Chapter 12

The sum-product algorithm

The sum-product algorithm is the basic "decoding" algorithm for codes on graphs. For finite cycle-free graphs, it is finite and exact. However, because all its operations are local, it may also be applied to graphs with cycles; then it becomes iterative and approximate, but in coding applications it often works very well. It has become the standard decoding algorithm for capacity-approaching codes (*e.g.,* turbo codes, LDPC codes).

There are many variants and applications of the sum-product algorithm. The most straightforward application is to *a posteriori* probability (APP) decoding. When applied to a trellis, it becomes the celebrated BCJR decoding algorithm. In the field of statistical inference, it becomes the even more widely known "belief propagation" (BP) algorithm. For Gaussian state-space models, it becomes the Kalman smoother.

There is also a "min-sum" or maximum-likelihood sequence detection (MLSD) version of the sum-product algorithm. When applied to a trellis, the min-sum algorithm gives the same result as the Viterbi algorithm.

12.1 The sum-product algorithm on cycle-free graphs

We will develop the sum-product algorithm as an APP decoding algorithm for a code $\mathcal{C}$ that has a cycle-free normal graph realization. We then discuss generalizations.

The code $\mathcal{C}$ is therefore described by a realization involving a certain set of symbol variables $\{Y_i, i \in \mathcal{I}\}$ represented by half-edges (dongles), a certain set of state variables $\{\Sigma_j, j \in \mathcal{J}\}$ represented by edges, and a certain set of constraint codes $\{\mathcal{C}_k, k \in \mathcal{K}\}$ of arbitrary degree, such that the graph of the realization is cycle-free; *i.e.,* every edge (and obviously every half-edge) is by itself a cut set.

APP decoding is defined in general as follows. We assume that a set of independent observations are made on all symbol variables $\{Y_i, i \in I\}$, resulting in a set of observations $\mathbf{r} = \{r_i, i \in \mathcal{I}\}$ and likelihood vectors $\{\{p(r_i \mid y_i), y_i \in \mathcal{Y}_i\}, i \in \mathcal{I}\}$, where $\mathcal{Y}_i$ is the alphabet of Y_i. The likelihood of a codeword $\mathbf{y} = \{y_i, i \in \mathcal{I}\} \in \mathcal{C}$ is then defined as the componentwise product $p(\mathbf{r} \mid \mathbf{y}) = \prod_{i \in \mathcal{I}} p(r_i \mid y_i)$.

Assuming equiprobable codewords, the *a posteriori* probabilities $\{p(\mathbf{y} \mid \mathbf{r}), \mathbf{y} \in \mathcal{C}\}$ (APPs) are proportional to the likelihoods $\{p(\mathbf{r} \mid \mathbf{y}), \mathbf{y} \in \mathcal{C}\}$, since by Bayes' law,

$$p(\mathbf{y} \mid \mathbf{r}) = \frac{p(\mathbf{r} \mid \mathbf{y})p(\mathbf{y})}{p(\mathbf{r})} \propto p(\mathbf{r} \mid \mathbf{y}), \quad \mathbf{y} \in \mathcal{C}.$$

Let $\mathcal{C}_i(y_i)$ denote the subset of codewords in which the symbol variable Y_i has the value $y_i \in \mathcal{Y}_i$. Then, up to a scale factor, the symbol APP vector $\{p(Y_i = y_i \mid \mathbf{r}), y_i \in \mathcal{Y}_i\}$ is given by

$$p(Y_i = y_i \mid \mathbf{r}) = \sum_{\mathbf{y} \in \mathcal{C}_i(y_i)} p(\mathbf{y} \mid \mathbf{r}) \propto \sum_{\mathbf{y} \in \mathcal{C}_i(y_i)} p(\mathbf{r} \mid \mathbf{y}) = \sum_{\mathbf{y} \in \mathcal{C}_i(y_i)} \prod_{i' \in \mathcal{I}} p(r_{i'} \mid y_{i'}), \quad y_i \in \mathcal{Y}_i. \tag{12.1}$$

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

$$p(\Sigma_j = s_j \mid \mathbf{r}) \quad \propto \sum_{\mathbf{y} \in \mathcal{C}_j(s_j)} \prod_{i \in \mathcal{I}} p(r_i \mid y_i), \quad s_j \in \mathcal{S}_j. \tag{12.2}$$

We see that the components of APP vectors are naturally expressed as sums of products. The sum-product algorithm aims to compute these APP vectors for every state and symbol variable.

