Channel Coding for IEEE 802.16e Mobile WiMAX

Matthew C. Valenti

Lane Department of Computer Science and Electrical Engineering
West Virginia University
U.S.A.

June 2009

Outline

Outline

IEEE 802.16

IEEE 802.16 is a family of standards for Wireless MAN's.

- Metropolitan area networks.
- Wireless at broadband speeds.

Applications of IEEE 802.16

- Wireless backhaul.
- Residential broadband.
- Cellular-like service.

Progression of standards

- 802.16
  - → December 2001.
  - → 10-66 GHz.
  - → Line-of-sight (LOS) only.
  - → Up to 134.4 Mbps operation using 28 MHz bandwidth.
- 802.16-2004
  - → June 2004.
  - → Added 2-11 GHz non-LOS operation.
  - → Up to 75 Mbps operation using 15 MHz bandwidth.
- 802.16e-2005
  - → December 2005.
  - → Added support for mobility.

Key Technologies

Advanced technologies supported by IEEE 802.16

- OFDM and OFDMA.
- Adaptive modulation: QPSK, 16-QAM, or 64-QAM.
- Adaptive turbo and LDPC codes.
- Hybrid-ARQ
- MIMO: Space-time codes and spatial multiplexing.
- Time-division duplexing.
- Multiuser diversity.
- Partial frequency reuse.

WiMAX Forum

The WiMAX forum is an consortium of over 500 companies whose purpose is to commercialize systems based on IEEE 802.16 technology.

The activities of the WiMAX forum include:

- Development of WiMAX system profiles.
- Certification of equipment.
- Standardization of higher-layer functionality.,

WiMAX vs. mobile WiMAX

WiMAX

- fixed system profile.
- OFDM PHY from IEEE 802.16-2004.
- 256 OFDM subcarriers (fixed).
- 3.5 MHz bandwidth.

mobile WiMAX

- mobility system profile.
- OFDMA PHY from IEEE 802.16e-2004.
- 128 to 2,048 subcarriers (scalable).
- 1.25 to 20 MHz bandwidths

Channel Codes Specified in IEEE 802.16e

Four codes are specified in IEEE 802.16e.

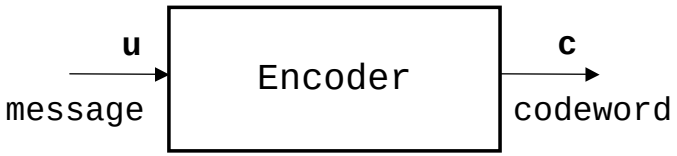
1. Tailbiting convolutional code.
2. Block turbo code (BTC).
3. Convolutional turbo code (CTC).
4. Low-density Parity-check (LDPC) code.

The goal of the remainder of this tutorial is to describe each of these codes in detail.

Outline

Code and Message Vectors

- A *binary code* $\mathcal{C}$ is a set of 2^k codewords $\mathbf{c}_i, 0 \leq i < 2^k$.
- Each *codeword* is represented by a length n binary vector.
- Each codeword is associated with a unique *message* $\mathbf{u}_i, 0 \leq i < 2^k$, which is a length k binary vector.
- The code must define the mapping from messages to codewords $\mathbf{u} \longrightarrow \mathbf{c}$.

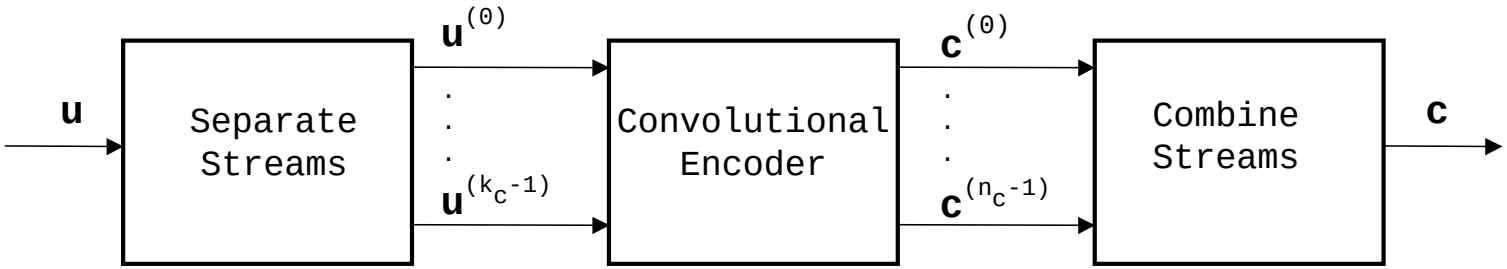


Linear codes.

- A code is *linear* if the modulo-2 sum of any two codewords is also a codeword.
  - Mathematically, if $\mathbf{c}_i \in \mathcal{C}$ and $\mathbf{c}_j \in \mathcal{C}$, then $\mathbf{c}_i + \mathbf{c}_j \in \mathcal{C}$.
  - Note that the addition is modulo-2.
- Because $\mathbf{c}_i + \mathbf{c}_i = \mathbf{0}$, it follows that all linear codes must contain the all-zeros codeword.
- All codes considered in this tutorial are linear.

Encoding

- A *convolutional encoder* is a device with k_c inputs and n_c outputs, where $n_c \geq k_c$.
- The input message $\mathbf{u}$ is split into k_c input *streams* $\mathbf{u}^{(i)}, 0 \leq i \leq k_c - 1$ each of length k/k_c.
- Similarly, the output codeword $\mathbf{c}$ is assembled from n_c output streams $\mathbf{c}^{(j)}, 0 \leq j \leq n_c - 1$ each of length n/n_c.
- In this tutorial, $1 \leq k_c \leq 2$ and $1 \leq n_c \leq 4$.



Convolutional Encoding when $k_c = 1$

- Suppose there is one input stream, $\mathbf{u}^{(0)} = \mathbf{u}$.
- Output stream $\mathbf{c}^{(j)}$ is found by convolving the input stream with a *generator sequence* $\mathbf{g}^{(j)}$ as follows:

$$\mathbf{c}^{(j)} = \mathbf{u} * \mathbf{g}^{(j)}$$

where the ℓ^{th} element of the output vector is

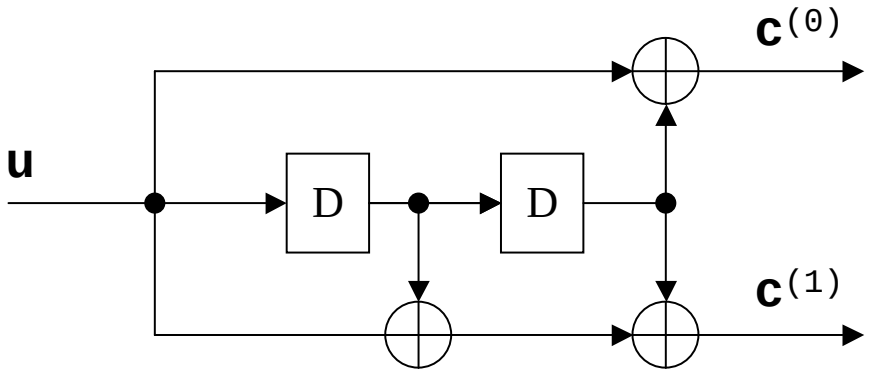
$$c_\ell^{(j)} = \sum_{k=0}^{m} u_{\ell-k} g_k^{(j)}$$

and the addition is modulo-2.

- m is the *memory* of the encoder.
- $\nu = m + 1$ is the *constraint length*.

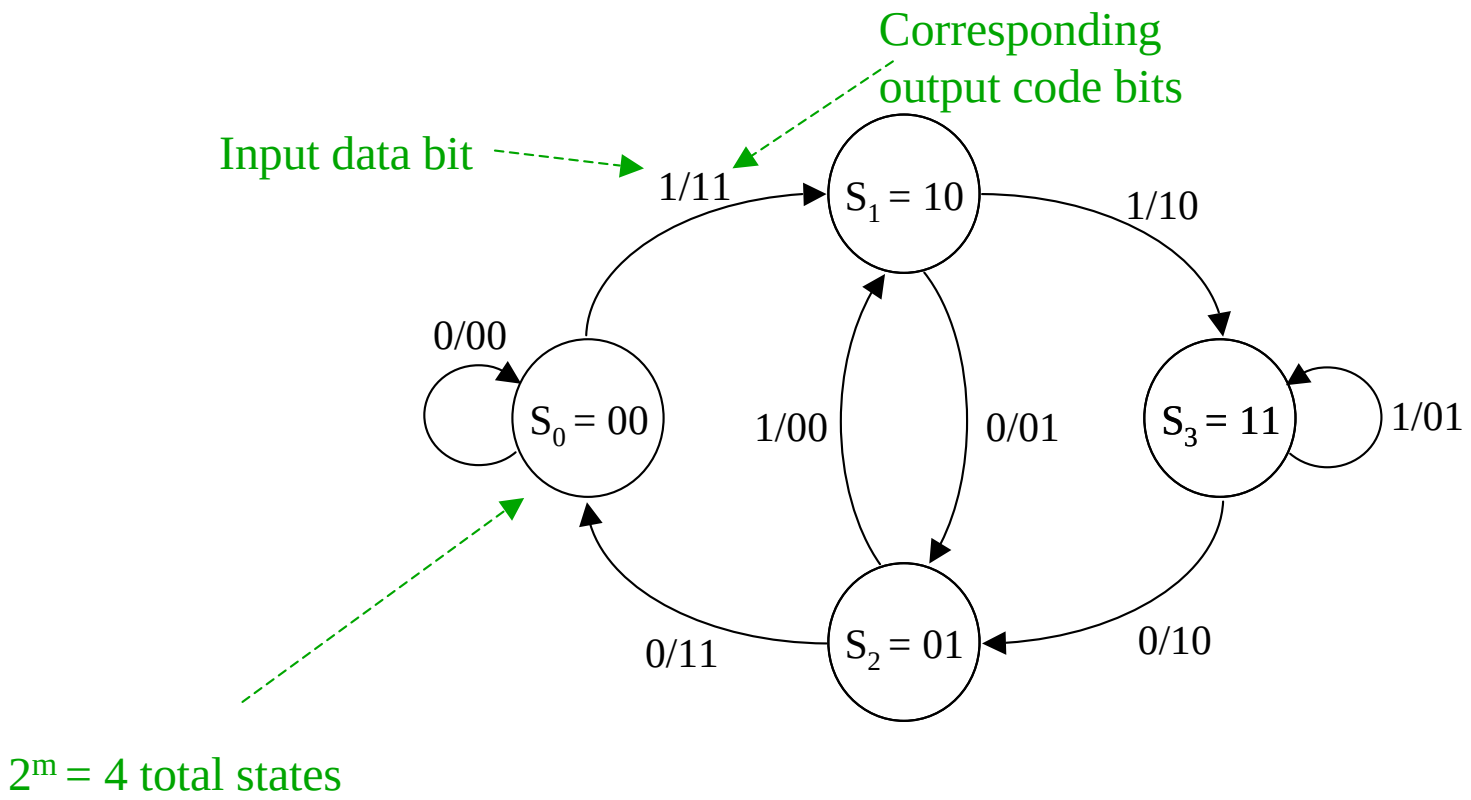
Encoder Diagram

- Let $\mathbf{g}^{(0)} = [101]$ and $\mathbf{g}^{(1)} = [111]$.
- The encoder may be realized with the following structure:



State diagram representation

A convolutional encoder is a finite state machine, and can be represented in terms of a state diagram.

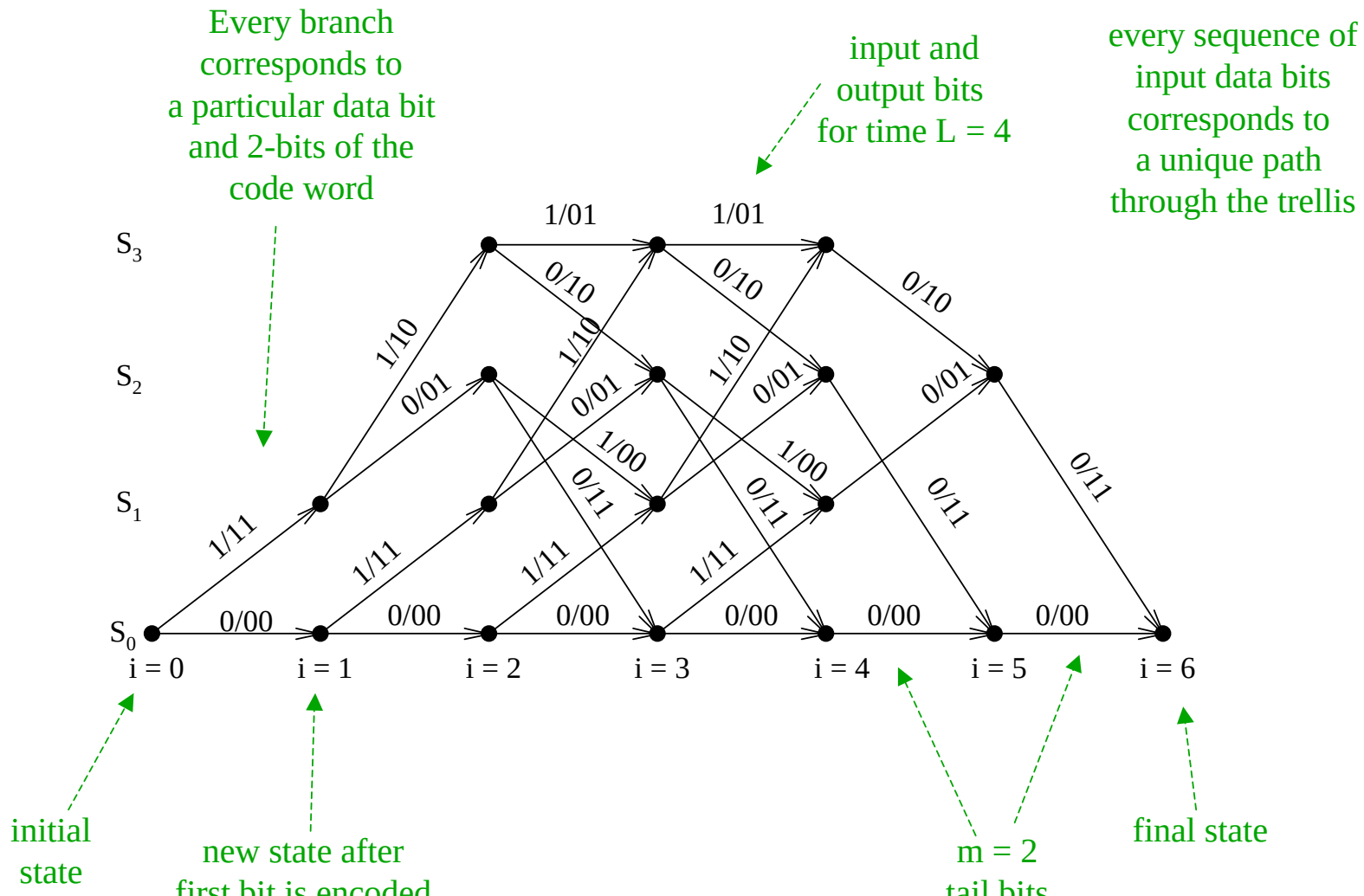


Initial and Terminating States

- The system needs a policy for choosing the initial and terminating states of the encoder.
- The usual convention is to start in the all-zeros state and then force the encoder to terminate in the all-zeros state.
- Termination in the all-zeros state requires a *tail* of m zeros.
- The tail results in a *fractional-rate loss*.
- *Tailbiting convolutional codes* operate such that the initial and terminating states are the same (but not necessarily all-zeros).
  - Tailbiting codes don't require a tail and have no fractional-rate loss.
  - *More on tailbiting codes later...*

Trellis representation

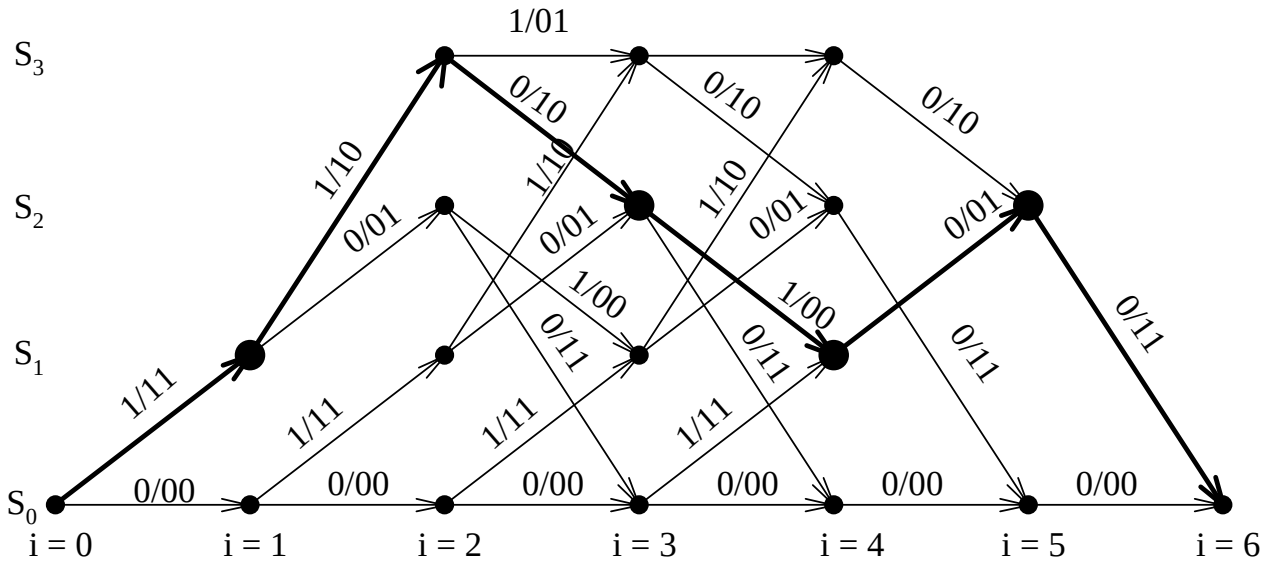
A *trellis* is an expansion of the state diagram which explicitly shows the passage of time.



Encoding Using the Trellis

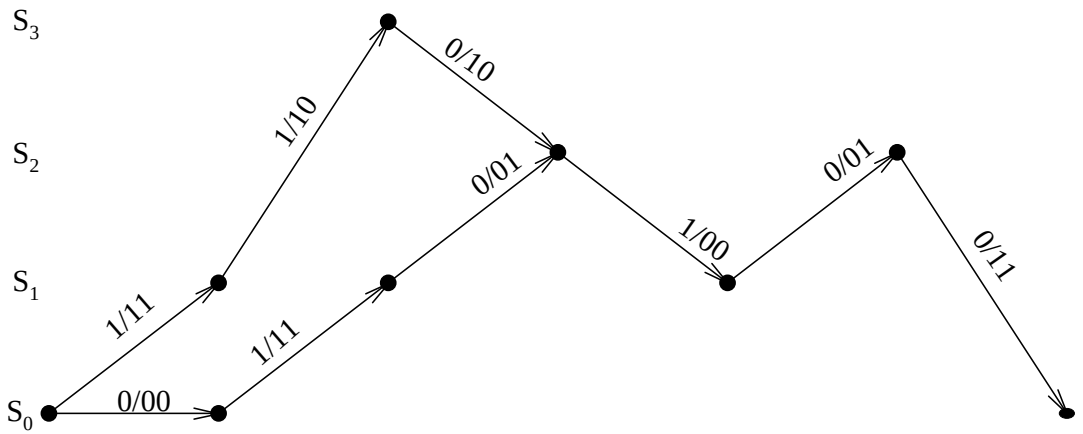
The trellis can be used to encode the message.
Use message bits to determine path, then read off the code bits.

`u = [1 1 0 1]`



`c = [1 1 1 0 1 0 0 0 0 1 1 1]`

The Viterbi Algorithm



- The Viterbi algorithm is used for ML Decoding.
- Exploiting the recursive structure of the trellis minimizes complexity.
- Steps:
  - A *forward sweep* through the trellis is performed.
  - Each node holds a *partial path metric*.
  - A *branch metric* is computed for each branch in the trellis.
  - At each node, an *add-compare-select* operation is performed.
  - Once the end of the trellis is reached, a traceback operation determines the value of the data bits.

Viterbi Algorithm: Example

- Suppose that the input to the convolutional encoder is:

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

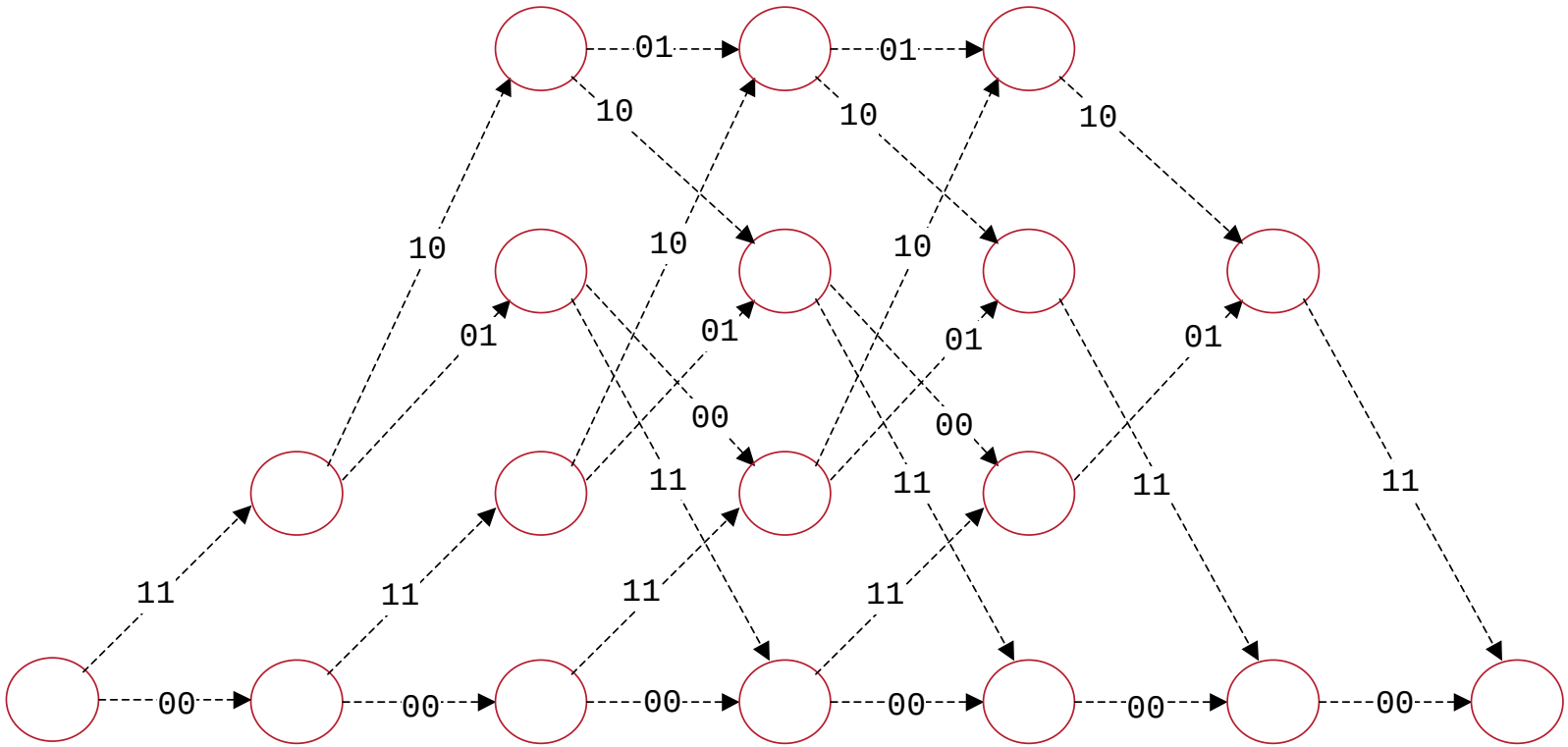
- Then the ouput of the encoder is:

