Faculty of Electrical Engineering and Computing, Department of Applied Computing

M. Brčić, D. Krleža

Advanced Algorithms and Data Structures

Lecture script

PUBLISHED BY FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING, DEPARTMENT OF APPLIED COMPUTING

First printing, November 2022

Contents

Introduction

The page is intentionally left blank

7 Compression algorithms

Compression is an operation that transforms input information to a more compact output from which satisfactory reconstruction of input information is possible by corresponding "reverse" operation - decompression. Algorithms that perform compression are called encoders and ones performing decompression are called decoders. In the upper definition "satisfactory" comprises two different kinds of compression:

- lossless compression which focuses on eliminating statistical redundancy and perfect reconstruction. Examples are universal compression algorithms behind formats gzip and ZIP.
- lossy compression focuses on semantic information relative to the purpose and target use. Loss of irrelevant information is accepted. Examples are multimedia compression algorithms behind format JPEG (image), MP3 (audio), MPEG4 (video).

Theoretically, two data modeling approaches are the basis of all compression methods:

- statistical information theory (SIT) by Claude Shannon which is based on entropy and ensembles [Shannon, 1948]
- algorithmic information theory (AIT) by Andrey Kolmogorov, Ray Solomonoff and Gregory Chaitin which is based on complexity, Turing machines, Solomonoff universal prior probability, etc. [Kolmogorov, 1963, Solomonoff, 1964a,b, Chaitin, 1969]

SIT-based methods dominate applied compression - the reason is greater simplicity and smaller computational complexity. Shannon entropy is commonly used to quantify randomness, even though advances in AIT have found better and more general characterization of randomness. SIT-based methods have weaknesses, such as with recursively generated data which is misidentified as random and hence incompressible by SIT-based compression approaches [1]. Moreover, entropy is good at estimating the compression for statistical compressors, but when combined with contextual compressors it fails. For these reasons, there are many manual transformation heuristics created to bridge the gap between statistical and algorithmic compressibility.

There is a profound connection between compression and learning through the principle of parsimony (Occam's razor) and

[1] Hector Zenil. Towards demystifying shannon entropy, lossless compression and approaches to statistical machine learning. *Proceedings*, 47(1), 2020. ISSN 2504-3900. DOI: 10.3390/proceedings2020047024. URL https://www.mdpi.com/2504-3900/47/1/24

Vapnik–Chervonenkis dimension, PAC learnability[A. Blumer and Warmuth, 1990, Littlestone and Warmuth, 1986, Moran and Yehudayoff, 2016]. While the compression focuses only on the description of input data, learning is focused on inductive inference - that is, use on the new data.

Herein, we shall deal with well-known and widely used lossless compression algorithms: Huffman coding and Lempel-Ziv family of algorithms. In the future, we plan to add a bit about lossy compression algorithms as well as an example of the AIT-based compression approach.

7.1 Statistical vs Algorithmic Information theory

Entropy can be seen as the number of binary decisions necessary to uniquely describe input data. Kolmogorov complexity is, on the other hand, the size of the minimal program that would generate the input data. The latter is the basis of program synthesis in data compression and causality discovery. And, yes - it is incomputable. That is the reason why it is not used often, for now. Entropy is a good enough metric that is used most often in applications.

Definition 7.1 (Entropy)**.** Entropy associated with set of n independent events $S = \{x_1, ..., x_n\}$ and its corresponding set of occurrence probabilities $P = \{p_1, ..., p_n\}$ is the function
$H(S) = -\sum_{i=1}^{n} p_i log_2 p_i$.

Definition 7.2 (Kolmogorov complexity)**.** Kolmogorov complexity of some sequence of characters x is the length of the shortest program p which, when ran on a reference universal machine U generates x as an output, i.e.:
$K_U(x) := min_p\{l(p) : U(p) = x\}$

As already stated, Kolmogorov complexity is incomputable. However, it is computably approximable from above (via a sequence of decreasing upper bounds).

When we transmit a sequence of bits, we need to separate the individual symbols to decode the incoming sequence. Separation can be achieved in several obvious ways, for example:

- fixed-length code - an example is ASCII coding where 8bit codes encode each character. Hence, they are easily separable in the incoming sequence of bits.
- fixed-separator code - example of this would be something similar to comma-separated-values, where an agreed-upon sequence of bits would act as separators between other symbols.

