4.6 Behavioral Models of Codes and Factor Graph Representation

It is well known that turbo codes, LDPC codes, and repeat-accumulate (RA) codes can approach Shannon limit very closely. A common feature of these capacity-approaching coding schemes is that they all may be understood as *codes defined on graphs*.

In this section, we will introduce the subject of codes on graphs. As we will see, (factor) graphs provide a means of visualizing the constraints that define the code. Moreover, the graphs directly specify iterative decoding algorithms.

4.6.1 Codes and Behavioral Modeling

Let $\mathbb{F}_q^n$ denote the vector space of all *n*-tuples over a finite field $\mathbb{F}_q$. We know that a linear (n, k, d) block code can be represented by several methods:

- By a set of *k* generators $\{\mathbf{g}_j, 1\le j\le k\}$. The code $\mathcal{C} = \{\mathbf{x}\in\mathbb{F}_q^n \mid \mathbf{x} = \sum_j a_j\mathbf{g}_j, a_j\in\mathbb{F}_q\}$.
- By a set of $(n\text{-}k)$ generators $\{\mathbf{h}_j, 1\le j\le n\text{-}k\}$ for the dual code. The code $\mathcal{C}$ is then $\mathcal{C} = \{\mathbf{x}\in\mathbb{F}_q^n \mid <\mathbf{x},\mathbf{h}_j>=0 \text{ for all } j\}$
- By a trellis representation. The code $\mathcal{C}$ is then the set of all *n*-tuples corresponding to paths through the trellis.

In the following we will see that these representations are all special cases of a general class of representations called *behavioral realizations*.

We will use the following notation. A symbol variable X_i takes values $x_i\in\mathcal{X}_i$ in a symbol alphabet $\mathcal{X}_i$. In coding, $\mathcal{X}_i$ is often a vector space over a finite field, e.g., $\mathbb{F}_q^m$, and x_i are the corresponding *m*-tuples. A *symbol configuration space* $\mathcal{X}$ is a Cartesian product

$$\mathcal{X} = \prod_{i\in I}\mathcal{X}_i$$

of a collection $\{\mathcal{X}_i, i\in\mathcal{I}\}$ of symbol alphabets, where $\mathcal{I}$ is any discrete index set (called *symbol index set*). The elements of $\mathcal{X}$ are denoted by $\mathbf{x} = \{x_i\in\mathcal{X}_i, i\in\mathcal{I}\}\in\mathcal{X}$, and will be called *symbol configurations*. In other words, a configuration is a particular assignment of values to all variables. The configuration space is the set of configurations.

For example, if all variables in Fig. 4.6.1 are binary, the configuration space $\mathcal{X}$ is the set $\{0, 1\}^5$ of all binary 5-tuples; if all variables in Fig. 4.6.1 are real, the configuration space is $\mathbb{R}^5$.

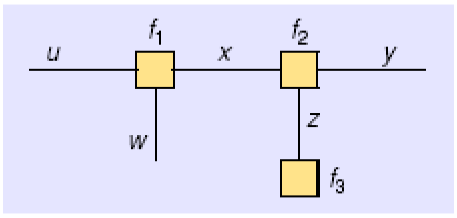


Figure 4.6.1 Illustration of the functions of several variables. A node f_i is connected with the edge representing some variable x iff f_i is a function of x.

By a *behavior* in $\mathcal{X}$, we mean any subset $\mathcal{B}$ of $\mathcal{X}$ (that is, a set of symbol configurations that satisfy a certain set of constraints). The elements of $\mathcal{B}$ are called *valid symbol configurations*. Since a system is specified via its behavior $\mathcal{B}$, this approach is known as *behavioral modeling*.

Behavioral modeling is natural for codes. A *code* $\mathcal{C}$ is a behavior in $\mathcal{X}$, and the valid symbol configurations are called *codewords*.

Whereas in system theory the index set $\mathcal{I}$ is usually ordered and regarded as a time axis, here $\mathcal{I}$ will not necessarily be ordered. We will also assume that $\mathcal{I}$ is finite; i.e., that $\mathcal{C}$ is a block code.

4.6.2 Behavioral Realizations

From the discussion above, a code $\mathcal{C} \subseteq \mathcal{X}$ may be characterized as the set of configurations $\mathbf{x} \in \mathcal{X}$ that satisfy a certain set of constraints. For example, a linear code $\mathcal{C}$ may be characterized as the set of $\mathbf{x} \in \mathcal{X}$ that satisfy a certain set of parity checks. Such a representation is now referred to as "behavioral", since it is specified by local constraints as in the behavioral system theory of Willems.

Formally, a *behavioral realization* of a code $\mathcal{C} \subseteq \mathcal{X}$ is described by a set $\{\mathcal{C}_k, k \in \mathcal{K}\}$ of *local constraints* ("local codes," "local behaviors"), where $\mathcal{K}$ is another discrete index set. Each local constraint $\mathcal{C}_k$ involves some subset of the symbol variables indexed by a certain subset $\mathcal{I}(k)$, and defines a subset

$$\mathcal{C}_k \subseteq \mathcal{X}(k) = \prod_{i\in\mathcal{I}(k)} \mathcal{X}_i$$

of the corresponding local Cartesian product set $\mathcal{X}(k)$.The local constraint thus defines a set of *valid local configurations* (“local codewords”) $\mathbf{x}_{|\mathcal{I}(k)} = \{x_i, i \in \mathcal{I}(k)\} \in \mathcal{C}_k$, where the notation $\mathbf{x}_{|\mathcal{I}(k)}$ denotes the projection of a configuration $\mathbf{x}$ onto the symbol variables X_i indexed by $\mathcal{I}(k)$. The code $\mathcal{C}$ is then the set of all configurations that satisfy all local constraints

$$\mathcal{C} = \left\{\mathbf{x} \in \mathcal{X} \mid \mathbf{x}_{|\mathcal{I}(k)} \in \mathcal{C}_k \text{ for all } k \in \mathcal{K}\right\}$$

For example, a linear code $\mathcal{C} \subseteq \mathcal{X}$ may be characterized as the set of $\mathbf{x}\in\mathcal{X}$ that satisfy the parity-check equations $<\mathbf{x}, \mathbf{h}_k> = 0$ for a certain set of check configurations $\{\mathbf{h}_k \in \mathcal{X},\ k \in \mathcal{K}\}$. The symbol variables X_i that are involved in the kth check are those for which $h_{ki} \neq 0$. Each local code $\mathcal{C}_k$ is then a linear (n_k, n_k-1, 2) *single-parity-check* (SPC) code, whose length n_k is equal to the number of symbols involved in $\mathcal{C}_k$.

A behavioral realization has a nature graphical model, which in coding theory is called a Tanner graph.

4.6.3 Tanner Graph

The above elementary linear behavioral realizations can be represented conveniently by a graphical model called Tanner graph.

<u>Graph Notation</u>

- A *graph G* is a triple consisting of a vertex set $V(G)$, an edge set $E(G)$, and a relation Φ that associate with each edge two vertices called its endpoints.

If vertex $v\in V$ is an endpoint of edge $e\in E$, then v and e are incident. The degree of a vertex $v\in V$ is the number of incident edges and will be denoted by $d(v)$.

- A graph G is called *regular* when all of its vertices have degree r; i.e., $\forall v \in V:\ d(v) = r$.
- *Bipartite Graph*: A bipartite graph is a graph whose vertices may be partitioned into two sets, and where edges may only connect two vertices not residing in the same set. A more precise definition is given below.

定义：若图 G 的节点集V 可以划分为两个子集V_1和V_2： $V_1 \cup V_2 = V, V_1 \cap V_2 = \phi$，使得

$\forall e \in E$，e 的一个端点属于 V_1，另一个端点属于 V_2，则称 G 为**二部图**(bipartite graph)。若节点集 V_1 中所有节点的度数都相同并且节点集 V_2 中所有节点的度数也都相同，则称 G 为**正则二部图**(regular bipartite graph)，否则称为**非正则二部图**(irregular bipartite graph)。

- A cycle is a subgraph $C = (V', E')$ of $G = (V, E)$, whose vertices can be placed around a circle. The length of a cycle C is defined as $l(C) = |V'| = |E'|$.
- The girth of G is the length of its shortest cycle: $g(G) = \min_{C \in \mathcal{C}} \{l(C)\}$, where $\mathcal{C}$ is the collection of all the cycles of G.

A *Tanner graph* is a bipartite graph in which a first set of vertices represents the symbol variables $\{X_i, i \in \mathcal{I}\}$, a second set of vertices represents the local constraints $\{\mathcal{C}_k, k \in \mathcal{K}\}$, and an edge connects a variable vertex to a constraint vertex if and only if (iff) the corresponding symbol variable is involved in the corresponding local constraint. Fig. 4.6.2 shows a Tanner graph for the (8, 4, 4) RM code defined by the parity-check matrix

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{4.1}$$

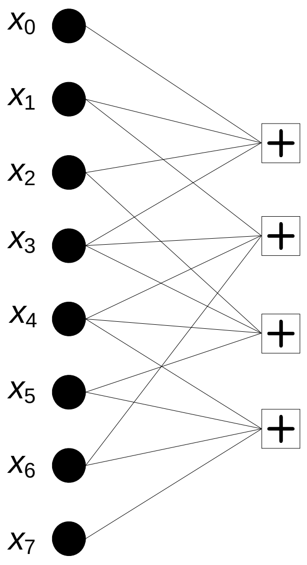


Fig. 4.6.2 Tanner graph of parity-check representation for (8, 4, 4) code.

Here, symbol variables are represented by filled circles, and constraints (checks) by squares labeled by a ‘+’ sign. The four check nodes (vertices) represent the binary linear equations that each codeword must satisfy. In a valid codeword, the neighbors of every check node (i.e., the variables connected to the check by a single edge) must form a configuration with a binary sum of zero. Notice that this Tanner graph contains circles.

The degree of a vertex is defined as the number of edges incident on it. Thus, the *degree of a variable node* is the number of local constraints (i.e., checks) that it is involved in, and the *degree of a constraint node* is the number of variables that it involves. Clearly, in a Tanner graph the sum of the variable degrees is equal to the sum of the constraint degrees since both are equal to the number of edges in the graph.

The degree of a variable or constraint will be defined as the degree of the corresponding graph vertex.

In Fig. 4.6.2, the local constraints consist of the following 4 parity-check equations (linear homogeneous equations).

$$\begin{aligned} x_0 + x_1 + x_2 + x_3 &= 0 \\ x_1 + x_3 + x_4 + x_6 &= 0 \\ x_2 + x_3 + x_4 + x_5 &= 0 \\ x_4 + x_5 + x_6 + x_7 &= 0 \end{aligned}$$

4.6.4 General Linear Behavioral Realizations of Linear Block Codes [1][5]

The above elementary realizations can be generalized by letting symbol variable to vector spaces of dimension *m* over $\mathbb{F}_q$, or more particularly *m*-tuples over $\mathbb{F}_q$. The elements of a general linear behavioral realization of a linear (*n, k, d*) block code over $\mathbb{F}_q$ are as follows.

- A set of symbol m_i-tuple $\{x_i \in \mathbb{F}_q^{m_i}, i \in \mathcal{I}\}$, where $\mathcal{I}$ denotes the (clustered) symbol variable index set. We define $n = \sum_{i\in\mathcal{I}} m_i$.
- A set of state variable μ_j-tuple $\{s_j \in \mathbb{F}_q^{\mu_j}, j \in \mathcal{J}\}$, where $\mathcal{J}$ denotes the state index set. We define $s = \sum_{j\in\mathcal{J}} \mu_j$.
- A set of (locally) linear constraint codes $\{\mathcal{C}_k, k\in\mathcal{K}\}$ over $\mathbb{F}_q$, where $\mathcal{K}$ denotes the constraint index set, and each code $\mathcal{C}_k$ involves a certain subset of the symbol and state variables, indexed by $\mathcal{I}(k)$ and $\mathcal{J}(k)$, respectively.

The symbol configuration space is the Cartesian product $\mathcal{X} = \prod_{i\in\mathcal{I}} \mathcal{X}_i$, where the symbol alphabets $\mathcal{X}_i = \mathbb{F}_q^{m_i}, i \in \mathcal{I}$. The state configuration space is the Cartesian product $\mathcal{S} = \prod_{j\in\mathcal{J}} \mathcal{S}_j$, where the state alphabets (state spaces) $\mathcal{S}_j = \mathbb{F}_q^{\mu_j}, j \in \mathcal{J}$. A local configuration is a set

$\left\{\left(x_i, i \in \mathcal{I}(k)\right), \left(s_j, j \in \mathcal{J}(k)\right)\right\} = \left(\mathbf{x}_{|\mathcal{I}(k)}, \mathbf{s}_{|\mathcal{J}(k)}\right)$ of local variable values. The set of all local configurations is a vector space

$$\mathcal{V}_k = \left(\prod_{i \in \mathcal{I}(k)} \mathcal{X}_i\right) \times \left(\prod_{j \in \mathcal{J}(k)} \mathcal{S}_j\right)$$

A linear local constraint code is a subspace $\mathcal{C}_k \subseteq \mathcal{V}_k$ whose codewords are precisely the valid local configurations which may actually occur.

