48

Convolutional Codes and Turbo Codes

This chapter follows tightly on from Chapter 25. It makes use of the ideas of codes and trellises and the forward–backward algorithm.

▶ 48.1 Introduction to convolutional codes

When we studied linear block codes, we described them in three ways:

1. The generator matrix describes how to turn a string of K arbitrary source bits into a transmission of N bits.

2. The parity-check matrix specifies the $M = N - K$ parity-check constraints that a valid codeword satisfies.

3. The trellis of the code describes its valid codewords in terms of paths through a trellis with labelled edges.

A fourth way of describing some block codes, the algebraic approach, is not covered in this book (a) because it has been well covered by numerous other books in coding theory; (b) because, as this part of the book discusses, the state-of-the-art in error-correcting codes makes little use of algebraic coding theory; and (c) because I am not competent to teach this subject.

We will now describe convolutional codes in two ways: first, in terms of mechanisms for generating transmissions $\mathbf{t}$ from source bits $\mathbf{s}$; and second, in terms of trellises that describe the constraints satisfied by valid transmissions.

▶ 48.2 Linear-feedback shift-registers

We generate a transmission with a convolutional code by putting a source stream through a linear filter. This filter makes use of a shift register, linear output functions, and, possibly, linear feedback.

I will draw the shift-register in a right-to-left orientation: bits roll from right to left as time goes on.

Figure 48.1 shows three linear-feedback shift-registers which could be used to define convolutional codes. The rectangular box surrounding the bits $z_1 \ldots z_7$ indicates the *memory* of the filter, also known as its *state*. All three filters have one input and two outputs. On each clock cycle, the source supplies one bit, and the filter outputs two bits $t^{(a)}$ and $t^{(b)}$. By concatenating together these bits we can obtain from our source stream $s_1 s_2 s_3 \ldots$ a transmission stream $t_1^{(a)} t_1^{(b)} t_2^{(a)} t_2^{(b)} t_3^{(a)} t_3^{(b)} \ldots$. Because there are two transmitted bits for every source bit, the codes shown in figure 48.1 have rate $1/2$. Because

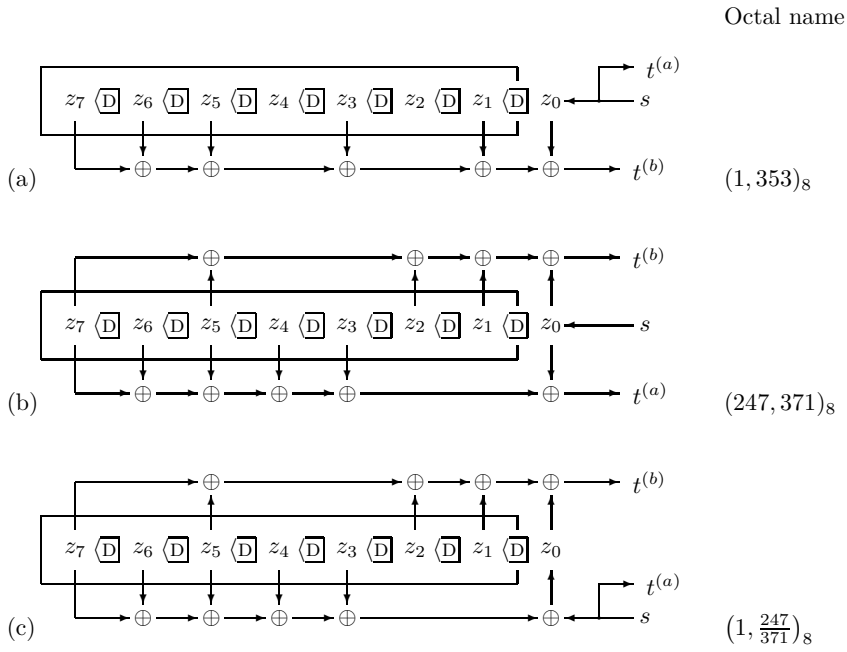


Figure 48.1. Linear-feedback shift-registers for generating convolutional codes with rate 1/2. The symbol ⟨D⟩ indicates a copying with a delay of one clock cycle. The symbol ⊕ denotes linear addition modulo 2 with no delay.
The filters are (a) systematic and nonrecursive; (b) nonsystematic and nonrecursive; (c) systematic and recursive.

these filters require $k = 7$ bits of memory, the codes they define are known as a *constraint-length 7 codes*.

