Universal Source Coding

Coding without distribution

Radu Trîmbițaș

UBB

December 2012

Outline I

1 Universal Compression
- Codes

2 Arithmetic Codes
- Arithmetic Coding
- Example
- Interval division
- Encoding
- Decoding
- Analysis

3 Lempel-Ziv Codes
- Dictionary Coding
- LZ77
- Improvements
- LZ78
- Improvements

Outline II

4 Optimality of Lempel-Ziv Algorithms
- Sliding Window Lempel-Ziv Algorithms

Universal Compression

- Sometimes the pmf is unknown and its estimation requires two passes over the data.
- Are there any purely algorithmic forms of compression that can be shown to relate to H and, ideally, can be shown to compress to the entropy rate?
- Note, that if it is algorithmic, it would be useful if it doesn't need to explicitly compute the probability distribution governing the symbols.
- I.e., do there exist compression algorithms that do not use the probability distribution but still give the entropy rate?

Fixed Length Block Codes

- We had fixed number of source symbols, fixed code length (fixed length codewords)
- ex: AEP, and the method of types
- Only a small number of source sequences gave code words
- Good for entropy and proofs of existence of codes, but not very practical

Symbol Codes

- Variable length codewords for each source symbol.
- More probable symbols gave shorter length encodings.
- Ex: Huffman codes
- Need distribution, and penalty if a mismatch of $D(p||q)$
- Still requires blocking of the symbols in order to achieve the entropy rate (which occurs in the limit).

Stream Codes

- Here, we do not necessarily emit bits for every source symbol, might need to wait and then emit bits for a sequence which could be variable length.
- More large number of source symbols can be represented with small number of bits without blocking.
- Can start encoding quickly, and can start decoding before encoding is even finished.
- Ex: Arithmetic codes (uses a subinterval of $[0, 1]$ corresponding to $p(x_n | x_1, \ldots, x_{n-1})$) and requires a model (ideally adaptive) of the source sequence.
- Ex: *adaptive arithmetic codes* (Dirichlet distribution).
- Another example: *Lempel-Ziv*: memorize strings that have already occurred, don't even model the source distribution (skip that step).

Universal Source Coding

- Huffman is inherently 2-pass. We use 1st pass to estimate $p(x)$, but this might not be feasible (we might want to start compressing right away, which is the reason for stream codes).
- Huffman pays at most one extra bit per symbol, so to achieve entropy rate might need a long block length (where jointly encoding independent source symbols can be beneficial in terms of average number of bits per symbol $\frac{nH(X)+1}{n}$).
- In general, we want to be able to code down to the entropy without needing the source distribution (explicitly).
- I.e., universal if $p(x)$ remains unknown, but we can still code at $H(X)$.
- Lempel-Ziv coding is universal (as we will see).
- It is also the algorithm used in `gzip`, the widely used text compression algorithm (although `bzip2` often compresses a bit better, which uses the Burrows-Wheeler block-sorting text compression algorithm along with Huffman coding).

Arithmetic Coding I

- This is the method used by DjVU (adaptive image compression used for printed material, overtaken by PDF but probably certain PDF formats use this as well).
- Assume we are given a probabilistic model of the source, i.e., X_i i.i.d.

$$p(x_{1:n}) = \prod_{i=1}^{n} p(x_i),$$

or alternatively (X_i) would be a 1st order Markov model

$$p(x_{1:n}) = p(x_1) \prod_{i=2}^{n} p(x_i | x_{i-1})$$

- Higher order Markov models often used as well (as we'll see).

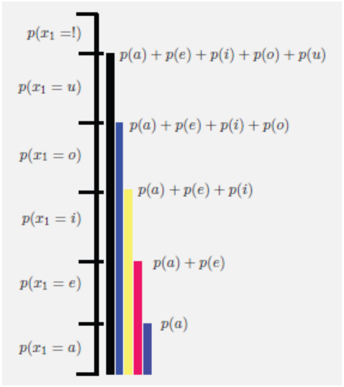
Arithmetic Coding II

- At each symbol, we use the conditional probability to provide the probability of the next symbol.
- Arithmetic coding can easily handle complex adaptive models of the source that produce context-dependent predictive distributions (so not necessarily. stationary); e.g., could use $p_t(x_t|x_1, \ldots, x_{t-1})$

Arithmetic Coding - Example I

- Let $\mathcal{X} = \{a, e, i, o, u, !\}$ so $|\mathcal{X}| = 6$.
- Source $X_1, X_2, \ldots$ need not be i.i.d.
- Assume that $p(x_n | x_1, x_2, \ldots, x_{n-1})$ is given to both encoder (sender, compressor) and receiver (decoder, uncompressor).
- Like in Shannon-Fano-Elias coding, we divide the unit interval up into segments of length according to the probabilities $p(X_1 = x)$ for $x = \{a, e, i, o, u, !\}$.
- Consider the following figure:

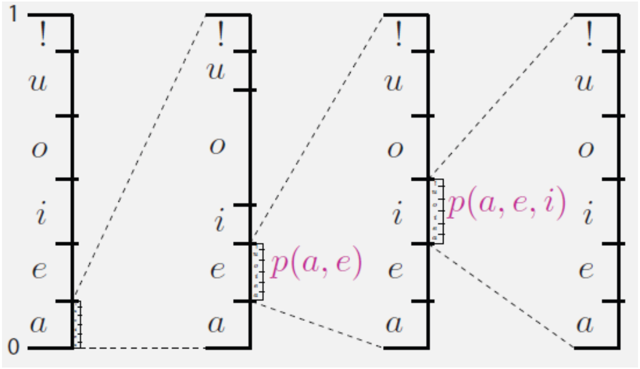
Arithmetic Coding - Example II



- Each subinterval may be further divided into segments of (relative) length $P(X_2 = x_2 | X_1 = x_1)$ or actual length $P(X_2 = x_2, X_1 = x_1)$.

Arithmetic Coding - Example III

- Relative lengths longer or shorter $P(X_1 = j) \rho P(X_2 = j | X_1 = k)$, where $\rho \in \{<, >, =\}$
- The following figure shows this, starting with $p(X_1 = a)$.