We shall see that both approaches are wasteful as the first one uses too many bits for encoding specific data, and the other one adds an overhead of encoding separators. We can achieve the separation in a more space-efficient manner. All the approaches mentioned above have the property of unique decodability.

Definition 7.3 (Unique decodability)**.** A code is uniquely decodable if and only if there is only one way of splitting the sequence of codes into separate codewords.

There is a possibility of different uniquely decodable codes. There are options beyond separators and fixed-length codes. For example, specially crafted variable-length codewords could be separated via parsing. The most general types of codes might need a lookahead to decide on separation. However, we prefer not to invest too much computational work into parsing. There is a special family of codes that do not need a lookahead. Such codes can decide on separation based only on the characters read so far. These codes are *prefix-free codes*, also known as self-delimiting codes and simply as *prefix codes*.

Definition 7.4 (Prefix-free property)**.** Code is prefix-free if no codeword is a prefix of some other codeword.

Prefix-free codes are uniquely decodable. Kraft's inequality is a great tool to stop the futile search when looking for optimal prefix-free code. The inequality characterizes the sets of codeword lengths that are possible in uniquely decodable codes.

Theorem 7.1 (Kraft's inequality)**.** There exists prefix-free binary code with codewords $C = \{c_1, ..., c_n\}$ and corresponding lengths $\{l_1, ..., l_n\}$ iff

$\sum_{i=1}^{n} 2^{-l_i} \leq 1$.

Proof. ($\Rightarrow$ direction) Let us assume we have a valid prefix-free code. Let $l_{max} = max\{l_1, ..., l_n\}$ and let us construct complete binary decision tree of depth $l_{max} + 1$ which has $2^{l_{max}}$ nodes at the lowest level. Then, prefix-free codeword of length l_i would correspond to some node on level $l_i + 1$. And since the code is prefix-free, the whole subtree of child codes corresponding to codewords are forbidden, which means that from the last level for each code of length l_i we lose $2^{l_{max}} - l_i$ leaves. When we sum up all such lost leaves, we must not lose more than possible:

$\sum_{i=1}^{n} 2^{l_{max}-l_i} \leq 2^{l_{max}}$, from which it follows: $\sum_{i=1}^{n} 2^{-l_i} \leq 1$. If this would not be the case, there woud exist at least one codeword in C that is prefix to some other codeword in C.

($\Leftarrow$ direction, proof by contradiction) Let us assume Kraft's inequality and try to run into the impossibility of constructing a prefix-free code. For simplicity, assume lengths of codewords are sorted in nondecreasing order, $l_1 \leq l_2 \leq ... \leq l_n = l_{max}$. Start by work on complete binary tree of depth $l_{max} + 1$ with $2^{l_{max}}$ leafs. Start by l_1 by picking some node A at level $l_1 + 1$ as the corresponding code. Since this must be prefix-free code, all nodes in the subtree of A must be forbidden for further assignments. Continue with this procedure through all lengths. Let us assume at some step $j < n$ that we cannot assign any node at depth $l_j + 1$. Due to Kraft's inequality, we know there are some unused leaves. This means (due to the prefix-free property up to that moment in construction) that there exists a path from the root to the leaf that is available for assignments, which also

includes some node on level $l_j + 1$. That contradicts the assumed impossibility. □

Below we give another theorem that will serve us well in argumentation for block-coding.

Theorem 7.2 (Shannon's fundamental discrete noiseless coding theorem)**.** For a source S with entropy $H(S)$, a prefix-free coding of sequences of k symbols from S is possible such that average code-word length L_k per element in S satisfies the following:
$H(S) \leq \frac{L_k}{k} < H(S) + \frac{1}{k}$.

The above theorem claims that with the increase of block size ("super-symbols"), the average codeword length per input symbol approaches the entropy of the source. That, in effect, leads to the "fractional" coding of the source. However, this also leads to the growth of the complexity of the code and the size of its coding table.

To show the difference between SIT and AIT based approaches, we shall look at the following idealized, somewhat familiar, stream of information:

3.14159265358979323846264338327950288419716939937 51...