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

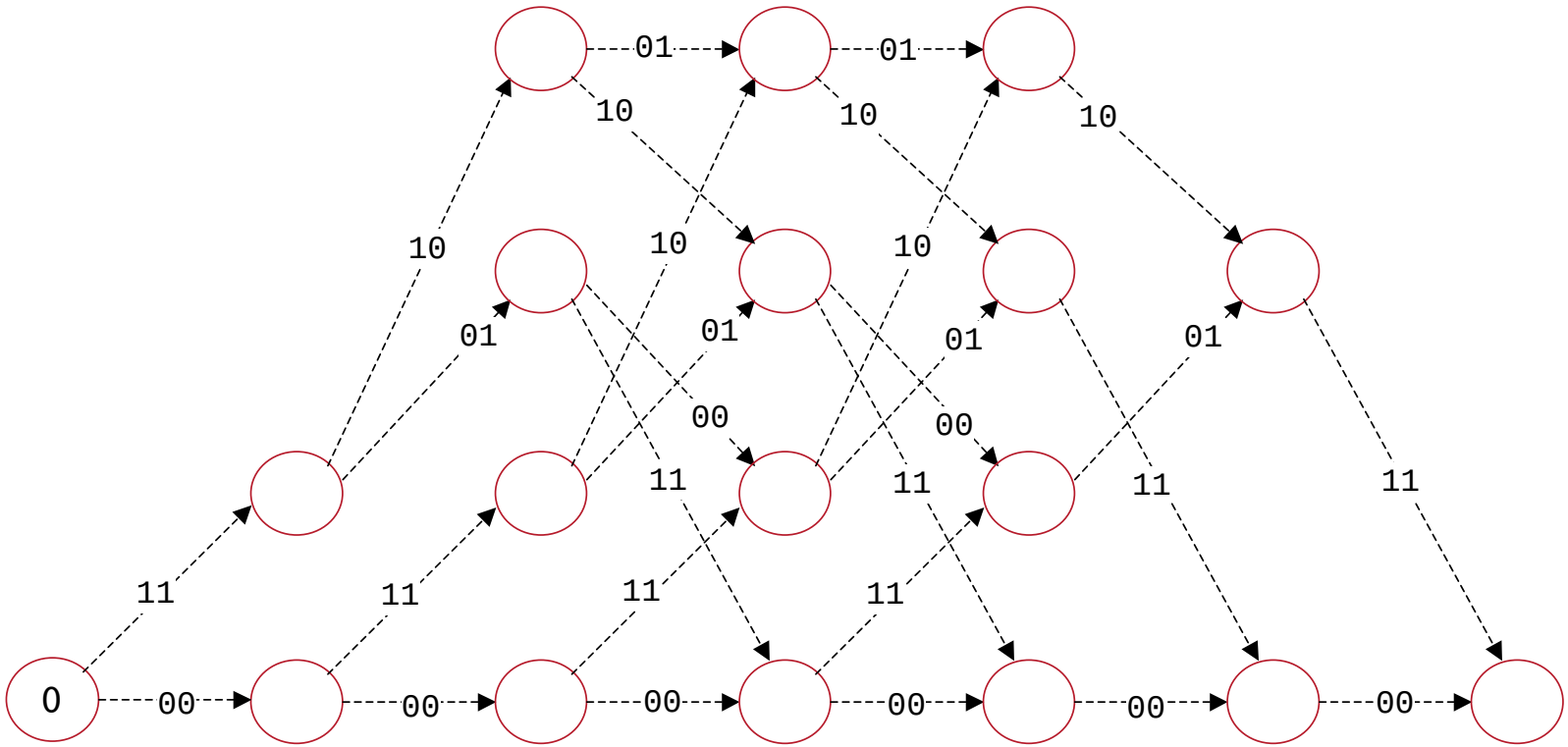
- Suppose every fourth bit is received in error:

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

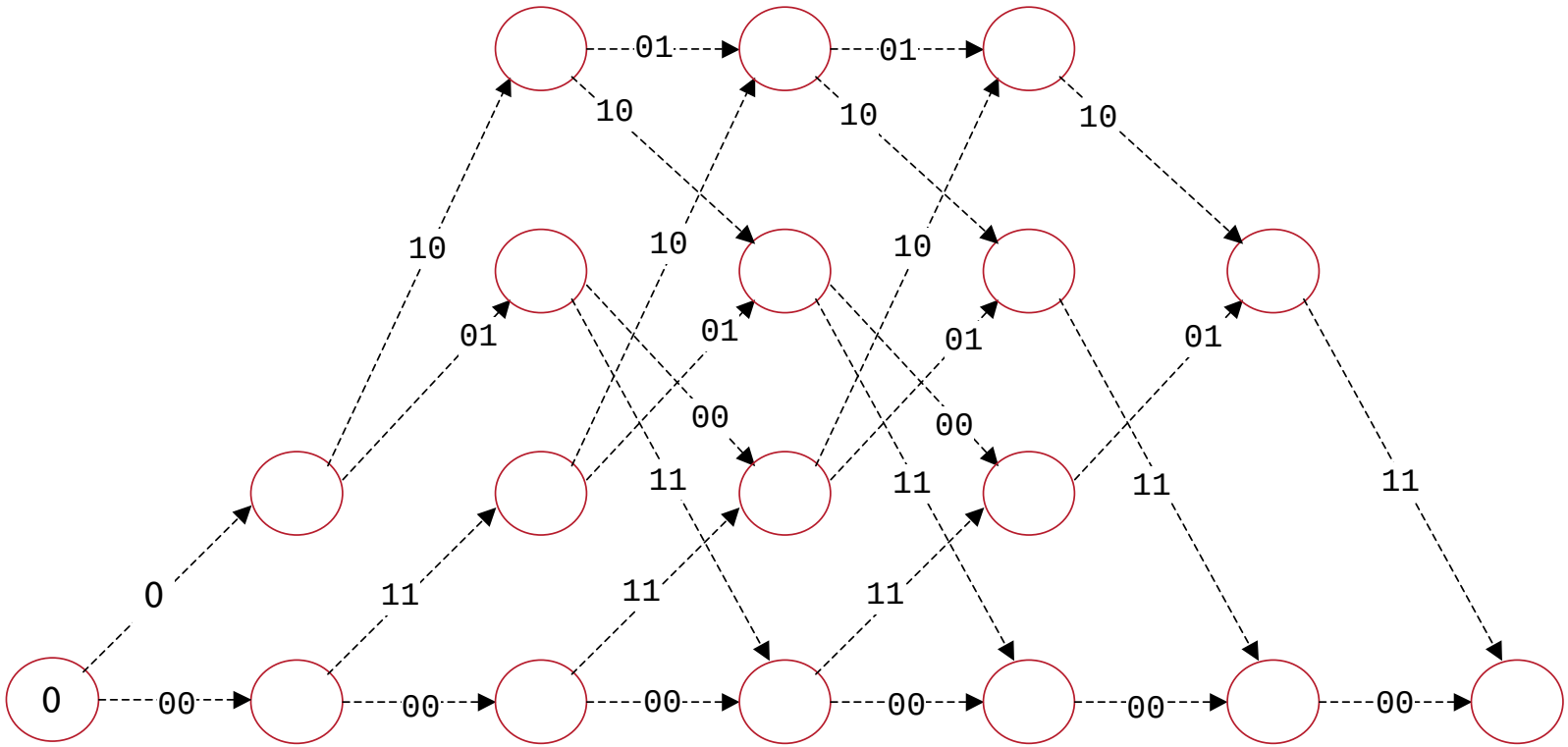
- Determine the most likely $\mathbf{u}$ given $\mathbf{r}$.
- For clarity, we will assume hard-decision decoding.



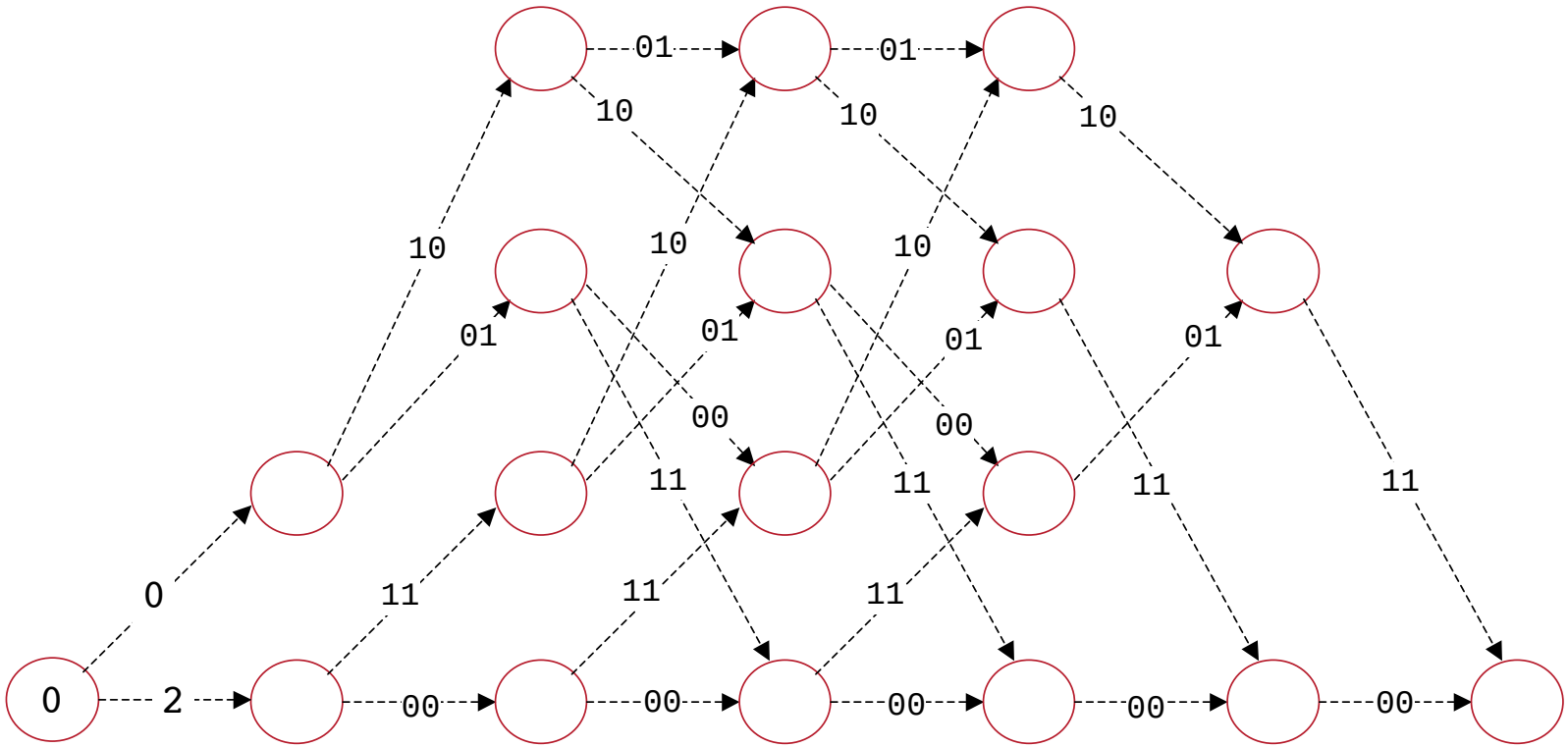
r = [1 1 1 1 1 0 0 1 0 1 1 0]



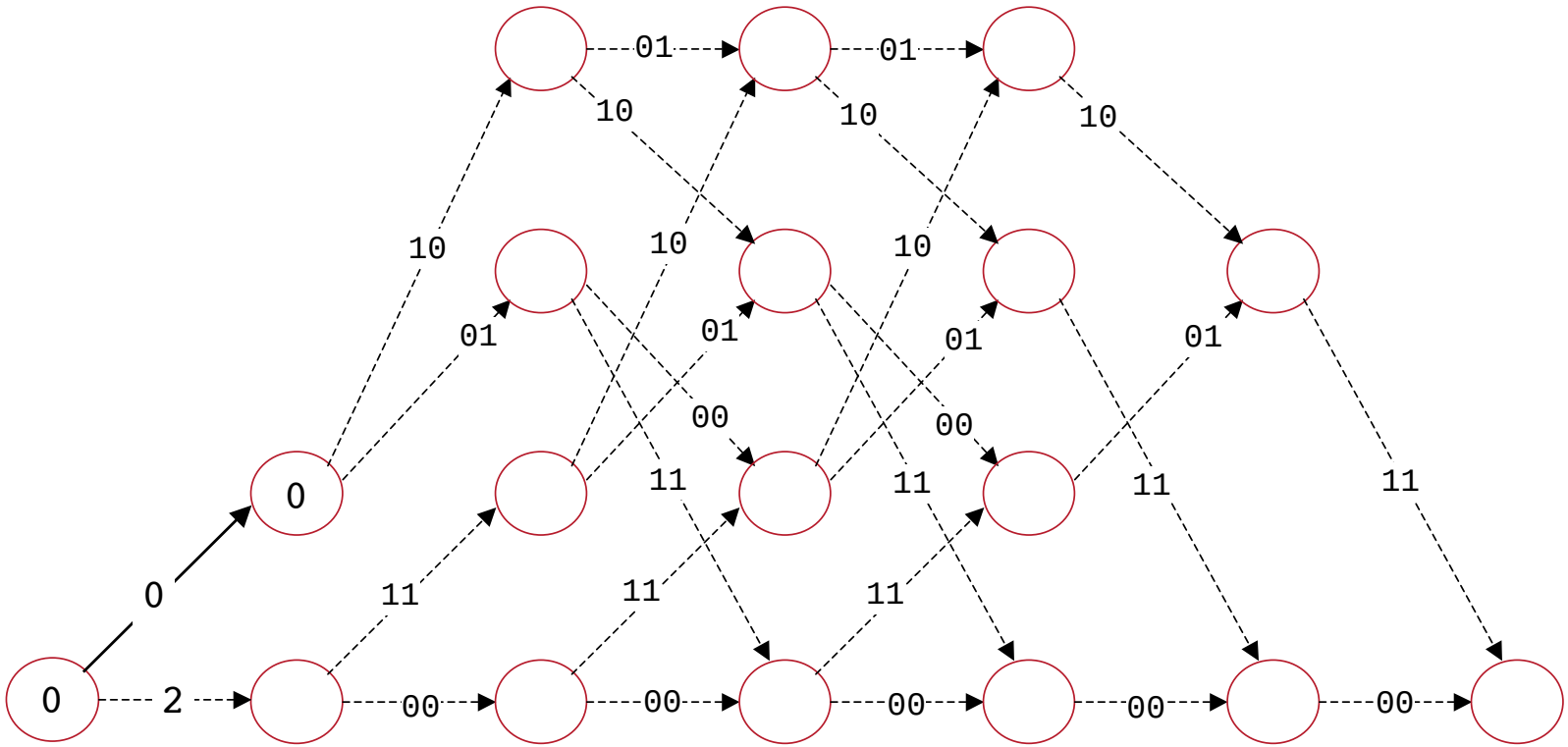
r = [1 1 1 1 1 0 0 1 0 1 1 0]



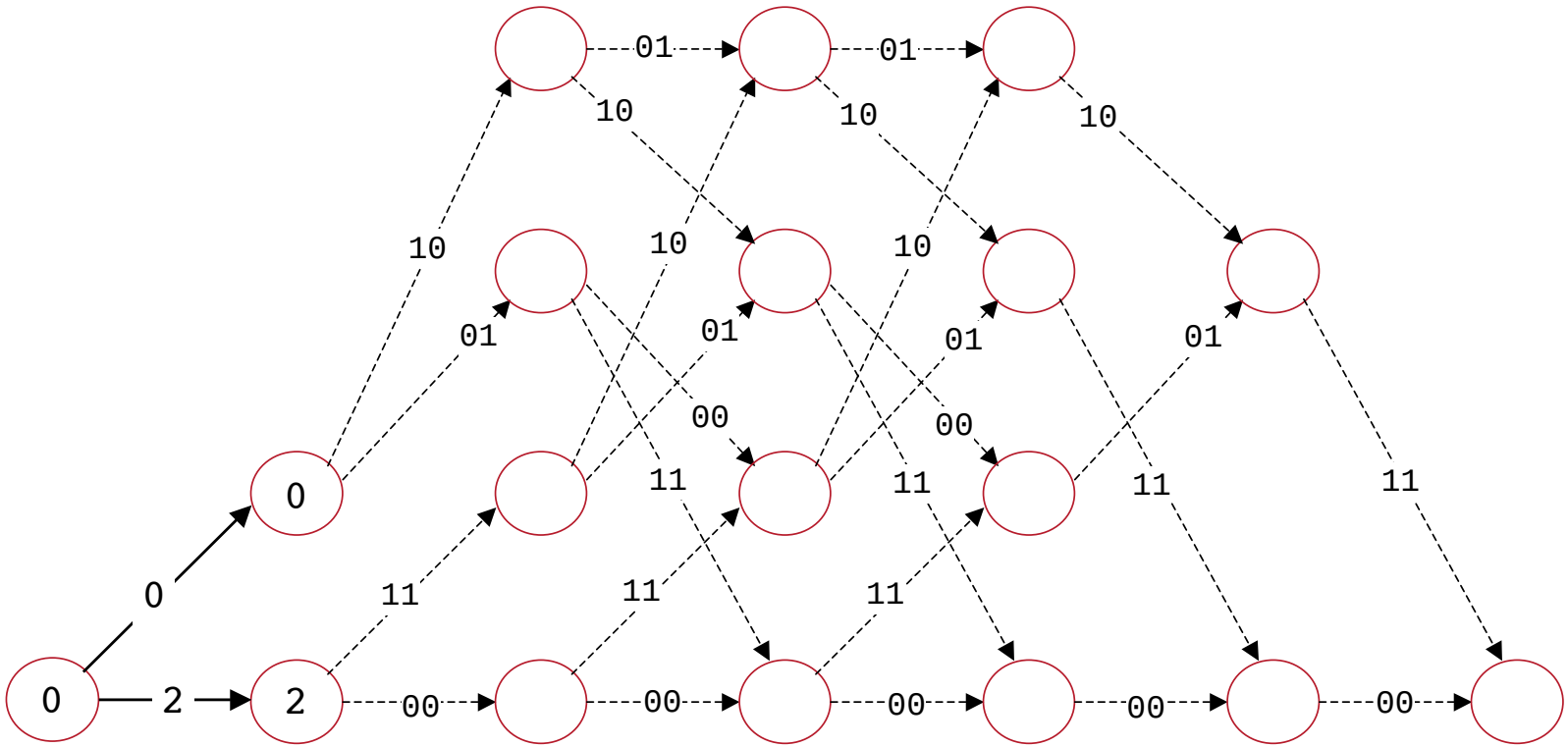
r = [1 1 1 1 1 0 0 1 0 1 1 0]

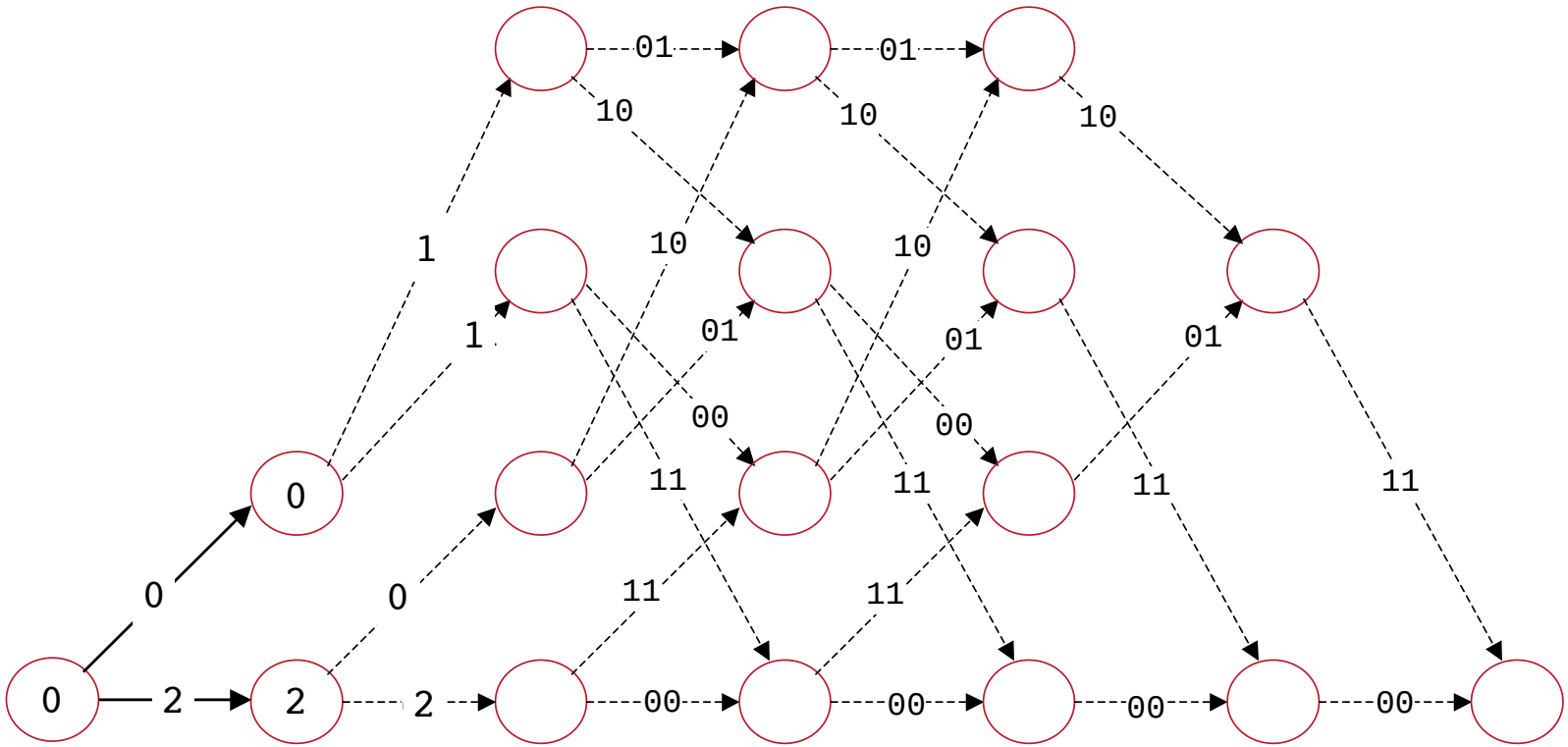


r = [1 1 1 1 1 0 0 1 0 1 1 0]

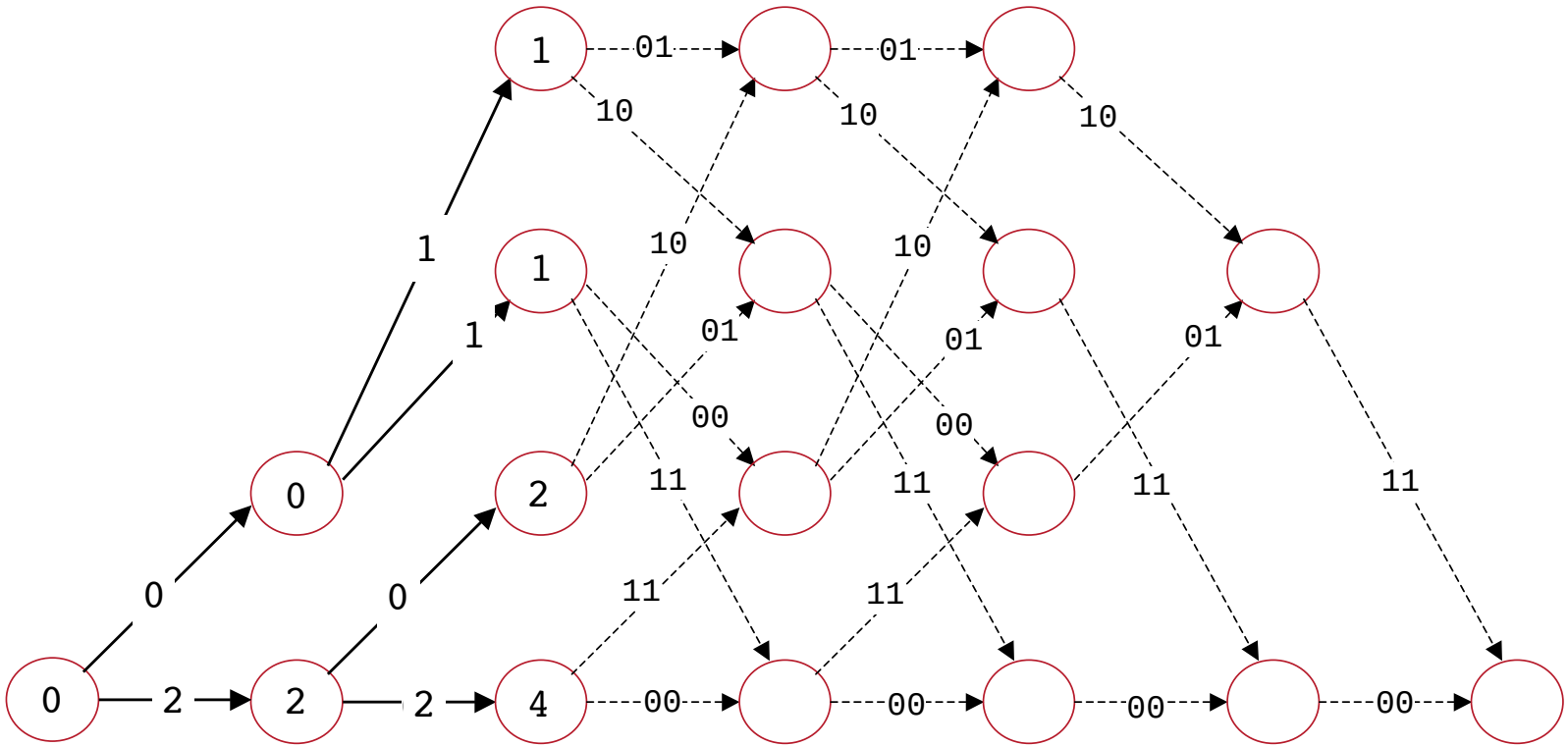


r = [1 1 1 1 1 0 0 1 0 1 1 0]