Arithmetic Coding - Example IV

- Length of interval for "ae" is $P(X_1 = a, X_2 = e) = P(X_1 = a)p(X_2 = e|X_1 = a)$.
- Intervals keep getting exponentially smaller with n larger.
- Key: at each stage, relative lengths of the intervals can change depending on history. At $t = 1$, relative interval fraction for "a" is $p(a)$, at $t = 2$, relative interval fraction for "a" is $p(a|X_1)$, which might change depending on X_1, and so on.
- This is different than Shannon-Fano-Elias coding which uses the same interval length at each step.
- Thus, if a symbol gets very probable, it uses a long relative interval (few bits), and if it gets very improbable, it uses short relative interval (more bits).

Arithmetic Coding

- How to code? Let i be the current source symbol number for X_i.
- We maintain a lower and an upper interval position.

$$L_n(i|x_1, x_2, \dots, x_{n-1}) = \sum_{j=1}^{i-1} p(x_n = j|x_1, x_2, \dots, x_{n-1})$$

$$U_n(i|x_1, x_2, \dots, x_{n-1}) = \sum_{j=1}^{i} p(x_n = j|x_1, x_2, \dots, x_{n-1})$$

- on arrival of nth input symbol, we divide the $(n-1)$st interval which is defined by L_n and U_n via the half-open interval $[L_n, U_n)$.

Interval Division

Example: initial interval is $[0, 1)$ and we divide it depending on the symbol we receive.

$$\begin{aligned}
a &\leftrightarrow [L_1(a); U_1(a)) = [0, p(X_1 = a))\\
e &\leftrightarrow [L_1(e); U_1(e)) = [p(X_1 = a), p(X_1 = a) + p(X_1 = e))\\
i &\leftrightarrow [L_1(i); U_1(i)) = [p(a) + p(e); p(a) + p(e) + p(i))\\
o &\leftrightarrow [L_1(o); U_1(o)) = [p(a) + p(e) + p(i); p(a) + p(e) + p(i) + p(o))\\
u &\leftrightarrow [L_1(u); U_1(u)) = \left[\sum_{x\in\{a,e,i,o\}} p(x), \sum_{x\in\{a,e,i,o,u\}} p(x)\right)\\
! &\leftrightarrow [L_1(u); U_1(u)) = \left[\sum_{x\in\{a,e,i,o,u\}} p(x), 1\right)
\end{aligned}$$

In general, we use an algorithm for the string $x_1, x_2, \ldots$ to derive the intervals $[\ell, u)$ at each time step where ℓ is the lower and u is the upper range.

Algorithm

Suppose we want to send N source symbols. Then we can follow the algorithm below.

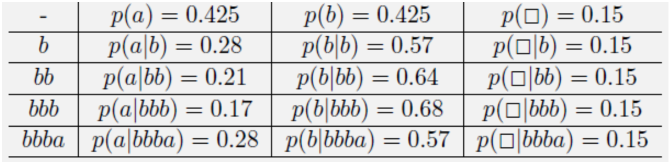
```
$\ell := 0;$
$u := \ell;$
$p := u - \ell;$
**for** $n = 1..N$ **do**
    $\{$First compute $U_i$ and $L_i$ for all $i \in \mathcal{X}\}$
    $u := \ell + pU_n(x_n|x_1, \dots, x_{n-1});$
    $\ell := \ell + pL_n(x_n|x_1, \dots, x_{n-1});$
    $p := u - \ell;$
**end for**
```

Encoding

- Once we have final interval, to encode we simply send any binary string that lives in the interval $[\ell; u)$ after running the algorithm.
- On the other hand, we can make the algorithm online, so that it starts writing out bits in the interval once they are known unambiguously.
- Analogous to Shannon-Fano-Elias coding, if the current interval is $[0.100101, 0.100110)$ then we can send the common prefix 1001 since that will not change.

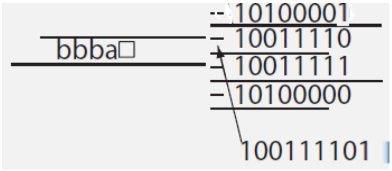
Example

Here is an example (Let $\square$ be a termination symbol):



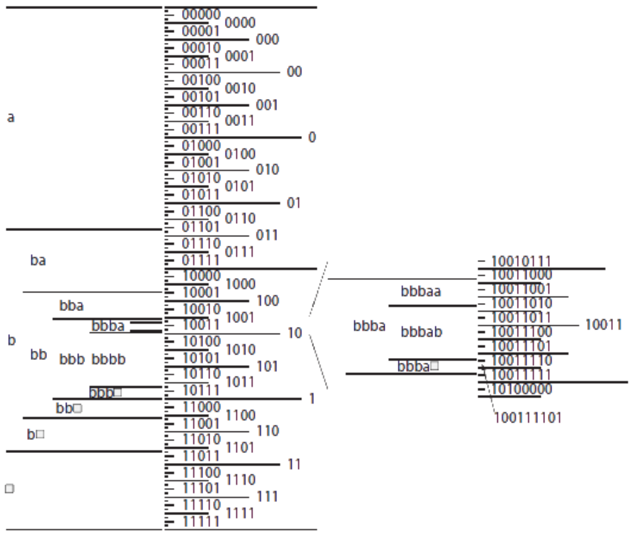
-	$p(a) = 0.425$	$p(b) = 0.425$	$p(\square) = 0.15$
b	$p(a\|b) = 0.28$	$p(b\|b) = 0.57$	$p(\square\|b) = 0.15$
bb	$p(a\|bb) = 0.21$	$p(b\|bb) = 0.64$	$p(\square\|bb) = 0.15$
bbb	$p(a\|bbb) = 0.17$	$p(b\|bbb) = 0.68$	$p(\square\|bbb) = 0.15$
$bbba$	$p(a\|bbba) = 0.28$	$p(b\|bbba) = 0.57$	$p(\square\|bbba) = 0.15$

- With these probabilities, we will consider encoding the string *bbba*□, and we'll get the final interval