(assume more than 1,000,000 of decimals)

This stream passes all the statistical tests for randomness and is hence incompressible from the aspect of statistical information theory. Practically, the compressed length is unchanged (except for the fact that only 11 symbols need to be coded).

Using AIT, it would be possible, in principle, to find a program such as the following (based on Ramanujan's approximation series and assuming arbitrary precision):

```
def stream(length):
  a = 2 * sqrt(2) / 9801
  b=1
  s=0
  n=0
  while b>10**-length:
    b=a*(fact(4*n)/pow(fact(n),4))*((26390*n+1103)/pow(396,4*n))
    s+=b
    n+=1
  return 1/s

stream(10**6)
```

which is less than 200 symbols in length, and when including the code for calculating factorials, square root, and powers are below 500 symbols. Adding the size of the arbitrary-precision arithmetic library would result in an additional increase for a more realistic size - do this for exercise. This suggests the upper bound on Kolmogorov's complexity of the input string which is well below the complexity suggested by SIT.

7.2 Lossless compression

Lossless compression methods are all about reducing statistical redundancy through the transformation of input data. This is achieved by:

- reducing the number of unique symbols
- and encoding more frequent symbols with fewer bits.

The goal is to minimize the average length of string $L_{avg} = \sum_i p_i l_i$.

Definition 7.5 (Compression ratio R)**.** $R = \dfrac{\text{uncompressed size}}{\text{compressed size}}$

There are many approaches and each is specialized in a way to the input data specifics. This is a form of algorithmic bias that the expert inputs into the encoder in the form of transformation heuristics trying to achieve the best compression ratio. There are different kinds of regularities in text, numbers, excel files, images, etc. Some algorithms achieve better compression ratios on some data types and worse on others.

The techniques used for better compression include the following:

- frequency based coding
- trees
- recursive dividing
- sampling
- quantization (rounding)
- dictionary for frequent words (blocking)
- functional transforms (eg. Fourier, wavelets)

In the following sections, we show two types of methods based on variable-length codes. The first is tree-based Huffman encoding, and the other is the dictionary-based Lempel-Ziv transformation method.

7.3 Huffman coding

Huffman coding [Huffman, 1952] is the first optimal variable length code (VLC) for translating from source alphabet $S = \{x_1, ..., x_n\}$ with its set of probabilities $P = \{p_1, ..., p_{n\}}$ into codewords $C = \{c_1, ..., c_n\}$ with corresponding lengths $L = \{l_1, ..., l_n\}$.

Theorem 7.3. For source S, there exists an optimal binary prefix code with properties:

1. $p_j > p_i \implies l_j \leq l_j$
2. codewords for the two least probable symbols are equal length
3. two longest codewords differ only in the last digit

There is some rounding around the negative bases of number 2 probabilities in the above theorem. We will address that point at the end of this section.

The basis of Huffman code is greedy recursive tree building that generates a left-skewed tree which results in the optimal ordering of assigned prefix codes with respect to the frequency. If the input alphabet X is not sorted according to P, we can do so before calling the Huffman algorithm. Or, we can use a priority queue within the algorithm. We shall assume below that X is pre-sorted. The implementation uses two queues: Q_1 for leaves and Q_2 for inner tree nodes.

Algorithm 1 Building Huffman tree

Input: source alphabet S and probabilities P sorted ascendingly according to P

Output: root of Huffman tree

1: **function** BUILDHUFFMANTREE(S, P)
2: $Q_1 \leftarrow Queue(), Q_2 \leftarrow Queue()$ ▷ Q_1 contains leafs, Q_2 contains subtrees
3: **for each** $x_i \in S$ **do**
4: $Q_1.enqueue((N_i \leftarrow Node(x_i), p_i))$
5: **while** $|Q_1| + |Q_2| \geq 1$ **do**
6: dequeue two nodes (N_a, p_a) and (N_b, p_b) with the lowest weights(probabilities), examining fronts of Q_1 and Q_2
7: $R \leftarrow Node(), R.left \leftarrow N_a, R.right = N_b$ ▷ left-right order invariant
8: $Q_2.enqueue(R, p_a + p_b)$
return $root = Q_2.dequeue()$

The resulting probability of the root node is 1. We traverse the tree for each symbol towards its leaf. We encode each left turn in the traversal as 0 and every right turn as 1. The coding results for each node are usually cached into a coding table to reduce the compute during encoding.

