Seventh Lecture
October 30, 2023

Solution of the home works:

Exercise 1: This exercise is similar to the previous example, only the numbers differ. Let the 3×3 transition probability matrix Π of a Markov chain $\mathbb{Z}$ with three states A, B, C have first row: $7/8, 1/8, 0$, second row: $0, 7/8, 1/8$, third row: $1/3, 1/3, 1/3$. Determine the entropy of the source whose outcome is the actual state of this Markov chain.

Solution: We need to calculate the stationary distribution. Let a, b, c denote the stationary probabilities of the system being in state A, B, C, respectively. Then from the first column we have $a = \frac{7}{8}a + \frac{1}{3}c$ giving $a = \frac{8}{3}c$ and from the third column we have $c = \frac{1}{8}b + \frac{1}{3}c$ giving $b = \frac{16}{3}c$. Using $a + b + c = 1$ we obtain $\frac{8}{3}c + \frac{16}{3}c + c = 1$ which implies $c = \frac{3}{27} = \frac{1}{9}$. Thus $a = \frac{8}{27}, b = \frac{16}{27}$ and the requested entropy value is

$$H(X) = \frac{8}{27}H(X_n|X_{n-1} = a) + \frac{16}{27}H(X_n|X_{n-1} = b) + \frac{1}{9}H(X_n|X_{n-1} = c) =$$
$$\frac{8}{27}h(1/8) + \frac{16}{27}h(1/8) + \frac{1}{9}\log 3 = \frac{8}{9}h(1/8) + \frac{1}{9}\log 3.$$

$\diamondsuit$

Exercise 2: Let $X_1, X_2, \ldots$ be a Markov chain for which $\text{Prob}(X_1 = 0) = \text{Prob}(X_) = 1) = \frac{1}{2}$ and let the transition probabilities for $i \geq 1$ be given by $\text{Prob}(X_{i+1} = 0|X_i = 0) = \text{Prob}(X_{i+1} = 1|X_i = 0) = \frac{1}{2}$, while $\text{Prob}(X_{i+1} = 0|X_i = 1) = 0$ and $\text{Prob}(X_{i+1} = 1|X_i = 1) = 1$. Calculate the entropy of the source whose outcome is the resulting sequence of random variables $X_1, X_2, \ldots$.

Intuitively the solution is quite clear: This source emits some number (perhaps zero) 0's first, but after the first 1 it will emit only 1's. As i gets larger and larger, the probability of $X_i = 0$ is smaller and smaller (in fact it will be $\frac{1}{2^i}$), so if i is large, then X_i is almost certainly 1. Therefore the uncertainty about the value of X_i approaches zero, so the entropy of the source should be 0.

This intuition is easy to confirm by calculation: by Theorem **??**

$$H(X) = \lim_{n\to\infty} H(X_n|X_1, \ldots, X_{n-1}) = \lim_{n\to\infty} H(X_n|X_{n-1}) =$$
$$\lim_{n\to\infty} \text{Prob}(X_{n-1} = 0)h(1/2) + \text{Prob}(X_{n-1} = 1)h(0) =$$
$$\lim_{n\to\infty} \frac{1}{2^{n-1}} + (1 - \frac{1}{2^{n-1}})0 = 0.$$

Note that the exercise can also be solved similarly as the previous one: realizing that the stationary distribution is concentrated on the value 1 we get that the entropy of the source is $0 \cdot h(1/2) + 1 \cdot h(1) = 0$.
$\diamondsuit$

Universal source coding, Lempel-Ziv type algorithms

Huffmann code gives optimal average length but it assumes knowledge of source statistics: we expected we know the probabilities p_i with which the characters x_i are emitted by the source. When we have to compress information it may not be so. Or we want to compress earlier than we could know such statistics. (Think about compressing a text. In principle we could first read it through, make the source statistics and then encode. But we may prefer to encode right in the moment we proceed with its reading) The term *universal source coding* refers to coding the source in such a way that we do not have to know the source statistics in advance. The following algorithms are devised to such situations. Although they usually cannot provide as good compression as the Huffman code, they still do pretty well. Perhaps surprisingly, it can be shown that their compression rate approaches the entropy rate of the source, that is the theoretical limit. (We will learn the technique without proving this.) The examples provided in different files (see references to them below) are from the textbook written in Hungarian by László Györfi, Sándor Győri, and István Vajda "Információ- és kódelmélet" (Information Theory and Coding Theory), published by Typotex, 2000, 2002.

