University of Alberta

Variable-Length Constrained-Sequence Codes

by

Andrew Steadman

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science
in
Communications

Electrical and Computer Engineering

©Andrew Steadman
Fall 2011
Edmonton, Alberta

Abstract

The use of variable-length codes to construct capacity approaching constrained sequence codes is examined. Constrained sequence codes are commonly used for digital storage and transmission, but have historically been implemented using block codes. A new technique is developed for constructing variable-length constrained sequence codes. These codes are based on Huffman encoding various partial extensions, a technique that is shown to be optimal in the sense that no other technique can result in a higher code rate given the same partial extension. Partial extensions are exhaustively searched in order to identify the highest rate code within a particular bound. Examples of the technique are provided for various constraints, including (d, k) and DC-free constraints. These examples are shown to have average rates within 1% of capacity for their respective constraints. Tables are also provided listing the optimal sourceword to codeword length mappings for various other constraints.

Acknowledgements

This work was made possible by the generous support of both the Natural Sciences and Engineering Research Council of Canada, and Alberta Innovates Technology Futures.

I would also like to acknowledge the support of my friends and family, without whom I would have neither the passion nor desire to explore and discover.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The use of modern digital media for storage of data is, at first, a seemingly trivial task. Consider, for instance, an optical storage medium such as a CD-ROM or DVD. The data, consisting of logical 1's and 0's, can be translated into either 'pits', small holes in the surface of the disc, or 'lands', unperturbed regions of the disc. Closer examination of such an ad hoc approach reveals that it is prone to error. The process of writing to the media, the degradation of the media over time, and the process of reading from the media, are all subject to the non-idealities of the physical world, and occasionally physical constraints. As a result, their operation may impose additional constraints on the system in order to ensure its proper operation. The same can be said of any storage media or digital communication channel; the only difference is the specific limitations to which the system is subject.

The extreme requirements of modern digital storage media with regard to data density has lead to the development of new coding techniques for data storage. In contrast to a wireless channel, where errors are common and the presence of noise and interference is significant, unpredictable and unpreventable, the designers of digital storage systems have far greater control over factors such as interference and noise. As a result, the quest for maximum data density has less to do with efficient error control and correction, and more to do with efficient error prevention. By satisfying the constraints of the storage channel before any data is written, the system is far less likely to encounter errors later on. A class of channel codes, referred to as constrained sequence codes (CSCs), has been developed with the intention of satisfying the constraints of a channel before the data is actually stored. When employed in a recording system, these codes are often called recording codes; in a transmission system, these codes are usually called line codes. These terms are used interchangeably in this thesis.

A typical digital storage system takes the form of Figure 1.1. Data to be stored is encoded in as many as three stages. The first stage is a source code to remove as much redundancy from the source sequence as possible. Then, it is encoded with an error correction code (ECC), which adds additional symbols that can be use to reconstruct the original sequence in the presence of some errors. The parity bits generated by the ECC stage are appended to the bits being stored, and the entire sequence is then re-encoded using a constrained sequence code. The CSC must directly precede the modulator in order for the code to function as intended. The modulator is responsible for converting the logical symbols into an appropriate physical representation for the channel.

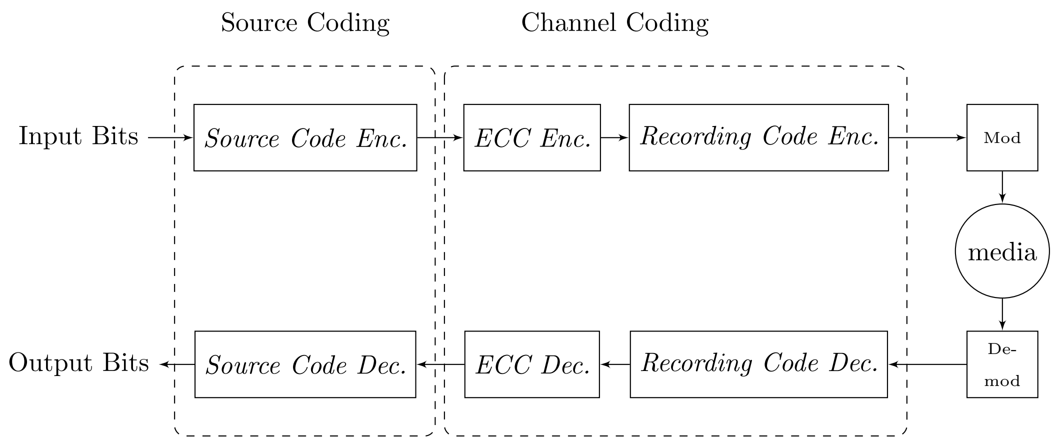

Figure 1.1: Structure of a digital storage system

Any errors that are introduced on the channel may be propagated to several more errors when the CSC is decoded. The ECC decoder must therefore be able to compensate not for the typical number of errors expected on the channel itself, but the typical number of errors present after the line decoder, potentially a much more difficult problem. Ideally, storage systems would implement joint ECC and constrained sequence codes, however how this should best be done remains an open question. Another issue with this approach is that the design of line codes typically assumes maxentropic input sequences, which is at odds with the concatenation scheme proposed in Figure 1.1, since redundancy is explicitly added by the ECC code. However, if the line encoder operates on significantly fewer bits than the ECC code during each encoding step, as will be assumed here, then the statistical dependence of the input sequences can usually be ignored.

Note that digital storage systems are one type of digital communication system. The general structure of a digital communication system is outlined in Figure 1.2; note the similarities with Figure 1.1.

1.1 Capacity

When discussing communication systems, one of the key underlying concepts is the notion of channel capacity, as pioneered by Claude Shannon. Shannon proposed two approaches to studying capacity: the noisy channel coding theorem [4], and the noiseless source coding

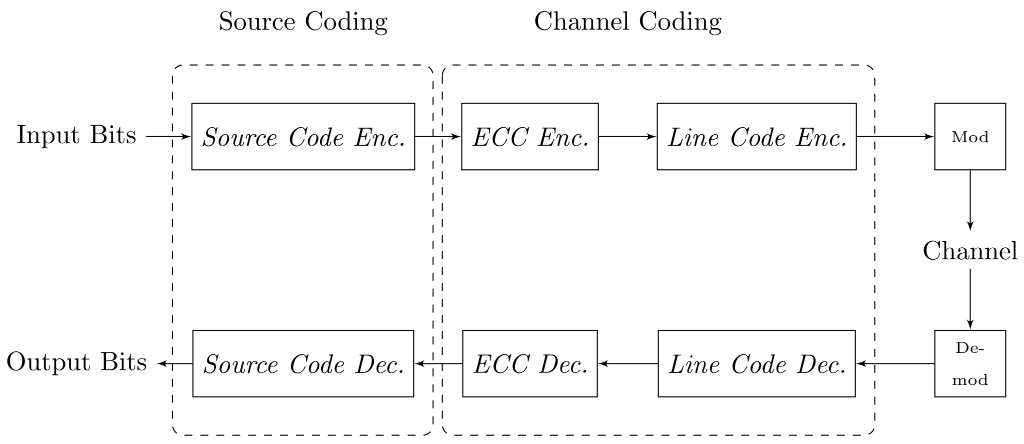

Figure 1.2: Structure of a digital communication system

theorem [5]. In both cases, the capacity is the upper-limit on the (error-free) rate of information flow that can be achieved by the communication system.

For the classical additive white Gaussian noise (AWGN) channel, the formula for channel capacity is well known:

$$C_{awgn} = W \log_2 \left(1 + \frac{\bar{P}}{N_0 W} \right) \text{ bits of information / s,} \tag{1.1}$$

where W represents the bandwidth in Hertz, $\bar{P}$ is the average power in Watts, and N_0 is the noise power spectral density in W/Hz. Here the distinction has been made between bits of information and binary digits. The term 'bits of information' makes explicit the information-theoretic concept of binary information. A single binary digit, depending on the context in which it is used, may contain at most 1 bit of information, but often contains less. For succinctness, this paper will use the term 'bits' to refer to bits of information, and will reserve 'binary digits', or 'bindigs', for all other uses.

Entropy of Memoryless Sources

The study of the information content of sequences is based on the entropy of such sequences. Detailed discussion of entropy is outside the scope of this text, but has been discussed at great length elsewhere [5, 6].

A memoryless source outputs symbols based on a statistical model that is time invariant and independent of previously emitted symbols. Let symbol X_i have a probability of occurrence defined by p_i, where it is assumed that the sum of all possible probabilities is unity. Furthermore, assume that all possible symbols have the same length. The information content of such a source relates directly with the a priori uncertainty of the symbols being generated. An expected symbol contains little to no information, while an unexpected symbol contains a good deal of information. Shannon proposed the following definition of entropy for a memoryless source [5]:

$$H(p_0, \ldots, p_N) = -\sum_i p_i \log_2 p_i \tag{1.2}$$

For a binary memoryless source, the equation for entropy reduces to:

$$h(p) = -p \log_2 p - (1-p) \log_2 (1-p) \tag{1.3}$$

where $h(p)$ is known as the binary entropy function, and p is a stochastic variable representing the probability of one of the symbols. It can be shown that the maximum entropy of a memoryless source occurs when all symbol probabilities are equal, and has a value of $log_2 N$, where N is the number of symbols. It can also be shown that a maxentropic binary source is memoryless, where the symbols '0' and '1' occur with equal probability. The entropy of such a source is 1 bit/bindig. Non-maxentropic binary sources are either not memoryless, or have symbols that occur with unequal probability. The entropy of such a source is less than 1 bit/bindig.

If a binary maxentropic source is encoded, the scenario assumed in this work, the entropy of the output sequence is the entropy of the input sequence (1 bit/bindig) multiplied by the average rate of the code (R_{avg} bindig/bindig) to give R_{avg} bits/bindig. For CSCs, there is an upper-limit on the best code rate that can be achieved by an encoder that enforces the constraint. Such an upper-limit is known as the capacity of the constraint, an important concept that will be discussed in depth later on in this chapter.

Markov Information Sources

One method of modeling the memory of a source is to use a Markov chain where the output symbols are a function of the Markov chain. Markov chains and their analysis will not be discussed in depth, but more information is available in the literature [6]. Analysis of constrained sequence sources is typically restricted to ergodic Markov chains, which are chains that are both irreducible and regular [7]. An irreducible Markov chain is one where it is possible to reach any state from a given arbitrary initial state. A regular Markov chain has states that are non-periodic. A periodic state is a state that can only be reached at integer multiples of some period. For coding purposes, the output symbols are assigned in such a way that the resulting Markov source is deemed unifilar [7]. A unifilar Markov source is one where the transitions between states can be uniquely determined by the observed output symbols and knowledge of the initial state. Markov information sources are used to model the allowed sequences of a constraint, and the same model can be used to calculate the capacity of a constraint. An example of a Markov information source is shown in Figure 1.3.

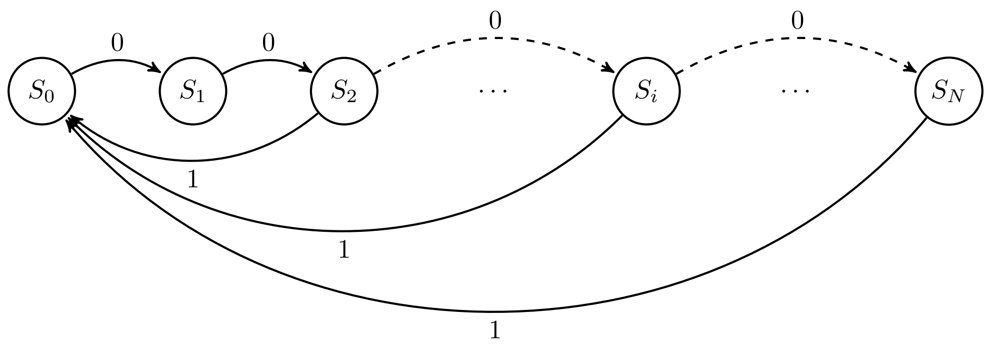

Figure 1.3: Example of a Markov Information Source

Deriving Capacity

For constrained sequence coding, the concept of capacity is slightly different from (1.1), as it relates to Shannon's noiseless coding theorem rather than his noisy channel coding theorem. Instead of studying the influence of noise on the transmission of information, the question is asked: in a noiseless system, what is the highest rate at which we can encode information and still satisfy the constraint? Ultimately, for any given constraint there is an upper limit on the rate of information that a sequence of symbols satisfying the constraint is capable of representing. Intuitively, a constraint on the sequences that are allowed in the output *must* reduce the maximum rate of information (in bits/symbol) such a sequence could contain as compared to an unconstrained sequence. A simplified model of capacity would examine the ratio of the number of sequences that satisfy the constraint to the total number of unconstrained sequences, in the limit as the sequence lengths tend to infinity. If a logarithm is used, then the numerator and denominator are written in terms of units of information, and the capacity can be viewed as a ratio of entropies.

$$C_{CSC} = \lim_{\text{sequence length}\to\infty} \frac{\log(\#\text{ of constraint-satisfying sequences})}{\log(\#\text{ of unconstrained sequences})} \tag{1.4}$$

This equation also gives some insight into the ideal behavior of a constrained sequence encoder. The entropy of an equiprobable and independent N-ary source is $\log_2 N$, where N is the number of elements in the set (see Entropy of Memoryless Sources, page 3). Note the similarity between this result and the numerator and denominator of equation (1.4). This relationship implies that as the sequence lengths extend to infinity, an encoder achieves capacity by translating equiprobable and independent input sequences into equiprobable and independent output sequences.

More often a separate technique is used to determine the capacity of a constraint, based on the Markov information sources discussed earlier. Assume, in the given Markov information source, only certain sequences (of a certain length) are output from a given state. If $\{L_{ij}\}$ denotes the set of lengths for each of the output sequences in the transition from state i to state j, then define the elements of a connection matrix $D(z)$ as:

$$d_{ij}(z) = \sum_{l \in \{L_{ij}\}} z^{-l}. \tag{1.5}$$

This connection matrix contains information regarding the number of output symbols in all possible transitions of the Markov chain, as well as the corresponding output lengths. Let λ_{max} represent the largest real root of the characteristic equation:

$$det[D(z) - I] = 0. \tag{1.6}$$

For sufficiently large sequence lengths, the number of paths between two states in the Markov chain has been shown to grow exponentially with the sequence length [1]. The growth factor is λ_{max}, as this eigenvalue will dominate in the limit as sequence lengths tend to infinity. Using this relationship to determine the number of unique sequences of a given length produced by the Markov source, and invoking (1.4), Shannon showed that the capacity of the constraint is:

$$C_{CSC} = \log_2 \lambda_{max}. \tag{1.7}$$

A code is a procedure for mapping arbitrary input sequences into other output sequences. The art of code design is to translate input sequences into output sequences at a rate as close as possible to the upper-limit, which is defined by the capacity. If the code is then implemented as part of a physical communication system, with well understood channel and noise models, then Shannon's noisy coding theorem also applies and a separate (smaller) capacity for the overall system can be derived. However, since the same codes can be used in many different systems with different channel and noise models, the first (noiseless) channel capacity derivation is useful in comparing the relative merits of different constrained sequence coding schemes.

1.2 Constrained Sequence Codes

There are many different types of constraints that may be required of digital sequences in order to make use of a given channel to the fullest extent possible. The rules for translating an arbitrary input sequence into a constraint-satisfying output sequence are given by a constrained sequence code. The goal of constrained sequence coding is to prevent errors that would have occurred had the constraints of the channel not been met. If a given code can achieve an average rate close to the capacity of the constraint for a given input sequence, then the information content of the output sequence is close to the theoretical maximum. High information density is a key requirement of modern digital storage systems, where it is desired that data be stored in as small a volumetric area as possible. Frequently, systems with binary inputs and binary outputs are examined; CSCs for such systems are properly referred to as binary CSCs. However, since these are the most common types of codes studied, we will simply refer to them as CSCs. Higher order alphabets will not be examined in this thesis.

The constraints present in digital storage systems are, interestingly, also present in almost all digital communication systems. For example, it is very common to have an AC-coupled channel in both data storage and communication applications. Magnetic media cannot properly store the low-frequency content of signals, while Ethernet uses transformers that prevent the transmission of DC and low-frequency signals. As a result, both systems use a CSC to prevent sequences with DC content from being stored/transmitted. This type of constraint is referred to as a DC-free constraint.

Another common constraint deals with timing accuracy and minimum detection time. In an asynchronous system, a long string of the same consecutive symbol may prevent a clock generator at the receiver from operating properly. As a result it may be desirable to insert frequent transitions in the data stream to facilitate proper reception of the data. On the other hand, there may be a lower bound on the smallest duration between transitions that enables the timing or symbol value to be properly detected by the receiver. In this sense, meeting some minimum time between transitions may also be desirable. Enforcing a minimum run of symbols before a transition, a maximum run of symbols before a transition, or both, is the responsibility of a runlength limited (RLL) code. These codes are common in magnetic and optical storage systems, but are equally applicable to any asynchronous communication system.

Frequently, a CSC that satisfies both a DC-free and a RLL constraint is required. For this reason, these are the two types of constrained sequence codes most often seen in literature. Other constraints are possible, such as peak-to-average power ratio (PAPR) constraints in OFDM modulated communication systems.

1.3 Runlength-limited Codes

Runlength-limited codes are a key area of research. RLL codes are typically referred to as (d, k) codes. A (d, k) code has a minimum runlength of $d + 1$ consecutive symbols and a maximum runlength of $k + 1$ consecutive symbols. This, somewhat odd, notation arises from the historic construction of (d, k) codes using differentially encoded (dk) sequences. A (dk)-sequence consists of a 1 followed by at least d 0's, and at most k 0's. Here, the differential encoder transitions whenever the input is a 1, and maintains its state whenever the input is a 0. It can be shown that the capacities of differentially encoded (dk) sequences and (d, k) sequences are the same. As a result, it is common in the literature to only discuss the construction of (dk) sequences when discussing RLL codes. However, recent studies of RLL block codes have found that although (dk) sequences can be used to construct RLL codes, the technique is not necessarily the most efficient method of doing so in terms of resources or error-propagation [8]. As a result, in the design of efficient RLL codes, it may be desirable to consider RLL sequences directly, rather than first constructing (dk) sequences. This work proposes an alternative approach that utilizes properties of variable-length sequences to reduce the resource requirements of implementing (d, k) codes.

A minimum runlength represents the minimum number of consecutive identical digits in a digital sequence. Consider a system where there must be at least two consecutive identical binary digits in a row, but no upper-limit on the number of consecutive like-valued symbols. According to the nomenclature discussed above, such a code is known as a $(1, \infty)$ code. This restriction results in a capacity of less than one bit of information per binary digit, since there are now constraints on the types of allowable sequences.

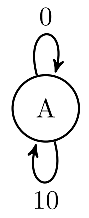

Figure 1.4: Single-state $(1, \infty)$ constraint graph

To determine the capacity of this constraint, consider the second approach discussed in §1.1. A state machine modeling the Markov chain for a $(1, \infty)$ constraint is depicted in Figure 1.4. This constraint can be represented by a state machine with a single state. In this state, a codeword is emitted with a length of either 1 or 2. As a result, the state-transition matrix as given by (1.5) is:

$$D(z) = \left[z^{-2} + z^{-1}\right] \tag{1.8}$$

The roots of the characteristic equation given by (1.6) are $(1 + \sqrt{5})/2$ and $(1 - \sqrt{5})/2$. The capacity of the constraint is simply the logarithm (base 2) of the largest real root:

$$C_{(1,\infty)} = \log_2 \left(\frac{1 + \sqrt{5}}{2} \right) \tag{1.9}$$

This results in a per digit capacity of approximately 0.6942 bits/bindig.

A minimum runlength in storage media allows the minimum on-disc feature size to be the same, but the effective bit-width to decrease. For instance, consider an example where the minimum feature size for an optical media is T_{min}. In an uncoded system, a single binary digit must be at least T_{min} long. The capacity of this uncoded system is 1 bit/symbol period. Consider what is gained by exploiting the properties of the code and halving the effective bit width of the data on the media. In the same symbol period of T_{min} we can now represent 2 binary digits instead of 1, albeit with the capacity of each binary digit reduced to 0.6942 bits/bindig. The net information stored in one symbol period for the $(1, \infty)$ constraint is:

$$2 \frac{\text{bindig}}{\text{symbol period}} * 0.6942 \frac{\text{bits}}{\text{bindig}} = 1.3884 \text{ bits/symbol period.} \tag{1.10}$$

This represents a 38.8% improvement over the original uncoded system. Even accounting for sub-capacity encoding, the minimum runlength constraint has increased the storage density of the device, at the expense of coding complexity/overhead and increased sensitivity to noise. Note that a single error in detection would represent an error in at least two binary digits instead of one, since the minimum detectable feature now represents two binary digits.

Note that this discussion assumes that the detection threshold is "non-linear" in the sense that although the detector could not correctly detect a single bit of size $T_{min}/2$, it *can* detect the difference between two consecutive identical digits (size T_{min}) and three consecutive identical digits (size $3T_{min}/2$). This is indeed the case for optical and magnetic storage devices using low-order codes where the effective bit-width is only 1/2 or 1/3 the minimum feature size. This process has been extended to an arbitrary value for minimum runlength in Figure 1.5. This type of implementation is possible because the CSC guarantees that there will always be at least $d + 1$ identical consecutive bits in the coded sequence, and therefore it will not violate the minimum feature size of the media.