Q1. What is the complexity of building a coding table?

Q2. What is the complexity of encoding with and without coding table?

Q3. How would you write decoder from the encoded stream?

For decoding to work, the decoder and encoder must agree on the exact coding. We have seen that the above-described algorithm is ambiguous and is underconstrained to give a unique table from the source data (think of tie-breaking). Hence, the conversion table must go with the data. The decoder would build the tree from the conversion table. The decoding is simple: you start with always at the root and read bits from the incoming encoded stream. You consume bits until hitting the leaf node that stores the corresponding symbol. After that, you again reset to the root and continue.

Algorithm 2 Decoding Huffman code

Input: Y: encoded bit-stream, T: Huffman tree
Output: decoded data

```
1:  function DecodeHuffman(Y, T)
2:      C ← T.root
3:      while Y is non-empty do
4:          if C is leaf node then
5:              Output C.symbol
6:              C ← T.root
7:          B ← Y.fetch()
8:          if B = 0 then
9:              C ← C.left
10:         else
11:             C ← C.right
```

The above procedure gives the best code for individual symbols, but not for a given set. The problem is that it uses an integer-based number of bits for codewords - rounding errors amass. Additionally, the coding is permutation-invariant.

The improvement is block-coding that approaches the optimal performance. We find frequently occurring groups of letters and use them as symbols, in addition to individual symbols. That can increase the efficiency through the calculated average codeword length (utilizing Theorem 7.2).

Some codings do not need symbol grouping to increase efficiency. An example of such is arithmetic coding which uses recursive subdivisions to encode the message as a decimal number which is then transmitted.

Huffman coding example

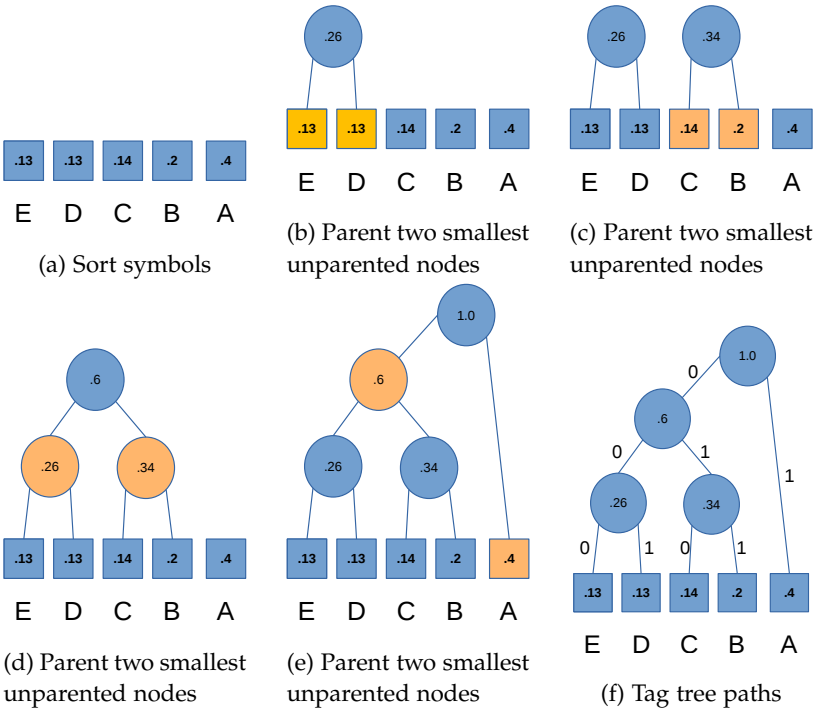
Assume source alphabet $S = \{A, B, C, D, E\}$ with probabilities $P = \{0.4, 0.2, 0.14, 0.13, 0.13\}$

a) Build Huffman coding table

b) Encode the input sequence *ABCABCABCE* and calculate the compression factor R.

c) Decode the bitstream 01110111.

