The complexity of minimum redundancy coding

Citation for published version (APA):
The complexity of minimum redundancy coding

Tjalling Tjalkens*
Eindhoven University of Technology
P.O.Box 513, 5600 MB Eindhoven, The Netherlands
t.j.tjalkens@tue.nl

1 Introduction

An efficient implementation of a Huffman code can be based on the Shannon-Fano construction. An important question is exactly how complex is such an implementation. In the past authors have considered this question assuming an ordered source symbol alphabet. In the case of the compression of blocks of binary symbols this ordering must be performed explicitly and it turns out to be the complexity bottleneck.

2 The Huffman-Shannon-Fano code

\( \{X_i\}_{i=1}^{\infty} \) is a binary, i.i.d. source with

\[
\Pr(X_i = 1) = p = \frac{1}{2}.
\]

We consider FV codes for blocks of \( n \) symbols, \( X^n \), that have the minimal possible redundancy, (Huffman codes [4]). The minimum-redundancy codes or Huffman-Shannon-Fano codes that we shall consider were first described by Connell [1].

First assign to each block \( x^n \) a unique index \( i(x^n) \in \{0, 1, \ldots, 2^n - 1\} \) such that for all pairs of blocks \( x^n, y^n \) holds

\[
\Pr(x^n) = \sum_{i \in \{0, 1, \ldots, 2^n - 1\}} P(x^n)w_i(x^n)
\]

is minimal.

For all \( i, j \in \{0, 1, \ldots, 2^n - 1\} \)

\[
i < j \Rightarrow w_i \leq w_j.
\]

*This work was performed while the author was visiting the Information Technology Department of Lunds Tekniska Högskola, Lund, Sweden.

So, \( w \) is a non-decreasing sequence given the index ordering \( i(x^n) \).

Now it is time to introduce the Huffman-Shannon-Fano encoding procedure briefly. Given the code word lengths \( w \) we determine the number of codewords \( v(i) \) of a given length \( i \) as

\[
\forall_{i \in \{1, 2, \ldots, 2^n - 1\}} : v(i) \triangleq |\{ j \in \{0, 1, \ldots, 2^n - 1 \} : w_j = i \} |.
\]  

(4)

We shall use the following notation for the shortest and longest code word length.

\[
w_+ \triangleq \max \{ w_i, i \in \{0, 1, \ldots, 2^n - 1 \} \},
\]

(5)

\[
w_- \triangleq \min \{ w_i, i \in \{0, 1, \ldots, 2^n - 1 \} \}.
\]

(6)

From Nemetz and Simon [6] we know that for all \( x^n \) holds

\[
|w_{i(x^n)} + \log_2 P(x^n)| = o(n),
\]

(7)

and with the fact that \( \Pr[0^n] = -n \log_2 (1 - p) \) and \( \Pr[1^n] = -n \log_2 p \) we obtain that

\[
w_+ = -n \log_2 p + o(n),
\]

(8)

\[
w_- = -n \log_2 (1 - p) + o(n).
\]

(9)

Now we can compute the 'base' values by

\[
\forall_{w \in \{w_1, \ldots, w_d\}} : \text{base}(w) \triangleq \sum_{j=w_1}^{w_{i-1}} v(j) 2^{w_{i-1} - j} - v(j).
\]

(10)

The encoding procedure is now as follows. Given a source sequence \( x^n \) do

1. determine the index \( i = i(x^n) \);  
2. determine the code word length \( w = w_i \);  
3. produce the code word for \( x^n \) from the binary representation of \( \text{base}(w) + i \) in \( w \) bits.

Similarly for the decoding we need a list \( \text{prefix}(w) \) that describes the value of the largest code word of length \( w \). The decoding procedure is now

1. Set \( w \leftarrow w_- \) and read the first \( w_- \) code symbols into \( s \), i.e. \( s \in \{0, 1, \ldots, 2^{w_-} - 1\} \).  
2. If \( s \leq \text{prefix}(w) \) then decode the sequence \( x^n \) with index \( i(x^n) = s - \text{base}(w) \) and stop. Else read the next code bit into \( s \), i.e. \( s \leftarrow 2 \cdot s + 1 \) bit; increment \( w \), i.e. \( w \leftarrow w + 1 \), and repeat this step 2.

