

Lecture 16: Viterbi Decoding of Convolutional codes

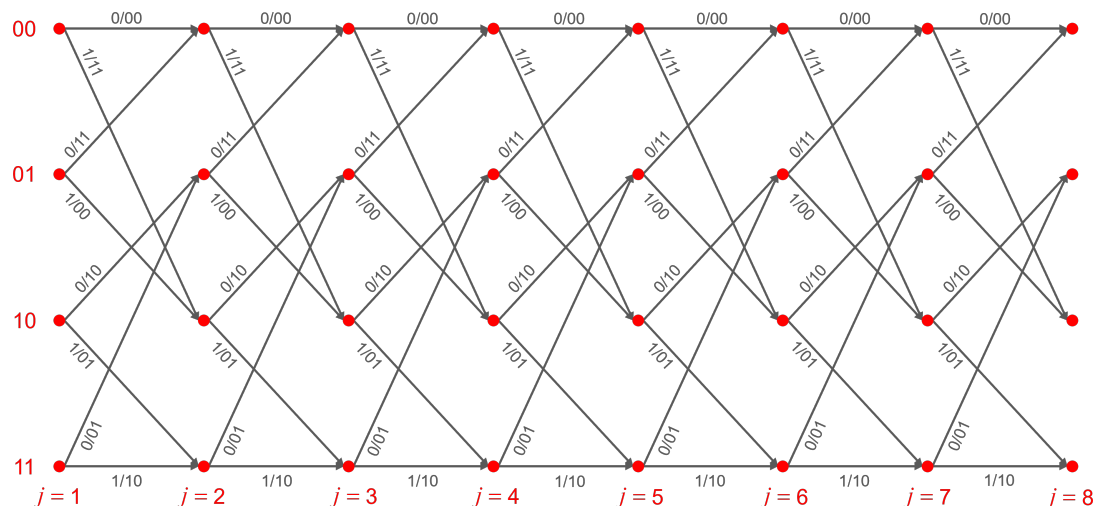
ENGR 76 lecture notes — May 21, 2026

David Tse¹, Stanford University

In the last lecture, we introduced the concept of convolutional codes and discussed how the encoding works. Today we turn to the problem of decoding. But first it is useful to introduce yet another representation of convolutional codes: the trellis.

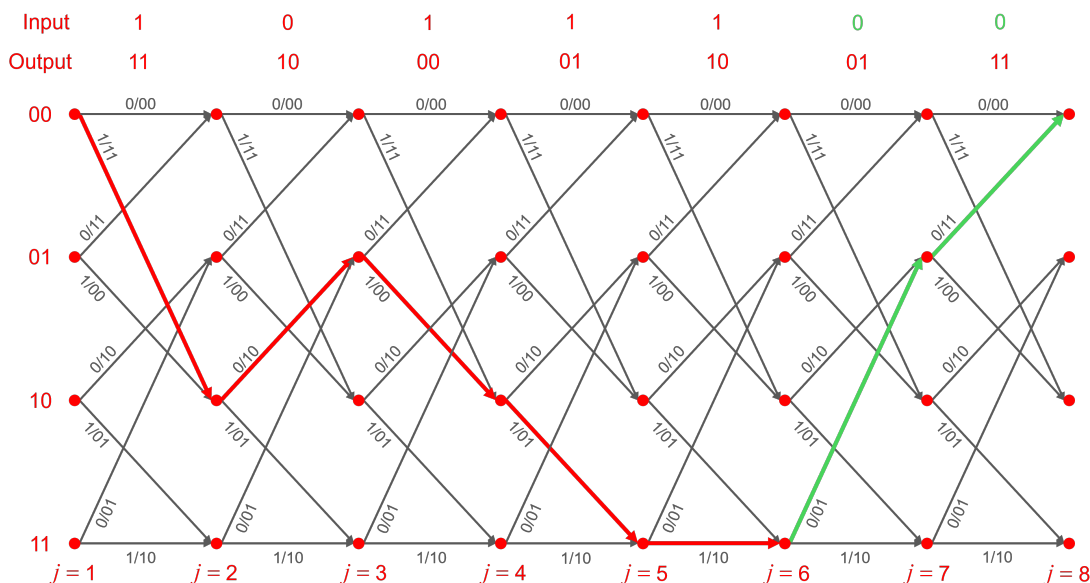
1 Trellis

The finite state machine discussed in the last lecture is a static view of the convolutional encoding process, i.e., it doesn't show the time evolution of the encoding process. It is also useful to study a more dynamic view showing the time evolution of the encoder. This is done using a trellis as shown below. Each row in the trellis shows a state and each column shows a time instant. The arrows show the same transitions as in the state diagram above, but now the transitions go from time j to time $j + 1$.