The full behavior $\mathfrak{B} \subseteq \mathcal{X} \times \mathcal{S}$ generated by the realization is the set of all combinations (**x**, **s**) (called trajectories) of symbol and state configurations that satisfy all local constraints

$$\mathfrak{B} = \left\{(\mathbf{x}, \mathbf{s}) \in \mathcal{X} \times \mathcal{S} \mid (\mathbf{x}_{|\mathcal{I}(k)}, \mathbf{s}_{|\mathcal{J}(k)}) \in \mathcal{C}_k \ , \ k \in \mathcal{K}\right\}$$

The code $\mathcal{C}$ generated by the realization is the set of all symbol sequences **x** that appear in any trajectory (**x**, **s**)$\in\mathfrak{B}$. In other words, $\mathcal{C} = \mathfrak{B}_{|\mathcal{X}}$.

Notice that the constraint imposed by a linear homogeneous equation that involves $r = n - k$ variables is equivalent to a constraint that these variables must lie in a certain (r-1)-dimensional subspace of $\mathbb{F}_q^r$; i.e., that they form a codeword in a (r, r-1) linear block code over $\mathbb{F}_q$.

<u>*Example:*</u> Consider a conventional state-space realization of $\mathcal{C}$ on the time axis $\mathcal{I} = [0, n)$.

A trellis diagram is a detailed graphical model of a conventional state-space realization. We define state space $\mathcal{S}_j = \mathbb{F}_q^{\mu_j}$ of dimension μ_j for the state time axis $\mathcal{J}=[0, n]$, where $\mu_0 = \mu_n = 0$; i.e., the starting and ending state space have size $|\mathcal{S}_0|=|\mathcal{S}_n|=1$. We then define each trellis section by a linear subspace (called branch space) $\mathcal{C}_i \subseteq \mathcal{S}_i \times \mathcal{S}_{i+1} \times \mathbb{F}_q$, $i \in \mathcal{I}$, which defines the set of state transitions $(s_i, s_{i+1}) \in \mathcal{S}_i \times \mathcal{S}_{i+1}$ that can possibly occur and the code symbol $x_i \in \mathcal{X}_i = \mathbb{F}_q$ associated with each such transition. In other words, the behavioral constraint at time i is that the tripe (s_i, s_{i+1}, x_i) must lie in a certain small linear block code $\mathcal{C}_i$ of length $n_i = \mu_i + \mu_{i+1} + 1$ over $\mathbb{F}_q$. Thus, each state variable S_i is involved in two constraints, $\mathcal{C}_i$ and $\mathcal{C}_{i-1}$, while each symbol variable X_i is involved in the single constraint $\mathcal{C}_{i-1}$. Each valid local configuration

$(s_i, s_{i+1}, x_i) \in \mathcal{C}_i$ is a set of variables that satisfy the constraints at time *i* and corresponds to a distinct valid trellis branch labeled by the corresponding (s_i, s_{i+1}, x_i), so the branch complexity at time *i* is the size $|\mathcal{C}_i|$ of the local constraint code $\mathcal{C}_i$.

The full behavior $\mathfrak{B}$ of this realization is then the set of all state/symbol sequences (**s**, **x**) such that $(s_i, s_{i+1}, x_i) \in \mathcal{C}_i$, $1 \le i \le n$, which is a set of linear homogeneous constraints. For each valid configuration (**s**, **x**) in $\mathfrak{B}$, the state sequence **s** represents a valid path through the code trellis, and the symbol sequence **x** represents the corresponding codeword. The code generated by the trellis diagram is the set of all path label sequences, namely, the projection $\mathcal{C}=\mathfrak{B}_{|\mathcal{X}}$.

As an example, Fig. 4.6.3 shows the trellis diagram for the (8, 4, 4) RM code. It is known that, for the (8,4,4) code, the minimal state complexity profile of any trellis diagram is given by $(|\mathcal{S}_0|,|\mathcal{S}_1|,|\mathcal{S}_2|,|\mathcal{S}_3|,|\mathcal{S}_4|,|\mathcal{S}_5|,|\mathcal{S}_6|,|\mathcal{S}_7|,|\mathcal{S}_8|) = (1,2,4,8,4,8,4,2,1)$. The state spaces at time i=1,2,… may be defined by their $\mathbb{F}_2$ components as follows: $s_1=(s_{11})$, $s_2=(s_{21}, s_{22})$, $s_3=(s_{31}, s_{32}, s_{33})$,….

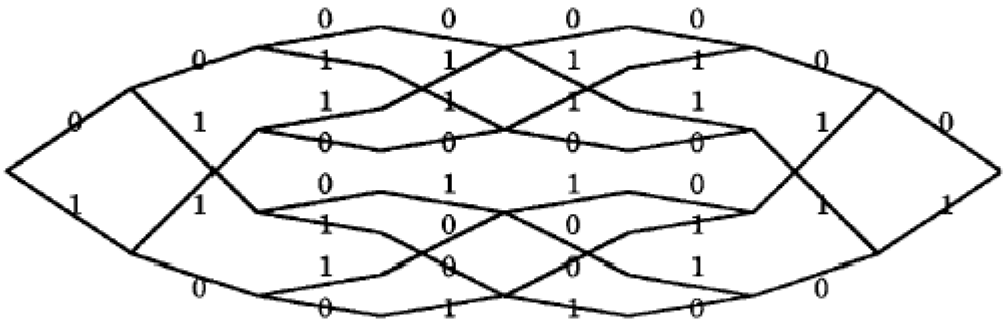


Figure 4.6.3 The trellis diagram for the (8, 4, 4) code.

The main difficulty with trellises and other cycle-free graph representations of codes is that as codes become more powerful, the alphabets (state space) of the state variables must necessarily become exponentially large, which eventually makes trellis-type decoding algorithms impractical.

4.6.5 Graphs of General Linear Behavioral Representations – Factor Graph

There are various styles for drawing graphs of general linear behavioral representations. We start with *generalized Tanner graphs* (now called *factor graphs*). A factor graph is a Tanner graph that may contain auxiliary (state) variables. In a factor graph, two types of vertices represent variable *m*-tuples and constraint codes, respectively. Again, an edge connects a variable vertex to a constraint vertex if and only if the corresponding variable is involved in the corresponding constraint code.

A generic factor graph is shown in Fig. 4.6.4, where symbol *m*-tuples are represented by filled circles, state μ-tuples by open circles, and constraints by squares. However, the squares may now be labeled to denote various types of constraint codes.

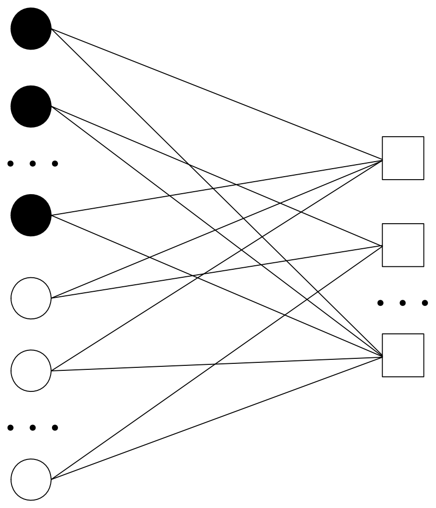


Figure 4.6.4 A factor graph

Fig.4.6.5 is a factor graph of the trellis realization of (8,4,4) code. Each state variable is labeled by the dimension of its alphabet. Each constraint code $\mathcal{C}_i$ is labeled by its length and dimension (*n*, *k*), where $n = |\mathcal{S}_i| + |\mathcal{S}_{i+1}| + 1$ and $k = \log_2 |\mathcal{C}_i|$. Since the symbol variables have degree 1, we use the special "dongle" symbol for them.

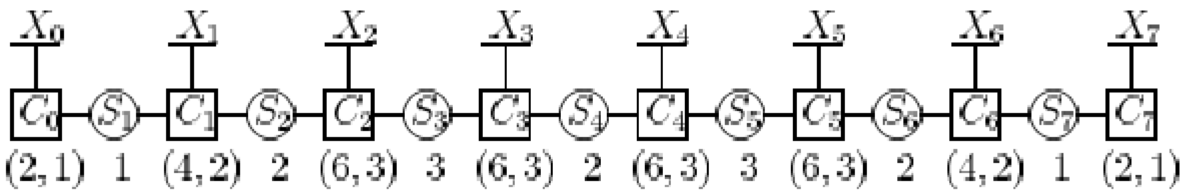


Figure 4.6.5 Factor graph for the trellis realization of (8, 4, 4) code

4.6.6 The Forney-Style Factor Graphs –- Normal Graphs

Normal graphs were proposed by Forney in 1998, in which constraint codes are represented by vertices, but variables are represented by edges (if the variable has degree 2) or by hyper edges (if the variable has degree other than 2).

Normal graphs are particularly nice when the realizations are *normal realizations*. We define a realization to be normal if the degree of every symbol variable is 1, and the degree of every state variable is 2. We then represent symbol variables as before by "half-edges" using the special "dongle" symbols, and state variables by ordinary edges. Fig.4.6.6 shows a normal graph for trellis representation of (8, 4, 4) code. It is seen that normal graph may be simpler.

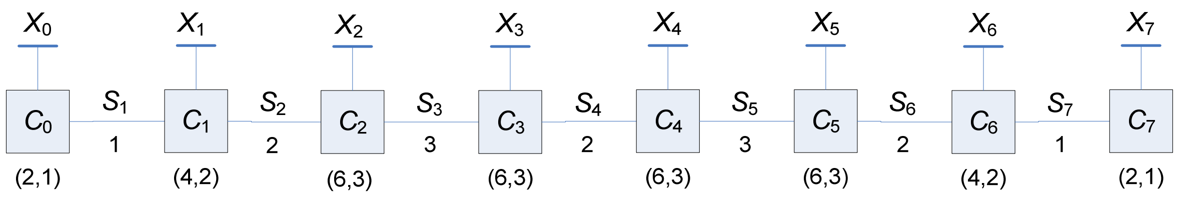


Figure 4.6.6 Normal graph for trellis representation of (8, 4, 4) code

Any realization may be transformed into a normal realization by simple conversion shown in Fig. 4.6.7. The conversion from a factor graph to a normal graph involves symbol replications and state replications, as shown in Fig. 4.6.8. In the normal graph, all graph vertices represent constraints, and the repetition constraints (or, equality constraints) are represented by vertices labeled by the symbol "=".

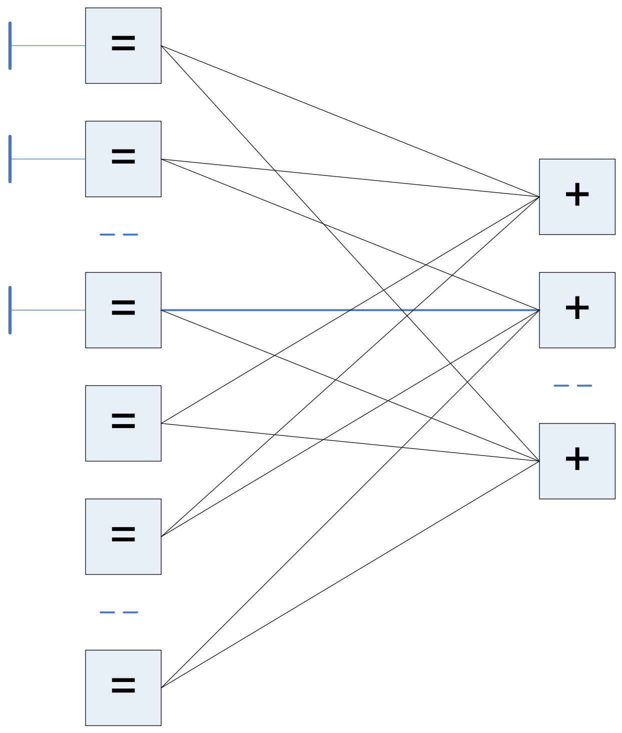


Figure 4.6.7 Normal graph with observed variables (represented by "dongles"), equality constraints and zero-sum constraints (represented by squares with "+").

