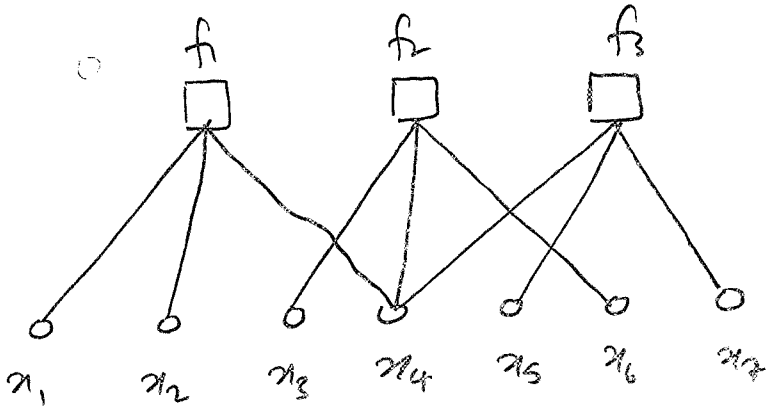
Let $\mathbb{F}_2$ denote binary field $\{0,1\}$

$$x = (x_1, \dots, x_7)$$

$$f(x_1, x_2, \dots, x_7) = \mathbb{1}_{\{x \in C(H)\}} = \begin{cases} 1 & x^T H = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$f(x_1, \dots, x_7) = \underbrace{\mathbb{1}_{\{x_1+x_2+x_4=0\}}}_{f_1} \underbrace{\mathbb{1}_{\{x_3+x_4+x_6=0\}}}_{f_2} \underbrace{\mathbb{1}_{\{x_4+x_5+x_7=0\}}}_{f_3}$$

The function f is also called code membership function since it tests whether a vector is a valid codeword or not.



Any binary linear block code has a Tanner graph representation.

Recursive Determination of Marginals

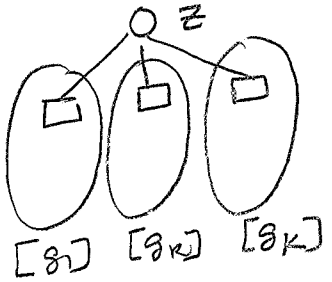
Consider a generic function g, and suppose the associated factor graph is a tree.

Suppose we want to compute

$$g(z) = \sum_{\sim z} g(z, \dots)$$

Since the factor graph of g is a bipartite tree, g has a generic factorization of the form

$$g(z, \dots) = \prod_{k=1}^{K} [g_k(z, \dots)]$$



for some integer K.

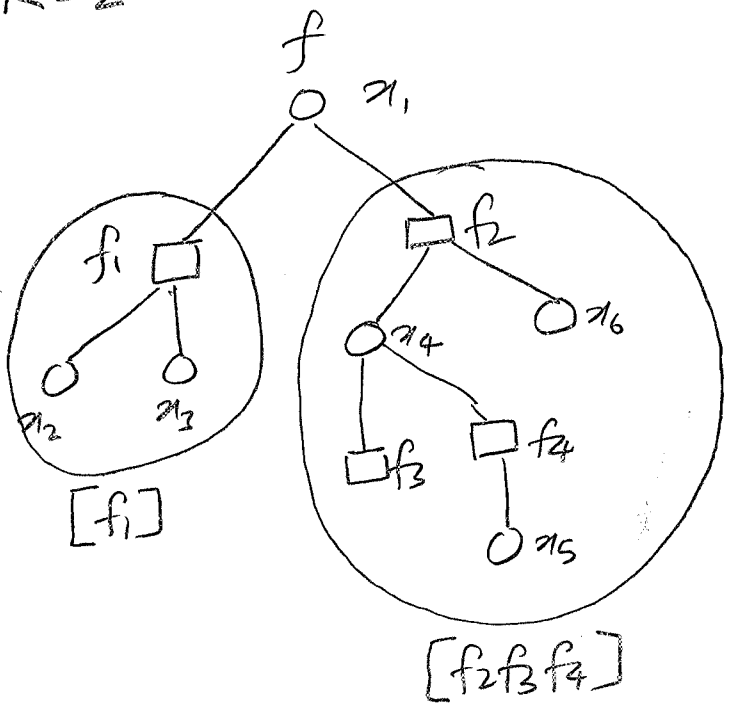
The above factorization has an important property (1) z appears in all factors
(2) all other variables appear in only one factor

Proof: Assume another variable is contained in two of the factors. This implies that there exists a path which connects these two factors, in addition to path via z. Contradiction since the factor graph is a tree.

Example (contd)

$$f(x_1, \ldots) = [f_1(x_1 x_2 x_3)]\,[f_2(x_1 x_4 x_6)\, f_3(x_4) f_4(x_4 x_5)]$$

$\Rightarrow K = 2$



$$\sum_{\sim Z}{}' g(Z, \ldots) = \sum_{\sim Z}{}' \prod_{k=1}^{K} [g_k(Z, \ldots)] \quad \left.\right\} \text{marginal of product}$$

$$= \prod_{k=1}^{K} \left[\sum_{\sim Z}{}' g_k(Z, \ldots) \right] \quad \left.\right\} \text{product of marginal}$$

$$g_1(Z, \ldots) = f_1(x_1 x_2 x_3)$$

$$g_2(Z, \ldots) = f_2(x_1 x_4 x_6)\, f_3(x_4) f_4(x_4 x_5)$$

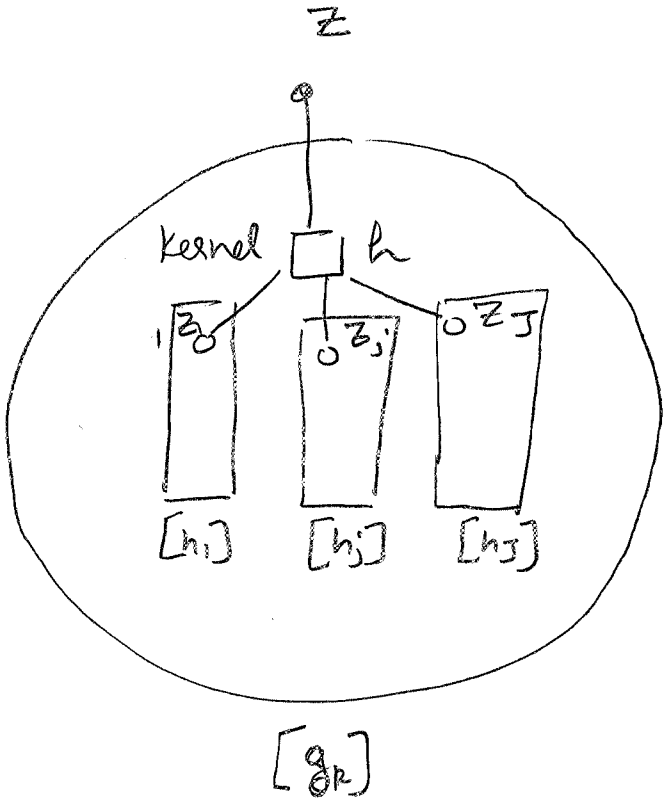
In general, each g_k is itself a product of factors. Since the factor graph is a bipartite tree, g_k must in turn have a generic factorization of the form

$$g_k(z, \dots) = \underbrace{h(z, z_1, \dots, z_J)}_{\text{Kernel}} \prod_{j=1}^{J} \underbrace{[h_j(z_j, \dots)]}_{\text{factors}}$$