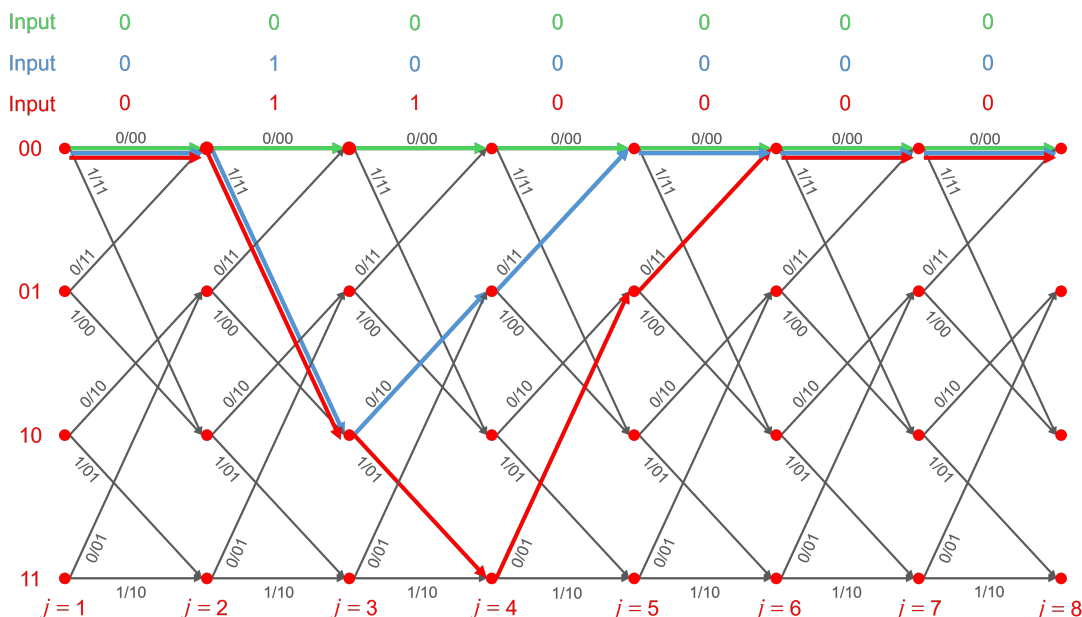


The encoding process can be thought of as following a path on the trellis, as shown below. Here the last two bits (in green) are so-called *padding bits* which are 0s added to the information bits to ensure that the final state is 00. Putting that aside for now, we see that the encoding process is simply a path on the trellis showing the sequence of states and the output sequence.

¹These notes are adapted from the 2025 Spring offering by Ayfer Özgür



Finally, we can use the trellis to visualize the error correction capabilities of the code, by looking at the state sequences for similar input sequences. In the figure below, we see that the all-zero input sequence produces an all-zero output (green). Even a single one in the input sequence (blue) leads to a significant deviation from the all zeros state sequence, with the output sequence now having 5 ones. This suggests that the convolutional code has a high minimum distance and strong error correction capabilities. We will look more deeply into the decoding process in the next section.



2 Minimum-distance Decoding

Over the past couple weeks, we have looking into a few families of codes, including the repetition code, the Hamming code and the convolutional code. Comparing the codes, we note that the repetition code

$\{000, 111\}$ had rate $\frac{1}{3}$ and distance 3. The code with $M = 4, L = 5$ ($\{00000, 11011, 11100, 00111\}$) had rate $\frac{2}{5}$ and distance 3. The Hamming code with $M = 16, L = 7$ had rate $\frac{4}{7}$ and distance 3. And finally, the convolutional code has rate $\frac{1}{2}$ and distance 5. Comparing to the repetition code of rate $\frac{1}{2}$ ($\{00, 11\}$) which has distance 2, we observe that the convolutional code has quite good distance properties. The rate being $\frac{1}{2}$ means that the code maps k information bits to $2k$ codeword bits (ignoring the padding of 2 dummy bits at the end for simplicity). That is to say, we use only 2^k codewords of length $2k$ out of a total of 2^{2k} binary sequences of that length. Thus, the fraction of sequences that are valid codewords is very small, just $\frac{1}{2^k}$. The fact that these codewords are well-separated is the basis for the error correction capabilities of the code.