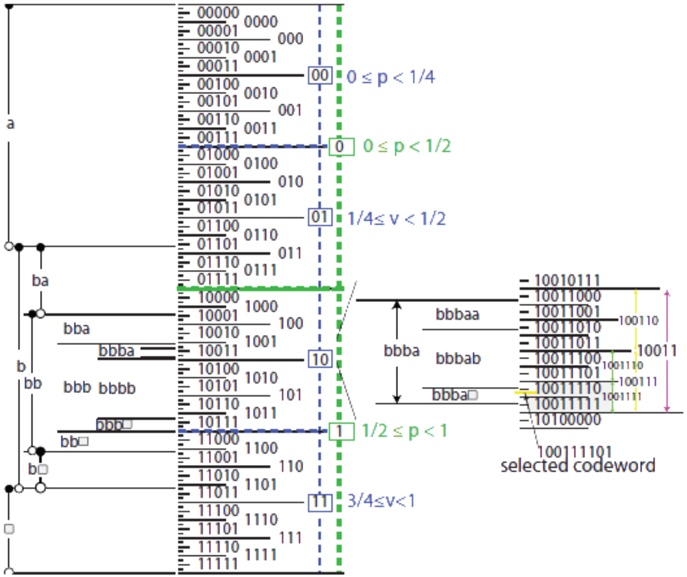


- I..e, the final code word will be 100111101
- Lets look at the entire picture

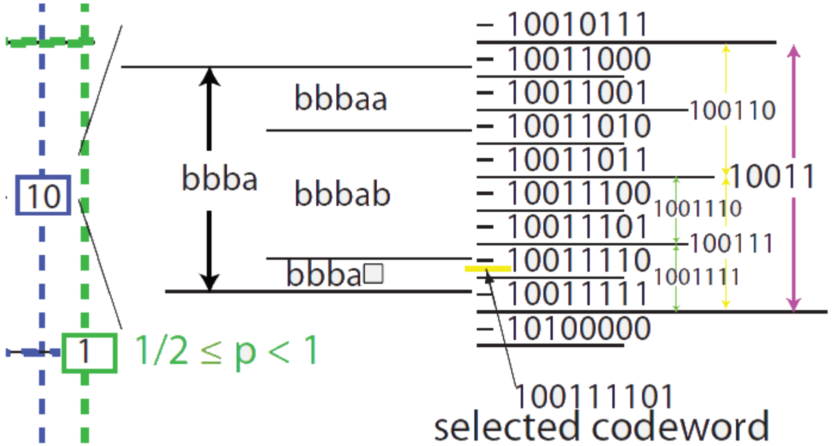
Coding Example from [2]



Coding Example from [2]



Coding Example from [2]



Q: Why can't we use 1001111?A: Because its interval is too large. Codeword 100111101's interval is entirely within *bbba*□'s interval, so we are prefix free.

Decoding

To decode a binary string, say $\alpha = 0.z_1 z_2 z_3 \dots$ we use the algorithm:

```
1: $\ell := 0$;
2: $u := \ell$;
3: $p := u - \ell$;
4: while special symbol $\square$ not received do
5:     find $i$ such that
           $$L_n(i|x_1,\dots,x_{n-1}) < \frac{\alpha-\ell}{u-\ell} < U_n(i|x_1,\dots,x_{n-1})$$
6:     $u := \ell + pU_n(i|x_1,\dots,x_{n-1})$;
7:     $\ell := \ell + pL_n(i|x_1,\dots,x_{n-1})$;
8: end while
```

Number of bits

- Problem is, a given number in the final interval $[L_n, U_n)$ could be arbitrarily long (e.g., periodic or irrational number). We only need to send enough to uniquely identify string.
- How do we choose the number of bits to send?
- Define
$$F_n(i|x_1, x_2, \dots, x_{n-1}) = \frac{1}{2}\left[L_n(i) + U_n(i)\right]$$
and $\lfloor F_n(i|x_1, x_2, \dots, x_{n-1}) \rfloor_\ell$ which is F_n truncated to ℓ bits.
- We could use $\ell(x_n|x_1, \dots, x_{n-1}) = \lceil \log 1/p(x_n|x_1, \dots, x_{n-1}) \rceil + 1$
- Instead, lets use the Shannon length of the entire code as
$$\ell(x_{1:n}) = \lceil \log 1/p(x_{1:n}) \rceil + 1.$$

Code length

- By the same arguments we made for the Shannon-Fano-Elias codes, this is a prefix code and thus is uniquely decodable, etc.
- Also, we have:

$$\begin{aligned} E\ell(x_{1:n}) &= \sum_{x_{1:n}} p(x_{1:n})\ell(x_{1:n}) \\ &= \sum_{x_{1:n}} p(x_{1:n}) \left(\lceil \log 1/p(x_{1:n}) \rceil + 1 \right) \\ &\le -\sum_{x_{1:n}} p(x_{1:n}) \log p(x_{1:n}) + 2 \\ &= H(X_{1:n}) + 2. \end{aligned}$$

- So the per symbol length $\le H(X_{1:n}) + 2/n \to H(\mathcal{X})$
- But this was not a block code.

Estimating $p(x_n|x_1, \ldots, x_{n-1})$

- We still have the problem that we need to estimate $p(x_n|x_1, \ldots, x_{n-1})$.
- We'd like to use adaptive models.
- One possibility is the Dirichlet model, having no independencies:

$$p(a|x_{1:n-1}) = \frac{N(a|x_{1:n-1}) + \alpha}{\sum_{a'} \left(N(a'|x_{1:n-1}) + \alpha\right)}$$

- Small α means more responsive
- Large α means more sluggish.
- How do we derive this? We can do so in a Bayesian setting.
- In general the problem of density estimation is a topic in and of itself.

Laplace's rule: Bayesian derivation

- For simplicity, assume binary alphabet, so $X = \{0, 1\}$.
- $N_0 = N(0|x_{1:n})$ and $N_1 = N(1|x_{1:n})$ counts, and $N = N0 + N1$.
- Assumptions: there exists p_0, p_1 and p_2 for 0, 1, and the termination symbol.
- Length of a given string has a geometric distribution, i.e.,

$$p(\ell) = (1 - p_\square)^\ell p_\square$$

so that $E\ell = 1/p_\square$ and $Var(\ell) = (1 - p_\square)p_\square^2$.
- Characters are drawn independently, so that

$$p(x_{1:n}|p_0; N) = p_0^{N_0} p_1^{N_1}$$

this is the "likelihood" function of the data.
- Uniform priors, i.e., $P(p_0) = 1$ for $p_0 \in [0, 1]$.

