Design and application of variable-to-variable length codes

Dissertation

zur

Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

der Fakultät für Informatik und Elektrotechnik
der Universität Rostock

vorgelegt von
Heiner Kirchhoffer, geb. am 20.04.1979 in Hermeskeil
aus Berlin

Rostock, 23. Februar 2016

urn:nbn:de:gbv:28-diss2016-0035-1

Tag der Einreichung: 28. August 2015
Tag der öffentlichen Verteidigung: 16. Februar 2016

Gutachter:

Prof. Dr.-Ing. habil. Erika Müller
Universität Rostock, Institut für Nachrichtentechnik

Prof. Dr.-Ing. Thomas Wiegand
Technische Universität Berlin, Institut für Telekommunikationssysteme

Prof. Dr.-Ing. Sascha Spors
Universität Rostock, Institut für Nachrichtentechnik

Abstract

Contemporary video codecs feature sophisticated entropy coding techniques as an integral part. A widely used approach is the context-based adaptive binary arithmetic coding (CABAC) scheme as employed by the video compression standards H.264/AVC and H.265/HEVC. Arithmetic coding is famous for its ability to approach the entropy rate of a source with known properties arbitrarily close. However, it is also known for its highly sequential processing structure, which limits the achievable throughput. The probability interval partitioning entropy (PIPE) coding scheme replaces binary arithmetic coding with a number of binary coders, each optimized to a different fixed probability. This provides a basis for massive parallel processing. In this work, the options for using so-called variable-to-variable length (V2V) codes as binary coders are studied. In order to find V2V codes for the whole probability interval, the code design algorithms use a variable probability instead of a predefined fixed probability value for source symbols. This leads to the concept of polynomial-based algorithms, which is the basis for the design of numerous V2V codes with minimum redundancy for particular constraints. Exhaustive search is used as a general strategy, however several properties of V2V codes are exploited in order to greatly reduce the size of the search space. The derived V2V codes are the basis for the design of numerous PIPE coders for H.265/HEVC that have a low redundancy. Only 6 V2V codes of moderate size are required to cause an average bit rate overhead of less than 0.45%. This compares to approximately 0.09% average bit rate overhead for the M Coder as used in H.265/HEVC. An essential building block for enabling massive parallel processing in PIPE coding is the so-called chunk-based multiplexing, which contains several improvements over previous schemes. Interestingly, when combined with suitable V2V codes, a nonparallelized software implementation of chunk-based multiplexing for H.265/HEVC is already faster than an optimized implementation of the M coder.