Now, thinking about the decoding process, we can look back at the repetition code $\{000, 111\}$. Recall that we used majority vote decoding, which is the same as saying that we decode the received sequence to the closest codeword in Hamming distance.

The minimum (Hamming) distance decoding rule can be applied to any code, not only the repetition code. It does not guarantee to output the originally transmitted codeword if the number of errors is very high (just like majority voting fails if more than half of the bits are flipped). However, it has this nice property: if the number of errors is less than half of d_{min} of the code, then minimum-distance decoding always returns the transmitted codeword. This follows from the fact that the Hamming distance is a metric that satisfies the triangular inequality.

In the case of convolutional codes, the decoder receives a sequence of length $n = 2k$ (ignoring padding) bits which may contain the bit flips, and the minimum distance decoding task is as follows.

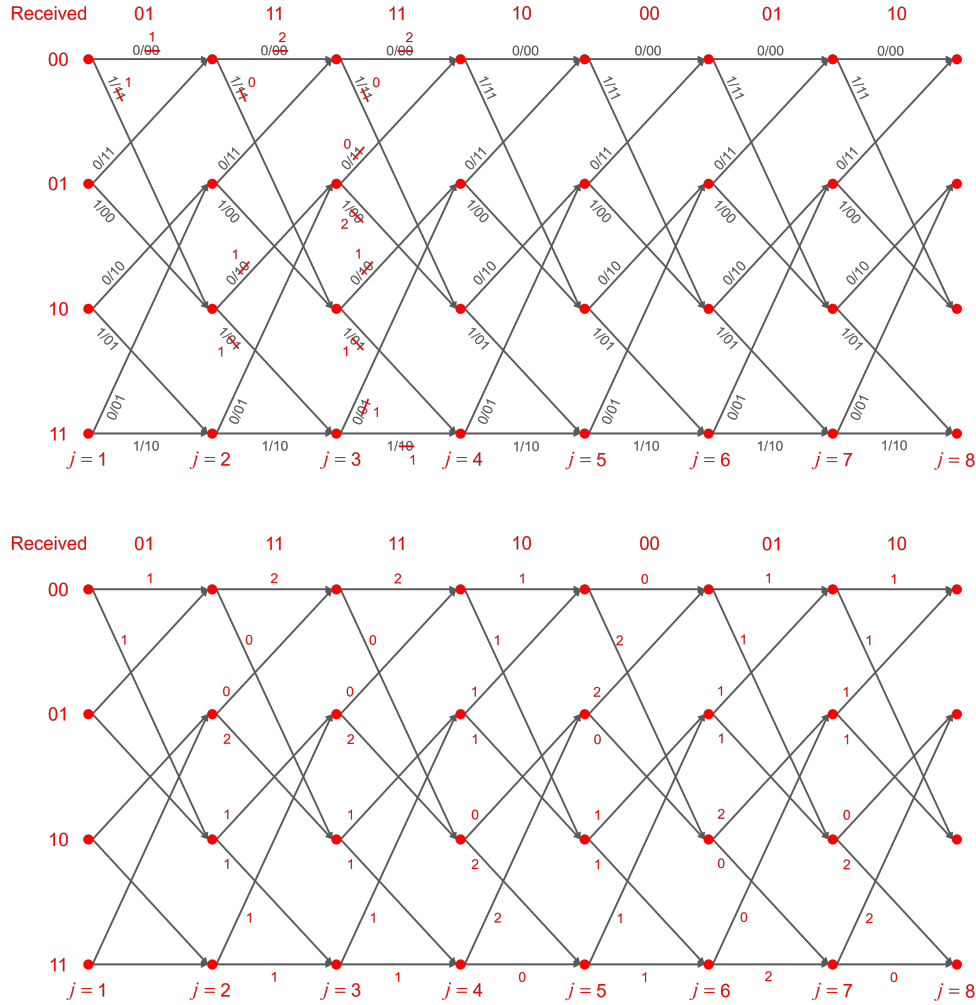
Decoding task: Among the 2^k possible codewords find the one that is closest in Hamming distance to the received sequence.