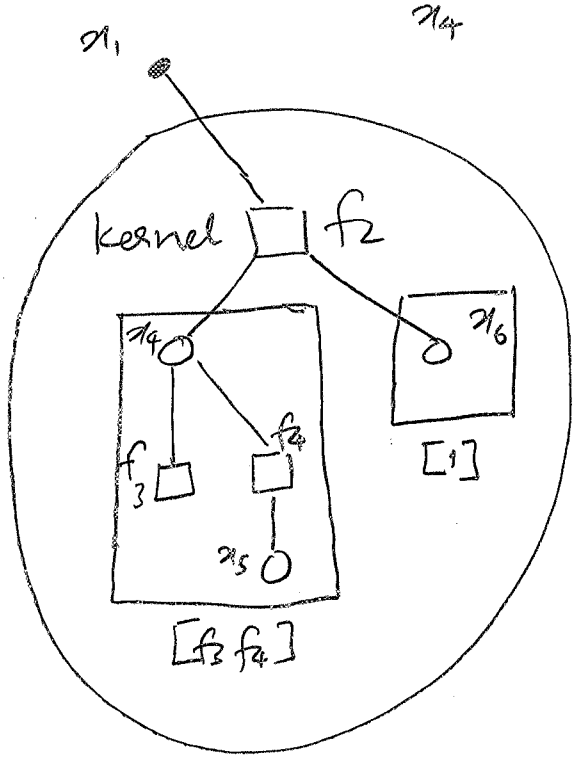
z appears only in the kernel, other variables appear maximum 2 places

Factor 2 of example

$$f_2(x_1, x_4 x_6)\, f_3(x_4)\, f_4(x_4 x_5) = f_2(x_1, x_4 x_6) \underbrace{[\underbrace{f_3(x_4) f_4(x_4 x_5)}_{x_4}]}_{x_6} [1]$$



$[g_k]$



$[f_2 f_3 f_4]$

$$\sum_{\sim z} g_k(z,\dots) = \sum_{\sim z} h(z,z_1,\dots,z_J) \prod_{j=1}^{J} [h_j(z_j,\dots)]$$

$$= \sum_{\sim z} h(z,z_1,\dots,z_J) \underbrace{\prod_{j=1}^{J} \left[\sum_{\sim z_j} h_j(z_j,\dots)\right]}_{\text{product of marginals}}$$

The desired marginal $\sum_{\sim z} g_k(z,\dots)$ can be computed by multiplying the kernel $h(z,z_1,\dots z_J)$ with individual marginals $\sum_{\sim z_j} h_j(z_j,\dots)$ and summing out all remaining variables other than z.

We are back to where we started.

Each factor $h_j(z_j,\dots)$ has the same generic form as the original function $g(z,\dots)$, so that we can continue to break down the marginalization task into smaller pieces.

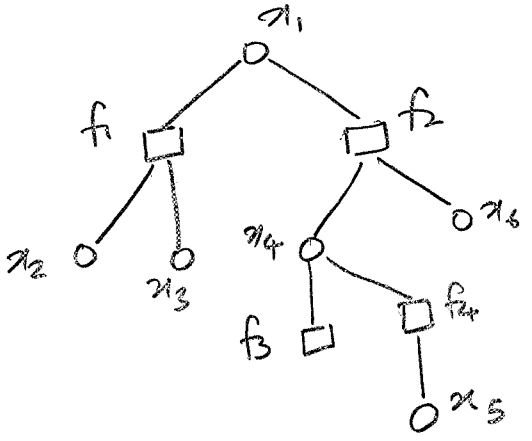
This recursive process continues until we have reached the leaves of the tree. The calculation of the marginals then follows the recursive splitting in reverse.

MARGINALIZATION Via MESSAGE PASSING

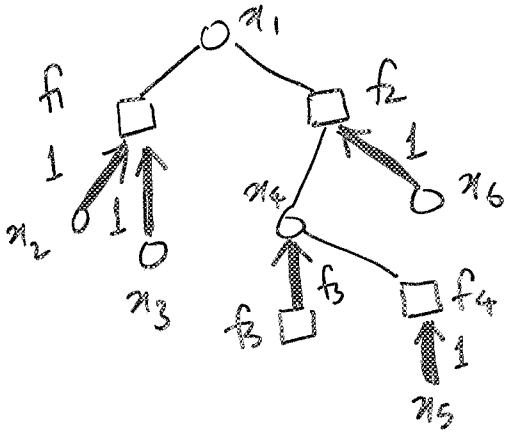
We will now use the graph representation to compute the marginal $f(x_1)$.

Example:

$$f(x_1,x_2,x_3,x_4,x_5,x_6) = f_1(x_1,x_2,x_3)\, f_2(x_1,x_4,x_6)\, f_3(x_4)\, f_4(x_4,x_5)$$



(1) Message passing starts at leaf nodes.
The variable leaf nodes x_2, x_3, x_5 and x_6 send the constant function '1'

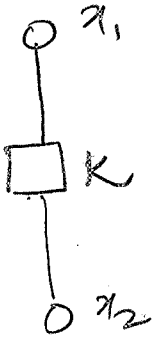


The factor leaf node f_3 sends the function f_3 to its parent node.

Why constant function?