Convolutional codes come in three flavours, corresponding to the three types of filter in figure 48.1.

Systematic nonrecursive

The filter shown in figure 48.1a has no feedback. It also has the property that one of the output bits, $t^{(a)}$, is identical to the source bit s. This encoder is thus called *systematic*, because the source bits are reproduced transparently in the transmitted stream, and *nonrecursive*, because it has no feedback. The other transmitted bit $t^{(b)}$ is a linear function of the state of the filter. One way of describing that function is as a dot product (modulo 2) between two binary vectors of length $k + 1$: a binary vector $\mathbf{g}^{(b)} = (1, 1, 1, 0, 1, 0, 1, 1)$ and the state vector $\mathbf{z} = (z_k, z_{k-1}, \ldots, z_1, z_0)$. We include in the state vector the bit z_0 that will be put into the first bit of the memory on the next cycle. The vector $\mathbf{g}^{(b)}$ has $g^{(b)}_\kappa = 1$ for every κ where there is a tap (a downward pointing arrow) from state bit z_κ into the transmitted bit $t^{(b)}$.

A convenient way to describe these binary tap vectors is in octal. Thus, this filter makes use of the tap vector 353_8. I have drawn the delay lines from right to left to make it easy to relate the diagrams to these octal numbers.

11	101	011
↓	↓	↓
3	5	3

Table 48.2. How taps in the delay line are converted to octal.

Nonsystematic nonrecursive

The filter shown in figure 48.1b also has no feedback, but it is not systematic. It makes use of two tap vectors $\mathbf{g}^{(a)}$ and $\mathbf{g}^{(b)}$ to create its two transmitted bits. This encoder is thus *nonsystematic* and *nonrecursive*. Because of their added complexity, nonsystematic codes can have error-correcting abilities superior to those of systematic nonrecursive codes with the same constraint length.

Systematic recursive

The filter shown in figure 48.1c is similar to the nonsystematic nonrecursive filter shown in figure 48.1b, but it uses the taps that formerly made up $\mathbf{g}^{(a)}$ to make a linear signal that is fed back into the shift register along with the source bit. The output $t^{(b)}$ is a linear function of the state vector as before. The other output is $t^{(a)} = s$, so this filter is systematic.

A recursive code is conventionally identified by an octal ratio, e.g., figure 48.1c's code is denoted by $(247/371)_8$.

Equivalence of systematic recursive and nonsystematic nonrecursive codes

The two filters in figure 48.1b,c are *code-equivalent* in that the *sets* of codewords that they define are identical. For every codeword of the nonsystematic nonrecursive code we can choose a source stream for the other encoder such that its output is identical (and *vice versa*).

To prove this, we denote by p the quantity $\sum_{\kappa=1}^{k} g_\kappa^{(a)} z_\kappa$, as shown in figure 48.3a and b, which shows a pair of smaller but otherwise equivalent filters. If the two transmissions are to be equivalent – that is, the $t^{(a)}$s are equal in both figures and so are the $t^{(b)}$s – then on every cycle the source bit in the systematic code must be $s = t^{(a)}$. So now we must simply confirm that for this choice of s, the systematic code's shift register will follow the same state sequence as that of the nonsystematic code, assuming that the states match initially. In figure 48.3a we have