Laplace's rule: Bayesian derivation

- Then

$$P(p_0|x_{1:n}, N) = \frac{P(x_{1:n}|p_0, N)P(p_0)}{P(x_{1:n}|N)}$$
$$P(p_0|x_{1:n}, N) = \frac{p_0^{N_0} p_1^{N_1}}{P(x_{1:n}|N)}$$

- Using the Beta integral, we can get:

$$\begin{aligned} P(x_{1:n}|N) &= \int_0^1 P(x_{1:n}|p_0, N)\,dp_0 = \int_0^1 p_0^{N_0} p_1^{N_1} P(p_0)\,dp_0 \\ &= \frac{\Gamma(N_0+1)\Gamma(N_1+1)}{\Gamma(N_0+N_1+2)} = \frac{N_0!N_1!}{(N_0+N_1+2)!} \end{aligned}$$

Laplace's rule: Bayesian derivation

- To make a prediction, we want:

$$
\begin{aligned}
P(X_{n+1} = 0|x_{1:n}, N) &= \int_0^1 P(0|p0)P(p_0|x_{1:n}, N)\,dp0 \\
&= \int_0^1 p_0 \frac{p_0^{N_0} p_1^{N_1}}{P(x_{1:n}|N)}\,dp_0 \\
&= \int_0^1 \frac{p_0^{N_0+1} p_1^{N_1}}{P(x_{1:n}|N)}\,dp_0 \\
&= \left[\frac{(N_0+1)!N_1!}{(N_0+N_1+2)!}\right] / \left[\frac{N_0!N_1!}{(N_0+N_1+1)!}\right] \\
&= \frac{N_0+1}{N_0+N_1+2} \qquad \text{Laplace's rule}
\end{aligned}
$$

- Dirichlet's approach is $(N_0+\alpha)/(N_0+N_1+2\alpha)$ so the only difference is the fractional α.

Lempel-Ziv Compression

- The Lempel Ziv Algorithm is an algorithm for lossless data compression.
- It is not a single algorithm, but a whole family of algorithms, stemming from the two algorithms proposed by Jacob Ziv and Abraham Lempel in their landmark papers in 1977 [3] and 1978 [4].
- The Lempel Ziv algorithms belong to yet another category of lossless compression techniques known as *dictionary coders*.
- Lempel Ziv algorithms are widely used in compression utilities such as gzip, GIF image compression and the V.42 modem standard.

Lempel-Ziv Family

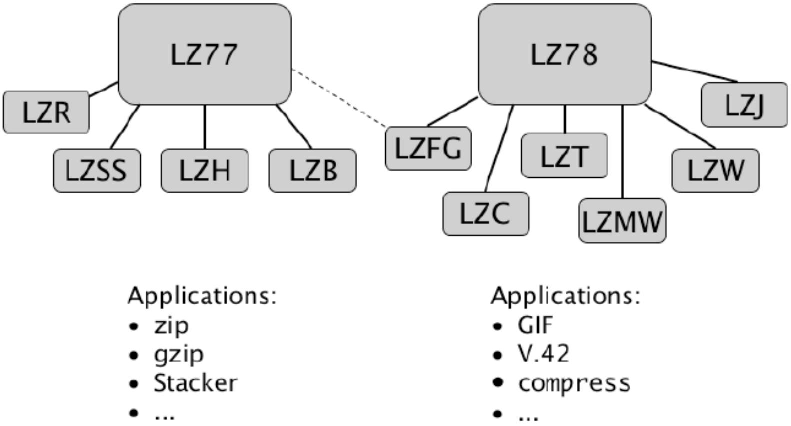


Figure: Lempel-Ziv family

Dictionary Coding

- Dictionary coding techniques rely upon the observation that there are correlations between parts of data (recurring patterns).
- The basic idea is to replace those repetitions by (shorter) references to a ”dictionary” containing the original.

Static Dictionary

- The simplest forms of dictionary coding use a static dictionary.
- Such a dictionary may contain frequently occurring phrases of arbitrary length, digrams (two-letter combinations) or n-grams.
- This kind of dictionary can easily be built upon an existing coding such as ASCII by using previously unused codewords or extending the length of the codewords to accommodate the dictionary entries.
- A static dictionary achieves little compression for most data sources. The dictionary can be completely unsuitable for compressing particular data, thus resulting in an increased message size (caused by the longer codewords needed for the dictionary).

Semi-Adaptive Dictionary

- The aforementioned problems can be avoided by using a semi-adaptive encoder.
- This class of encoders creates a dictionary custom-tailored for the message to be compressed. Unfortunately, this makes it necessary to transmit/store the dictionary together with the data.
- This method usually requires two passes over the data, one to build the dictionary and another one to compress the data.
- A question arising with the use of this technique is how to create an optimal dictionary for a given message. It has been shown that this problem is NP-complete (vertex cover problem).
- Fortunately, there exist heuristic algorithms for finding near-optimal dictionaries.

Adaptive Dictionary

- The Lempel Ziv algorithms belong to this third category of dictionary coders.
- The dictionary is being built in a single pass, while at the same time also encoding the data.
- As we will see, it is not necessary to explicitly transmit/store the dictionary because the decoder can build up the dictionary in the same way as the encoder while decompressing the data.

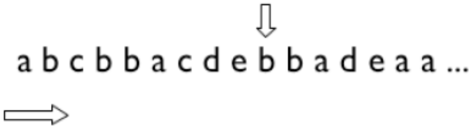
Principle I

- So far we have always been talking about the dictionary as if it were some kind of data structure that is being filled with entries when the dictionary is being built.
- It turns out that it is not necessary to use an explicit dictionary.

Example 1

The string "`abcbbacdebbadeaa`" is to be encoded. The algorithm is working from left to right and has already encoded the left part (the string $E =$ "`abcbbacde`"). The string $S =$ "`bbadeaa`" is the the data yet to be encoded.

Principle II



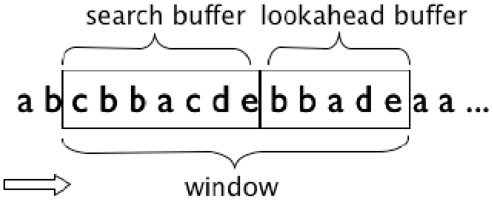
- First, the algorithm searches for the longest string in the encoded data *E* matching a prefix of S. In this particular case, the longest match is the string "bba" starting at the third position (counting from zero).
- Therefore it is possible to code the first two characters of S, "bba", as a reference to the third and fourth character of the whole string.
- References are encoded as a fixed-length codeword consisting of three elements: position, length and first non-matching symbol. In our case, the codeword would be 33d. In it, four characters have been coded with just one codeword.