12.1.1 Past-future decomposition rule

The sum-product algorithm is based on two fundamental principles, which we shall call here the *past/future decomposition rule* and the *sum-product update rule*. Both of these rules are based on set-theoretic decompositions that are derived from the code graph.

The past/future decomposition rule is based on the Cartesian-product decomposition of the cut-set bound (Chapter 11). In this case every edge Σ_j is a cut set, so the subset of codewords that are consistent with the state variable Σ_j having the value s_j is the Cartesian product

$$\mathcal{C}_j(s_j) = Y_{|\mathcal{P}}(s_j) \times Y_{|\mathcal{F}}(s_j), \tag{12.3}$$

where $\mathcal{P}$ and $\mathcal{F}$ denote the two components of the disconnected graph which results from deleting the edge representing Σ_j, and $Y_{|\mathcal{P}}(s_j)$ and $Y_{|\mathcal{F}}(s_j)$ are the sets of symbol values in each component that are consistent with Σ_j taking the value s_j.

We now apply an elementary Cartesian-product lemma:

Lemma 12.1 (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*

$$\sum_{(x,y) \in \mathcal{X} \times \mathcal{Y}} f(x)g(y) = \left(\sum_{x \in \mathcal{X}} f(x) \right) \left(\sum_{y \in \mathcal{Y}} g(y) \right). \tag{12.4}$$

This lemma may be proved simply by writing the terms on the right in a rectangular array and then identifying them with the terms on the left. It 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 $\mathcal{Y}$. This simple lemma lies at the heart of many "fast" algorithms.

Using (12.3) and applying this lemma in (12.2), we obtain the past/future decomposition rule

$$\begin{aligned} p(\Sigma_j = s_j \mid \mathbf{r}) &\propto \left(\sum_{\mathbf{y}_{|\mathcal{P}} \in Y_{|\mathcal{P}}(s_j)} \prod_{i \in \mathcal{I}_\mathcal{P}} p(r_i \mid y_i) \right) \left(\sum_{\mathbf{y}_{|\mathcal{F}} \in Y_{|\mathcal{F}}(s_j)} \prod_{i \in \mathcal{I}_\mathcal{F}} p(r_i \mid y_i) \right) \\ &\propto p(\Sigma_j = s_j \mid \mathbf{r}_{|\mathcal{P}}) p(\Sigma_j = s_j \mid \mathbf{r}_{|\mathcal{F}}), \end{aligned} \tag{12.5}$$

in which the two terms $p(\Sigma_j = s_j \mid \mathbf{r}_{|\mathcal{P}})$ and $p(\Sigma_j = s_j \mid \mathbf{r}_{|\mathcal{F}})$ depend only on the likelihoods of the past symbols $\mathbf{y}_{|\mathcal{P}} = \{Y_i, i \in \mathcal{I}_\mathcal{P}\}$ and future symbols $\mathbf{y}_{|\mathcal{F}} = \{Y_i, i \in \mathcal{I}_\mathcal{F}\}$, respectively.

The sum-product algorithm therefore computes the APP vectors $\{p(\Sigma_j = s_j \mid \mathbf{r}_{|\mathcal{P}})\}$ and $\{p(\Sigma_j = s_j \mid \mathbf{r}_{|\mathcal{F}})\}$ separately, and multiplies them componentwise to obtain $\{p(\Sigma_j = s_j \mid \mathbf{r})\}$. This is the past/future decomposition rule for state variables.