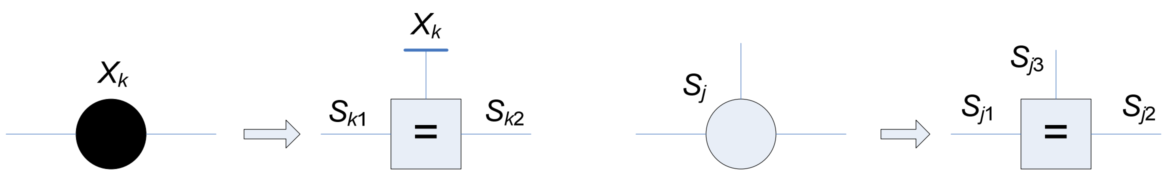


Figure 4.6.8

As an example, Fig. 4.6.9 shows the normal graph of the (8, 4, 4) code defined in (4.1).

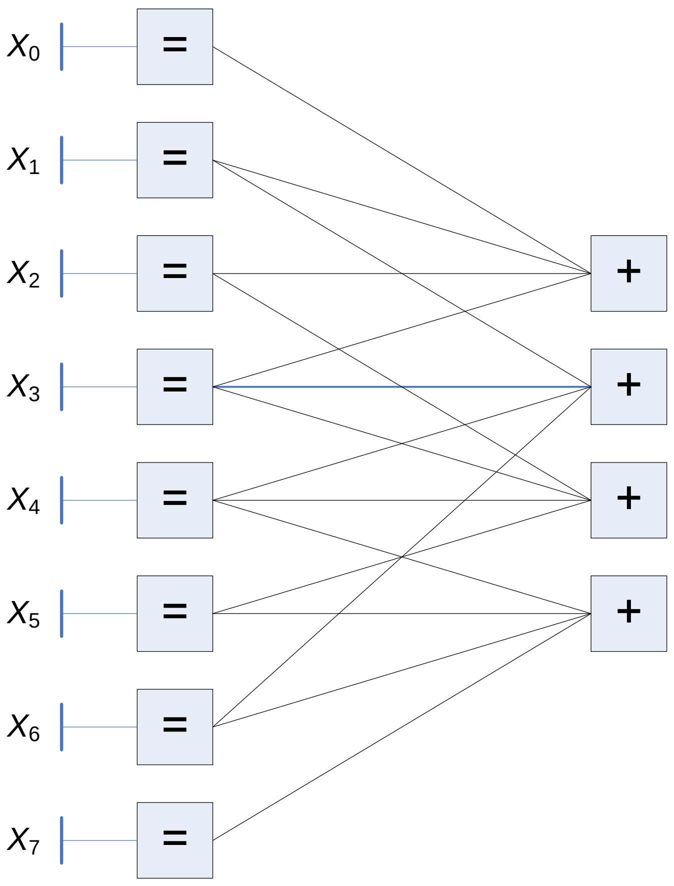


Figure 4.6.9 Normal graph for parity-check realization of (8, 4, 4) code

Normal graphs have some conceptual advantages over Tanner graphs.

1) Clean functional separation:
   - symbol variables (half edges) are for I/O;
   - state variables (edges) are for communication (message-passing);
   - constraints (vertices) are for computation.
2) Block diagrams as directed normal graphs (i.e., compatible with standard block diagrams).
3) Whereas a factor graph is bipartite, a normal graph has only one kind of vertices, and there are no restrictions on graph topology.
4) Simplest formulation of the sum-product message update rule.
5) Suited for hierarchical modeling (“boxes within boxes”).
6) Natural setting for Forney’s results on Fourier transforms and duality.

Therefore, normal graph provides an attractive notation for modeling a wide variety of information transmission and signal processing problems.

Fig. 4.6.10 depicts a normal graph of a binary (7, 4, 3) Hamming code that is defined by the parity-check matrix

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{4.2}$$

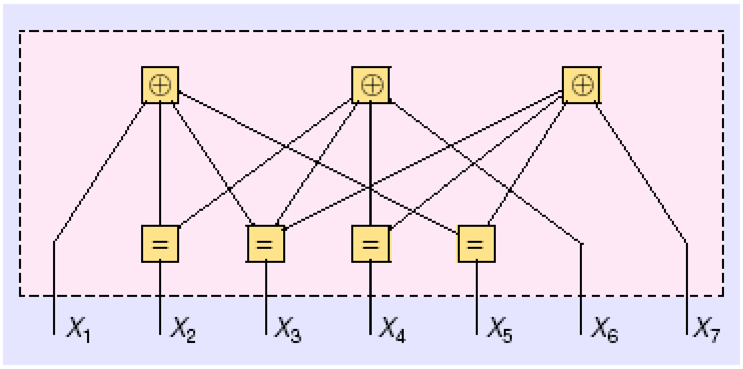


Figure 4.6.10 Normal graph for parity-check realization of (7, 4, 3) Hamming code.

Fig. 4.6.11 shows a normal graph of a turbo code.

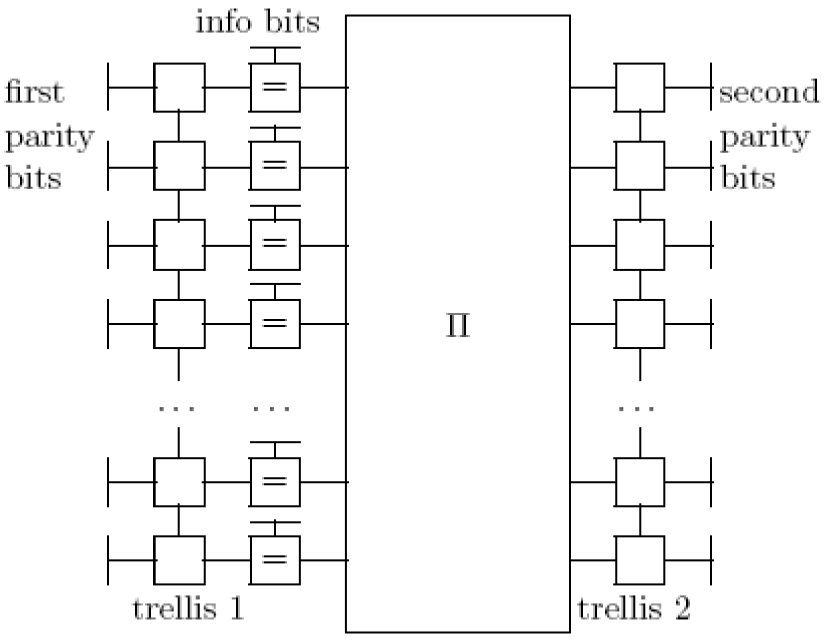


Fig. 4.6.11 Normal graph of a classical turbo code

■ A More General Description from a Factorization Perspective

Normal graph is also called the Forney-style factor graph (FFG). The term “factor graph” results from the fact that an FFG is a diagram that represents the factorization of a function of several variables. Assume, for example, that some function $f(u, w, x, y, z)$ can be factored as

$$f(u,w,x,y,z) = f_1(u,w,x)\, f_2(x,y,z)\, f_3(z) \tag{4.3}$$

This factorization is expressed by the FFG shown in Fig. 4.6.12. The factors are also called local functions, and their product is called the global function. In (4.3), the global function is f, and f_1, f_2, f_3 are the local functions. In general, an FFG is defined by the following rules:

- There is a (unique) node for every factor.
- There is a (unique) edge or half edge for every variable.
- The node representing some factor g is connected with the edge (or half edge) representing some variable x if and only if g is a function of x.

Implicit in this definition is the assumption that no variable appears in more than two factors. This restriction can be easily circumvented by using the repetition constraint that corresponds to, e.g., the factor $f(x,x',x'') \equiv \delta(x-x')\delta(x-x'')$, where $\delta(X)$ is the Kronecker delta function if X is a discrete variable or the Dirac delta if X is a continuous variable. (The distinction between these two cases is usually obvious in concrete examples.). For example, the factorization $f(x) = f_1(x)f_2(x)f_3(x)$ can be expand into $f(x) = f_1(x)\, f_2(x')f_3(x'')\delta(x\text{-}x')\delta(x\text{-}x'')$. See Fig. 4.6.12b.

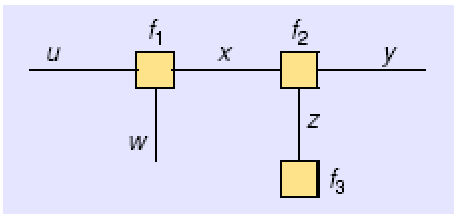


Figure 4.6.12 An FFG

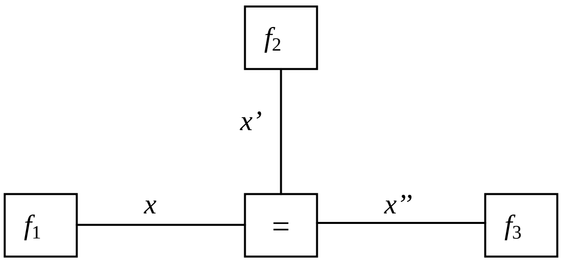


Figure 4.6.12b

We will primarily consider the case where f is a function from the configuration space, $\mathcal{X}$, to the set of nonnegative real numbers, $\mathbb{R}^+$. In this case, a configuration $\mathbf{x} \in \mathcal{X}$ will be called valid if $f(\mathbf{x}) \neq 0$.

In every fixed configuration $\mathbf{x} \in \mathcal{X}$, every variable has some definite value. We may therefore consider also the variables in a factor graph as functions with domain (定义域)$\mathcal{X}$. Using the standard notation for random variables, we will denote such functions by capital letters. Therefore, if $\boldsymbol{y}$ takes values in some set $\mathcal{Y}$, we will write

$$Y : \mathcal{X} \to \mathcal{Y} : \mathbf{x} \mapsto \mathbf{y} = Y(\mathbf{x})$$

■ Graphs of Codes

By a factor graph for some code $\mathcal{C}$, we mean a factor graph for (some factorization of) the membership indicator function of $\mathcal{C}$. Denote the Iverson function by

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

The membership indicator function $I_C : \mathbb{F}_q^n \to \{0,1\}$ that expresses the membership of an n-tuple $\mathbf{x}$ in $\mathcal{C}$ is given by

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

Consider, for example, the binary (7, 4, 3) Hamming code shown in Fig. 4.5.10. The membership indicator function of this code may be written as

$$I_C\left[(x_1,...,x_7) \in \mathcal{C}\right] = I_C[x_1 \oplus x_2 \oplus x_3 \oplus x_5 = 0] \cdot I_C[(x_2 \oplus x_3 \oplus x_4 \oplus x_6 = 0 \cdot I_C[x_3 \oplus x_4 \oplus x_5 \oplus x_7 = 0]$$

or $I_C(x_1,...,x_7) = \delta(x_1 \oplus x_2 \oplus x_3 \oplus x_5) \cdot \delta(x_2 \oplus x_3 \oplus x_4 \oplus x_6) \cdot \delta(x_3 \oplus x_4 \oplus x_5 \oplus x_7)$ (4.4)

where $\oplus$ denotes modulo-2 addition. Note that each factor in (4.4) corresponds to one row of the parity check matrix in (4.2). From (4.4), we obtain the FFG of Fig. 4.6.10.

Example 2 [Trellis code]: $I_C\left[\mathbf{x} \in \mathcal{C}\right] = \prod_{k=1}^{n} I_C\left[x_k \in \mathcal{C}_k\right]$

<u>*FFG for Dual code*</u>

The dual code of a linear code $\mathcal{C}$ is $\mathcal{C}^\perp = \left\{\mathbf{y} \in \mathbb{F}_q^n \mid \left\langle \mathbf{y}, \mathbf{x}^T \right\rangle = \mathbf{0} \text{ for all } \mathbf{x} \in \mathcal{C}\right\}$. The FFG for the code $\mathcal{C}^\perp$ can be easily obtained via the following theorem [?] (which is a special case of a sweepingly general result on the Fourier transform of an FFG).

Duality Theorem for Binary Linear Codes: Consider an FFG for some binary linear code $\mathcal{C}$. Assume that the FFG contains only parity-check nodes and equality constraint nodes, and assume that all code symbols $x_1, \ldots, x_n$ are external variables (i.e., represented by half edges). Then an FFG for the dual code $\mathcal{C}^\perp$ is obtained from the original FFG by replacing all parity-check nodes with equality constraint nodes and vice versa.
For example, Fig. 4.6.15 shows an FFG for the dual code of the (7, 4, 3) Hamming code.