Principle III

- When the matches get longer, those coded references will consume significantly fewer space than, for example, coding everything in ASCII.
- Probably you have already spotted the weakness of the outlined algorithm.
- What happens if the input is very long and therefore references (and lengths) become very large numbers?
- Well, the previous example was not yet the actual LZ77 algorithm. The LZ77 algorithm employs a principle called *sliding-window*: It looks at the data through a window of fixed-size, anything outside this window can neither be referenced nor encoded.
- As more data is being encoded, the window slides along, removing the oldest encoded data from the view and adding new unencoded data to it.

Principle IV

- The window is divided into a *search buffer* containing the data that has already been processed, and a *lookahead buffer* containing the data yet to be encoded.



The Algorithm

The actual LZ77 algorithm

```
1: while lookAheadBuffer not empty do
2:     get a reference(position, length) to longest match
3:     if length > 0 then
4:         output(position, length, nextsymbol)
5:         shift the window length + 1 positions along
6:     else
7:         output(0,0, first symbol in the lookAheadBuffer)
8:         shift the window 1 position along
9:     end if
10: end while
```

- The algorithms starts out with the lookahead buffer filled with the first symbols of the data to be encoded, and the search buffer filled with a predefined symbol of the input alphabet (zeros, for example).

Example I

- The following is the example given in [3]:

$$\begin{aligned} S &= 001010210210212021021200\ldots \text{ (input string)} \\ L_s &= 9 \text{ (length of look ahead buffer)} \\ n &= 18 \text{ (window size)} \end{aligned}$$

- The search buffer is loaded with zeros and the lookahead buffer is loaded with the first 9 characters of S.
- The algorithm searches for the longest match in the search buffer. Since it is filled with zeros, any substring of length 2 can be used.
- In the example, the substring starting at the last position (8, if counting from 0) of the search buffer is being used. Note that the match extends into the lookahead buffer!

Example II

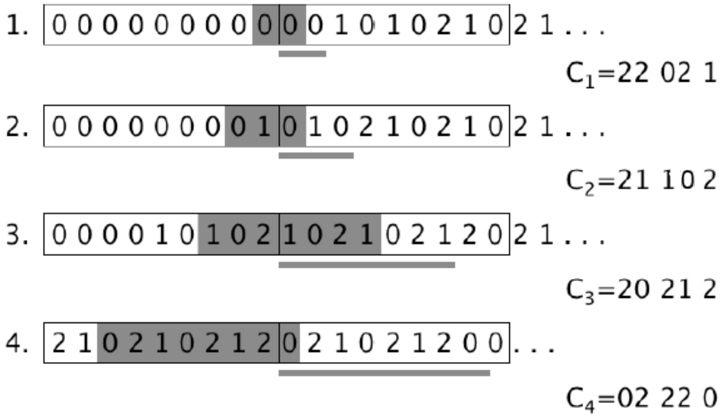


Figure: Encoding

Example III

- Matches are encoded as codewords consisting of the position of the match, the length of the match and the first symbol following the prefix of the lookahead buffer that a match has been found for. Therefore, codewords need to have the length (D size of alphabet)

$$L_c = \log_D(n - L_s) + \log_D(L_s) + 1$$

- In the example the length is $L_s = \log_3(9) + \log_3(9) + 1 = 5$.
- The first match is encoded as the codeword $C_1 = 22021$. 22 is the position of the match in radix-3 representation ($8_{10} = 22_3$). The two following positions represent the length of the match ($2_{10} = 2_3$, 02 because 2 positions are reserved for it according to the formula for L_c). The last element is the first symbol following the match, which in our case is 1.

Example IV

- The algorithm now shifts the window 3 positions to the right, resulting in the situation depicted in 2. This time, the algorithm finds the longest match at the 7th position of the search buffer, the length of the match is 3 because once again it is possible to extend the match into the lookahead buffer. The resulting codeword is $C_2 = 21102$. Steps 3 and 4 result in the codewords $C_3 = 20212$ and $C_4 = 02220$.
- The decoder also starts out with a search buffer filled with zeros and reverses the process as shown in Figure 3

Example V

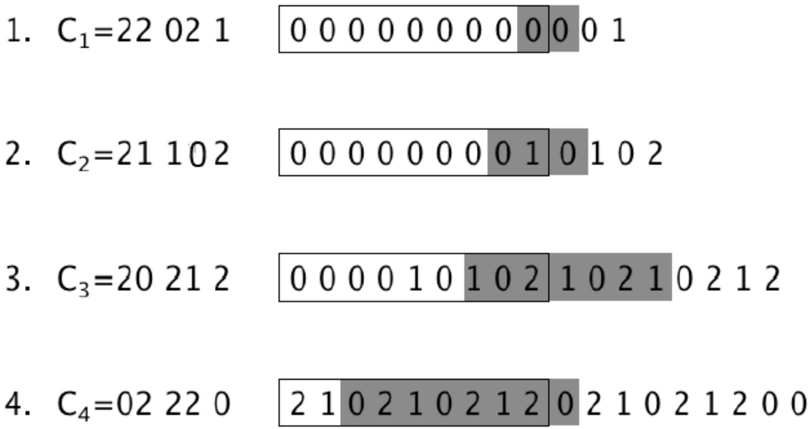


Figure: Decoding

Improvements

LZR

The LZR modification allows pointers to reference anything that has already been encoded without being limited by the length of the search buffer (window size exceeds size of expected input). Since the position and length values can be arbitrarily large, a variable-length representation is being used positions and lengths of the matches.

Improvements I
LZSS

- The mandatory inclusion of the next non-matching symbol into each codeword will lead to situations in which the symbol is being explicitly coded despite the possibility of it being part of the next match. Example: In "abbca|caabb", the first match is a reference to "ca" (with the first non-matching symbol being "a") and the next match then is "bb" while it could have been "abb" if there were no requirement to explicitly code the first non-matching symbol.
- The popular modification by Storer and Szymanski (1982) removes this requirement. Their algorithm uses fixed-length codewords consisting of offset (into the search buffer) and length (of the match) to denote references. Only symbols for which no match can be found or where the references would take up more space than the codes for the symbols are still explicitly coded.