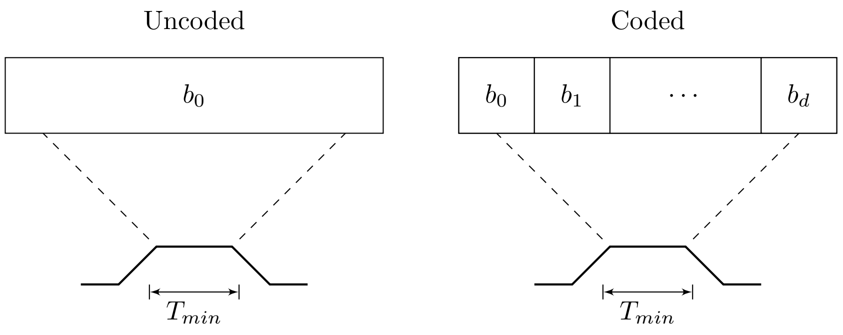

Figure 1.5: The use of a CSC has allowed the effective bit-width of the media to be reduced, without affecting the minimum feature size constraint

A maximum runlength assists in clock recovery at the receiver. Without a maximum runlength constraint, a receiver may incorrectly determine that a long string of X consecutive digits was actually $X \pm 1$ digits, due to timing inaccuracy and insufficient transitions to synchronize to the clock generator. If the hardware in question is guaranteed to correctly detect at least $k + 1$ consecutive digits without timing errors, then a $(0, k)$ code will ensure that the receiver never encounters such an error. Many systems use a combination of minimum and maximum runlength constraints.

It is important to note that the minimum runlength places a greater restriction on the resulting capacity than the maximum runlength. For example, a $(0, 4)$ code has a capacity of 0.9752 bits/bindig, whereas a $(4, \infty)$ code has a capacity of 0.4057 bits/bindig.

1.4 DC-free Codes

DC-free codes are important because they eliminate problems such as baseline wander in AC-coupled systems. Basic DC-free code construction is based on a simple observation: there must be a bound on the amount of charge allowed to accumulate, as explained below. It is typical to map binary sequences into sequences with positive "charge", $+1$, and negative "charge", -1. This can be accomplished by mapping logic 0's to -1 and logic 1's to $+1$. Denote the ith symbol in this sequence as x_i, where $x_i \in \{-1, 1\}$. The running digital sum (RDS) of this sequence, denoted by z_i, is defined as the accumulation of charge values up to symbol x_i [9]:

$$z_i = \sum_{j=-\infty}^{i} x_j. \tag{1.11}$$

The DC-free constraint translates directly into a bound on the RDS: the sequence is DC-free if and only if a bound exists on the RDS [10]. If the sequences satisfies such a bound and the RDS takes only values from a finite set, then it is possible to identify the total number of different RDS values, denoted by N:

$$N = \max(z_i) - \min(z_i) + 1. \tag{1.12}$$

It is important to note that the initial value of z_i is entirely arbitrary, for convenience it can be viewed as 0, but the definition of N and the DC-free properties of the resulting code are equally applicable for any initial value. If a Markov source is constructed, with the states corresponding to allowed values of z_i, and transitions between the states governed by the output symbols and corresponding changes to z_i, the capacity of DC-free codes can be determined as a function of N. This is possible because the Markov chains have a simple and predictable structure as a function of N. A detailed analysis can be found in [1], where it is shown that for DC-free sequences:

$$C(N) = \log_2 \left(2 \cos \frac{\pi}{N+1} \right), \quad N \geq 3 \tag{1.13}$$

Clearly, as N increases, the capacity of the DC-free constraint approaches 1. For $N = 10$, the resulting capacity is already as high as 0.94, representing a fairly minimal loss in rate.

Techniques are available for determining the maxentropic power spectral density of RDS-constrained channels. A basic outline of the technique is available in [1]. For a given power spectrum, the cut-off frequency can be defined as the frequency ω_0 where the power is

1/2 (-3dB). For maxentropic RDS-constrained codes, there is an approximately inverse proportional relationship between the cut-off frequency and the so called sum-variance of the code, s_z^2 [11], where:

$$s_z^2 = \mathbb{E}\big[z_i^2\big]. \tag{1.14}$$

For maxentropic RDS-constrained sequences, this relationship implies that if N is increased in order to increase $C(N)$, then a price is payed in terms of the bandwidth of the spectral null. If a larger suppressed bandwidth is desired without decreasing N, then additional constraints can be applied to s_z^2 such that the approximate relationship between s_z^2 and ω_0 no longer holds. These constraints place bounds not only on the RDS, but also on the running digital sum of the running digital sum, the so-called RDSS. These constraints can be extended in a similar fashion indefinitely. As a result, for the same value of N, we can achieve superior spectral suppression of low-frequency components. There is still a price to pay, however, as the additional constraints imposed by this procedure provide fewer allowable sequences, and therefore also reduce the capacity. As a result, these more complicated codes are only used in applications with strict bandwidth restrictions.

The RDS constraint ensures that the mean value of the auto-correlation of x_i is zero, and hence the PSD of x_i has no DC-component. As a result, there is no appreciable accumulation of charge over time. The RDSS constraint ensures that not only is their no appreciable accumulation of charge in x_i over time, but also that the auto-correlation of the RDS has zero mean as well. Similarly, a bound on the RDSSS ensures that there is no statistical accumulation of charge in the RDSS over time, and so on. Since integration is a low-pass filter, and summing is discrete-time integration, then these higher order constraints ensure that the various low-frequency components (those passing through successive low-pass filters) have zero contribution to the power spectral density.

The simplest method of constructing DC-free codes is to use zero-disparity codewords. A zero-disparity codeword has no net contribution to the RDS. A kth order zero-disparity codeword has no net contribution to the k-th sum of the RDS, as well as all lower-order sums of the RDSs. Zero-disparity codewords can be concatenated statelessly. If non-zero-disparity codewords are to be used, then the encoder must keep track of the kth order (and below) running digital sums and ensure that all the running digital sums are bounded.

1.5 Block Codes

There are many possible ways to implement codes. A simple approach is to take in m binary digits and output n binary digits. A code that operates on fixed block lengths m and n is known as a block code. Block codes which always output the same n binary digits for the same m input bits are known as state-independently encoded block codes. If they output a different n binary digits depending on the internal state of the encoder, then they are known as state-dependently encoded block codes. Likewise, a decoder may always map the same n binary digits back to the same m bits, known as state-independent decoding, or it too may attempt to keep track of an internal state. It is desirable for block codes to be constructed with state-dependent encoders, which result in (generally) higher rate codes, but state-independent decoders, which limit error propagation because it is not possible for errors to occur as the result of an incorrect assessment of the current state.

The rate (R) of a block code is solely dependent on the block sizes m and n:

$$R = \frac{m}{n}. \tag{1.15}$$

The rate of the code does not depend on any other factors, such as source probabilities or encoder state. Constrained sequence codes are often nonlinear, and when m and n permit, they are typically implemented with a look-up table (LUT).

Construction of Block Codes

There is no generalized technique to construct CSCs using a block code. For RLL codes, Immink [1] has outlined an entire chapter on various construction techniques for RLL block codes. The basic technique involves selecting a input codeword length, m, and the constraint (d, k). From there, output sequences are selected along with merging bits according to a set of merging rules, which dictates the code rate m/n. Of importance is the use of Franaszek's recursive elimination procedure to construct so called *optimal* block codes [12]. These codes are optimal in the sense that they maximize the number of messages that can be transmitted, and not in the sense of maximizing the code rate.

For other constraints, construction techniques vary. Typically, constraint satisfying sequences are identified and mapped to input sequences. This can be accomplished through look-up tables or enumerative encoding [1]. Various other block coding techniques have also been proposed [13, 14, 15].

1.6 Variable-length Codes

Variable-length (VL) codes have a very different structure than block codes. In the general case, a VL code converts an arbitrary number of input digits into an arbitrary number of output digits, according to some predetermined scheme (the code). Like block codes, they can be state-dependent or state-independent. For simplicity, the work in this thesis only considers state-independent encoding and decoding of variable-length codes.

Unlike block-codes, VL codes do not have a fixed rate. Their average rate depends on the statistics of the input sequences:

$$R_{avg} = \frac{\mathbb{E}[L_{in}]}{\mathbb{E}[L_{out}]}, \tag{1.16}$$

where L_{in} denotes the set of input lengths, L_{out} represents the set of corresponding output lengths, and $\mathbb{E}[X]$ represents the expected value of a random variable taking values from set X. Obviously, the code rate is bounded by the limits on code rates associated with individual codewords.

$$\begin{aligned} R_{min} &= \min_i \left(\frac{L_{in,i}}{L_{out,i}} \right) \\ R_{max} &= \max_i \left(\frac{L_{in,i}}{L_{out,i}} \right) \end{aligned} \tag{1.17}$$

It is important to note that it is possible for R_{max} to exceed the capacity of the constraint, but that the average code rate, as sequence lengths tend toward infinity, will still be less than capacity for a maxentropic source and any code that satisfies the constraint. If the input sequence is not maxentropic then the information content of the input stream is below 1 bit/bindig, and the product of the input entropy and R_{avg} will still be below capacity, even if R_{avg} is above capacity. This is because the information content of the output sequence cannot exceed capacity by definition [5]. For this reason, it is typically assumed that the input sequence is maxentropic. If it is not maxentropic, then the CSC can be preceded by a source code that compresses the binary input sequence such that it has an entropy close to 1 bit/bindig. Such an assumption is typical in CSC design.

Construction of VL codes

The construction of VL codes is the focus of this thesis. Previous work in this area includes Franaszek's [12] *synchronous variable-length codes*, the variable-length bit stuffing algorithm proposed by Bender and Wolf [16], and the extended *bit-flip* [17] and *symbol-sliding* [18] techniques. Previous researchers have also noted the duality between source coding and constrained sequence coding, and examined the use of source codes, some of which are variable length, in the construction of CSCs [19, 20, 21]. A detailed analysis of these various approaches to VL constrained sequence code design is discussed in Chapter 2.

Some authors consider the variable nature of the code rate in VL systems to be undesirable, largely due to the block-oriented nature of some digital storage systems. Without a fixed code rate, it cannot be determined in advance how long the output sequence will be. Also, a padding scheme must be developed in order to complete the encoding procedure, as the input sequence may not align along the sourceword boundaries. The first issue can be addressed by understanding the bounds on code rate (1.17), and properly conditioning the input sequences using source coding or scrambling to mitigate the effect of non-maxentropic inputs on code rate. The second issue is an unfair criticism, as block-oriented systems still use a padding scheme in order to deal with input sequences of arbitrary length that also do not align along sourceword boundaries.

1.7 Example of block code vs VL code design

VL codes provide an extra degree of freedom not available in block codes, and are particularly well suited to some constraints. Consider the $(1, \infty)$ code whose capacity was derived in (1.9). The value of capacity in this case is an irrational number. Indeed, it has been shown that the capacity of binary (d, k) codes is always irrational except for one trivial case [22]. Such a property makes it clear that all block codes, which have a fixed rate equal to the rational number $R = m/n$, are sub-optimal for (d, k) constraints. For maxentropic binary input sequences, the rate of a VL code (see (1.16)) has a numerator and denominator that are both linear combinations of powers of 1/2, resulting in a rational, and therefore also sub-optimal, code rate. However, as shown in the following example, because VL codes can exploit input sequence probabilities, the implementation of a VL CSC can potentially be much simpler than an equivalent rate block code.

Continuing the example of a $(1, \infty)$ code, a possible implementation of a stateless block code based on the discussion in [1] is examined. Values of m and n have been restricted to numbers less than 50, an arbitrary number representing a reasonable upper limit of what can be expected in terms of implementing a look-up table with current memory technology (not necessarily practically). In Table 1.1, the analysis of [1] has been further restricted to hypothetical state-independent encoding and decoding of block codes with rates operating within 1% of capacity. Such analysis will be used to contrast with similar stateless VL look-up tables later on. The size of a stateless encoder look-up table has been estimated as $2^m * n$ bits, and the size of a stateless decoder look-up table has been estimated as $2^m * (m+n)$ bits. This approach allows the decoder to rapidly search through the 2^m different n bit allowed sequences and quickly identify the corresponding m bit word. Other implementations of look-up tables may vary in size. For analysis purposes the metric units gigabytes (GB) and kilobytes (kB) have been used, representing 1 billion and 1 thousand bytes respectively.

m	n	$(1 - R/C)\%$	Enc LUT	Dec LUT
34	49	0.0525	$> 105GB$	$> 178GB$
9	13	0.2786	$> 0.8kB$	$> 1.4kB$
11	16	0.9711	$> 4kB$	$> 6.9kB$

Table 1.1: Practical values of m and n such that $R/C \geq 0.99$ (based on Immink [1])

Clearly, a good “practical” choice would be the rate $R = 9/13$ block code that operates within 0.2786% of capacity. A hypothetical (not necessarily possible) stateless encoder would use a look-up table roughly 1kB in size. The decoder look-up table would be similar in size, depending on implementation. Such a design is very practical, and some $(1, k)$ codes do operate at this rate [23].

Now, consider a VL code implementation with an average rate given by (1.16). If the source being encoded is a maxentropic binary source, then input sequences of length l_i have a sequence probability of 2^{-l_i}. If the code outputs a constraint-satisfying sequence of length

o_i for this input, then the resulting code rate is given by:

$$R = \frac{\sum_i (2^{-l_i} * l_i)}{\sum_i (2^{-li} * o_i)}. \tag{1.18}$$

Notice that this equation has a numerator and a denominator that are linear combinations of powers of 1/2. Even though l_i and o_i are restricted to be integer values, the numerator and denominator of (1.18) can now take non-integer values. This is not possible with block codes. As a result, rates such as 2.25/3.25 and 4.5/6.5, both equivalent to a 9/13 block code, may now be possible. Indeed, early analysis of $(1, \infty)$ codes quickly revealed a simple VL code operating at the average rate 2.25/3.25. The codebook for this code is given in Table 1.2.

Input	Output
01	010
10	100
11	000
000	1010
001	0010

Table 1.2: A stateless VL $(1, \infty)$ code in differential form

The average code rate for this VL code is:

$$\begin{aligned} \bar{R} &= \frac{0.25(2+2+2)+0.125(3+3)}{0.25(3+3+3)+0.125(4+4)} = \frac{2.25}{3.25} \\ &= 9/13 \approx 0.6923. \end{aligned} \tag{1.19}$$

A common metric, used throughout this work, will be to identify, as a percentage, the distance that the average rate of a code is from capacity. In other words, at what percent *below* capacity does the code operate on average:

$$\left(1 - \frac{\bar{R}}{C}\right) * 100 = 0.2786\%. \tag{1.20}$$

Another commonly used metric is the efficiency of the code, which is simply the ratio of average code rate to capacity:

$$E = \frac{\bar{R}}{C}. \tag{1.21}$$

Implementation of a simple look-up table for the mappings in Table 1.2 can take on various forms. However, even allowing for practical concerns, the entire lookup table could easily fit within 8 bytes. This represents a 100 fold improvement over the implementation of the hypothetical stateless block code with equivalent rate presented earlier. Actual block code implementations would likely require several states and therefore would be several times larger than the size estimated here. Note that even if faced with non-maxentropic inputs, the code rate of Table 1.2 can be no worse than 2/3, which is still within 4% of capacity.

Consider again the example on page 8 regarding minimum runlength encoding for minimum feature size constraints. A capacity-achieving code would be a 38.8% improvement over the uncoded case. The code presented in Table 1.2 is on average 38.5% better, and at worst 33.3% better, than the uncoded case. A more detailed analysis of this code will be outlined in Chapter 4.

1.8 Thesis Outline

Clearly VL codes have the potential to provide improvements of several orders of magnitude over equivalent block code implementations. Equivalently, VL codes of a given LUT size can potentially achieve much higher rates than block codes of the same LUT size. The advantage of a simple, high-rate code may justify the use of variable-length, non-block oriented logic. The necessity of implementing a padding scheme in order to deal with arbitrary-length inputs applies equally to both block-codes and VL-codes, although their implementations will differ. While not necessarily suitable for all constrained-sequence code applications, this chapter has outlined several key motivators for considering variable-length code implementations.

In Chapter 2, existing research in the area of variable-length constrained sequence codes is examined in detail. In Chapter 3 various properties of VL CSCs are examined. These properties are used in Chapter 4 to demonstrate a new technique for VL code-construction, and where new, efficient, and easily implemented VL CSCs are also presented. Chapter 5 concludes the work, summarizes the key points, and identifies some of the open questions remaining to be studied.

Chapter 2

Existing Research

There are several possible techniques used to construct VL CSCs. This chapter will outline some of the most important contributions to date in the area. As discussed in Chapter 1, Shannon originally discussed the theory of variable-length code construction for constrained sequence codes. However, practical implementations of VL CSCs have only recently been studied, largely due to the perceived difficulty of dealing with variable-length inputs and outputs. Research developments in this area outlined below.

The duality between source coding and constrained sequence coding was identified by Martin et al.[20]. Given the duality, many researchers have used variable-length techniques from source coding in order to construct VL CSCs [2, 21]. To mitigate some of the issues in dealing with variable-length codes, Franaszek introduced synchronous variable-length codes [12, 3]. These codes have variable-length inputs and outputs, however the ratio of input length to output length for each codeword pair has been fixed, resulting in a fixed-rate code. More recently, the bit-stuffing [16], bit-flipping [17], and symbol-sliding [18] techniques have been introduced. Each of these topics is discussed in this chapter.

2.1 Coding Duality

Since both source codes and constrained sequence codes are based on Shannon's noiseless source coding theorem, it is natural to question whether there is a relationship between them. One of the first papers to make explicit the duality of the two techniques was by Martin et al.[20]. Within the context of arithmetic coding, the authors discuss how the encoder of a source code implements a compression operation, and how the decoder implements an expansion operation. In contrast, the encoder of a constrained sequence code implements an expansion operation, while the decoder implements a compression operation. A comparison of the two approaches is outlined in Figure 2.1.

As discussed in Chapter 1, constraints on the channel necessarily introduce redundancy. In constrained-sequence coding, arbitrary sequences with little or no redundancy are ex-

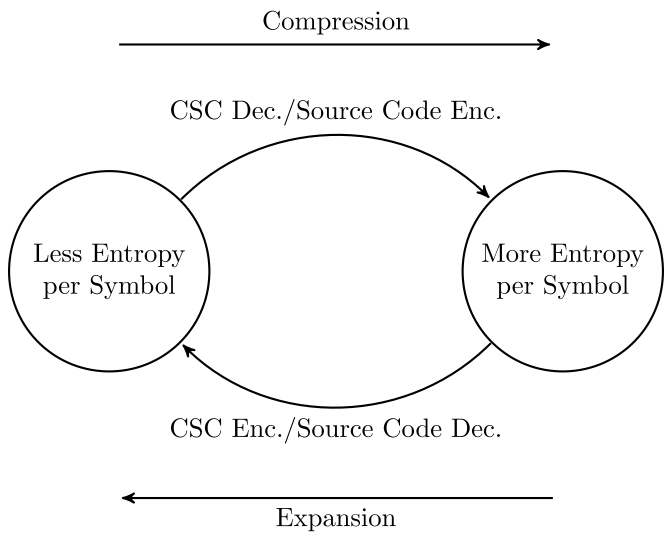

Figure 2.1: Coding Duality

panded into constraint-satisfying sequences with intentional and well understood redundancy. In source coding, sequences with redundancy are compressed, based on the statistical properties of the input sequence, into sequences with little or no redundancy. Therefore, if we construct a code in the "compression" direction, then code construction follows a source coding paradigm. On the other hand, if we construct a code in the "expansion" direction, then code construction follows a constrained-sequence coding paradigm. Clearly the goals of these two techniques are the related, however the roles of encoding and decoding are reversed. Of course, it should be understood that there is no direct relationship between the inputs of source codes, which are typically unconstrained, and the outputs of constrained sequence codes, which are by definition constrained.

When source coding techniques are used to construct constrained sequence codes, the following general procedure is followed. The "source" is a list of sequences that can be generated by a Markov information source satisfying the constraint in question (see Chapter 1). For state-independent operation, this list must contain only sequences that may be concatenated in any combination without violating the constraint. One technique for ensuring this property is to enumerate only the sequences that return to the same initial state in the Markov chain. For example, a $(1, \infty)$ Markov information source could generate the sequences "10" and "0". Clearly, these sequences can be concatenated in any order without violating the $(1, \infty)$ constraint. Extensions of the source enumerate possible combinations of these sequences. For example, a complete first order extension of the source enumerates all possible combinations of these two base sequences: "1010","100","010", and "00". Extensions of the source will be discussed in more detail in Chapter 3.

Once the source sequences have been identified, it is necessarily to determine their corresponding maxentropic probability distributions. This can be determined by analysis of the Markov chain, or directly using techniques outlined in Chapter 3. Once the sequences and their corresponding probabilities are identified, classical source coding techniques are used, with subtle differences. In this instance there is an additional constraint not necessarily present in source coding (although generally desired): the source code must generate a complete code. In a complete code the set of input sequences is exhaustive, which also implies that the Kraft inequality holds with strict equality. For example, using the symbols "0" and "1" as the base set, the set of "10", "01" and "11" is not complete, as the symbols "1" and "0" could also be combined to form the sequence "00". A more detailed discussion of what is meant by a complete code will be discussed in Chapter 3. In general, in source coding it is not necessary that the code generated be complete, however with CSC coding it *is* required since the code will eventually be used in "reverse" and must accept any arbitrary input.

Moreover, the unique decodability requirement is also reversed. A constrained sequence decoder must still map channel bits uniquely back to unconstrained bits, however a sequence of unconstrained bits could potentially map to one of several sequences of channel bits, provided the channel bits have not been assigned to any other set of input sequences. This reversal of constraints is due to the source code / constrained sequence code duality and is discussed by Martin et al. [20]. Once the source coding technique has been applied, the generated mappings are used in reverse to construct the constrained sequence encoder.