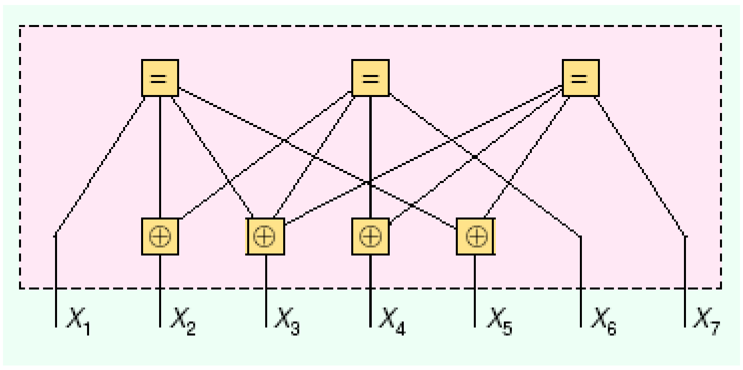


Figure 4.6.15 Dualizing Figure 4.6.10 yields an FFG for the dual code

4.6.7 Graph-Theoretic Properties of Realizations

A realization has certain graph-theoretic properties, such as connectedness or cycle-freedom.

■ <u>Connectedness and Independence</u>

A code $\mathcal{C}$ has a realization whose graph is disconnected if and only if $\mathcal{C}$ is the Cartesian product of shorter codes. In this case, the realization of $\mathcal{C}$ may be constructed from independent realizations of shorter (component) codes. Thus disconnectedness is a graph-theoretic expression of independence.

For example, if a code $\mathcal{C} = \mathcal{C}_1 \times \mathcal{C}_2$ is the Cartesian product of two codes; that is, $\mathcal{C} = \{(\mathbf{c}_1, \mathbf{c}_2) \mid \mathbf{c}_1 \in \mathcal{C}_1, \mathbf{c}_2 \in \mathcal{C}_2\}$. Then a realization of $\mathcal{C}$ may be constructed from independent realizations of $\mathcal{C}_1$ and $\mathcal{C}_2$. A graph of such a realization is a disconnected graph, with two component subgraphs representing $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively.

■ <u>Cut Sets and Conditional Independence</u>

A *cut set* of a connected graph is a minimal set of edges such that removal of the set partitions the graph into two disconnected subgraphs. Notice that *a connected graph is cycle-free if and only if every edge is by itself a cut set*.

In a normal graph, a cut set consists of a set of ordinary (state) edges, and may be specified by the corresponding subset $\mathcal{X} \subseteq \mathcal{J}$ of the state index set $\mathcal{J}$. The cut-set alphabet is then the Cartesian product $\mathcal{S}_{\mathcal{X}} = \prod_{j \in \mathcal{X}} \mathcal{S}_j$ with values $\mathbf{s}_{\mathcal{X}} = \{s_j, j \in \mathcal{X}\}$ and size $|\mathcal{S}_{\mathcal{X}}| = \prod_{j \in \mathcal{X}} |\mathcal{S}_j|$. Clearly, a superstate variable can be defined over this cut-set alphabet. Fig. 4.6.16 gives a high-level view of a realization with a cut set $\mathcal{X}$.

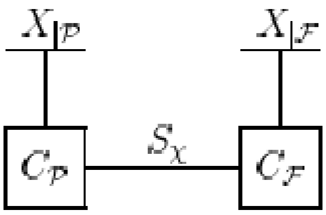


Figure 4.6.16 A realization with a cut set $\mathcal{X}$

Since removal of a cut set partitions a graph into two disconnected subgraphs, it follows that the symbol variables, the constraint codes, and the states not in χ are partitioned by the cut set into two disjoint subsets connected only by the states in χ. We label these two components arbitrarily as the "past" $\mathcal{P}$ and the "future" $\mathcal{F}$ relative to the cut set χ.

The constraints and internal variables in the past and future components are agglomerated into aggregate constraints $\mathcal{C}_{\mathcal{P}}$ and $\mathcal{C}_{\mathcal{F}}$, respectively. Let $X_{|\mathcal{P}}(\mathbf{s}_{\mathcal{X}})$ and $X_{|\mathcal{F}}(\mathbf{s}_{\mathcal{X}})$ denote the sets of possible past and future symbol values that are consistent with a given superstate value $\mathbf{s}_{\mathcal{X}}$. Then the code has the following decomposition:

$$\mathcal{C} = \bigcup_{\mathbf{s}_{\mathcal{X}} \in \mathcal{S}_{\mathcal{X}}} X_{|\mathcal{P}}(\mathbf{s}_{\mathcal{X}}) \times X_{|\mathcal{F}}(\mathbf{s}_{\mathcal{X}})$$

Cut-Set Independence Theorem [Markov property]***:*** Assume that an FFG represents the joint probability distribution (or the joint probability density) of several random variables. Assume further that the edges corresponding to some variables $Y_1, \ldots, Y_n$ form a cut-set of the graph. In this case, conditioned on $Y_1 = y_1, \ldots, Y_n = y_n$ (for any fixed $y_1, \ldots, y_n$), every random variable (or every set of random variables) in one component of the graph is independent of every random variable (or every set of random variables) in the other component.

4.7 The Sum-Product Algorithm

Graphical models are often associated with particular algorithms. For example, the Viterbi decoding algorithm is naturally described by means of a trellis diagram.

The sum-product algorithm (SPA) is the basic decoding algorithm for codes defined on factor graphs. For cycle-free graphs, it is finite and exact. Furthermore, because all its operations are local, it can also applied to graphs with cycles; then it becomes iterative and approximate, but it often works very well.

There are many variations and applications of the sum-product algorithm:

- BCJR/APP decoding algorithm (applied to a trellis)
- Statistical inference (Bayes network): belief propagation (BP) algorithm
- Gaussian state-space models: Kalman filter

4.7.1 The Distributive Law

Let $\mathbb{F}$ be a field and let a, b, $c \in \mathbb{F}$. The distributive law then states that

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

This simple axiom, properly applied, can lead to significant reductions in computational complexity. An obvious instance is

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

Let $A(x)$ and $B(y)$ be two functions with x and y taking values in (some Cartesian product of) $\mathbb{F}$. Then the above equation implies that

$$\sum_{x,y} A(x)B(y) = \left(\sum_x A(x)\right)\left(\sum_y B(y)\right)$$

It is at the heart of many fast algorithms including the sum-product algorithm.

4.7.2 The Sum-Product Algorithm on Forney-Style Factor Graphs **(SPA 的描述)**

The factor graph approach represents the appropriate framework in order to systematically take advantage of instances where the distributive law can be applied.

■ *The Basic Decoding Problem:*

Assume that we are transmitting over a memoryless channel using a (n, k) linear code $\mathcal{C}$ defined by its parity-check matrix $\mathbf{H} = [h_{ij}]$. The basic symbol-APP decoding problem can be stated as follows.

$$\begin{aligned}\hat{x}_i &= \arg\max_{x_i \in \{\pm 1\}} P_{X|Y}(x_i \mid \mathbf{y}) = \arg\max_{x_i \in \{\pm 1\}} \sum_{x_1} \cdots \sum_{x_{i-1}} \sum_{x_{i+1}} \cdots \sum_{x_n} P_{X|Y}(\mathbf{x} \mid \mathbf{y}) \\ &= \arg\max_{x_i \in \{\pm 1\}} \sum_{\sim x_i} P_{X|Y}(\mathbf{x} \mid \mathbf{y}) \\ &= \arg\max_{x_i \in \{\pm 1\}} \sum_{\sim x_i} p(\mathbf{y} \mid \mathbf{x}) P(\mathbf{x}) \\ &= \arg\max_{x_i \in \{\pm 1\}} \sum_{\sim x_i} \left(\prod_{j=1}^{n} p(y_j \mid x_j)\right) \frac{1}{|\mathcal{C}|} I_C[\mathbf{x} \in \mathcal{C}] \\ &= \arg\max_{x_i \in \{\pm 1\}} \sum_{\sim x_i} \left(\prod_{j=1}^{n} p(y_j \mid x_j)\right) \left(\prod_{l=1}^{n-k} I_C\left[\sum_{j=1}^{n} h_{lj} x_j = 0\right]\right)\end{aligned}$$

where $I_C(P)$ is the *code-membership indicator function* defined by

$$I_C[P] = \begin{cases} 1, & \text{if } P \text{ evaluates to true} \\ 0, & \text{otherwise} \end{cases}$$

The notation $\sum_{\sim x}$ denote a summation over all variables contained in the expression except the one listed (e.g., x). Clearly, the general decoding problem is equivalent to calculating the marginal of a factorized function. For example, if the code $\mathcal{C}$ is a binary linear code with parity-check matrix

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

then we have

$$\arg\max_{x_i \in \{\pm 1\}} P_{X|Y}(x_i \mid \mathbf{y})$$

$$= \arg\max_{x_i \in \{\pm 1\}} \sum_{\sim x_i} \left(\prod_{j=1}^{6} p(y_j \mid x_j) I_C[x_1 \oplus x_2 \oplus x_5 = 0] I_C[x_2 \oplus x_3 \oplus x_6 = 0] I_C[x_1 \oplus x_3 \oplus x_4 = 0] \right)$$

The corresponding factor graph is shown in Fig. 4.7.1.

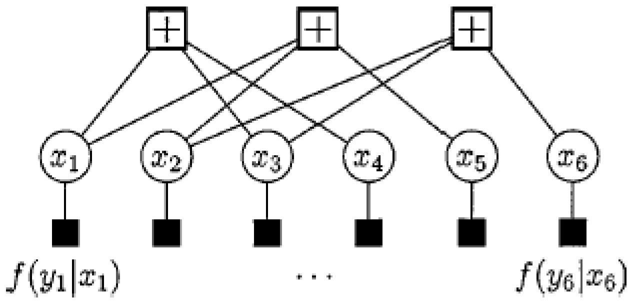


Figure 4.7.1

The *sum-product algorithm* is a procedure that can be used to organize the simultaneous computation of marginals. The sum-product algorithm operating in an FFG can be described as follows. Refer to Fig. 4.7.2.

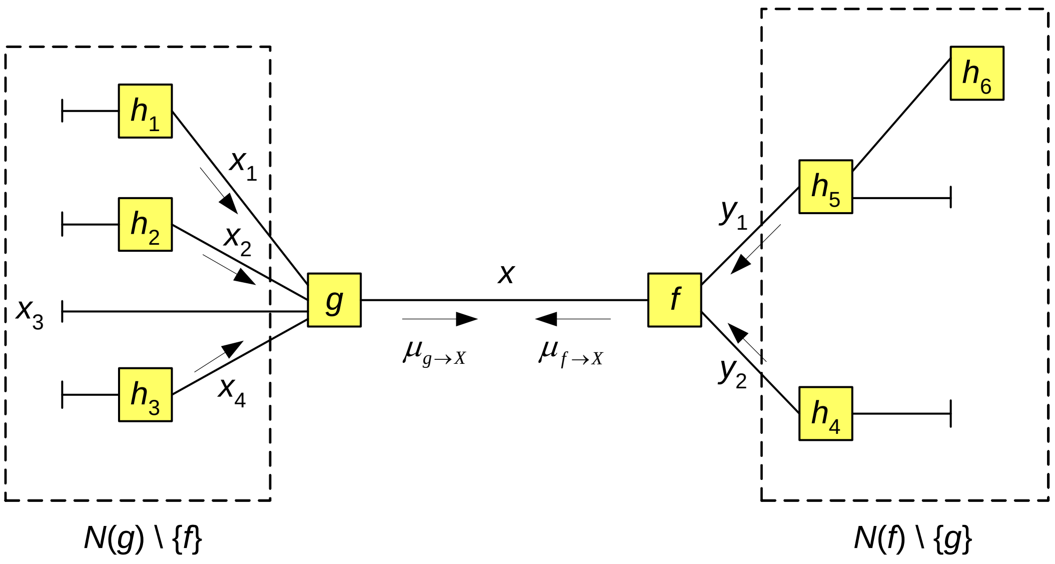


Figure 4.7.2 Update rules of the sum-product algorithm in an FFG

1) Regardless of the message direction, the message passed over an edge incident on a vertex representing local constraint is always a function over the alphabet on which the symbol or state variable X (associated with that edge) is defined. For example, suppose that X is a binary variable. Then messages passed on the edge corresponding to X will be functions of the form $\mu(x)$; such functions can be specified by vector $[\mu(0), \mu(1)]$, or, by the ratio $\log(\mu(0)/\mu(1))$. We will denote the message arriving at a node f along the edge X by $\mu_{X\to f}(x)$, and the message sent from a factor node f (to a factor node g) along the edge x by $\mu_{f\to X}(x)$. Let $N(v)$ denote the set of neighbors of a given vertex v in a factor graph, and the set $N(v)\backslash\{w\}$ denote the neighbors of v other than w. Besides, we use $E(v)$ to denote the set of edges that connect v to nodes in $N(v)$.