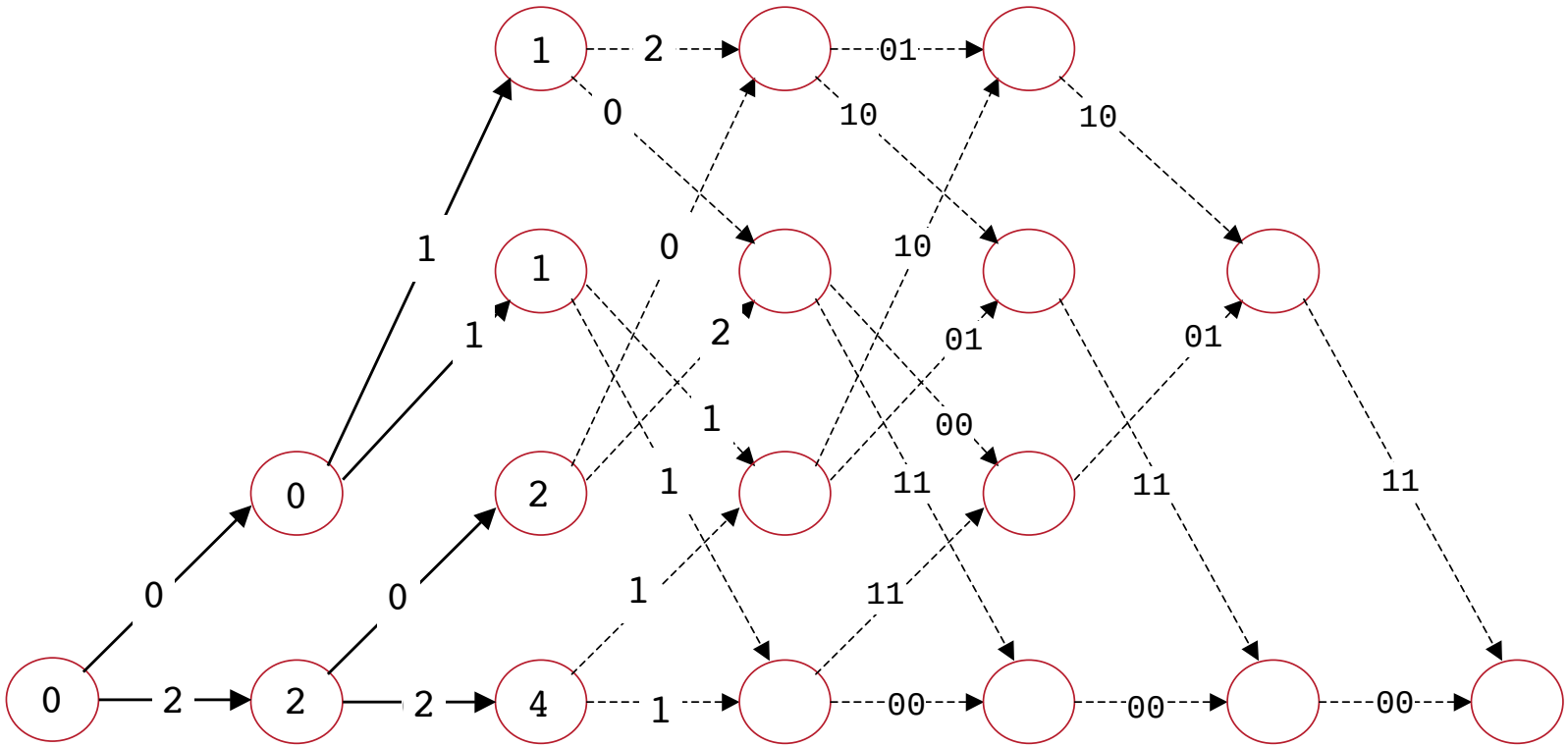

01
01
10
10
10
10
10
10
01
01
01
01
00
00
11
11
11
11
11
11
0
0
2
2
00
00
00
00
00
r = [1 1 1 1 1 0 0 1 0 1 1 0]

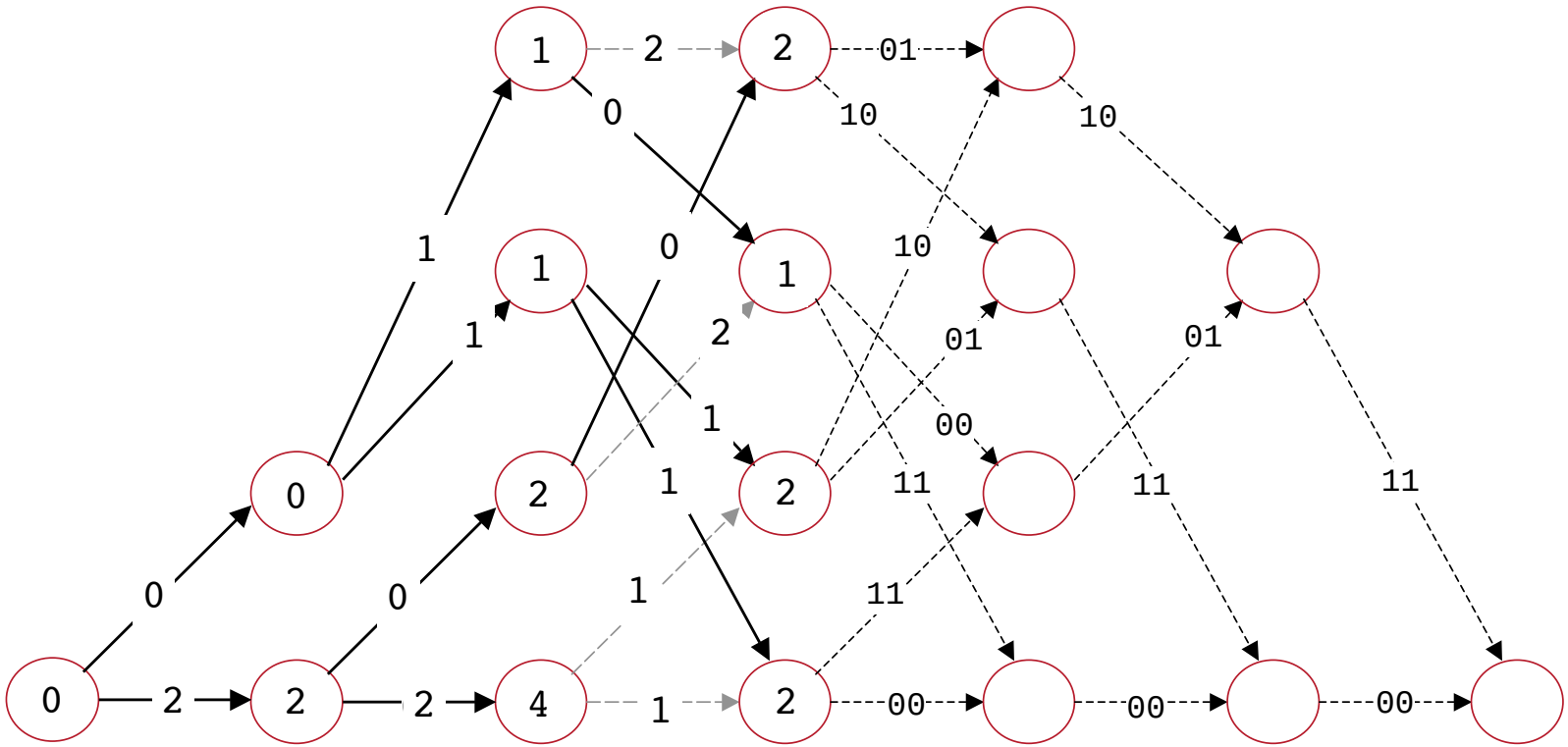


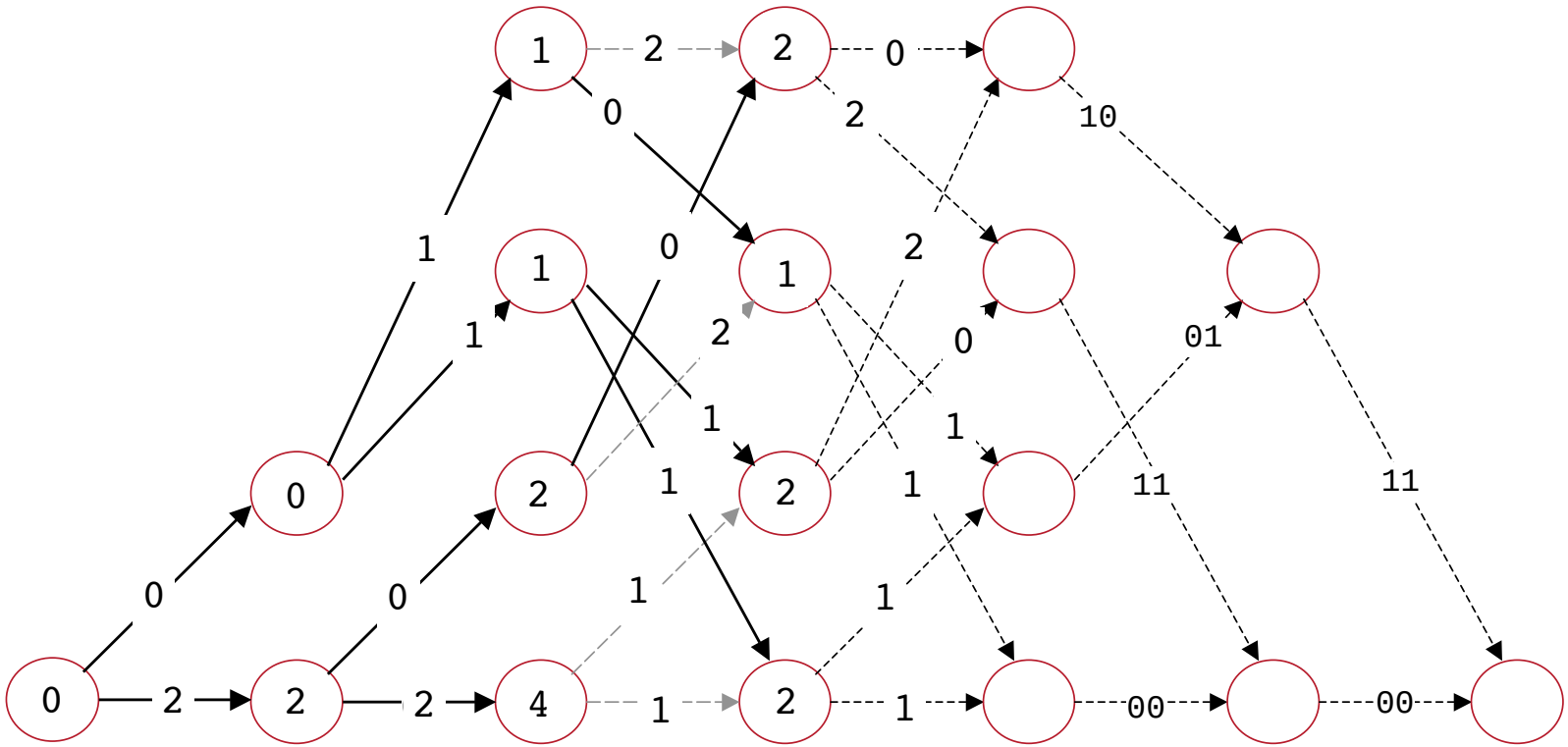
r = [1 1 1 1 1 0 0 1 0 1 1 0]

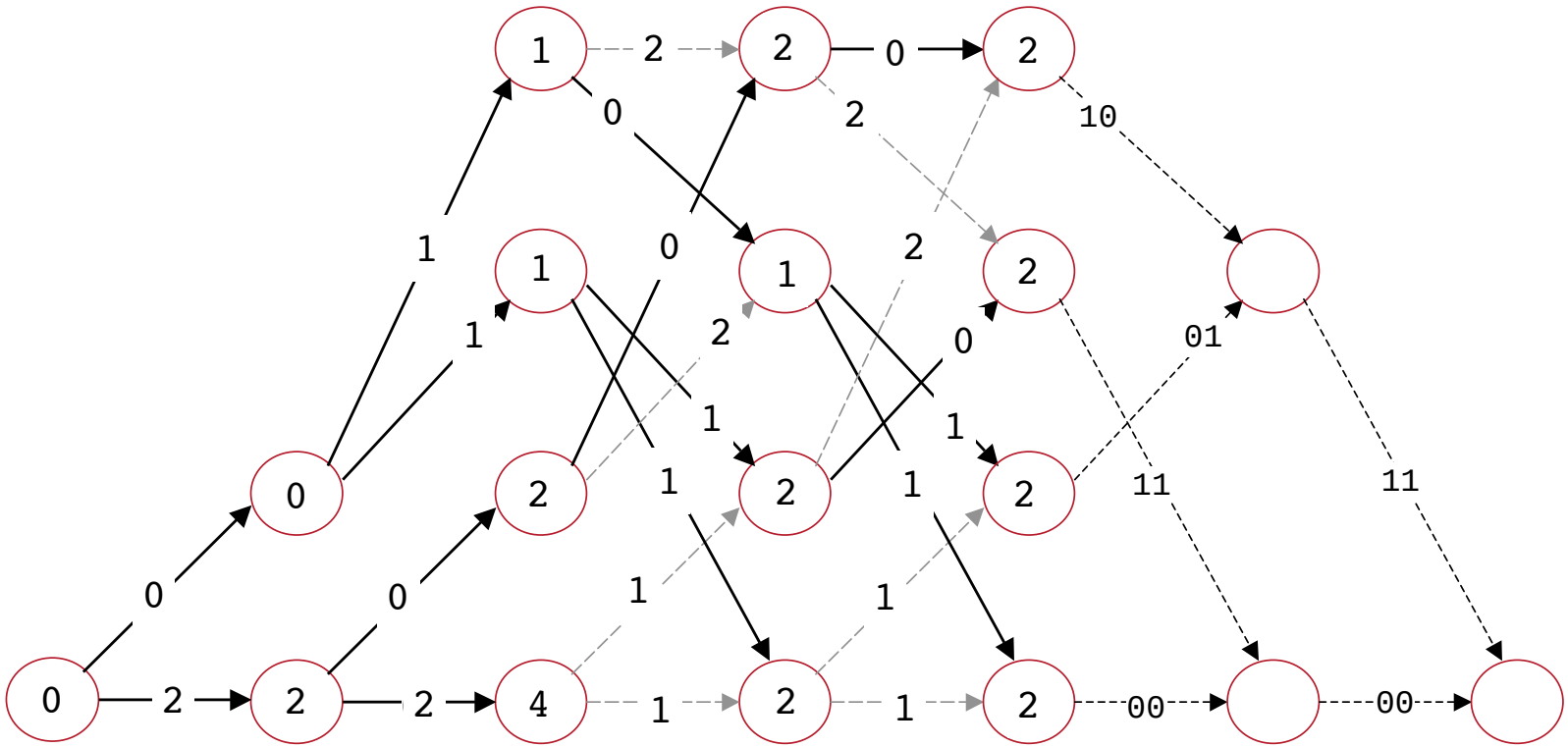


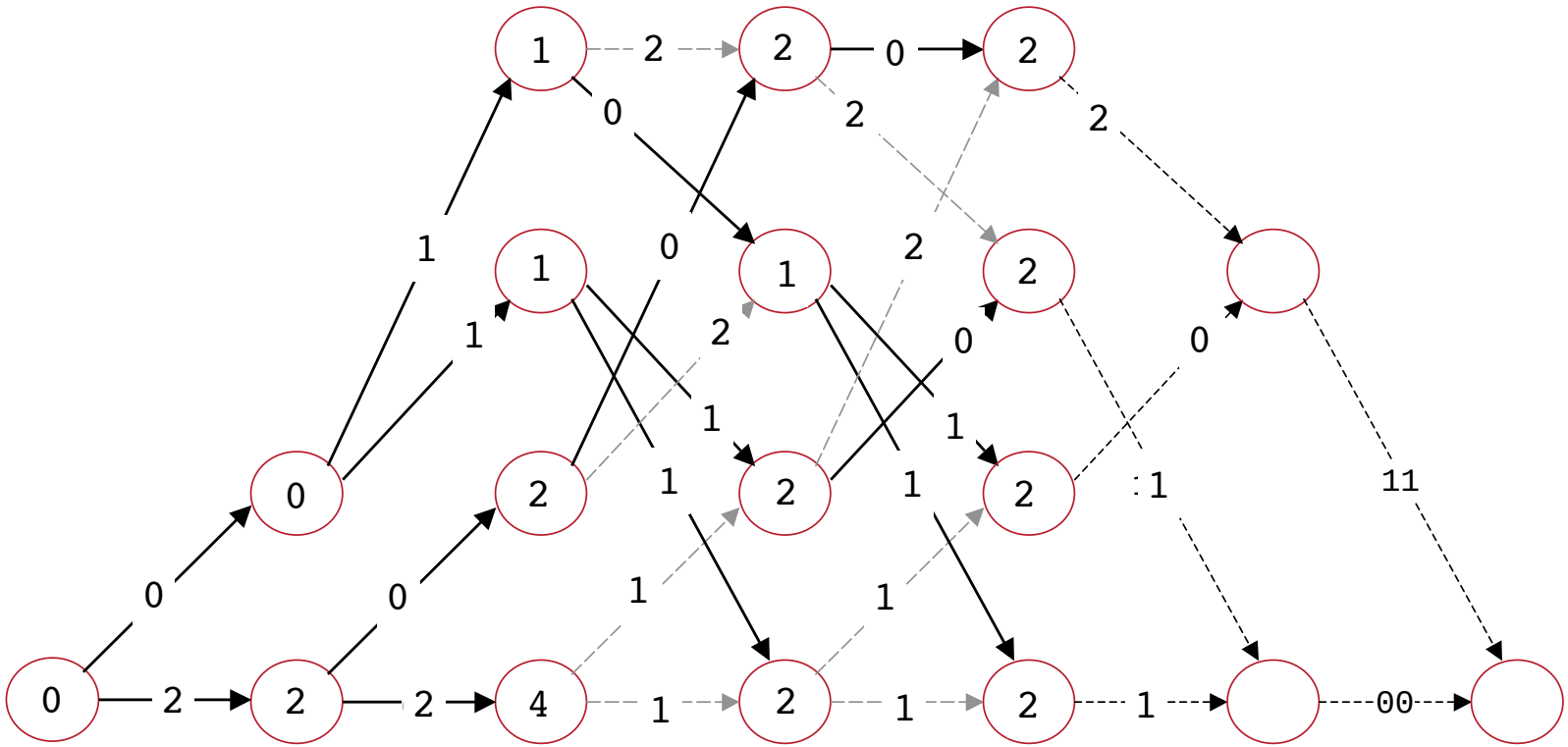
r = [1 1 1 1 1 0 0 1 0 1 1 0]

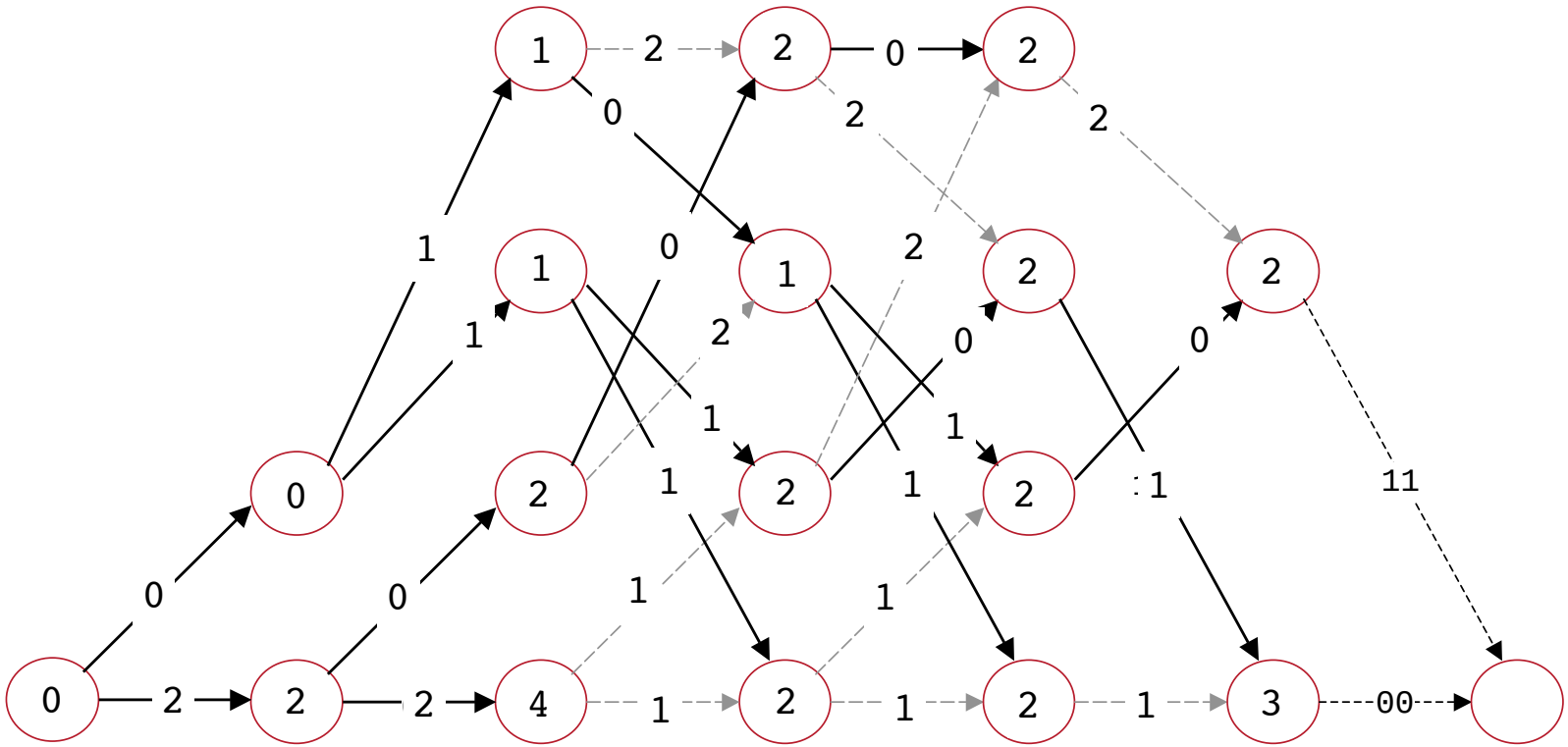

0
2
4
1
2
01
10
00
11
r = [1 1 1 1 1 0 0 1 0 1 1 0]

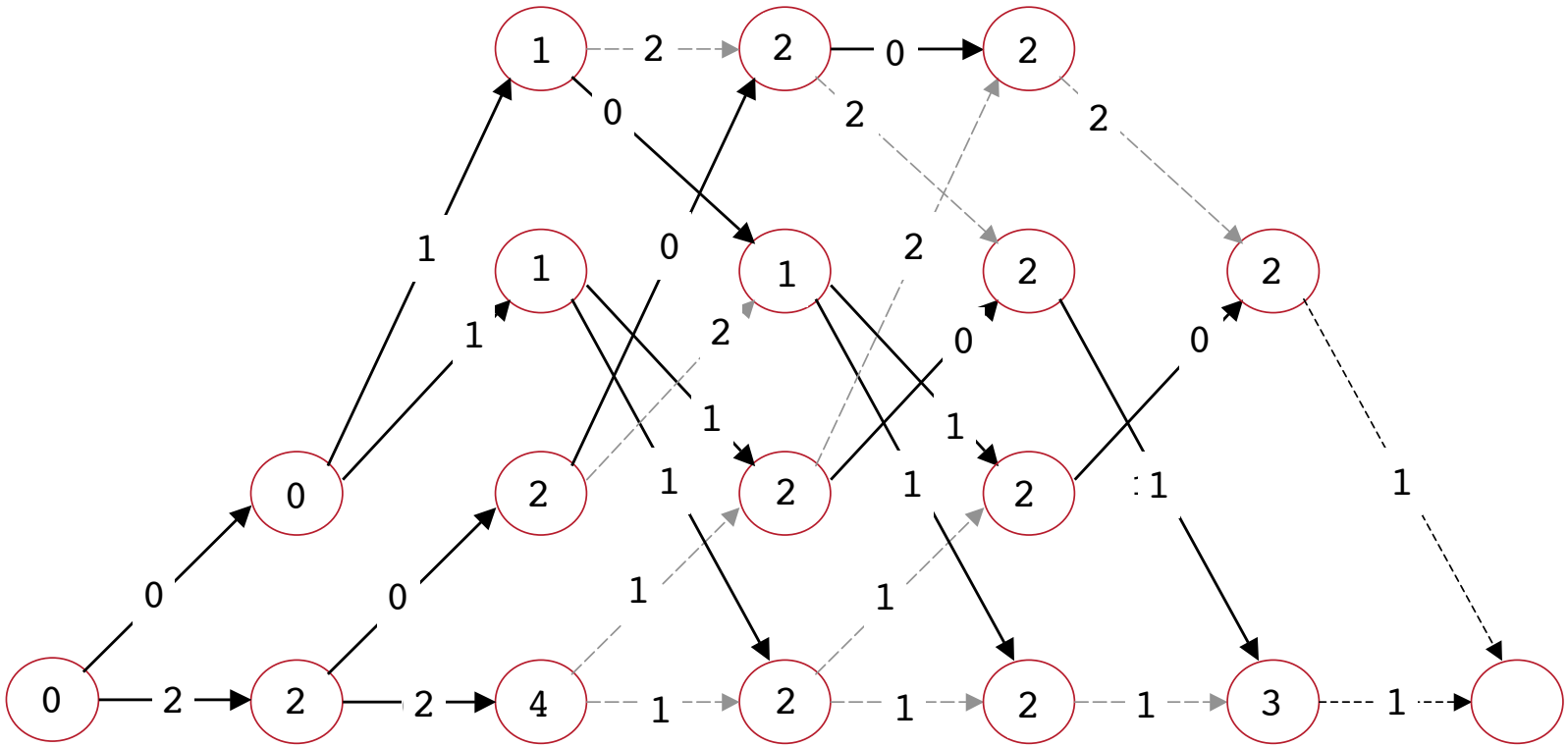

r = [1 1 1 1 1 0 0 1 0 1 1 0]