One possible way to achieve this is to use the **brute-force approach** where we first list all 2^k possible codewords (legitimate outputs of the encoder) of length $n = 2k$ bits. Then we compare each of them to the received sequence (where comparing means computing the Hamming distance), and pick the codeword with the minimum Hamming distance. This is a very general approach that can be used for any code, but unfortunately it's not computationally tractable. If we set $k = 100$, that means we have $M = 2^{100} \approx 10^{30}$ codewords. A modern VLSI chip that does 10^9 comparisons per second would still take 10^{21} seconds (4×10^{13} years!) to perform the comparisons, which is longer than the age of the universe (2×10^{10} years). And in practice, we often want to use block sizes in the thousands or even tens of thousands of bits. Thus, we clearly need a better decoding approach, which we present next.

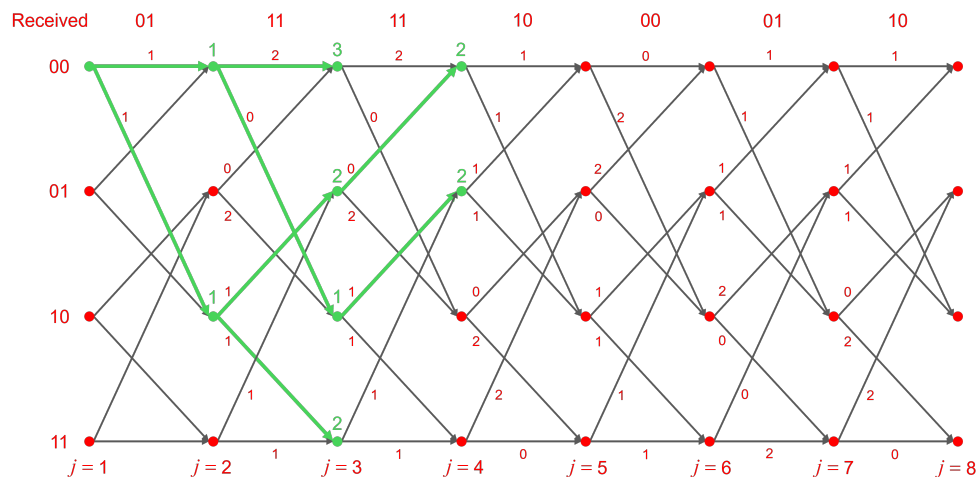
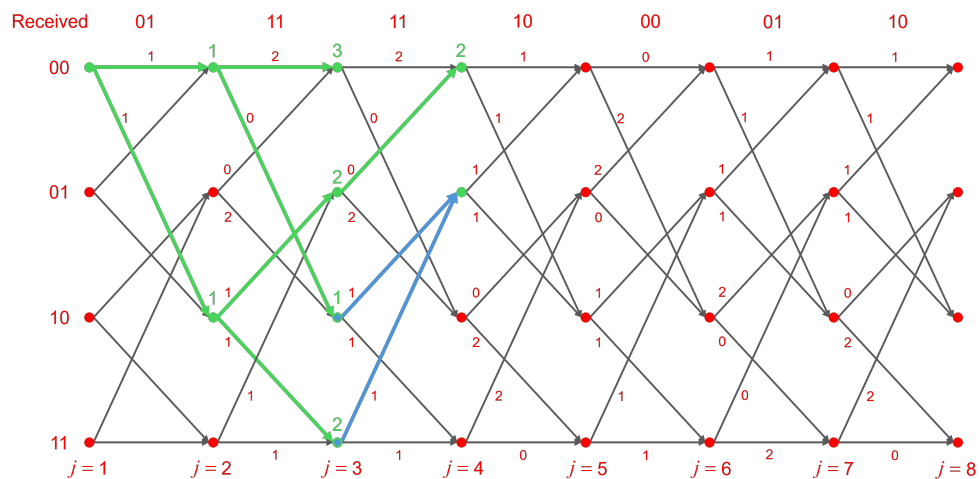
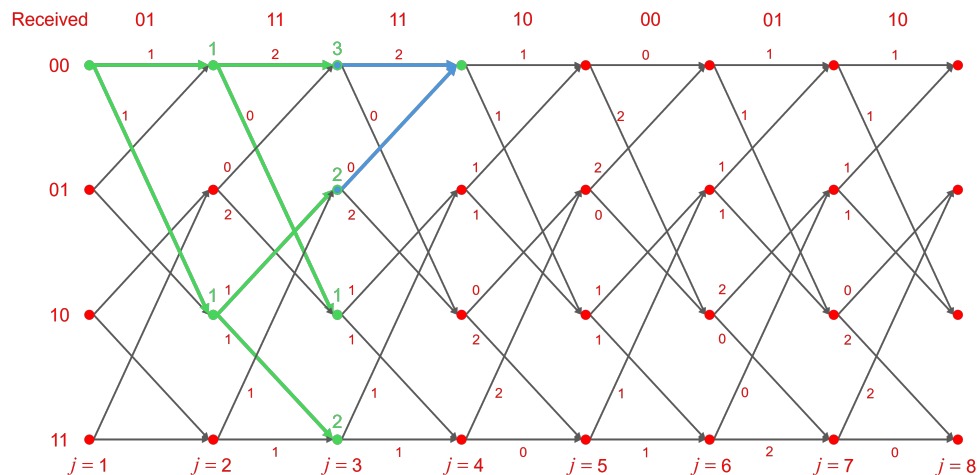
3 Viterbi decoding