APP vectors for symbol variables are computed similarly. In this case, since symbol variables have degree 1, one of the two components of the graph induced by a cut is just the symbol variable itself, while the other component is the rest of the graph. The past/future decomposition rule thus reduces to the following simple factorization of (12.1):

$$p(Y_i = y_i \mid \mathbf{r}) \propto p(r_i \mid y_i) \left(\sum_{\mathbf{y} \in \mathcal{C}_i(y_i)} \prod_{i' \neq i} p(r_{i'} \mid y_{i'}) \right) \propto p(y_i \mid r_i) p(y_i \mid \mathbf{r}_{|i' \neq i}). \tag{12.6}$$

In the turbo code literature, the first term, $p(y_i \mid r_i)$, is called the *intrinsic* information, while the second term, $p(y_i \mid \mathbf{r}_{|i' \neq i})$, is called the *extrinsic* information.

12.1.2 Sum-product update rule

The second fundamental principle of the sum-product algorithm is the sum-product update rule. This is a local rule for the calculation of an APP vector, *e.g.*, $\{p(\Sigma_j = s_j \mid \mathbf{r}_{|\mathcal{P}}), s_j \in \mathcal{S}_j\}$, from APP vectors that lie one step further upstream.

The local configuration with respect to the edge corresponding to the state variable Σ_j is illustrated in Figure 1. The edge must be incident on a unique past vertex corresponding to a constraint code $\mathcal{C}_k$. If the degree of $\mathcal{C}_k$ is δ_k, then there are $\delta_k - 1$ edges further upstream of $\mathcal{C}_k$, corresponding to further past state or symbol variables. For simplicity, we suppose that these are all state variables $\{\Sigma_{j'}, j' \in \mathcal{K}_{jk}\}$, where we denote their index set by $\mathcal{K}_{jk} \subseteq \mathcal{K}_{|\mathcal{P}}$.

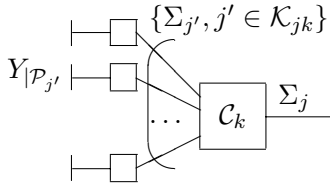

Figure 1. Local configuration for sum-product update rule.

Since the graph is cycle-free, each of these past edges has its own independent past $\mathcal{P}_{j'}$. The corresponding sets $Y_{|\mathcal{P}_{j'}}$ of input symbols must be disjoint, and their union must be $Y_{|\mathcal{P}}$. Thus if $\mathcal{C}_k(s_j)$ is the set of codewords in the local constraint code $\mathcal{C}_k$ that are consistent with $\Sigma_j = s_j$, and $Y_{|\mathcal{P}_{j'}}(s_{j'})$ is the set of $\mathbf{y}_{|\mathcal{P}_{j'}} \in Y_{|\mathcal{P}_{j'}}$ that are consistent with $\Sigma_{j'} = s_{j'}$, then we have

$$Y_{|\mathcal{P}}(s_j) = \bigoplus_{\mathcal{C}_k(s_j)} \bigotimes_{j' \in \mathcal{K}_{jk}} Y_{|\mathcal{P}_{j'}}(s_{j'}), \tag{12.7}$$

where the plus sign indicates a disjoint union, and the product sign indicates a Cartesian product. In other words, for each codeword in $\mathcal{C}_k$ for which $\Sigma_j = s_j$, 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.

Now, again using the Cartesian-product distributive law, it follows from (12.7) that

$$p(\Sigma_j = s_j \mid \mathbf{r}_{|\mathcal{P}}) = \sum_{\mathcal{C}_k(s_j)} \prod_{j' \in \mathcal{K}_{jk}} p(\Sigma_{j'} = s_{j'} \mid \mathbf{r}_{|\mathcal{P}_{j'}}). \tag{12.8}$$

Thus if we know all the upstream APP vectors $\{p(\Sigma_{j'} = s_{j'} \mid \mathbf{r}_{|\mathcal{P}_{j'}}), s_{j'} \in \mathcal{S}_{j'}\}$, then we can compute the APP vector $\{p(\Sigma_j = s_j \mid \mathbf{r}_{|\mathcal{P}}), s_j \in \mathcal{S}_j\}$.