2.2 Source Coding applied to CSCs

Since the use of source coding techniques to construct constrained sequence codes was proposed by Martin et al. [20], numerous researchers have developed similar approaches. The original authors used the technique of arithmetic coding to construct fixed-rate constrained sequence codes. Some of the issues addressed by these authors include the problems associated with finite precision arithmetic and approximating probability values. In a subsequent work [19], they propose a general fixed rate arithmetic coding method for constrained channels. However, these codes are fixed-rate and they do not exploit the variable-length (and variable-rate) nature of the underlying technique to the fullest extent possible.

Kerpez did a great deal of work related to adapting source codes for use as run-length codes [2]. Among other contributions, he proved that various source coding techniques have average rates that converge to capacity in the limit as the number of source extensions tends toward infinity. Separate proofs are necessary for fixed and variable length code constructions. His work notes that the inverse of a Huffman code is a prefix-free, complete (d, k) code that is well suited to a constrained sequence code. He showed that, by utilizing the maxentropic probability distribution of the constrained sequences and merging the least two probable sequences during each merging operation, it is possible to create a code which has an average rate that converges toward capacity as the number of source extensions increases. Using the Huffman coding technique, Kerpez presents two simple $(0, 2)$ codes whose rates approach 97% and 98% of capacity. The higher rate code is listed in Table 2.1. This code was constructed using a second order complete extension of the source. A modification of this technique will be discussed in Chapter 4, and some of the new results developed in this thesis will be contrasted with this code.

Input	Output
00	11
011	101
100	011
010	1001
101	0101
110	0011
1110	01001
11110	00101
11111	001001

Table 2.1: A stateless VL $(0, 2)$ code in differential form constructed by Kerpez [2] using a Huffman source code technique

Kerpez uses a similar proof to justify that enumerative (d, k) codes also approach capacity with increasing block length. His enumeration technique is based on defining a number, of length m, to each of the permutations of typical runs in length n sequences based on the number of times a run of length j occurs.

Kerpez also discusses a code construction he refers to as block-to-variable-length source codes, which he describes as the dual of Huffman coding [2]. Other authors refer to this technique as Tunstall coding [24, 25]. The source extensions continue along the highest probability sequences until there are at least 2^m codewords for some desired value of m. If there are any extra sequences, then the longest ones are ignored. Each of the 2^m sequences are then assigned to a unique m bit source word. Once again, Kerpez presents a proof that shows that as the number of extensions tends to infinity, the average rate converges to capacity.

Finally, Kerpez concludes with a novel technique that simultaneously performs source coding and (d, k) coding, while once again proving that the rate of the technique converges toward capacity for input sequence lengths approaching infinity. This technique is based on the assumption of a Bernoulli source with non-equiprobable outputs, and is inherently variable-length to variable-length. Kerpez does not, however, discuss any of the issues associated with implementing the technique, such as finite-precision arithmetic, code complexity, or error propagation.

In a closely related work, Arikan [21] also discusses Elias (Arithmetic) coding. He presents a variable-to-fixed length implementation that makes a trade-off between coding complexity and code-rate. When applied to a $(2, 7)$ run-length constraint, the technique achieves an experimental average code rate within 0.23% of capacity. The author shows that in order to achieve higher rates, the same algorithm may simply be run using higher precision. Unfortunately, this technique suffers from catastrophic error propagation, as do several other VL techniques, and no elegant technique is presented to address the issue.

2.3 Synchronous Variable-Length codes

Some of the perceived drawbacks associated with the variable-length nature of VL codes are mitigated through the use of synchronous variable-length codes. A synchronous variable-length code has variable-length input and output codewords, but the rate of the code is fixed. For instance, consider a code with a desired rate of $R = m/n$, where m and n are integers. Given this rate R, the source words are of length o_iR and the corresponding codewords are of length o_i. Here both the output codeword length and the input codeword length are restricted to be integer: $\{o_i, o_iR\} \in \mathbb{Z}$. For a source where the probability of sourceword i is denoted by p_i, the average rate of the VL code is:

$$\begin{aligned} R_{avg} &= \frac{\sum_i (p_i o_i R)}{\sum_i (p_i o_i)} \\ &= R\left(\frac{\sum_i p_i o_i}{\sum_i p_i o_i}\right) \\ &= R \end{aligned} \tag{2.1}$$

A technique for the construction of synchronous variable-length codes was developed by Franaszek[12, 3]. Fittingly, the existence of a synchronous variable-length code with a set of desired parameters can be determined based on a modification of Franaszek's recursive elimination procedure for block codes [1]. As in the other VL codes discussed above, the use of prefix-free output codewords is generally desired as it simplifies decoder operation.

Unfortunately, although the rate of synchronous variable-length codes is fixed, some of the issues associated with VL codes still remain. For example, a padding scheme still needs to be devised in order to pad the input sequences up to a required length for the encoder. Also, if the underlying system is frame or block oriented, then padding on the output sequences may also be necessarily. Another disadvantage of this approach is that, although the codewords are variable length, the additional requirement of a constant rate effectively negates one of the significant advantages of VL codes outlined in Chapter 1: the ability to exploit non-integer average input and output codeword lengths to approach capacity. The main advantage of this technique over block codes is that due to its VL nature, it may provide a simpler implementation for some constraints than block codes.

In order to provide some insight into the properties of synchronous VL codes, Immink has outlined an alternate construction technique based on $(dklr)$ sequences [1], as opposed to Franaszek's recursive procedure for designing block codes. A $(dklr)$ sequence is a dk-sequence with at most l leading zeros and at most r trailing zeros. Clearly, l and r must be chosen such that $k = l + r$. In Immink's discussion, l and r are chosen by trial and error. To construct the code, he suggests writing down all $(dklr)$ sequences of length n, $2n$, $3n$, $\ldots$, but removing those that start with a codeword that has already been written, and proceeding with this technique until the corresponding set of input lengths is exhaustive (ie: exactly satisfies the Kraft equality). If there are not enough codewords, the procedure should be repeated with a different value of R, l, or r.

In a patent, Franaszek outlines the $R = 1/2$ $(2, 7)$ synchronous VL code in Table 2.2 [3]. However, subsequent research has identified that this particular mapping is not optimal in

Input	Output
10	1000
11	0100
011	000100
010	001000
000	100100
0011	00100100
0010	00001000

Table 2.2: Franaszek's $(2, 7)$ synchronous VL code [3]

terms of error propagation. The details of error propagation for this code have been given by Howe and Hilden [26]. An alternative $(2, 7)$ code with reduced error propagation has been proposed by Eggenberger & Hodges [27]. Unfortunately, it is still not fully understood how codewords should be assigned in order to reduce error propagation in synchronous variable-length codes. Similarly, optimal mappings for other synchronous variable-length codes are not well understood.

2.4 Bit stuffing, flipping, and sliding

Most recent research in the area of variable-length constrained sequence coding is derived from 'a universal algorithm for generating optimal and nearly optimal run-length limited, charge-constrained binary sequences' that was proposed by Bender and Wolf [16]. This paper presented a new universal technique for encoding and decoding joint DC-free and RLL CSCs known as bit-stuffing. Two modifications to the technique have also been proposed: bit-flipping by Aviran et al. [17]; and symbol-sliding by Sankarasubramaniam and McLaughlin [18].

Bit-stuffing

The bit-stuffing algorithm is very simple. It is specified in terms of a (d, k, c) constraint, where d and k represent the parameters of a (d, k) constraint, and c represents the absolute value of the maximum allowable accumulation of charge for a DC-free constraint ($N = 2c - 1$). The bit-stuffing encoder directly translates input sequences to output sequences, but keeps track of two variables: the current run-length, and the current accumulation of charge. If the run-length constraint is about to be violated, then a '1' is inserted. Likewise, if the charge constraint is about to be violated, then a '1' is also inserted. Recall that in this notation, a '1' represents a transition and a '0' represents a repetition of the previous symbol. After every '1' in the output sequence, d zeros are also inserted in order to prevent violation of the d constraint. The decoder operates in a similar manor, keeping track of both the current accumulation of charge, and the run-length constraints, and using this information to remove the extra bits inserted by the encoder. Although not discussed in the original paper, the use of a state-dependent decoder could cause significant error propagation issues since errors can cause the decoder to incorrectly determine the current runlength and accumulation of charge, and therefore remove data bits instead of extraneous channel bits.

In order to achieve high data rates with this technique, it is necessary to first transform the incoming data so that it has appropriate symbol probabilities. For purposes of analysis, the input to the encoder is modeled as an independent Bernoulli source with symbol probabilities p and $1 - p$ for zero and one respectively. In practice, such a distribution would be approximated by a "distribution transformer" that biases the probabilities of the source symbols to the appropriate probability distribution. By maximizing the average information rate of the code over all possible probability distributions, Bender and Wolf [16] were able to show that the maximum average rate of the code is exactly equal to capacity for the (d, ∞, ∞), $(d, d+1, \infty)$, and $(2c-2, \infty, c)$ constraints. For all other constraints, the encoder does not produce sequence lengths aligned with the maxentropic distributions, resulting in sub-optimal information rates. However, numerical analysis showed that the remaining constraints are "nearly" optimal.

One of the key practical issues with this technique is implementing the necessary distribution transformer. This transformer must output a desired probability distribution without introducing a rate loss. A failure to achieve the desired distribution also translates into a rate loss. At the decoder side, the operation of this distribution transformer must also be uniquely reversible: a given output sequence must map uniquely back to a specific input sequence. Additionally, it is desired that the error-propagation of the reverse distribution transformer be limited, or else the coding technique suffers from further amplification of any errors that may occur. Therefore, while "optimal" results can be achieved very simply by the bit-stuffing technique, most of the difficulties in implementing the constrained sequence code have simply been transformed into corresponding difficulties in implementing the necessary distribution transformer, the design and operation of which are ignored by the original authors.

Bit-flipping

Shortly after the bit-stuffing algorithm was introduced, an improvement to the algorithm was found for (d, k) constrained sequences [17]. This new technique is known as bit-flipping, and it can achieve higher maximum average rates than the bit stuffing algorithm for most constraints where $d > 1$ and $d+2 \leq k \leq \infty$. This technique defines a new parameter, l, such that $d + 1 \leq l \leq k - 1$. For run-lengths smaller than l, continue as usual. For run-lengths greater than or equal to l, flip the next bit. Analysis is used to determine the optimal values for l, and the authors demonstrate that the resultant bit-flipping rate is often greater than the corresponding bit-stuffing rate for most (d, k) constraints. Although analytical results show that this technique is an improvement to standard bit-stuffing, it suffers from the same drawbacks as bit-stuffing in terms of error propagation during decoding and the necessity of a separate probability transformer.

Symbol-sliding

In subsequent work by Sankarasubramaniam and McLaughlin [18], the above two techniques are summarized and examined in a different context. First, the use of source codes to construct RLL codes, as discussed in §2.2, is understood to be a distribution transformer from an i.i.d Bernoulli(1/2) source to a higher-order alphabet with the appropriate maxentropic distribution, denoted as $\mathbf{\Lambda}_{d,k}$. This paper then summarizes that bit-stuffing has been shown to be optimal for (d, ∞) and $(d, d+1)$ RLL constraints. Bit-flipping on the other hand, has been shown to improve upon bit-stuffing for many constraints, as well as achieve capacity for the $(2, 4)$ constraint. These two techniques are then discussed in the context of aligning the bit-stuff phrase probability with the maxentropic output distribution. The authors note that the bit-flipping algorithm serves to exchange the first two phrase probabilities (when ordered by increasing maxentropic probability) in order to better match $\mathbf{\Lambda}_{d,k}$. The authors then go on to show that instead of swapping, a better approximation of $\mathbf{\Lambda}_{d,k}$ can be obtained if the first phrase probability is placed in position j, and positions two through j are "slid" up the list of assigned phrase probabilities. In this context, bit-stuffing is symbol-sliding with j set as the first position, and bit-flipping is symbol-sliding with j set as the second position. Since bit-stuffing and bit-flipping are special cases of symbol-sliding, the new technique shares the same proofs of optimality for the cases where bit-stuffing and bit-flipping are optimal. Additionally, the new technique also achieves capacity for the class of $(d, 2d + 1)$ constraints, the proof of which recognizes that the capacity of a $(d, 2d + 1)$ constraint is equal to the capacity of a $(d + 1, \infty)$ constraint [22].

2.5 Other Work

In addition to the symbol-sliding algorithm, Sankarasubramaniam and McLaughlin also introduce a second algorithm based on interleaving multiple biased bit streams, which is optimal for all $(d, d+2^m-1)$ constraints, where $2 \leq m < \infty$, and can be implemented with m distribution transformers [18]. The same authors have also implemented a unique fixed-rate bit stuff (FRB) algorithm for maximum-run-length-limited constraints. This technique first preprocesses the input iteratively, then bit-stuffs, and finally appends padding to achieve a fixed output length. A fairly detailed analysis of this technique is outlined in [28].

One of the most recent results in the field was reported by Aviran et al. [24]. This research could be considered a logical extension of bit-stuffing, bit-flipping, and symbol-sliding. As with the previous techniques, the authors consider p-biased source sequences, however here they consider a general form of the problem where the biased sequences are parsed into words, and the words are mapped to channel symbols. The mapping of p-biased sequences to channel symbols can be implemented by a parsing-tree. The result is a more generalized analysis of the underlying motivation for symbol-sliding: a method to generate a better match of the maxentropic probability distribution, $\mathbf{\Lambda}_{d,k}$, from biased data. Joint optimization of the tree-structure and p is possible. Some trends for small $k-d$ values are noted: first, symbol-sliding is a sub-optimal form of all of the available parsing trees in many cases; second, as d increases, the optimal parsing tree appears to converge to a specific tree; finally, for a fixed bias p and a fixed value of $k-d$, and for sufficiently large d, the optimal (d,k)-code corresponds to a Tunstall tree. Many open questions related to this research remain. Are the convergent trees asymptotically optimal? From what value of d do the trees converge? Is there a general algorithm for deriving the optimal tree given a bias and a particular (d,k) constraint?

2.6 Conclusion

While there are several examples in the literature of variable-length techniques being used to generate constrained sequence codes, they are based on one of the three basic techniques: directly approximating a maxentropic probability distribution $\mathbf{\Lambda}_{d,k}$ using a reverse source code, synchronous variable-length codes based on Franaszek's recursive procedure, or biasing the input data using a distribution transformer and either assigning channel symbols on the fly (bit-stuffing and bit-flipping) or according to some predetermined scheme (symbol-sliding and parsing trees). The first technique has been shown to be optimal in the asymptotic sense, as the number of source-extensions tends to infinity. The second technique is strictly sub-optimal for all constraints where the capacity is irrational, as the code rate is fixed to be a rational number. In the third technique, the purpose of the distribution transformer is to better align the source distribution with the maxentropic distribution, but designing a practical distribution transformer suffers from many of the same caveats as designing a constrained sequence code in the first place. Since distribution transformers can be implemented with reverse source codes, improvements in the reverse source code techniques may directly translate into improved implementations of any of the distribution transformer based techniques. In Chapter 3, a modification of the source coding technique will be discussed, and examples in Chapter 4 will demonstrate some of the results.

Chapter 3

Analysis of VL Constrained Sequence Codes

This chapter outlines various properties of variable-length sets of codewords, such as the capacity and the maxentropic distribution of the set. Also, an optimal technique for mapping sourcewords to codewords is discussed. These properties will serve as the basis for the code construction technique outlined in Chapter 4.

3.1 Properties of Variable-Length Sets

The technique used for constrained sequence code design in this thesis relies heavily on several fundamental properties of variable-length sets which are discussed below. An overview of these properties may also be found in other related sources, however the definitions and terminology used vary from source to source. The properties discussed here relate specifically to the constrained sequence code design proposed in Chapter 4.

Minimal Set

The minimal set for a given constraint is a set of constraint-satisfying variable-length sequences that can be concatenated together in any order without violating the constraint. Minimal sets will be used as the base set of codewords from which a constrained sequence code will be constructed. Often minimal sets are chosen to fully encompass all possible constraint-satisfying sequences for a given constraint, but this is not a requirement. One method for obtaining a minimal set is to select an initial state from the constraint graph and enumerate the output sequences corresponding to all of the possible paths that return to the initial state. Because of the structure of the constraint graphs for (d, k) codes, a minimal set can be constructed for these constraints that completely represents the desired constraint. A constraint is completely represented by a minimal set if it is possible to generate all valid constraint satisfying sequences by concatenating the elements of the minimal set. Other constraints, such as DC-free RLL constraints, may require the implementation of a state machine and several minimal sets, one set for each state. A constraint graph for a

$(d, k, N) = (1, 3, 5)$ DC-free RLL code is given in Figure 3.1. It is the presence of loops in this constraint graph that requires the use of multiple states. The construction of the code trees for these constraints will not be discussed in this thesis, but follows logically as an extension of the technique presented here into multiple states.

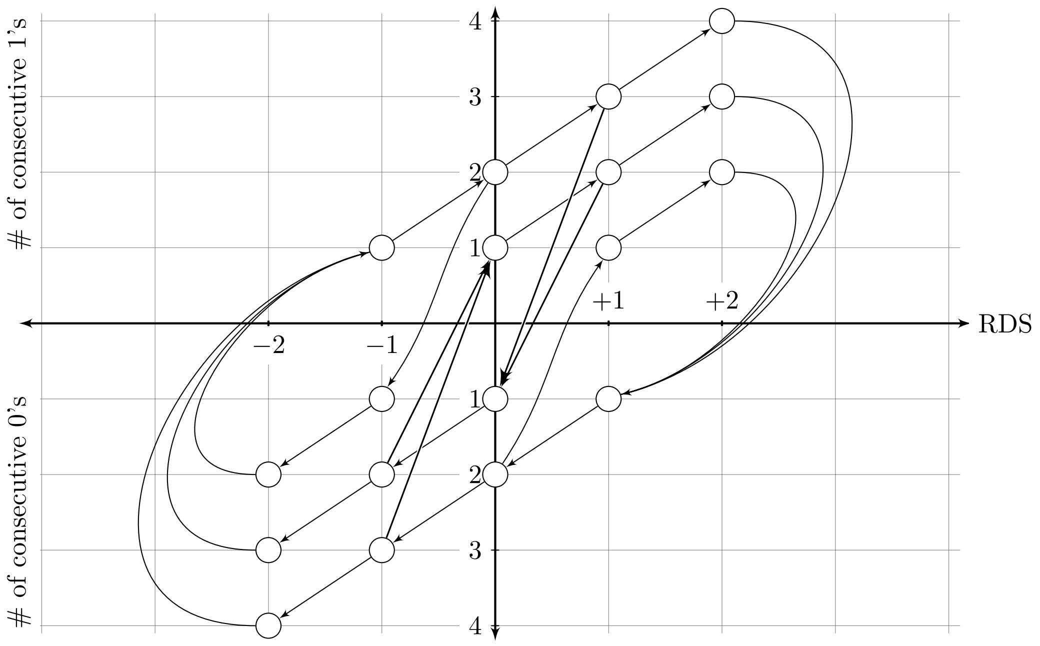

Figure 3.1: $(1, 3, 5)$ DC-free RLL constraint graph with loops

As an example of a minimal set, consider the $(1, \infty)$ constraint. For this constraint, every '1' must be followed by at least a single '0'. One possible minimal set for this constraint is {'10', '0'}. However, {'01','0'} is another possible minimal set, and therefore minimal sets are not unique. Either solution is valid, although the former may be preferred because it is prefix-free and is therefore instantaneously decodeable. For (d, k) constraints, one possible minimal set consists of the $k - d + 1$ sequences of the form '$0^{(i)}1$', where $0^{(i)}$ represents a sequence of i consecutive zeros, and i takes values from d through k. This can be verified by examining the output sequences corresponding to the paths outlined in Figure 3.2. For $k = \infty$ constraints, the underlying structure of the constraint can be slightly modified to require only two codewords in the minimal set: {'$10^{(d)}$','0'}.

In order to maximize the entropy of a minimal set, the codewords should be independent of each other. This also allows them to be concatenated in any order without violating the constraint. As a result, code design is considerably simplified.

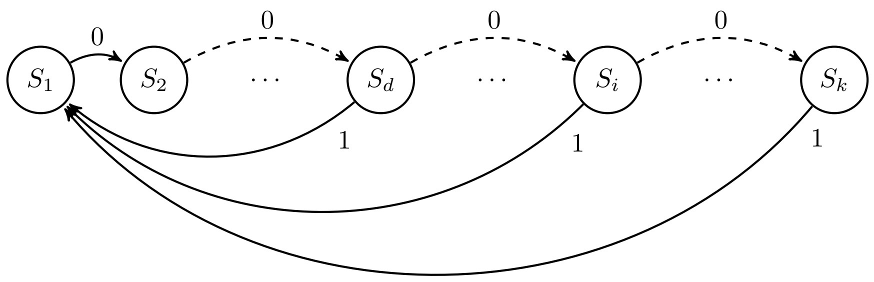

Figure 3.2: (d, k) constraint graph

Capacity of a VL Set

Recall the evaluation of capacity for a constrained sequence code as discussed in Chapter 1:

$$d_{ij}(z) = \sum_{l \in \{L_{ij}\}} z^{-l} \tag{3.1}$$

$$\lambda_{max} = \arg\max_{z \in \mathbb{R}} \left(det[D(z) - I] = 0 \right) \tag{3.2}$$

$$C_{CSC} = \log_2 \lambda_{max}. \tag{3.3}$$

Here L_{ij} is the set of codeword lengths available for the transition between state i and state j in the finite state machine model of the constraint. If all of the available codewords can be emitted independently, as is the case where the constraint can be modeled with only a single state, then the characteristic equation leading to the determination of λ_{max} simplifies to [1]:

$$z^{-l_1} + \cdots + z^{-l_N} = 1, \tag{3.4}$$

where the size of the set of available codewords is assumed to be N. Sometimes it is possible to convert a multi-state constraint graph with fixed length codewords into a single-state Markov source with variable-length codewords, for example, the set of $k - d + 1$ codewords for a (d, k) constraint discussed in the previous section. The case where all codewords are independent is the simplest case to work with, and will be the case discussed in the greatest detail here. It is possible to extend this analysis to multiple-state representations.

