Coursework 2

In this document we briefly describe the design of the Adaptive Run-length Huffman Compressor (ARHC).

1 Motivation

For the task of compressing $N = 10,000$ bits, each having probability $p = 0.01$ of being a 1, the extra bit overhead of the Huffman algorithm prohibits naive application of it; since the 0 symbol is highly probable, its observation conveys virtually no information and so encoding it by a bit is wasteful.

Imagine that we observe the input file bit by bit. Then the numbers of bits that pass before ones are observed fully describe the file, and in addition all have high information content. This motivates the approach to define each of these zero-counts, also called run-lengths, as the symbols of our alphabet. However, to make sure that our approach is well defined, we pick a maximum run-length n that defines the largest number of zeros denoted as a symbol. More precisely, let

$$\mathcal{A}_X = \big(1, 01, 001, \ldots, \underbrace{0\cdots01}_{n-1}, \underbrace{0\cdots0}_{n}\big),$$
$$\mathcal{P}_X = \big(p, (1-p)p, \ldots, (1-p)^{n-1}p, (1-p)^n\big)$$

be the ensemble that will be encoded by a Huffman code C_X.

It remains to determine an appropriate maximum run-length. Let L denote the expected codelength and let H denote the entropy of the distribution over the symbols in our symbol set. Then the expected number of wasted bits per symbol is $L - H$. The red line in Figure 1 shows the ratio L/H where L is determined by explicitly computing the Huffman code for the associated n, and the dashed green line shows an approximation.[1] We see that this ratio attains extrema at multiples of 69 and that L closely approximates H at these points.

The expected codeword length of the Huffman code is equal to the entropy when the distribution over the symbols in our symbol set is dyadic. Since all-zero symbol has probability 0.99^n, in order to have L/H close to one, we must have that $0.99^n \approx 2^{-k}$ for any natural k, which implies that $n \approx k \times 69$. We choose to use $n = 69$.

[1] We can determine the optimal run-length observed in Figure 1 by approximating the behaviour of the Huffman algorithm. First we exploit the structure of the symbols in $\mathcal{A}_X$ and decompose the entropy via

$$H(X) = \underbrace{H_2(p) + p(H_2(p) + p(H_2(p) + \ldots))}_{n \text{ terms}} = H_2(p)\sum_{i=0}^{n-1} p = \frac{H_2(p)}{p}(1-(1-p)^n).$$

Motivated by the fact that $1-p$ is small, we assume that the result of running the Huffman algorithm on the first $n-1$ symbols in $\mathcal{A}_X$ is similar to the result if they have equal probability. Exercise 5.28 in [1] yields that the first $\lfloor (n-1)f^- \rfloor$ symbols in $\mathcal{A}_X$ will have length $\lfloor \log_2(n-1) \rfloor + 1$ and the next $n - 1 - \lceil (n-1)f^- \rceil$ symbols in $\mathcal{A}_X$ will have length $\lceil \log_2(n-1) \rceil + 1$ where $f^- = 2^{\lceil \log_2(n-1) \rceil}/(n-1) - 1$. We can then approximate the expected length by

$$L \approx \sum_{i=1}^{\lfloor (n-1)f^- \rfloor} (1-p)^{i-1}p \cdot (\lfloor \log_2(n-1) \rfloor + 1) + \sum_{i=\lceil (n-1)f^- \rceil}^{n} (1-p)^{i-1}p \cdot (\lceil \log_2(n-1) \rceil + 1) + (1-p)^n.$$

The approximation of L/H using the above expression for H and above approximation for L is shown by the dashed green line in Figure 1. We see that the approximation holds perfectly for $n \leqslant 100$ where inspection of the computed Huffman codes shows that the assumption indeed holds.