So, we must study the time and storage complexity of the index determination and the arrays \( w \) and base (and prefix).

### 3 Complexity issues

**Storage complexity**: We shall consider only the storage requirements for the encoding (and decoding) of a block \( X^n \). So, we do not take into account the cost of the preprocessing (designing the code). Also, we equate arithmetic computations with storage in the following way. Assuming that a simple operation, i.e. an addition, is performed on two \( b \)-bit operands, we say that its storage complexity is \( O(b) \) because we need a number of logic units, linear in \( b \), to compute this, just as we need a linear number of logic units to store a \( b \)-bit value. Note: we don't claim that we need the same number of units in both cases.

**Time complexity**: We require that the total time complexity is \( O(n) \). So, we don't want to pay more than a fixed amount of time per source symbol.

Usually we can interchange storage and time complexity by adding more units to perform more operations in parallel thus increasing the storage complexity while decreasing the time complexity and vice versa. We can implement the Huffman-Shannon-Fano code using a combination of tables and arithmetic. All these parts have a storage and time cost associated with it.

**Array access**: An array of length \( a \) containing integers in the range \( [0, 2^b - 1] \) requires \( a \cdot b \) bits of storage. We say that the storage complexity of such an array is \( O(a \cdot b) \) and the time complexity of a single access is \( O(1) \).

**Additions and subtractions**: The operation \( x = x + y \), where \( x \) and \( y \) are integers in the range \( [0, 2^n - 1] \) has a storage complexity of \( O(b) \) bits for the operands, results, and the computation. The time complexity of such an operation is then \( O(1) \).

**Multiplications and divisions**: A multiplication \( z = x \cdot y \), of integers \( x \) and \( y \) in the range \( [0, 2^n - 1] \) and a division \( y = z / x \) can be implemented by \( O(b^2) \) operations. A more efficient multiplication algorithm, known as the Karatsuba Multiplication, see [5, 9], requires \( O(b \ln b \ln \ln b) \) operations. We shall need a time complexity of \( O(1) \) and thus need a storage complexity of \( O(b^2) \) bits (naive implementation) or \( O(b \ln b \ln \ln b) \) bits for the Karatsuba algorithm.

Divisions however, cannot be computed in \( O(1) \) time using the long division method, while faster methods require too much memory. So, a division has a time complexity of \( O(n) \) and a storage complexity of \( O(n) \).

### 4 The complexity of the base array

The base array contains \( w_+ - w_- + 1 \) elements, so its length is \( O(n) \). From Connel [1] we know that the base \( (w) \) array contains values in the range \( [0, 2^n - 1] \) and so it requires \( n \) bits of storage per value. Thus the storage is \( O(n^2) \). The time complexity of the code word generating process, given the index \( i \) and the code word length \( w_i \) was described above and is \( O(1) \). Because we allow a time complexity of \( O(n) \) there is another option, namely, to compute the base resp. prefix values when needed. This can be done in \( O(n) \) time.
5 The complexity of the code word length array

Next we consider the cost of the $w$ array. This array consists of $2^n$ entries ranging in value from $w_-$ to $w_+$ in non-decreasing order. And since the lengths are the lengths of a binary Huffman code the entries satisfy Kraft’s inequality with equality.

We now show that the minimum amount of memory required to access a $w_i$ value is at least equal to the logarithm of the number of different arrays $w$ and state that fact in the following lemma whose trivial proof we omit here.