Consider a simpler example

$$K(x_1) = \sum_{\sim x_2} K(x_1, x_2)$$



Our general message passing rule is

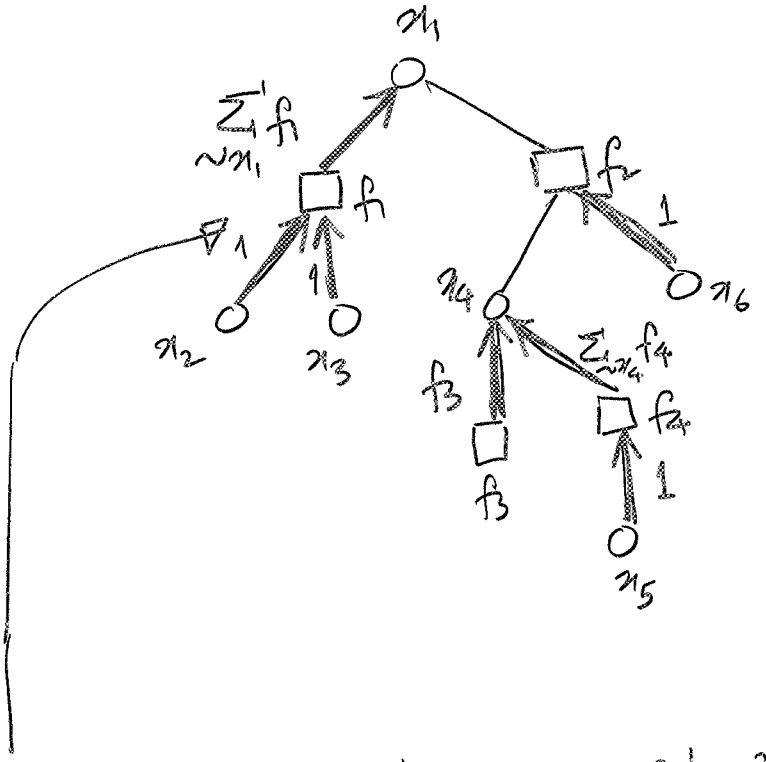
$$\sum_{\sim z} g(z, \ldots) = \sum_{\sim z} \underbrace{h(z, z_1, \ldots z_J)}_{\text{Kernel}} \underbrace{\prod_{j=1}^{J} \sum_{\sim z_j} h_j(z_j \ldots)}_{\text{product of marginals}}$$

Kernel $h(z, z_1) = K(x_1, x_2)$

$$h_1(z_1) = 1$$

$$K(x_1) = \sum_{\sim x_2} \underbrace{K(x_1, x_2)}_{\text{Kernel}} \cdot \underbrace{1}_{x_2}$$

↑ correct initialization for x_2



The factor node f_1 has received messages from both its children and proceeds to send message up.

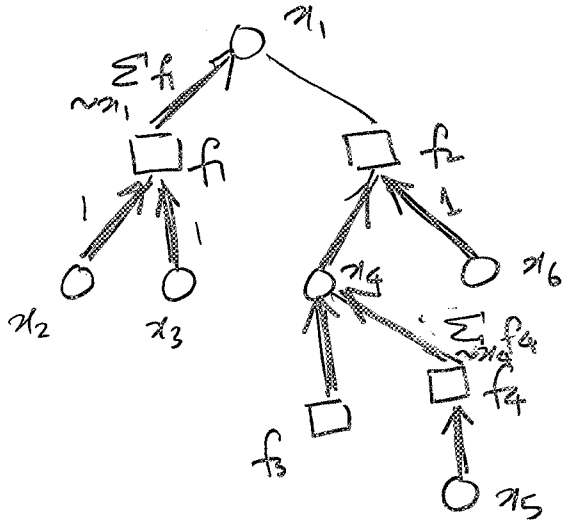
$$\text{Kernel} = f_1(x_1, x_2 x_3)$$

$$\text{marginals} = \underbrace{1}_{x_2} \,,\, \underbrace{1}_{x_3}$$

$$\text{message} = \sum_{\sim x_1} f_1(x_1, x_2 x_3) \cdot 1 \cdot 1$$

Node f_4 has received message from x_5 and sends

$$\sum_{\sim x_4} f_4(x_4 x_5)$$



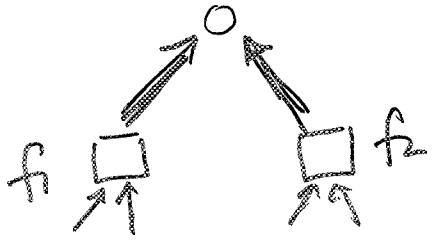
Now node x_4 has received all the messages.
It sends a message up, which is a function of x_4

$$\text{The kernel} = f_3(x_4)$$

$$\text{marginal} = \sum_{\sim x_4} f_4(x_4 x_5)$$

$$\text{Message} = f_3(x_4) \sum_{\sim x_4}' f_4(x_3 x_4)$$

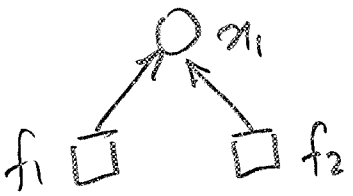
(4)



Now f_2 can forward its message

$$\sum_{\sim x_1}' f_2(x_1, x_4 x_6) \underbrace{\quad}_{\text{Kernel}} \left[f_3(x_4) \sum_{\sim x_4}' f_4(x_3 x_4) \right] [1]$$

(5) Now node x_1 has received messages from both its children,



$$\sum_{\sim x_1}{}' f_1 f_2 f_3 f_4 = \underbrace{\sum_{\sim x_1} f_1}_{\text{coming from } f_1} \underbrace{\sum_{\sim x_1}{}' f_2 f_3 f_4}_{\text{coming from } f_2}$$

NOTE : So far, we have only considered marginalization of function f w.r.t only x_1.
In practice we will be interested in marginalization with respect to all variables.
We saw that a single marginalization can be performed efficiently if the factor graph of f is a tree, and that the complexity of the computation depends on the
- largest degree of the factor graph ($2 \to \mathcal{O}(|\mathcal{X}|^2)$)

and - the alphabet size ($|\mathcal{X}|$) $\rightsquigarrow \mathcal{O}(|\mathcal{X}|^3)$

Now consider the problem of computing all marginals.

Method 1

- Draw a tree rooted at a variable ($x_1, x_2 \ldots x_6$)
- Execute single marginal message-passing algorithm for each tree

It is easy to see that the algorithm does not depend on which node is the root of the tree. In fact, all computations can be performed simultaneously on a single tree.

Method 2

- Start at all the leaf nodes
- For every edge, compute the outgoing message along this edge as soon as the node connected to this edge has received all messages along all it incoming edges.
- Continue till a message has been sent in both directions along every edge.

Method 2 computes all marginals, so it is more complex than computing a single marginal but only by a factor roughly equal to the average degree of the nodes.

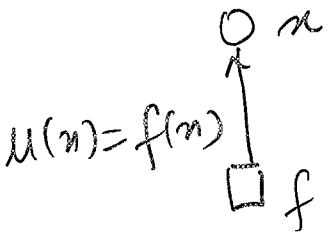
Message-passing Rules

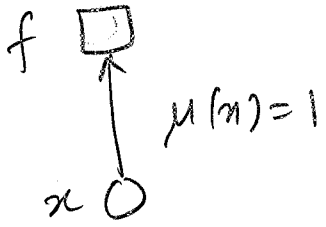
- Denote messages as μ ← functions on X
- Message passing starts at leaf nodes
- Consider a node and one of its adjacent edges (e). As soon as the incoming messages to the node along all other adjacent edges have been received, process them & send it along e.
- This process continues until messages along all edges in the tree have been processed.
- In the final step, the marginals are computed by combining all messages which enter a particular variable node.

Messages often will be probabilities or beliefs
⇒ called belief propagation (BP).