The capacity of the minimal set of (d, k) codewords is the logarithm of the largest real root of Equation (3.4). For this constraint, the set of available codeword lengths and the resulting equation for capacity is:

$$L = \{d + 1, \ldots, k + 1\} \tag{3.5}$$

$$C = \log_2 \left(\arg\max_{z \in \mathbb{R}} \left(\sum_{l \in L} z^{-l} = 1 \right) \right) \tag{3.6}$$

Maxentropic Probabilities of VL set

Firstly, consider a set of N independent codewords. To determine the maxentropic probability distribution for each of the N codewords in the VL set, consider an N-ary source that is mapped in a one-to-one fashion with each of the N codewords. Let the i-th symbol have probability p_i, and map it to a codeword of length o_i. The average rate of the code that maps the symbols to codewords is:

$$\begin{aligned} R_{avg} &= \frac{\mathbb{E}[L_{in}]}{\mathbb{E}[L_{out}]} \\ &= \frac{H(in)}{\mathbb{E}[L_{out}]} \\ &= \frac{-\sum_i^N p_i \log_2 p_i}{\sum_i^N p_i o_i}. \end{aligned} \tag{3.7}$$

Here $\mathbb{E}[L]$ is the expected length of a sequence in bits. For an N-ary source, this length is equivalent to the entropy of the source, $H(in)$, because the entropy represents the average information content of the source in bits. Because of the one-to-one mapping of the code, the codeword with length o_i has probability p_i, and therefore the expected output length is easily calculated.

Determining the maxentropic probability distribution for each of the codewords in the variable-length set is equivalent to determining the probability distributions for which Equation (3.7) equals capacity. One approach is to maximize (3.7) using Lagrange multipliers, keeping in mind the necessary condition $\sum p_i = 1$.

$$\Lambda(p_1; \ldots ; p_N; \lambda) : \frac{-\sum_i^N p_i \log_2 p_i}{\sum_i^N p_i o_i} + \lambda \left(\sum_{i=1}^N p_i - 1 \right) \tag{3.8}$$

$$\begin{aligned} \frac{\delta \Lambda}{\delta p_i} : & \frac{\left[\sum_j p_j o_j\right] \left[-\log_2 p_i - \frac{1}{\ln 2}\right] + o_i \left[\sum_j p_j \log_2 p_j\right]}{\left[\sum_j p_j o_j\right]^2} & +\lambda(1) = 0 \\ - & \frac{\bar{L}\left[\log_2 p_i + \frac{1}{\ln 2}\right] + o_i \left[H(in)\right]}{\left[\bar{L}\right]^2} & +\lambda = 0 \\ - & \frac{\left[\log_2 p_i + \frac{1}{\ln 2}\right] + o_i \left[R_{avg}\right]}{\left[\bar{L}\right]} & +\lambda = 0 \end{aligned} \tag{3.9}$$

$$\begin{aligned} \frac{\delta \Lambda}{\delta \lambda} : \sum_i^N p_i - 1 &= 0 \\ \sum_i^N p_i &= 1 \end{aligned} \tag{3.10}$$

Note that $\bar{L}$ has been substituted for the calculation of the average output length of the set, $H(in)$ for the calculation of the average input length of the set, and R_{avg} for the calculation of the average rate given by Equation (3.7). To solve for the maxentropic probability distribution, re-arrange Equation (3.9) and solve for p_i:

$$\begin{aligned} \log_2 p_i &= \lambda \bar{L} - o_i R_{avg} - \frac{1}{\ln 2} \\ p_i &= 2^{\left(\lambda \bar{L} - o_i R_{avg} - \frac{1}{\ln 2}\right)}. \end{aligned} \tag{3.11}$$

Substituting this equation for p_i into Equation (3.10) and solving for λ yields:

$$\begin{aligned} \sum_i 2^{\left(\lambda \bar{L} - o_i R_{avg} - \frac{1}{\ln 2}\right)} &= 1 \\ 2^{\left(\lambda \bar{L}\right)} 2^{\frac{-1}{\ln 2}} \sum_i 2^{\left(-o_i R_{avg}\right)} &= 1 \\ -\frac{\log_2 \left(2^{\frac{-1}{\ln 2}} \sum_i 2^{\left(-o_i R_{avg}\right)}\right)}{\bar{L}} &= \lambda \\ \frac{\frac{1}{\ln 2} - \log_2 \left(\sum_i 2^{\left(-o_i R_{avg}\right)}\right)}{\bar{L}} &= \lambda. \end{aligned} \tag{3.12}$$

It is important to recognize that this is the maxentropic case, so $R_{avg} = C = \log_2 \lambda_{max}$. Therefore $2^{(-o_i R_{avg})} = \lambda_{max}^{(-o_i)}$. With this substitution, it is clear that the summation in Equation (3.12) is equivalent to the summation in Equation (3.4), of which λ_{max} is a solution by definition. As a result, the solution for λ simplifies to:

$$\lambda = \frac{\frac{1}{\ln 2} - \log_2 (1)}{\bar{L}} = \frac{1}{\bar{L} \ln 2}. \tag{3.13}$$

Substituting this solution back into the equation for p_i reveals that the maxentropic probabilities for independent variable-length codewords are:

$$p_i = 2^{-o_i C} = \lambda_{max}{}^{-o_i}. \tag{3.14}$$

For a set of variable-length codewords, the maxentropic probability for each of the codewords is not only a function of the capacity of the set, but also inversely proportional to the codeword length. Note that this result has been derived previously, typically when discussing maxentropic probabilities for (d, k) constraints [29], but has not typically been discussed within the context of sets of variable-length codewords.

For more complex constraints where the codewords are not independent and must be modeled with a state machine, the state transition probabilities that result in the maximum output information must be evaluated. It has been shown [1] that these probabilities can be evaluated given the connection matrix $D(z)$, the maximum real eigenvalue λ_{max}, and the corresponding eigenvector $x = (x_1, \ldots, x_N)$:

$$q_{ij} = d_{ij}(\lambda_{max}) \frac{x_j}{x_i} \tag{3.15}$$

where $d_{ij}(z)$ was defined in (3.1), and q_{ij} represents the maxentropic steady-state probability of transitioning from state i to state j.

Consider a set of variable-length independent codewords. Since they are independent, there is no restriction on the value of the next codeword. In follows from (3.14) that the maxentropic distribution of the codewords is not equiprobable unless the codewords all have the same length. Also, the maxentropic distribution will have the shortest codewords occurring more often than the longer codewords. A maxentropic distribution is desirable: if it can be achieved then the output sequences will have maximum entropy. Unfortunately, analysis in the strictly sub-optimal regime where the maxentropic distribution is not *exactly* achieved is quite difficult, and no general statement about the probability distributions that maximize entropy within this regime should be inferred from the analysis above. In particular, the derivative in (3.9) is no longer zero, and $R_{avg} \neq C$, so many of the simplifications discussed here no longer apply. However, as was shown in Chapter 2, and will be demonstrated again in Chapter 4, codes constructed with techniques that closely approximate the maxentropic distribution can have code rates that are within a percentage point or two of capacity, and therefore within a percentage point or two of maximum entropy.

Extensions of a Source

For the purposes of this section, denote a set of independent variable-length codewords as the minimal set, as well as the source for subsequent iterations of processing. Note that the term “source” is being used in a very broad sense, as this set could be used to represent either the input or output sequences of an actual code. However, the actual purpose of this set is irrelevant for the remainder of this discussion. Given this source, it is possible to identify higher order extensions of this source, similar to those found in classical discussions on information theory [6]. However, the distinction between a *complete* extension of a source, and a *partial* extension of a source, will be made.

A complete extension of a source enumerates all possible pairs of codewords from the current set with those from the minimal set. Note that the codewords from the current set act as prefixes to the new codewords, and codewords from the minimal set act as suffixes. For example, given a source with codewords ‘1’ and ‘0’, the complete first order extension of this source is ‘00’, ‘01’, ‘10’, and ‘11’. The complete second-order extension is ‘000’, ‘001’, ‘010’, ‘011’, ‘100’, ‘101’, ‘110’, and ‘111’. Other works may define the second-order extension differently, but this is the notation that will be used here.

A partial extension of a source enumerates all possible pairs of *some* of the codewords from the current sent with *all* of the codewords from the minimal set. For example, one possible first-order partial extension of the ‘1’,‘0’ source is ‘00’, ‘01’, and ‘1’. Here, the ‘1’ codeword remains in the final set, while the ‘0’ codeword is now a prefix to both codewords ‘00’ and ‘01’. Note that the complete extension of the source is a possible partial extension, wherein all of the codewords are expanded. To keep the order of partial extensions as unambiguous as possible, the order of the partial extension will be denoted by the first possible order for which the same set of codewords could have been generated. So although the set of codewords '00', '01', '10', and '11' could be generated from a partial extension of '00', '01', and '1', this would not be a second order extension because it was also one of the possible first order extensions, in this case the complete first order extension.

When discussing partial extensions, it is often convenient to analyze an extension in terms of a tree structure. The initial tree has N_0 leaves, one for each of the codewords in the minimal set. A partial extension will add exactly N_0 branches to k leaves in the tree, where k is less than or equal to the size of the current set. To be consistent with the notation discussed above, the N_0 branches can only be added to the leaves of highest depth, and k is less than or equal to the number of leaves at that depth. This procedure is outlined in Figure 3.3.

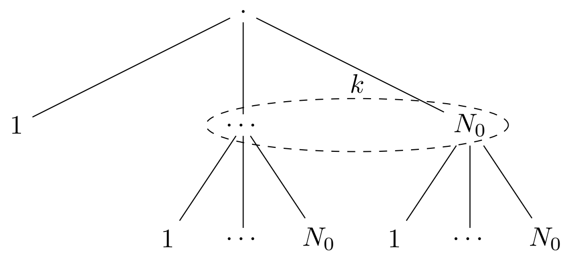

Figure 3.3: The tree structure of a partial extension

Clearly, partial extensions provide more flexibility than complete extensions. Although partial extensions are in some sense "incomplete", it is readily apparent that all of the sequences that could have been generated with the minimal set can *still* be generated by the new partially extended set. Since the same number of unique sequences can be generated, the capacity of the new set is the *same* as the capacity of the minimal set:

$$C_n = C_{n-1}, \tag{3.16}$$

where C_n represents the capacity of the n-th partial extension, and C_0 is the capacity of the minimal set. The key property of partial extensions that allows the capacity to remain constant is the use of *all* of the codewords from the minimal set in every partial extension. Since every codeword from the minimal set is used, the number of unique sequences remains the same. It is, however, possible that the maxentropic probabilities may have changed, as will be discussed in the next section.

Maxentropic Probabilities of Partial Extensions

As discussed above, the capacity of a partial extension is identical to the capacity of the minimal set. However, there are now two potentially different approaches to determining the maxentropic probabilities of the new codeword set.

The first approach is to consider the new codeword set in isolation and determine the maxentropic probabilities as outlined previously. Let the minimal set have N_0 codewords with codeword lengths $(o_1, \ldots, o_{N_0})$. The first order partial extension will have codewords of length $o_i + o_j$. As discussed above, since the capacity of the the two codeword sets are the same, the only variable relevant to the maxentropic probabilities is codeword length. The new maxentropic codeword probability associated with a codeword that has been partially extended is:

$$p_{i,j} = {\lambda_{max}}^{-(o_i+o_j)}. \tag{3.17}$$

The other approach to determining maxentropic probabilities is to exploit the independence of the original codewords and simply multiply the two original maxentropic probabilities together to determine the new joint probability:

$$p_{i,j} = {\lambda_{max}}^{-o_i} {\lambda_{max}}^{-o_j}. \tag{3.18}$$

Clearly the two approaches are equivalent. This property extends to higher order extensions as well:

$$p_{i,j,k} = {\lambda_{max}}^{-(o_i+o_j+o_k)} = {\lambda_{max}}^{-o_i} {\lambda_{max}}^{-o_j} {\lambda_{max}}^{-o_k}. \tag{3.19}$$

3.2 Code Trees

A code tree is a representation of the structure of a variable-length code. As discussed in Chapter 1, a code maps input sequences into output sequences. In order to use a code, it is also important that output sequences map uniquely back to input sequences, as this allows for the perfect reconstruction of the original sequence in the absence of errors. A code tree is a graphical technique to describe the mapping of variable-length sequences to other variable-length sequences. Separate but related code trees can be drawn for both the sourceword to codeword mapping and the codeword to sourceword mapping. To clarify the discussion, input sequences will be referred to as sourcewords wherever possible. However, they may also be indirectly referred to as codewords when the discussion could equally apply to either the input or output sequences. The codes discussed here will be constructed using partial extensions of a VL minimal set for a desired constraint. The difference between a code tree and a diagram of the internal tree structure of a partial extension is simply the addition of a mapping for some (or all) of the leaves into codewords. This property is depicted in Figure 3.4.

Note that in the broad class of VL codes discussed here, both the input sequences and the output sequences will have a tree structure. In these cases, it may be desirable to analyze both the mapping and the respective tree structures. In this case, code trees for the sourceword to codeword mapping and codeword to sourceword mapping can be drawn back-to-back, as demonstrated in Figure 3.5.

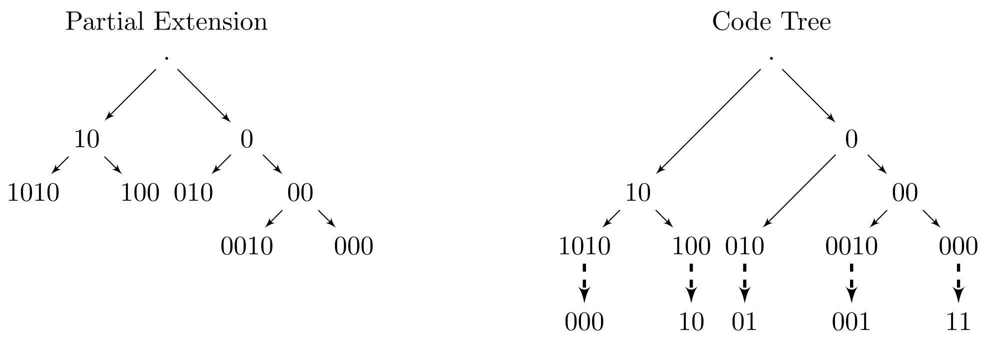

Figure 3.4: Partial extension graph vs. a Code tree

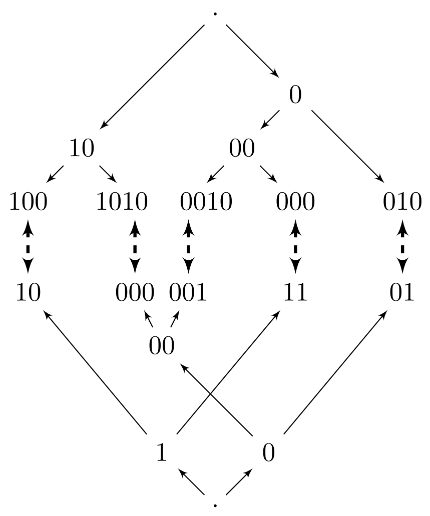

Figure 3.5: The input-to-output and output-to-input code trees drawn back-to-back

When dealing with partial extensions of minimal sets, the independence of the codewords and the non-uniqueness of the minimal sets results in the use of codeword lengths as the most important factor during code design, rather than the sequences themselves. This is primarily due to the fact that the maxentropic probabilities of the codewords are a function of the capacity of the constraint, which is constant, and the codeword lengths (see Equation (3.14) and Equation (3.17)). As a result, it is often convenient to specify a code tree with only codeword lengths, as any constraint-satisfying set with the same codeword lengths can be considered equivalent. This shorthand form is depicted in Figure 3.6.

Greedy Trees

When a shorthand notation, such as codeword lengths, is used to describe a tree there can be some ambiguity. Different tree structures can produce the same output codeword lengths. For example, Figure 3.7 pictures three trees where the leaves have the same sequence lengths, and therefore the same capacity, but have completely different underlying

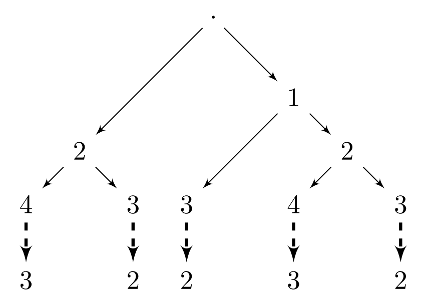

Figure 3.6: A code-tree specifying codeword and sourceword lengths

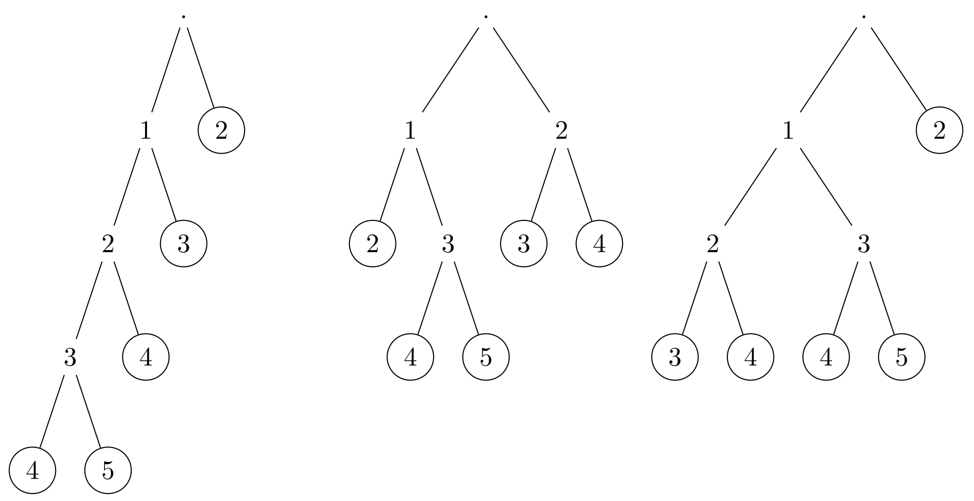

Figure 3.7: Three different trees with the leaves representing the same sequence lengths

tree structures. For all intents and purposes, these trees are considered equivalent as the codewords compromise codes that have the same capacity. However, to help resolve the ambiguity when dealing with variable length sets, this work introduces the concept of a greedy tree. A greedy tree is a method of selecting a specific tree from the possibly many trees that result in the same codeword lengths. They are used later on in code construction to reconstruct a tree from a set of codeword lengths, which avoids the necessity of storing the underlying tree structure when searching through partial extensions.

Given a set of required sequence lengths, and the minimal set of codeword lengths originally used to construct it, there exists a unique tree structure that represents this required set which is known as a greedy tree. If no partial tree extension of the minimal set will result in the required sequence lengths, then this procedure will fail and it will be impossible to completely remove all codeword lengths from the required list. Greedy trees are constructed through partial extensions of the minimal set according the the following procedure:

1. Sort the set of required codeword lengths from smallest to largest.

2. Starting with the set of leaves corresponding to the parent with the smallest codeword length, remove a codeword length from the required list if there exists a leaf with that length, and mark the corresponding leaf.

3. Repeat for all other sets of leaves, removing as many codeword lengths from the required list as possible. This approach of assigning a codeword length to the first possible leaf representing that length gives the greedy tree its name.

4. For each leaf that remains unmarked, do a complete partial extension: attach a branch for every codeword in the minimal set to every leaf that is unmarked.

5. Within each set of new branches, ensure that the leaves are sorted by codeword length. The new codeword lengths are equal to the length of the parent plus the length of the corresponding codeword in the minimal set.

6. Repeat from Step 2 until all of the leaves have been marked.

Figure 3.8 demonstrates this procedure for the set of codeword lengths $\{2, 3, 4, 4, 5\}$ and the minimal set $\{1, 2\}$.

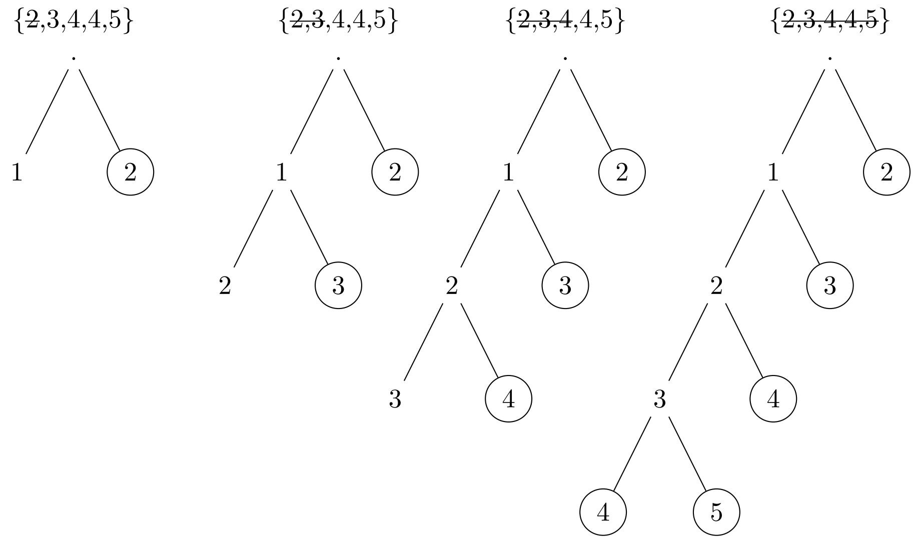

Figure 3.8: Construction of a greedy tree

A complete code

A complete code is a code where all of the available codewords have a mapping to a sourceword. In a code tree, a complete code is a tree where all of the leaves have been assigned a sourceword. In contrast, an example of an incomplete code tree is given in Figure 3.9 where the leaf '0010' has not been assigned a sourceword. Recall that for this code tree, the leaves of the tree are codewords, and the mappings are sourcewords.