Building the table. We must build the Huffman tree. We shall use squares for leaves, and circles for all added inner nodes. We show the procedure in Fig.7.1. First, we have to sort the symbols (Fig. 7.1a). Then, we take two unparented nodes with the smallest probabilities. In this case, such nodes correspond to symbols E and D. We create a new inner node that becomes their parent, and its probability is the sum of probabilities of its children i.e. 0.26 (Fig.7.1b). We repeat the procedure for nodes corresponding to C and B (Fig. 7.1c). In the following step (Fig.7.1d), the two smallest unparented nodes



(a) Sort symbols

(b) Parent two smallest unparented nodes

(c) Parent two smallest unparented nodes

(d) Parent two smallest unparented nodes

(e) Parent two smallest unparented nodes

(f) Tag tree paths

Figure 7.1: Example of building Huffman tree

are two inner nodes with probabilities 0.26 and 0.34 where we create their common parent with sum-probability of 0.6. Another iteration creates a node of probability 1.0 that is a parent of the node corresponding to A and the inner node parenting the rest of the symbols (Fig.7.1e). This node is the root of the Huffman tree. The final step for creating the code is tagging the paths in the tree, whereby we tag every traversal to the left subtree as 0, and 1 otherwise (Fig.7.1f). We obtain the coding of every symbol by walking from the root to the symbol's leaf node and concatenating traversal tags. For example, for symbol C the code is 010 since the traversal from the root collects and concatenates those edge-tags. We can always encode directly from the tree, but a more efficient way is to cache the symbol coding into a coding table. That is what we have done in Table7.1 for the above example.

symbol	code
A	1
B	011
C	010
D	001
E	000

Table 7.1: Huffman coding table for the example from Fig.7.1

Encoding data. Encoding the sequence ABCABCABCE is simple, using the Table7.1. We take symbol by symbol from the input and we output the corresponding code for each symbol. That results in an encoding: 101101010110101011010000. If we were to send the original sequence, assuming ASCII encoding, we would have to spend $10 * 8 = 80$bits. In the case of our coding, we have 24bits for the data. But, we also need to send the coding table to the decoder and that is an additional $5 * 8 + 1 + 4 * 3 = 53$bits of data (assuming it was not sent before). The total size sent is 77bits. The compression factor is $R = \frac{80}{77} = 1.04$.

Decoding data. We need the Huffman tree for decoding the sequence 01110111. The tree or the coding table is transmitted to the decoder. In the case of a coding table, the Huffman tree needs to

be rebuilt at the side of the decoder. Using the tree from Figure7.1f, we take bit-by-bit from the input and output the symbols reached in traversal while resetting the traversal from the root. For example, using bits 011 results in reaching leaf B. Restarting from the root, the following bit 1 reaches leaf A. We repeat the procedure and the resulting decoded data is "BABA".

7.4 Lempel-Ziv77 algorithm

We have seen that the efficiency of Huffman coding can be increased by block-encoding, that is finding frequently occurring groups of symbols and constructing super-symbols. Instead of doing such groupings manually, as we proposed before, let us introduce a representative of dictionary methods. They automatically find frequently occurring groups and encode them in shorter representations. Such methods transform the data prior to statistical encoders. Hence, it tries to produce a dataset that is smaller and more compressible than the source stream.

Dictionaries can be **static** (do not change throughout the process of encoding and decoding) or can be **dynamic** (adapt to the stream). In this section, we shall explain Lempel-Ziv77 (LZ77) [Ziv and Lempel, 1977] algorithm which is a dynamic dictionary approach that uses a sliding window. There is another well-known variant Lempel-Ziv78 (LZ78) [Ziv and Lempel, 1978] which we shall not explain in detail.

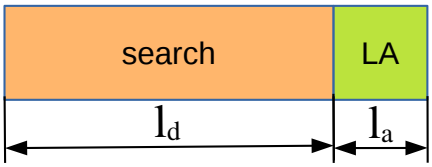


Figure 7.2: Buffer in LZ77

LZ77 algorithm maintains a buffer which is divided into two parts (check Fig. 7.2):

1. dictionary(search) buffer of size l_d which holds l_d most recently encoded symbols. l_d is usually in thousands of symbols, usually powers of 2 (e.g. 8192, 32768, etc)

2. lookahead buffer of size l_a which holds l_a symbols to be encoded. l_a is usually several tens of symbols (e.g. 16,32, etc.)

The encoder pseudocode is given in Algorithm 3.