Message-passing Rules

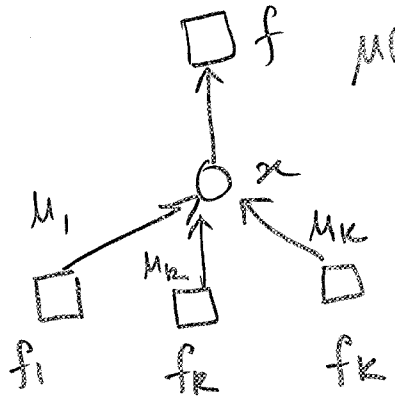
Initialization @ leaf nodes



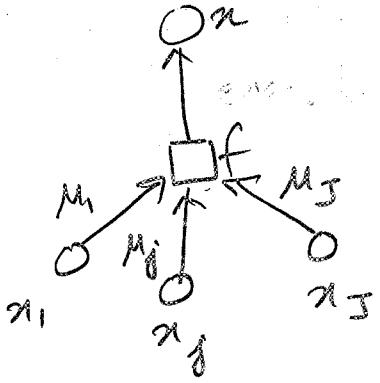


Variable / function node processing

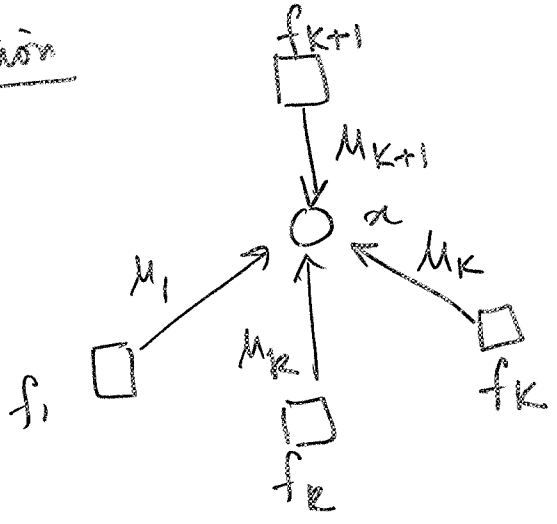
$$\mu(x) = \prod_{k=1}^{K} \mu_k(x)$$



$$\mu(x) = \sum_{\sim x} f(x, x_1, \ldots x_J) \prod_{j=1}^{J} \mu_j(x)$$



Marginalization



$$\prod_{k=1}^{K} \mu_k(x)$$

DECODING VIA MESSAGE PASSING

Bit-wise MAP Decoding

Assume we transmit over binary-input ($x_i \in \{0,1\}$) memoryless channel using a linear code $C(H)$ defined by its parity check matrix H.

Assume that codewords are chosen at random.

Since the channel is memoryless

$$p_{Y|X}(y|x) = \prod_{i=1}^{n} p_{Y_i|X_i}(y_i|x_i)$$

Bit-wise MAP decoder is

$$\hat{x}_i^{MAP}(y) = \arg\max_{x_i \in \{0,1\}} p_{X_i|Y}(x_i|y)$$

$$= \arg\max_{x_i \in \{0,1\}} \sum_{\sim x_i} p_{X|Y}(x|y)$$

$$= \arg\max_{x_i \in \{0,1\}} \sum_{\sim x_i} p_{Y|X}(y|x)\, p_X(x)$$

($p_{Y|X}$: memoryless; p_X: uniform prior)

$$= \arg\max_{x_i \in \{0,1\}} \sum_{\sim x_i} \left(\prod_j p_{Y_j|X_j}(y_j|x_j) \right) \mathbb{1}_{\{x \in C\}} \quad (*)$$

sum-product algebra

$ab + ac = a(b+c)$

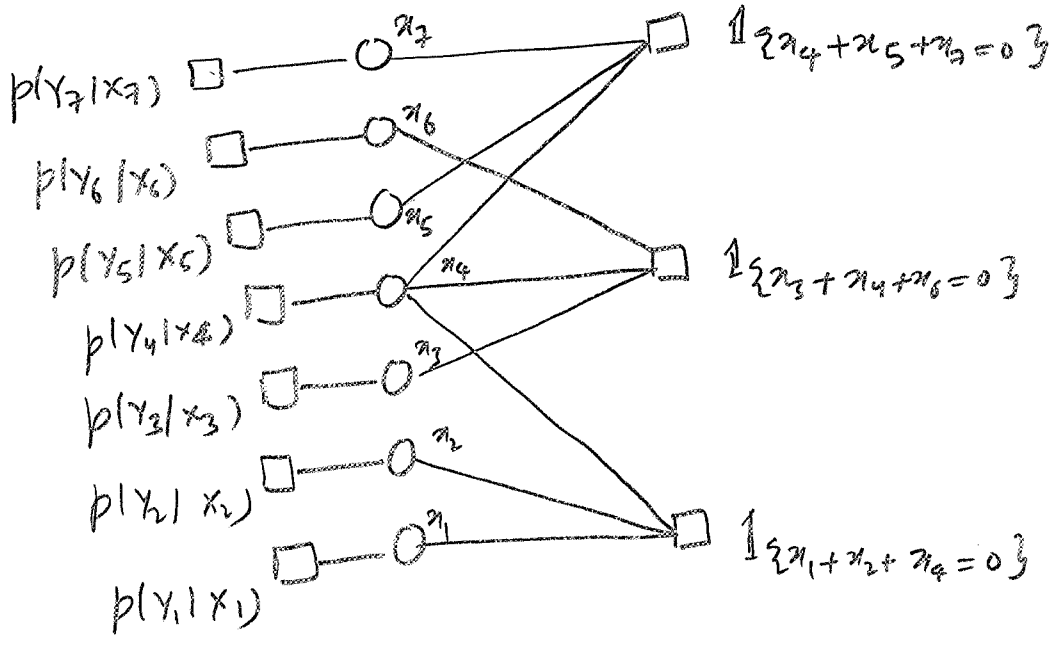
Assume that the code membership function $\mathbb{1}_{\{x \in C\}}$ has a factorized form. From (*), we can see that bit-wise decoding problem is equivalent to calculating the marginal of a factorized function and choosing the value that maximizes this marginal.