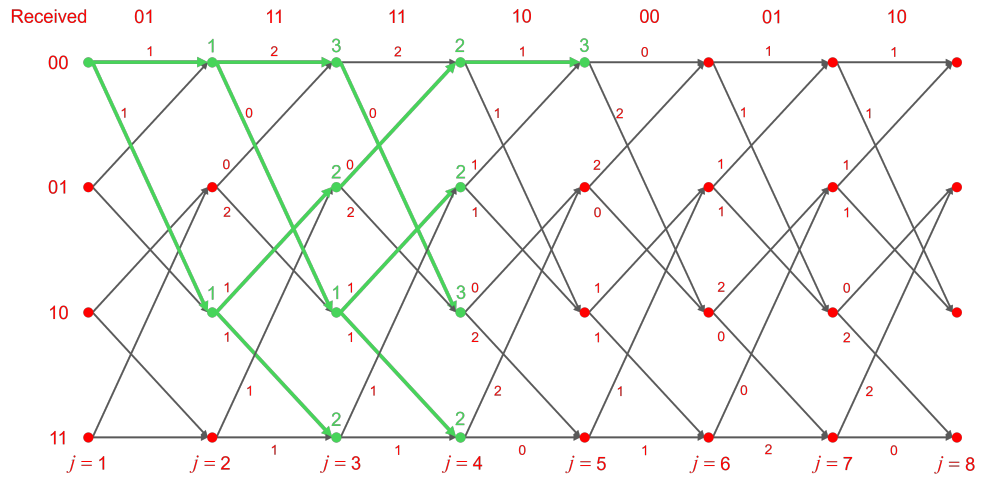
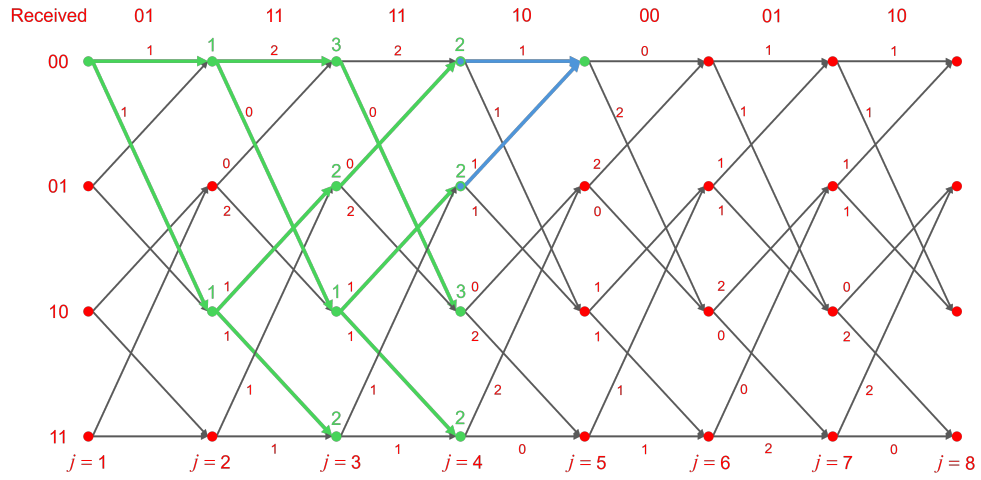
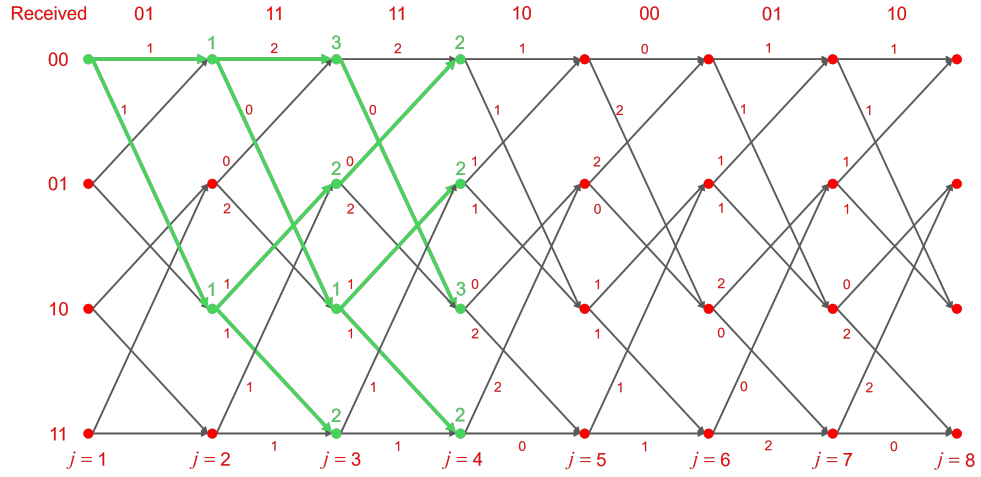
Now we describe an efficient decoding algorithm named after Andrew Viterbi who discovered this algorithm and co-founded Qualcomm. The decoding is best explained using the trellis representation from above. In class, we looked at an example with $k = 5$ message bits. Recall that we pad two 0s at the end of the message bits which means the trellis begins and ends at the state 00.