Remark: We discussed when we learned about Huffman codes that Huffman encoding can also be adapted to the situation when we cannot create the optimal code in advance. Then we simply use the empirical source statistics. This means that the probability of a character is considered to be equal to the proportion of the number of its appearances up to a certain point to the number of characters seen altogether up to that point. This statistics is updated after every single character. For every new character we use the Huffman code that would belong to the source statistics we calculated right before the arrival of that character. We encode the character according to that code and then update the statistics and modify the code accordingly. This can be done simultaneously at the decoder (without communication), so the decoder will always be aware of the code the encoder is using. For the latter we need a rule how to handle equal probabilities that could result in different optimal encodings, but such rules are not hard to agree on. Also, the encoder and the decoder has to agree on a starting code. This can be one that assumes uniform distribution on the source characters or one that estimates the source statistics according to some a priori knowledge if that exists. (For example, if the source is an English text, we can use the more or less known frequency of each letter in the English language as the starting distribution and a Huffman encoding of that.) Although this is possible, the methods to be discussed below are simpler.

We are going to discuss three versions of the same method: LZ77 (suggested by Lempel and Ziv in 1977), LZ78 (suggested by Lempel and Ziv in 1978) and LZW (which is a modification of LZ78 suggested by Welch).

First version: LZ77

There is s sliding window in which we see $l_w = l_b + l_a$ characters, where l_b is the number of characters we see backwards and l_a is the number of characters we see ahead. The algorithm looks at the not yet encoded part of the character flow in the "ahead part" of the window and looks for the longest identical subsequence in the window that starts earlier. The output of the encoder is then a triple (o, l, c), where o is the number of characters we have to step backwards to the start of the longest subsequence identical to what comes ahead, l is the length of this longest identical subsequence, and c is the codeword for the first new character that is already not fitting in this longest subsequence. Note that the longest identical subsequence should start in the backward part of the window but may end in the ahead part, so o may be less than l.

For example, when we are encoding the sequence *...cabracadabrarrarrad...*, the *...cabraca* part is already encoded (so the coming part is *dabrarrarrad...*), and we have $l_b = 7, l_a = 6$, then the first triple sent is $(0, 0, f(d))$ (where $f(.)$ is the codeword for the character in the argument), the second triple sent is $(7, 4, f(r))$, etc. Note that for the next triple we will have $(3, 5, f(d))$ showing an example when $o < l$. We discussed that the decoder can get back the original text from this code, since by the time such a

character should be copied that has not been encoded before, i.e. was not in the search window, it is already in its place.

Second version: LZ78

In LZ77 we build on the belief that similar parts of the text come close to each other. The LZ78 version needs only that substantial parts are repeated but they do not have to be right after each other. Also, LZ77 is sensitive to the window size. As an example suppose that we wish to encode the following text: $abcdefgabcdefgabcdefg$.... If $l_b = 6$, then we never find an identical subsequence though our text is periodic. In LZ78 we do not have this disadvantage.

We build a codebook and each time we encode we look for the longest new segment that already appears in the codebook. The output is a pair (i, c) where i is the index of the longest coming segment that already has a codeword and c is the first new character after it. Apart from producing this output the algorithm also extends the codebook by putting into it the shortest not yet found segment, which is the concatenation of the segment with index i we found and the character c. This new, one longer segment gets the next index and then we go on with the encoding. The words of the codebook are stored in *TRIE* datastructure. It is a tree, where the nodes are the indexes and the edges are labelled with the characters. We get the words of the codebook by concatenating the labels of neighbouring edges starting from the root.