Example Consider the parity check

$$H = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \end{matrix} \\ \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

$$\mathbb{1}_{\{x \in C\}} = \underbrace{\mathbb{1}_{\{x_1+x_2+x_4=0\}} \mathbb{1}_{\{x_3+x_4+x_6=0\}} \mathbb{1}_{\{x_4+x_5+x_7=0\}}}$$

$$\arg\max_{x_i \in \{0,1\} \; \forall x_i} \sum \prod_{j=1}^{7} p_{Y_j|X_j}(y_j|x_j) \, \mathbb{1}_{\{x \in C\}}$$



Block-wise MAP decoding

$$\hat{x}^{MAP}(y) = \arg\max_x p_{X|Y}(x|y)$$

$$= \arg\max_x p_{Y|X}(y|x)\, p_X(x) \quad \leftarrow \text{uniform prior on codewords}$$

$$= \arg\max_x \prod_j p_{Y_j|X_j}(y_j|x_j)\, \mathbb{1}_{\{x \in C\}}$$

Consider the i^{th} bit of $\hat{x}^{MAP}(y)$, $(\hat{x}^{MAP}(y))_i$

$$(\hat{x}^{MAP}(y))_i = \arg\max_{x_i} \max_{\sim x_i} \prod_j p_{Y_j|X_j}(y_j|x_j)\, \mathbb{1}_{\{x \in C\}}$$

$$\text{(**)} \quad = \arg\max_{x_i} \max_{\sim x_i} \sum_j \log p_{Y_j|X_j}(y_j|x_j) + \log(\mathbb{1}_{\{x \in C\}})$$

Compare (*) and (**)

(*) addition goes into maximization & multiplication goes into addition

(**) $\max_{\sim x_i}$ goes into maximization & additions into $\max_{\sim x_i}$

max-sum algebra

$$\max\{x+y, x+3\} = x + \max\{y, 3\}$$

$$(\hat{x}^{MAP}(y))_i = \arg \min_{x_i \in \{0,1\}} \min_{\sim x_i} \sum_{j=1}^{n} -\log p_{Y_j|X_j}(y_j|x_j) - \log(\mathbb{1}_{\{x \in C\}})$$

If the channel output is discrete, then we are dealing with probability mass function. In this case, the above form is more convenient since

$$-\log p_{Y_j|X_j}(y_j|x_j) \text{ is positive.}$$

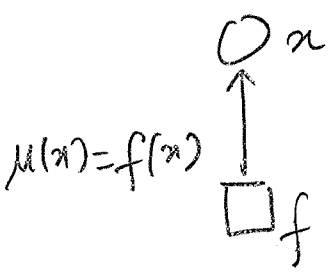
This leads to use of min-sum algebra.

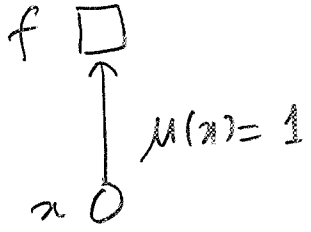
What is marginalization of a function $f(x_1, \cdots x_n)$ in the max-sum algebra?

$$f(x_1) = \max_{x_2, \cdots, x_n} f(x_1, \cdots, x_n) = \max_{\sim x_1} f(x_1, \cdots x_n)$$

Message-passing rules for max-sum

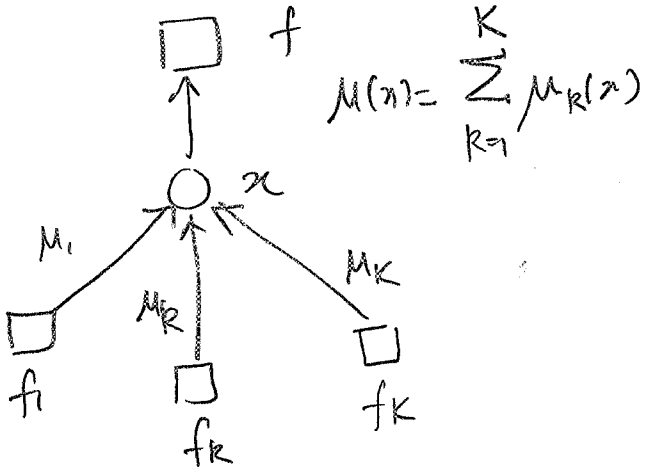
Initialization



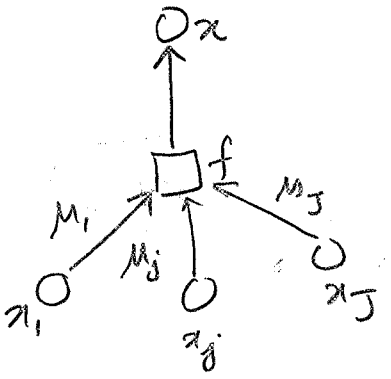


Variable/function node processing

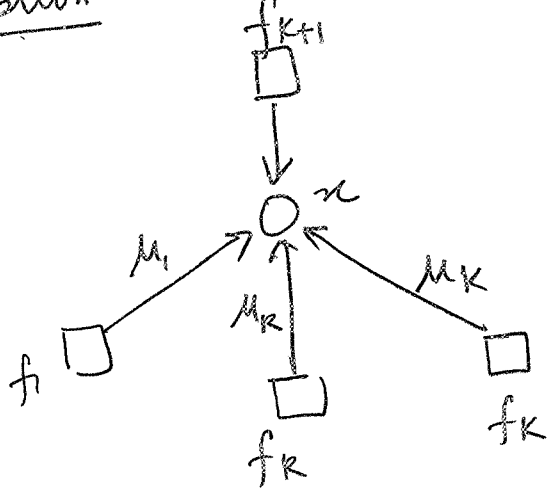
$$\mu(x) = \sum_{k=1}^{K} \mu_k(x)$$



$$\mu(x) = \max_{\sim x} \left\{ f(x, x_1, \ldots x_J) + \sum_{j=1}^{J} \mu_j(x_j) \right\}$$



Marginalization



$$\sum_{k=1}^{K+1} \mu_k(x)$$

Message passing is an efficient way to compute marginals and then choose the value that maximizes the marginal.

$\Rightarrow$ It is optimal for a tanner graph which is a tree.

However class of codes which admit a cycle-free tanner graph is not powerful enough to perform well.

Result: Let C be a binary linear code of rate r that admits a binary Tanner graph that is a forest. Then C contains at least $\left(\frac{2r-1}{2}\right)n$ codewords of weight 2.

Result (Relation between bit-wise and block-wise MAP decoding)

If a binary linear code has minimum distance d_{min}, then, for any given channel, the codeword bit error probability of bit-wise decoder, p_b and the block error probability of the ML(MAP) decoder, p_B are related by

$$p_B \geq p_b \geq \frac{1}{2}\frac{d_{min}}{N} p_B$$

Simplification of Message-passing Rules for Bit-wise MAP decoding

In the binary case, a message $\mu(x)$ can be thought of as a real-valued vector of length 2

$$(\mu(1), \mu(0))$$

The initial such message sent from the factor leaf node representing the ith channel realization to the variable node x_i is

$$(p_{Y_i|X_i}(y_i|1), p_{Y_i|X_i}(y_i|0))$$

Recall that at a variable node of degree $K+1$, the message-passing rule requires a pointwise multiplication:

$$\mu(1) = \prod_{k=1}^{K} \mu_k(1) \quad , \quad \mu(0) = \prod_{k=1}^{K} \mu_k(0)$$

Introduce the likelihood ratios

$$r_k = \frac{\mu_k(1)}{\mu_k(0)}$$

In the first step, these likelihood ratios are associated with channel observations.