2) *Sum-Product Update rule*: The message $\mu_{f\to X}(x)$ sent by a factor node f to a neighbor node $g\in N(f)$ along the edge X is the function

$$\mu_{f\to X}(x)=\sum_{y_1}\cdots\sum_{y_m}\left(f(x,y_1,\dots,y_m)\cdot\mu_{Y_1\to f}(y_1)\cdots\mu_{Y_n\to f}(y_n)\right)$$

$$=\sum_{\sim x}\left(f(x,y_1,\dots,y_m)\prod_{Y\in E(f)\backslash\{X\}}\mu_{Y\to f}(y)\right) \tag{4.5}$$

This rule is at the heart of the whole sum-product algorithm, which can be stated as a more general rule as follows.

Summary-Product Rule: **The message out of a factor node *f(x, . . .)* along the edge *x* is the product of *f(x, . . .)* and all messages towards *f* along all edges except *x*, summarized over all variables except *x*.**

3) In the special case where the local constraint is a equality constraint, (4.5) becomes

$$\mu_f(x)=\prod_{X\in E(f)}\mu_{X\to f}(x)$$

which is sometimes called the product update rule. It gives the marginal function for x.

With the sum-product rule, the evaluation of some probability distribution function in a factor graph may be greatly simplified. As a simple example, consider the FFG shown in Fig.4.6.19. Let $f(x_1, \dots, x_8)$ be some discrete probability mass function, which can be written as

$$f(x_1,\dots,x_8)=\left(f_1(x_1)f_2(x_2)f_3(x_1,x_2,x_3,x_4)\right)$$
$$\cdot\left(f_4(x_4,x_5,x_6)f_5(x_5)\cdot\left(f_6(x_6,x_7,x_8)f_7(x_7)\right)\right) \tag{4.6}$$

Note that the brackets in (4.6) correspond to the dashed boxes in Figure 4.7.3.

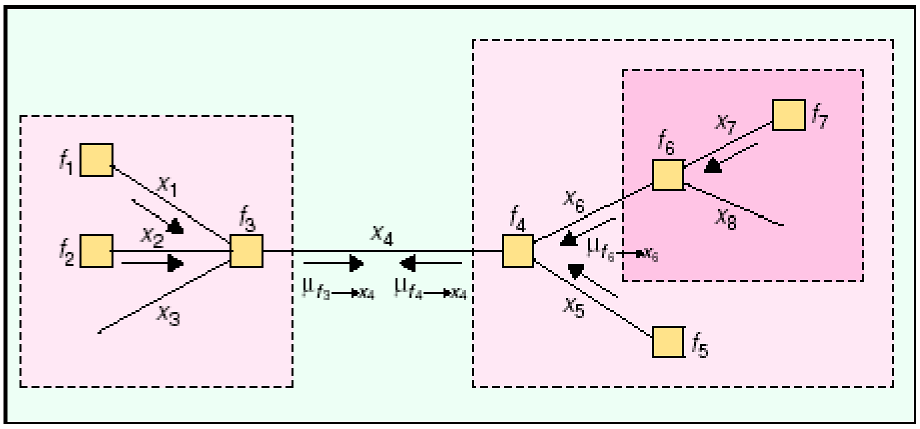


Figure 4.7.3 “Summarized” factors as messages in the FFG

Suppose that we now are interested in the marginal probability

$$p(x_4) = \sum_{\sim x_4} f(x_1, ..., x_8) \tag{4.7}$$

Inserting (4.6) into (4.7) and applying the distributive law yields

$$p(x_4) = \overbrace{\left(\sum_{x_1} \sum_{x_2} \sum_{x_3} f_3(x_1, x_2, x_3, x_4) f_1(x_1) f_2(x_2) \right)}^{\mu_{f_3 \to x_4}}$$
$$\cdot \underbrace{\left(\sum_{x_5} \sum_{x_6} f_4(x_4, x_5, x_6) f_5(x_5) \underbrace{\left(\sum_{x_7} \sum_{x_8} f_6(x_6, x_7, x_8) f_7(x_7) \right)}_{\mu_{f_6 \to x_6}} \right)}_{\mu_{f_4 \to x_4}}$$

This expression can be interpreted as “closing” the dashed boxes in Figure 4.7.3 by summarizing over their internal variables. The resulting expression

$$p(x_4) = \mu_{f_3 \to x_4}(x_4) \cdot \mu_{f_4 \to x_4}(x_4) \tag{4.8}$$

corresponds to the FFG of Figure 4.7.3 with the dashed boxes closed.

Table 1 shows the sum-product update rule for two typical local constraints that are the building blocks of low-density parity-check codes. It is quite popular to write these messages in terms of the single parameters

$$L_X \equiv \log \frac{\mu_X(0)}{\mu_X(1)}$$

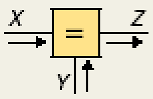
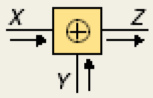
$$\text{or} \quad \Delta \equiv \frac{\mu(0)-\mu(1)}{\mu(0)+\mu(1)}$$

and the corresponding versions of the update rules are also given in Table 1.

For the decoding of LDPC codes, the typical update schedule alternates between updating the messages out of equality constraint nodes and updating the messages out of parity-check nodes.

Please refer to [2] for a detailed example. And a simple numerical example is provided in [4]. See the Box on pp.21.

Table 1. Sum-product message update rules for binary parity-check codes.

 $\delta(x-y)\delta(x-z)$	$\begin{pmatrix}\mu_Z(0)\\ \mu_Z(1)\end{pmatrix} = \begin{pmatrix}\mu_X(0)\,\mu_Y(0)\\ \mu_X(1)\,\mu_Y(1)\end{pmatrix}$ $\Delta_Z = \frac{\Delta_X+\Delta_Y}{1+\Delta_X\Delta_Y}$ $L_Z = L_X + L_Y$
 $\delta(x \oplus y \oplus z)$	$\begin{pmatrix}\mu_Z(0)\\ \mu_Z(1)\end{pmatrix} = \begin{pmatrix}\mu_X(0)\,\mu_Y(0)+\mu_X(1)\,\mu_Y(1)\\ \mu_X(0)\,\mu_Y(1)+\mu_X(1)\,\mu_Y(0)\end{pmatrix}$ $\Delta_Z = \Delta_X \cdot \Delta_Y$ $\tanh(L_Z/2) = \tanh(L_X/2)\cdot\tanh(L_Y/2)$

<u>A Summary</u>: In its general form, the sum-product algorithm computes two messages for each edge in the graph, one in each direction. Each message is computed according to the sum-product rule (4.5).

A sharp distinction divides graphs with cycles from graphs without cycles. If the graph has no cycles, then it is efficient to begin the message computation from the leaves and to successively compute messages as their required "input'' messages become available. In this way, each message is computed exactly once. It is then obvious from the previous section that summaries/marginals as in (4.7) can be computed as the product of messages as in (4.8) simultaneously for all variables.

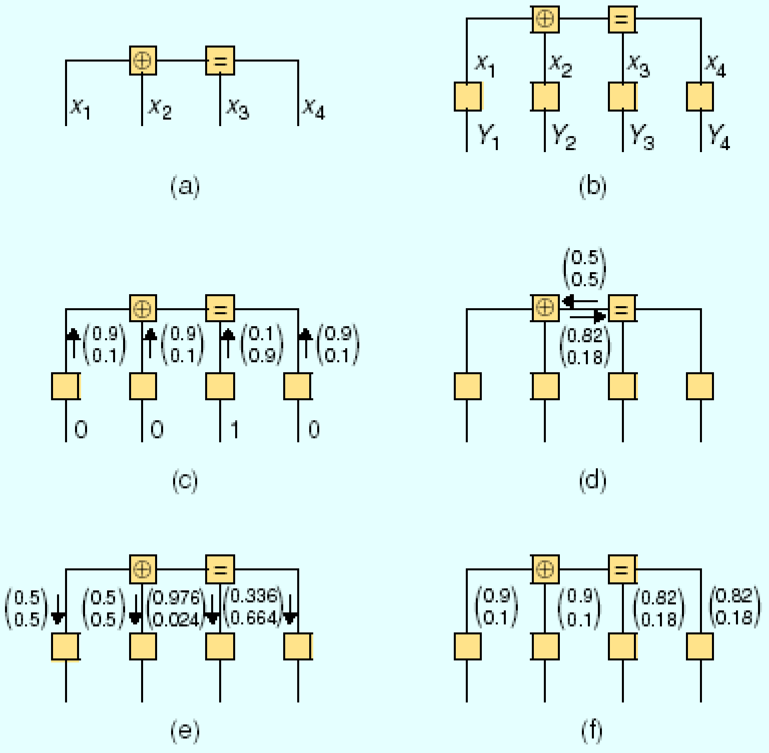
Sum-Product (Belief Propagation) Algorithm: An Example

Consider a simple binary code

$$C = \{(0,0,0,0), (0,1,1,1), (1,0,1,1), (1,1,0,0)\},$$

which is represented by the FFG in (a) below. Assume that a codeword $(X_1, \ldots, X_4)$ is transmitted over a binary symmetric channel with crossover probability $\varepsilon = 0.1$ and assume that $(Y_1, \ldots, Y_4) = (0, 0, 1, 0)$ is received. The figures below show the messages of the sum-product algorithm. The messages μ are represented as $\begin{pmatrix}\mu(0)\\ \mu(1)\end{pmatrix}$ scaled such that $\mu(0) + \mu(1) = 1$.

The final result in (f) is the a posteriori probability $p(x_\ell | y_1, \ldots, y_4)$ for $\ell = 1, \ldots, 4$.



⊕
=
X_1
X_2
X_3
X_4
(a)
Y_1
Y_2
Y_3
Y_4
(b)
$\begin{pmatrix}0.9\\0.1\end{pmatrix}$
$\begin{pmatrix}0.9\\0.1\end{pmatrix}$
$\begin{pmatrix}0.1\\0.9\end{pmatrix}$
$\begin{pmatrix}0.9\\0.1\end{pmatrix}$
0
0
1
0
(c)
$\begin{pmatrix}0.5\\0.5\end{pmatrix}$
$\begin{pmatrix}0.82\\0.18\end{pmatrix}$
(d)
$\begin{pmatrix}0.5\\0.5\end{pmatrix}$
$\begin{pmatrix}0.5\\0.5\end{pmatrix}$
$\begin{pmatrix}0.976\\0.024\end{pmatrix}$
$\begin{pmatrix}0.336\\0.664\end{pmatrix}$
(e)
$\begin{pmatrix}0.9\\0.1\end{pmatrix}$
$\begin{pmatrix}0.9\\0.1\end{pmatrix}$
$\begin{pmatrix}0.82\\0.18\end{pmatrix}$
$\begin{pmatrix}0.82\\0.18\end{pmatrix}$
(f)

(a) FFG of the code; (b) code/channel model; (c) computing messages ...; (d) computing messages ...; (e) computing messages ...; (f) a posteriori probabilities obtained from (c) and (e).

4.7.3 Principles of the Sum-Product Algorithm on Cycle-Free Normal Graphs （SPA 原理分析）[5]

We now develop the sum-product algorithm as an APP decoding algorithm for a code $\mathcal{C}$ that has a cycle-free normal graph realization. Assume that the code $\mathcal{C}$ is described by a realization involving a certain set of symbol variables $\{X_i, i \in \mathcal{I}\}$ of degree 1, a certain set of state variables $\{S_j, j \in \mathcal{J}\}$ of degree 2, and a certain set of constraint codes $\{\mathcal{C}_k, k \in \mathcal{K}\}$ such that the graph of the realization is cycle-free.