Lemma 1 Let $f: \mathcal{W} \times \{0, 1, \ldots, 2^n - 1\} \rightarrow \mathbb{N}$ be the function that produces for every $w \in \mathcal{W}$ ($\mathcal{W}$ is the set of all possible $w$ arrays), and $i \in \{0, 1, \ldots, 2^n - 1\}$ the corresponding code word length $w_i$. If $\mathcal{F}$ is a machine that computes $f(w_i, i)$, initially given an array $w$, for any possible index $i$, then $\mathcal{F}$ requires at least $\log_2 |\mathcal{W}|$ bits of storage.

Now consider the number of different $w$ arrays of non-decreasing values in the range $w_-$ to $w_+$. As we shall see, it turns out that we can find a machine $\mathcal{F}$ that needs $O(n^2)$ bits of storage.

5.1 Huffman code for arbitrary sources

It is well known that the number of binary, planar, rooted trees with $2^n$ leaves is given by

$$|\mathcal{W}_a| = \frac{1}{2^{n+1} - 1} \left(\frac{2^{n+1} - 1}{2^n} \right).$$

(11)

All these trees are Huffman codes for some sources. So, with $\log_2 \binom{n}{k} = \frac{1}{2} n \log_2 n + O(\log_2 n)$ we obtain $\log_2 |\mathcal{W}_a| = 2^{n+1} + O(n) = O(2^n)$, which is of the same order as the amount of storage needed to describe the Huffman code word table directly.

However, we don’t need all these trees because two trees with the same number of leaves, or terminal nodes, on each level are considered equivalent for our purpose. Norwood [7] considered the problem of determining this number of non-equivalent Huffman codes. She found a recursive description that seems to indicate that the number of possible trees is by no means the function that produces for every $w \in \mathcal{W}$ ($\mathcal{W}$ is the set of all possible $w$ arrays), and $i \in \{0, 1, \ldots, 2^n - 1\}$ the corresponding code word length $w_i$. If $\mathcal{F}$ is a machine that computes $f(w_i, i)$, initially given an array $w$, for any possible index $i$, then $\mathcal{F}$ requires at least $\log_2 |\mathcal{W}|$ bits of storage.

Later Flajolet and Prodinger [3] proved that, as $n \rightarrow \infty$, $|\mathcal{W}_a| \rightarrow k-1.7942^n$.

Let us also consider the number of different $w$ arrays of length $2^n$ with values in the range $1$ to $2^n - 1$. In order to simplify the analysis we shall overestimate the actual value by not requiring that the $w$ array satisfies the Kraft inequality. We show in appendix A that $\log_2 |\mathcal{W}_a| = 2^{n+1} + O(n) = O(2^n)$. So all three estimates indicate that the storage cost of specifying the $w$ array is $O(2^n)$.

5.2 Huffman codes for memoryless sources

The array $w$ that we must describe contains only values in the range $w_-$ to $w_+$ inclusive. We can expect that there are less codes given this extra restriction. In appendix B we show that

$$\log_2 |\mathcal{W}_d| = O(n^2).$$

(12)

The actual value derived in this appendix is an overestimation of the true number of $w$ arrays when we take Kraft’s inequality into account. We could however assume that here too, as in section 5.1, the order will be correct, but even if we don’t accept this, a cost of $O(n^2)$ for $w$ is acceptable because we need at least that amount of storage for the index computation.

Moreover, we can compute $w_i = f(w_i, i)$ with a machine that needs $O(n^2)$ bits of storage and $O(\log n)$ operations to produce $w_i$ in the following way. Using a storage of $O(n^2)$ we can use an array $Y$ where

$$Y_j = 1, 2, \ldots, 2^j : Y_j = \sum_{i=1}^{j-1} v_i,$$ 

(13)

and $v_i$ is defined in (4). From (20), in Appendix A, we know that we must find the index $j$ such that $Y_j < i < Y_{j+1}$ and set $w_i = j$. This can be done using a binary search on the $Y$ array, requiring a time complexity of $O(\log n)$.

6 The complexity of the index computation

The computation of the index $i(x^n)$ is yet another type of problem. While the $w$ and base($w$) arrays depend on the source probability $p$, the index only depends on the chosen index ordering.