$$t^{(a)} = p \oplus z_0^{\text{nonrecursive}} \tag{48.1}$$

whereas in figure 48.3b we have

$$z_0^{\text{recursive}} = t^{(a)} \oplus p. \tag{48.2}$$

Substituting for $t^{(a)}$, and using $p \oplus p = 0$ we immediately find

$$z_0^{\text{recursive}} = z_0^{\text{nonrecursive}}. \tag{48.3}$$

Thus, any codeword of a nonsystematic nonrecursive code is a codeword of a systematic recursive code with the same taps – the same taps in the sense that there are vertical arrows in all the same places in figures 48.3(a) and (b), though one of the arrows points up instead of down in (b).

Now, while these two codes are equivalent, the two encoders behave differently. The nonrecursive encoder has a *finite impulse response*, that is, if one puts in a string that is all zeroes except for a single one, the resulting output stream contains a finite number of ones. Once the one bit has passed through all the states of the memory, the delay line returns to the all-zero state. Figure 48.4a shows the state sequence resulting from the source string $\mathbf{s} = (0, 0, 1, 0, 0, 0, 0, 0)$.

Figure 48.4b shows the trellis of the recursive code of figure 48.3b and the response of this filter to the same source string $\mathbf{s} = (0, 0, 1, 0, 0, 0, 0, 0)$. The filter has an *infinite impulse response*. The response settles into a periodic state with period equal to three clock cycles.

▷ Exercise 48.1.[1] What is the input to the recursive filter such that its state sequence and the transmission are the same as those of the nonrecursive filter? (Hint: see figure 48.5.)

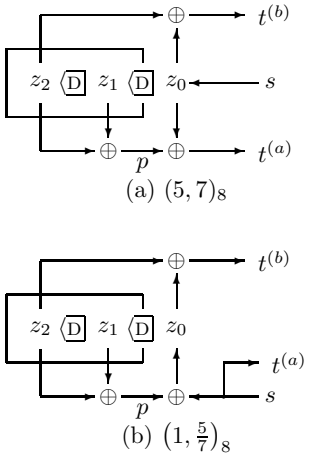
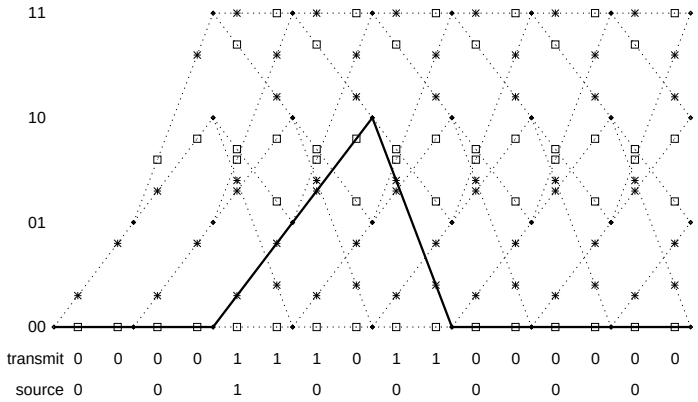


Figure 48.3. Two rate-1/2 convolutional codes with constraint length $k = 2$: (a) non-recursive; (b) recursive. The two codes are equivalent.

(a)



(b)