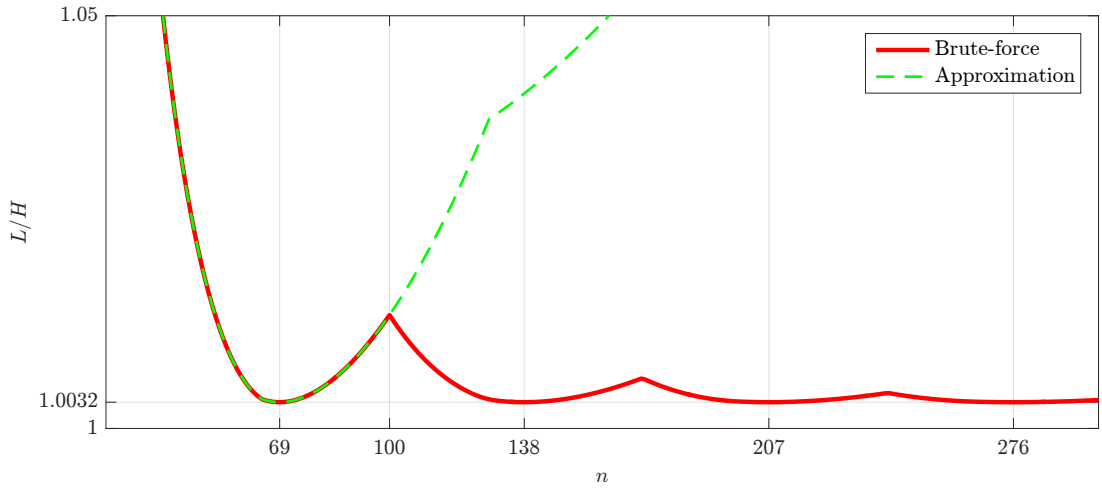


Figure 1: Ratio L/H for different run-lengths n

2 Description

Algorithm 1 provides a high-level overview of ARHC. Descriptions of the components of the program can be found in the source code or in the documentation in `documentation.epub`. Run scripts `squash` and `unsquash` to perform the exercise. Additionally, run `squash.py --help` for more advanced run-time options.

The implementation of the Huffman algorithm is facilitated through `Leaf` and `Node` objects that define a binary tree. The algorithm is initialised by encapsulating each symbol and its probability in a `Leaf` object, and the merge operation is performed by combining two `Leaf` objects into a `Node` object. A code word can be mapped to its symbol by providing it as an argument to the `decode` operation of the root `Node` of the tree. Additionally, the encoding of all code words can be obtained by calling the `getEncoding` operation of the root `Node` of the tree.

If p is unknown, then the user can provide parameters α_0, α_1 for a prior $\mathrm{Beta}(\alpha_0, \alpha_1)$ over p. On lines 17 to 19 of Algorithm 1, the posterior over p is updated according to the observation S.

The operation on line 6 of Algorithm 1 dynamically adjusts the run-length so that

(1) the run-length is near optimal for the current estimate of p and

(2) the symbols in $\mathcal{A}_X$ are able to entirely encode and decode the input stream[2] without the use of an end-of-transmission symbol or the length of the transmitted message.

Note that ARHC is able to perform partial compression and partial decompression; the compressor writes a symbol directly after the associated input bits are read and similarly the decompressor writes the input bits directly after the associated symbol is read.

3 Analysis

The source coding theorem tells us that we cannot compress the N-bits input to fewer than $NH_2(p) \approx 807.9$ bits. We can approximate the length of the compressed input by $\hat{N} = NL/L_i$ where L_i represents the expected

[2]When $n > N$, without adjustment $\mathcal{A}_X$ might not be able to encode the remaining bits in the input stream, and some symbols in $\mathcal{A}_X$ will never be used.

Algorithm 1 Adaptive Run-length Huffman Compressor

function ARHC(I, O, N, p or (α_0 and α_1)) ▷ I represents the input stream and O represents the output stream
  if adaptation **then**
    $p \leftarrow \alpha_1/(\alpha_0 + \alpha_1)$
  end if
  while $N > 0$ **do**
    $n \leftarrow \max\{1, \min\{\text{round}(-1/\log_2(1-p)), N\}\}$
    Obtain the code C_X by running the Huffman algorithm on the ensemble

$$\mathcal{A}_X = \big(1, 01, 001, \ldots, \underbrace{0\cdots01}_{n-1}, \underbrace{0\cdots0}_{n}\big),$$
$$\mathcal{P}_X = \big(p, (1-p)p, \ldots, (1-p)^{n-1}p, (1-p)^n\big)$$

    if compression **then**
      $S \leftarrow$ Read from I the first symbol that corresponds with a symbol in $\mathcal{A}_X$
      Write to O the code word of S according to C_X
    else if decompression **then**
      $S \leftarrow$ Read from I the first symbol that corresponds with a code word in C_X
      Write to O the symbol of S in $\mathcal{A}_X$ according to C_X
    end if
    $N \leftarrow N - \text{length}(S)$
    if adaptation **then**
      $\alpha_0 \leftarrow \alpha_0 + \text{count}_0(S)$
      $\alpha_1 \leftarrow \alpha_1 + \text{count}_1(S)$
      $p \leftarrow \alpha_1/(\alpha_0 + \alpha_1)$
    end if
  end while
end function

number of inputs bits a symbol in $\mathcal{A}_X$ represents. We calculate L_i via

$$L_i = \sum_{i=1}^{n-1} (1-p)^{i-1} p \cdot i + (1-p)^n \cdot n$$

which yields for $n = 69$ that $\hat{N} = 810.5$ bits; by averaging over 500 runs we indeed found that $\hat{N} \approx 812.3$. Table 1 shows an overview of the compression performance of ARHC.