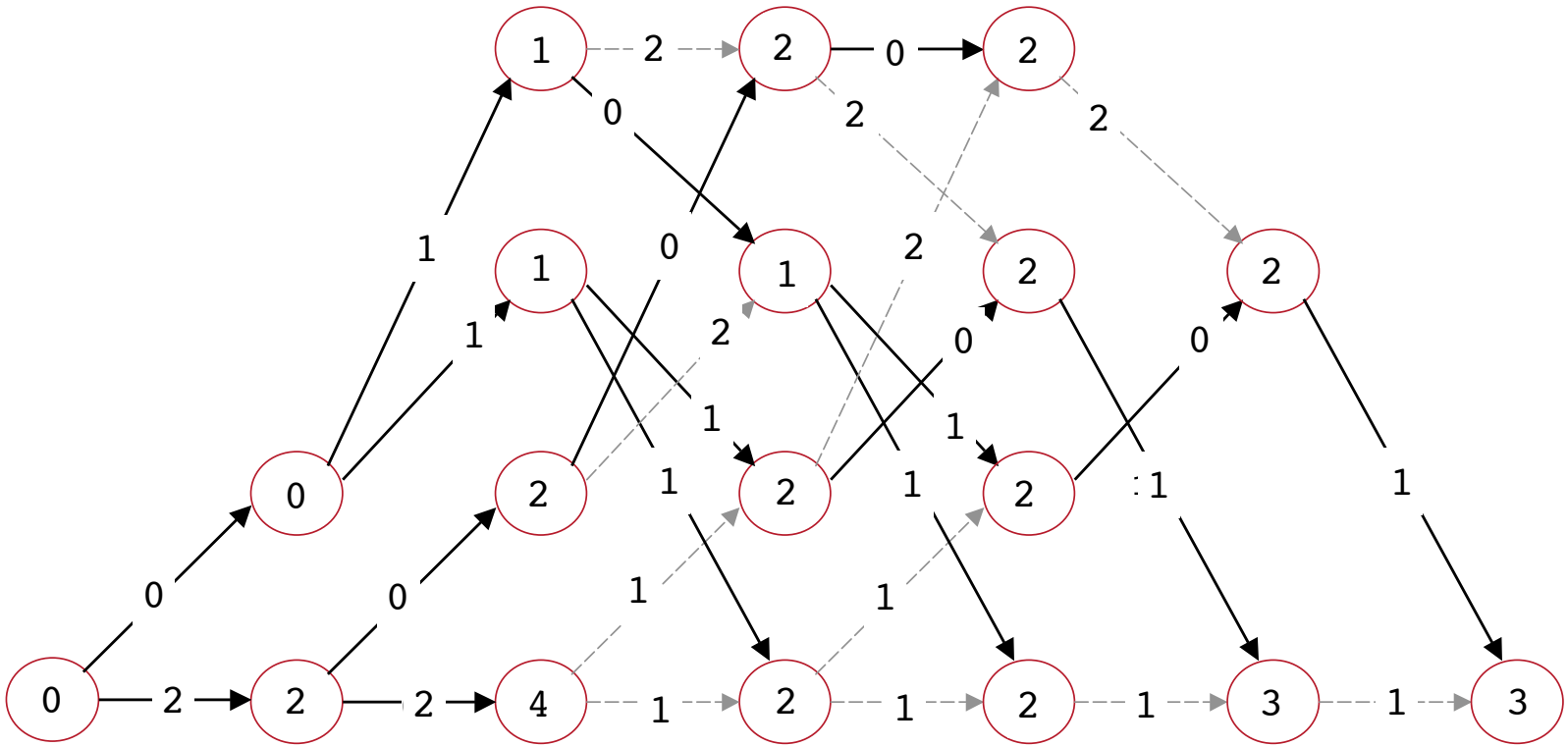

r = [1 1 1 1 1 0 0 1 0 1 1 0]

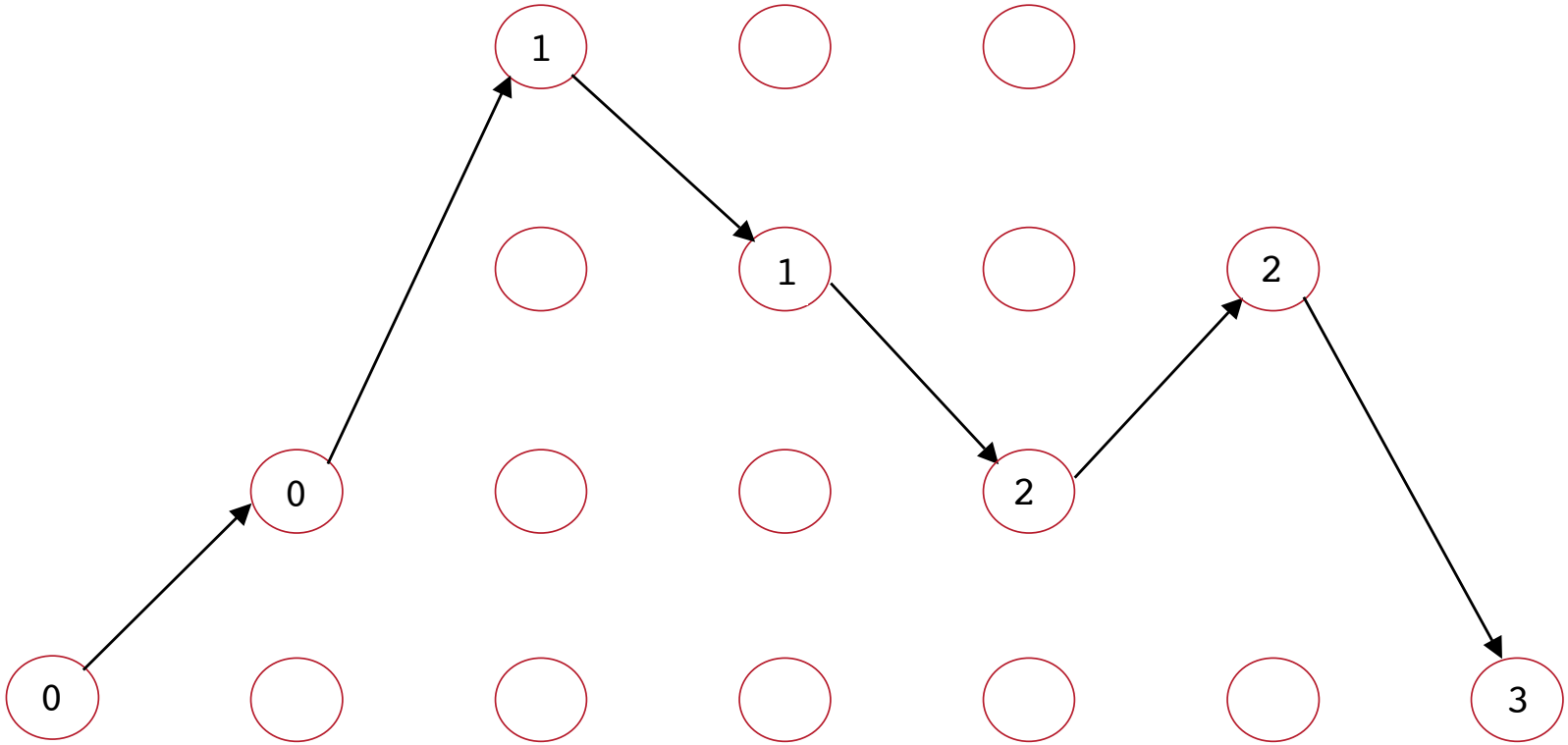

r = [1 1 1 1 1 0 0 1 0 1 1 0]


r = [1 1 1 1 1 0 0 1 0 1 1 0]

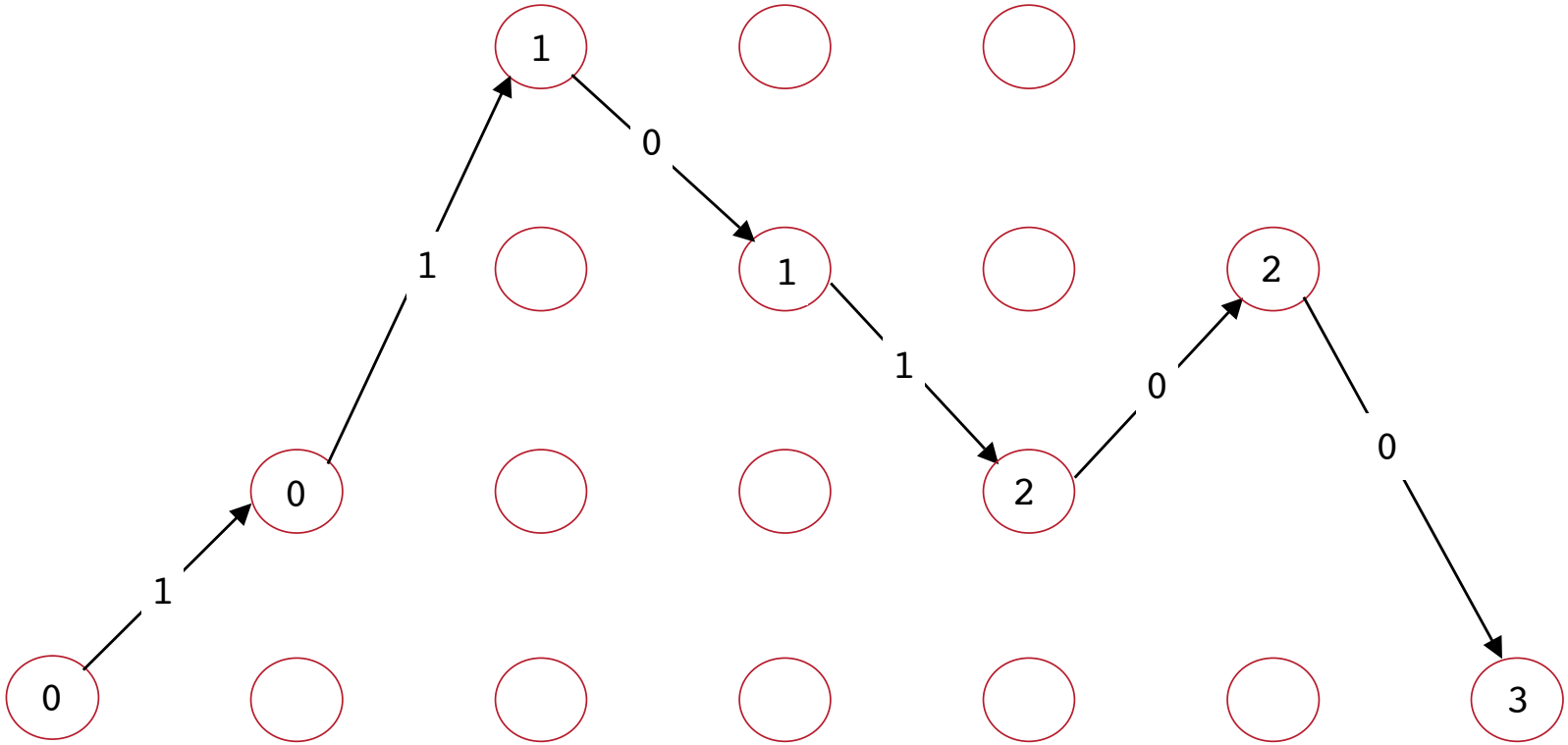

r = [1 1 1 1 1 0 0 1 0 1 1 0]


r = [1 1 1 1 1 0 0 1 0 1 1 0]


r = [1 1 1 1 1 0 0 1 0 1 1 0]



r = [1 1 1 1 1 0 0 1 0 1 1 0]


1
0
1
1
0
1
1
2
0
2
0
3
r = [1 1 1 1 1 0 0 1 0 1 1 0]

Soft-decision Decoding

- Instead of using hard-decisions on the bits, soft-decisions could be used.
- Requires that the input to the decoder be a *log-likelihood ratio* (LLR) in the form:

$$\lambda_i = \log \frac{P[c_i = 1|r_i]}{P[c_i = 0|r_i]}$$

- For BPSK modulation in AWGN, the LLR is

$$\lambda_i = \frac{2}{\sigma^2} r_i$$

- The branch metric for a particular state transition $S_j \rightarrow S_\ell$ is:

$$\gamma_{j,\ell} = \sum_{i=0}^{n_c-1} c_i \lambda_i$$

- Goal is to *maximize* the metric, rather than minimize it.
- Thus, the ACS will select the *larger* branch instead of the *smaller* one.

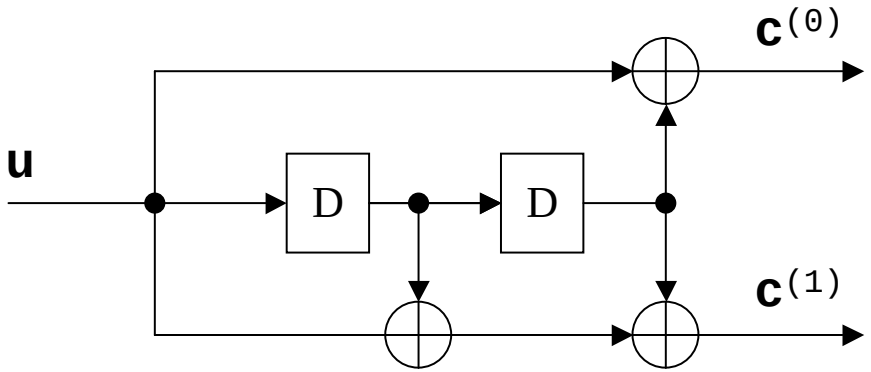
The Coded Modulation Library

- CML is a software library for simulating coding and modulation.
- Developed by me and my students.
- Runs in matlab, though much is written in C.
- Supports all four codes decribed in this tutorial.
- Open source under the lesser GPL license.
- Download at `www.iterativesolutions.com`
- Extract files, open matlab, cd to ./cml, and type `CmlStartup`

Encoding a Convolutional Code in CML

```
>> help ConvEncode
  ConvEncode encodes a NSC or RSC convolutional code with a tail.

  The calling syntax is:
      [output] = ConvEncode(input, g_encoder, [code_type] )

      output = code word

      Required inputs:
      input  = data word
      g_encoder = generator matrix for convolutional code
                  (If RSC, then feedback polynomial is first)

  Optional inputs:
      code_type = 0 for recursive systematic convolutional (RSC) code (default)
                = 1 for non-systematic convolutional (NSC) code
                = 2 for tail-biting NSC code

  Copyright (C) 2005-2008, Matthew C. Valenti
```

Encoding a Convolutional Code in CML




```
>> g = [1 0 1
        1 1 1]; % enter the generators
>> data = [1 1 0 1]; % enter the data sequence
>> c = ConvEncode( data, g, 1 )  % command to encode
c =
     1   1   1   0   1   0   0   0   0   1   1   1
```

Note that the '1' is required as a third argument to specify that this is a *nonsystematic* convolutional code and that the trellis is terminated with a tail.

Decoding a Convolutional Code in CML

```
>> help ViterbiDecode
   ViterbiDecode performs soft-in/hard-out decoding for a convolutional code using the Viterbi algorithm

   The calling syntax is:
       [output_u] = ViterbiDecode( input_c, g_encoder, [code_type], [depth] )

       output_u = hard decisions on the data bits (0 or 1)

   Required inputs:
       input_c = LLR of the code bits (based on channel observations)
       g_encoder = generator matrix for convolutional code
                   (If RSC, then feedback polynomial is first)

   Optional inputs:
       code_type = 0 for recursive systematic convolutional (RSC) code (default)
                 = 1 for non-systematic convolutional (NSC) code
                 = 2 for tail-biting NSC code
       depth = wrap depth used for tail-biting decoding
               default is 6 times the constraint length

   Copyright (C) 2005-2008, Matthew C. Valenti
```

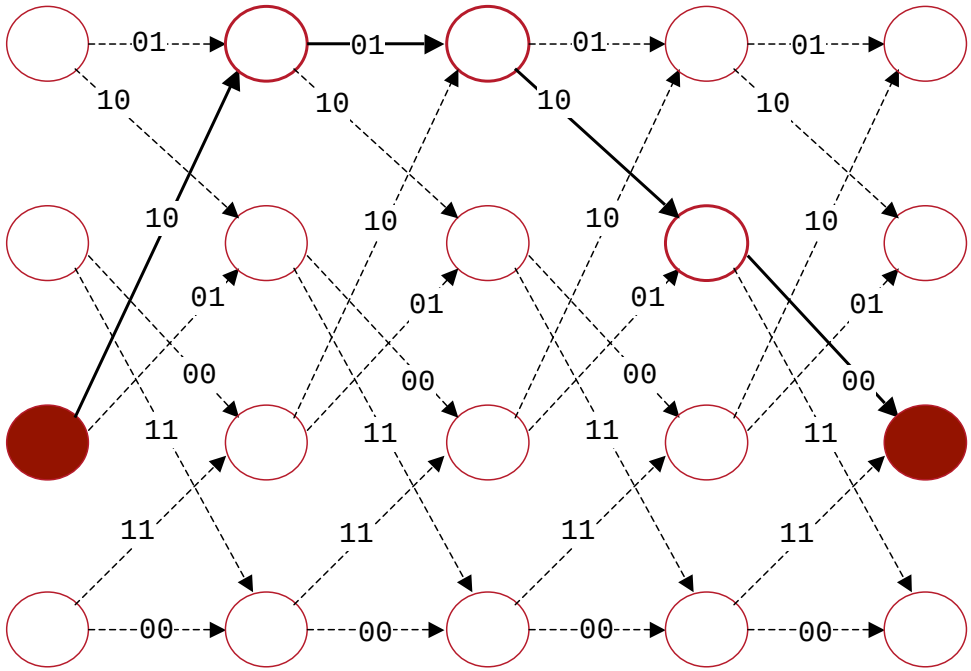
Decoding a Convolutional Code in CML

```
>> s = 2*c-1;  % BPSK modulate
>> variance = 1/2; % noise variance for Es/No = 0 dB or Eb/No = 3 dB
>> noise = sqrt(variance)*randn(size(s));  % generate noise
>> r = s + noise; % add noise to signal
>> [s' r']' % compare transmitted and received signals
ans
    1.0000   1.0000   1.0000  -1.0000   1.0000  -1.0000  -1.0000  -1.0000  -1.0000   1.0000   1.0000   1.0000
   -0.5349   0.9581   0.2854  -0.5655   1.3590   0.1967  -0.5819  -1.4551  -0.7311   0.2864   0.9862   0.9659
>> llr = 2*r/variance;  % compute the LLR
>> dataout = ViterbiDecode( llr, g, 1 ) % pass through Viterbi decoder
dataout =
     1     1     0     1
```

The '1' is again required as a third argument to specify that this is a *nonsystematic* convolutional code and that the trellis is terminated with a tail.

The Tailbiting Concept

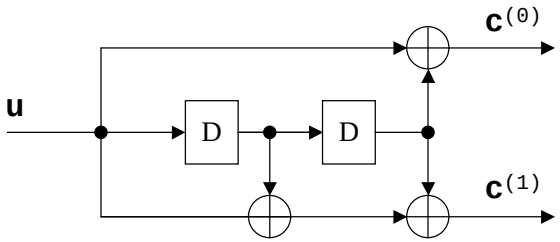
The idea behind a *tailbiting* convolutional code is to equate the initial and terminating states

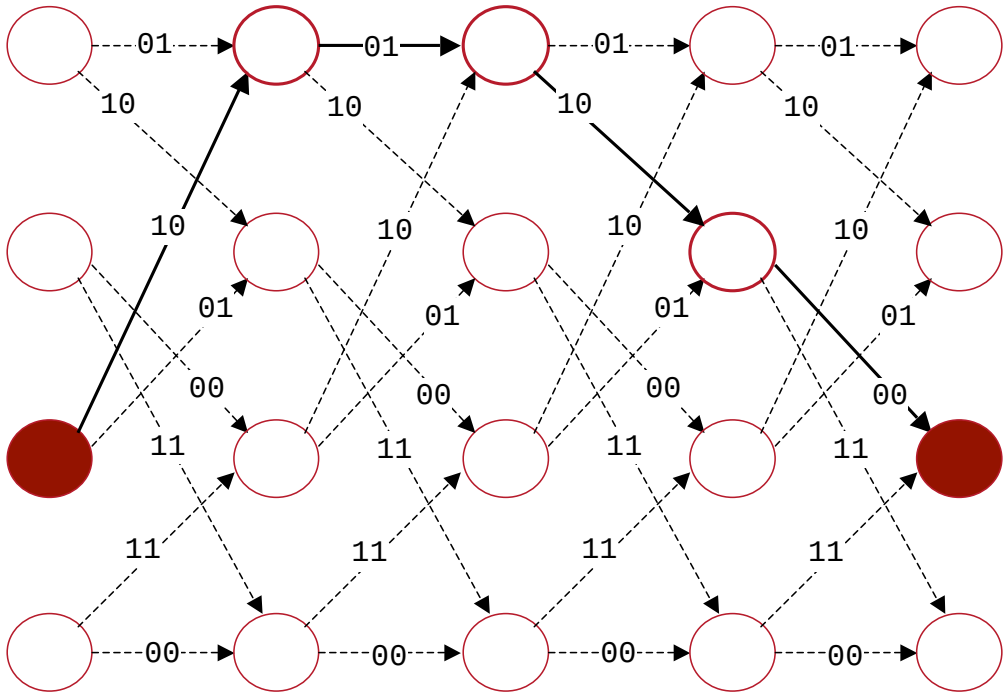


The benefit is that there is no tail or fractional rate loss.

Encoding of Tailbiting Codes

Example $\mathbf{u} = [1101]$.

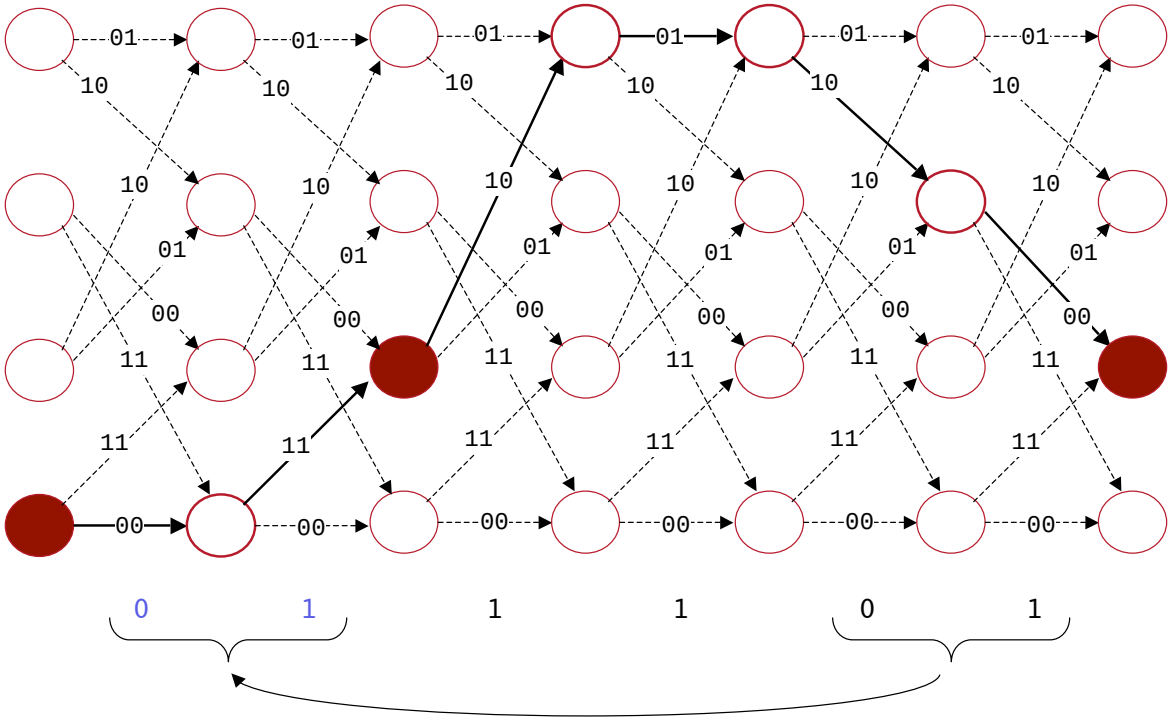




- Since the last two bits are 01, the final state is 10.
- To encode, determine final state from the last m bits, then set the initial state.

Encoding Using a Cyclic Prefix

To encode, copy the last m data bits to the beginning of the message.



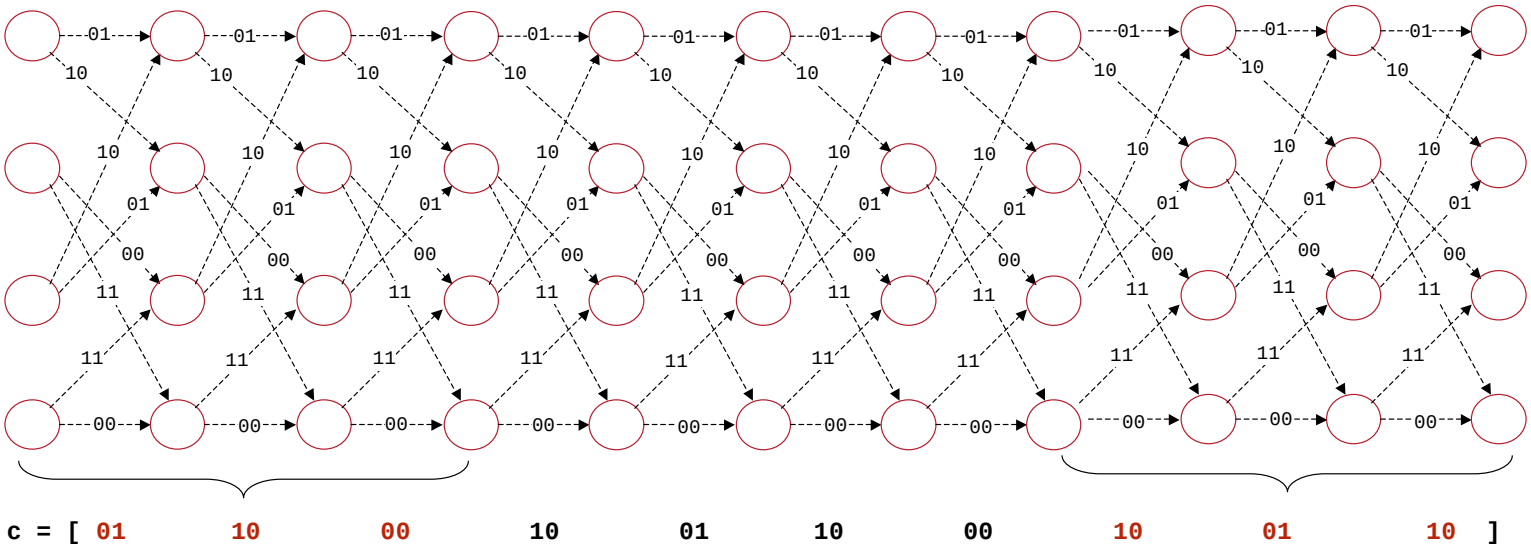
The bits were used only to determine the initial state, so don't transmit the associated code bits.