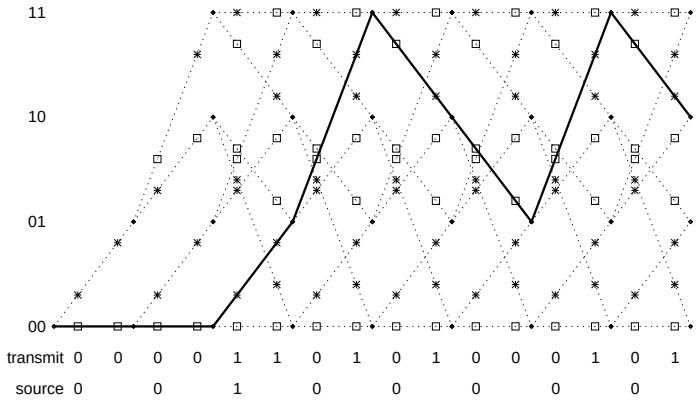


Figure 48.4. Trellises of the rate-1/2 convolutional codes of figure 48.3. It is assumed that the initial state of the filter is $(z_2, z_1) = (0, 0)$. Time is on the horizontal axis and the state of the filter at each time step is the vertical coordinate. On the line segments are shown the emitted symbols $t^{(a)}$ and $t^{(b)}$, with stars for '1' and boxes for '0'. The paths taken through the trellises when the source sequence is 00100000 are highlighted with a solid line. The light dotted lines show the state trajectories that are possible for other source sequences.

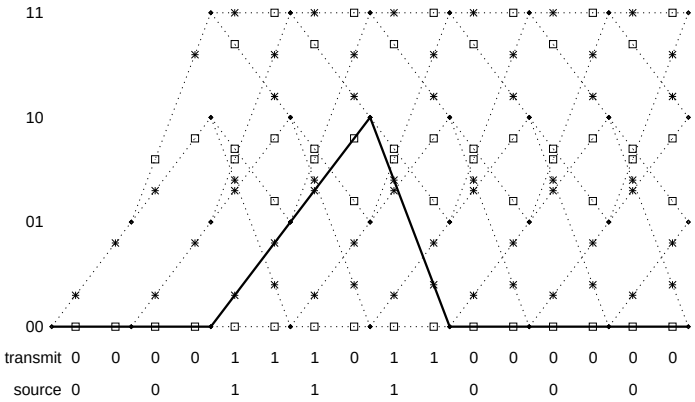


Figure 48.5. The source sequence for the systematic recursive code 00111000 produces the same path through the trellis as 00100000 does in the nonsystematic nonrecursive case.

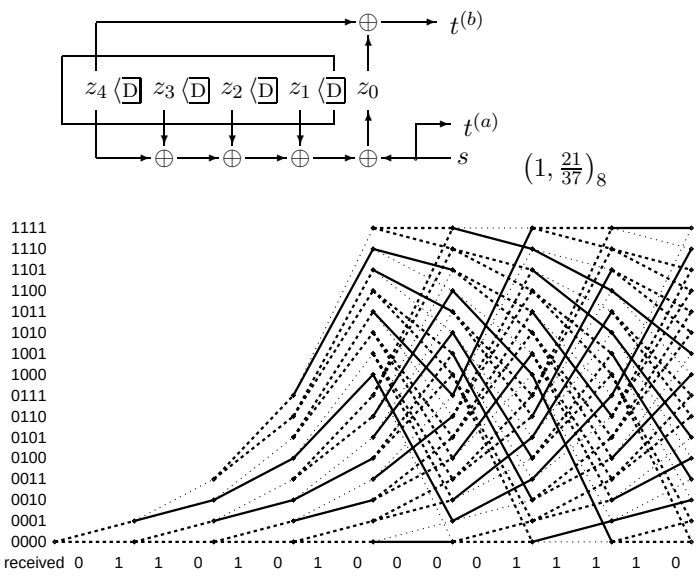


Figure 48.6. The trellis for a $k = 4$ code painted with the likelihood function when the received vector is equal to a codeword with just one bit flipped. There are three line styles, depending on the value of the likelihood: thick solid lines show the edges in the trellis that match the corresponding two bits of the received string exactly; thick dotted lines show edges that match one bit but mismatch the other; and thin dotted lines show the edges that mismatch both bits.