Equation (12.8) 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 $\delta_k - 1$ terms. Its complexity is thus proportional to the size $|\mathcal{C}_k|$ of the constraint code $\mathcal{C}_k$. In a trellis, this is what we call the branch complexity.

In the special case where $\mathcal{C}_k$ is a repetition code, there is only one codeword corresponding to each $s_j \in \mathcal{S}_j$, so (12.8) becomes simply the following *product update rule*:

$$p(\Sigma_j = s_j \mid \mathbf{r}_{|\mathcal{P}}) = \prod_{j' \in \mathcal{K}_{jk}} p(\Sigma_{j'} = s_j \mid \mathbf{r}_{|\mathcal{P}_{j'}}); \tag{12.9}$$

i.e., the components of the upstream APP vectors are simply multiplied componentwise. When the sum-product algorithm is described for Tanner graphs, the product update rule is often stated as a separate rule for variable nodes, because variable nodes in Tanner graphs correspond to repetition codes in normal graphs.

Note that for a repetition code of degree 2, the product update rule of (12.9) simply becomes a pass-through of the APP vector; no computation is required. This seems like a good reason to suppress state nodes of degree 2, as we do in normal graphs.

12.1.3 The sum-product algorithm

Now we describe the complete sum-product algorithm for a finite cycle-free normal graph, using the past/future decomposition rule (12.5) and the sum-product update rule (12.8).

Because the graph is cycle-free, it is a tree. Symbol variables have degree 1 and correspond to leaves of the tree. State variables have degree 2 and correspond to branches.

For each edge, we wish to compute two APP 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 (see Figure 1). 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.

Because each edge is the root of a finite past tree and a finite future tree, there is a maximum number d of edges to get from any given edge to the furthest leaf node in either direction, which is called its *depth* d. If a message has depth d, then the depth of any upstream message can be no greater than $d - 1$. All symbol half-edges have depth $d = 0$, and all state edges have depth $d \geq 1$. The *diameter* $d_{\max}$ of the tree is the maximum depth of any message.

Initially, incoming messages (intrinsic information) are available at all leaves of the tree. All depth-1 messages can then be computed from these depth-0 messages; all depth-2 messages can then be computed from depth-1 and depth-0 messages; etc. In a synchronous (clocked) system, all messages can therefore be computed in $d_{\max}$ clock cycles.

Finally, given the two messages on each edge in both directions, all *a posteriori* probabilities (APPs) can be computed using the past/future decomposition rule (12.5).

In summary, given a finite cycle-free normal graph of diameter $d_{\max}$ and intrinsic information for each symbol variable, the sum-product algorithm computes the APPs of all symbol and state variables in $d_{\max}$ clock cycles. One message (APP vector) of size $|\mathcal{S}_j|$ is computed for each state variable Σ_j in each direction. The computational complexity at a vertex corresponding to a constraint code $\mathcal{C}_k$ is of the order of $|\mathcal{C}_k|$. (More precisely, the number of pairwise multiplications required is $\delta_k(\delta_k - 2)|\mathcal{C}_k|$.)

The sum-product algorithm does not actually require a clock. In an asynchronous implementation, each vertex processor can continuously generate outgoing messages on all incident edges, using whatever incoming messages are available. Eventually all messages must be correct. An analog asynchronous implementation can be extremely fast.