Decoding of Tailbiting Codes

- A tailbiting trellis can be visualized as a cylinder by connecting the starting and ending states.
- The Viterbi algorithm can be run on the cylinder.
- In theory, the algorithm would have to run forever by cycling around the cylinder.
- In practice, it is sufficient to limit the cycling around the cylinder.
- The *wrap depth* is the amount of trellis on the cylinder that is traversed more than once.

Wrap Depth

Here, the wrap depth is $\nu = 3$ trellis stages.



- Expand trellis by 3 sections before and after code sequence
- Append first 3 received pairs to end of sequence.
- Prepend last 3 received pairs to start of sequence.
- Assume all starting and ending states are equally likely.

Tailbiting Convolutional Code in CML

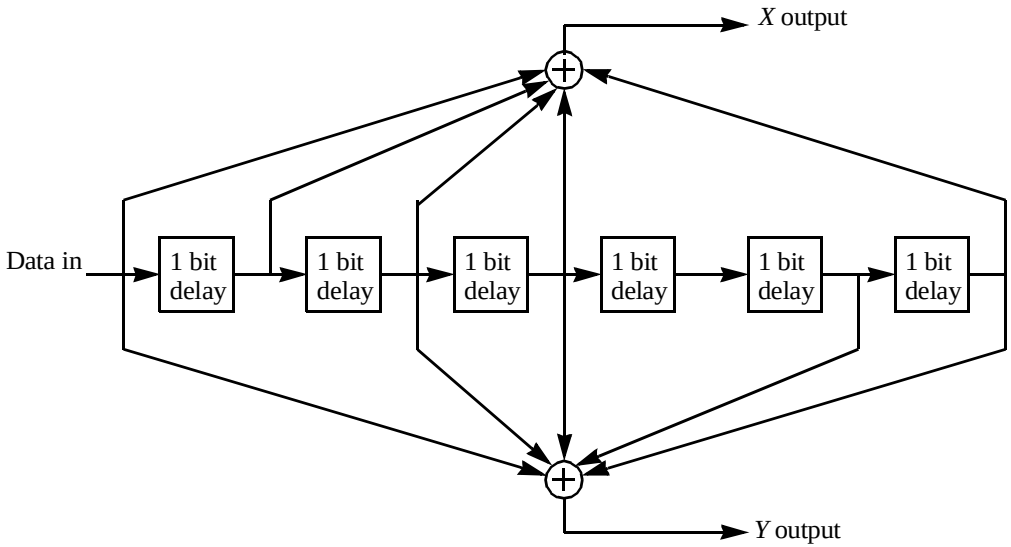
Setting the third argument of *ConvEncode* and *ViterbiDecode* to 2 indicates a tailbiting NSC code.

```
>> g = [1 0 1
        1 1 1]; % enter the generators
>> data = [1 1 0 1]; % enter the data sequence
>> c = ConvEncode( data, g, 2 )
c =
     1    0    0    1    1    0    0    0
>> s = 2*c-1;  % BPSK modulate
>> variance = 1/2; % noise variance for Es/No = 0 dB or Eb/No = 3 dB
>> noise = sqrt(variance)*randn(size(s));  % generate noise
>> r = s + noise; % add noise to signal
>> [s' r']' % compare transmitted and received signals
ans
    1.0000   -1.0000   -1.0000    1.0000    1.0000   -1.0000   -1.0000   -1.0000
    1.0000   -1.2248   -0.2257   -0.3251    1.3028   -0.3667   -0.4831   -0.5914
>> llr = 2*r/variance;  % compute the LLR
>> dataout = ViterbiDecode( llr, g, 2, 6 ) % pass through Viterbi decoder
dataout =
     1     1     0     1
```

The fourth argument to *ViterbiDecode* is the wrap depth (in multiples of ν).

The 802.16e Standard Tailbiting CC

The generators are $\mathbf{g}^{(0)} = [1111001]$ and $\mathbf{g}^{(1)} = [1011011]$



Message length is 6 to 36 bytes, depending on mode of operation.

Puncturing

The data rate can be increased above $1/2$ by *puncturing*.

- Puncturing: Periodically deleting code bits.
- At the decoder, insert an erasure in the place of the punctured bit.
- For soft-decision decoding, an erasure is $\lambda_i = 0$.

	Code Rates			
Rate	1/2	2/3	3/4	5/6
d_{free}	*10*	6	5	4
X	1	10	101	10101
Y	1	11	110	11010
XY	X_1Y_1	$X_1Y_1Y_2$	$X_1Y_1Y_2X_3$	$X_1Y_1Y_2X_3Y_4X_5$

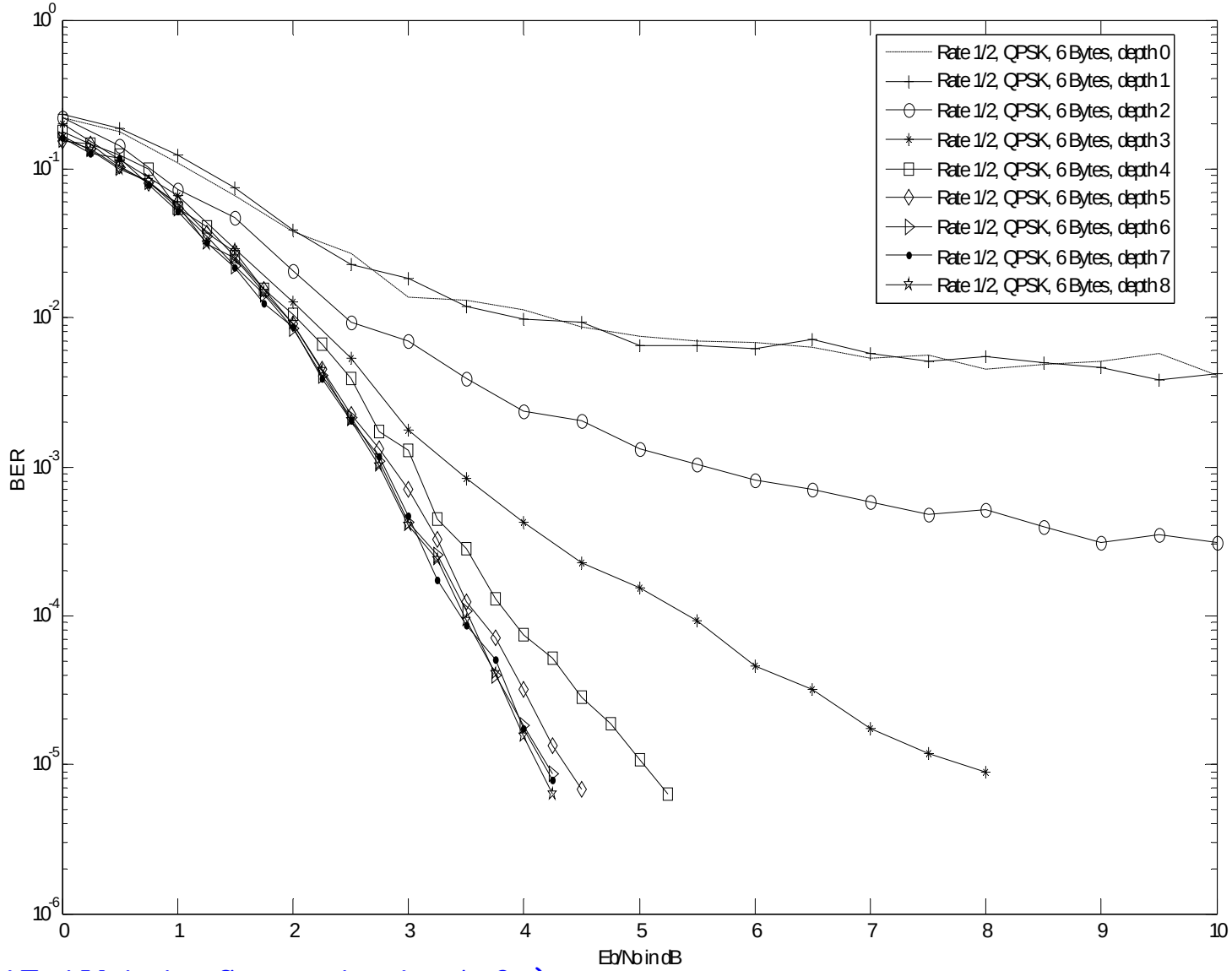
In the above table, '1' means transmit and '0' means delete.
Puncturing is implemented in CML with the *Puncture* and *Depuncture* functions.

Payload Sizes

Depending on the modulation and rate, different size payloads may be accommodated.

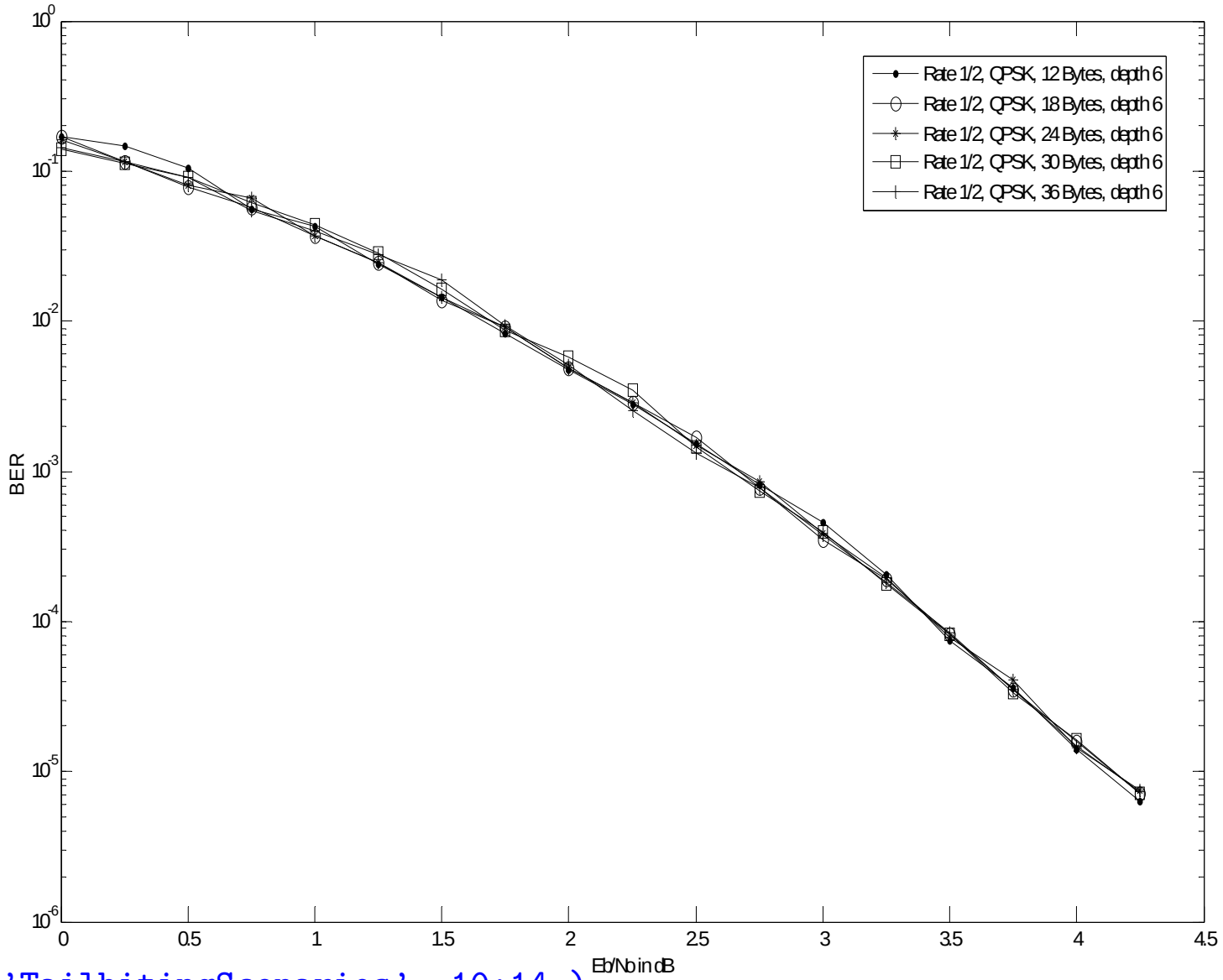
	QPSK		**16 QAM**		**64 QAM**		
Encoding rate	R=1/2	R=3/4	R=1/2	R=3/4	R=1/2	R=2/3	R=3/4
Data payload (bytes)	6						
		9					
	12		12				
	18	18		18	18		
	24		24			24	
		27					27
	30						
	36	36	36	36	36		

Influence of Wrap Depth (QPSK, r=1/2, 6 byte payload)



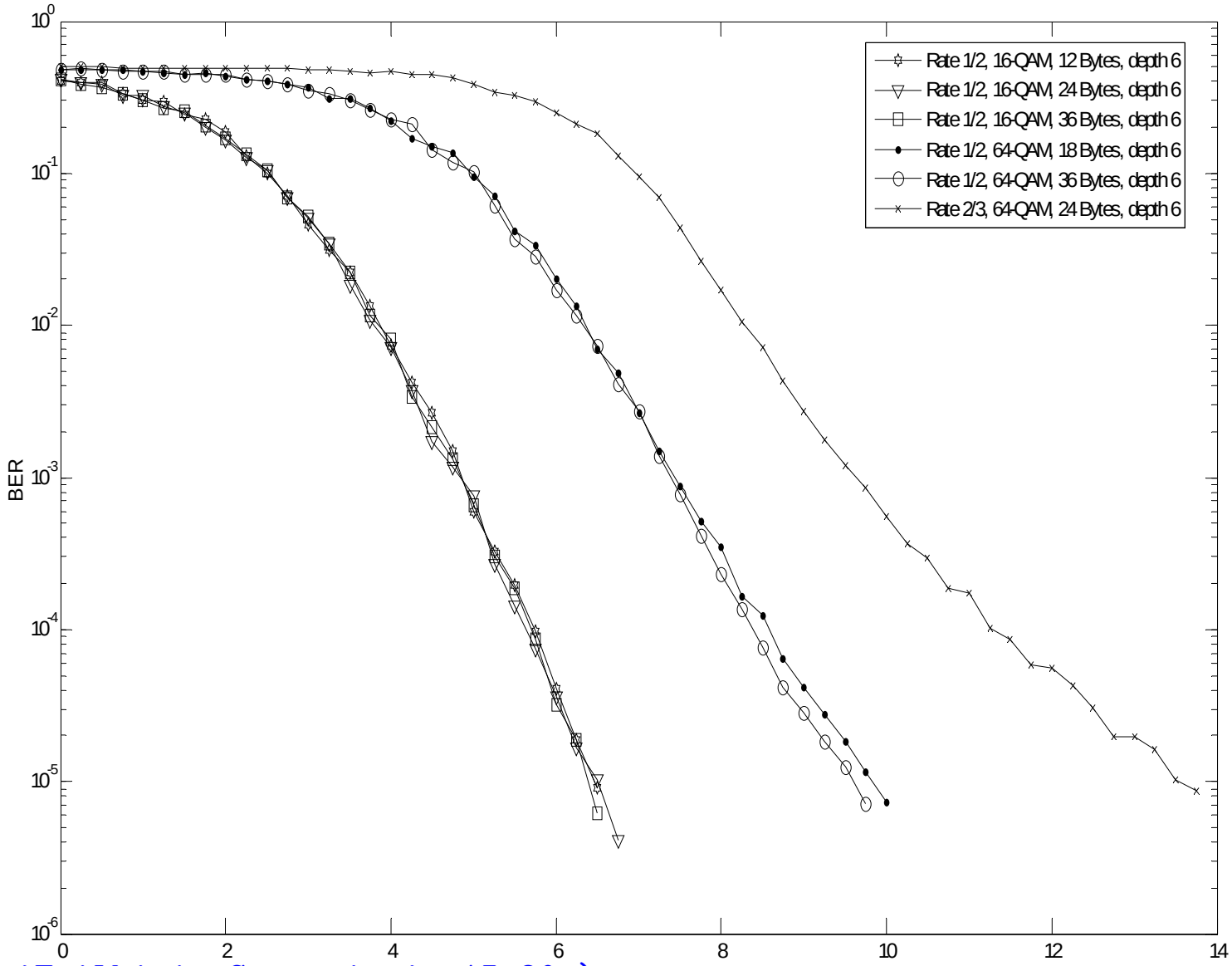

```
CmlPlot( 'TailbitingScenarios', 1:9 )
```

Performance with QPSK (wrap depth 6ν)




```
CmlPlot( 'TailbitingScenarios', 10:14 )
```

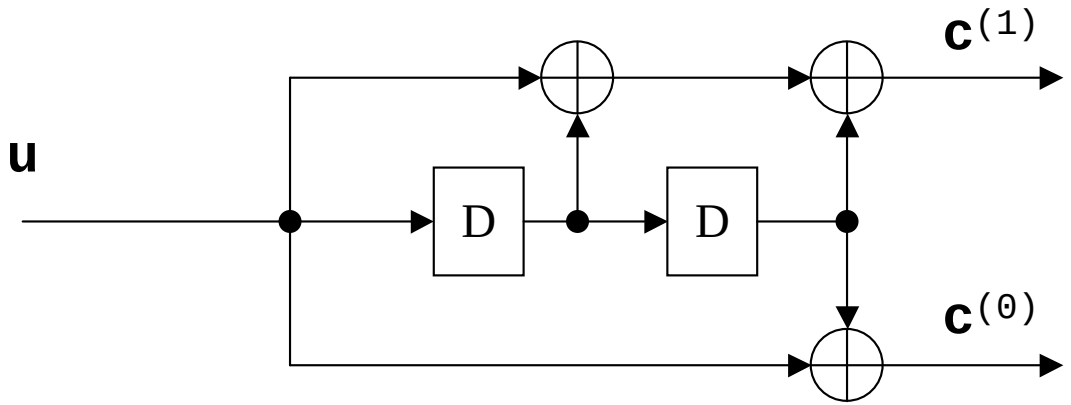
Performance with QAM (wrap depth 6ν)




```
CmlPlot( 'TailbitingScenarios', 15:20 )
```

RSC Codes

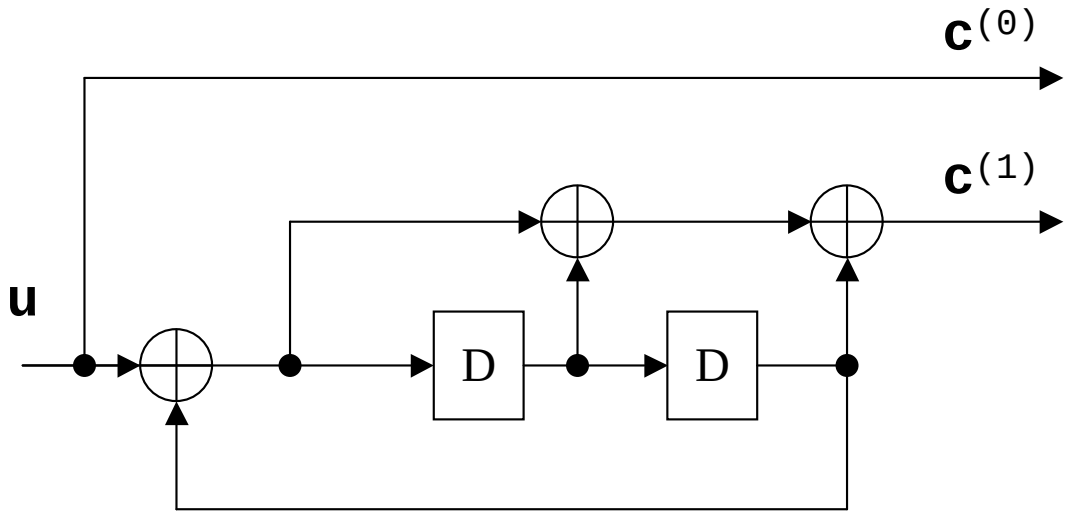
- Turbo codes use *recursive systematic convolutional* (RSC) codes.
- An RSC may be constructed from a standard convolutional encoder by feeding back one of the outputs.



The feedback of one output allows one of the outputs to be the input, hence it is *systematic*.

RSC Codes

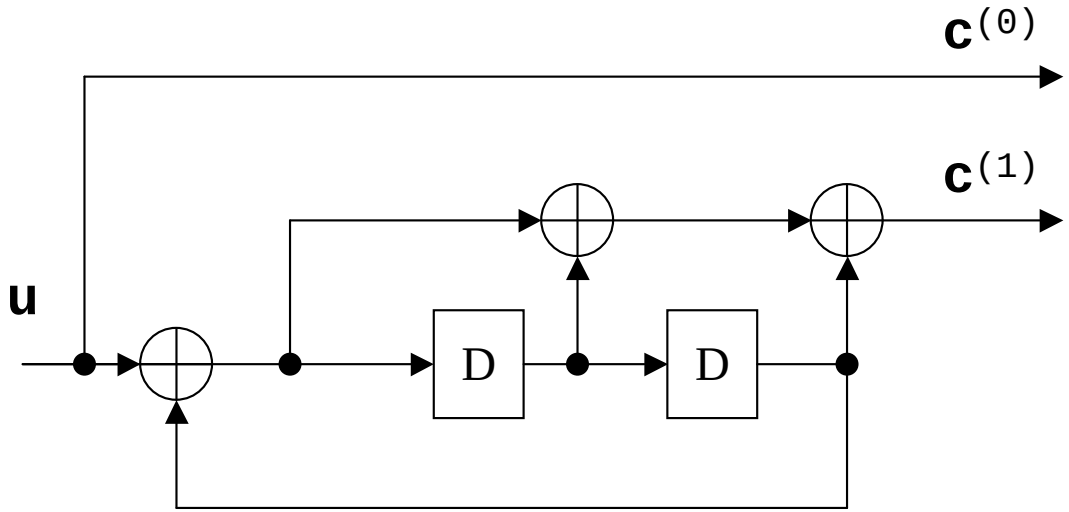
- Turbo codes use *recursive systematic convolutional* (RSC) codes.
- An RSC may be constructed from a standard convolutional encoder by feeding back one of the outputs.



The feedback of one output allows one of the outputs to be the input, hence it is *systematic*.

Trellis Termination for RSC Codes

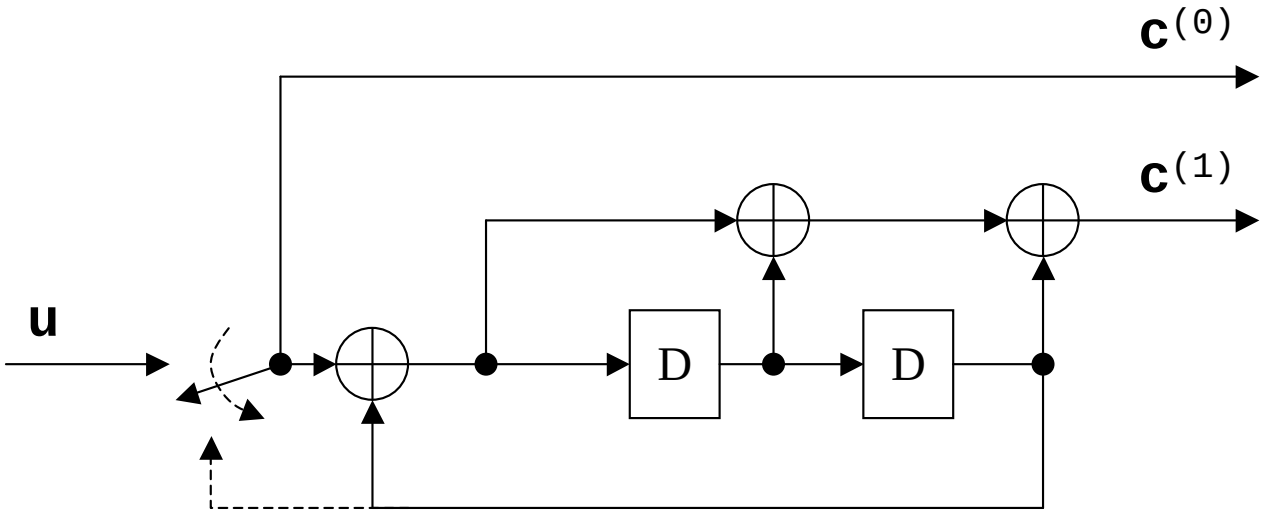
The usual convention is to start and end in the all-zeros state. However, this cannot necessarily be done with an all-zeros tail.



Throwing the switch down after the k data bits have been encoded creates a tail that will bring the encoder back into the all-zeros state.