LZ78

uses real dictionary / codebook

important to ensure that the encoder & the decoder build the same dict.

code: $\langle i, c \rangle$

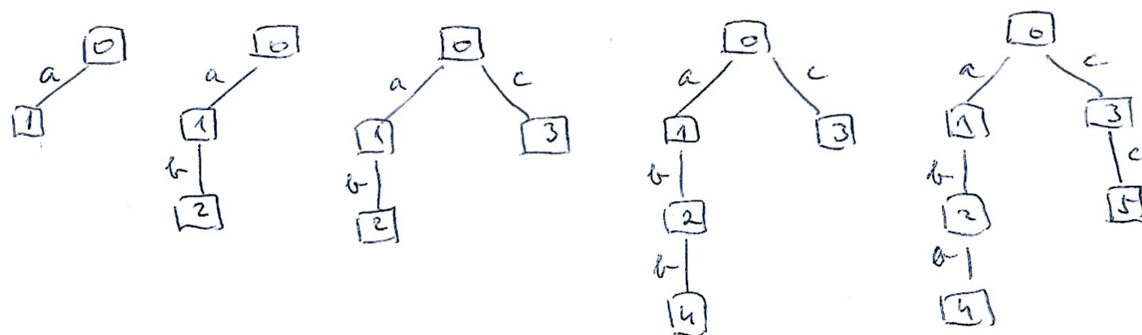
↑ index of the word in the dict.

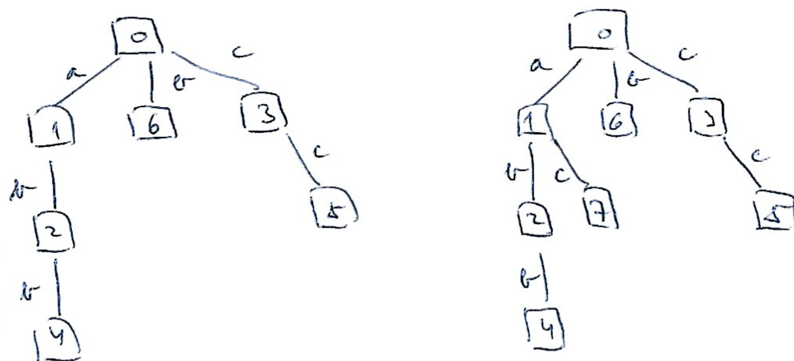
↳ code of the next symbol

and this new word (longer by 1 symbol) is put into the dict.

dictionary – ~~[illegible]~~ TRIE data structure (retrieval → centering 4 letters)

example: $X = \{a, b, c\}$ a|ab|c|abb|cc|b|ac|





Code: $\langle 0, a \rangle$ $\langle 1, b \rangle$ $\langle 0, c \rangle$ $\langle 2, b \rangle$ $\langle 3, c \rangle$ $\langle 0, b \rangle$

$\langle 1, c \rangle$

Third version: LZW

This is the most popular version of the algorithm that is a modification of LZ78 as suggested by Welch. We now start with a codebook that already contains all the one-character sequences. (They have an index which serves as a codeword for them; we can think about their codeword as the s-ary, or simply binary representation of this index.) We now read the longest new part p of the text that can be found in the codebook and the next character, let it be a. Then the output is simply the index of p, we extend the codebook with the new sequence pa (that we obtain by simply putting a to the end of p) giving it the next index, and we consider the extra character a as the beginning of the not yet encoded part of the text.

See the example on the next page.

Home-work

Let the source alphabet be $\mathcal{X} = \{a, b, c\}$ and the initial dictionary contain the letters a, b and c with their indexes (1, 2 and 3 respectively). Using the Lempel-Ziv-Welch algorithm
(a) encode the sequence *cabcbcbcb*
(b) decode the sequence $3, 4, 5, 6, 7, 1$

dabbacdabbacdabbacdabbacdeecdeecdee

4, 1, 2, 2, 1, 3, 6, 8, 10, 12, 9, 11, 7, 16, 4, 5, 5, 11, 21, 23, 5

index	bejegyzés
1	*a*
2	*b*
3	*c*
4	*d*
5	*e*
6	*da*
7	*ab*
8	*bb*
9	*ba*
10	*ac*
11	*cd*
12	*dab*
13	*bba*

index	bejegyzés
14	*acd*
15	*dabb*
16	*bac*
17	*cda*
18	*abb*
19	*bacd*
20	*de*
21	*ee*
22	*ec*
23	*cde*
24	*eec*
25	*cdee*