Algorithm 3 LZ77 encoder

Input: I: input data, L: lookahead buffer, D: dictionary buffer
Output: encoded data

1: **function** LZ77ENCODE(I, L, D)
2: $L \leftarrow$ first l_a symbols within I
3: Fill D with copies of $L[0]$
4: $Output(L[0])$
5: **while** L is non-empty **do**
6: Find longest prefix p of L which starts in D ▷ special cases
7: $i \leftarrow$ distance of p from the end of D
8: $j \leftarrow$ length of p
9: $k \leftarrow$ first symbol after matched p in L
10: $Output(i, j, k)$
11: move composition D, L, I to the right by $j + 1$ symbols

Special cases in Alg.3, step 6 are:

Case 1. **spillover** - where matching pattern can spillover to lookahead buffer L and use the characters from it. All that matters is that the matching pattern starts within the dictionary buffer. When decoding such case, special care must be taken in Alg.4, step 6 to calculate the slice iteratively ("unroll the carpet", instead of "batch operation"). This is because a suffix of the string might not be available at the beginning of the step and can be revealed iteratively. Check the example of spillover decoding at the end of this section.

Case 2. **"empty" match** - when no matching prefix is found. Then the input is encoded with $(0, 0, L[0])$, where only the first symbol from L is used.

Case 3. **"full" match** - when complete L can be matched. In that case we encode the triple as if the last character of L was not matched.

The decoder (Algorithm 4) is computationally less intensive.

Algorithm 4 LZ77 decoder

Input: E: encoded data, L: lookahead buffer, D: dictionary buffer, B: concatenated buffer [D,L]
Output: decoded data

1: **function** LZ77DECODE(E, L, D)
2: Fill D with copies of consumed $E[0]$
3: **while** E is non-empty **do**
4: Consume first triplet (i, j, k) from E
5: $start = l_d - i, end = start + j$
6: $word \leftarrow B[start : end] + k$ ▷ python slicing notation
7: $Output(word)$
8: Right-shift B by $j + 1$
9: $B \leftarrow B + word$ ▷ concatenation

LZ77 encoding example

PROBLEM. Encode the input data ABCABCABCE using LZ77 and calculate the compression factor. Let buffer sizes be $l_a = 4, l_d = 4$.

The example of encoding using LZ77 is given in Table 7.2 and below we describe each step.

1. In the first row, we simply output the first symbol from the input, A which is the initialization symbol for the decoder. We also fill the dictionary buffer with the copies of this character (visible in the second row).

2. In the second row, we finally consume $l_a = 4$ symbols from the input into the lookahead buffer. We find the A to be the maximal matching prefix starting within the dictionary buffer. Out of all options for positions of a prefix, we pick the one closest to the lookahead buffer. The matching prefix in D is underlined, while the matched prefix in LA is colored green. We output $(i, j, k) = (0, 1, B)$ because $i = 0$ is the distance from the end of D, $j = 1$ for the length of matching and $k = B$ because it is the first symbol after the matching prefix. We move the composition D, L, I to the right by $j + 1 = 2$ places.

3. (Special case of the empty match) The current state of the buffer can be seen on the third row of Table 7.2. It is evident that there is no matching prefix in D that starts with symbol C. We will output $(i, j, k) = (0, 0, C)$, since $j = 0$ is the length of matching prefix, the first symbol after the (empty) prefix is C. The first element is practically arbitrary, but we choose $i = 0$ for simplicity. We move the composition D,L,I to the right by $j + 1 = 1$ places.

4. (Special cases of full match and spillover) On the fourth row, we could match the whole prefix ABCA using symbols ABC from the dictionary and lookahead spillover symbol A (spillover matching is possible). However, we must not match the whole lookahead since triple must always have the symbol after the prefix. Hence, we match only ABC. Triple $(2, 3, A)$ is output since the length of the matching sequence is 3, it starts on position 2 from the end of D. D, L, I is moved to the right by 4 positions.

5. Lookahead prefix that can be matched is BC. Hence, we output (2,2,E) triplet.

6. Lookahead buffer is empty, so we stop with the encoding procedure.

The final output is A(0,1,B)(0,0,C)(2,3,A)(2,2,E). This takes up 8bits for initializer symbol A and 12bits per each triple, which is in total $8 + 4 * 12 = 56$bits. Compression factor is $R = \frac{80}{56} = 1.43$.