Improvements II
LZSS

```
1: while *lookAheadBuffer* not empty do
2:    get a pointer$(position, match)$ to the longest match
3:    if $length > MinimumMatchLength$ then
4:       output$(PointerFlag, position, length)$
5:       shift the window *length* characters along
6:    else
7:       output$(SymbolFlag$, first symbol of $lookAheadBuffer)$;
8:       shift the window 1 character along;
9:    end if
10: end while
```

Improvements

- **LZB** uses an elaborate scheme for encoding the references and lengths with varying sizes.
- The **LZH** implementation employs Huffman coding to compress the pointers.

Principle

- The LZ78 is a dictionary-based compression algorithm that maintains an explicit dictionary.
- The codewords output by the algorithm consist of two elements:
    - an index referring to the longest matching dictionary entry and
    - the first non-matching symbol.
- In addition to outputting the codeword for storage/transmission, the algorithm also adds the index and symbol pair to the dictionary.
- When a symbol that not yet in the dictionary is encountered, the codeword has the index value 0 and it is added to the dictionary as well.
- With this method, the algorithm gradually builds up a dictionary.

Algorithm I

```
$w := NIL$;
**while** there is input **do**
    $K :=$ next symbol from input;
    **if** $wK$ exists in the dictionary **then**
        $w := wK$;
    **else**
        output (index($w$), $K$);
        add $wK$ to the dictionary;
        $w := NIL$;
    **end if**
**end while**
```

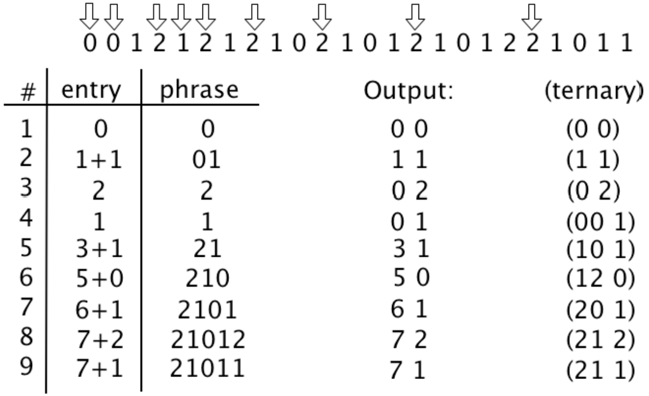
- Note that this simplified pseudo-code version of the algorithm does not prevent the dictionary from growing forever.

Algorithm II

- There are various solutions to limit dictionary size, the easiest being to stop adding entries and continue like a static dictionary coder or to throw the dictionary away and start from scratch after a certain number of entries has been reached.
- Those and more sophisticated approaches will be presented in section Improvements.

Example

The string $S = 001212121021012101221011$ is to be encoded. Figure 4 shows the encoding process.



0 0 1 2 1 2 1 2 1 0 2 1 0 1 2 1 0 1 2 2 1 0 1 1

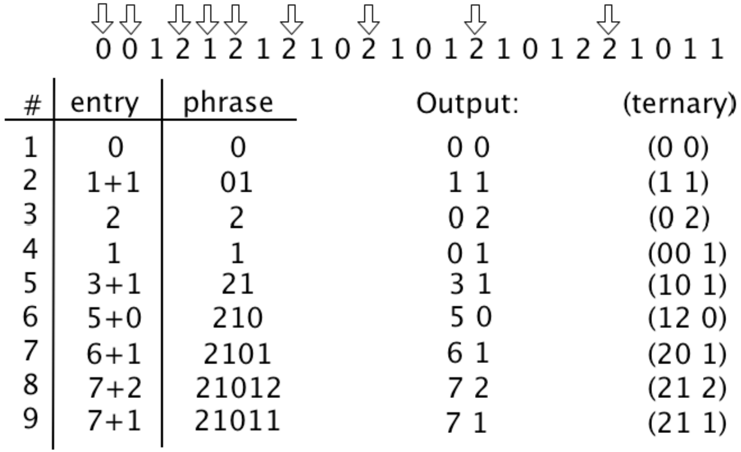
#	entry	phrase	Output:	(ternary)
1	0	0	0 0	(0 0)
2	1+1	01	1 1	(1 1)
3	2	2	0 2	(0 2)
4	1	1	0 1	(00 1)
5	3+1	21	3 1	(10 1)
6	5+0	210	5 0	(12 0)
7	6+1	2101	6 1	(20 1)
8	7+2	21012	7 2	(21 2)
9	7+1	21011	7 1	(21 1)

Figure: Encoding

Example - continued I

- In the first step, 0 is encountered and added to the dictionary. The output is 00 because is no match (index 0) and the first non-matching character is 0. The encoder then proceeds to the second position, encountering 0, which is already dictionary. The following 1 is not yet in the dictionary, so the encoder adds the string 01 to the dictionary (a reference to the first entry plus the symbol 1) and outputs this pair. The next steps follow the same scheme until the end of the input is reached.
- The decoding process is shown in figure 5. The decoder receives the reference 0 0, with the index 0 indicating that a previously unknown symbol (0) needs to be added to the dictionary and to the uncompressed data. The next codeword is 1 1 resulting in the entry 01 (a reference to entry 1 plus the symbol 1) being added to the dictionary and the string 01 appended to the uncompressed data. The decoder continues this way until all codewords have been decoded.

Example - continued II



```
⇩ ⇩   ⇩ ⇩ ⇩   ⇩     ⇩       ⇩         ⇩
0 0 1 2 1 2 1 2 1 0 2 1 0 1 2 1 0 1 2 2 1 0 1 1
```

#	entry	phrase	Output:	(ternary)
1	0	0	0 0	(0 0)
2	1+1	01	1 1	(1 1)
3	2	2	0 2	(0 2)
4	1	1	0 1	(00 1)
5	3+1	21	3 1	(10 1)
6	5+0	210	5 0	(12 0)
7	6+1	2101	6 1	(20 1)
8	7+2	21012	7 2	(21 2)
9	7+1	21011	7 1	(21 1)

Figure: Decoding

Improvements

- LZ78 has several weaknesses. First of all, the dictionary grows without bounds.
- Various methods have been introduced to prevent this, the easiest being to become either static once the dictionary is full or to throw away the dictionary and start creating a new one from scratch.
- There are also more sophisticated techniques to prevent the dictionary from growing unreasonably large, some of these will be presented in the sequel.
- The dictionary building process of LZ78 yields long phrases only fairly late in the dictionary building process and only includes few substrings of the processed data into the dictionary.
- The inclusion of an explicitly coded symbol into every match may cause the next match to be worse than it could be if it were allowed to include this symbol.