Trellis Termination for RSC Codes

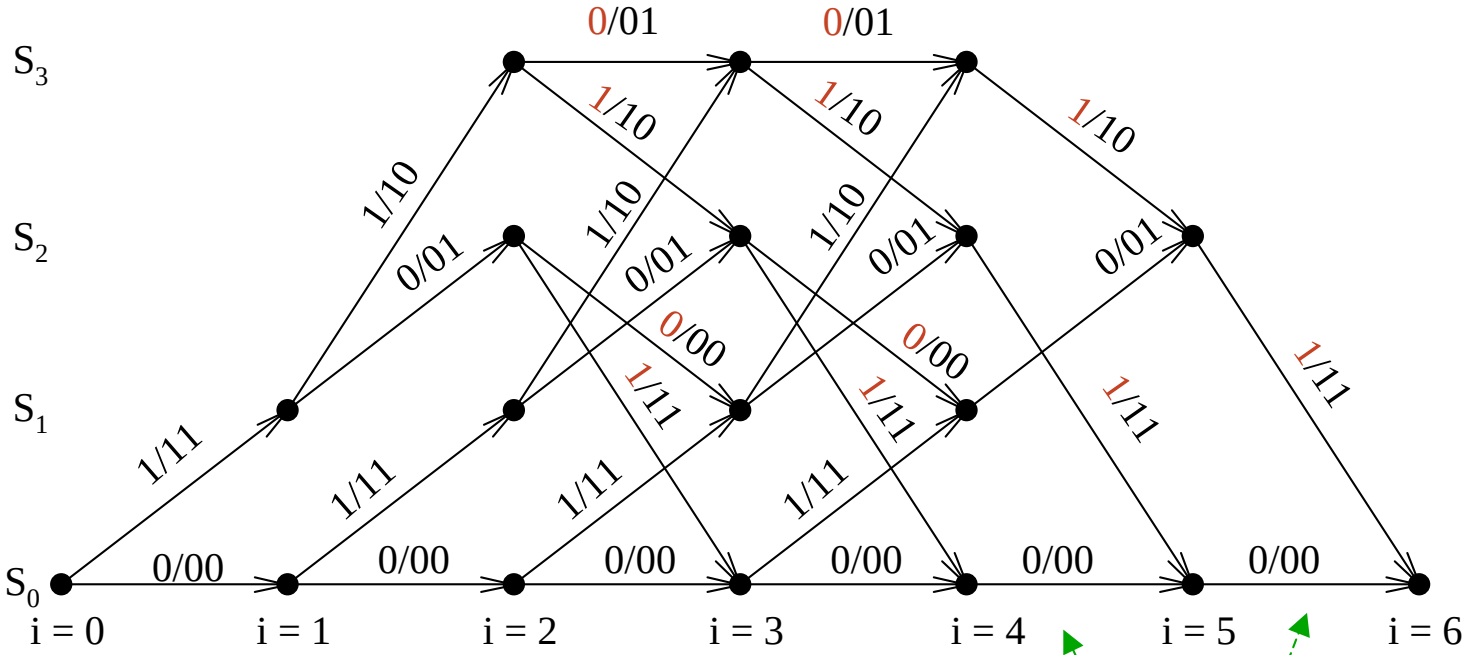
The usual convention is to start and end in the all-zeros state. However, this cannot necessarily be done with an all-zeros tail.



Throwing the switch down after the k data bits have been encoded creates a tail that will bring the encoder back into the all-zeros state.

Trellis of an RSC code

The trellis is identical to that of an NSC code, except that some input labels have been reversed.



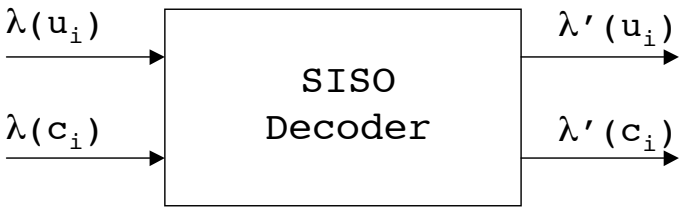
RSC Codes in CML

Setting the third argument of *ConvEncode* and *ViterbiDecode* to 0 indicates an RSC code with terminated trellis.

```
>> g = [1 0 1
        1 1 1]; % enter the generators
>> data = [1 1 0 1]; % enter the data sequence
>> c = ConvEncode( data, g, 0 )
c =
     1   1   1   0   0   1   1   0   1   1   0   0
>> s = 2*c-1;  % BPSK modulate
>> variance = 1/2; % noise variance for Es/No = 0 dB or Eb/No = 3 dB
>> noise = sqrt(variance)*randn(size(s));  % generate noise
>> r = s + noise; % add noise to signal
>> [s' r']' % compare transmitted and received signals
ans
    1.0000  1.0000  1.0000 -1.0000 -1.0000  1.0000  1.0000 -1.0000  1.0000  1.0000 -1.0000 -1.0000
    0.7331  0.7908 -0.0431 -1.1655 -0.9162  1.2226  2.0207 -1.2482  1.4407  1.5650 -0.3347 -1.7015
>> llr = 2*r/variance;  % compute the LLR
>> dataout = ViterbiDecode( llr, g, 0 ) % pass through Viterbi decoder
dataout =
     1     1     0     1
```

SISO Decoding

- The Viterbi algorithm provides a *hard* output.
- With turbo codes, multiple decoders exchange *soft* information.
- Therefore, turbo decoders require the ability to obtain *soft* outputs from the constituent decoders.



- A *soft-input, soft-output* (SISO) decoder updates the LLR's of the code bits and/or message bits by using knowledge of the underlying code structure.

SISO Decoding Overview

The SISO Decoder has three main steps:

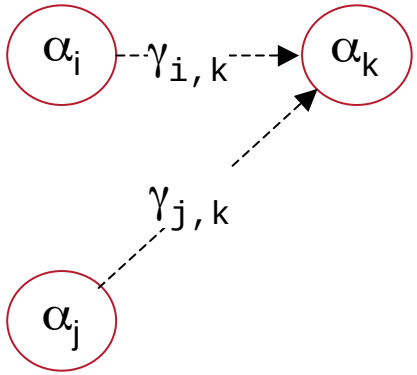
- Forward sweep through trellis.
- Backward sweep through trellis.
- Update LLR.

Because there are two sweeps, complexity is roughly twice that of Viterbi.

The algorithm has several names:

- BCJR.
- MAP.
- APP.
- log-MAP.

Forward Sweep



- Sweep through the trellis, just as in the Viterbi algorithm.
- Instead of the ACS operation at each node, use:

$$\alpha_k = \max*\left[\left(\alpha_i + \gamma_{i,k}\right), \left(\alpha_j + \gamma_{j,k}\right)\right]$$

- Where

$$\begin{aligned}\max*[x,y] &= \log(e^x + e^y) \\ &= \max[x,y] + \underbrace{\log\left(1 + e^{-|y-x|}\right)}_{f_c(|y-x|)}\end{aligned}$$

The max-log-MAP algorithm

Note that

$$\max*[x, y] \quad = \quad \max[x, y] + f_c\left(|y - x|\right)$$

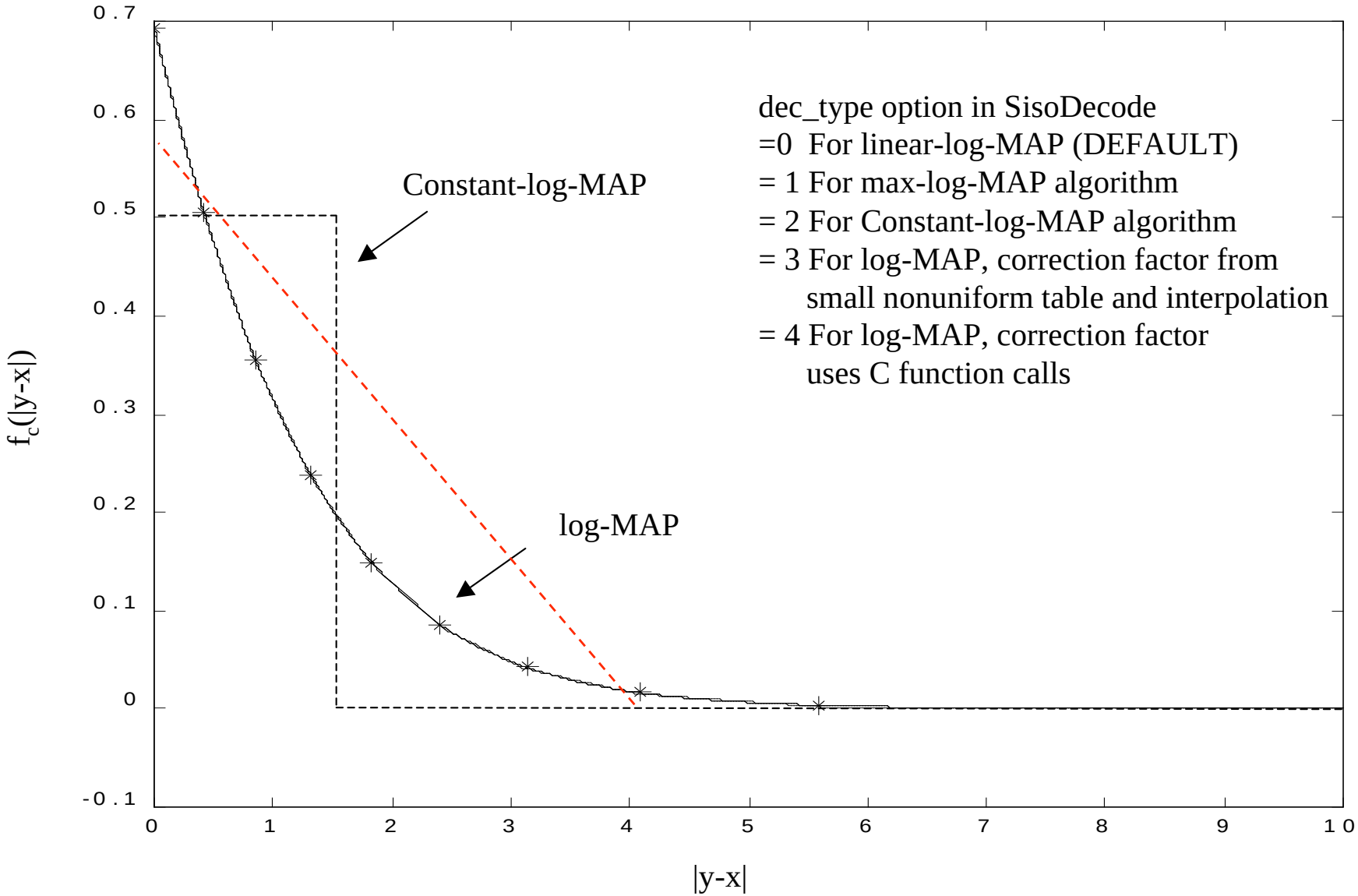
may be approximated by

$$\max*[x, y] \quad \approx \quad \max[x, y]$$

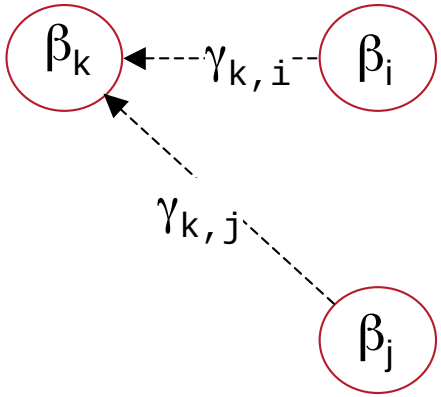
If this approximation is used, the algorithm is called the *max-log-MAP* algorithm.

The forward sweep is identical to the Viterbi algorithm.

The max-star Operator



Reverse Sweep



- Starting from the end of the trellis, sweep *backwards* to the start of the trellis.
- At each node, compute a metric

$$\beta_k \quad = \quad \max * \left[\left(\beta_i + \gamma_{k,i} \right), \left(\beta_j + \gamma_{k,j} \right) \right]$$

- If the trellis is terminated with a tail, the initial conditions are:

$$\begin{aligned} \beta_0 &= 0 \\ \beta_i &= -\infty, \quad i > 0 \end{aligned}$$

Updating the LLR's

The log-likelihood of branch $i \to j$ is

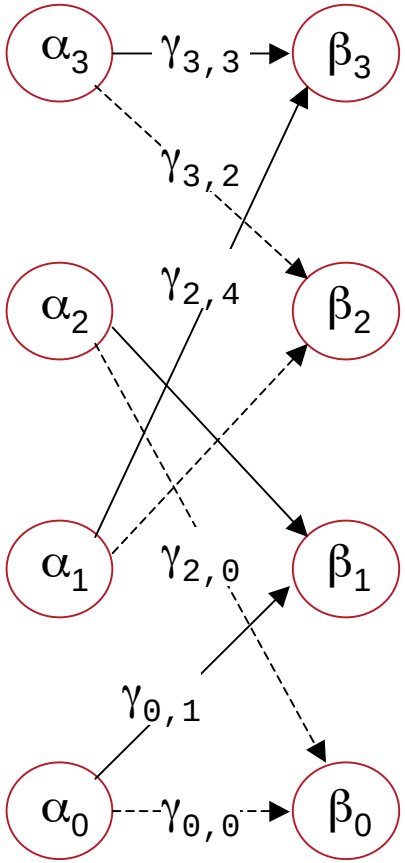
$$\Lambda(i \to j) \quad = \quad \alpha_i + \gamma_{i,j} + \beta_j$$

The log-likelihood that a message bit u is a 1

$$\max\!*_{i \to j : u=1} \{\Lambda(i \to j)\}$$

The log-likelihood that a message bit u is a 0

$$\max\!*_{i \to j : u=0} \{\Lambda(i \to j)\}$$



The output LLR is

$$\lambda'(u) \quad = \quad \max\!*_{i \to j : u=1} \{\Lambda(i \to j)\} - \max\!*_{i \to j : u=0} \{\Lambda(i \to j)\}$$

SISO Decoding in CML

Use *SisoDecode* instead of *ViterbiDecode*.

```
>> help SisoDecode
  SisoDecode performs soft-in/soft-out decoding of a convolutional code.

  The calling syntax is:
      [output_u, output_c] = SisoDecode(input_u, input_c, g_encoder, [code_type], [dec_type] )

      output_u = LLR of the data bits
      output_c = LLR of the code bits

  Required inputs:
      input_u = APP of the data bits
      input_c = APP of the code bits
      g_encoder = generator matrix for convolutional code
                (If RSC, then feedback polynomial is first)

  Optional inputs:
      code_type = 0 for RSC outer code (default)
                = 1 for NSC outer code
      dec_type = the decoder type:
               = 0 For linear approximation to log-MAP (DEFAULT)
               = 1 For max-log-MAP algorithm (i.e. max*(x,y) = max(x,y) )
               = 2 For Constant-log-MAP algorithm
               = 3 For log-MAP, correction factor from small nonuniform table and interpolation
               = 4 For log-MAP, correction factor uses C function calls (slow)  )

  Copyright (C) 2005, Matthew C. Valenti
```

SISO Decoding in CML

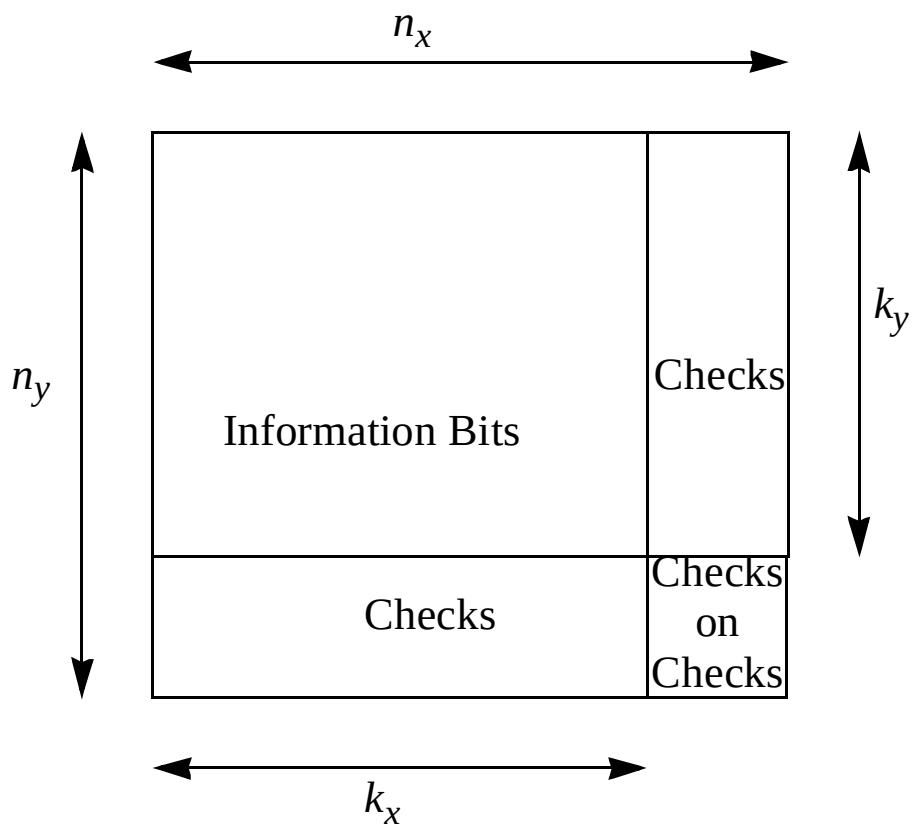
```
>> code_type = 0; % RSC code, set to 1 for NSC code.
>> g = [1 0 1
        1 1 1]; % enter the generators
>> data = [1 1 0 1]; % enter the data sequence
>> c = ConvEncode( data, g, code_type )
c =
     1   1   1   0   0   1   1   0   1   1   0   0
>> s = 2*c-1;  % BPSK modulate
>> variance = 1/2; % noise variance for Es/No = 0 dB or Eb/No = 3 dB
>> noise = sqrt(variance)*randn(size(s));  % generate noise
>> r = s + noise; % add noise to signal
>> llr = 2*r/variance;  % compute the LLR
>> input_c = llr;  % channel input
>> input_u = zeros( length(data) );  % a priori information (used in turbo decoding)
>> dec_type = 4; % this is the most accurate (but slowest) decoder type
>> [output_u, output_c] = SisoDecode( input_u, input_c, g, code_type, dec_type );
>> output_u % now the output is soft
output_u =
   18.4795   23.8975  -19.8492   22.1813
>> [output_u > 0]  % hard decision on the bits
ans =
     1     1     0     1
```

Outline

The Idea Behind Turbo Codes

- A *turbo code* is created by *concatenating* two or more *constituent* codes.
  - The constituent codes can be convolutional or block codes.
  - The encoder usually has some sort of *interleaver* to reorder the data at the input of the different encoders.
  - At least one code should be recursive (e.g. RSC).
- Decoding is iterative
  - There is a constituent decoder for each constituent code.
  - The constituent decoders are SISO.
  - The decoders exchange information after each iteration.
  - Iterations proceed until either data is correct or a maximum number of iterations is reached.

Product Codes



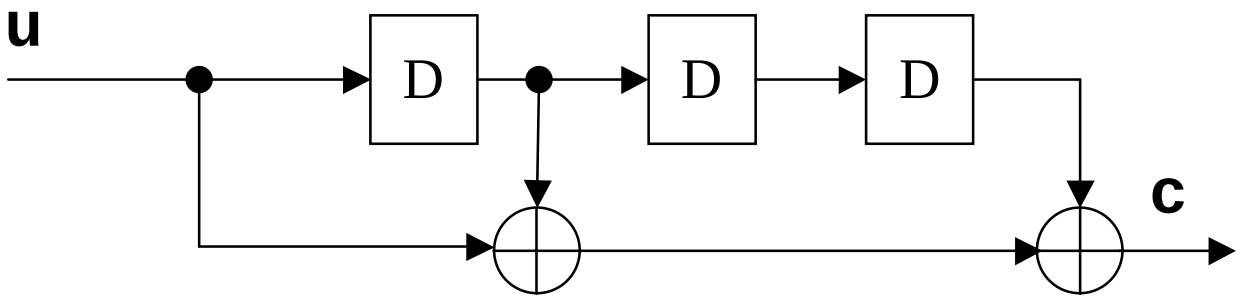
- Place data into k_y by k_x matrix.
- Encode each row by a (n_x, k_x) systematic code.
- Encode each column by a (n_y, k_y) systematic code.
- Transmit the $n_x n_y$ code bits.
- The BTC specified by IEEEE 802.16e is a product code.

Cyclic Codes

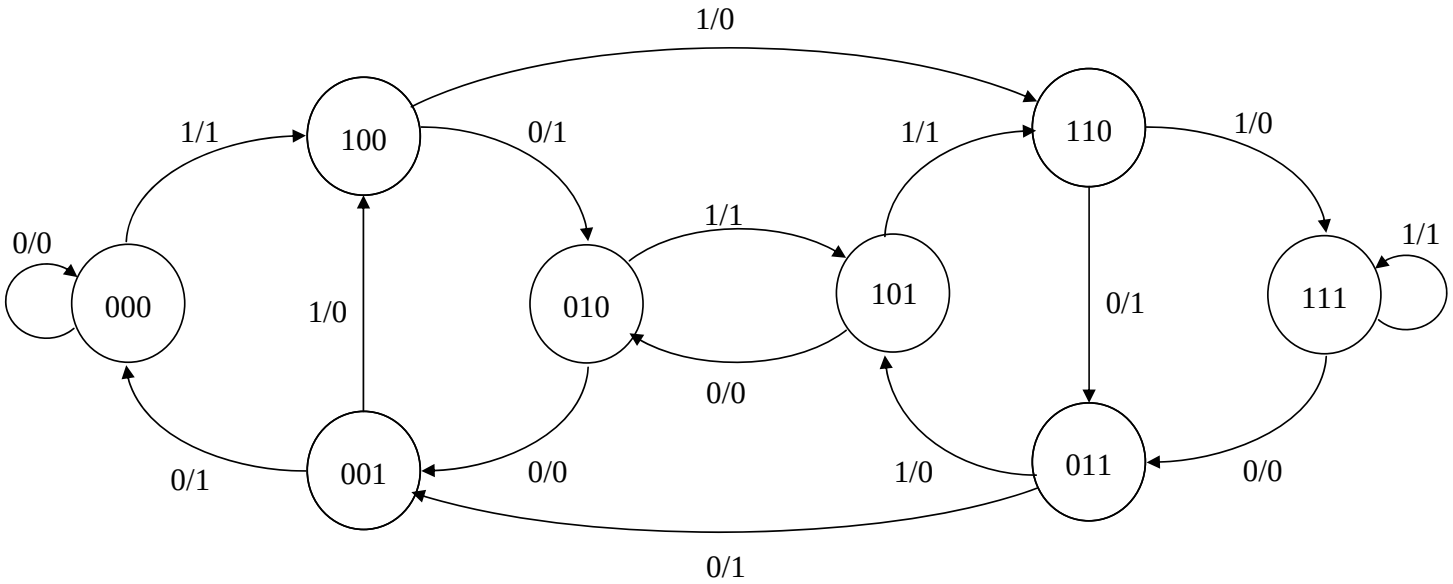
- Block Turbo Codes use systematic *cyclic codes*.
- A code is *cyclic* if a cyclic shift of any codeword produces another valid codeword.
- Like convolutional codes, cyclic codes are produced by performing the discrete-time convolution of the message with a generator.
- Cyclic codes are specified by a degree m generator polynomial.
- Cyclic codes used in 802.16
  - Hamming, $n = 2^m - 1$.
  - Single parity check (SPC), $g(X) = 1 + X$ and $n =$ anything.
  - Extended Hamming: $g(X) = g_{Hamming}(X)g_{SPC}(X)$ and $n = 2^m$.
- $k = n - m$.
- Example Hamming code.
  - $g(X) = 1 + X + X^3$.
  - $\mathbf{g} = [1101]$.
  - $m = 3$, $n = 2^m - 1 = 7$, $k = n - m = 4$.

Encoding Cyclic Codes

- Convolving the length k message with the length $\nu = m + 1$ generator results in a length $n = k + \nu - 1 = k + m$ codeword.
- The codeword may be generated using a rate-1 convolutional encoder.
- The trellis must be terminated with a tail of m zeros.