We have

$$r = \frac{\mu(1)}{\mu(0)} = \frac{\prod_{k=1}^{K} \mu_k(1)}{\prod_{k=1}^{K} \mu_k(0)} = \prod_{k=1}^{K} r_k$$

$\Rightarrow$ the ratio of the outgoing message at a variable node is the product of the incoming ratios. If we define the log-likelihood ratios $l_k = \log(r_k)$, then the processing rule is

$$l = \log r = \sum_{k=1}^{K} \log r_k = \sum_{k=1}^{K} l_k$$

Now consider the ratio of an outgoing message at a check node which has degree $J+1$. For a check node, the associated kernel is

$$f(x, x_1, \ldots x_J) = \mathbb{1}\{\sum_{j=1}^{J} x_j = x\}$$

$$r = \frac{\mu(1)}{\mu(0)} = \frac{\sum_{\sim x} f(1, x_1, x_2 \ldots, x_J) \prod_{j=1}^{J} \mu_j(x_j)}{\sum_{\sim x} f(0, x_1, x_2, \ldots, x_J) \prod_{j=1}^{J} \mu_j(x_j)}$$

$$= \frac{\sum_{\sim x : \sum x_j = 1} \prod_{j=1}^{J} \mu_j(x_j)}{\sum_{\sim x : \sum x_j = 0} \prod_{j=1}^{J} \mu_j(x_j)} = \frac{\sum_{\sim x : \sum x_j = 1} \prod \frac{\mu_j(x_j)}{\mu_j(0)}}{\sum_{\sim x : \sum x_j = 0} \prod \frac{\mu_j(x_j)}{\mu_j(0)}}$$

$= r_j$ when $x_j = 1$

$$= \frac{\sum_{x : \sum x_j = 1} \prod_{j=1}^{J} r_j^{x_j}}{\sum_{x : \sum x_j = 0} \prod_{j=1}^{J} r_j^{x_j}} \quad \leftarrow \left(r_j^{x_j} = \begin{cases} r_j & x_j = 1 \\ 1 & x_j = 0 \end{cases} \right)$$

check by expanding it

$$= \frac{\prod_{j=1}^{J} (r_j + 1) + \prod_{j=1}^{J} (r_j - 1)}{\prod_{j=1}^{J} (r_j + 1) - \prod_{j=1}^{J} (r_j - 1)}$$

Divide both numerator and denominator by $\prod_{j=1}^{J} (r_j + 1)$

$$r = \frac{1 + \prod_j \frac{r_j - 1}{r_j + 1}}{1 - \prod_j \frac{r_j - 1}{r_j + 1}}$$

$$\Rightarrow \frac{r-1}{r+1} = \prod_j \frac{r_j - 1}{r_j + 1} \quad - (A)$$

Now $l = \log(r) \Rightarrow r = e^l$

$$\Rightarrow \frac{r-1}{r+1} = \tanh(l/2) \quad - (B)$$

Combining (A) & (B)

$$\tanh(\ell/2) = \frac{\alpha-1}{\alpha+1} = \prod_{j=1}^{J} \frac{\alpha_{j-1}}{\alpha_{j+1}} = \prod_{j=1}^{J} \tanh(\ell_j/2)$$

$$\Rightarrow \quad \ell = 2 \tanh^{-1}\left(\prod_{j=1}^{J} \tanh(\ell_j/2) \right) \quad \text{—} \blacksquare$$

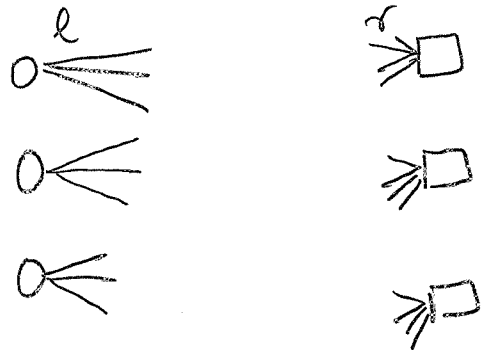
To summarize,

- the messages for binary channels can be compressed to a single real quantity
- if we choose this quantity to be log-likelihood, then the processing is very simple
  - * at variable nodes add $\sum_{k=1}^{K} \ell_k$
  - * at check nodes use $\blacksquare$

Low-density Parity-check Codes

LDPC codes are linear codes that have at least one sparse Tanner graph. The primary reason for focusing on such codes is that they have very good performance under message-passing decoding.

Definition: (l, r)-regular LDPC code is a binary linear code such that every variable node has degree l and every check node has degree r.



The number of edges in the Tanner graph of a (l, r)-regular LDPC code is ln, where n is the length of the code.

As n increases, the number of edges in Tanner graph grows linearly in n. This is in contrast to other linear codes (among class of all linear codes) where edge grow square of code length

The behavior of LDPC codes can be significantly improved by allowing nodes of different degrees. An <u>irregular LDPC code</u> is an LDPC code for which the degrees of nodes are chosen according to some distribution.

Assume that the LDPC has length n and that the number of variable nodes of degree i is Λ_i, so that $\sum_i \Lambda_i = n$.

Analogously, denote the number of check nodes of degree i by ρ_i,

$$\sum_i \rho_i = n(1-R)$$

R: Rate

Since the total edge counts must match up

$$\sum_i i\Lambda_i = \sum_i i\rho_i$$

Convenient of work with polynomials

$$\Lambda(x) = \sum_{i=1}^{l_{max}} \Lambda_i x^i$$

$$P(x) = \sum_{i=1}^{r_{max}} \rho_i x^i$$

Note $\Lambda(1) = n$, $P(1) = n(1-R)$ } $R(\Lambda, P) = 1 - \frac{P(1)}{\Lambda(1)}$

$\Lambda'(1) = P'(1)$

$\Lambda(x)$: variable degree distribution (from a node perspective)

$P(x)$: check degree distribution (")