We see that there is a clean separation of functions when the sum-product algorithm is implemented on a normal graph. All computations take place at vertices, and the computational complexity at a vertex is proportional to the vertex (constraint code) complexity. The function of ordinary edges (state variables) is purely message-passing (communications), and the communications complexity (bandwidth) is proportional to the edge complexity (state space size). The function of half-edges (symbol variables) is purely input/output; the inputs are the intrinsic APPs, and the ultimate outputs are the extrinsic APP vectors, which combine with the inputs to form the symbol APPs. In integrated-circuit terminology, the constraint codes, state variables and symbol variables correspond to logic, interconnect, and I/O, respectively.

12.2 The BCJR algorithm

The chain graph of a trellis (state-space) representation is the archetype of a cycle-free graph. The sum-product algorithm therefore may be used for exact APP decoding on any trellis (state-space) graph. In coding, the resulting algorithm is known as the Bahl-Cocke-Jelinek-Raviv (BCJR) algorithm. (In statistical inference, it is known as the forward-backward algorithm. If all probability distributions are Gaussian, then it becomes the Kalman smoother.)

Figure 2 shows the flow of messages and computations when the sum-product algorithm is applied to a trellis.

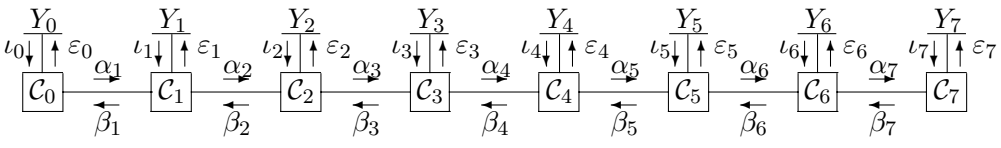

Figure 2. Flow of messages and computations in the sum-product algorithm on a trellis.

The input messages are the intrinsic APP vectors $\iota_i = \{p(r_i \mid y_i), y_i \in \mathcal{Y}_i\}$, derived from the observations r_i; the output messages are the extrinsic APP vectors $\varepsilon_i = \{p(y_i \mid \mathbf{r}_{|i' \neq i}), y_i \in \mathcal{Y}_i\}$. The intermediate messages are the forward state APP vectors $\alpha_j = p(s_j \mid \mathbf{r}_{|\mathcal{P}_j}), s_j \in \mathcal{S}_j\}$ and the backward state APP vectors $\beta_j = p(s_j \mid \mathbf{r}_{|\mathcal{F}_j}), s_j \in \mathcal{S}_j\}$, where $\mathbf{r}_{|\mathcal{P}_j}$ and $\mathbf{r}_{|\mathcal{F}_j}$ denote the observations before and after s_j, respectively.

The algorithm proceeds independently in the forward and backward directions. In the forward direction, the messages α_j are computed from left to right; α_j may be computed by the sum-product rule from the previous message α_{j-1} and the most recent input message ι_{j-1}. In the backward direction, the messages β_j are computed from right to left; β_j may be computed by the sum-product rule from β_{j+1} and ι_j.

Finally, each output message ε_i may be computed by the sum-product update rule from the messages α_i and β_{i+1}, giving the extrinsic information for each symbol. To find the APP vector of an input symbol Y_i, the intrinsic and extrinsic messages ι_i and ε_i are multiplied componentwise, according to the past/future decomposition rule. (In turbo decoding, the desired output is actually the extrinsic likelihood vector, not the APP vector.) Similarly, to find the APP vector of a state variable Σ_j, the forward and backward messages α_j and β_j are multiplied componentwise.

Exercise 1. Consider the two-state trellis diagram for the binary $(7, 6, 2)$ SPC code shown in Figure 3 of Chapter 10. Suppose that a codeword is chosen equiprobably at random, that the transmitter maps $\{0, 1\}$ to $\{\pm 1\}$ as usual, that the resulting real numbers are sent through a discrete-time AWGN channel with noise variance $\sigma^2 = 1$ per symbol, and that the received sequence is $\mathbf{r} = (0.4, -1.0, -0.1, 0.6, 0.7, -0.5, 0.2)$. Use the sum-product algorithm to determine the APP that each input bit Y_i is a 0 or a 1.