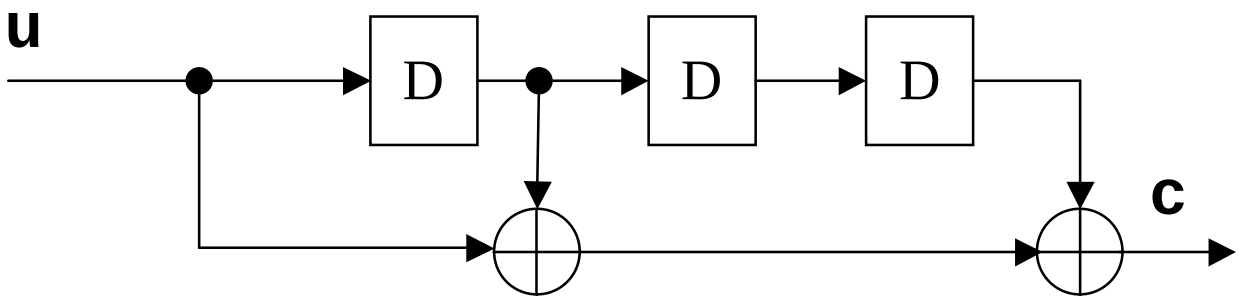


State Diagram of a Cyclic Code



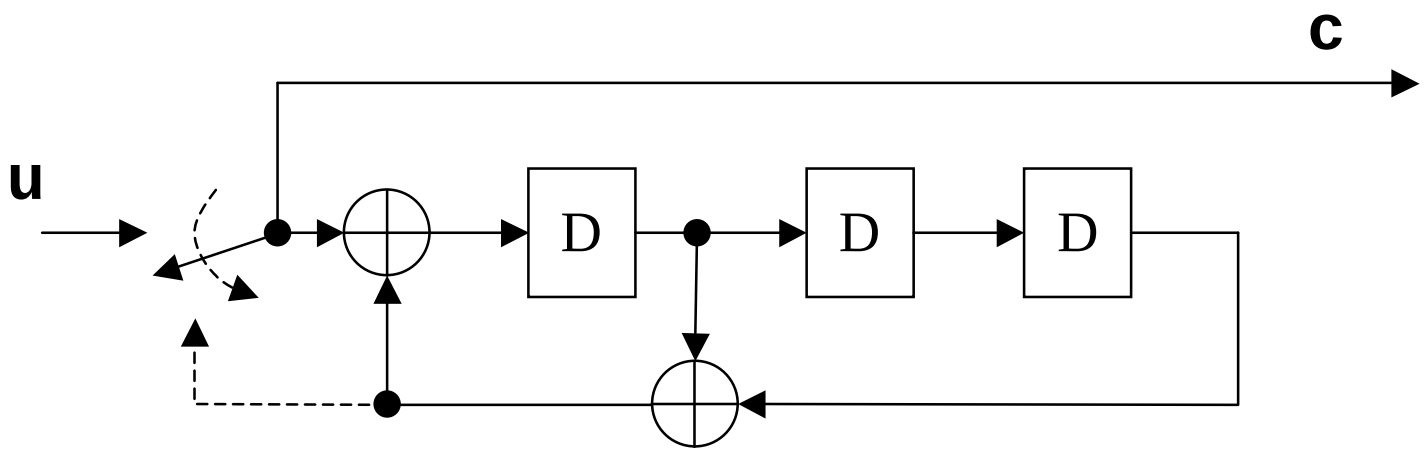
Systematic Cyclic Codes

- BTC use *systematic* cyclic codes.
- Just like an RSC, the code can be made systematic by feeding the output back to the input.
- As with an RSC, the trellis is terminated by a tail of m bits, not necessarily zeros.

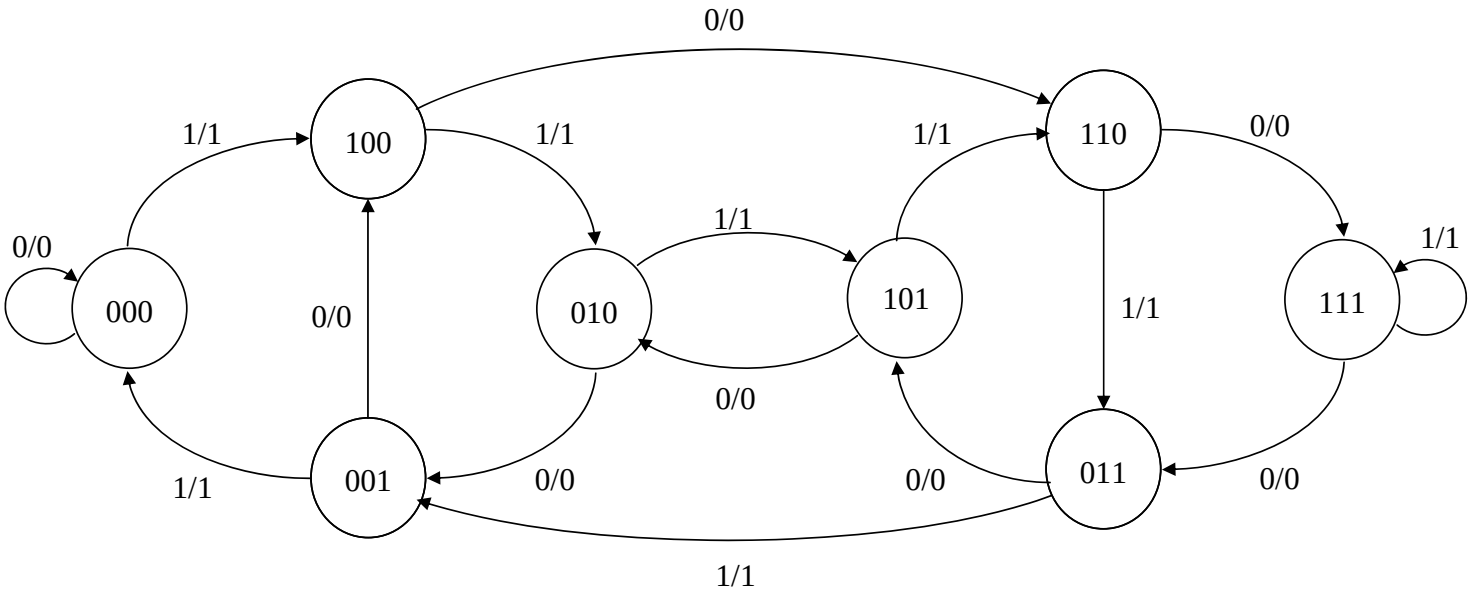


Systematic Cyclic Codes

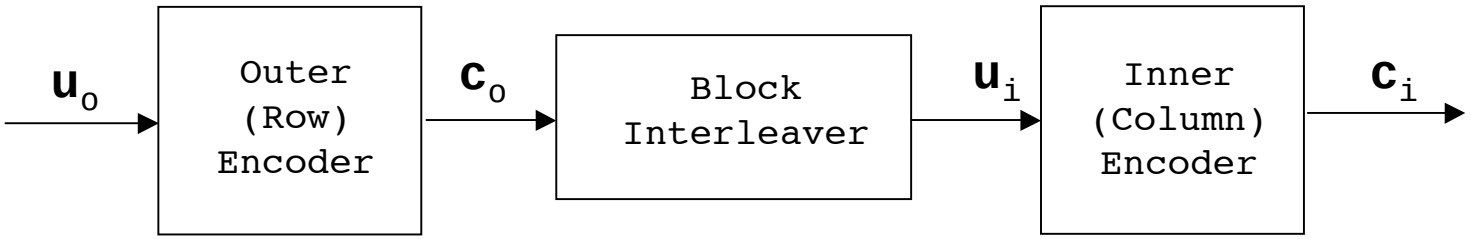
- BTC use *systematic* cyclic codes.
- Just like an RSC, the code can be made systematic by feeding the output back to the input.
- As with an RSC, the trellis is terminated by a tail of m bits, not necessarily zeros.



State Diagram of a Systematic Cyclic Code

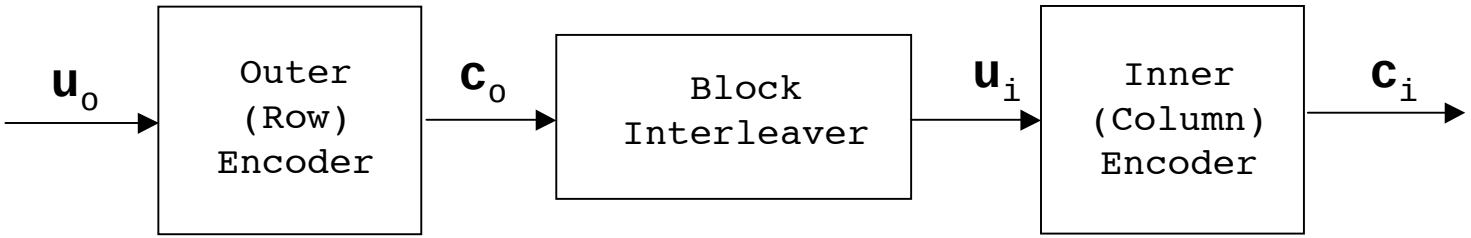


Encoder Block Diagram



- The outer encoder produces k_y row codewords.
- Each row codeword written into the rows of the block interleaver.
- Information read from each column of the block interleaver.
- The inner encoder produces n_x column codewords.
- The column codewords are transmitted over the channel.

Decoder Block Diagram

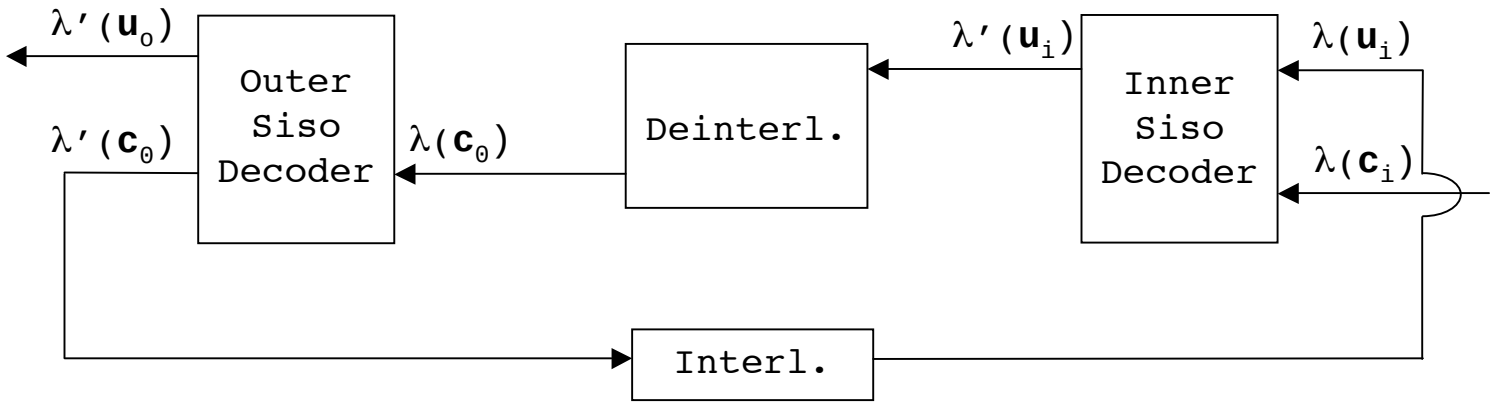


- Initially, pass soft outputs from outer decoder to inner decoder.
- The deinterleaver reverses the action of the interleaver.
- Make a hard decision on $\lambda'(u_o)$.
- If data is incorrect, feed back soft-info from inner to outer decoder.
- To prevent positive feedback, subtract input from output so that it is actually *extrinsic information* that gets fed back.

Decoder Block Diagram

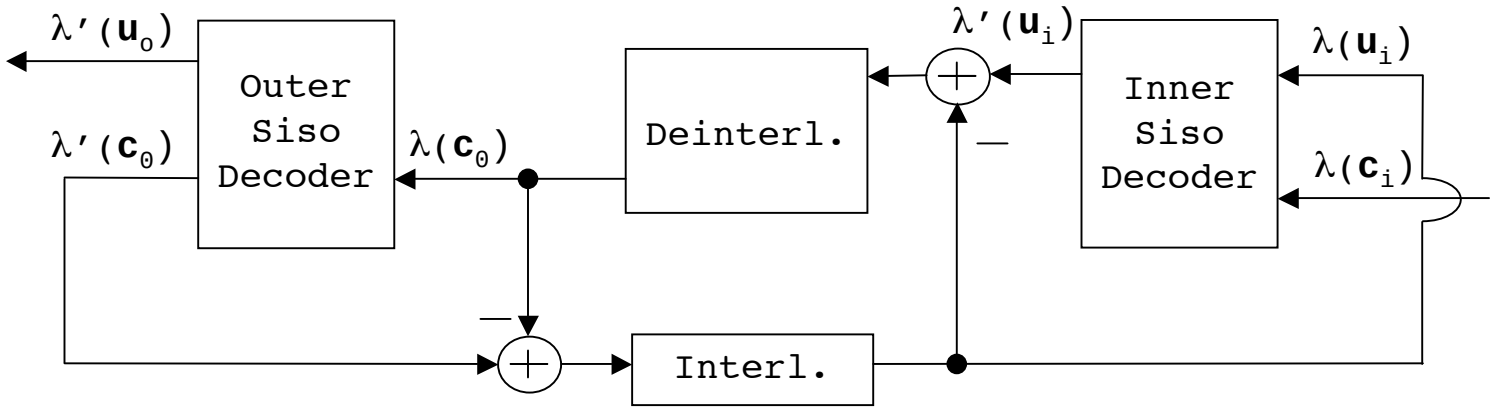
- Initially, pass soft outputs from outer decoder to inner decoder.
- The deinterleaver reverses the action of the interleaver.
- Make a hard decision on $\lambda'(u_o)$.
- If data is incorrect, feed back soft-info from inner to outer decoder.
- To prevent positive feedback, subtract input from output so that it is actually *extrinsic information* that gets fed back.

Decoder Block Diagram



- Initially, pass soft outputs from outer decoder to inner decoder.
- The deinterleaver reverses the action of the interleaver.
- Make a hard decision on $\lambda'(u_o)$.
- If data is incorrect, feed back soft-info from inner to outer decoder.
- To prevent positive feedback, subtract input from output so that it is actually *extrinsic information* that gets fed back.

Decoder Block Diagram



- Initially, pass soft outputs from outer decoder to inner decoder.
- The deinterleaver reverses the action of the interleaver.
- Make a hard decision on $\lambda'(u_o)$.
- If data is incorrect, feed back soft-info from inner to outer decoder.
- To prevent positive feedback, subtract input from output so that it is actually *extrinsic information* that gets fed back.

Constituent Codes Used by 802.16e

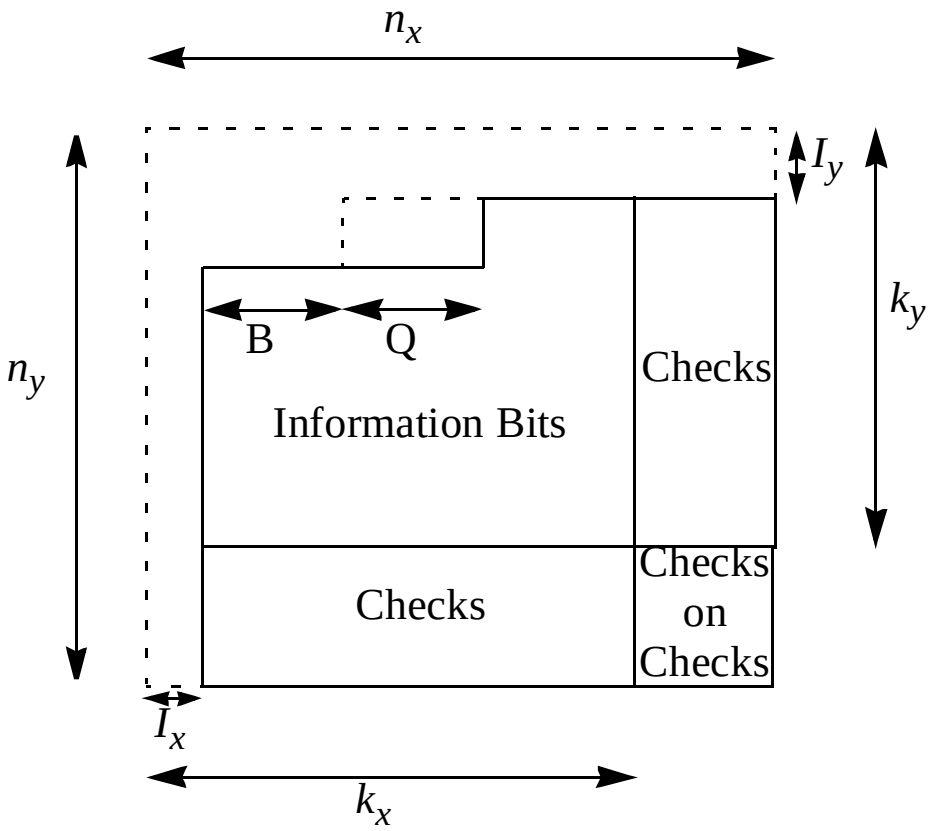
The constituent block codes may be generated using the following polynomials:

Generator polynomial	Generator vector	Shorthand
$1 + X$	$[11]$	g1
$1 + X^2 + X^4 + X^5$	$[101011]$	g5
$1 + X + X^2 + X^3 + X^5 + X^6$	$[1111011]$	g6
$1 + X^2 + X^6 + X^7$	$[10100011]$	g7

- Generator $1 + X$ is a single-parity check (SPC) code.
- The other three generators are *extended* Hamming codes.

Shortened Block Structure

Instead of a standard product code structure, the following structure is used to match the codeword size to the OFDM symbol size:



- Data size is $(k_y - I_y)(k_x - I_x) - (B + Q)$ bits.
- Pre-pad input with $(B + Q)$ zeros.
- Delete the B zeros prior to transmission.

BTC Encoder

```
>> help BtcEncode
  BTCEncode encodes a data sequence using a block turbo encoder.

  The calling syntax is:
      codeword = BtcEncode( data, grows, gcols, k_per_row, k_per_col, B, Q )

      codeword = the codeword generated by the encoder,

      data = the row vector of data bits
      grows = the generator used to encode the rows
      gcols = the generator used to encode the columns
      k_per_row = number of data bits per row
      k_per_col = number of data bits per column
      B = number of zeros padded before data but not transmitted
      Q = number of zeros padded before data and transmitted

  Copyright (C) 2008, Matthew C. Valenti and Sushma Mamidipaka
```

$\texttt{k\_per\_row} = k_x - I_x.$

$\texttt{k\_per\_col} = k_y - I_y.$

BTC Parameters

Code bytes	Data bytes	grows	gcols	k_per_row	k_per_col	B	Q
12	6	g1	g6	3	18	0	6
12	9	g1	g1	9	9	4	5
24	12	g1	g5	17	6	6	0
24	20	g1	g1	13	13	4	5
36	18	g1	g5	26	6	9	3
36	25	g1	g7	5	41	0	5
48	23	g6	g5	22	9	8	6
48	35	g6	g1	26	11	0	6
60	31	g6	g6	16	16	4	4
72	40	g6	g6	18	18	0	4

BTC Decoder

```
>> help BtcDecode
  BTCEncode decodes a block turbo code

  The calling syntax is:
      [detected_data, errors] = BTCDecode( symbol_likelihood, data, grows, gcols, k_per_row, ...
      k_per_col, B, Q, max_iterations, decoder_type )

      detected_data = a row vector containing hard decisions on the detected data
      errors = a column vector containing the number of errors per iteration

      symbol_likelihood = the decoder input, in the form of bit LLRs
      data = the row vector of data bits (used to count errors and for early halting of iterative decoding)
      grows = the generator used to encode the rows
      gcols = the generator used to encode the columns
      k_per_row = number of data bits per row
      k_per_col = number of data bits per column
      B = number of zeros padded before data but not transmitted
      Q = number of zeros padded before data and transmitted
      max_iterations = the number of turbo iterations
      decoder_type = the decoder type
               = 0 For linear-log-MAP algorithm, i.e. correction function is a straght line.
               = 1 For max-log-MAP algorithm (i.e. max*(x,y) = max(x,y) ), i.e. correction function = 0.
               = 2 For Constant-log-MAP algorithm, i.e. correction function is a constant.
               = 3 For log-MAP, correction factor from small nonuniform table and interpolation.
               = 4 For log-MAP, correction factor uses C function calls.

  Copyright (C) 2008, Matthew C. Valenti and Sushma Mamidipaka
```

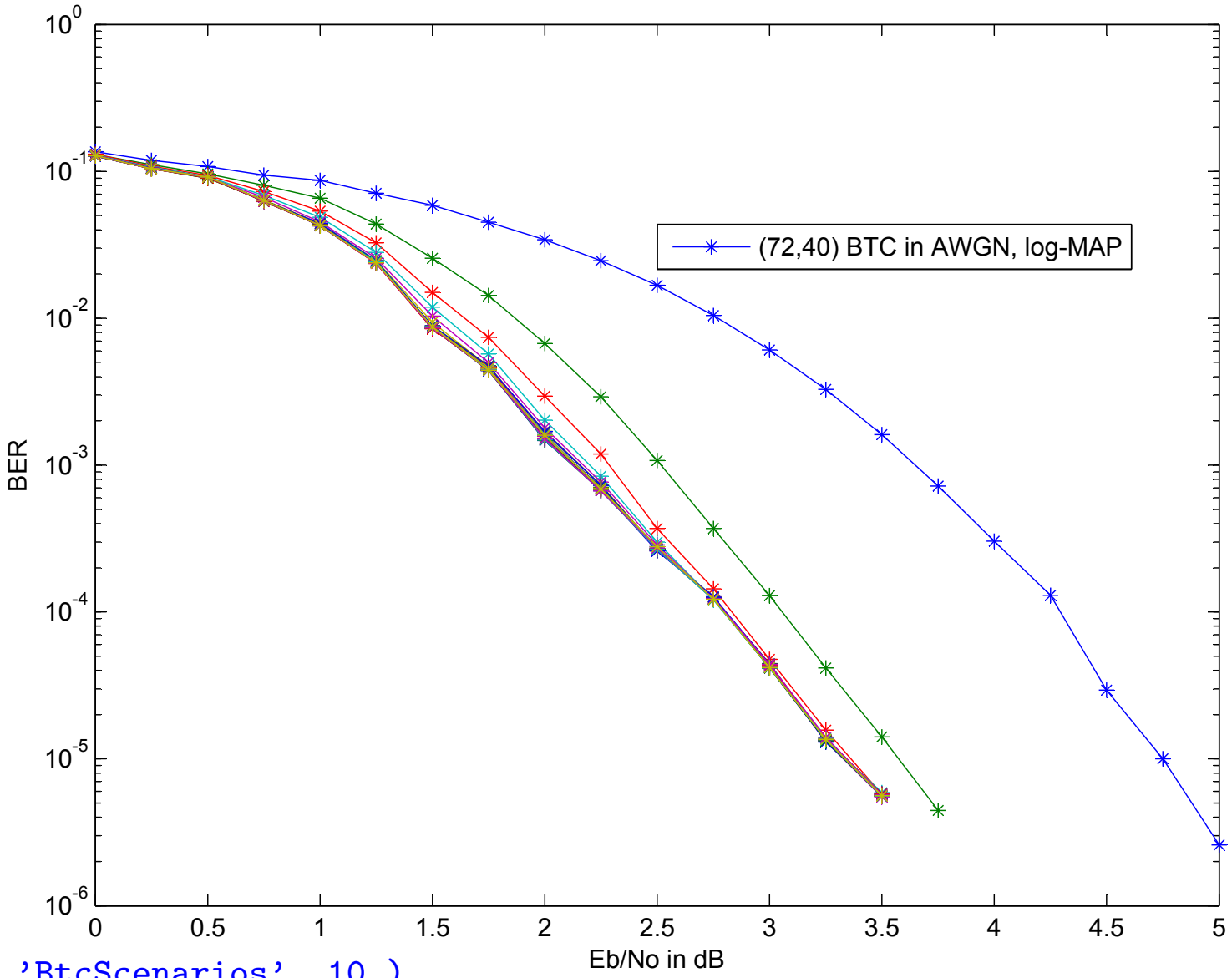
BTC in CML

Code bytes	Data bytes	grows	gcols	k_per_row	k_per_col	B	Q
36	18	g1	g5	26	6	9	3

```
>> grows = [1 1];
>> gcols = [1 0 1 0 1 1];
>> k_per_row = 26;
>> k_per_col = 6;
>> B=9;
>> Q=3;
>> data = round(rand(1,144)); % 18 data bytes
>> c = BtcEncode( data, grows, gcols, k_per_row, k_per_col, B, Q );
>> s = 2*c-1;
>> variance = 1;
>> noise = sqrt(variance)*randn(size(c));
>> r = s + noise;
>> llr = 2*r/variance;
>> [out,errors] = BtcDecode( llr, data, grows, gcols, k_per_row, k_per_col, B, Q, 6, 4 );
>> errors
errors =
     2
     2
     2
     1
     0
     0
```

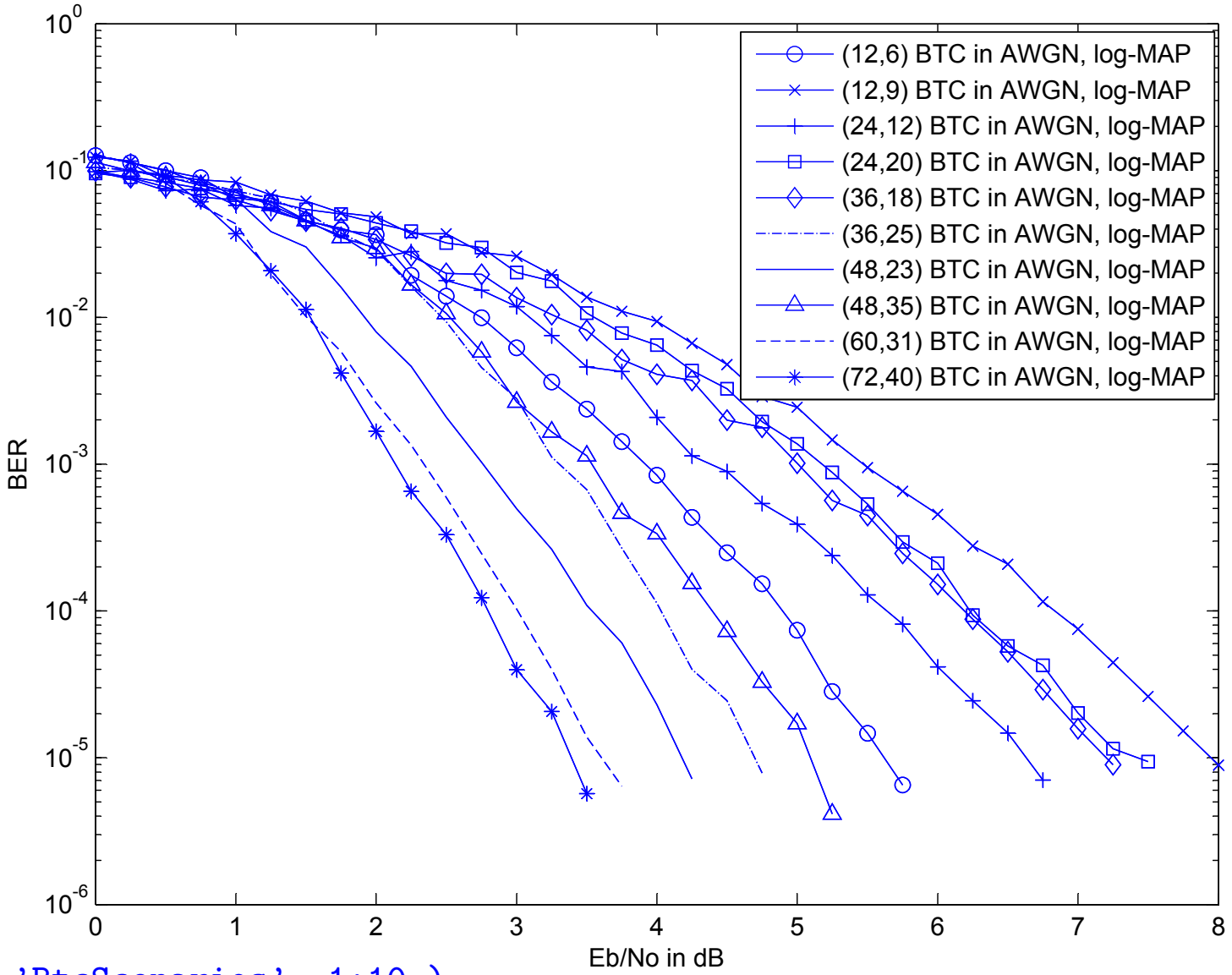
After the fifth iteration, the data is correct.