	Expected length of the compressed input	Associated compression ratio
Optimal	807.9 bits	0.9192
ARHC	810.5 bits	0.9190

Table 1: Compression performance of ARHC

Figure 2 shows the compression performance of the static and adaptive compression scheme for different inputs lengths N. We see that for small N, the static compression scheme attains better compression ratio than the adaptive compression scheme, but also has higher variance. For large N, the static compression scheme and the adaptive compression scheme seem to perform similarly.

As the input is fed bit by bit to the compressor[3], Table 2 shows the immediate output, i.e. partial compression and decompression output, of the compressor and decompressor. This allows us to inspect the dynamics of the adaptive compression scheme. We see that as the adaptive compressor observes more bits, it learns to efficiently encode the all-zero symbol.

Figure 3 shows the compression performance in the case where the parameter p of the static compression scheme matches the probability of the bent coin and the case where it does not. We see that in the former case the static compression scheme converges more quickly to the optimal compression ratio than the adaptive compression scheme, though with higher variance, but that in the latter case the adaptive compression scheme is able to adapt to the mismatched probability and eventually outperform the static compression scheme.

Finally, Figure 4 shows the effect of different settings of α_0 and α_1 on the compression performance of the adaptive compression scheme. We see that for small α_0 and α_1 the compressor is able to adapt more quickly, but also has higher variance. Furthermore, we observe that $\alpha_0 = 1.0$, $\alpha_1 = 1.0$ initially attains a better compression ratio. This shows that an appropriate prior can indeed help compression performance when little is known about the input.

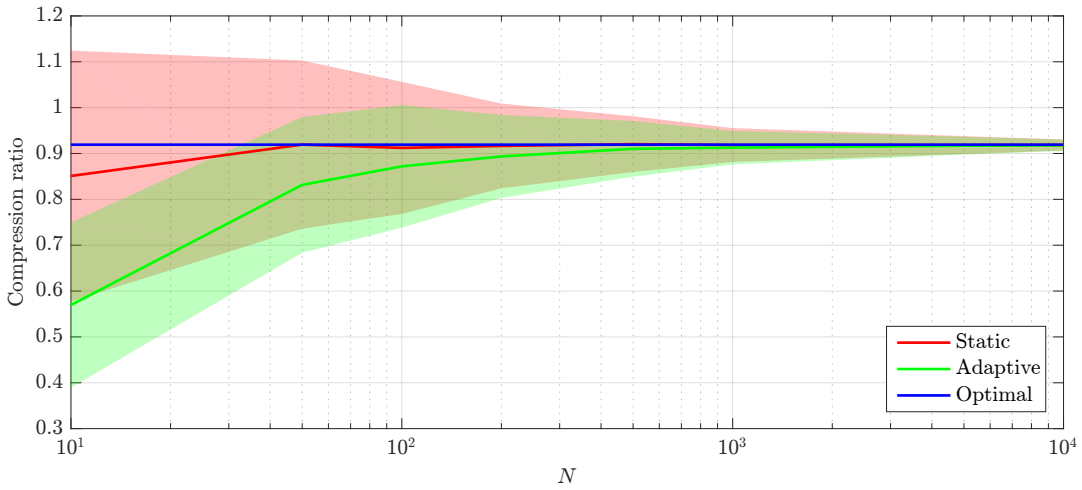
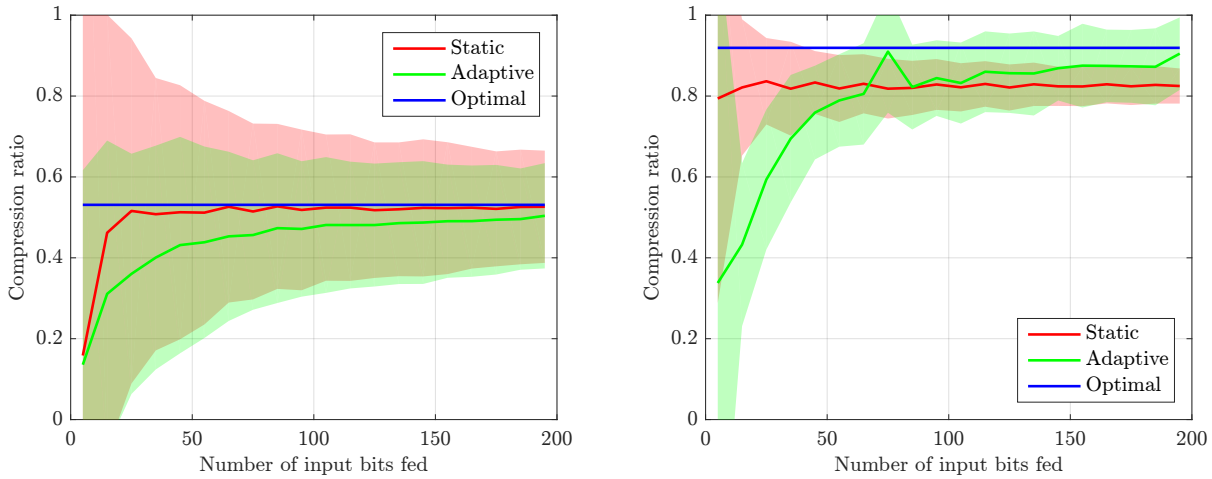


