Codes on Graphs

Telecommunications Laboratory
Alex Balatsoukas-Stimming

Technical University of Crete

November 27th, 2008

Outline

1. Marginalization of a Function
   - Factor Graphs
   - The Sum-Product Algorithm
2. Codes on Graphs
   - The Iverson Function
   - Graph of a Code
   - Decoding on a Graph: Using the Sum-Product Algorithm
3. LDPC Codes
   - Desirable Properties
   - Construction of LDPC Codes

Symbol MAP decoding

- Assume that we want to minimize the error probability of a single symbol of the code word. In this case, we must do symbol-MAP decoding.
- The rule is:

$$\hat{x}_i = \arg\max_{x_i} p(x_i|\mathbf{y})$$

where

$$p(x_i|\mathbf{y}) = \sum_{\mathbf{x} \in C_i(x_i)} p(\mathbf{x}|\mathbf{y})$$

- The above is a marginalization problem.

Marginalization (1/2)

- We associate with a function $f(x_1, x_2, \ldots, x_n)$ its n *marginals*, defined as follows:

$$f_i(x_i) = \sum_{x_1} \cdots \sum_{x_{i-1}} \sum_{x_{i+1}} \cdots \sum_{x_n} f(x_1, x_2, \ldots, x_n)$$

- For each value of x_i, these are obtained by summing the function f over all of its arguments consistent with x_i.
- A more convenient notation for the above is:

$$f_i(x_i) = \sum_{\sim x_i} f(x_1, \ldots, x_n)$$

Marginalization (2/2)

- If $x_i \in \mathcal{X}$, then the complexity of the above summation grows as $|\mathcal{X}|^{n-1}$
- If f can be factored as a product of functions, the computation can be simplified.
- For example, consider the function:

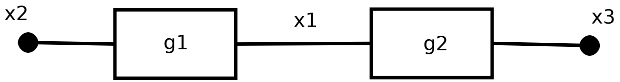
$$f(x_1, x_2, x_3) = g_1(x_1, x_2) g_2(x_1, x_3)$$

- The marginal $f_1(x_1)$ can be computed as:

$$\begin{aligned} f_1(x_1) &= \sum_{\sim x_1} f(x_1, x_2, x_3) = \sum_{x_2} \sum_{x_3} g_1(x_1, x_2) g_2(x_1, x_3) \\ &= \sum_{x_2} g_1(x_1, x_2) \cdot \sum_{x_3} g_2(x_1, x_3) \end{aligned}$$

Factor Graphs (1/2)

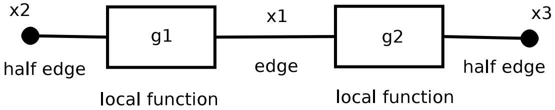
- The above procedure can be graphically represented with a factor graph:



- The nodes are viewed as processors which compute a function whose arguments label the incoming edges.
- The edges are channels by which these processors exchange data.

Factor Graphs (2/2)

- Every factor corresponds to a unique node, and every variable to a unique edge or half edge.
- The factors g_i are called *local functions* or *constraints*.
- The function f is called the *global function*.
- Edges connect nodes, half edges connect variables with nodes.
- A cycle of length λ is a path that includes λ edges and closes back on itself. The *girth* of a graph is the minimum cycle length of the graph.
- In a *normal* factor graph, no variable appears in more than two factors.

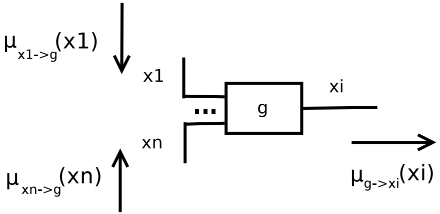


The Sum-Product Algorithm (1/3)

- We will now introduce an algorithm for the efficient computation of the marginals of a function described as a (normal) factor graph.
- This algorithm works when the graph is cycle-free, and yields the marginal function corresponding to each variable associated with an edge.
- The Sum-Product Algorithm is a message passing algorithm, because at each iteration, messages are passed along the edges of the graph.

The Sum-Product Algorithm (2/3)

- Consider the node representing the factor $g(x_1, \dots, x_n)$ of a global function $f(x_1, \dots, x_m)$



- The message passed along the edge corresponding to x_i is:

$$\mu_{g \to x_i}(x_i) = \sum_{\sim x_i} g(x_1, \dots, x_n) \prod_{\lambda \neq i} \mu_{x_\lambda \to g}(x_\lambda)$$

which is the product of g and all messages towards g along all edges except x_i, summed over all the variables except x_i.