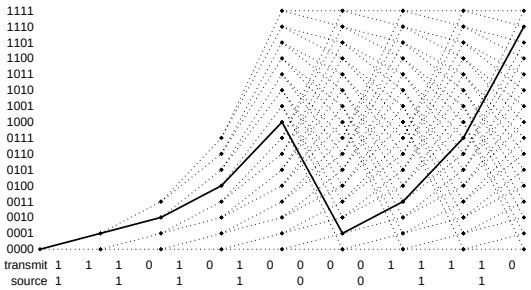
In general a linear-feedback shift-register with k bits of memory has an impulse response that is periodic with a period that is at most $2^k - 1$, corresponding to the filter visiting every non-zero state in its state space.

Incidentally, cheap pseudorandom number generators and cheap cryptographic products make use of exactly these periodic sequences, though with larger values of k than 7; the random number seed or cryptographic key selects the initial state of the memory. There is thus a close connection between certain cryptanalysis problems and the decoding of convolutional codes.

▶ 48.3 Decoding convolutional codes

The receiver receives a bit stream, and wishes to infer the state sequence and thence the source stream. The posterior probability of each bit can be found by the sum–product algorithm (also known as the forward–backward or BCJR algorithm), which was introduced in section 25.3. The most probable state sequence can be found using the min–sum algorithm of section 25.3 (also known as the Viterbi algorithm). The nature of this task is illustrated in figure 48.6, which shows the cost associated with each edge in the trellis for the case of a sixteen-state code; the channel is assumed to be a binary symmetric channel and the received vector is equal to a codeword except that one bit has been flipped. There are three line styles, depending on the value of the likelihood: thick solid lines show the edges in the trellis that match the corresponding two bits of the received string exactly; thick dotted lines show edges that match one bit but mismatch the other; and thin dotted lines show the edges that mismatch both bits. The min–sum algorithm seeks the path through the trellis that uses as many solid lines as possible; more precisely, it minimizes the cost of the path, where the cost is zero for a solid line, one for a thick dotted line, and two for a thin dotted line.

▷ Exercise 48.2.[1, p.581] Can you spot the most probable path and the flipped bit?



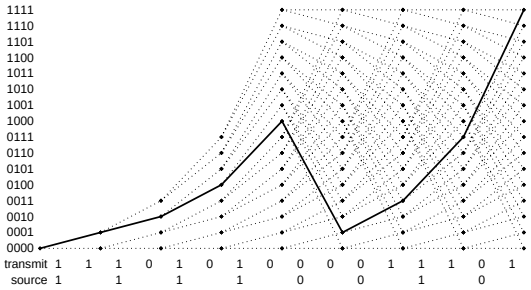


Figure 48.7. Two paths that differ in two transmitted bits only.

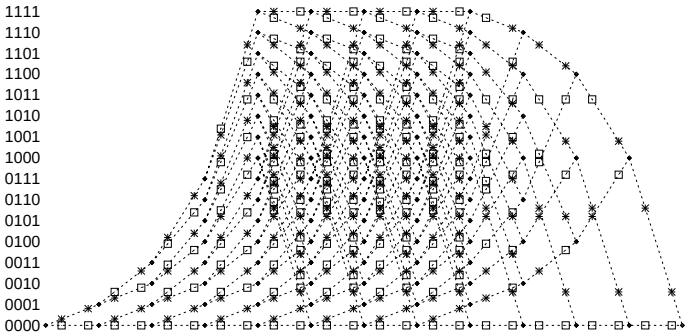


Figure 48.8. A terminated trellis. When any codeword is completed, the filter state is 0000.

Unequal protection

A defect of the convolutional codes presented thus far is that they offer unequal protection to the source bits. Figure 48.7 shows two paths through the trellis that differ in only two transmitted bits. The last source bit is less well protected than the other source bits. This unequal protection of bits motivates the *termination* of the trellis.

A terminated trellis is shown in figure 48.8. Termination slightly reduces the number of source bits used per codeword. Here, four source bits are turned into parity bits because the $k = 4$ memory bits must be returned to zero.