12.3 The min-sum algorithm and ML decoding

We now show that with minor modifications the sum-product algorithm may be used to perform a variant of maximum-likelihood (ML) sequence decoding rather than APP decoding. On a trellis, the resulting "min-sum" algorithm becomes a variant of the Viterbi algorithm.

With the same notation as in the previous section, the min-sum algorithm is defined as follows. Again, let $\mathcal{C}_i(y_i)$ denote the subset of codewords in which the symbol variable Y_i has the value $y_i \in \mathcal{Y}_i$. Then the *metric* $m_i(y_i)$ of y_i is defined as the maximum likelihood of any codeword $\mathbf{y} \in \mathcal{C}_i(Y_i)$; *i.e.,*

$$m_i(y_i) = \max_{\mathbf{y} \in \mathcal{C}_i(Y_i)} p(\mathbf{r} \mid \mathbf{y}) = \max_{\mathbf{y} \in \mathcal{C}_i(Y_i)} \prod_{i' \in I} p(r_{i'} \mid y_{i'}), \quad y_i \in \mathcal{Y}_i. \tag{12.10}$$

It is clear that the symbol value y_i with the maximum metric $m_i(y_i)$ will be the value of y_i in the codeword $\mathbf{y} \in \mathcal{C}$ that has the maximum global likelihood.

Similarly, if $\mathcal{C}_j(s_j)$ denotes the subset of codewords that are consistent with the state variable Σ_j having the value s_j in the state alphabet $\mathcal{S}_j$, then the metric $m_j(s_j)$ of s_j will be defined as the maximum likelihood of any codeword $\mathbf{y} \in \mathcal{C}_j(s_j)$:

$$m_j(s_j) \quad = \quad \max_{\mathbf{y} \in \mathcal{C}_j(s_j)} \prod_{i \in I} p(r_i \mid y_i), \quad s_j \in \mathcal{S}_j. \tag{12.11}$$

We recognize that (12.10) and (12.11) are almost identical to (12.1) and (12.2), with the exception that the sum operator is replaced by a max operator. This suggests that these metrics could be computed by a version of the sum-product algorithm in which "sum" is replaced by "max" everywhere, giving what is called the "max-product algorithm."

In fact this works. The reason is that the operators "max" and "product" operate on probability vectors defined on sets according to the same rules as "sum" and "product." In particular, assuming that all quantities are non-negative, we have

(a) the distributive law: $a \max\{b, c\} = \max\{ab, ac\}$;

(b) the Cartesian-product distributive law:

$$\max_{(x,y)\in\mathcal{X}\times\mathcal{Y}} f(x)g(y) = \left(\max_{x\in\mathcal{X}} f(x)\right) \left(\max_{y\in\mathcal{Y}} g(y)\right). \tag{12.12}$$

Consequently, the derivation of the previous section goes through with just this one change. From (12.3), we now obtain the past/future decomposition rule

$$m_j(s_j) = \left(\max_{\mathbf{y}_{|\mathcal{P}}\in Y_{|\mathcal{P}}(s_j)} \prod_{i\in\mathcal{I}_\mathcal{P}} p(r_i \mid y_i)\right) \left(\max_{\mathbf{y}_{|\mathcal{F}}\in Y_{|\mathcal{F}}(s_j)} \prod_{i\in\mathcal{I}_\mathcal{F}} p(r_i \mid y_i)\right) = m_j(s_j \mid \mathbf{r}_{|\mathcal{P}}) m_j(s_j \mid \mathbf{r}_{|\mathcal{F}}),$$