Improvements I

LZW

- This improved version of the original LZ78 algorithm is perhaps the most famous modification and is sometimes even mistakenly referred to as the Lempel Ziv algorithm.
- Published by Terry Welch in 1984, it basically applies the LZSS principle of not explicitly transmitting the next nonmatching symbol to the LZ78 algorithm.
- The only remaining output of this improved algorithm are fixed-length references to the dictionary (indexes).
- Of course, we can't just remove all symbols from the output and add nothing elsewhere. Therefore the dictionary has to be initialized with all the symbols of the input alphabet and this initial dictionary needs to be made known to the decoder.

Improvements II

LZW

In the original proposal, the pointer size is chosen to be 12 bit, allowing for up to 4096 dictionary entries. Once this limit has been reached, the dictionary becomes static.

Improvements I
LZC

- LZC is the variant of the LZW algorithm that is used in the once popular UNIX `compress` program. It compresses according to the LZW principle but returns to variable-length pointers like the original LZ78 algorithm.
- The maximum index length can be set by the user of the compress program taking into account to the memory available (from 9 to 16 bits). It first starts with 9-bit indexes for the first 512 dictionary entries, then continues with 10-bit codes and so on until the user-specified limit has been reached. Finally, the algorithm becomes a static encoder, regularly checking the compression ratio.
- When it detects a decrease, it throws away the dictionary and starts building up a new one from scratch.

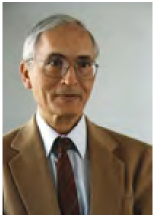
Other Improvements I

- **LZT** is another variation on the LZW theme, the algorithm is almost like the LZC variant, the only difference being that it makes room for new phrases to be added to the dictionary by removing the least recently used entry (LRU replacement).
- **LZMS** creates new dictionary entries not by appending the first non-matching character, but by concatenating the last two phrases that have been encoded. This leads to the quick building of rather long entries, but in turn leaves out many of the prefixes of those long entries.
- The dictionary used by **LZJ** contains every unique string of the input up to a certain length, coded by a fixed-length index. Once the dictionary is full, all strings that have only been used once are removed. This is continued until the dictionary becomes static.

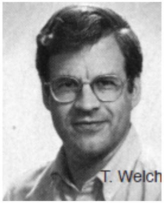
Other Improvements II

- **LZFG** uses the dictionary building technique from the original LZ78 algorithm, but stores the elements in a trie data structure. In addition, a sliding window like LZ77's is used to remove the oldest entries from the dictionary.

Abraham Lempel (1936 -) is an Israeli computer scientist and one of the fathers of the LZ family of lossless data compression algorithms. Prizes: 2007 IEEE Richard W. Hamming Medal,.



Jacob Ziv (1931 -) is an Israeli computer scientist who, along with Abraham Lempel, developed the LZ family of lossless data compression algorithms. Prizes: 1995 the IEEE Richard W. Hamming Medal, 1997 Claude E. Shannon Award from the IEEE Information Theory Society.



Terry Welch (? - 1985) inventor of LZW algorithm. Senior manager for Digital Equipment Corporation. BS, MS, and PhD degrees were received from MIT in electrical engineering, senior member of IEEE.

Preparing Results I

- Assume we have a stationary and ergodic process defined for time from $-\infty$ to ∞ and that both the encoder and decoder have access to $\ldots, X_{-2}, X_{-1}$, the infinite past of the sequence.
- To encode $X_0, X_1, \ldots, X_{n-1}$ (a block of length n), we find the last time we have seen these n symbols in the past.
- Let

$$\begin{gathered} R_n\left(X_0, X_1, \ldots, X_{n-1}\right) = \\ \max\left\{j < 0 : \left(X_{-j}, X_{-j+1}, \ldots, X_{-j+n-1}\right) = \left(X_0, X_1, \ldots, X_{n-1}\right)\right\}. \end{gathered} \tag{1}$$

- Then to represent $X_0, X_1, \ldots, X_{n-1}$, we need only to send R_n to the receiver, who can then look back R_n into the past and recover $X_0, X_1, \ldots, X_{n-1}$.

Preparing Results II

- Thus, the cost of the encoding is the cost to represent R_n. The asymptotic optimality of the algorithm means

$$\frac{1}{n} E \log R_n \to H\left(\mathcal{X}\right).$$

Lemma 2

There exists a prefix-free code for the integers such that the length of the codeword for integer k is $\log k + 2 \log \log k + O(1)$.

Proof of Lemma 2

Proof.

If we know that $k \leq m$, codelength $\leq \log m$ bits. Otherwise we first represent $\lceil \log k \rceil$ in unary and k in binary

$$C_1(k) = \underbrace{00\ldots0}_{\lceil \log k \rceil}1\,\underbrace{xx\ldots x}_{k \text{ in binary}}.$$

The length of $C_1(k)$ is $2\lceil \log k \rceil + 1 \leq 2\log k + 3$. If we use $C_1(k)$ to represent $\log k$, the length is less than $\log k + 2\log\log k + 4$. □

Kac's Lemma I

Lemma 3 (Kac)

Let $\ldots, U_{-2}, U_{-1}, U_0, U_1, \ldots$ be a stationary ergodic process on a countable alphabet. For any u such that $p(u) > 0$ and for $i = 1, 2, \ldots,$ let

$$Q_u(i) = P\left\{U_{-i} = u : U_j \neq u, \ -i < j < 0 | U_0 = u\right\} \tag{2}$$

Then

$$E\left(R_1(U) | X_0 = u\right) = \sum_i iQ_u(i) = \frac{1}{p(u)}. \tag{3}$$

Thus, the conditional expected waiting time to see the symbol u again, looking backward from zero, is $1/p(u)$.

Kac's Lemma II

Notes: 1. The expected recurence time is

$$E\left(R_1(U)\right) = \sum p(u)\frac{1}{p(u)} = m = |\mathcal{X}|.$$

2. $Q_u(i)$ is the conditional probability that the most recent previous occurence of the symbol u is i, given that $U_0 = u$.

Proof of Kac Lemma. . . .