The Sum-Product Algorithm (3/3)

- The messages $\mu_{x_j \to g}(x_j)$ are either the values of x_j, if we have a half edge, or the message coming from the node at the other end of the edge.
- The marginal of the global function with respect to x_i, is given by the product of all messages exchanged by the SPA over the edge corresponding to x_i:

$$f_{x_i}(x_i) = \prod_j \mu_{g_j \to x_i}(x_i)$$

The Sum-Product Algorithm (Example)

- Consider a burglar alarm which is sensitive not only to burglary, but also to earthquakes. There are 3 binary variables: a, b, e (for *alarm*, *burglary*, and *earthquake* respectively). A value of 1 indicates that the corresponding event has occured. We want to infer the probability of the two possible causes, given that the the alarm went off:

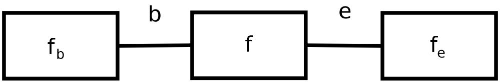
$$p(b|a = 1) \text{ and } p(e|a = 1)$$

- These can be computed by marginalizing $p(b, e|a = 1)$
- Asumming that b, e are independent, we have:

$$(b, e|a = 1) = \frac{p(a = 1|b, e)p(b)p(e)}{p(a = 1)} \propto p(a = 1|b, e)p(b)p(e)$$

The Sum-Product Algorithm (Example)

- So, we have factored the function we want to marginalize and the corresponding factor graph is:



where

$$f_b(b) \triangleq p(b) \qquad f_e(e) \triangleq p(e) \qquad f(b,e) \triangleq p(a=1|b,e)$$

- We are given the following data:

$$\begin{array}{ll} f_b(0) = 0.9 & f_b(1) = 0.1 \\ f_e(0) = 0.9 & f_e(1) = 0.1 \end{array}$$

and

$$\begin{array}{ll} f(0,0) = 0.001 & f(1,0) = 0.368 \\ f(0,1) = 0.135 & f(1,1) = 0.607 \end{array}$$

The Sum-Product Algorithm (Example)

- The messages from nodes f_b and f_e to node f will be:

$$\mu_{f_b \to b}(b) = (0.9, 0.1)$$
$$\mu_{f_e \to e}(e) = (0.9, 0.1)$$

- Once node f has received the above messages, it can compute the messages for nodes f_b and f_e:

$$\begin{aligned}
\mu_{f \to b}(b) &= \sum_e f(b, e)\mu_{f_e \to e}(e) \\
&= (\underbrace{\underbrace{0.001 \times 0.9}_{e=0} + \underbrace{0.135 \times 0.1}_{e=1}}_{b=0}, \underbrace{\underbrace{0.368 \times 0.9}_{e=0} + \underbrace{0.607 \times 0.1}_{e=1}}_{b=1}) \\
&= (0.0144, 0.3919)
\end{aligned}$$

The Sum-Product Algorithm (Example)

- Similarly, we can compute $\mu_{f \to e}(e) = (0.0377, 0.1822)$
- The marginals sought can now be written as:

$$p(b|a=1) \propto \mu_{f_b \to b}(b) \cdot \mu_{f \to b}(b) = (0.01296, 0.03919)$$

and

$$p(e|a=1) \propto \mu_{f_e \to e}(e) \cdot \mu_{f \to e}(e) = (0.03393, 0.01822)$$

- After scaling of these vectors so the sum of their elements is 1, we have:

$$p(b|a=1) = (0.249, 0.751)$$
$$p(e|a=1) = (0.651, 0.349)$$

Codes on Graphs

The Iverson Function

- Let P denote a proposition that may be either true or false.
- The Iverson function is defined as:

$$[P] = \begin{cases} 1, & \text{P is true} \\ 0, & \text{P is false} \end{cases}$$

- If we have n propositions, we have the factorization:

$$[P_1 \text{ and } P_2 \text{ and } \ldots \text{ and } P_n] = [P_1][P_2]\ldots[P_n]$$

Graph of a Code (1/4)

- Consider the parity-check matrix:

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

  of a $(7, 4, 3)$ Hamming code.
- All codewords satisfy the following parity checks:

$$\begin{aligned} x_1 + x_4 + x_6 + x_7 &= 0 \\ x_2 + x_4 + x_5 + x_6 &= 0 \\ x_3 + x_5 + x_6 + x_7 &= 0 \end{aligned}$$