Assume that a set of independent observations are made on all symbol variables $\{X_i, i \in \mathcal{I}\}$, resulting in a set of observations $\mathbf{y}=\{y_i, i \in \mathcal{I}\}$ and likelihood vectors $\left\{w_i = \{p(y_i \mid x_i), x_i \in \mathcal{X}_i\}, i \in \mathcal{I}\right\}$, where $\mathcal{X}_i$ is the alphabet of X_i. The likelihood of a codeword $\mathbf{x} = \{x_i, i \in \mathcal{I}\} \in \mathcal{C}$ is then defined as the component-wise product

$$p(\mathbf{y} \mid \mathbf{x}) = \prod_{i \in \mathcal{I}} p(y_i \mid x_i)$$

Assuming equiprobable codewords, the *a posteriori* probabilities $\{P(\mathbf{x} \mid \mathbf{y}), \mathbf{x} \in \mathcal{C}\}$ can be expressed as

$$P(\mathbf{x} \mid \mathbf{y}) = \frac{p(\mathbf{y} \mid \mathbf{x}) P(\mathbf{x})}{p(\mathbf{y})} \propto p(\mathbf{y} \mid \mathbf{x}), \qquad \mathbf{x} \in \mathcal{C} \tag{4.9}$$

Let $\mathcal{C}_i(x_i)$ denote the subset of codewords that are consistent with x_i; i.e., whose the *i*th symbol variable X_i has the value $x_i \in \mathcal{X}_i$. Then the symbol APP is given up to a scale factor by

$$P(X_i = x_i \mid \mathbf{y}) = \sum_{\mathbf{x} \in \mathcal{C}_i(x_i)} P(\mathbf{x} \mid \mathbf{y}) \propto \sum_{\mathbf{x} \in \mathcal{C}_i(x_i)} p(\mathbf{y} \mid \mathbf{x}) = \sum_{\mathbf{x} \in \mathcal{C}_i(x_i)} \prod_{i' \in \mathcal{I}} p(y_{i'} \mid x_{i'}), \qquad x_i \in \mathcal{X}_i \tag{4.10}$$

Similarly, if $\mathcal{C}_j(s_j)$ denotes the subset of codewords that are consistent with the state variable S_j having the value s_j in the state alphabet $\mathcal{S}_j$, then the state APP vector $\{P(S_j = s_j \mid \mathbf{y}), s_j \in \mathcal{S}_j\}$ is given up to a scale factor by

$$P(S_j = s_j \mid \mathbf{y}) \propto \sum_{\mathbf{x} \in \mathcal{C}_j(s_j)} \prod_{i \in \mathcal{I}} p(y_i \mid x_i), \qquad s_j \in \mathcal{S}_j \tag{4.11}$$

We see that the components of APP vectors are naturally expressed as sums of products. The sum-product algorithm is based on two fundamental principles:

1) *Past / future decomposition rule;*
2) *Sum-product update rule.*

The first principle is based on the Cartesian product decomposition in Section 4.7.1. In this case, every state S_j (i.e., the jth edge in the normal graph) is a cut set whose removal partitions the graph into two disconnected sub-graphs, which we label arbitrarily as $\mathcal{P}$ and $\mathcal{F}$. So $\mathcal{C}$ may be decomposed as a union of Cartesian product

$$\mathcal{C} = \bigcup_{s_j \in \mathcal{S}_j} \mathcal{C}_j(s_j)$$

$$\mathcal{C}_j(s_j) = X_{|\mathcal{P}}(s_j) \times X_{|\mathcal{F}}(s_j) \tag{4.12}$$

where $X_{|\mathcal{P}}(s_j)$ and $X_{|\mathcal{F}}(s_j)$ are sets of symbol values in each sub-graph that are consistent with S_j taking the value s_j.

We now apply an elementary Cartesian-product lemma.

Lemma1 (Cartesian-product distributive law): If $\mathcal{X}$ and $\mathcal{Y}$ are disjoint discrete sets and $f(x)$ and $g(y)$ are any two functions defined on $\mathcal{X}$ and $\mathcal{Y}$, then

$$\sum_{(x,y)\in\mathcal{X}\times\mathcal{Y}} f(x)g(y) = \left(\sum_{x\in\mathcal{X}} f(x)\right)\left(\sum_{y\in\mathcal{Y}} g(y)\right) \tag{4.13}$$

This lemma says that rather than computing the sum of $|\mathcal{X}||\mathcal{Y}|$ products, we can just compute a single product of independent sums over $\mathcal{X}$ and $\mathcal{Y}$. It lies at the heart of many fast algorithms.

Using (4.12) and applying the lemma in (4.13), we obtain

$$\begin{aligned} P(S_j = s_j \mid \mathbf{y}) &\propto \sum_{\mathbf{x}\in\mathcal{C}_j(s_j)} \left(\prod_{i\in\mathcal{I}_\mathcal{P}} p(y_i \mid x_i)\right)\left(\prod_{i\in\mathcal{I}_\mathcal{F}} p(y_i \mid x_i)\right) \\ &= \left(\sum_{\mathbf{x}_{|\mathcal{P}}\in X_{|\mathcal{P}}(s_j)} \prod_{i\in\mathcal{I}_\mathcal{P}} p(y_i \mid x_i)\right)\left(\sum_{\mathbf{x}_{|\mathcal{F}}\in X_{|\mathcal{F}}(s_j)} \prod_{i\in\mathcal{I}_\mathcal{F}} p(y_i \mid x_i)\right) \\ &\propto P(S_j = s_j \mid \mathbf{y}_{|\mathcal{P}}) P(S_j = s_j \mid \mathbf{y}_{|\mathcal{F}}) \end{aligned} \tag{4.14}$$

where $\mathbf{x}_{|\mathcal{P}} = \{x_i, i \in \mathcal{I}_\mathcal{P}\}$ and $\mathbf{x}_{|\mathcal{F}} = \{x_i, i \in \mathcal{I}_\mathcal{F}\}$. The sum-product algorithm therefore computes the likelihood vectors $\left\{P(S_j = s_j \mid \mathbf{y}_{|\mathcal{P}})\right\}$ and $\left\{P(S_j = s_j \mid \mathbf{y}_{|\mathcal{F}})\right\}$ separately, and then multiplies them component-wise to obtain $\left\{P(S_j = s_j \mid \mathbf{y})\right\}$. This is the past/future decomposition rule for state variables.

Likelihood values for symbol variables are computed similarly. In this case, since symbol variables have degree 1, one of the two components (i.e., subgraphs) of graph induced by a cut is just the symbol variable itself. The past/future decomposition rule thus reduces to

$$P(X_i = x_i \mid \mathbf{y}) \propto p(y_i \mid x_i)\left(\sum_{\mathbf{x}\in\mathcal{C}_i(x_i)} \prod_{i'\neq i} p(y_{i'} \mid x_{i'})\right)$$

$$\propto P(x_i \mid y_i)P(x_i \mid \mathbf{y}_{|i'\neq i}) \tag{4.15}$$

where $\mathbf{y}_{|i'\neq i} = \{y_{i'}, i' \in \mathcal{I} \setminus \{i\}\}$. In the turbo code literature, the first term $P(x_i \mid y_i)$ is called the intrinsic likelihood of x_i, and the 2nd term $P(x_i \mid \mathbf{y}_{|i'\neq i})$ is called the extrinsic likelihood of x_i. To compute likelihoods, the algorithm proceeds recursively according to the following sum-product update rule.

- The sum-product update rule is a local rule for the calculation of a likelihood vector such as $\{P(S_j = s_j \mid \mathbf{y}_{|\mathcal{P}}), s_j \in \mathcal{S}_j\}$ from likelihood vectors that are one step further upstream. The local configuration with respect to the edge corresponding to the S_j is illustrated in Fig. 4.7.4. Let $\mathcal{C}_k$ be the constraint code corresponding to past vertex. If the degree of $\mathcal{C}_k$ is δ_k, then there are δ_k-1 edges further upstream of $\mathcal{C}_k$ corresponding to further past state or symbol variables. For simplicity, suppose that these are all state variables $\{S_{j'}, j' \in \mathcal{K}_{jk}\}$, where $\mathcal{K}_{jk} \subseteq \mathcal{K}_{|\mathcal{P}}$ denotes the set of indexes of all other edges incident on the kth vertex (corresponding to a constraint code $\mathcal{C}_k$). Since the graph is cycle-free, each of these past edges has its own independent past $\mathcal{P}_{j'}$. The corresponding set $X_{|\mathcal{P}_{j'}}$ of input symbols must be disjoint, and their union must be $X_{|\mathcal{P}}$. Thus if $\mathcal{C}_k(s_j)$ is the set of codewords in the local constraint code $\mathcal{C}_k$ that are consistent with $S_j = s_j$, and $X_{|\mathcal{P}_{j'}}(s_{j'})$ is the set of $\mathbf{x}_{|\mathcal{P}_{j'}} \in X_{|\mathcal{P}_{j'}}$ that are consistent with $S_{j'} = s_{j'}$, then we have

$$X_{|\mathcal{P}}(s_j) = \bigcup_{\mathcal{C}_k(s_j)} \bigotimes_{j'\in\mathcal{K}_{jk}} X_{|\mathcal{P}_{j'}}(s_{j'}) \tag{4.16}$$

That is, for each codeword in $\mathcal{C}_k$ for which $S_j = s_j$, the set of possible pasts is the Cartesian product of possible pasts of the other state values $\{s_{j'}, j' \in \mathcal{K}_{jk}\}$, and the total set of possible pasts is the disjoint union of these Cartesian products.

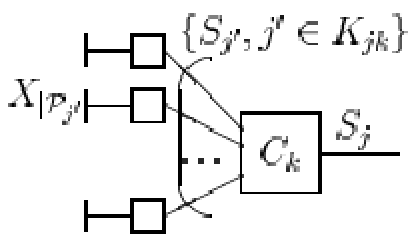


Figure 4.7.4 Local configuration for sum-product update rule

Using the Cartesian-product distributive law, it follows from (4.16) that

$$P(S_j = s_j \mid \mathbf{y}_{|\mathcal{P}}) = \sum_{\mathcal{C}_k(s_j)} \prod_{j' \in \mathcal{K}_{jk}} P(S_{j'} = s_{j'} \mid \mathbf{y}_{|\mathcal{P}_{j'}}) \tag{4.17}$$

This is the sum-product update rule. We can see that for each $s_j \in \mathcal{S}_j$, it involves a sum of $|\mathcal{C}_k|$ products of δ_k-1 terms. Its complexity is thus proportional to $|\mathcal{C}_k|$. Note that for a conventional state realization, $|\mathcal{C}_k|$ is the branch complexity of the kth trellis section. In the special case where $\mathcal{C}_k$ is a repetition code, (4.17) reduces to the product update rule:

$$P(S_j = s_j \mid \mathbf{y}_{|\mathcal{P}}) = \prod_{j' \in \mathcal{K}_{jk}} P(S_{j'} = s_{j'} \mid \mathbf{y}_{|\mathcal{P}_{j'}}) \tag{4.18}$$

In many descriptions of the sum-product algorithm for factor graphs, the product update rule is often stated as a separate rule for variable nodes.

- *Message-Passing*: For each edge in a normal graph, we wish to compute two likelihood vectors, corresponding to past and future. These two vectors can be thought of as two messages going in opposite directions. Using the sum-product update rule, each message may be computed after all upstream messages have been received at the upstream vertex. Therefore we can think of each vertex as a processor that computes an outgoing message on each edge after it has received incoming messages on all other edges.
- There are many possible schedules for this iteration. Two typical message-passing schedules are listed below:
  - Sequential schedule
  - Parallel schedule (or Flood schedule)
- The BCJR algorithm: The sum-product algorithm may be used for exact APP decoding on any trellis graph, the resulting algorithm is known as the BCJR algorithm. Fig. 4.7.5 shows the flow of messages and computations when the sum-product algorithm is applied to a trellis.

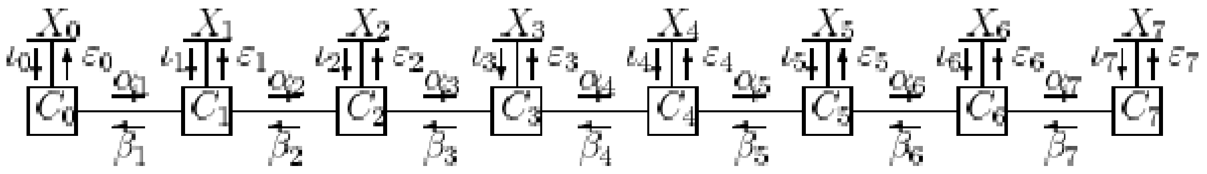

X_0 X_1 X_2 X_3 X_4 X_5 X_6 X_7
ι_0 ε_0 ι_1 ε_1 ι_2 ε_2 ι_3 ε_3 ι_4 ε_4 ι_5 ε_5 ι_6 ε_6 ι_7 ε_7
α_1 α_2 α_3 α_4 α_5 α_6 α_7
C_0 C_1 C_2 C_3 C_4 C_5 C_6 C_7
β_1 β_2 β_3 β_4 β_5 β_6 β_7

Figure 4.7.5 The sum-product algorithm on a trellis

The input messages are the intrinsic likelihood vectors $w_i = \{p(y_i \mid x_i), x_i \in \mathcal{X}_i\}$, and the output messages are the extrinsic APP vectors $\varepsilon_i = \{P(x_i \mid \mathbf{y}_{|i' \neq i}), x_i \in \mathcal{X}_i\}$. The intermediate messages are forward state APP vectors $\alpha_j = \{P(s_j \mid \mathbf{y}_{|\mathcal{P}_j}), s_j \in \mathcal{S}_j\}$ and the backward state APP vectors $\beta_j = \{P(s_j \mid \mathbf{y}_{|\mathcal{F}_j}), s_j \in \mathcal{S}_j\}$.