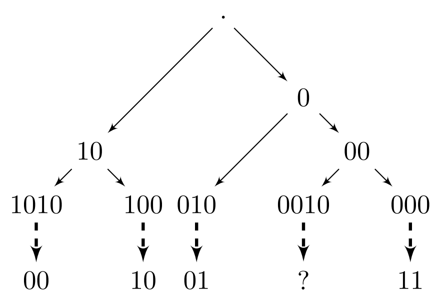

Figure 3.9: A codeword to sourceword code tree of an incomplete code

Incomplete codes typically represent either a flaw or an inefficiency in code design. In the codeword to sourceword code tree, for example Figure 3.9, an unused leaf represents a valid constraint-satisfying sequence that will not be generated by the encoder, and which therefore reduces the capacity of the codeword set. In the sourceword to codeword code tree, however, an unused leaf represents a valid input sequence that has no corresponding codeword mapping in the code. This may either be a flaw in code design, or intentionally part of the design based on prior knowledge of the source. However, in the general case it represents an unmapped sourceword, and therefore the code design discussed here will focus solely on sourceword to codeword mappings where the sourceword tree is complete. In other words, the set of sourcewords used in a code must be exhaustive. This property was also noted in Chapter 2.

3.3 Optimal Sourceword Lengths

As shown in Equation (3.14), the maxentropic probability for a codeword of length o_i is:

$$p_i = 2^{-o_i C}. \tag{3.20}$$

In an ideal code design, this codeword is mapped to a sourceword with this exact probability, and the result is a capacity achieving code. For a binary source where the probability of a '1' is equal to the probability of a '0', and where '1's and '0's are emitted independently, the probability of a sourceword with length l_i is:

$$p_i = 2^{-l_i}. \tag{3.21}$$

Clearly, to achieve a maxentropic probability distribution, and therefore a maxentropic code rate, sourcewords should be of the following length:

$$l_i = C * o_i. \tag{3.22}$$

For some constraints, such as most RLL constraints, C is an irrational number [22] and $C*o_i$ cannot therefore be a whole number. To avoid this problem, the probability transformers discussed in Chapter 2 attempt to generate the maxentropic probability distribution from an equiprobable binary source, and then simply map these probabilities to the corresponding codewords.

3.4 Assigning sourcewords based on source codes

The use of source codes to generate sourcewords for constrained sequence coding is discussed in depth in Chapter 2. Specifically, Kerpez [2] demonstrated the efficacy of several source codes, typically in the limit as the number of complete code extensions tends to infinity. Existing research does not, however, answer the question: For a finite size set of independent constrained sequence codewords, what is the optimal technique of assigning sourcewords to these codewords such that the code rate is maximized? This section will demonstrate that assigning sourcewords based on a Huffman source code [30] is optimal in the sense that no other sourceword assignment for the same set of codewords can result in a better rate when the source sequence consists of independent and equiprobable symbols.

Huffman coding has long been understood to be optimal in the "forward" or compression direction. However, its use for the purpose described here is not as clear. The problem is that the compression operation is imperfect, and the probabilities of '1's and '0's are neither independent nor equiprobable. However, the code rate of concern in the present CSC application is actually in the expansion direction. In this direction, a binary i.i.d source is explicitly assumed, and therefore the input probabilities are *forced* to be independent and equal. This disconnect between the compression and expansion operations means that Huffman coding cannot be considered optimal without an explicit analysis of Huffman coding when used for this purpose.

Properties of optimal and Huffman codes

This section will demonstrate a key property of an optimal code, and show that the same property also holds for codes generated using the Huffman procedure.

An Optimal Code

For a given set of output codewords there exists at least one complete mapping of sourcewords to codewords that achieves the best possible average code rate for that set of codewords. Potentially, there could be even more than one mapping that results in the highest possible average rate. However, there is an identifying property that is true of at least one of the highest rate sourceword to codeword mappings, and for the purposes of clarity, only codes that obey this property will be considered "optimal".

Recall that the rate of a variable length code which maps binary input sequences of length l_i to binary output sequences of length o_i is:

$$R = \frac{\sum_i 2^{-l_i} l_i}{\sum_i 2^{-l_i} o_i}, \tag{3.23}$$

assuming an ideal binary source where 1's and 0's occur independently and with the same probability. This assumption is necessary for the proof of optimality given here to hold. Other sources, with different distributions, may or may not share this result.

Lemma 1. *There exists an optimal complete code with the following properties:*

1. *Sorted sourceword and codeword lengths such that if $o_k \geq o_j$, then $l_k \geq l_j$*
2. *The longest two codewords have the same sourceword lengths*

Proof. Let R_c represent the code rate of an *optimal* code, and let R_c^* represent the rate of a code with the input sourcewords l_j and l_k swapped. The output codeword lengths o_i are ordered such that:

$$o_1 \leq \cdots \leq o_j \leq o_k \leq \cdots \leq o_N \tag{3.24}$$

If $o_k = o_j$, then the swapping operation has no effect on the code rate. Therefore $R_c = R_c^*$, and there exists an optimal code with sorted assignments such that $l_k \geq l_j$.

If $o_k > o_j$, and R_c is optimal, then $1/R_c$ is a minimum.

$$\frac{1}{R_c^*} - \frac{1}{R_c} \geq 0 \tag{3.25}$$

$$\frac{\sum_i 2^{-l_i^*} o_i}{\sum_i 2^{-l_i^*} l_i^*} - \frac{\sum_i 2^{-l_i} o_i}{\sum_i 2^{-l_i} l_i} \geq 0 \tag{3.26}$$

Clearly, swapping sourcewords l_j and l_k has no effect on the denominator; it is constant and can be ignored. All of the remaining values cancel except where l_j and l_k have been swapped:

$$\begin{aligned} 2^{-l_j} o_k + 2^{-l_k} o_j - 2^{-l_j} o_j - 2^{-l_k} o_k &\geq 0 \\ \left(2^{-l_j} - 2^{-l_k}\right)(o_k - o_j) &\geq 0 \end{aligned} \tag{3.27}$$

Since the codewords have been ordered such that $o_k > o_j$, then the following must also be true:

$$\begin{aligned} 2^{-l_j} - 2^{-l_k} &\geq 0 \\ 2^{-l_j} &\geq 2^{-l_k} \\ -l_j &\geq -l_k \\ l_j &\leq l_k \end{aligned} \tag{3.28}$$

This proves that there exists an optimal code with the first property. In fact (3.28) demonstrates that when two codeword lengths are different, it is necessary for their assigned sourcewords to have sorted lengths in all optimal codes.

To prove that this code has the longest two codewords assigned to the same sourceword length, it is helpful to visualize the sourceword to codeword tree, as in Figure 3.10. In this figure the longest codeword, with length o_N, is assigned to the longest sourceword, with length l_N. If the second longest codeword, with length o_{N-1}, does not have the same sourceword length then it must have a sourceword length of $(l_N - 1)$ due to the sorted property of this code. Since there are no codewords between the longest and second longest

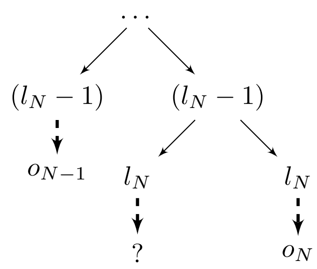

Figure 3.10: Incomplete code tree

codewords, this leaves one of the sourcewords unassigned. As discussed above, an incomplete sourceword code tree represents a flaw in code design, in this case, a sourceword that has no coded representation, so it cannot be communicated across the channel. Therefore, there exists an optimal complete code in which the codeword code tree is complete and has the two longest codewords assigned to sourcewords of the same length. □

A Huffman Code

Lemma 2. *Huffman coding the maxentropic probabilities corresponding with the sorted codeword lengths $\{o_1, \ldots, o_N\}$ results in a sorted assignment of sourceword lengths, where if $o_k > o_j$, then $l_k \geq l_j$.*

Proof. The maxentropic probabilities associated with the output codeword lengths are (see Equation (3.14)):

$$p_i = \lambda_{max}^{-o_i} \tag{3.29}$$

Since $\{o_1, \ldots, o_N\}$ are sorted such that $o_1 \leq \cdots \leq o_j \leq o_k \leq \cdots \leq o_N$, then:

$$p_1 \geq \cdots \geq p_j \geq p_k \geq \cdots \geq p_N \tag{3.30}$$

A Huffman code has the following property [6]: If $p[x_j] > p[x_k]$ then $l_j \leq l_k$. Therefore, if the input codewords are ordered such that if $o_k > o_j$, then $p_j > p_k$ and a Huffman code will have sourceword lengths such that $l_j \leq l_k$. □

Properties of Merging

A Huffman code is constructed by repeatedly merging the two least probable codewords, assigning each of these merged codewords a unique symbol in the process, and when the procedure is complete, using the strings of assigned symbols as sourcewords. The properties associated with merging, as well as the definitions of a Huffman and a non-Huffman merging rule, are discussed in this section.

Merging

Define the set of available output codeword lengths as $\{o_i\},\ i = 1 \ldots N$. From (3.14), each codeword has a probability of $p_i = \lambda_{max}^{-o_i}$. If two output codewords are merged, the probability of the "new" codeword becomes the sum of the probabilities of the merged codewords. The use of an $*$ will indicate values in the "new" set where merging has already taken place. The probability of the new codeword i that resulted from merging codewords j and k is:

$$p_i^* = p_j + p_k = \lambda_{max}^{-o_j} + \lambda_{max}^{-o_k}. \tag{3.31}$$

For consistency with the probability relationship defined above, this new codeword is defined to have have an effective codeword length of:

$$o_i^* = -\log_{\lambda_{max}} \left(\lambda_{max}^{-o_j} + \lambda_{max}^{-o_k}\right). \tag{3.32}$$

The sourcewords assigned to output codewords j and k in the original set have the same length, and will be one bit longer than the "equivalent" sourceword length assigned to the new codeword i:

$$l_j = l_k = l_i^* + 1. \tag{3.33}$$

The two sourcewords will differ in their last digit: one will have a zero, while the other will have a one.

Huffman Merging Rule

Consider the set of output codeword lengths $\{o_i\},\ i = 1 \ldots N$. Sort these lengths such that $o_1 \leq \cdots \leq o_j \leq o_k \leq \cdots \leq o_N$. The two least probable symbols will have lengths o_{N-1} and o_N, due to the relationship $p_i = \lambda_{max}^{-o_i}$. Create a new (unsorted) set of codeword lengths by copying all but the two least probable codewords into the new set. Merge the two least probable codewords and generate a new effective codeword length for the merged codeword based on (3.32).

$$o_i^* = o_i,\ i = 1 \ldots (N-2) \tag{3.34}$$

$$\begin{aligned} o_{N-1}^* &= -\log_{\lambda_{max}} \left(\lambda_{max}^{-o_N} + \lambda_{max}^{-o_{N-1}}\right) \\ &= -\log_{\lambda_{max}} \left(\lambda_{max}^{-(o_{N-1}+d)} + \lambda_{max}^{-o_{N-1}}\right) \\ &= -\log_{\lambda_{max}} \left(\lambda_{max}^{-o_{N-1}} \left(1 + \lambda_{max}^{-(d)}\right)\right) \\ &= o_{N-1} - \log_{\lambda_{max}} \left(1 + \lambda_{max}^{-(d)}\right), \end{aligned} \tag{3.35}$$

where $d = o_N - o_{N-1}$ and is strictly non-negative. Therefore, the new effective output codeword length is shorter than either of the two original codeword lengths. The set of unsorted codeword lengths after merging is:

$$\{o_i^*\} = \{o_{N-1}^*, o_{N-2}, \ldots, o_j, \ldots, o_1\}. \tag{3.36}$$

Note that o_{N-1}^* can be smaller or larger than any codeword length on the list, depending on the initial values.

A Huffman code for the set is created by repeating the Huffman merging rule until there remains only a single codeword length in the set. A Huffman code constructed using the above merging rule will have the property that the two longest output codewords during each iteration of the merging rule will be assigned the longest sourceword length for that iteration. This process is outlined in Figure 3.11. Note that the minimal set in the example has a λ_{max} of $(1+\sqrt{5})/2$.

$$\begin{tikzcd}
o_i = \{ & 4 \arrow[drrr] & 4 \arrow[drr] & 3 & 3 & 3 & \} \\
o_i = \{ & 3 \arrow[drr] & 3 \arrow[dr] & 3 & 2.56 & \} & \\
o_i = \{ & 3 \arrow[dr] & 2.56 \arrow[d] & 1.56 & \} & & \\
o_i = \{ & 1.56 \arrow[d] & 1.33 \arrow[dl] & \} & & & \\
o_i = \{ & 0 & \} & & & &
\end{tikzcd}$$

Figure 3.11: An example of four rounds of Huffman merging: the two longest (least probable) codewords are merged in each iteration

Non-Huffman Merging

When considering the sorted list of codeword lengths $\{o_i\},\ i = 1 \ldots N$, the Huffman technique always merges the last two codewords, since these two codewords are the least probable. Consider an alternative approach, where the largest codeword length o_N is instead merged with the codeword length o_j, where:

$$o_N \geq o_{N-1} > o_j. \tag{3.37}$$

This condition is mutually exclusive to Huffman merging. After merging, the new output codeword has an effective length of:

$$\begin{aligned} o_M^* &= -\log_{\lambda_{max}} \left(\lambda_{max}^{-o_N} + \lambda_{max}^{-o_j}\right) \\ &= -\log_{\lambda_{max}} \left(\lambda_{max}^{-(o_j+d_M)} + \lambda_{max}^{-o_j}\right) \\ &= -\log_{\lambda_{max}} \left(\lambda_{max}^{-o_j}\left(1+\lambda_{max}^{-(d_M)}\right)\right) \\ &= o_j - \log_{\lambda_{max}} \left(1+\lambda_{max}^{-(d_M)}\right), \end{aligned} \tag{3.38}$$

where $d_M = o_N - o_J$ is strictly non-negative. Once again, the new effective output codeword length is shorter than either of the two original codeword lengths. A single round of non-Huffman merging is demonstrated in Figure 3.12 for the same set of codewords and λ_{max} as in Figure 3.11.

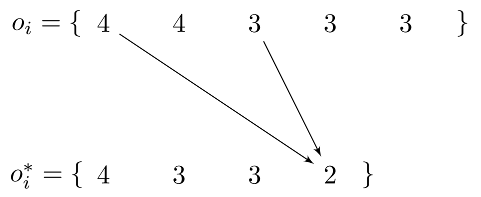

Figure 3.12: A single round of non-Huffman merging: the longest codeword is merged with a codeword other than the second largest

The new sorted codeword lengths after merging are:

$$\{o_i^*\} = \{o_{N-1}, o_{N-2}, \ldots, o_M^*, \ldots, o_1\} \tag{3.39}$$

Due to the sorted property of the sourcewords and the condition in (3.37):

$$l_N \geq l_{N-1} \geq l_j \tag{3.40}$$

But, after merging $l_j = l_N$, so the following condition must be true:

$$l_N = l_{N-1} = l_j \tag{3.41}$$

For this to be true, there must be at least one more codeword with a sourceword of the same length, since merging occurs in pairs.

$$l_N = l_{N-1} = l_k = l_j \tag{3.42}$$

As a result, choosing a non-Huffman merging rule guarantees that at least four of the terms will have the same sourceword length for this iteration, whereas Huffman merging places this condition on only two of the codewords.

Proof of Huffman Optimality

The two lemmas outlined above prove that there exists an optimal code with sorted input and output codeword lengths, and that a Huffman code also produces such a sorted mapping. What remains to be shown is that out of all of the merging options for a given set of codewords that result in sorted mappings, Huffman merging during each iteration results in an optimal code. Here, optimal means that for the Huffman code rate, R_{Huff}, and code rate of any code that generates a sorted mapping, R_{sort}:

$$R_{Huff} \geq R_{sort}. \tag{3.43}$$

To prove this, it will be shown that considering only Huffman merging during each iteration is sufficient to construct an optimal code.

Theorem 1. *The Huffman coding technique is a method of assigning sourceword lengths to a given set of codeword lengths such that the resulting code has the same rate as the optimal code outlined in Lemma 1.*

Proof. Consider three codes:

1. an optimal code;
2. a code constructed using the Huffman merging rule for the first iteration, and an optimal merging rule for all other iterations;
3. a code constructed using a non-Huffman merging rule for the first iteration, and an optimal merging rule for all other iterations (where the optimal merging rule obeys the sorted sourceword length constraint).

Input / Output mappings					
Optimal		Huffman		Non-Huffman	
l_1	o_1	l_1	o_1	l_1	o_1
...	...	...	...	...	...
l_a	o_j	l_c	o_j	$\mathbf{l_M^{NH}}$	o_j
...	...	...	...	...	...
l_b	o_k	l_d	o_k	$\mathbf{l_M^{NH}}$	o_k
...	...	...	...	...	...
$\mathbf{l_M^O}$	o_{N-1}	$\mathbf{l_M^H}$	o_{N-1}	$\mathbf{l_M^{NH}}$	o_{N-1}
$\mathbf{l_M^O}$	o_N	$\mathbf{l_M^H}$	o_N	$\mathbf{l_M^{NH}}$	o_N

Table 3.1: Sourceword length assignment for optimal merging, Huffman merging, and non-Huffman merging

The construction of Table 3.1 follows from the previous sections. In this table, l_M^O denotes the sourceword length assigned after optimal merging, l_M^H denotes the length assigned after Huffman merging, and l_M^{NH} denotes the length assigned after non-Huffman merging. When the optimal sourceword length assignments l_a and l_b do not both equal l_M^O, then non-Huffman merging necessarily results in a sourceword length to codeword length mapping different from the optimal case. Note that the merging rule for the optimal case and the Huffman case are equivalent in the first iteration. In all later iterations, optimal merging has been assumed, and therefore $l_a = l_c$ and $l_b = l_d$ due to the equivalence of the first iteration. Only for codes where the optimal solution has values of $l_a = l_b = l_M^O$ does the optimal solution not specifically exclude the non-Huffman solution, however in this case the Huffman solution remains equivalent.

By induction, it follows that if the Huffman merging rule is equivalent to the optimal solution in the first iteration, then for the "new" set of codewords generated by the merging procedure, the Huffman merging rule is also equivalent to the optimal solution. This equivalence is due to the fact that all output codewords with the same input sourceword length affect the rate equation the same, independent of any merging rule used during code construction. As a result, any merging rule that generates the same input length to output length mapping will have the same code rate. Since the discussion has been restricted to codes with sorted input and output codewords, assigning the two longest sourcewords to the two longest codewords in each iteration is an obvious choice. This choice was shown above to be equivalent to an optimal solution with sorted sourceword and codeword lengths. While there may also be optimal codes without this restriction, it has already been demonstrated that there is at least one optimal code with this property. Choosing a non-Huffman merging rule will exclude the optimal solution for some subset of codeword lengths, while a Huffman merging rule will not. Therefore, the Huffman merging rule is optimal in the sense that the technique is sufficient for generating a code with an average rate higher than or equal to any other complete code. $\square$

3.5 Optimal partial extensions

For a given set of codewords the Huffman coding technique generates an optimal set of sourcewords. Sets of variable-length constraint-satisfying codewords can be constructed using partial extensions of a minimal set. Clearly, different partial extensions may result in different sets of codeword lengths. Different sets of codeword lengths will result in different maxentropic probability distributions. Different probability distributions may result in a different mappings of sourceword lengths to codeword lengths after the Huffman coding procedure. Some of the the sourceword to codeword mappings will have a higher average rate than others. To generate a code with the best possible average code rate, the only remaining task is to identify the partial extension of the minimal set that results in the highest rate code. Unfortunately, this is a difficult analytical problem, and at this time no solution is known to exist. To address this issue, Chapter 4 will discuss a technique that exhaustively searches over all possible partial extensions, within some bound, in order to determine the highest rate code within the bound.

3.6 Conclusion

This chapter has considered the use of variable-length sequences to construct constrained sequence codes. Various key properties of variable-length sets have been examined, particularly the properties most relevant to code construction: capacity and maxentropic probabilities. The notion of a minimal set for a constraint was also introduced. Once an appropriate minimal set has been identified, Huffman encoding the maxentropic probabilities of the partial extensions of the minimal set results in an optimal code for each particular partial extension. Here, optimal means that no other assignment of sourcewords to codewords for that particular partial extension can result in a higher code rate. Note that the choice of partial extension is critical to improving the code rate, but no optimal analytic technique for determining the best partial extension is currently known.

Chapter 4

Variable-Length Constrained Sequence Code Construction

The premise for developing a high-rate variable-length constrained sequence code has been discussed in Chapter 3. Despite the optimality of Huffman coding, the overall proposed technique is still somewhat limiting. Different partial extensions of the minimal set will result in different sets of codeword lengths, and therefore different sourceword lengths, and ultimately a different average code rate. Therefore, as in source coding, while the Huffman construction approach is optimal for a given set of codewords, different sets constructed through different partial extensions may result in different code rates.