Figure 2: Compression performance of the static compression scheme and the adaptive compression scheme for different input lengths N

References

[1] D. J. MacKay, *Information Theory, Inference and Learning Algorithms*. Cambridge University Press. [Online]. Available: `http://www.inference.phy.cam.ac.uk/itila/book.html`.

[3]Feeding the input bit by bit to the compressor and decompressor turned out to be difficult; it required us to flag the input and output streams as non-blocking and take into account the processing time of the compressor and decompressor.



(a) Compressor matches reality; the parameter of the static compressor scheme is $p = 0.1$ and the bent coin has probability $p = 0.1$

(b) Compressor does not match reality; the parameter of the static compressor scheme is $p = 0.1$ and the bent coin has probability $p = 0.01$

Figure 3: Comparison of the static compression scheme and the adaptive compression scheme when a random 200-bit input is fed bit by bit. The parameters of the prior are $\alpha_0 = 0.1$ and $\alpha_1 = 0.1$. The lines and associated areas show respectively the means and 95% confidence regions determined over 200 runs.

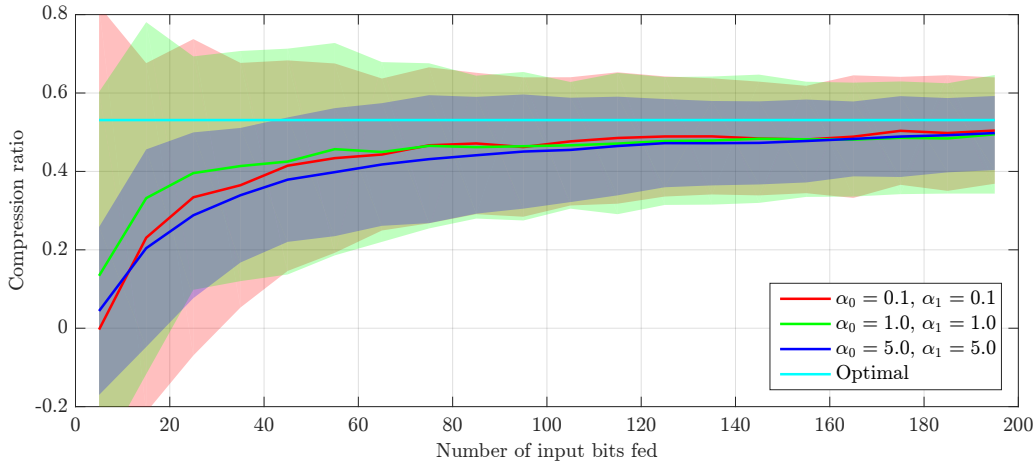


Figure 4: Compression performance of the adaptive compression scheme for different α_0 and α_1 when a random 200-bit input is fed bit by bit. The lines and associated areas show respectively the means and 95% confidence regions determined over 50 runs.

Input bit	Compressor output	Decompressor output
0		
1	1111001	01
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
1	1011100	0000000000000000000000000···
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
1	0100100	0000000000000000000000000···
0		
0	1	00

(a) Static compression scheme

Input bit	Compressor output	Decompressor output
0	0	0
1	011	1
0	0	0
0		
0	0	00
0		
0		
0	0	000
0		
0		
0		
0	1	0000
0		
0		
0		
0		
0		
0		
0	0	0000000
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0	0	00000000000
0		
0		
1	01100	001
0		
0		
0		
0		
0		
0		
0		
0		
0		
0	1	0000000000
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0	1	0000000000000
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0	1	00000000000000000
0		
0		
0		
0		
1	0100	00001
0		
0	1	00

(b) Adaptive compression scheme

Table 2: Output of the compressor and decompressor when the input is fed bit by bit. The input consists of the first 80 bits of the bent-coin benchmark file `filep.01.10000NR`. The parameters of the prior are $\alpha_0 = 0.2$ and $\alpha_1 = 0.2$. The static compression scheme compresses down to 22 bits, the adaptive compression scheme compresses down to 23 bits and the SCT tells us that we cannot compress further than $80\mathrm{H}_2(0.01) \approx 6.5$ bits.