▶ 48.4 Turbo codes

An (N, K) turbo code is defined by a number of constituent convolutional encoders (often, two) and an equal number of *interleavers* which are $K \times K$ permutation matrices. Without loss of generality, we take the first interleaver to be the identity matrix. A string of K source bits is encoded by feeding them into each constituent encoder in the order defined by the associated interleaver, and transmitting the bits that come out of each constituent encoder. Often the first constituent encoder is chosen to be a systematic encoder, just like the recursive filter shown in figure 48.6, and the second is a non-systematic one of rate 1 that emits parity bits only. The transmitted codeword then consists of

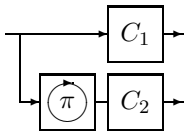


Figure 48.10. The encoder of a turbo code. Each box C_1, C_2, contains a convolutional code. The source bits are reordered using a permutation π before they are fed to C_2. The transmitted codeword is obtained by concatenating or interleaving the outputs of the two convolutional codes.

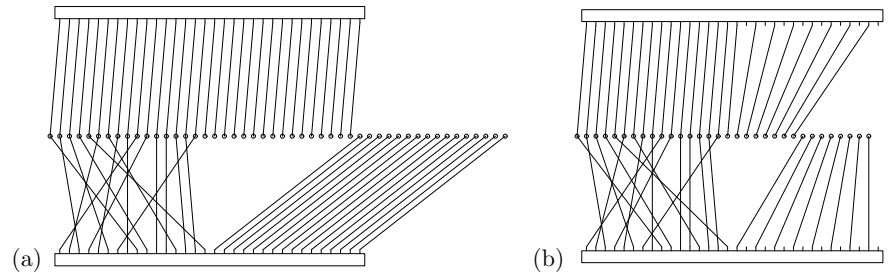


Figure 48.9. Rate-1/3 (a) and rate-1/2 (b) turbo codes represented as factor graphs. The circles represent the codeword bits. The two rectangles represent trellises of rate-1/2 convolutional codes, with the systematic bits occupying the left half of the rectangle and the parity bits occupying the right half. The puncturing of these constituent codes in the rate-1/2 turbo code is represented by the lack of connections to half of the parity bits in each trellis.

K source bits followed by M_1 parity bits generated by the first convolutional code and M_2 parity bits from the second. The resulting turbo code has rate $1/3$.

The turbo code can be represented by a factor graph in which the two trellises are represented by two large rectangular nodes (figure 48.9a); the K source bits and the first M_1 parity bits participate in the first trellis and the K source bits and the last M_2 parity bits participate in the second trellis. Each codeword bit participates in either one or two trellises, depending on whether it is a parity bit or a source bit. Each trellis node contains a trellis exactly like the terminated trellis shown in figure 48.8, except one thousand times as long. [There are other factor graph representations for turbo codes that make use of more elementary nodes, but the factor graph given here yields the standard version of the sum–product algorithm used for turbo codes.]

If a turbo code of smaller rate such as $1/2$ is required, a standard modification to the rate-$1/3$ code is to *puncture* some of the parity bits (figure 48.9b).

Turbo codes are decoded using the sum–product algorithm described in Chapter 26. On the first iteration, each trellis receives the channel likelihoods, and runs the forward–backward algorithm to compute, for each bit, the relative likelihood of its being `1` or `0`, given the information about the other bits. These likelihoods are then passed across from each trellis to the other, and multiplied by the channel likelihoods on the way. We are then ready for the second iteration: the forward–backward algorithm is run again in each trellis using the updated probabilities. After about ten or twenty such iterations, it's hoped that the correct decoding will be found. It is common practice to stop after some fixed number of iterations, but we can do better.