LZ77 decoding examples

PROBLEM. Encode the encoded data A(0,1,B)(0,0,C)(2,3,A)(2,2,E) using LZ77. Let the buffer sizes be $l_a = 4, l_d = 4$.

buffer (4 dictionary; 4 lookahead)	input	output
	ABCABCABCE	A
AAAA;ABCA	BCABCE	(0,1,B)
AAAB;CABC	ABCE	(0,0,C)
AABC;ABCA	BCE	(2,3,A)
ABCA;BCE		(2,2,E)
ABCE;		

Table 7.2: LZ77 encoding example

input	buffer (4 dict; 4 lookahead)	output	shifted buffer
A	AAAA;		AAAA;
(0,1,B)	AAAA;AB	AB	AAAB;
(0,0,C)	AAAB;C	C	AABC;
(2,3,A)	AABC;ABCA	ABCA	ABCA;
(2,2,E)	ABCE;	BCE	ABCE;
	ABCE;		

Table 7.3: LZ77 decoding example

The decoding procedure is shown in Table 7.3 and below we describe each step.

1. Symbol A is consumed and dictionary buffer is filled with its copies
2. First triplet $(i, j, k) = (0, 1, B)$ is read. We fetch $j = 1$ symbol from $i = 0^{th}$ position from the end of dictionary buffer. This prefix is A onto which we concatenate the symbol $k = B$ to get $word = AB$. This *word* is output and put at the end of the filled portion of the lookahead buffer. The composition D,L is shifted to the right by $j + 1 = 2$ positions.
3. Prefix to copy is empty, so $word = C$. This is output and put in L.
4. Prefix is fetched by taking 3 symbols starting from the 2^{nd} position from the end of the dictionary, which is ABC. Hence $word = ABCA$ which is output and also put into L. D,L is shifted to the right by 4 positions.
5. Prefix is fetched by taking 2 symbols starting from the 2^{nd} position from the end of the dictionary, which is BC. Hence, $word = BCE$ which is output and also put into L. D, L is shifted to the right by 3 positions.
6. Encoded data source E is empty, so we stop with the decoding.

EXAMPLE (spillover decoding). The following problem setting has a much bigger lookahead buffer than the search buffer to exacerbate the spillover effect. Such a setting is inefficient in practice and works well only with highly repetitive strings, such as the one in this example.

Assume $l_d = 8, l_a = 20$ and that the current buffer state is *obuffalo;*. Also, let the input triplet is $(6, 16, g)$. The incoming triplet describes the 16-character range and we have only 7 referenced characters in the buffer! Naive batch decoding does not work here, since we are missing the suffix of 9 characters. We proceed iteratively:

1. the beginning of the range is at the position 6 from the end, at character 'b'.
2. Copy the 7 symbols that we have ("buffalo") to the end of the lookahead buffer. The current buffer state is *obuffalo;buffalo*. We have the remainder of 9 symbols.
3. Just copied symbols give us additional 7 symbols for copying. So we copy those 7 to the end of the lookahead buffer. The current buffer state is *obuffalo;buffalobuffalo*. We have the remainder of 2 symbols.
4. Take the remaining 2 symbols from the part copied in the previous step and copy them to the end of lookahead buffer. The current buffer state is *obuffalo;buffalobuffalobu*
5. Concatenate symbol 'g' to the end of lookahead buffer: *obuffalo;buffalobuffalobug*. Output the decoded word "buffalobuffalobug" and shift the buffer by 17 positions to the right.

7.5 Applications

Congratulations! You have learned two very commonly used algorithms for lossless compression. The list of applications is not short and they (accounting also for variants) include:

1. ZIP and gzip compression (commonly used variant uses LZ77 algorithms and Huffman coding combined)
2. PNG uses the above compression
3. GIF image format uses a variant of LZ77

These algorithms are so widely used that they are commonly supported at the hardware level in generally-used processors as well as special-purpose chips.

Moreover, the lengths of compressed data are measured through Lempel-Ziv complexity measure [2]. It is related to Kolmogorov complexity if programs are constrained to use only recursive copy operation (i.e. shallow copy). It is, however, the advancement from purely entropy-based approaches such as Huffman coding.