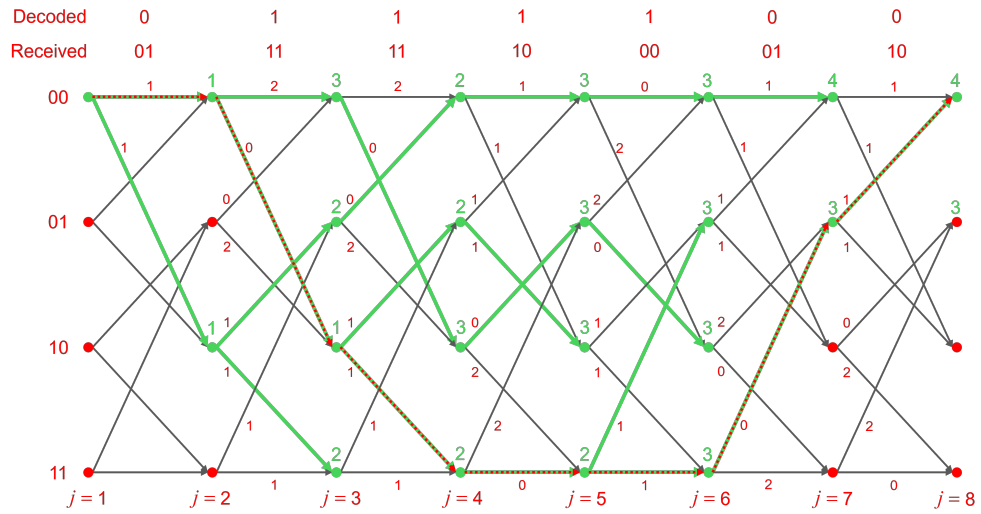
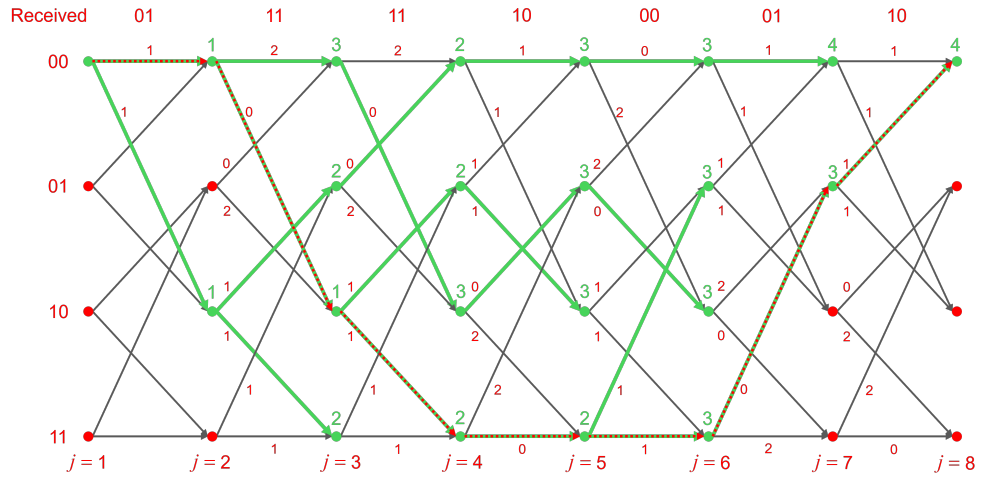
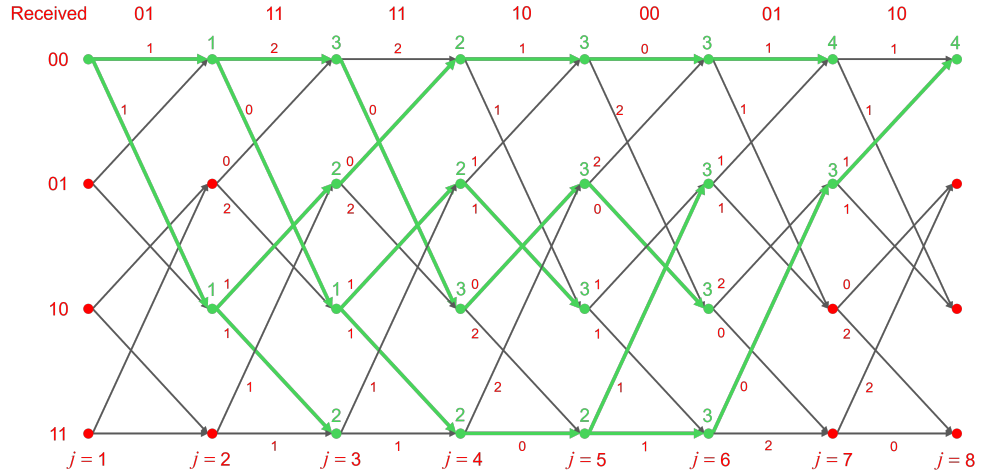
Step I: Divide the received sequence into blocks of 2 bits (since each step generates 2 output bits). For each edge on the trellis compute the Hamming distance between the 2 output bits and the corresponding received bits. This quantity is called the edge metric (also called the branch metric in the lecture slides). The figures below illustrate this process for an example. Given this, our aim is to find the path with the least total score starting from the state 00 at $j = 1$ and ending at state 00 at the last j .