As discussed in §3.5 in Chapter 3, at this time it remains an open question as to how to determine, without a brute force search, the optimal partial extension technique for a given constraint, as well as how to determine the partial extension necessary for a given average code rate. In light of these unknowns, a brute force technique of code construction has been used to find the optimal set of codeword lengths over all possible partial extensions of a given maximal depth. This choice of bound is arbitrary, and other potential bounds could also be considered, such as limiting the maximum number of codewords in the partial extension. Exhaustively searching overall all codes within the bound is unfortunately only practical when the search space is relatively small. However, it is important to note that this complexity occurs only during code design; the resulting code may in fact be very simple to implement. Also, the examples included in this chapter demonstrate that even allowing for only a small number of partial extensions can result in codes that are less than a percentage point away from capacity. As a result, the increased complexity of designing a code based on a greater number of partial extensions may be both unjustified and unnecessary.

In contrast to the method proposed by Kerpez [2] for constrained sequence code construction using a Huffman-source coding algorithm, the extensions proposed here are partial extensions as opposed to only the complete extensions implied within his article. As well, it was shown in Chapter 3 that the Huffman technique generates optimal i.i.d sourceword mappings for a finite set of codewords, a more powerful conclusion than Kerpez demonstrated regarding the asymptotic convergence of the code rate toward capacity. In many respects, the approach developed in this thesis can be viewed as combining the Huffman and "block-to-variable-length" coding techniques discussed by Kerpez into a hybrid "variable-length-to-variable-length" approach. The use of partial extensions provides an added degree of freedom that allows the maxentropic probabilities of the allowed sequences to be more closely aligned with the probabilities generated by the Huffman source code, improving the overall code rate.

Recognizing that the code rate is dependent on the codeword lengths of a partial extension, which are derived from the codeword lengths available in the minimal set, it is possible to list optimal sourceword length to codeword length mappings as a function of the number of partial extensions considered and the codeword lengths of the minimal set. Tables of these mappings are determined and listed for several common minimal sets considering two and three partial extensions. These tables are then used to construct several example (d, k) codes. The underlying technique is then applied to the construction of a DC-free code, and its use for designing probability transformers is also considered.

4.1 Tables of optimal sourceword to codeword length mappings

By using an exhaustive search over all partial extensions less than a particular depth, it is possible to identify the highest rate mappings between sourceword lengths and codeword lengths for a given minimal set. As discussed in Chapter 3, the minimal sets for (d, k) constraints, with finite d and k, follow a specific pattern: they consist of codewords of length $(d+1)$ through $(k+1)$. Tables for the word length mappings corresponding with the highest average rate codes for various (d, k) constraints are listed in the following sections, under a limitation for the maximum number of partial extensions considered. It is important to note that although these tables are identified with (d, k) constraints in mind, they are equally applicable to all constraints that have minimal sets with codewords of the same length. For instance, a $(1, \infty)$ constraint does not have a finite value for k, however it can be represented by a minimal set with two codewords: {'10','0'}. As a result, its minimal set has codewords of identical length to the minimal set for a $(0, 1)$ constraint, and therefore any code that can be constructed to satisfy a $(0, 1)$ constraint can also satisfy a $(1, \infty)$ constraint simply by modifying the codewords in the minimal set and undergoing the same procedure. To emphasize the universality of the results, we consider (d, k)-like minimal sets that apply to any minimal set with codeword lengths from $d + 1$ to $k + 1$.

The number of partial extensions considered has been limited by current computing technology and the time available to identify them. For the tables listed here, the number of partial extensions has been limited to two, except for cases where the number of codewords in the minimal set is at most four, where results are also given for up to three partial extensions.

$(0, k)$-like minimal sets

$(0, k)$ minimal sets have $k+1$ codewords, with codeword lengths from 1 to $k+1$. For minimal sets of this type with no more than two partial extensions, the highest average rate sourceword to codeword length mappings are listed in Table 4.1, along with their average rate and efficiency. Recall that efficiency was defined in (1.21), and is simply the ratio of average code rate to capacity. In these tables, the sourceword to codeword length mappings are listed in columns with the sourceword lengths on the left and codeword lengths on the right. The corresponding information regarding the minimal set and code rate is identified at the top of these columns.

k	1		2		3		4		5		6		7		8		9		10	
$\bar{R}$	0.667		0.864		0.936		0.969		0.984		0.992		0.996		0.998		0.999		0.9995	
E	96.02%		98.30%		98.86%		99.33%		99.63%		99.80%		99.90%		99.95%		99.97%		99.99%	
L_{in} -> L_{out}	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	1	2	3	3	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
			3	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	3	3
			3	4	4	5	5	5	5	5	5	5	5	5	4	4	4	4	4	4
			4	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	5	5
			5	5	5	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7
			5	6	5	6	6	6	7	7	7	7	7	7	7	7	7	7	8	8
					6	7	6	6	7	7	7	7	7	7	8	8	8	8	8	8
					7	7	6	7	7	7	7	7	7	7	8	8	8	8	9	9
					7	8	6	7	7	8	8	8	8	8	8	8	8	8	9	9
							6	7	7	8	8	8	8	8	9	9	9	9	9	9
							7	8	8	8	8	8	8	8	9	9	9	9	10	10
							7	8	8	9	8	8	8	8	9	9	9	9	10	10
							7	8	8	9	8	9	9	9	9	9	9	9	10	10
							8	9	8	9	9	9	9	9	10	10	10	10	10	10
							9	9	9	10	9	9	9	9	10	10	10	10	11	11
							9	10	9	10	9	9	9	9	10	10	10	10	11	11
									9	10	9	10	9	9	10	10	10	10	11	11
									10	11	9	10	10	10	10	10	10	10	11	11
									11	11	9	10	10	10	11	11	11	11	11	11
									11	12	9	10	10	10	11	11	11	11	12	12
											10	11	10	10	11	11	11	11	12	12
											10	11	10	10	11	11	11	11	12	12
											10	11	10	11	11	11	11	11	12	12
											10	11	10	11	11	12	11	11	12	12
											11	12	10	11	11	12	12	12	12	12
											11	12	10	11	11	12	12	12	13	13
											11	12	10	11	11	12	12	12	13	13
											12	13	11	12	12	12	12	12	13	13
											13	13	11	12	12	13	12	12	13	13
											13	14	11	12	12	13	12	12	13	13
													11	12	12	13	12	13	13	13
													11	12	12	13	13	13	13	14
													12	13	12	13	13	13	14	14
													12	13	13	14	12	13	14	14
													12	13	13	14	12	13	13	14
													12	13	13	14	13	13	13	14
													13	14	13	14	13	14	14	14
													13	14	13	14	13	14	14	15
													13	14	14	15	13	14	14	15
													14	15	14	15	13	14	14	15
													15	15	14	15	13	14	14	15
													15	16	14	15	13	14	14	15
															15	16	14	15	15	15
															15	16	14	15	15	16
															15	16	14	15	15	16
															16	17	14	15	15	16
															17	17	14	15	15	16
															17	18	14	15	15	16
																	15	16	15	16
																	15	16	16	17
																	15	16	16	17
																	15	16	16	17
																	15	16	16	17
																	16	17	16	17
																	16	17	16	17
																	16	17	17	18
																	16	17	17	18
																	17	18	17	18
																	17	18	17	18
																	17	18	17	18
																	18	19	18	19
																	19	19	18	19
																	19	20	18	19
																			18	19
																			19	20
																			19	20
																			19	20
																			20	21
																			21	21
																			21	22

Table 4.1: Highest rate sourceword to codeword length mappings for $(0, k)$-like minimal sets with no more than two partial extensions

Note that as k increases, λ_{max} approaches 2 and the probability distribution resulting from Huffman encoding is very close to the maxentropic distribution, resulting in increasing code efficiencies. For minimal sets with less than five codewords the optimal mappings with up to three partial extensions have also been determined and are listed later in Table 4.4.

$(1, k)$-like minimal sets

$(1, k)$ minimal sets have k codewords, with codeword lengths from 2 to $k+1$. For minimal sets of this type with no more than two partial extensions, the highest rate sourceword to codeword length mappings are listed in Table 4.2, along with their average rate and efficiency. Unlike $(0, k)$ codes, increasing k does not appear to directly increase efficiency. This implies that there is no direct link between the value of k and how well the maxentropic distribution is approximated by the Huffman code. However, for minimal sets with a large number of codewords, allowing for even only one more partial extension would drastically increases the size of the search space, and could potentially result in far more efficient codes. For minimal sets with less than five codewords, the optimal mappings with up to three partial extensions have also been determined and are listed in Table 4.4.

$(2, k)$-like minimal sets

$(2, k)$ minimal sets have $k-1$ codewords, with codeword lengths from 3 to $k+1$. For minimal sets of this type with no more than two partial extensions, the highest average rate sourceword to codeword length mappings are listed in Table 4.3, along with their average rate and efficiency. As with the $(1, k)$-like constraints, the trend between code efficiency and k is not clear, however the efficiencies are still very high. For minimal sets with less than five codewords the optimal mappings with up to three partial extensions have also been determined and are listed in Table 4.4.

Small (d, k)-like minimal sets with three partial extensions

By increasing the search space to allow for three or more partial extensions, it may be possible to improve the achievable code rate. Table 4.4 contains the highest rate word length mappings for up to three partial extensions of a (d, k)-like minimal set having no more than four codewords. Theses sets had some of the lowest average code rates in the previous sections. This table will serve as the basis for the high-rate (d, k) codes constructed below.

k	2		3		4		5		6		7		8		9		10	
$\bar{R}$	0.4		0.5455		0.61		0.6462		0.6653		0.6755		0.6815		0.6852		0.6872	
E	98.60%		98.91%		98.79%		99.27%		99.45%		99.44%		99.45%		99.47%		99.46%	
L_{in} -> L_{out}	1	2	1	2	3	4	2	3	2	3	2	3	2	3	2	3	2	3
	1	3	2	3	3	5	3	4	3	4	3	4	3	4	3	4	3	4
			2	4	3	5	3	5	3	5	3	5	3	5	3	5	3	5
					4	6	3	5	4	6	4	6	4	6	4	6	4	6
					4	6	4	6	4	6	4	6	4	6	4	6	4	6
					4	6	4	6	5	7	4	6	4	6	4	6	5	7
					4	7	4	6	5	7	5	7	5	7	5	7	5	7
					4	7	4	7	5	7	5	7	5	7	5	7	5	7
					4	7	5	7	5	8	5	7	5	7	5	7	5	7
					4	7	5	8	5	8	5	8	6	8	5	7	6	8
					5	8	5	8	5	8	6	8	6	8	6	8	6	8
					5	8	6	9	5	8	6	8	6	8	6	8	6	8
					5	8	6	10	6	9	6	9	6	9	6	8	6	8
					5	9			6	9	6	9	6	9	6	9	6	9
					5	9			6	9	6	9	6	9	6	9	6	9
					5	10			6	9	6	9	6	9	6	9	6	9
									6	9	7	10	6	9	7	10	6	9
									7	10	7	10	7	10	7	10	7	10
									7	10	7	10	7	10	7	10	7	10
									7	10	7	10	7	10	7	10	7	10
									7	10	7	10	7	10	7	10	7	10
									7	11	7	11	7	10	8	11	7	10
									7	11	7	11	7	11	8	11	8	11
									7	11	7	11	8	11	8	11	8	11
									8	11	8	11	8	11	8	11	8	11
									8	12	8	12	8	11	8	11	8	11
									8	12	8	12	8	11	8	12	8	11
									8	12	8	12	8	12	8	12	8	11
									8	13	8	12	8	12	8	12	8	12
									9	13	9	13	8	12	8	12	8	12
									9	14	9	13	8	12	8	12	8	12
											9	13	9	13	9	13	8	12
											9	14	9	13	9	13	8	12
											10	14	9	13	9	13	8	12
											10	15	9	13	9	13	8	12
											10	15	9	14	10	14	9	13
											10	16	10	14	10	14	9	13
													10	14	10	14	9	13
													10	15	10	14	9	13
													10	15	10	15	9	13
													10	16	11	15	9	13
													11	16	11	15	9	13
													11	17	11	16	9	13
															11	16	10	14
															12	17	10	14
															12	17	10	14
															12	18	10	14
															13	18	10	14
															13	19	10	14
																	10	14
																	10	15
																	10	15
																	10	15
																	10	15
																	10	15
																	10	15
																	10	15
																	11	16
																	11	16
																	11	16
																	11	16
																	11	16
																	11	16
																	12	17
																	12	17
																	12	17
																	12	17
																	12	17
																	12	18
																	12	18
																	12	18
																	13	18
																	13	19
																	13	19
																	13	19
																	13	19
																	14	20
																	14	20
																	14	20
																	14	21
																	14	21
																	14	22

Table 4.2: Highest rate sourceword to codeword length mappings for $(1, k)$-like minimal sets with no more than two partial extensions

k	3		4		5		6		7		8		9		10	
$\bar{R}$	0.2857		0.4		0.4621		0.495		0.5154		0.5267		0.534		0.5388	
E	99.29%		98.60%		99.39%		99.43%		99.63%		99.51%		99.46%		99.44%	
L_{in} -> L_{out}	1	3	2	4	3	6	2	4	2	4	3	6	3	6	2	4
	1	4	2	5	3	7	3	6	3	6	4	7	4	7	3	6
			2	6	3	7	3	6	3	6	4	7	4	7	4	7
			3	7	4	8	4	7	4	7	4	7	4	7	4	7
			3	8	4	8	4	8	4	8	4	8	4	8	4	8
					4	8	4	8	4	8	4	8	4	8	4	8
					4	9	4	9	5	9	4	8	4	8	5	9
					4	9	4	9	5	9	5	9	5	9	5	9
					4	9	5	10	5	10	5	9	5	9	5	9
					4	9	5	10	5	10	5	9	5	9	5	9
					5	10	5	10	5	10	5	9	5	9	6	10
					5	10	5	11	6	11	5	9	5	9	6	10
					5	10	6	11	6	11	5	10	5	10	6	10
					5	11	6	12	6	11	5	10	5	10	6	11
					5	11	6	12	6	11	5	10	5	10	6	11
					5	12	7	13	6	12	5	10	5	10	6	11
							7	14	6	12	6	11	6	11	6	11
									6	12	6	11	6	11	6	11
									7	13	6	11	6	11	7	12
									7	13	6	11	6	11	7	12
									7	13	6	11	6	12	7	12
									7	14	6	12	7	12	7	12
									8	14	6	12	7	12	7	13
									8	15	6	12	7	12	7	13
									8	15	6	12	7	12	7	13
									8	16	6	12	7	13	7	13
											7	13	7	13	7	13
											7	13	7	13	8	14
											7	13	7	13	8	14
											7	13	7	13	8	14
											7	14	8	14	8	14
											7	14	8	14	8	14
											8	14	8	14	8	15
											8	15	8	14	8	15
											8	15	8	14	8	15
											9	16	8	15	8	15
											9	17	8	15	9	16
													8	15	9	16
													8	15	9	16
													9	16	9	16
													9	16	9	17
													9	16	9	17
													9	17	9	17
													10	17	9	18
													10	18	10	18
													10	18	10	19
													11	19	10	19
													11	20	11	20
															11	21

Table 4.3: Highest rate sourceword to codeword length mappings for $(2, k)$-like minimal sets with no more than 2 partial extensions

(d, k)	$(0, 1)$		$(0, 2)$		$(0, 3)$		$(1, 2)$		$(1, 3)$		$(1, 4)$		$(2, 3)$		$(2, 4)$		$(2, 5)$	
$\bar{R}$	0.692		0.871		0.939		0.40		0.547		0.615		0.286		0.404		0.463	
E	99.72%		99.12%		99.17%		98.60%		99.26%		99.60%		99.29%		99.46%		99.58%	
L_{in} -> L_{out}	2	3	1	1	1	1	1	2	1	2	3	5	1	3	3	7	2	4
	2	3	4	4	2	2	1	3	3	5	3	5	2	7	3	7	3	6
	2	3	4	5	5	5			3	6	3	5	3	11	3	8	3	7
	3	4	4	5	6	6			4	7	4	6	3	12	3	8	4	9
	3	4	4	5	6	6			4	7	4	7			3	8	4	9
			5	6	6	6			5	8	4	7			4	9	4	9
			5	6	6	7			5	9	5	8			4	10	5	11
			5	6	6	7			6	10	5	8			4	10	5	11
			5	6	6	7			6	10	5	8			5	11	5	11
			5	6	6	7			6	11	5	8			5	12	5	11
			6	6	6	7			6	12	5	8			5	12	6	12
			6	7	7	8					5	8			5	13	6	12
			6	7	7	8					6	9			5	13	6	13
			6	7	7	8					6	9			6	14	6	13
			6	7	7	8					6	9			6	14	6	13
			7	8	7	8					6	9					6	13
			7	8	7	8					6	9					6	14
					7	8					6	9					7	14
					8	9					6	10					7	14
					8	9					6	10					7	14
					8	9					6	10					7	15
					8	9					6	10					7	15
					8	9					6	10					7	15
					8	9					7	11					7	16
					8	9					7	11					7	16
					9	10					7	11					7	16
					9	10					7	11					8	17
					9	10					7	11					8	18
					9	10					7	11						
					10	10					7	12						
					10	11					7	12						
					10	11					7	12						
					11	11					8	13						
					11	12					8	14						

Table 4.4: Highest-rate sourceword to codeword length mappings for small (d, k)-like minimal sets with no more than three partial extensions

4.2 (d, k) Codes

As discussed in Chapter 3, (d, k) codes for finite d and k are easily represented by the minimal set:

$$w_{(i-d)} = 0^{(i)}1,\ i = [d, \ldots, k], \tag{4.1}$$

where 0^i denotes repetition of symbol X i times. Also, (d, ∞) constraints can be represented by a minimal set of only two codewords: $\{10^d, 0\}$ (see Minimal Set, page 28). Using these minimal sets and the tables from the previous section, it is possible to identify and construct high rate (d, k) codes. Recall from Chapter 1, if the output of a (d, k) code is fed into a differential encoder, then the output is a run-length limited code. Note that the example codes listed here are, in general, not the only codes that achieve the highest possible rate listed in the tables above, but have been selected arbitrarily from the available choices. The code designer may alternatively select codes from the available choices based on other considerations, such as implementation complexity or error propagation.

Example: $(1, \infty)$ Code

To demonstrate the proposed technique, the high-rate $(1, \infty)$ code identified in Chapter 1 is examined.

First, the constraint graph, capacity, and minimal set describing the constraint must be identified. The constraint graph for a $(1, \infty)$ constraint is pictured in Figure 4.1. From state 'A' it is possible to output either a '0' or a '1'. However, from state 'B', it is possible to output only a '0'. This ensures that there will exist at least one zero, and possibly an unlimited number of zeros, between successive logic ones.

One technique for determining the minimal set for a constraint is to identify a principle state, for example state 'A', and to enumerate all possible output sequences corresponding with all possible paths along the constraint graph that return back to that principle state. From state 'A', the path corresponding with output '0' returns directly to state 'A', so this is the first codeword in the minimal set. The path corresponding with the output sequence '10' also returns to state 'A', so this is another codeword in the minimal set. There are no other unique paths that both initiate in state 'A' and terminate in state 'A', and therefore the entire minimal set is: $\{0, 10\}$. As discussed above, this minimal is a $(0, 1)$-like minimal set in that it has codewords of the same length as the $(0, 1)$ minimal set. This choice for the minimal set is complete, as all possible paths on the constraint graph that initiate and terminate in state 'A' are traversed by paths corresponding with elements from the minimal set. For other constraints it may be impossible or undesirable to identify a minimal set that traverses all possible paths. An example of code construction for such a case will be discussed in the DC-free code construction example later on in the chapter.

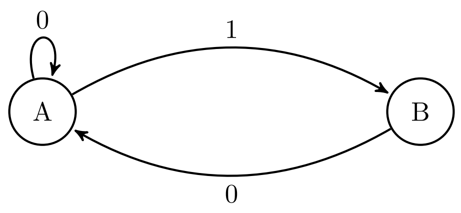

Figure 4.1: Constraint graph for a $(1, \infty)$ code

The capacity derivation for a VL set was identified in Chapter 3. Solving the characteristic equation given in (3.4) for this minimal set results in the following roots:

$$\begin{aligned} -1 + z^{-1} + z^{-2} &= 0 \\ z^2 - z^1 - 1 &= 0 \end{aligned} \tag{4.2}$$

$$z = \frac{1 \pm \sqrt{1+4}}{2} \tag{4.3}$$

Both roots are real, and the largest root is $\frac{1+\sqrt{5}}{2}$. The capacity of the constraint is therefore:

$$C_{1,\infty} = \log_2\left(\frac{1+\sqrt{5}}{2}\right) \approx 0.6942. \tag{4.4}$$

Next the possible partial extensions are identified. Since, as the depth of the tree grows, there are an infinite number of possibilities, the discussion will be limited to partial extensions with a maximum depth less than or equal to three. For coding purposes, it is only necessary to identify the unique sets of codeword lengths, and not the exact tree structure or codeword assignment, as they have no bearing on code rate. Recall from Chapter 3, the rate of a VL code for an i.i.d binary source depends only on sourceword lengths l_i and codeword lengths o_i. Any code tree that results in the same sourceword length to codeword length mapping will have the same code rate. In Table 4.5, the codeword lengths resulting from all possible partial extensions up to a depth of three are listed.