The algorithm proceeds independently in the forward and backward directions. In the forward direction (i.e., from left to right), the messages α_i are computed by the sum-product rule from the previous message $\alpha_{i\text{-}1}$ and the most recent input message $w_{i\text{-}1}$. In the backward direction, the messages β_j are computed by the sum-product rule from β_{j+1} and w_j. Finally, each output message w_i may be computed by the sum-product rule from the messages α_i and β_{j+1}. And the APP vector of an input symbol is given by the component-wise product of w_i and ε_i according to (4.15).

4.7.4 The Min-Sum Algorithm and ML decoding

As mentioned earlier, the standard trellis-based decoding algorithms are instances of the sum product algorithm, which works on any factor graph. In particular, when applied to a trellis, the sum-product algorithm becomes the BCJR algorithm [?] and the max-product algorithm (or the min-sum algorithm applied in the logarithmic domain) becomes a soft-output version of the Viterbi algorithm [?].

Sum-product algorithm ➔ *Perform APP decoding* $\xrightarrow{\textit{On}\text{ trellis}}$ *BCJR algorithm*

Min-Sum algorithm $\xrightarrow{\text{Perform ML decoding on a trellis}}$ *Viterbi algorithm*

Again, let $\mathcal{C}_i(x_i)$ denote the subset of codewords in which the symbol variable X_i has the value $x_i \in \mathcal{X}_i$. Then the metric $m_i(x_i)$ of x_i is defined as

$$m_i(x_i) = \max_{\mathbf{x} \in \mathcal{C}_i(x_i)} p(\mathbf{y} \mid \mathbf{x}) = \max_{\mathbf{x} \in \mathcal{C}_i(x_i)} \prod_{i' \in \mathcal{I}} p(y_{i'} \mid x_{i'}), \qquad x_i \in \mathcal{X}_i \qquad (4.19)$$

Clearly, the symbol value x_i with the maximum metric $m_i(x_i)$ will be the value of X_i in the codeword $\mathbf{x} \in \mathcal{C}$ that has the maximum global likelihood.

Similarly, if $\mathcal{C}_j(s_j)$ denote the subset of codewords that are consistent with the state variable $S_j = s_j \in \mathcal{S}_j$, then the metric $m_j(s_j)$ of s_j will be defined as

$$m_j(s_j) = \max_{\mathbf{x} \in \mathcal{C}_j(x_j)} \prod_{i \in I} p(y_i \mid x_i), \qquad s_j \in \mathcal{S}_j \qquad (4.20)$$

We can see that these metrics could be computed by a version of the sum-product algorithm in which "sum" is replaced by "max" everywhere, resulting in the so-called "max-product algorithm".

In particular, for non-negative real-valued quantities, we have

- The distributive law: $a(\max(b, c)) = \max(ab, ac)$
- The Cartesian-product distributive law:

$$\max_{(x,y)\in\mathcal{X}\times\mathcal{Y}} f(x)g(y) = \left(\max_{x\in\mathcal{X}} f(x)\right)\left(\max_{y\in\mathcal{Y}} g(y)\right) \tag{4.21}$$

From (4.12), we now obtain the past/future decomposition rule

$$m_j(s_j) = \left(\max_{\mathbf{x}_{|\mathcal{P}}\in X_{|\mathcal{P}}(s_j)} \prod_{i\in\mathcal{I}_\mathcal{P}} p(y_i \mid x_i)\right)\left(\max_{\mathbf{x}_{|\mathcal{F}}\in X_{|\mathcal{F}}(s_j)} \prod_{i\in\mathcal{I}_\mathcal{F}} p(y_i \mid x_i)\right)$$

$$= m_j(s_j \mid \mathbf{y}_{|\mathcal{P}})m_j(s_j \mid \mathbf{y}_{|\mathcal{F}}) \tag{4.22}$$

where $\mathbf{y}_{|\mathcal{P}}$ and $\mathbf{y}_{|\mathcal{F}}$ are observations on the past symbols $\mathbf{x}_{|\mathcal{P}} = \{x_i, i\in\mathcal{I}_\mathcal{P}\}$ and future symbols $\mathbf{x}_{|\mathcal{F}} = \{x_i, i\in\mathcal{I}_\mathcal{F}\}$, repectively.

Similarly, we obtain the max-product update rule

$$m_j(s_j \mid \mathbf{y}_{|\mathcal{P}}) = \max_{\mathcal{C}_k(s_j)} \prod_{j'\in K_{jk}} m_{j'}(s_{j'} \mid \mathbf{y}_{|\mathcal{P}_{j'}}) \tag{4.23}$$

where the notation is as in the sum-product update rule (4.17)

In practice, the algorithm is often carried out in the negative log-likelihood domain. Thus, the “product” operation becomes a “sum” and the “max” operation becomes a “min” operation, yielding the *min-sum algorithm*. On a trellis, the result is a *bidirectional Viterbi algorithm* (VA). The forward part of any of these algorithms is equivalent to the VA. The update rule (4.23) becomes add-compare-select operation.

4.7.5 The Sum-Product Algorithm on Graphs with Cycles

On a graph with cycles, there are several basic approaches to decoding.

- Cluster the graph enough to eliminate the cycles, and then apply the sum-product algorithm. The decoding will be exact. However, the complexity advantage of a realization on a graph with cycles will be lost.
- The 2nd approach is simply to apply the sum-product algorithm to the graph with cycles. Because the sum-product update rule is local, it may be used in an iterative decoding algorithm. One must then specify an initialization rule, a schedule and a stopping criterion. *The decoding performance is in general suboptomal*. There is no guarantee that the sum-product algorithm will converge. Even if the algorithm converges, there is now no guarantee that it will converge to the correct likelihoods. However, such approximate iterative sum-product algorithms often work very well.

Iterative algorithm:. First, all edges are initialized with a neutral message, i.e., a factor $\mu(\cdot)$ = 1. All messages are then repeatedly updated, according to some schedule. The computation stops when a given number of iterations is achieved or when some other stopping condition is satisfied (e.g., when a valid codeword was found).

Appendix A

BCJR Trellis of a Linear Block Code

Consider a binary (n, k, d) linear block code $\mathcal{C}$ with parity-check matrix $\mathbf{H}=[\mathbf{h}_0 \quad \mathbf{h}_1 \quad \cdots \quad \mathbf{h}_{n-1}]$. Let $\mathbf{x}=(x_0,x_2,...,x_{n-1})\in\mathcal{C}$ be an arbitrary codeword, and let r=n-k. We use the column vectors $\mathbf{s}_i$, $0 \le i \le n$-1, to denote the state variables of the trellis corresponding to $\mathcal{C}$. For BCJR trellis, $\mathbf{s}_i \in \mathbb{F}_2^r$ is defined by

$$\mathbf{s}_0 = \mathbf{0},$$

$$\mathbf{s}_{i+1}=\mathbf{s}_i + x_i\mathbf{h}_i = \sum_{t=0}^{i} x_t\mathbf{h}_t\ , \qquad 0\le i < n\text{-}1 \tag{A1}$$

Clearly, $\mathbf{s}_n$ is the syndrome of $\mathbf{x}$ and $\mathbf{s}_n = \mathbf{0}$ for all codewords. The next state $\mathbf{s}_{i+1}$ is determined by the current state $\mathbf{s}_i$ and the current input x_i. The tripe ($\mathbf{s}_i$, $\mathbf{s}_{i+1}$, x_i) specifies the trellis branches in the trellis diagram. Thus a trellis can be constructed using (A1). The resulting trellis structure is irregular compared to the trellis of a convolutional code. The maximum number of states in the BCJR trellis is $\min\{2^k, 2^{n-k}\}$. The BCJR trellis has the property that it has the smallest number of states. A trellis with this property is called a *minimal trellis*.

As an example, the BCJR trellis for the (7, 4) cyclic Hamming code with

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

is shown in Fig. A1, where the trellis state $\mathbf{s}_i=(s_0,s_1,s_2)^T$ is labeled with the corresponding m-tuples over GF(2).
The trellis has maximum of 8 states at time i = 3 and i = 4.

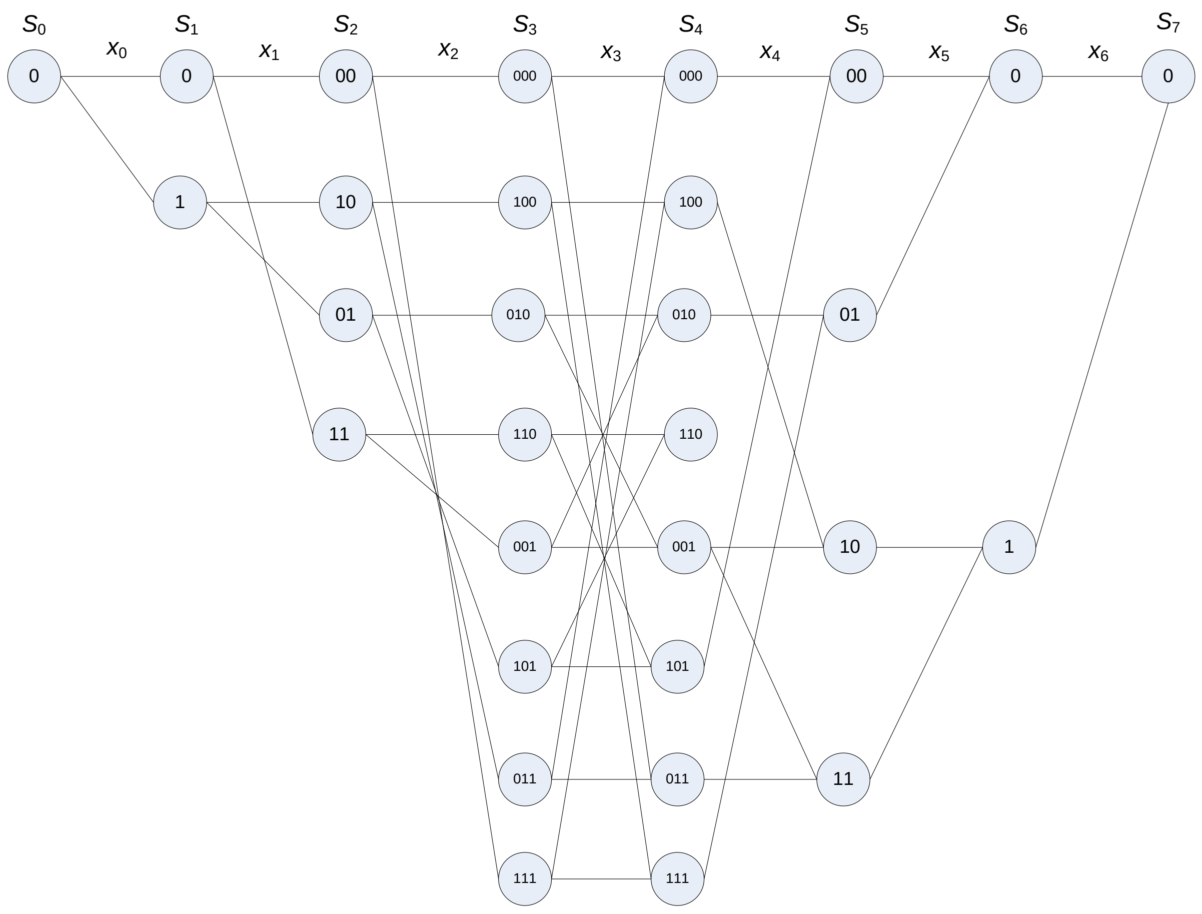


Figure A1. A trellis diagram for the (7, 4) Hamming code.

The trellis diagram represents the 16 codewords of the Hamming code as the set of labeled paths obtained starting from the leftmost vertex and proceeding rightwards in the graph. The *i*th trellis section constrains the possible combinations of $(\mathbf{s}_{i-1}, x_{i-1}, \mathbf{s}_i)$; in fact, in this linear trellis example, these triples always form a linear code. For example, the 3rd section forms the local code that consists of the eight binary linear combinations of (00, 0, 000), (00, 1, 111), (10, 0, 100), (10, 1, 011), and etc. This code may be regarded as a binary (6, 3) code, and is labeled as such in Fig. A3.