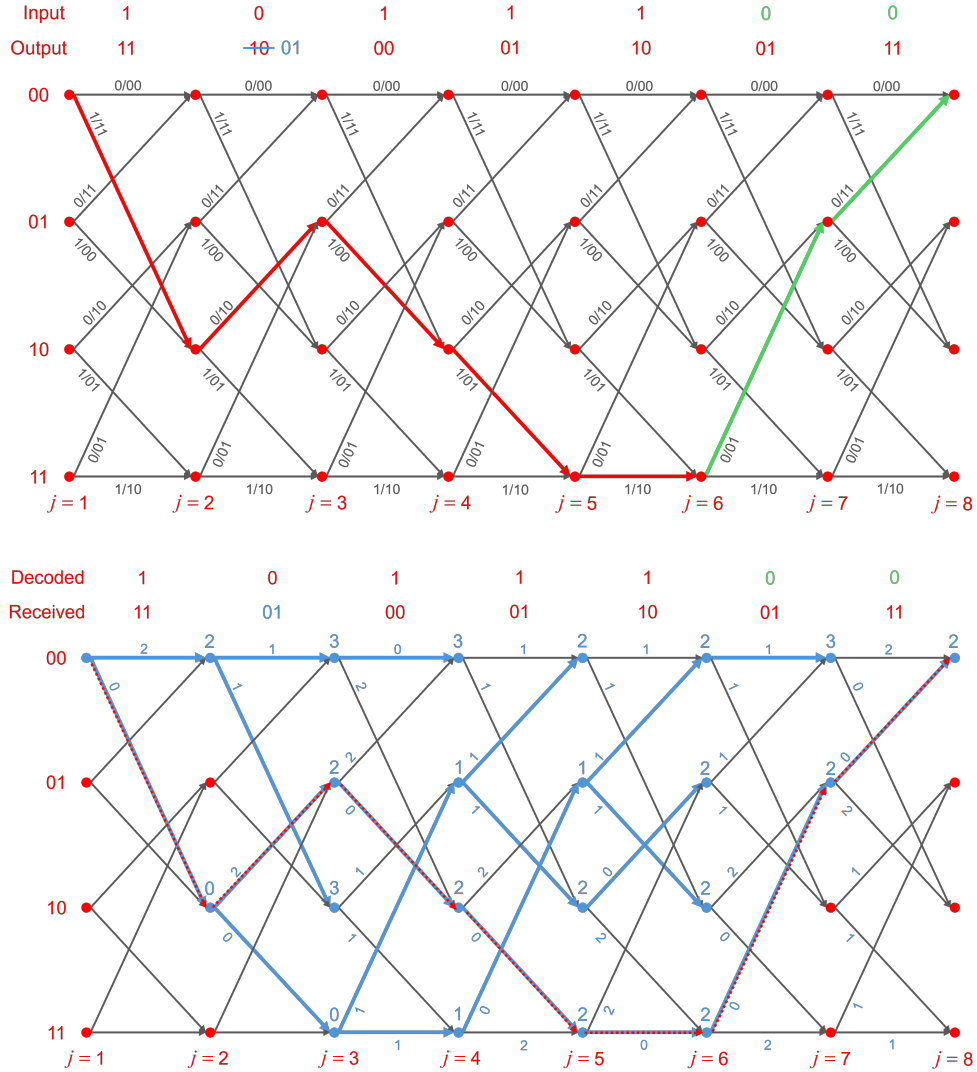
Step II: We proceed sequentially: first we assign a score of 0 to the state 00 at $j = 1$ (to denote the initial state). Then we proceed with $j = 2$ onward. For each value of j , we go through each state and look at the two incoming edges (from $j - 1$) to that state. For the two edges we compute the total score as the score of the previous state plus the edge metric. Then we select the edge with the lower score and assign that score to the state at j . This way, we always keep track of the best path ending at each state at each j . The score of a state at a particular time is the cost of the shortest path from the initial state to this state (it is also called the path metric in the slides). If we encounter a tie, i.e., two incoming edges leading to the same score, we make an arbitrary choice. Note that the first two and last two stages are a bit different since the set of allowed states is restricted. At the end of the process, we look at the best path ending at 00 and from that we can read off the decoded message. The following figures illustrate some steps of this process.