Graph of a Code (2/4)

- Using the Iverson function, we can express membership of a codeword $\mathbf{x}$ in the code as follows:

$$[\mathbf{x} \in \mathcal{C}] = [\mathbf{H}\mathbf{x}^T = \mathbf{0}]$$

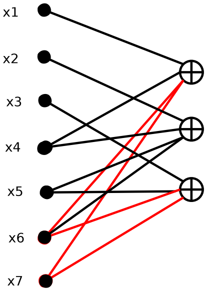
- In our case, the above function can be factored as follows:

$$[\mathbf{x} \in \mathcal{C}] = [x_1 + x_4 + x_6 + x_7 = 0][x_2 + x_4 + x_5 + x_6 = 0][x_3 + x_5 + x_6 + x_7 = 0]$$

- A *Tanner* graph is a graphical representation of a linear block code corresponding to the set of parity checks that specify the code.
- Each symbol (variable node) is represented by a filled circle ($\bullet$), and every parity check by a check node ($\oplus$).

Graph of a Code (3/4)

- Tanner graphs are bipartite, meaning that variable nodes can only be connected to check nodes, and vice versa.
- A Tanner graph may contain cycles, but since they are bipartite, their minimum girth is 4.
- For the Hamming code we defined above, the corresponding Tanner graph will be:



Graph of a Code (4/4)

- Since a given code can be represented by several parity-check matrices, the same code can be represented by several Tanner graphs. Some representations may have cycles while others may be cycle-free.
- Each variable node ($\bullet$) corresponds to one bit of the codeword, i.e. to one column of $\mathbf{H}$
- Each check node ($\oplus$) corresponds to one parity check equation, i.e. to one row of $\mathbf{H}$
- A connection between variable node j and check node i only exists if $\mathbf{H}_{ij} = 1$

Decoding on a Graph: Using the Sum-Product Algorithm (1/2)

- Consider the symbol-MAP decoding problem stated earlier:

$$p(\mathbf{x}|\mathbf{y}) \propto p(\mathbf{y}|\mathbf{x})p(\mathbf{x})$$

- For a stationary memoryless channel, we have:

$$p(\mathbf{y}|\mathbf{x}) = \prod_{i=1}^{n} p(y_i|x_i)$$

- Using the Iverson function and assuming that the a priori distribution of codewords is uniform, we can write:

$$p(\mathbf{x}) = [\mathbf{x} \in \mathcal{C}]\frac{1}{|\mathcal{C}|}$$

Decoding on a Graph: Using the Sum-Product Algorithm (2/2)

- From the above we get that:

$$p(\mathbf{x}|\mathbf{y}) \propto [\mathbf{x} \in \mathcal{C}] \prod_{i=1}^{n} p(y_i|x_i)$$

  which is in a factored form, so it can be represented by a normal factor graph.

- To compute $p(x_i|y_i)$ we can marginalize the above function using the sum-product algorithm on the graph.

The Sum-Product Algorithm on Graphs with Cycles

- On a cycle-free graph, the SPA yields the exact APP distribution of the code word symbols in a finite number of steps.
- Codes whose Tanner graphs are cycle-free have a rather poor performance.
- On a graph with cycles, the algorithm may not converge, or it may converge to a wrong result.
- If short cycles are avoided, in most practical cases, the algorithm does converge and yields the correct answer.
- For LDPC codes it is proved that as n grows asymptotically large, the assumption of a graph without cycles holds.

LDPC Codes

Low-Density Parity-Check Codes (1/2)

- LDPC codes are long linear block codes. As the name implies, their parity-check matrix has a low density of non-zero entries.
- Specifically, for a *regular* LDPC code, **H** contains a small number of 1s in each column, denoted w_c, and a small number of 1s in each row, denoted w_r.
- For irregular LDPC codes, the values of w_c and w_r are not constant.
- Since each column corresponds to one bit of the codeword, w_c tells us in how many parity check equations that bit participates.
- Accordingly, since each row corresponds to a parity check equation, w_r tells us how many bits participate in each equation.
- If the block length is n, we say that **H** characterizes a $\langle n, w_c, w_r \rangle$ LDPC code.