As a stopping criterion, the following procedure can be used at every iteration. For each time-step in each trellis, we identify the most probable edge, according to the local messages. If these most probable edges join up into two valid paths, one in each trellis, and if these two paths are consistent with each other, it is reasonable to stop, as subsequent iterations are unlikely to take the decoder away from this codeword. If a maximum number of iterations is reached without this stopping criterion being satisfied, a decoding error can be reported. This stopping procedure is recommended for several reasons: it allows a big saving in decoding time with no loss in error probability; it allows decoding failures that are detected by the decoder to be so identified – knowing that a particular block is definitely corrupted is surely useful information for the receiver! And when we distinguish between detected and undetected errors, the undetected errors give helpful insights into the low-weight codewords

of the code, which may improve the process of code design.

Turbo codes as described here have excellent performance down to decoded error probabilities of about 10^{-5}, but randomly-constructed turbo codes tend to have an *error floor* starting at that level. This error floor is caused by low-weight codewords. To reduce the height of the error floor, one can attempt to modify the random construction to increase the weight of these low-weight codewords. The tweaking of turbo codes is a black art, and it never succeeds in totalling eliminating low-weight codewords; more precisely, the low-weight codewords can be eliminated only by sacrificing the turbo code's excellent performance. In contrast, low-density parity-check codes rarely have error floors, as long as their number of weight–2 columns is not too large (cf. exercise 47.3, p.572).

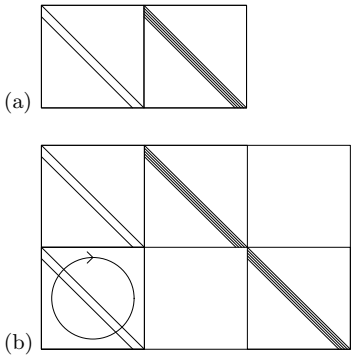


Figure 48.11. Schematic pictures of the parity-check matrices of (a) a convolutional code, rate 1/2, and (b) a turbo code, rate 1/3. Notation: A diagonal line represents an identity matrix. A band of diagonal lines represent a band of diagonal 1s. A circle inside a square represents the random permutation of all the columns in that square. A number inside a square represents the number of random permutation matrices superposed in that square. Horizontal and vertical lines indicate the boundaries of the blocks within the matrix.

▶ 48.5 Parity-check matrices of convolutional codes and turbo codes

We close by discussing the parity-check matrix of a rate-$1/2$ convolutional code viewed as a linear block code. We adopt the convention that the N bits of one block are made up of the $N/2$ bits $t^{(a)}$ followed by the $N/2$ bits $t^{(b)}$.

▷ Exercise 48.3.[2] Prove that a convolutional code has a low-density parity-check matrix as shown schematically in figure 48.11a.

Hint: It's easiest to figure out the parity constraints satisfied by a convolutional code by thinking about the nonsystematic nonrecursive encoder (figure 48.1b). Consider putting through filter a a stream that's been through convolutional filter b, and *vice versa*; compare the two resulting streams. Ignore termination of the trellises.

The parity-check matrix of a turbo code can be written down by listing the constraints satisfied by the two constituent trellises (figure 48.11b). So turbo codes are also special cases of low-density parity-check codes. If a turbo code is punctured, it no longer necessarily has a low-density parity-check matrix, but it always has a *generalized parity-check matrix* that is sparse, as explained in the next chapter.

Further reading

For further reading about convolutional codes, Johannesson and Zigangirov (1999) is highly recommended. One topic I would have liked to include is *sequential decoding*. Sequential decoding explores only the most promising paths in the trellis, and backtracks when evidence accumulates that a wrong turning has been taken. Sequential decoding is used when the trellis is too big for us to be able to apply the maximum likelihood algorithm, the min–sum algorithm. You can read about sequential decoding in Johannesson and Zigangirov (1999).

For further information about the use of the sum–product algorithm in turbo codes, and the rarely-used but highly recommended stopping criteria for halting their decoding, Frey (1998) is essential reading. (And there's lots more good stuff in the same book!)

▶ 48.6 Solutions

Solution to exercise 48.2 (p.578). The first bit was flipped. The most probable path is the upper one in figure 48.7.