Sometimes their normalized versions are more convenient

$$L(x) = \frac{\Lambda(x)}{\Lambda(1)} \qquad R(x) = \frac{P(x)}{P(1)}$$

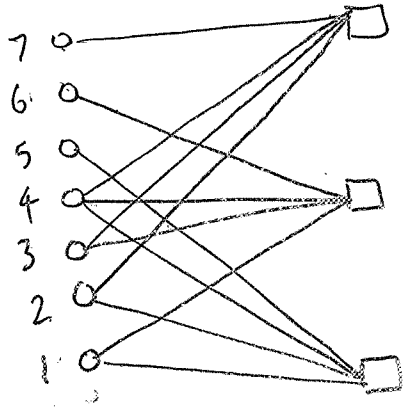
Example : [7,4] Hamming code

All check nodes have degree 4

3 variable nodes have degree 1

3 variable nodes have dg 2

1 variable node have dg 3



$\Lambda(x) = 3x + 3x^2 + x^3$ $\qquad P(x) = 3x^4$

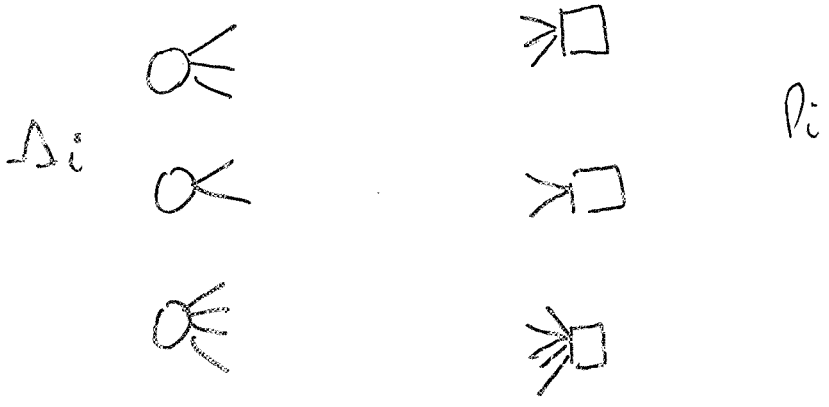
$L(x) = \frac{3}{7}x + \frac{3}{7}x^2 + \frac{1}{7}x^3$ $\qquad P(x) = x^4$

Definition: The Standard Ensemble $LDPC(\Lambda, P)$

Given a degree distribution pair (Λ, P), define an ensemble of bipartite graphs $LDPC(\Lambda, P)$ in the following way.

Each graph in $LDPC(\Lambda, P)$ has $\Lambda(1)$ variable nodes and $P(1)$ check nodes such that Λ_i variable nodes and P_i check nodes have degree i.

A node of degree i has i sockets from which the i edges emanate, so that in total there are $\Lambda'(1) = P'(1)$ sockets on each side.



Label the sockets on each side with the set $[\Lambda'(1)] = \{1, \ldots, \Lambda'(1)\}$ in some arbitrary & fixed way. Let σ be the permutation on $[\Lambda'(1)]$.

Associate to σ a bipartite graph by connecting i^{th} socket on the variable side on the $\sigma(i)$-th socket on the check side.

Let σ run over all possible permutations. This generates a set of bipartite graphs.

Finally, define a probability distribution over the set of graphs by placing a uniform distribution on the set of permutations.

This is the ensemble of bipartite graphs $LDPC(\Lambda, P)$. [In the random graph lit, this is known as configuration model]

Now we can associate a code with each graph. The parity check matrix H has a 1 on i-th row and j^{th} column if i-th check node is connected to j-th variable node.

NOTE: Our graphs are labeled (they have labeled sockets) and have a uniform distribution. The induced codes are unlabeled and the pdf on codes is not necessarily a uniform one.

$\text{Rate}(\Lambda, \rho) = 1 - \frac{\rho(1)}{\Lambda(1)}$ is called the design rate, because the parity check matrices in the standard ensemble can have linearly dependent rows. However as $n \to \infty$, actual rate of any random element is close to design rate with high probability.

Theorem: Consider $LDPC(\Lambda, \rho)$. Let $R(\Lambda, \rho)$ be the design rate, and $R(G)$ actual rate of code $G \in LDPC(\Lambda, \rho)$. Then under some conditions on Λ, ρ (see richardson & urbanke, pg 80), there exists $n > n(\xi)$ for $\xi > 0$

$$\text{Prob}\left[R(G) - R(\Lambda, \rho) > \xi\right] \leq e^{-n\xi \frac{\Lambda'(1)}{2}}$$

CONCENTRATION

Rather than analyzing individual codes, it suffices to assess the ensemble average performance.

This makes a lot of sense since every element of the ensemble behave with high probability close to the ensemble average.

Theorem (Concentration around ensemble average)

Let G, chosen uniformly at random from $LDPC(\Lambda,\rho)$ of length n, be used for transmission over $BEC(\epsilon)$. Assume that the decoder performs ℓ rounds of message-passing decoding and let $P_b(G,\epsilon,\ell)$ denote the resulting bit erasure probability. Then for ℓ fixed and for any $\delta>0$, there exists an $\alpha>0$, $\alpha=\alpha(\Lambda,\rho,\epsilon,\delta,\ell)$ such that

$$\text{Prob}\left[\left|P_b(G,\epsilon,\ell)-\mathop{\mathbb{E}}_{G'\in LDPC(\Lambda,\rho)}[P_b(G',\epsilon,\ell)]\right|>\delta\right]\le e^{-\alpha n}$$

Thus all except an exponentially small fraction of codes behave within an arbitrary small δ from the ensemble average.

It's not just about dmin!

We know we can guarantee correction of $t \approx \frac{d_{min}}{2}$ errors. So coding theorists spent decades searching for codes with large dmin. And most decoding methods were designed to correct t errors and no more. They are called bounded distance decoders.

Ideally, we will like to have a codebook where t-spheres around codewords fill up the whole space

Recall such codes are called perfect codes. If we could find perfect codes, we will love to use them. However there are almost no perfect codes. The only nontrivial perfect codes are

- Hamming codes (single error correcting, $t=1$)
- Repetition codes of odd blocklength N
- binary Golay code ($N=23$, $t=3$)

There are no other binary perfect codes. In fact, there are almost no 'near-perfect' codes either. In fact the picture of Hamming spheres of size t filling up the whole space is misleading.

For most codes (good or bad), almost all the Hamming space is taken up by the space between t-spheres (shaded portion in the picture on the page before).

Why is perfectness unattainable?

Shannon proved that for binary symmetric channel with any noise ϵ, there exists codes with large block-length N and rate as close as you like to capacity $C(\epsilon) = 1 - H_2(\epsilon)$, that

$$[H_2(\epsilon) = \epsilon \log \frac{1}{\epsilon} + (1-\epsilon) \log \frac{1}{(1-\epsilon)}]$$

can attain arbitrarily small error probability.

For large block length N, the number of errors per block will typically be about ϵN.