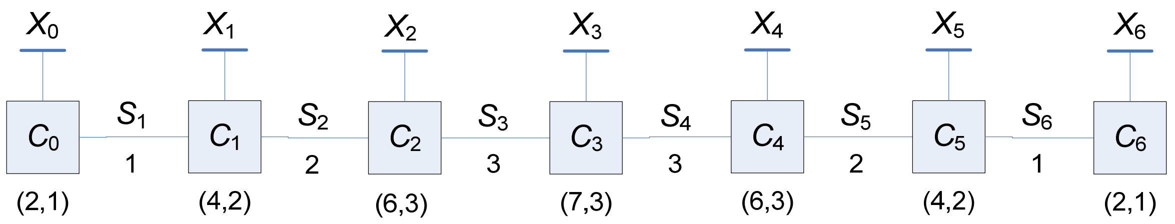


Figure A3. The corresponding factor graph of figure A1

Fig. A4 is the trellis of the (5,3) block code defined by $\mathbf{H} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \end{bmatrix}$.

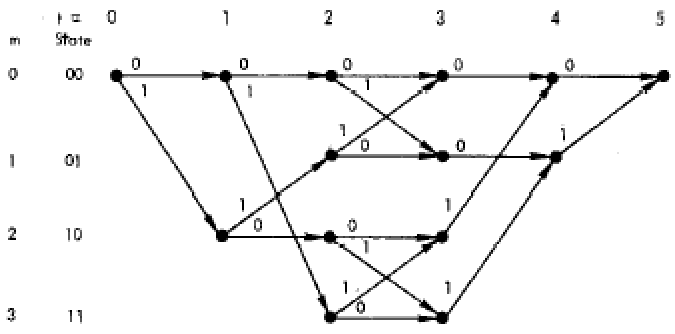


Figure A4

It is interesting to note that with a syndrome trellis there is no need to label the branches with the coded bits. A transition between two states with the same label (i.e., a horizontal branch) corresponds to coded bit 0, as seen from equation (A1)*: If the coded bit is 0, then the sum does not change*.

For some classes of codes, such as extended BCH codes and Reed-Muller codes, the trellis can be divided into sections. This results in a more regular and symmetric trellis structure with many parallel subtrellises, which may be used to build very high-speed Viterbi decoders for block codes.

Appendix B

Application Issues of Forney-Style Factor Graphs

Factor graphs originate in coding theory, but they are applicable to many other areas. For example, a large number of practical algorithms for a wide variety of detection and estimation problems in signal processing can be derived as summary propagation algorithms. The algorithms derived in this way often include the best previously known algorithms as special cases or as obvious approximations. We now discuss the FFG in a more general sense, and describe the FFG models for several applications. Much of the material in this section is taken from [4].

Application Examples:

- A main application of factor graphs is probabilistic models. (In this case, the sample space can usually be identified with the configuration space $\mathcal{X}$.) For example, let X, Y, and Z be random variables that form a Markov chain. Then their joint probability density (or their joint probability mass function) $p_{XYZ}(x,y,z)$ can written as

$$p_{XYZ}(x,y,z) = p_X(x)\, p_{Y|X}(y \mid x)\, p_{Z|Y}(z \mid y)$$

This factorization is expressed by the FFG of Fig. 4.A5.

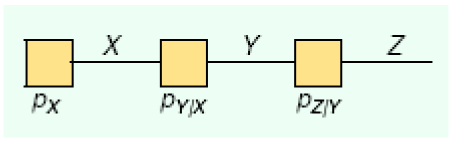


Figure 4.A5 An FFG of a Markov chain

- A deterministic block diagram may also be viewed as a factor graph. Consider, for example, the block diagram of Fig. 4.A6, which expresses the two equations

$$X = g(U, W) \tag{A2}$$

$$Z = h(X, Y) \tag{A3}$$

In the factor graph interpretation, the function block *X = g(U, W)* in the block diagram is interpreted as representing the factor $\delta\left(x - g(U,W)\right)$, where $\delta(.)$ is the Kronecker delta function if *X* is a discrete variable or the Dirac delta if *X* is a continuous variable. Considered as a factor graph, Fig. 4.A6 thus expresses the factorization

$$f(u,w,x,y,z) = \delta(x - g(u,w)) \cdot \delta(z - h(x,y))$$

Note that this function is nonzero (i.e., the configuration is valid) if and only if the configuration is consistent with both (A2) and (A3).

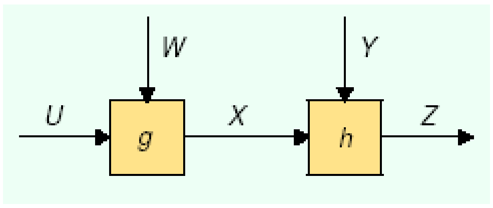


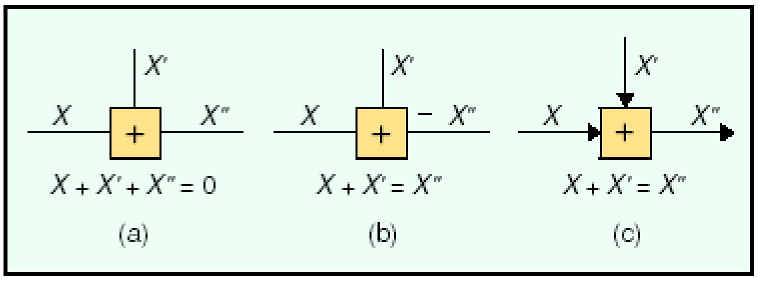
Figure 4.A6 A block diagram

Figure 4.A6 (b) The corresponding FFG

- Besides the symbol representing equality constraint, other special symbols are also used for frequently occurring local functions. For example, we will use the zero-sum constraint node shown in Fig. 4.A7(a), which represents the local function

$$f_+(x, x', x'') = \delta(x + x' + x'')$$

Clearly, *X* +*X'* +*X''* = 0 holds for every valid configuration. Both the nodes in Figs. 4.A7 (b) and (c) represent the constraint *X* +*X'* =*X''*, or, equivalently, the factor $\delta(x + x' - x'')$. Both the equality constraint and the zero-sum constraint can obviously be extended to more than three variables.



▲ 5. Zero-sum constraint node.

Fig. 4.A7

■ FFG for Channel Models

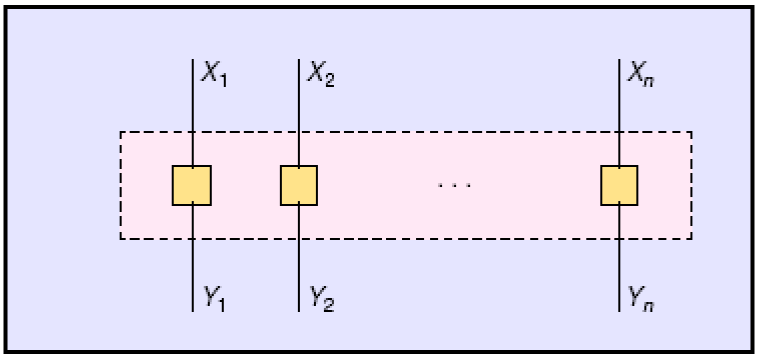
A channel model is a family $p(y|x)$ of probability distributions over a block $\mathbf{y}=(y_1,y_2,\ldots,y_N)$ of channel output symbols given any block $\mathbf{x}=(x_1,x_2,\ldots,x_N)$ of channel input symbols. Two examples of channel models are shown in Figs. 4.A8 and A9. Fig. 4.A8 shows a memoryless channel with

$$p(\mathbf{y}\,|\,\mathbf{x})=\prod_{n=1}^{N}p(y_n\,|\,x_n)$$

Fig. 4.A9 shows a state-space representation with internal states $S_0, S_1, \ldots, S_n$:

$$p(\mathbf{y},\mathbf{s}\,|\,\mathbf{x})=p(s_0)\prod_{n=1}^{N}p(y_n,s_n\,|\,x_n,s_{n-1})$$

Such a state space representation might be, e.g., a finite-state trellis as in Fig. 4.6.6., or a linear model.



▲ 14. Memoryless channel.

Figure 4.A8

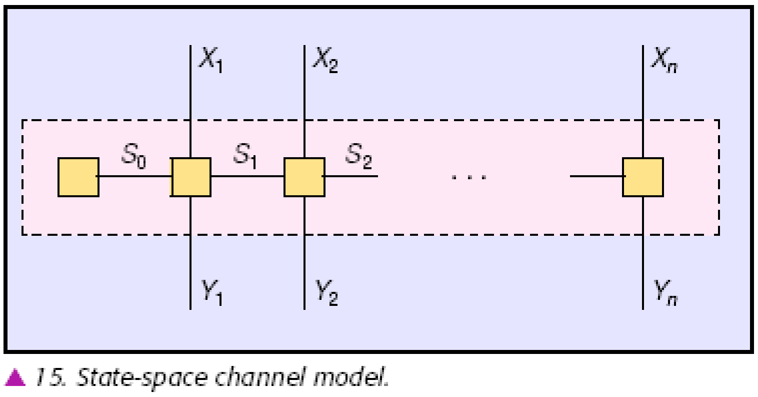


Figure 4.A9

■ **FFG for Signal mapper**

Consider the mapper shown in Fig. 4.A10, where two binary symbols, X_A and X_B, are mapped to a 4-PAM symbol Z. Let $f : \mathbb{F}_2 \times \mathbb{F}_2 \to \{-3, -1, +1, +3\}$ be this mapping and assume that x_A is mapped to the more significant bit of z. In an FFG, the mapper becomes a factor node with local function

$$I_f(x_A, x_B, z) \equiv \begin{cases} 1, & \text{if } f(x_A, x_B) = z \\ 0, & \text{otherwise} \end{cases}$$

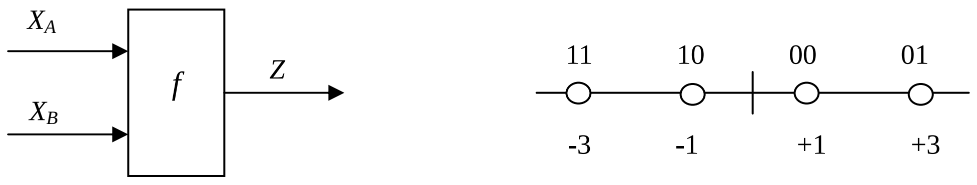


Figure 4.A10 Bits-to-symbol mapper

The computation of all messages in and out of the node (cf. Fig. 4.A11) is immediate from the sum-product rule. For example, we have

$$\mu_{\text{in}X_A}(x_A) = \sum_{x_B, z} I_f(x_A, x_B, z)\mu_{\text{out}X_B}(x_B)\mu_{\text{in}Z}(z)$$

which expands to

$$\mu_{\text{in}X_A}(0) = \mu_{\text{out}X_B}(1) \cdot \mu_{\text{in}Z}(+3) + \mu_{\text{out}X_B}(0) \cdot \mu_{\text{in}Z}(+1)$$

$$\mu_{\text{in}X_A}(1) = \mu_{\text{out}X_B}(0) \cdot \mu_{\text{in}Z}(-1) + \mu_{\text{out}X_B}(1) \cdot \mu_{\text{in}Z}(-3)$$

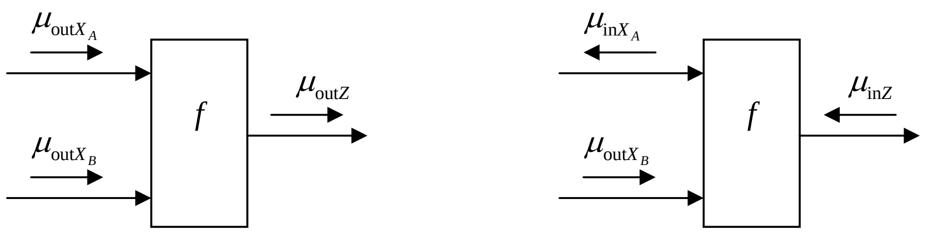


Figure 4.A11 Messages through the mapper

References

[1] G. D. Forney, Jr., "Codes on graphs: Normal realizations," *IEEE Trans. Inform. Theory,* vol.47, no.2, pp.520-548, Feb. 2001.
[2] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Trans. Inform. Theory*, vol.47, no.2, pp.498-519, Feb. 2001.
[3] S. M. Aji and R. J. McEliece, "The generalized distributive law," *IEEE Trans. Inform. Theory*, vol.46, pp.325-343, Mar. 2000.
[4] H.-A. Loeliger, "An introduction to factor graphs," *IEEE Signal Processing Mag*. pp.28-41, Jan. 2004.
[5] G. D. Forney, *Principles of Digital Communication - Part II*, Lecture notes. MIT, 2005.