in which the partial metrics $m_j(s_j \mid \mathbf{r}_{|\mathcal{P}})$ and $m_j(s_j \mid \mathbf{r}_{|\mathcal{F}})$ are the maximum likelihoods over the past symbols $\mathbf{y}_{|\mathcal{P}} = \{y_i, i \in \mathcal{I}_\mathcal{P}\}$ and future symbols $\mathbf{y}_{|\mathcal{F}} = \{y_i, i \in \mathcal{I}_\mathcal{F}\}$, respectively. Similarly, we obtain the max-product update rule

$$m_j(s_j \mid \mathbf{r}_{|\mathcal{P}}) = \max_{\mathcal{C}_k(s_j)} \prod_{j'\in\mathcal{K}_{jk}} m_{j'}(s_{j'} \mid \mathbf{r}_{|\mathcal{P}_{j'}}), \tag{12.13}$$

where the notation is as in the sum-product update rule (12.8).

In practice, likelihoods are usually converted to log likelihoods, which converts products to sums and yields the max-sum algorithm. Or, log likelihoods may be converted to negative log likelihoods, which converts max to min and yields the min-sum algorithm. These variations are all trivially equivalent.

On a trellis, the forward part of any of these algorithms is equivalent to the Viterbi algorithm (VA). The update rule (12.13) becomes the add-compare-select operation, which is carried out at each state to determine the new metric $m_j(s_j \mid \mathbf{r}_{|\mathcal{P}})$ of each state. The VA avoids the backward part of the algorithm by also remembering the survivor history at each state, and then doing a traceback when it gets to the end of the trellis; this traceback corresponds in some sense to the backward part of the sum-product algorithm.

Exercise 2. Repeat Exercise 1, using the min-sum algorithm instead of the sum-product algorithm. Decode the same sequence using the Viterbi algorithm, and show how the two computations correspond. Decode the same sequence using Wagner decoding, and show how Wagner decoding relates to the other two methods.

12.4 The sum-product algorithm on graphs with cycles

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

One approach is to agglomerate the graph enough to eliminate the cycles, and then apply the sum-product algorithm, which will now be exact. The problem is that the complexity of decoding of a cycle-free graph of $\mathcal{C}$ cannot be significantly less than the complexity of decoding some trellis for $\mathcal{C}$, as we saw in Chapter 11. Moreover, as we saw in Chapter 10, the complexity of a minimal trellis for a sequence of codes with positive rates and coding gains must increase exponentially with code length.

A second approach is simply to apply the sum-product algorithm to the graph with cycles and hope for the best.

Because the sum-product rule is local, it may be implemented at any vertex of the graph, using whatever incoming messages are currently available. In a *parallel* or "flooding" schedule, the sum-product rule is computed at each vertex at all possible times, converting the incoming messages to a set of outgoing messages on all edges. Other schedules are possible, as we will discuss in the next chapter.

There is now no guarantee that the sum-product algorithm will converge. In practice, the sum-product algorithm converges with probability near 1 when the code rate is below some threshold which is below but near the Shannon limit. Convergence is slow when the code rate is near the threshold, but rapid when the code rate is somewhat lower. The identification of fixed points of the sum-product algorithm is a topic of current research.

Even if the sum-product algorithm converges, there is no guarantee that it will converge to the correct likelihoods or APPs. In general, the converged APPs will be too optimistic (overconfident), because they assume that all messages are from independent inputs, whereas in fact messages enter repeatedly into sum-product updates because of graph cycles. Consequently, decoding performance is suboptimal. In general, the suboptimality is great when the graph has many short cycles, and becomes negligible as cycles get long and sparse (the graph becomes "locally tree-like"). This is why belief propagation has long been considered to be inapplicable to most graphical models with cycles, which typically are based on physical models with inherently short cycles; in coding, by contrast, cycles can be designed to be very long with high probability.

A third approach is to beef up the sum-product algorithm so that it still performs well on certain classes of graphs with cycles. Because the sum-product algorithm already works so well in coding applications, this approach is not really needed for coding. However, this is a current topic of research for more general applications in artificial intelligence, optimization and physics.