The figures below show another example, where we see that the code is able to correct two bit flips.



Computational complexity

Note that this process is not brute force decoding as we do not look at all possible codewords, instead we restrict attention to the best possibilities at each step. We can compute the number of operations done:

- In step I, we do 8 Hamming distance computations for binary sequences of length 2, for each j . This is $8k$ operations.
- In step II, we need to do 8 additions for each j to get the candidate scores. This is $8k$ operations.
- In step II, we need to do 4 comparisons for each j to get the best candidate for each state. This is $4k$ operations.

Overall, we see that this is linear complexity in k , and not exponential as in the case of brute force decoding.

Dynamic programming

Viterbi decoding is an instance of dynamic programming which is a class of algorithms. Dynamic programming comes up in the context of NLP (natural language processing) as well as several graph algorithms. In

particular, Viterbi decoding is closely connected to the problem of finding the shortest path between two nodes on a graph, and the famous Dijkstra's algorithm for solving it (see animation in slides).

Optimality

Viterbi algorithm guarantees that we find the closest codeword to the received sequence (one of the codewords if it's not unique).

We'll see that this is the optimal thing to do when the bits are flipped independently and randomly with the probability of each bit flip being less than $\frac{1}{2}$ in the following lectures.

Generalizations

We have looked at a very specific convolutional code till now, but these can be generalized by varying the following parameters:

- The number k_0 of source symbols entering the encoder at each step. This was 1 for our code, but in general can be higher.
- The number n_0 of output symbols produced by the encoder at each step. This was 2 for our code, and this together with k_0 determines the rate of the code.
- The constraint length m_0 : the number of bits used to determine an output n_0 tuple. This was 3 in our case (one more than the length of the state). This is an important parameter that influences the error correction capability of the code. This also influences the computational complexity of decoding which is linear in the number of states, and hence exponential in the constraint length.
- The encoding function specifying how the output bits are generated from the bits.