Influence of the Number of Iterations



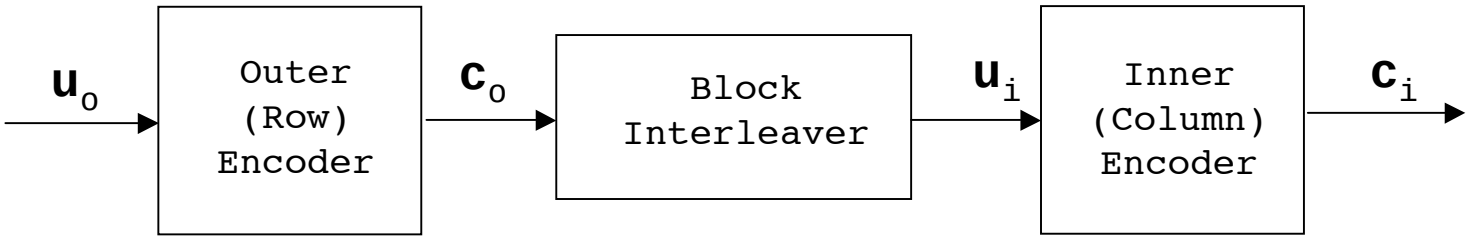

```
CmlPlot( 'BtcScenarios', 10 )
```

Performance of BTC in AWGN




```
CmlPlot( 'BtcScenarios', 1:10 )
```

Serially Concatenated Codes

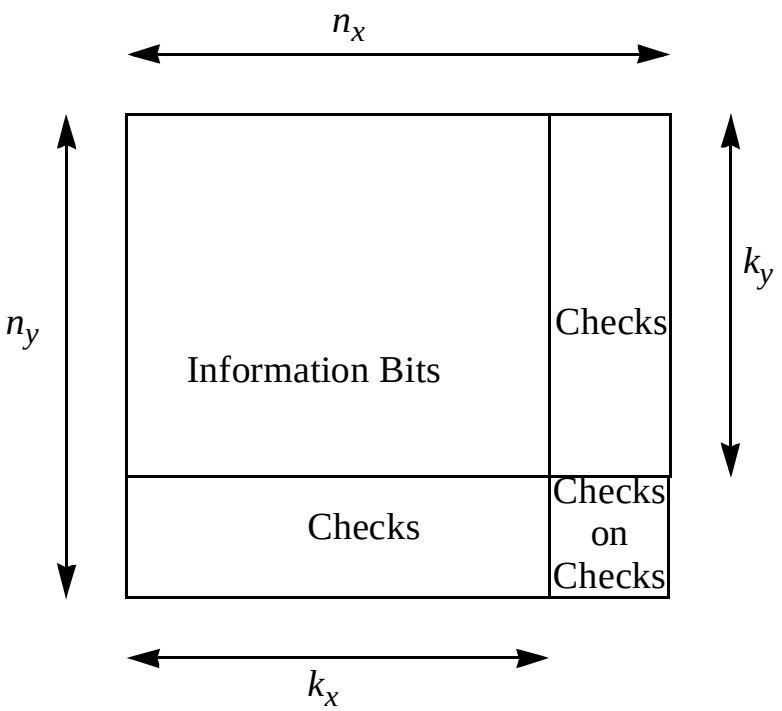


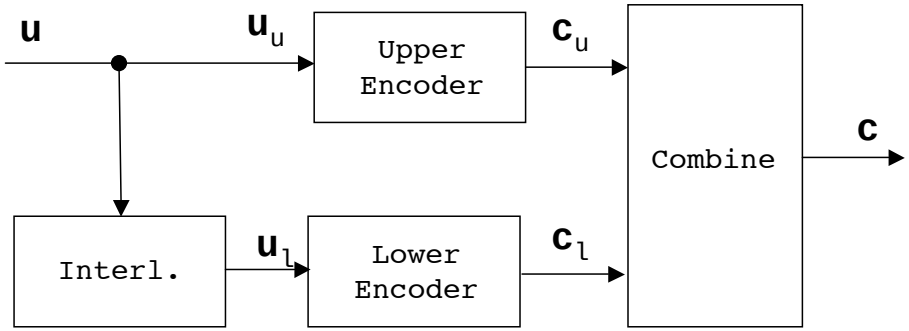
- Inner and outer codes need not be cyclic codes.
  - For instance, they could be *convolutional* codes.
- Instead of one codeword per row or column, the inner and outer encoders could each produce one long codeword.
- The interleaver need not be a *block* interleaver.
  - Instead, a *pseudo-random* interleaver could be used.
- The decoder is identical to the one shown earlier.

Serially Concatenated Codes

- Inner and outer codes need not be cyclic codes.
  - For instance, they could be *convolutional* codes.
- Instead of one codeword per row or column, the inner and outer encoders could each produce one long codeword.
- The interleaver need not be a *block* interleaver.
  - Instead, a *pseudo-random* interleaver could be used.
- The decoder is identical to the one shown earlier.

Parallel Concatenated Codes

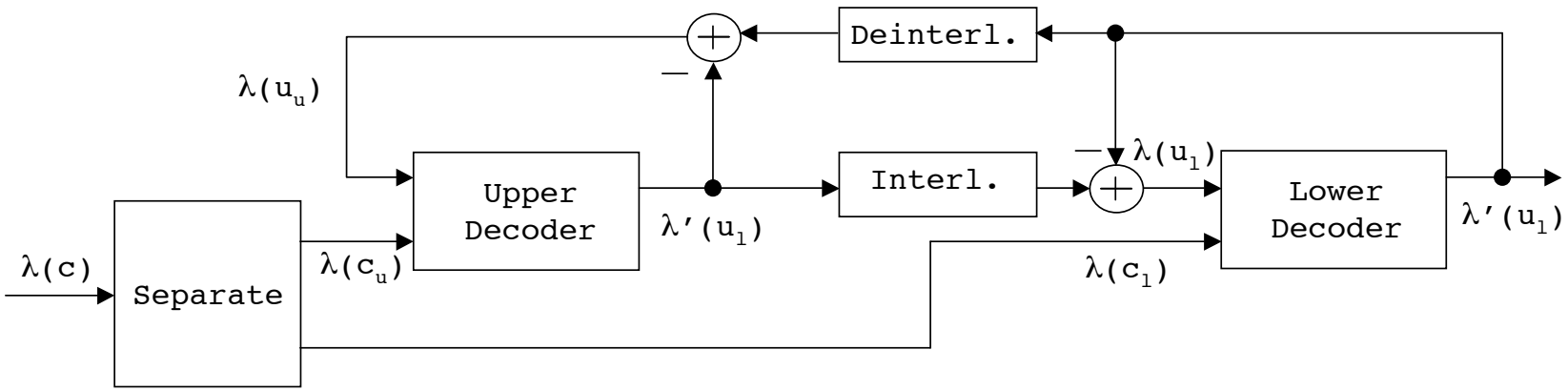




- If the double-parity (*checks on checks*) component is punctured, then encoding could be done in parallel.
- *Convolutional turbo codes* are parallel concatenated codes with convolutional constituent codes.

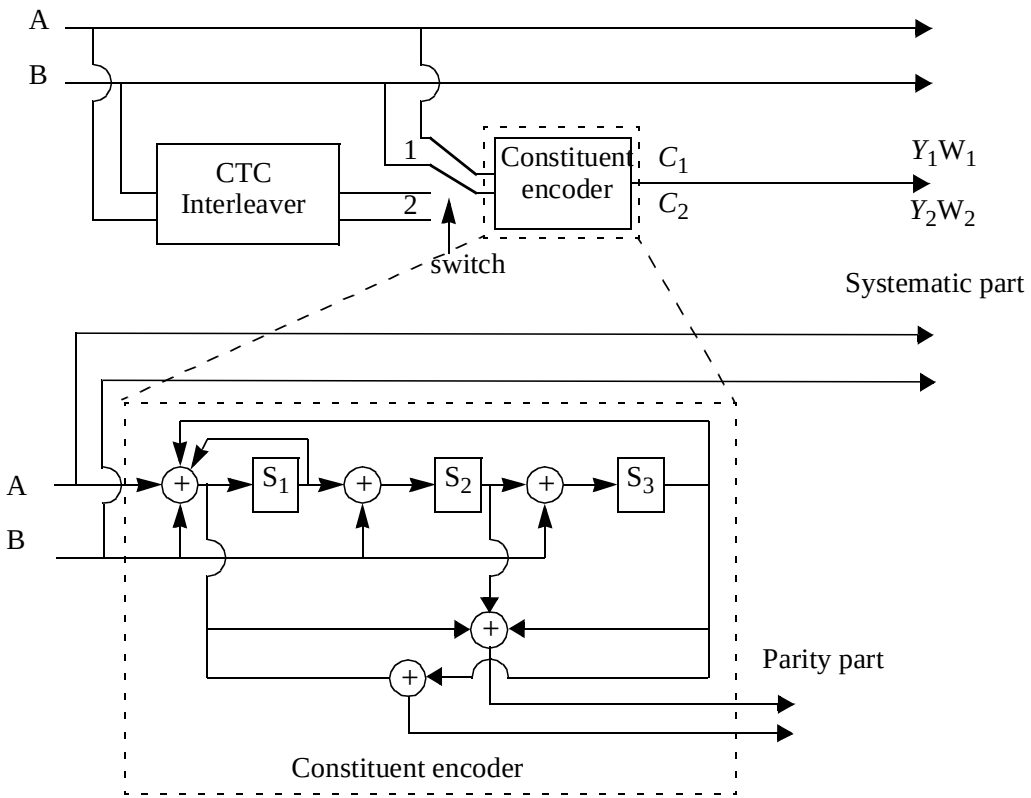
CTC Decoder

- The serial decoder could be used by setting $\lambda(c) = 0$ in the positions of the double-parity bits.
- A more efficient alternative is as follows:



CTC Encoder Used by 802.16e

The CTC specified in IEEE 802.16e uses the following constituent encoder:



Key features:

- The code is duo-binary (rate $2/4$).
- Tailbiting is used to force starting and ending states to be the same.

Encoder operation:

- With switch up, encode the data in its natural order.
- Throw switch down and encode the interleaved data.
- Puncturing can be used to increase rate above $1/2$.

CTC parameters

QPSK

- Rate 1/2 with data block size: 6, 12, 18, 24, 30, 36, 48, 54, 60 bytes
- Rate 3/4 with data block size: 9, 18, 27, 36, 45, 54 bytes

16-QAM

- Rate 1/2 with data block size: 12, 24, 36, 48, 60 bytes
- Rate 3/4 with data block size: 18, 36, 54 bytes

64-QAM

- Rate 1/2 with data block size: 36, 54 bytes
- Rate 2/3 with data block size: 24, 48 bytes
- Rate 3/4 with data block size: 27, 54 bytes
- Rate 5/6 with data block size: 30, 60 bytes

Preliminaries: Creating the Interleaver

```
>> help CreateWimaxInterleaver
  CreateWimaxInterleaver intializes an interleaver for use with the duobinary
  tailbiting turbo code (CTC) from the mobile WiMAX (IEEE 802.16e) standard.

  The calling syntax is:
      code_interleaver = CreateWimaxInterleaver(Nbits)

      code_interleaver = a structure with the following members
      code_interleaver.subblk_intl = subblock bit-wise interleaver, 1 by Nbits/2 vector
      code_interleaver.info_intl = information interleaver over GF(2),
                                   a row vector of length Nbits.

      Nbits = number of bits (i.e. twice the number of couples)
              valid range = {24 36 48 72 96 108 120 144 180 192 216 240 480 960 1440 1920 2400}

  Copyright (C) 2007, Matthew C. Valenti and Shi Cheng
```

Preliminaries: Creating the Puncturing Pattern

```
>> help CreateWimaxPuncturingPattern
  CreateWimaxPuncturingPattern creates the puncturing pattern for use with the duobinary
  tailbiting turbo code (CTC) from the mobile WiMAX (IEEE 802.16e) standard.

  The calling syntax is:
      pun_pattern = CreateWimaxPuncturingPattern( Nbits_pun, Nbits_unpun )

      pun_pattern = the puncturing pattern (length Nbits_pun vector)

      Nbits_pun = the number of bits after puncturing
      Nbits_unpun = the number of bits prior to puncturing (not used)

  Copyright (C) 2007, Matthew C. Valenti and Shi Cheng

  Last updated on Oct. 12, 2007
```

CTC Encoder

```
>> help TurboDuobinaryCRSCEncode
  TurboDuobinaryCRSCEncode encodes a data sequence using a duobinary tailbiting
  turbo encoder.

  The calling syntax is:
      codeword = TurboDuobinaryCRSCEncode( data, code_interleaver, pun_pattern )

      codeword = the codeword generated by the encoder (a row vector)

      data = the row vector of data bits
      code_interleaver = the turbo interleaver
      pun_pattern = the puncturing pattern

  Copyright (C) 2007, Matthew C. Valenti and Shi Cheng
```

CTC Decoder

```
>> help TurboDuobinaryCRSCDecode
  TurboDuobinaryCRSCDecode deccodes a data sequence that was encoded by a
  duobinary tailbiting turbo encoder.

  The calling syntax is:
      [detected_data, errors] = TurboDuobinaryCRSCDecode( llr, code_interleaver, pun_pattern, data, ...
                                max_iterations, decoder_type )

      detected_data = a row vector containing the detected data
      errors = a column vector containing the number of errors per
               iteration for all the codewords.

      llr = received LLR of the code bits.
      code_interleaver = the turbo interleaver
      pun_pattern = the puncturing pattern
      data = the row vector of data bits (for early halting)
      max_iterations = maximum number of decoder iterations
      decoder_type = the decoder type
               = 0 For linear-log-MAP algorithm, i.e. correction function is a straght line.
               = 1 For max-log-MAP algorithm (i.e. max*(x,y) = max(x,y) ), i.e. correction function = 0.
               = 2 For Constant-log-MAP algorithm, i.e. correction function is a constant.
               = 3 For log-MAP, correction factor from small nonuniform table and interpolation.
               = 4 For log-MAP, correction factor uses C function calls.

  Copyright (C) 2007, Matthew C. Valenti and Shi Cheng
```

CTC in CML

```
>> n = 768; % 96 code bytes
>> k = 384; % 48 data bytes
>> interl = CreateWimaxInterleaver( k ); % Create the Interleaver
>> punc = CreateWimaxPuncturingPattern( n ); % Create the Puncturing Pattern
>> Sqam = CreateConstellation( 'QAM', 16); % A 16-QAM constellation
>> data = round( rand(1,k) ); % random data
>> codeword = TurboDuobinaryCRSCEncode( data, interl, punc ); % encode
>> s = Modulate( codeword, Sqam ); % 16-QAM modulator
>> EsNo = 10^(8/10); % Es/No = 8 dB
>> variance = 1/(2*EsNo);
>> noise = sqrt(variance)*(randn(size(s))+j*randn(size(s))); % 2-D (complex) noise
>> r = s + noise; % add noise
>> sym_likelihood = Demod2D( r, Sqam, EsNo ); % 16-QAM demod ~ front end
>> llr = Somap( sym_likelihood ); % back end of demod
>> [out, errors] = TurboDuobinaryCRSCDecode( llr, interl, punc, data, 8, 4 );
>> errors
errors =
    39
    22
     9
    13
     1
     0
     0
     0
```

After the sixth iteration, the data is correct.

For simulation results, refer to the *WiMaxCTCScenarios.m* file.

Outline

Parity-check Matrices.

- Let H be a rank $m = n - k$ matrix such that for every $\mathbf{c} \in \mathcal{C}$

$$\mathbf{c}H^T = \mathbf{0}$$

- A given $\mathcal{C}$ may be represented by more than one H.
- H has n columns and (at least) m rows.
- Example: Cyclic Hamming code with $g(X) = 1 + X + X^3$

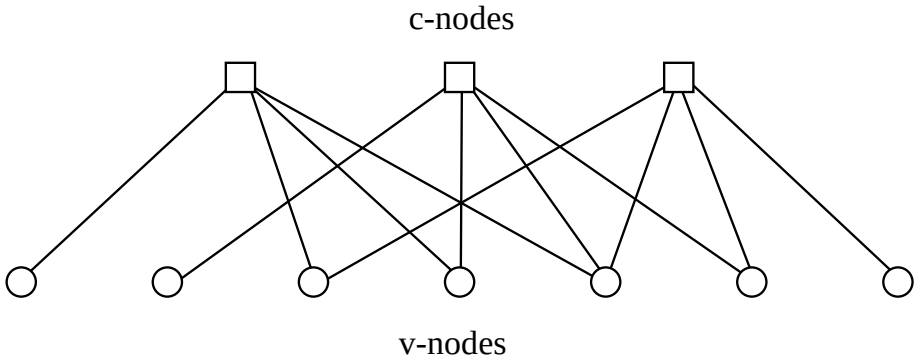
$$H = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

- An equivalent parity check matrix:

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

Tanner Graphs and MP Decoding

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



A *Tanner graph* is a graphical representation of H.

- Each row of H is represented by a *check* node.
- Each column of H is represented by a *variable* node.

Decoding is via a *message passing* (MP) algorithm.

- Likelihoods passed back and forth between c-nodes and v-nodes.
- Iterative process.
- Sum-product or min-sum.

LDPC Codes

$$H = \left[\begin{array}{cccc|cccc|cccc|cccc}
1 & 1 & 1 & 1 & & & & & & & & & & & & \\
 & & & & 1 & 1 & 1 & 1 & & & & & & & & \\
 & & & & & & & & 1 & 1 & 1 & 1 & & & & \\
 & & & & & & & & & & & & 1 & 1 & 1 & 1 \\
\hline
1 & & & & 1 & & & & 1 & & & & 1 & & & \\
 & 1 & & & & 1 & & & & 1 & & & & 1 & & \\
 & & 1 & & & & 1 & & & & 1 & & & & 1 & \\
 & & & 1 & & & & 1 & & & & 1 & & & & 1 \\
\hline
 & & 1 & & & & & 1 & 1 & & & & & & & 1 \\
 & 1 & & & 1 & & & & & 1 & & & & & 1 & \\
 & & & 1 & & 1 & & & & & & 1 & & 1 & & \\
1 & & & & & & 1 & & & & 1 & & 1 & & & \\
\end{array}\right]$$

- A *low-density parity check* code is a code that may be represented by a sparse H matrix.
- LDPC codes may be *regular* or *irregular*.
- LDPC codes are iteratively decoded using a message-passing decoder.

LDPC Codes and Encoding

Encoding of LDPC codes is not necessarily straightforward.

- "Systematic-form" H
  - Using Gaussian elimination, find $H = [P\ I]$.
  - Then $\mathbf{c} = \mathbf{u}G$ where $G = \left[I\ P^T\right]$.
  - However, P is likely to be high-density (complex encoding).
- Back-substitution.
  - If H is in an appropriate form, then $\mathbf{c}$ can be encoded using *back substitution*
  - Example, $\mathbf{c}H^T = \mathbf{0}$, where

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

- For any LDPC code, it is possible to encode with an H matrix that permits back-subsitution.
  - When this is done, a few rows will be high-density and therefore have higher complexity
  - The high-density part of the encoding H is called the *gap*.

The IEEE 802.16e standard LDPC Codes

Code rates:

- $r = 1/2, 2/3, 3/4$, and $5/6$

Block sizes:

- 72 to 288 code bytes.
- Increments of 12 bytes for QPSK.
- Increments of 24 bytes for 16-QAM
- Increments of 36 bytes for 64-QAM.

H matrices specified in standard

- H is made up of circulant submatrices.
- Encoding via back-substitution with gap of $n/24$.

Preliminaries: Creating the Parity-check Matrix

```
>> help InitializeWiMaxLDPC
    InitializeWiMaxLDPC initializes the WiMax LDPC encoder/decoder

     The calling syntax is:
        [H_rows, H_cols, P] = InitializeWiMaxLDPC( rate, size, ind )

     Where:
        H_rows = a M-row matrix containing the indices of the non-zero rows of H
                 excluding the dual-diagonal portion of H.
        H_cols = a (N-M)+z-row matrix containing the indices of the non-zeros rows of H.
        P      = a z times z matrix used in encoding
        rate = the code rate
        size = the size of the code (number of code bits):
             = 576:96:2304
        ind  = Selects either code 'A' or 'B' for rates 2/3 and 3/4
             = 0 for code rate type 'A'
             = 1 for code rate type 'B'
             = [empty array] for all other code rates

     Copyright (C) 2007-2008, Rohit Iyer Seshadri and Matthew C. Valenti
```

LDPC Encoder

```
>> help LdpcEncode
  LdpcEncode encodes an LDPC codeword.  Code must be an "eIRA-LDPC" type code, such as the one
  in the DVB-S2 standard, or WiMax standard.

  The calling syntax is:
  codeword    = LdpcEncode( data, H_rows, [P])

  Where:
       codeword = the encoded codeword
       data   = a row vector containing the data
       H_rows = a M-row matrix containing the indices of the non-zero rows of H
                excluding the dual-diagonal portion of H.
       P      = (optional) z times z matrix used to generate the first z check bits
                for WiMax (default =[])
```

LDPC Decoder

```
>> help MpDecode
  MpDecode decodes a block code (e.g. LDPC) using the message passing algorithm.

  The calling syntax is:
      [output, errors] = MpDecode(input, H_rows, H_cols, [max_iter], [dec_type], ...
                         [r_scale_factor], [q_scale_factor], [data] )

      Outputs:
      output = matrix of dimension maxiter by N that has the decoded code bits for each iteration
      errors = (optional) column vector showing the number of (code bit) errors after each iteration.

      Required inputs:
      input  = the decoder input in LLR form
      H_cols = a N row matrix specifying the locations of the nonzero entries in each column of the H matrix.
               The number or columns in the matrix is the max column weight.
               OR
               a K row matrix specifying locations of the nonzero entries in each coulmn
               of an extended IRA type sparse H1 matrix
      H_rows = a N-K row matrix specifying the locations of the nonzero entries in each row of the H matrix.
               The number or columns in the matrix is the max row weight, unless this is for an H1 matrix,
               in which case the last n-k columns of the H matrix are equal to a known H2 matrix.

      Optional inputs:
      max_iter = the maximum number of decoder iterations (default = 30).
      dec_type = the decoder type:
               = 0 Sum-product (default)
               = 1 Min-sum
      r_scale_factor = amount to scale extrinsic output of c-nodes in min-sum decoding (default = 1)
      q_scale_factor = amount to scale extrinsic output of v-nodes in min-sum decoding (default = 1)
      data = a vector containing the data bits (used for counting errors and for early halting)
             (default all zeros)
```

LDPC in CML

```
>> n = 576; % number of code bits
>> rate = 1/2;  % code rate
>> [H_rows, H_cols, P] = InitializeWiMaxLDPC( rate, n );
>> k = length( H_cols ) - length( P ); % number of data bits
>> Sqam = CreateConstellation( 'QAM', 16); % A 16-QAM constellation
>> data = round( rand(1,k) ); % random data
>> codeword = LdpcEncode( data, H_rows, P); % encode
>> s = Modulate( codeword, Sqam ); % 16-QAM modulator
>> EsNo = 10^(8/10); % Es/No = 8 dB
>> variance = 1/(2*EsNo);
>> noise = sqrt(variance)*(randn(size(s))+j*randn(size(s))); % 2-D (complex) noise
>> r = s + noise; % add noise
>> sym_likelihood = Demod2D( r, Sqam, EsNo ); % 16-QAM demod ~ front end
>> llr = Somap( sym_likelihood ); % back end of demod
>> [output, errors] = MpDecode(-llr, H_rows, H_cols, 20, 0, 1, 1, data );
>> errors
errors =
    35
    34
    31          ****************************************
    20          * For simulation results,                *
    23          * refer to the WiMaxLDPCScenarios.m file *
    24          ****************************************
    18
    17
    13
     9
     5
     2
     2
     0 <-- After the 14th iteration, the data is correct
```

Outline

Conclusion

- IEEE 802.16e-2005 contains a wide variety of channel coding options.
- The WiMAX profiles help to constrain the channel coding options.
- Codeword lengths are small to moderate length, to match the size of an OFDM symbol.
- Use of tailbiting convolutional codes eliminates the overhead of a tail.
- Turbo and LDPC codes offer near-capacity performance with iterative decoding.
- Implementations of all codes in the standard are freely available in CML.

Thank You.