The whole field of compression, which is very vast in a multitude of different algorithms in different variants, is trying to go beyond entropy-based approaches by different hand-tuned heuristics that approach Kolmogorov complexity. These heuristics are based on assumptions on the characteristics of an input stream. For that reason, different algorithms are well suited only to some input types. A

[2] A. Lempel and J. Ziv. On the Complexity of Finite Sequences. *IEEE Transactions on Information Theory*, 22 (1):75–81, January 1976. ISSN 1557-9654. DOI: 10.1109/TIT.1976.1055501. Conference Name: IEEE Transactions on Information Theory

general algorithm would be able to learn the most compressible representation from any stream. Such general compression algorithms aiming at Kolmogorov complexity today are still not used due to the computational complexity of such approaches.

7.6 Conclusion

You have learned the most commonly used building blocks for lossless compression. Not bad! As you have seen, commonly used methods in lossless compression algorithms use the idea of coding more frequent combinations of symbols with shorter codes. The basic idea of all these approaches is based on statistical information theory based on entropy. There are many heuristics that try to circumvent the permutation-invariant basic approach of entropic encoders, that in fact attempt to bridge the gap between SIT and AIT by different transformations that make the input stream more compressible.

Lossless compression algorithms on the other hand willingly throw away information that is, relative to the receiver, semantically redundant for the intended purpose. The input signal is transformed into other domains where some components are quantized.

Bibliography

D. Haussler A. Blumer, A. Ehrenfeucht and M.K. Warmuth. Occam's razor. In *Readings in machine learning*, pages 201–204. Morgan Kaufmann, Inc., 1990.

Gregory J. Chaitin. On the Simplicity and Speed of Programs for Computing Infinite Sets of Natural Numbers. *Journal of the ACM*, 16(3):407–422, July 1969. ISSN 0004-5411. DOI: 10.1145/321526.321530. URL https://doi.org/10.1145/321526.321530.

David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, September 1952. ISSN 2162-6634. DOI: 10.1109/JRPROC.1952.273898. Conference Name: Proceedings of the IRE.

A. N. Kolmogorov. On Tables of Random Numbers. *Sankhyā: The Indian Journal of Statistics, Series A (1961-2002)*, 25(4):369–376, 1963. ISSN 0581-572X. URL https://www.jstor.org/stable/25049284. Publisher: Springer.

A. Lempel and J. Ziv. On the Complexity of Finite Sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, January 1976. ISSN 1557-9654. DOI: 10.1109/TIT.1976.1055501. Conference Name: IEEE Transactions on Information Theory.

Nick Littlestone and Manfred K. Warmuth. Relating Data Compression and Learnability. Technical report, 1986.

Shay Moran and Amir Yehudayoff. Sample Compression Schemes for VC Classes. *Journal of the ACM*, 63(3):21:1–21:10, June 2016. ISSN 0004-5411. DOI: 10.1145/2890490. URL https://doi.org/10.1145/2890490.

C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27(3):379–423, 1948. ISSN 1538-7305. DOI: 10.1002/j.1538-7305.1948.tb01338.x. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1948.tb01338.x. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.1538-7305.1948.tb01338.x.

R. J. Solomonoff. A formal theory of inductive inference. Part I. *Information and Control*, 7(1):1–22, March 1964a. ISSN 0019-

9958. DOI: 10.1016/S0019-9958(64)90223-2. URL https://www.sciencedirect.com/science/article/pii/S0019995864902232.

R. J. Solomonoff. A formal theory of inductive inference. Part II. *Information and Control*, 7(2):224–254, June 1964b. ISSN 0019-9958. DOI: 10.1016/S0019-9958(64)90131-7. URL https://www.sciencedirect.com/science/article/pii/S0019995864901317.

Hector Zenil. Towards demystifying shannon entropy, lossless compression and approaches to statistical machine learning. *Proceedings*, 47(1), 2020. ISSN 2504-3900. DOI: 10.3390/proceedings2020047024. URL https://www.mdpi.com/2504-3900/47/1/24.

J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977. ISSN 1557-9654. DOI: 10.1109/TIT.1977.1055714. Conference Name: IEEE Transactions on Information Theory.

J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, September 1978. ISSN 1557-9654. DOI: 10.1109/TIT.1978.1055934. Conference Name: IEEE Transactions on Information Theory.