Low-Density Parity-Check Codes (2/2)

- If there are m parity check equations, each involving w_r bits, and each of the n coded symbols participates in w_c equations, it must hold that:

$$nw_c = mw_r \Leftrightarrow m = \frac{nw_c}{w_r}$$

  where m is the number of rows in $\mathbf{H}$.
- If $\mathbf{H}$ is full rank, then the rate of the code is:

$$\frac{n-m}{n} = 1 - \frac{w_c}{w_r}$$

  which yields the constraint $w_c \leq w_r$
- The actual rate of the code might be higher than the above, if the parity checks are not independent. We call $\rho^* \triangleq 1 - w_c/w_r$ the *design rate* of the code.

Desirable Properties

- The Tanner graph corresponding to the code should have a large girth, for good convergence properties of the iterative decoding algorithm.
- Regularity of the code eases implementation.
- For good performance at high SNR on the AWGN channel, the minimum Hamming distance must be large. LDPC codes are known to achieve a large value of $d_{H_{min}}$
- The techniques for the design of parity-check matrices of LDPC codes can be classified under two main categories:
  1. *Random* constructions.
  2. *Algebraic* constructions.

Random Constructions (1/2)

- They are based on generating the parity-check matrix randomly filled with 0s and 1s, while satisfying some constraints.
- After selecting values for the parameters n, ρ^*, w_c, we can compute the value of w_r.
- For a regular code, row and column weights of $\mathbf{H}$ must be exactly w_r and w_c respectively.
- Additional constraints can be included, e.g. the number of 1s in common between any two columns (or rows) should not exceed one, in order to avoid length-4 cycles.

Random Constructions (2/2)

- A method for the random construction of $\mathbf{H}$ was developed by Gallager:
- The parity-check matrix $\mathbf{H}$ of a regular $\langle n, w_c, w_r \rangle$ LDPC code has the form:

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_1 \\ \mathbf{H}_2 \\ \vdots \\ \mathbf{H}_{w_c} \end{bmatrix}$$

- $\mathbf{H}_1$ has n columns and n/w_r rows, contains a single 1 in each column, and contains 1s in its i-th row from column $(i-1)w_r + 1$ to column iw_r.
- All other matrices are obtained by randomly permuting the columns of $\mathbf{H}_1$.

- For example, for $\rho^* = 1/2$, $w_c = 2$ and $n = 12$, we have:

$$\rho^* = 1 - \frac{w_c}{w_r} \Rightarrow w_r = 4$$

- Using the method mentioned above, we have:

$$\mathbf{H}_1 = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

- The column weight of $\mathbf{H}_1$ is 1, the row weight is w_r and the matrix is $n/w_r \times n$.
- We will need $w_c - 1 = 1$ permutation of this matrix to create $\mathbf{H}$.

Random Constructions (Example)

- One possible permutation is:

$$\mathbf{H}_2 = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

- So, the final matrix will be:

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_1 \\ \mathbf{H}_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Algebraic Constructions (1/2)

- Algebraic LDPC codes are more easily decodeable than random codes.
- A simple algebraic construction works as follows: choose $p > (w_c - 1)(w_r - 1)$ and consider the $p \times p$ matrix obtained from the identity matrix $\mathbf{I}_p$ by cyclically shifting its rows by one position to the right:

$$\mathbf{J} = \begin{bmatrix} 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 \\ 1 & 0 & 0 & 0 & \dots & 0 \end{bmatrix}$$

- The λ-th power of $\mathbf{J}$ is obtained from $\mathbf{I}_p$ by cyclically shifting its rows by $(\lambda \mod p)$ positions to the right.

- Construct the matrix:

$$\mathbf{H} = \begin{bmatrix} \mathbf{J}^0 & \mathbf{J}^0 & \mathbf{J}^0 & \dots & \mathbf{J}^0 \\ \mathbf{J}^0 & \mathbf{J}^1 & \mathbf{J}^2 & \dots & \mathbf{J}^{w_r-1} \\ \mathbf{J}^0 & \mathbf{J}^2 & \mathbf{J}^4 & \dots & \mathbf{J}^{2(w_r-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{J}^0 & \mathbf{J}^{w_c-1} & \mathbf{J}^{2(w_c-1)} & \dots & \mathbf{J}^{(w_c-1)(w_r-1)} \end{bmatrix}$$

  where $\mathbf{J}^{\mathbf{0}} = \mathbf{I}_p$

- This matrix has $w_c p$ rows and $w_r p$ columns. The number of 1s in each row and column is exactly w_r and w_c respectively.
- It can be proven that no length-4 cycles are present.