Note that for any of the sets of codeword lengths identified in Table 4.5, it is possible to create a tree structure based on the greedy tree construction outlined in §3.2. Given these codeword lengths and the underlying tree structure, it is possible to assign sourcewords to codewords based on Huffman coding each of the sets of codeword lengths outlined in Table 4.5. It is then possible to identify the code with the highest resulting code rate. Since the minimal set for this example has codewords the same length as the minimal set for a $(0, 1)$ constraint, then according to Table 4.4, the highest rate code, considering partial extensions no longer than three, has sourcewords of lengths $[2, 2, 2, 3, 3]$ and codewords of lengths $[3, 3, 3, 4, 4]$. This set of codeword lengths has been identified with an '*' in Table 4.5.

Codeword lengths
$[1,2]$
$[1,3,4]$
$[2,2,3]$
$[2,3,3,4]$
$[1,3,5,6]$
$[1,4,4,5]$
$[2,2,4,5]$
$[2,3,3,4]$
$[2,3,3,5,6]$
$[2,3,4,4,5]$
$[3,3,3,4,4]^*$
$[1,4,5,5,6]$
$[2,3,4,4,5]$
$[2,3,4,5,5,6]$
$[2,4,4,4,5,5]$
$[3,3,3,4,5,6]$
$[3,3,4,4,4,5]$
$[2,4,4,5,5,5,6]$
$[3,3,4,4,5,5,6]$
$[3,4,4,4,4,5,5]$
$[3,4,4,4,5,5,5,6]$

Table 4.5: Sets of possible codeword lengths for partial extensions of the $(1,\infty)$ minimal set with a maximum depth of three

This code has an average rate of:

$$\begin{aligned}\bar{R} &= \frac{\sum_i (2^{-l_i} * l_i)}{\sum_i (2^{-li} * o_i)} \\ &= \frac{0.25(2+2+2)+0.125(3+3)}{0.25(3+3+3)+0.125(4+4)} = \frac{2.25}{3.25} \\ &= 9/13 \approx 0.6923\end{aligned} \tag{4.5}$$

The efficiency of this code is:

$$\begin{aligned}E &= \frac{\bar{R}}{C_{(1,\infty)}} = \frac{9}{13C_{(1,\infty)}} \\ &\simeq 0.9972\end{aligned} \tag{4.6}$$

In other words, on average, this code operates within 0.28% of capacity. A complete codeword to sourceword code tree for this code is pictured in Figure 4.2. A look-up table for this code was given in Chapter 1, Table 1.2.

The bounds on code rate were discussed in Chapter 1. The code can operate at a rate no worse than 2/3 at any point in time, but could potentially operate at rates as high as 3/4. With equiprobable source bits, after a sufficient number of encoded words the overall average code rate will converge toward the expected average rate of 9/13.

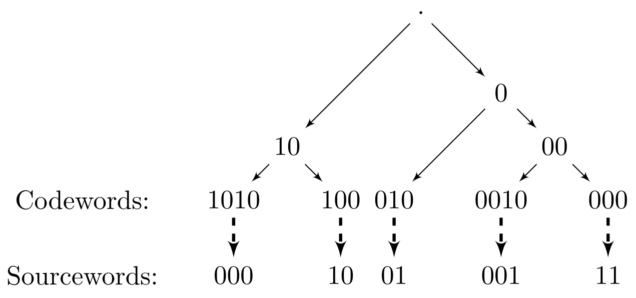

Figure 4.2: Code tree for a $(1, \infty)$ code

Example: $(0, 2)$ code

A minimal set for this constraint is: $\{1, 01, 001\}$. The largest real root to the equation $z^{-1} + z^{-2} + z^{-3} - 1 = 0$ is approximately 1.8393, resulting in a capacity of 0.8791 bits/bindig.

After exhaustively searching all partial trees with a maximal depth of three, the optimal sourceword to codeword length mapping for the $(0, 2)$ constraint was determined, and is listed in Table 4.4. A code with these lengths is outlined in Table 4.6. Other sourceword to codeword mappings for this constraint are possible, however this choice of word length mappings is optimal in the sense that no other word length mappings for an i.i.d binary source, constructed based on a partial extension tree depth no larger than three, result in a higher average rate. The specific assignment of sourcewords to codewords has been chosen arbitrarily, but could instead be optimized for other performance metrics such as error propagation, implementation complexity, etc.

Input	Output
0	1
1000	0101
1101	01001
1110	00101
1111	01101
10101	001001
11000	011001
11001	001101
10110	011101
10111	001111
100100	011111
101000	0011001
101001	0111001
100110	0011101
100111	0111101
1001010	00111001
1001011	01111001

Table 4.6: A $(0, 2)$ code representing an RLL constraint in NRZI notation

The rate for this code is:

$$\begin{aligned} \bar{R} &= \frac{2.8594}{3.2812} \simeq 0.8714 \\ \left(1 - \frac{\bar{R}}{C}\right) &* 100 = 0.876\% \end{aligned} \tag{4.7}$$

The bounds on code rate are between 4/5 and 1. As a result, the code can operate at a rate no worse than 4/5, representing 91% of capacity, but typically operates at much higher rates. The overall average code rate for a sufficient number of encoded bits will converge toward the expected code rate of approximately 0.8714.

In Chapter 2, an example of a $(0, 2)$ code given by Kerpez [2] was discussed. This code was constructed by Huffman coding the complete second order extension of the minimal set. As shown in Table 2.1, it has a total of nine codewords and a code rate of 0.862, which is roughly 98% of capacity. Contrasting this code to the code in Table 4.6, with slightly less than double the number of code words, it is possible to improve the efficiency by over 1%.

Now consider all of the possible partial extensions with the same number of codewords as the code constructed by Kerpez. It can be asked whether or not there is a partial extension with the same number of codewords that produces a higher code rate. As demonstrated in Table 4.7, the answer is yes. This code was identified by modifying the procedure to examine all possible partial extensions with exactly nine codewords, and in this case the resulting code requires no more than three partial extensions. This code has a code rate of $\frac{2.3438}{2.7031} \simeq 0.8671$, which is a 0.5% improvement over the Kerpez code. This example clearly demonstrates the advantages of using partial extensions over complete extensions. The bounds on code rate for this code are exactly the same as the bounds for Kerpez's code. The minimum bound is 3/4, while the upper bound is 1. For a sufficient number of encoded bits, the overall average code rate converges toward the expected code rate of 0.8671.

Input	Output
0	1
100	100
111	1010
1011	10100
1100	10110
1101	10111
10101	101100
101000	101110
101001	1011100

Table 4.7: High-rate $(0, 2)$ code with nine codewords constructed using partial tree extensions

Example: $(2,5)$ **code**

A minimal set for this constraint is: $\{001, 0001, 00001, 000001\}$. The capacity of this constraint is approximately 0.465 bits/bindig.

The optimal word lengths from Table 4.4 were used to develop the $(2,5)$ code in Table 4.8. Once again, the codeword assignments are non-unique and have been chosen arbitrarily, but could be optimized for a specific application based on other parameters of interest.

Input	Output	Input	Output
01	0001	001111	0000100001001
000	000001	110100	0010010001001
111	0010001	110111	00100001000001
1010	001000001	0010000	00001001000001
1011	000010001	0010001	00001000010001
1100	001001001	0010110	00100100010001
10010	00001000001	0010111	000010000100001
10011	00100100001	0011000	001001000100001
10000	00100001001	0011001	000010010001001
10001	00001001001	1101011	0000100001000001
001001	001001000001	1101100	0010010001000001
001010	001000010001	1101101	0000100100010001
001101	0010000100001	11010100	00001001000100001
001110	0000100100001	11010101	000010010001000001

Table 4.8: A $(2,5)$ code representing an RLL constraint in NRZI notation

The rate of this code is:

$$\begin{aligned} \bar{R} &= \frac{3.8359}{8.2852} \simeq 0.463 \\ \left(1 - \frac{\bar{R}}{C}\right) &* 100 = 0.42\% \end{aligned} \tag{4.8}$$

The bounds on code rate are between 3/7 and 1/2.

4.3 DC-Free Codes

It is also possible to use the same technique to construct DC-free codes. For a review of DC-free constraints, see Chapter 1. Strictly speaking, for a given bound on the RDS, and an initial value for the RDS z_0, there exist perfectly valid DC-free sequences that do not return to z_0 but still remain within the bound. These sequences cannot be represented by a finite size minimal set. However, the longer the sequences are, the less often they occur in the maxentropic output. We can therefore choose a minimal set that encapsulates the most probable constraint satisfying sequences such that a significant majority of the most likely sequences are represented within the minimal set. These sets are "incomplete" because they are not exhaustive of all possible constraint satisfying sequences, and as a result the capacity of the minimal set is strictly less than the capacity of the constraint. However, if the code rate is high enough, and the capacity of the minimal set is close enough to the capacity of the constraint, then the resulting codes can still operate at an average rate very close to the capacity of the constraint.

The structure of minimal sets for DC-free constraints are different from the structure of (d, k)-like minimal sets. The construction approach considered here will examine zero-disparity minimal sets, where the value of the RDS is the same before and after emitting any of the variable-length codewords in the minimal set. Most DC-free block codes introduce codewords with disparity and then keep track of the RDS with a finite state machine based encoder in order to guarantee that the RDS is bounded. This approach could also be considered with variable-length codes, although the underlying technique would depend on a multiple state variable-length encoder, which is beyond the scope of this thesis.

Example: DC-free $N = 5$ code

Consider a DC-free sequence that allows the running digital sum to take on five different possible values. The capacity of this constraint is 0.7925 [1]. One possible (incomplete) minimal set is { 01, 10, 1100, 0011, 110100, 001011, 11010100, 00101011, 1101010100, 0010101011}. This set is incomplete because, as noted above, there exist valid unbalanced sequences that remain within the RDS bounds. However, as the length of these sequences increases, their probability decreases. This minimal set comprises the balanced sequences that are most likely to occur.

Note that for every codeword in this minimal set there is another codeword in the minimal set that is the same except the values for '1' and '0' have been swapped. This is because the codewords in this minimal set are zero-disparity and therefore if the values for '1' and '0' are swapped, the resulting codeword is also zero disparity. For zero-disparity codewords, if the initial value of the RDS is assumed to be 0, then the RDS after any of the codewords is emitted is also 0. These zero-disparity codewords are also independent, as they can be concatenated in any order without affecting the bound on the RDS. This minimal set was constructed by first considering sequences that deviate from the initial value of the RDS by a maximum of ± 1, then considering enough sequences that deviate from the initial value of the RDS by a maximum of ± 2 such that the capacity of the minimal set represents a significant portion of the capacity of the constraint. The range of possible values the RDS assumes is $N = 2 - (-2) + 1 = 5$.

Note, however, that the capacity of this set is only 0.7905, or roughly 99.74% of the capacity of a DC-free $N = 5$ constraint as given by (1.13). As a result, this choice of a minimal sets the limit of the best possible code rate to below 99.74% of capacity. This rate loss is due to the fact that certain sequences corresponding to valid paths in the constraint graph are not generated by this minimal set. For instance, '1101010100' is a valid constraint satisfying sequence that cannot be produced by concatenating elements from this minimal set. This choice of minimal set represents a compromise between better representing the constraint and the complexity introduced by having a large number of codewords in the minimal set. It also serves as an example of utilizing the proposed variable-length technique with minimal sets that do not fully represent all possible valid sequences of the constraint, a necessity for single-state implementations of DC-free constraints.

To reduce the size of the table, only partial extensions up to a maximal depth of two have been considered. Within these limitations, a code with the best rate is displayed in Table 4.9. The rate of this code is:

$$\begin{aligned} \bar{R} &= \frac{4.3833}{5.5689} \simeq 0.7871 \\ \left(1 - \frac{\bar{R}}{C}\right) &* 100 = 0.68\% \end{aligned} \tag{4.9}$$

The bounds on code rate for this code are 0.75 and 0.875.

Despite the initial choice of a minimal set that was incapable of achieving capacity, the resultant code is still less than 1% away from capacity. Note that this code is also a RLL DC-free code, as there are never more than four like-valued symbols in a row. Since this is true for all DC-free $N = 5$ codes, the capacity of a $(d, k, N) = (0, 3, 5)$ code is the same as the capacity of an $N = 5$ code.

4.4 Distribution Transformers

Although the properties and techniques outlined in this chapter are intended for use in constrained sequence code design, they may have a separate use in efficient probability transformer design. In this case, the minimal set is no longer a set of binary constraint satisfying sequences, but an N-ary alphabet. As discussed in Chapter 2, probability transformers play a key role in the design of other variable-length constrained sequence codes. A key difference in this approach is that the effective codeword lengths for the minimal set are a function of the desired probability distribution, rather than the other way around.

In order to design an efficient distribution transformer, let the probability distribution of the minimal set be equal to the desired probability distribution, and let $\lambda_{max} = 2^{H(x)}$, where $H(x)$ is the entropy of the desired probability distribution.

Example

Consider a probability transformer with a desired probability distribution of $p(A) = 0.6$ and $p(B) = 0.4$. The entropy of this distribution is:

$$\begin{aligned} H(x) &= -0.6 \log_2(0.6) - 0.4 \log_2(0.4) \\ &= 0.971 \end{aligned} \tag{4.10}$$

Input	Output	Input	Output
011	1100	0000001010	011101010100
100	0011	0000001011	010010101011
101	1010	0000001000	011011010100
110	1001	0000001001	011000101011
00001	110100	0000001110	010111010100
01000	001011	0000001111	010100101011
01001	101100	0000001100	010111001100
00110	100011	0000001101	010111000011
00111	011100	0000010010	010100111100
00100	010011	0000010011	010100110011
00101	011010	0000010000	010100110110
00010	011001	0000010001	010100110101
00011	010110	11100011010	01101101010100
01010	010101	11100011011	01100010101011
111001	11010100	11100011000	01011101010100
111110	00101011	11100011001	01010010101011
111111	10110100	11100011110	01011100110100
111100	10001011	11100011111	01011100001011
111101	01110100	11100011100	01010011110100
111010	01001011	11100011101	01010011001011
111011	01101100	00000101110	01010011011100
0000000	01100011	00000101111	01010011010011
11100000	1101010100	0000010110001	0101110011010100
11100001	0010101011	0000010110100	0101110000101011
01011010	1011010100	0000010110101	0101001111010100
01011011	1000101011	0000010110010	0101001100101011
01011000	0111010100	0000010110011	0101001101110100
01011001	0100101011	0000010110110	0101001101001011
01011110	0110110100	00000101101111	010111001101010100
01011111	0110001011	000001011000010	010111000010101011
01011100	0101110100	000001011000011	010100111101010100
01011101	0101001011	000001011000000	010100110010101011
00000110	0101110010	000001011000001	010100110111010100
00000111	0101110001	000001011011100	010100110100101011
11100010	0101001110	0000010110111010	0101001101110101010100
0000010100	101101010100	0000010110111011	0101001101001010101011
0000010101	100010101011		

Table 4.9: A DC-free $N = 5$ code

The resulting value for λ_{max} is:

$$\lambda_{max} = 2^{H(x)} = 2^{0.971} \approx 1.96 \tag{4.11}$$

Using the relationship $p_i = \lambda_{max}^{-o_i}$ the equivalent codeword length distribution for symbols 'A' and 'B' is:

$$\begin{aligned} o_A &= -\log_{\lambda_{max}} p_A = 0.7590 \\ o_B &= -\log_{\lambda_{max}} p_B = 1.3615 \end{aligned} \tag{4.12}$$

Searching all partial trees with a maximal depth of four reveals that the following distribution for effective codeword lengths corresponds with the highest code rate:

$$o = [2.8795, 2.8795, 2.8795, 3.0361, 3.4820, 3.6385, 4.0844, 4.2410, 4.2410, 4.8434, 4.8434]. \tag{4.13}$$

Since the equivalent codeword lengths for the minimal set are non-integer in this example, the codeword lengths of the best partial extension, necessarily a linear combination of the lengths from the minimal set, are in this case non-integer as well. This is generally true for most distribution transformers.

A set of output codewords with these effective lengths can be determined by using the same greedy tree approach as before, noting the effective codeword lengths for symbols 'A' and 'B' as outlined in (4.12). Note that for this example it is possible to exactly generate the desired probability distribution. This is possible because the desired distribution can be exactly represented by a linear combination of sourceword probabilities, where sourceword probabilities are restricted to be powers of 1/2 due to the assumption of an i.i.d binary source. Note that in spite of the fact that the desired output distribution was achieved, the code operates at a rate below capacity because the maxentropic distribution for the codewords of the partial extension was not achieved. For example, the codeword 'AAB' has a maxentropic probability of $(0.6)^2 * 0.4 = 0.144$, however it has been assigned to a sourceword with a probability of $2^{-3} = 0.125$. There is a trade off between achieving the desired output probability distribution, achieving the desired maxentropic codeword probability distribution, and the resulting code rate.

The sourcewords for this code have been determined using a Huffman code as before. The complete code is outlined in Table 4.10.

Input	Output
000	AAB
001	ABA
010	BAA
011	AAAA
110	BBA
111	AAAB
1001	BBB
1010	ABBA
1011	BABA
10000	ABBB
10001	BABB

Table 4.10: A VL distribution transformer where $p(A) = 0.6$, and $p(B) = 0.4$

The rate of this code is:

$$\begin{aligned}\bar{R} &= \frac{3.3125}{3.4375} \simeq 0.9636 \\ \left(1 - \frac{\bar{R}}{H(x)}\right) &* 100 = 0.76\%\end{aligned} \tag{4.14}$$

The probability of symbol 'A' in the output is the probability of generating an 'A' divided by the average length of the output.

$$p(A) = \frac{2^{-3}(2+2+2+4+1+3)+2^{-4}(2+2)+2^{-5}(1+1)}{3.4375} = 0.6 \tag{4.15}$$

The probability of symbol 'B' is $1 - p(A)$.

4.5 Conclusion

In this chapter, the "optimal" sourceword length to codeword length mappings were given for several classes of minimal sets. The construction of these tables follows from the technique presented in Chapter 3. These mappings are optimal in that no other word length mapping within the stated search space can result in a higher rate code. These tables were then used to demonstrate code construction for $(1, \infty)$, $(0, 2)$, and $(2, 5)$ RLL constraints. Additionally, a high performance DC-free code was demonstrated. The use of this technique in the construction of probability transformers was also demonstrated. As discussed in Chapter 2, probability transformers are commonly used in other techniques of variable-length constrained sequence code design. It is interesting to note that all of the examples demonstrated here operate within 1% of capacity for their respective constraint, in spite of the fact that only a small number of partial extensions have been considered (in most instances only two, in some instances three, and four in the case of the distribution transformer). Moreover, all of the examples are quite simple: they contain a small number of sourceword/codeword mappings, and operate without the use of a complex state machine for encoding and decoding.

Chapter 5

Conclusion

This chapter summarizes the content of this thesis, and proposes directions for future research.

5.1 Summary

The purpose of this work was to demonstrate a new variable-length technique for constrained sequence code construction. In Chapter 1, constrained sequence codes were introduced. These codes are a type of channel code commonly used in modern digital storage and transmission systems, and often function as a form of "error prevention". This is contrast to other channel codes, which are typically designed for either error detection or error correction. Constrained sequence codes allow arbitrary sequences to be transformed into sequences that better suit the constraints of the channel. For example, if a channel is AC-coupled, then using a DC-free code will prevent the errors that would have occurred had the constrained code not been used. Alternatively, constrained sequence codes are sometimes used to more efficiently pack the desired information onto the channel. For example, a minimum-runlength constrained code can squeeze more channel bits into a given physical feature size than an unconstrained sequence. These codes necessarily operate immediately before the bits are placed on the channel. As a result, efficient utilization of the available channel is a key concern.

A high-efficiency constrained sequence code will operate at a code rate very close to capacity. The capacity of a constraint is the best possible code rate that can be achieved while still satisfying the constraints of the channel. While constrained sequence coding has been studied since the time of Shannon, the traditional use of block codes has provided few choices for improving code rate. In a block code, the only method available to improve the code rate is to change the block sizes for the input and output codewords. Since the capacity of a given constraint is fixed, and the code rate is the ratio of input block size to output block size, the choice of practical block codes that can be implemented within the constraints of modern technology may be somewhat limiting. As a result, the pursuit of high-efficiency block codes has often resulted in code designers modifying the underlying constraint rather than improving the code rate. For example, if the code rate is fixed, but the constraint satisfied by the code changes such that the new constraint has a smaller capacity, then the efficiency of the code increases, where efficiency is defined as the ratio of code rate to capacity. Unfortunately, this increase in efficiency is misleading if the limits of the channel are more accurately represented by the original constraint.

Variable-length constrained sequences codes are useful because they provide another mechanism for improving code rate: sequence probabilities. Therefore, with variable-length codes it is possible to improve the efficiency of a code without resorting to modifying the constraint in question. This advantage is sometimes viewed as a disadvantage, because the instantaneous code rate of a variable length code is actually variable, and only the average code rate is "fixed". Additionally, the rate of VL constrained sequence codes is susceptible to changes in source statistics. If the source is not maxentropic, as is typically assumed during code design, then the rate of a variable length code may not converge to the desired average rate. In contrast, a block code will have the same rate regardless of the source statistics. However, even if a source is not maxentropic, the code rate of a VL code is bounded by the worst possible instantaneous rate and the best possible instantaneous rate. Therefore, if the source-statistics in question are of concern, code design should continue until the bounds on variable-length code rate are deemed acceptable. Alternatively, variation in source-statistics could be mitigated by using either an appropriate source code or a scrambler. Concerns about the use of padding schemes in VL code implementations are also misleading, as source data rarely aligns with the block sizes of an arbitrary block code, and therefore padding schemes are necessary when implementing either technique.