A possible index ordering is

order the $x^n$’s in decreasing number of zeroes, i.e. in decreasing probability order, and within each group of equal weight blocks order them lexicographically.

It is obvious that the lengths resulting from the Huffman procedure can be rearranged such that $w$ is non-decreasing with the above mentioned ordering of blocks. This index ordering seems reasonable as we have to make sure that the sequence is in decreasing probability order and this requires that it is in increasing weight (number of ones per word) order, because $p \leq \frac{1}{2}$. The only freedom we have is in ordering sequences within a block of fixed weight. If we use the lexicographical order we can use the enumerative scheme of [8], see also [2].

We shall denote the number of ones in a sequence $x^n$ by $n_1(x^n)$. So, let $S_n$ be the set of all binary sequences of length $n$ with $b$ ones and $n-b$ zeros. It is well known that $|S_n| = \binom{n}{b}$. We define the sequence index $i(x^n)$ of a sequence containing $b = n_1(x^n)$ ones to be $i(x^n) = B(b) + j(x^n)$, with

$$B(b) \triangleq \sum_{j=0}^{b-1} \binom{n}{j},$$

(14)

$$j(x^n) \triangleq \sum_{k=1}^{n} \left[ n_1(x^{k-1}) - n_1(x^n) \right].$$

(15)

The first part $B(b)$ can be stored in an array of length $n + 1$ which requires $O(n^2)$ bits of storage, which is of the same order as the other arrays. However, the last part, $j(x^n)$, must be treated more carefully. In the next two sections we consider the determination of the index using either a table of binomials or the computation of the required coefficients.
6.1 Index computation using a table of binomial coefficients

Suppose we store all possible coefficients \( \binom{n}{k} \) for \( a = 0, 1, \ldots, n \) and \( b = 0, 1, \ldots, a \) in an array \( \text{Bin}(a,b) \). Obviously, we need to store \( O(n^2) \) coefficients, each one has a size of \( n \) bits. So the total storage cost is \( O(n^3) \).

The total time complexity of the index computation becomes \( O(n) \) and the storage complexity is dominated by the \( \text{Bin}(a,b) \) table and thus is \( O(n^3) \).

The inverse procedure, finding \( x^n \) from \( i(x^n) \) can be done in a similar way. Just as with the index computation we find that the time complexity of the decoder is \( O(n^2) \) and the storage complexity is \( O(n^3) \).

6.2 Index computation using computed binomial coefficients

A storage complexity of \( O(n^3) \) is more than we would like to have. Another option is to compute the coefficients as we need them.

The algorithm based on the computation of the coefficients requires \( O(n) \) additions and subtractions, \( O(n) \) multiplications, and \( O(n) \) divisions. The additions and subtractions cost \( O(n) \) storage and \( O(n) \) time. The multiplications can also be done in \( O(n) \) time with \( O(n^2) \) storage. However, the cost of a division using long division is \( O(n) \) time and \( O(n) \) space, and because the \( O(n) \) divisions required have to be performed sequentially, the total time complexity is \( O(n^2) \). So in this case we find a total storage complexity of \( O(n^2) \) and a time complexity of \( O(n^2) \).

7 Conclusions

Summarizing the complexity:

- The cost of the code word generation. When we store the base array the time complexity is \( O(1) \) and the storage cost is \( O(n^2) \) bits. We also showed that it is possible to compute the base resp. prefix values when we need them in \( O(n) \) time. The storage cost then is \( O(n) \) because we must save the resulting codeword.

- The cost of the code word length array. We described an algorithm that produces the required code word length from the source sequence index in \( O(\log n) \) time and \( O(n^2) \) storage space.

- The index computation is still an open question. We have two options, either we use a table of binomial coefficients, Pascal's triangle, or we compute the required coefficients. Pascal's triangle requires \( O(n^2) \) bits of storage and the computation of the index then costs \( O(n) \) time. Computing the coefficients requires \( O(n) \) divisions that must be performed sequentially thus resulting in a time complexity of \( O(n^3) \), which is unacceptable.