Let $U_0 = u$. For $j = 1, 2, \ldots$ and $k = 0, 1, 2, \ldots$ we define

$$A_{jk} = \{U_{-j} = u, U_{-\ell} \neq u, -j < \ell < k, U_k = u\}.$$

We see that $A_{jk} \cap A_{vw} = \emptyset$ for $(j,k) \neq (u,v)$ and $P\left(\bigcup_{j,k} A_{jk}\right) = 1$.

$$\begin{aligned} 1 &= P\left(\bigcup_{j,k} A_{jk}\right) = \sum_{j=1}^{\infty}\sum_{k=0}^{\infty} P\left(A_{jk}\right) \\ &= \sum_{j=1}^{\infty}\sum_{k=0}^{\infty} P\left(U_k = u\right) P\left(U_{-j} = u, U_\ell \neq u, -j < \ell < k \middle| U_k = u\right) \\ &= \sum_{j=1}^{\infty}\sum_{k=0}^{\infty} P\left(U_k = u\right) Q_u(j+k) =: S \qquad // \text{ definition of } Q_u \end{aligned}$$

. . .

. . . proof of Kac's Lemma (continued).

But

$$\begin{aligned} S &= \sum_{j=1}^{\infty}\sum_{k=0}^{\infty} P\left(U_0 = u\right) Q_u(j+k) \qquad //\text{stationarity} \\ &= P\left(U_0 = u\right) \sum_{j=1}^{\infty}\sum_{k=0}^{\infty} Q_u(j+k) \\ &= P\left(U_0 = u\right) \underbrace{\sum_{i=1}^{\infty} iQ_u(i)}_{E(R_1(U)|X_0=u)} . \end{aligned}$$

The last equality follows from the fact that $\exists(j,k)$ such that $j+k=i$. The conclusion follows from

$$1 = S = P\left(U_0 = u\right) E\left(R_1(U)|X_0 = u\right).$$

□

Preparing Results

Corollary 4

Let $\ldots, U_{-2}, U_{-1}, U_0, U_1, \ldots$ *be a stationary ergodic process. Then*

$$E\left[R_n\left(X_0, X_1, \ldots, X_{n-1}\right) \mid \left(X_0, X_1, \ldots, X_{n-1}\right) = x_0^{-1}\right] = \frac{1}{p\left(x_0^{n-1}\right)}.$$

Proof.

Define the process $U_i = \left(X_i, X_{i+1}, \ldots, X_{i+n-1}\right)$. It is stationary and ergodic, and thus by Kac's lemma the average recurence time for U conditioned by $U_0 = u$ is $1/p(u)$. Translating this to the X process proves the corollary. □

Main Theorem

Theorem 5

Let $L_n\left(X_0^{n-1}\right) = \log R_n + 2\log\log R_n + O(1)$ *be the description length for* X_0^{n-1} *in the simple algorithm described above. Then*

$$\frac{1}{n} E L_n\left(X_0^{n-1}\right) \to H\left(\mathcal{X}\right) \tag{4}$$

as $n \to \infty$, *where* $H\left(\mathcal{X}\right)$ *is the entropy rate of the process* $\{X_i\}$.

Proof of main theorem. . . .

We look for upper and lower bounds for EL_n. But, $EL_n \geq nH$ for any prefix code. We first show that

$$\overline{\lim} \frac{1}{n} E \log R_n \leq H \tag{5}$$

To prove the bound for $E \log R_n$, we expand the expectation by conditioning on X_0^{n-1} and applying Jensen's inequality.

$$\frac{1}{n} E \log R_n = \frac{1}{n} \sum_{x_0^{n-1}} p\left(x_0^{n-1}\right) E\left[\log R_n\left(X_0^{n-1} | X_0^{n-1} = x_0^{n-1}\right)\right] \tag{6}$$

$$\leq \frac{1}{n} \sum_{x_0^{n-1}} p\left(x_0^{n-1}\right) \log E\left[R_n\left(X_0^{n-1} | X_0^{n-1} = x_0^{n-1}\right)\right] \tag{7}$$

$$= \frac{1}{n} \sum_{x_0^{n-1}} p\left(x_0^{n-1}\right) \log \frac{1}{p\left(x_0^{n-1}\right)} = \frac{1}{n} H\left(x_0^{n-1}\right) \searrow H\left(\mathcal{X}\right) \tag{8}$$

. . .

. . . Proof of main theorem.

The second term in the expression for L_n is $\log\log R_n$, and we wish to show that

$$\frac{1}{n}E\left[\log\log R_n\left(x_0^{n-1}\right)\right] \to 0.$$

Again, we use Jensen's inequality,

$$\frac{1}{n}E\log\log R_n \leq \frac{1}{n}\log E\left[\log R_n\left(x_0^{n-1}\right)\right] \leq \frac{1}{n}\log H\left(x_0^{n-1}\right)$$

where the Ist inequality follows from (8). $\forall\varepsilon > 0$, for n large enough, $H\left(x_0^{n-1}\right) \leq n(H+\varepsilon)$, and therefore

$$\frac{1}{n}\log\log R_n < \frac{1}{n}\log n + \frac{1}{n}\log\left(H+\varepsilon\right) \to 0.$$

$\square$

Comments

- This scheme is not practical and not realistic
- if entropy rate is $\frac{1}{2}$ and string length is 200 bits, one would have to look an average of $2^{100} \approx 10^{30}$ bits on the past to find a match, not feasible
- Proof of optimality for finite window is in [5]

References I

Thomas M. Cover, Joy A. Thomas, *Elements of Information Theory*, 2nd edition, Wiley, 2006.

David J.C. MacKay, *Information Theory, Inference, and Learning Algorithms*, Cambridge University Press, 2003.

Robert M. Gray, *Entropy and Information Theory*, Springer, 2009

David Salomon, A Concise Introduction to Data Compression, Springer, 2008

Khalid Sayood, *Introduction to Data Compression*, third edition, Academic Press, San Diego, CA, 2006

D. A. Huffman, A method for the construction of minimum redundancy codes, *Proc. IRE*, **40**: 1098–1101,1952

References II

- Terry A. Welch, A technique for high-performance data compression, *Computer*, **17**: 8-19, 1984
- J. Ziv, A. Lempel, A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* **23:** 337–343, 1977.
- J. Ziv, A. Lempel, A., Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory **24:** 530–536, 1978.
- A. D. Wyner, J. Ziv, The sliding window Lempel-Ziv algorithm is asymptoticaly optimal, Proc. IEEE, 82: 872-877, 1994