In Chapter 2, various existing techniques for VL constrained sequence code design were examined. These techniques have historically been based on one of three basic techniques: source codes used in reverse, synchronous variable-length codes based on Franaszek's recursive procedure, or distribution transformers. Source coding as a technique for constrained sequence code design has been understood for some time, dating back to Martin et al. [20], but more recently examined by Kerpez [2]. Synchronous variable-length codes were introduced by Franaszek [12, 3], and operate at a fixed code rate much like block codes. Techniques that rely on distribution transformers are relatively new, and stem from a universal constrained coding technique proposed by Bender and Wolf [16]. A potential issue with the latter technique is the design of the distribution transformers themselves, which has been largely ignored by the existing literature.

The variable-length code construction technique presented in Chapter 3 and Chapter 4 utilizes some aspects of all three of the basic techniques. The key advantage of this technique is that it provides a degree of freedom that is not found in some of the other approaches. This degree of freedom, provided by the use of partial extensions, allows the code rate to improve without significantly increasing the complexity of the code. For a minimal set of size N_0, a single partial extension increases the number of codewords by only $N_0 - 1$, as opposed to the exponential growth associated with complete partial extensions. The choice of which leaf to extend is crucial to the technique and the resulting code rate. Since the possible choices increase with every partial extension, a brute force search to determine the optimal choice is computationally burdensome during code design, particularly as the number of partial extensions increases. However, despite the computational cost of code design, the resulting code complexity is strictly less than or equal to the complexity of a complete extension of the same order, as has been proposed previously, and the code rate is necessarily equal or better. This is because the complete code extension is one of the possible choices for a partial extension.

This work also proves that using a Huffman code to determine the sourceword lengths that are to be mapped to the constrained codeword lengths is optimal in the sense that no other mapping can produce a higher rate for a given partial extension. This proof is significant because it demonstrates that this technique is optimal for finite length codewords, not just asymptotically as the codeword lengths grow. However, increasing the number or assignment of partial extensions can have a significant effect on code rate. No technique is known at this time for optimally determining the best choice of partial extension, so an exhaustive search technique has been proposed. Research into this area could reduce the computational burden of code design for this technique.

However, it is desirable in terms of encoder and decoder implementation to use fewer partial extensions during the design process, as this reduces the number and length of codewords in the code. Therefore, it could be argued that the computational burden associated with a higher number of partial extensions is not that critical since searching through large partial extension trees might be unwarranted. In fact, Chapter 4 provides examples for commonly used constraints where code rates within 1% of capacity are achieved by partial extensions with an order no higher than four. Such low orders are computable by the computers available today.

In particular, examples of (d, ∞), $(0, k)$ and (d, k) constraints were examined in Chapter 4. These codes can be used to construct run-length-limited codes with a minimum runlength of $d + 1$ and a maximum runlength of $k + 1$, when combined with the use of a differential encoder. Tables of optimal mappings for sourceword lengths to codeword lengths, given a limited number of partial extensions, were given.

Also, an example of the design of DC-free codes using this technique was also given. Most DC-free constraints cannot be fully represented by a finite minimal set of codewords. In spite of this fact, it is often possible to construct finite minimal sets corresponding to capacities within a percentage point or two of the actual capacity, and these minimal sets can be used in conjunction with this technique to generate high rate VL DC-free codes.

Finally, the use of this technique in the design of distribution transformers was also discussed. Distribution transformers were discussed in Chapter 2 where they were shown to be an integral part of alternative approaches to constrained sequence code design. As an example of a distribution transformer designed with the technique developed in this thesis, an example code that demonstrates the conversion of a i.i.d binary source to a source with symbol probabilities of 0.6 and 0.4 was presented.

5.2 Areas for Future Research

There are several possible areas for future research that would build upon the work presented in this thesis.

Error Propagation

As with most variable-length codes, error propagation during decoding is a potential concern. For block codes, a state-independent decoder limits a finite number of errors in the channel bits to a finite number of errors in the decoded bits. However, with variable-length techniques there is the potential for a single error to cause the decoder to lose synchronization with the appropriate sequence lengths and to introduce an indefinite number of errors in the decoded output. Judicious assignment of input sourcewords to output codewords will limit the effect of this error propagation to some extent, but the techniques necessary to do so are not well understood. Future research should explicitly examine the error propagation of VL constrained sequence codes, and propose an approach for mapping sourcewords to codewords that mitigates the effect of channel errors. Fortunately, real world implementations typically pad variable-length sequences into fixed length blocks (see Chapter 1), and therefore there is in practice a finite bound on the number of errors in the decoded output, although the bound can potentially be quite large.

Multi-state implementations

As discussed above, some constraints do not lend themselves to implementation in a VL state-independent encoder. There is no reason that the code construction technique developed in this thesis cannot be extended to multi-state state machines where the input-to-output mapping in each state is governed by a VL code, analogous to most commonly used multi-state block codes. The broad class of DC-free RLL codes was not explicitly examined in this thesis for this reason. A small finite minimal set of independent codewords does not generally have a capacity close enough to the actual capacity associated with these constraints to justify code design. To address this problem, it is desirable to extend the technique presented here to a multi-state state machine. Such a state machine accurately represents the constraint graph of, for example, DC-free RLL constraints, and therefore might result in higher-rate implementations than the single state implementation discussed in Chapter 4.

Partial Extensions

Clearly, the proper choice of partial extension during code construction is critical to achieving the best code rate. Unfortunately, exhaustively searching over all possible partial extensions to determine the best code rate is computationally difficult. For this reason, the exhaustive search is typically bounded to some maximal depth. Research into understanding why certain partial extensions generate higher code rates than others could result in a technique for determining the best partial extension without an exhaustive search, drastically reducing the computational complexity of code design.

Distribution Transformers

As discussed in Chapter 4, the technique presented here is also well suited to the design of N-ary distribution transformers. Given the critical role that distribution transformers play in other approaches to constrained sequence code design, and the limited study of distribution transformers in the literature, the use of this technique for this purpose should be examined in detail.

Spectral Analysis

The spectral properties of the VL constrained sequence codes presented in this thesis are largely unstudied. However, the spectral properties of VL codes in general have been studied by Cariolaro et al. [31], and a similar analysis could be performed on the codes presented here. The power spectral density of a code is particularly important for DC-free constraints, but is also relevant for RLL constraints. A comparison of the power spectrum of the VL codes developed in this thesis and the power spectrum of a maxentropic code would be particularly important.

Code rate variance

Initial simulations have verified that VL constrained sequence codes converge toward the average code rate for well behaved sources. However, the instantaneous code rate of the encoder varies from codeword to codeword. An analysis of the code rate probability distribution, as well as the behavior of the aggregate code rate over time, should be undertaken in order to develop a practical understanding of the expected real-world code rates for practical implementations of VL constrained sequence codes.

Non-ideal sources

The technique presented here has been based on the assumption of ideal i.i.d sources. This is clearly not the case if the code has been preceded by another code that explicitly introduces redundancy, such as an error correcting code. Also, most source data is inherently correlated to begin with, and the binary representation is rarely exactly equiprobable. It is possible to mitigate some of these issues by using a source code and/or a scrambler, but ideally the code construction technique developed in this thesis would be modified to directly account for source non-idealities.

5.3 Conclusion

The primary contribution of this thesis is in the development of a new method for constructing variable-length constrained sequence codes. The main idea is to identify a suitable minimal set of words that describe the constraint, and consider partial extensions of this minimal set until Huffman coding the leaves of the corresponding tree results in a sufficiently high-rate code. Huffman coding is used to identify the sourceword length to codeword length mappings of the code, and this technique has been shown to result in the highest average code rate for a given partial extension.

Empirical results have demonstrated that this technique can result in code rates that are within 1% of capacity for several common constraints. Additionally, the number of partial extensions necessary to achieve the demonstrated code rates corresponds to trees of a depth no larger than four. Several (d, k) constraints, typically used to construct runlength-limited codes, and a DC-free constraint, were examined in detail. The underlying technique is applicable to a broad class of constraints, and can be extended to multi-state implementations suitable for use with other classes of constraints. Moreover, the technique may also be used to construct the distribution transformers used in other approaches to constrained sequence code design.

Chapter 6

Addendum

This chapter summarizes a recent advancement in constrained sequence code construction that came to light after this thesis was written and defended, and its applicability to the content of this thesis.

6.1 Geometric Huffman Coding

The analysis presented in this thesis is based on the classical approach to Huffman coding: the probability of a merged codeword is set to be the sum of the probabilities of the two merged codewords. For instance, see the discussion regarding (3.31). This approach is required during the typical application of the Huffman algorithm for source coding in order to exactly reconstruct the original source, and this approach is also the one considered by Kerpez [2] in his work regarding the use of source codes to construct constrained sequence codes. Additionally, the analysis of Chapter 3, specifically the derivation regarding optimality, assumes the use of this approach.

Recently [32], it has been shown, as proven in this thesis, that the Huffman approach results in optimal constrained codes, given the code construction technique considered in this thesis. However, it has also been shown that the typical Huffman approach to merging is neither necessary nor optimal for constrained sequence code construction. The authors of [32] demonstrate that an alternative merging condition can result in higher-rate codes. Specifically, within the context of the codes discussed in this thesis, the appropriate definition for merging codewords of length o_j and o_k, where $o_j \leq o_k$, is:

$$o_i^* = \begin{cases} \frac{o_j + o_k}{2} - \frac{1}{R} & , o_k - \frac{2}{R} < o_j \\ o_j & , \text{otherwise} \end{cases}, \tag{6.1}$$

where o_i^* is the equivalent codeword length of the merged codeword and R is the rate of the overall code. Note that the second condition implies removing the codeword associated with o_k if it is excessively long. The motivation for this definition is discussed in the next section.

An implication of this approach to merging is that a valid output sequence may be ignored if it is significantly longer than the next longest valid output sequence. Since merged sequences are assigned source sequences of the same probability, this prevents overly long sequences from reducing the overall code rate. However, a difficulty with this technique is that it requires prior knowledge of the code rate, which is not defined until after the encoding process. To deal with this problem, the authors of [32] propose an iterative approach to constrained sequence code construction known as Normalized Geometric Huffman Coding (nGHC) [33].

6.2 Motivation

The motivation for the merging technique in (6.1) is simple. The final code will, on average, translate a single input bit into $1/R$ binary digits. The merging process results in the sourcewords assigned to the two codewords that are merged to be one bit longer than the length of the sourceword that would "effectively" be assigned to the merged codeword. Therefore, the codewords that are merged should have an average length that is exactly $1/R$ longer than the effective length of the merged codeword. In other words, the resulting merged codeword should be assigned an effective codeword length equal to the average length of the two merged codewords minus $1/R$. However, if the merging process does not result in a decrease in length, as is the case when o_k is too large, then the longer codeword should simply be discarded. Similar to the analysis performed in this thesis, the alternative merging technique also assumes that the source is binary and maxentropic. Formally, this technique is shown to minimize the Kullback-Leibler distance between the sourceword probabilities and the maxentropic codeword probabilities [32], which is equivalent to maximizing the average code rate.

6.3 Implication

Note that for both classical and geometric merging, the basic Huffman technique, which merges the two least probable codewords in each iteration, remains optimal. For classical merging, the proof is outlined in Chapter 3, while for the new non-classical approach, the proof by Böcherer and Mathar [32] demonstrates that geometric merging minimizes the Kullback-Leibler distance and therefore maximizes the code rate. Using either merging approach, the overall code construction presented in this thesis remains the same: partial extensions of a minimal set can be exhaustively searched over a given search space, encoded using the Huffman technique, and the highest rate code can be selected.

Of all of the examples presented in Chapter 4, only three result in different code rates when using the alternative "geometric" merging approach. The optimal mappings resulting from the alternative approach for these three examples are presented in Table 6.1. Note that for the cases examined in Chapter 4, the optimal partial extension remains the same; only the source word mapping has changed. Also, compared to the results obtained in Chapter 4, the alternative merging approach results in an improvement to the code efficiency by no greater than 0.0008%.

$\bar{R}$		0.6755	0.6755		0.6815	0.6815		0.6872	0.6872
E		99.4401%	99.4405%		99.4547%	99.4555%		99.4593%	99.4595%
	L_{out}	Old L_{in}	nGHC L_{in}	L_{out}	Old L_{in}	nGHC L_{in}	L_{out}	Old L_{in}	nGHC L_{in}
	3	2	2	3	2	2	3	2	2
	4	3	3	4	3	3	4	3	3
	5	3	3	5	3	3	5	3	3
	6	4	4	6	4	4	6	4	4
	6	4	4	6	4	4	6	4	4
	6	4	4	6	4	4	7	5	5
	7	5	5	7	5	5	7	5	5
	7	5	5	7	5	5	7	5	5
	7	5	5	7	5	5	7	5	5
	8	5	5	8	6	6	8	6	6
	8	6	6	8	6	6	8	6	6
	8	6	6	8	6	6	8	6	6
	9	6	6	9	6	6	8	6	6
	9	6	6	9	6	6	9	6	6
	9	6	6	9	6	6	9	6	6
	9	6	6	9	6	6	9	6	6
	10	7	7	9	6	6	9	6	6
	10	7	7	10	7	7	10	7	7
	10	7	7	10	7	7	10	7	7
	10	7	7	10	7	7	10	7	7
	10	7	7	10	7	7	10	7	7
	11	7	7	10	7	7	10	7	7
	11	7	7	11	7	7	11	8	8
	11	7	7	11	8	8	11	8	8
	11	8	8	11	8	8	11	8	8
	12	8	8	11	8	8	11	8	8
	12	8	8	11	8	8	11	8	8
	12	8	8	12	8	8	11	8	8
	12	8	8	12	8	8	12	8	8
	13	9	9	12	8	8	12	8	8
	13	9	9	12	8	8	12	8	8
	13	9	9	13	9	9	12	8	8
	14	9	9	13	9	9	12	8	8
	14	**10**	**9**	13	9	9	12	8	8
	15	10	10	13	9	9	12	8	8
	15	**10**	**11**	14	9	9	13	9	9
	16	**10**	**11**	14	**10**	**9**	13	9	9
				14	10	10	13	9	9
				15	10	10	13	9	9
				15	10	10	13	9	9
				16	**10**	**11**	13	9	9
				16	**11**	**12**	13	9	9
				17	**11**	**12**	13	9	9
							14	10	10
							14	10	10
							14	10	10
							14	10	10
							14	10	10
							14	10	10
							14	10	10
							15	10	10
							15	10	10
							15	10	10
							15	10	10
							15	10	10
							15	10	10
							15	10	10
							16	11	11
							16	11	11
							16	11	11
							16	11	11
							16	11	11
							16	11	11
							17	12	12
							17	12	12
							17	12	12
							17	12	12
							17	12	12
							18	12	12
							18	12	12
							18	12	12
							18	**13**	**12**
							19	13	13
							19	13	13
							19	13	13
							19	13	13
							20	14	14
							20	14	14
							20	14	14
							21	**14**	**15**
							21	**14**	**16**
							22	**14**	**16**

Table 6.1: Changes to the sourceword to codeword mappings from Chapter 4 resulting from the use of the alternative geometric merging rule. The changes in source word lengths are shown in bold.

6.4 Conclusion

Recently it has been demonstrated that an alternative merging approach combined with the Huffman construction technique can result in higher rate codes than the classical merging approach considered in this thesis. However, the overall construction technique still follows the basic Huffman approach; only the definition for the probability of the merged symbol changes. For the cases examined within this thesis, the alternative approach makes little or no improvement to the code rate. In the three instances where an improvement was observed, the improvement in code efficiency was less than 0.0008%.

Bibliography

[1] K. A. S. Immink, *Codes for mass data storage systems*, 2nd ed. Shannon Foundation Publisher, 2004.

[2] K. J. Kerpez, "Runlength codes from source codes," *IEEE Trans. Inf. Theory*, vol. 37, no. 3, pp. 682–687, May 1991.

[3] P. A. Franaszek, "Run-length-limited variable length coding with error propagation limitation," U.S. Patent 3 689 899, Sept., 1972.

[4] C. E. Shannon, "A mathematical theory of communication (Parts III-V)," *Bell Syst. Tech. J.*, vol. 27, pp. 623–656, Oct. 1948.

[5] ——, "A mathematical theory of communication (Parts I and II)," *Bell Syst. Tech. J.*, vol. 27, pp. 379–423, July 1948.

[6] T. M. Cover and J. A. Thomas, *Elements of information theory.* New York, NY, USA: Wiley-Interscience, 1991.

[7] J. Kemeny and J. Snell, *Finite markov chains.* London: van Nostrand, 1960.

[8] K. A. S. Immink, "Block-decodable runlength-limited codes via look-ahead technique," *Philips J. Res.*, vol. 46, no. 6, pp. 293 – 310, 1991.

[9] P. A. Franaszek, "Sequence-state coding for digital transmission," *Bell Sys. Tech. J.*, vol. 47, no. 1, pp. 143 – 157, 1968.

[10] G. Pierobon, "Codes for zero spectral density at zero frequency (corresp.)," *IEEE Trans. Inf. Theory*, vol. 30, no. 2, pp. 435 – 439, Mar. 1984.

[11] J. Justesen, "Information rates and power spectra of digital codes," *IEEE Trans. Inf. Theory*, vol. 28, no. 3, pp. 457 – 472, May 1982.

[12] P. A. Franaszek, "Sequence-state methods for run-length-limited coding," *IBM J. Res. Dev.*, vol. 14, no. 4, pp. 376–383, 1970.

[13] ——, "On future-dependent block coding for input-restricted channels." *IBM J. Res. Dev.*, vol. 23, no. 1, pp. 75 – 80, 1979.

[14] R. Adler, D. Coppersmith, and M. Hassner, "Algorithms for sliding block codes—an application of symbolic dynamics to information theory," *IEEE Trans. Inf. Theory*, vol. 29, no. 1, pp. 5 – 22, Jan. 1983.

[15] D. Tang and L. Bahl, "Block codes for a class of constrained noiseless channels," *Inform. and Control*, vol. 17, no. 5, pp. 436 – 461, Feb. 1970.

[16] P. E. Bender and J. K. Wolf, "A universal algorithm for generating optimal and nearly optimal run-length-limited, charge-constrained binary sequences," in *Proc. IEEE Int. Symp. on Inform. Theory (ISIT'93)*, Jan. 1993, p. 6.

[17] S. Aviran, P. H. Siegel, and J. K. Wolf, "An improvement to the bit stuffing algorithm," *IEEE Trans. Inf. Theory*, vol. 51, no. 8, pp. 2885 – 2891, Aug. 2005.

[18] Y. Sankarasubramaniam and S. W. McLaughlin, "Capacity achieving code constructions for two classes of (d, k) constraints," *IEEE Trans. Inf. Theory*, vol. 52, no. 7, pp. 3333–3343, July 2006.

[19] S. J. P. Todd, G. G. Langdon, and G. N. N. Martin, "A general fixed rate arithmetic coding method for constrained channels," *IBM J. Res. Dev.*, vol. 27, no. 2, pp. 107–115, 1983.

[20] G. N. N. Martin, G. G. Langdon, and S. J. P. Todd, "Arithmetic codes for constrained channels," *IBM J. Res. Dev.*, vol. 27, no. 2, pp. 94–106, 1983.

[21] E. Arikan, "An implementation of Elias coding for input-restricted channels," *IEEE Trans. Inf. Theory*, vol. 36, no. 1, pp. 162–165, Jan. 1990.

[22] J. Ashley and P. Siegel, "A note on the Shannon capacity of run-length-limited codes (corresp.)," *IEEE Trans. Inf. Theory*, vol. 33, no. 4, pp. 601 – 605, July 1987.

[23] K. A. S. Immink, J.-Y. Kim, S.-W. Suh, and S. K. Ahn, "Efficient DC-free RLL codes for optical recording," *IEEE Trans. Commun.*, vol. 51, no. 3, pp. 326 – 331, Mar. 2003.

[24] S. Aviran, P. H. Siegel, and J. K. Wolf, "Optimal parsing trees for run-length coding of biased data," *IEEE Trans. Inf. Theory*, vol. 54, no. 2, pp. 841–849, Feb. 2008.

[25] B. Tunstall, "Synthesis of noiseless compression codes," Ph.D. dissertation, Georgia Institute of Technology, 1967.

[26] D. Howe and H. Hilden, "Shift error propagation in (2,7) modulation code," *IEEE J. Sel. Areas Commun.*, vol. 10, no. 1, pp. 223–232, Jan. 1992.

[27] J. S. Eggenberger and P. Hodges, "Sequential Encoding and Decoding of Variable Word Length, Fixed Rate Data Codes," U.S. Patent 4 115 768, 1978.

[28] Y. Sankarasubramaniam and S. W. McLaughlin, "Fixed-rate maximum-runlength-limited codes from variable-rate bit stuffing," *IEEE Trans. Inf. Theory*, vol. 53, no. 8, pp. 2769–2790, Aug. 2007.

[29] K. A. S. Immink, “Some statistical properties of maxentropic runlength-limited sequences.” *Philips J. Res.*, vol. 38, no. 3, pp. 138 – 149, 1983.

[30] D. A. Huffman, “A method for the construction of minimum-redundancy codes,” *Proc. IRE*, vol. 40, no. 9, pp. 1098 –1101, Sept. 1952.

[31] G. Cariolaro, G. Pierobon, and S. Pupolin, “Spectral analysis of variable-length coded digital signals,” *IEEE Trans. Inf. Theory*, vol. 28, no. 3, pp. 473 – 481, May 1982.

[32] G. Böcherer and R. Mathar, “Matching dyadic distributions to channels,” presented at the Data Compression Conf. (DCC’11), Snowbird, UT, Mar. 2011.

[33] G. Böcherer, “Geometric Huffman coding,” Aug. 2011. [Online]. Available: http://www.georg-boecherer.de/ghc.html