A The number of \( w \) arrays

First we shall give a simple combinatorial lemma that gives the number of partitions of an integer.

Lemma 2 Let \( k \in \{1, 2, \ldots\} \) and \( r \in \{1, 2, \ldots\} \) be given and let \( \underline{y} = v_1 v_2 \cdots v_k \) be an array of non-negative integer values. The conditions on \( \underline{y} \) are

\[
\forall i \in \{1, 2, \ldots, k\} : v_i \in \{0, 1, 2, \ldots\}
\]

\[
\sum_{i=1}^{k} v_i = r.
\]

So \( \underline{y} \) is a partition of the integer \( r \) into \( k \) parts.

The number of \( \underline{y} \) arrays that satisfy these conditions is given by

\[
|\underline{y}| = \binom{k + r - 1}{k - 1}.
\]

For reason of brevity we omit the proof of this lemma.

Now consider the array \( \underline{w} \). It has \( 2^n \) elements and each element is a code word length between 1 and \( 2^n - 1 \) inclusive. The array \( \underline{w} \) is ordered in non-decreasing order. We can define another array \( \underline{y} \) that describes how many code words of a given length are in the code. It has \( (2^n - 1) - (1) + 1 \) entries, one for each possible code word length. Consider the relation between \( \underline{w} \) and \( \underline{y} \). We have by definition of \( \underline{y} \)

\[
\forall_{i=0, \ldots, 2^n-1} w_i = \left\lfloor \frac{r}{j} : w_j = i \right\rfloor.
\]

And because \( \underline{w} \) is an ordered array we can find \( w_i \) from the \( \underline{y} \) array by

\[
\forall_{i=0, \ldots, 2^n-1} w_i = \max\{j \in \{1, \ldots, 2^n - 1\} : \sum_{k=i}^{j-1} v_k < i\}.
\]

So the relation between \( \underline{y} \) and \( \underline{w} \) is one-to-one. \( \underline{y} \) satisfies the following conditions:

\[
\forall i \in \{1, 2, \ldots, 2^n - 1\} : v_i \in \{0, 1, 2, \ldots\}
\]

\[
\sum_{i=1}^{2^n-1} v_i = 2^n.
\]

The number of \( \underline{y} \) arrays is given by Lemma 2, so also the number of \( \underline{w} \) arrays, \( |W| \) is given as \( |W| = \binom{2^n - 1}{2^n - 1} \). And we obtain that

\[
\log_2 |W| = 2^{2^n - 1} + O(n).
\]

B The number of \( w \) arrays with \( O(n) \) values

Consider the number of \( \underline{w} \) arrays given that there are two integers \( a \) and \( b \) with \( 1 \leq a \leq b \leq 2^n - 1 \) such that for all \( \underline{w} \) holds \( a = w_0 \leq w_1 \leq \cdots \leq w_{2^n - 1} = b \). As in Appendix A we consider an array \( \underline{y} = v_1, \ldots, v_{2^n - 1} \) that satisfies (4). Now it is also clear that \( \underline{y} \) is a partition of \( 2^n \) into \( b - a + 1 \) parts and with Lemma 2 we find

\[
\log_2 |W(a,b)| = (2^{2^n - 1} - 1) b \left( \frac{b - a}{2^{2^n - 1} - 1} \right) + O(n).
\]
From (8) and (9) we know that for our codes \( b - a = w_+ - w_- = cn + o(n) \), where \( c = \log_2 \frac{1-p}{p} \), and thus, with \( N \triangleq 2^{n+1} - 1 \), we find
\[
\log_2 |W(w_-; w_+)| = Nh\left(\frac{c \log_2 N}{N}\right) + o(n).
\] (25)

All that remains to examine is the function \( f(m) \triangleq mh(c \log_2 n) \). Using a Taylor series expansion we can finally conclude that
\[
\log_2 |W(w_-; w_+)| = cn^2 + o\left(n \log_2 n\right) = o(n^3).
\] (26)

References