$\Rightarrow$ Shannon's codes are

'almost-certainly-ϵN-error-correcting'

Now let's pick a noisy channel

$$\epsilon \in (1/3, 1/2)$$

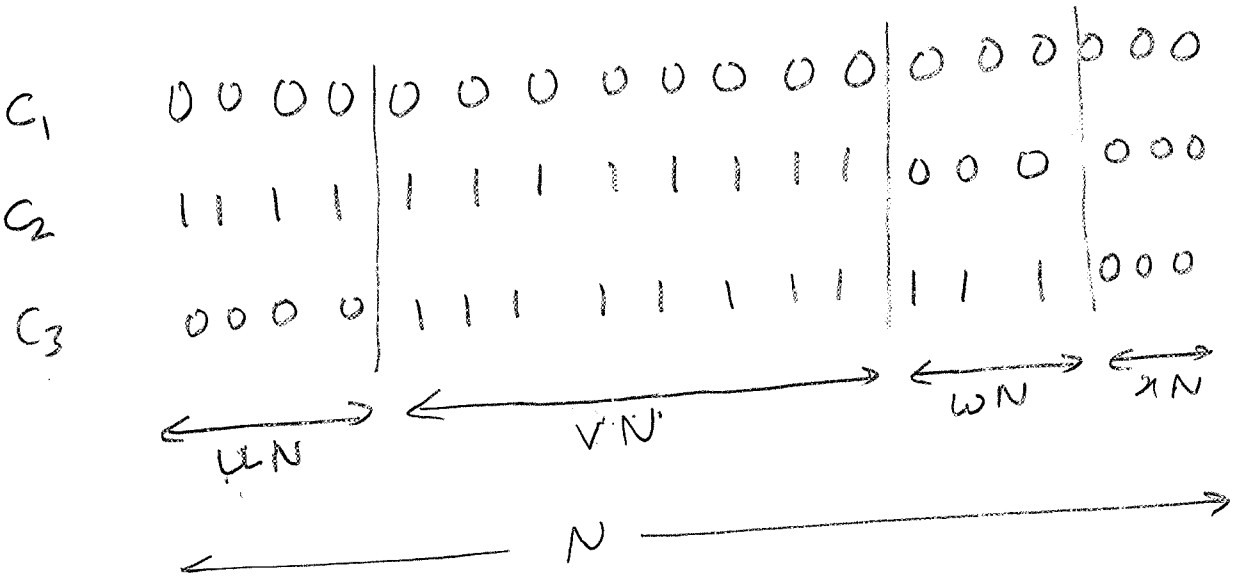
and try to find a large perfect code that is ϵN-error-correcting.

Assume such a code has been found ☺

- Examine three of its codewords

[Recall for rate $R = 1 - H_2(\epsilon) - \delta$, we have an enormous # of codewords 2^{NR}]

- Let one of the codewords be all zero and other two as shown below



$$\frac{d(c_2, c_1)}{N} = u + v$$

$$\frac{d(c_2, c_3)}{N} = u + w$$

$$\frac{d(c_1, c_3)}{N} = v + w$$

fraction χ of coordinates ... is zero on all codewords.

Now if the codebook is ϵN-error-correcting, its minimum distance must be $> 2\epsilon N$.

$$\Rightarrow \quad \left.\begin{aligned} U+V &> 2\epsilon \\ U+W &> 2\epsilon \\ V+W &> 2\epsilon \end{aligned}\right\} \text{ since the code is perfect}$$

Sum the 3 inequalities and divide by 2

$$U+V+W > 3\epsilon$$

So if $\epsilon > 1/3$, we get

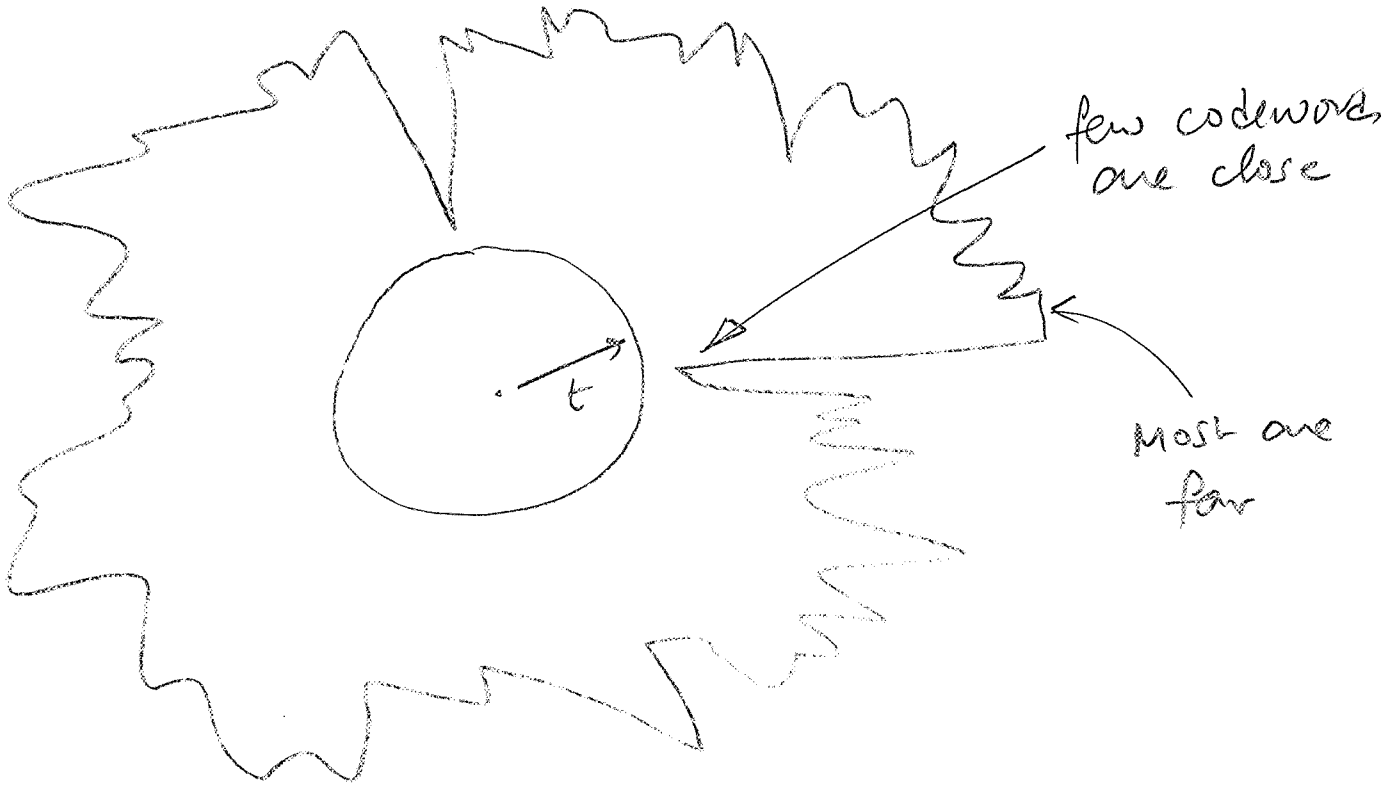
$$U+V+W > 1 \quad \Rightarrow x < 0$$

Impossible

$\Rightarrow$ the code cannot even 3 codewords, let alone 2^{NR}!

While Shannon proved many good codes exist, there are almost no perfect code among them.

In fact, for large N, the space around a codeword to which it is closest to looks something like this



In terms of weight distribution of code

$$A(x) = A_{d_{min}} x^{d_{min}} + A_{d_{min}+1} x^{d_{min}+1} + A_{d_{min}+2} x^{d_{min}+2} + \dots$$

A_i for small i is also small, most codewords are far from each other

LDPC codes have poor minimum distance but good weight distribution (distance spectrum), which is important at low SNR (high ϵ).