Zusammenfassung

Hochentwickelte Entropiekodiertechniken sind ein wesentlicher Bestandteil moderner Videocodecs. Ein oft benutztes Verfahren ist das Context-based Adaptive Binary Arithmetic Coding (CABAC) Schema, das in den Videokodierstandards H.264/AVC und H.265/HEVC zum Einsatz kommt. Arithmetisches Kodieren wird wegen seiner Fähigkeit, die Entropierate einer Quelle mit bekannten Eigenschaften beliebig genau anzunähern, geschätzt. Jedoch ist es auch für seine stark sequenziell arbeitende Struktur bekannt, die den erreichbaren Datendurchsatz begrenzt. Im Probability Interval Partitioning Entropy (PIPE) Kodierverfahren wird das arithmetische Kodieren durch eine Anzahl binärer Kodierer ersetzt, von denen jeder auf eine andere Wahrscheinlichkeit optimiert ist. Damit wird die Grundlage für umfangreiche Parallelisierungsmöglichkeiten geschaffen. In der vorliegenden Arbeit werden die Möglichkeiten zur Verwendung sogenannter variable-to-variable length (V2V) Kodes in den binären Kodierern untersucht. Um V2V Kodes für das gesamte Wahrscheinlichkeitsintervall zu erhalten, wird in den Algorithmen zum Kodeentwurf eine variable Wahrscheinlichkeit anstelle eines festen Wahrscheinlichkeitswerts für die Quellsymbole verwendet. Dies führt zum Konzept der polynombasierten Algorithmen, welches die Basis für den Entwurf vielfältiger V2V Kodes mit minimaler Redundanz für bestimmte Randbedingungen ist. Die erschöpfende Suche kommt als generelles Prinzip zum Einsatz, jedoch werden hierbei mehrere Eigenschaften von V2V Kodes ausgenutzt um die Größe des Suchraums massiv zu reduzieren. Die so erzeugten V2V Kodes sind die Basis für den Entwurf einer Vielzahl von PIPE Kodierern mit niedriger Redundanz. Bereits mit 6 V2V codes moderater Größe wird ein mittlerer Bitratenüberschuss von nur $0{,}45\%$ erreicht. Im Vergleich dazu erzeugt der M Coder, wie er in H.265/HEVC benutzt wird, einen Bitratenüberschuss von $0{,}09\%$. Ein wesentlicher Bestandteil zur Verbesserung der Parallelisierungsmöglichkeiten im PIPE Kodiersystem stellt das sogenannte chunk-basierte Multiplexen dar, welches mehrere Verbesserungen gegenüber seiner Vorgänger enthält. Interessanterweise ist schon eine nicht parallelisierte Softwareimplementierung des chunk-basierten Multiplexens für H.265/HEVC, bei Verwendung geeigneter V2V Kodes, schneller als eine optimierte Implementierung des M Coders.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Erika Müller, for giving me the opportunity to pursue my doctorate at the University of Rostock. I would also like to express my sincere thanks to Professor Thomas Wiegand for enabling and supporting my doctoral research project at the Fraunhofer Institute for Telecommunications, Heinrich-Hertz-Institute (HHI) in Berlin. It always was an honor and pleasure to work in his research group. I am especially grateful to my mentor at HHI, Dr. Detlev Marpe, for his constant support, encouragement and guidance. I have greatly benefited from his profound expertise. I am very thankful to Dr. Jonathan Pfaff for spending his time in reviewing my thesis and for many valuable comments. I would like to thank my colleagues Dr. Heiko Schwarz, Dr. Martin Winken, Philipp Helle, Mischa Siekmann, Jan Stegemann, Christian Bartnik, André Roth, and Paul Haase for supporting me in many ways, for many inspiring discussions and for their valuable feedback. Very special thanks go to my parents, to my siblings and their families who provided moral and emotional support.

Contents

1

Introduction

The worldwide Internet traffic has seen a steady growth in recent years. Consumer Internet video is estimated to amount to 64% of the consumer Internet traffic in 2015 according to a study by Cisco [1]. The success of Internet video is made possible by efficient video compression algorithms like the widely used video coding standard H.264/AVC [2] of ITU-T and ISO/IEC. Any improvement in video compression efficiency can greatly contribute to a better exploitation of the available Internet infrastructure, which creates a persistent interest in new video compression schemes.

Moreover, the technological advance leads to increasing video resolutions and frame rates of video capture and playback devices, which corresponds to steadily increasing bit rates, making the task of video compression and decompression more and more demanding. This naturally leads to a desire for parallel processing capabilities since the operating frequency of a particular hardware encoder or decoder implementation cannot be increased arbitrarily.

During the standardization of H.265/HEVC [3], the most recent video coding standard of ITU-T and ISO/IEC, much effort has been spent to incorporate several techniques with regard to the above mentioned aspects. The compression efficiency is greatly improved relative to its predecessor H.264/AVC, and several techniques for parallelized encoding and decoding, like e.g. wavefront parallel processing, are included [4]. The entropy coding stage of H.264/AVC and H.265/HEVC, known as context-based adaptive binary arithmetic coding (CABAC)[5], substantially contributes to the compression efficiency. The binary arithmetic coding stage of CABAC, known as M coder, has a sequential processing structure with strong symbol to symbol dependencies, which makes the joint or parallel processing of binary symbols difficult to achieve.

The so-called *probability interval partitioning entropy* (PIPE) coding concept [6, 7, 8] evolved during the standardization process of H.265/HEVC, where it was proposed as entropy coding engine [9]. This scheme has the potential to improve the parallelization capabilities. While binarization, context modeling, and probability estimation techniques are inherited from CABAC, the M coder is replaced with a number of binary coders. The symbols to be encoded are distributed to the binary coders according to a partitioning of the probability interval and each coder is optimized to encode symbols with a predefined fixed probability. An efficient way of implementing binary coders is the use of so-called *variable-to-variable length* (V2V) codes, which map variable length words of symbols to be encoded to variable length code words.

In this thesis, the properties of V2V codes are studied and algorithms for the design of V2V codes are derived. Numerous V2V codes of different types are generated and their suitability for an application in the PIPE coding concept is discussed. Several exemplary PIPE coders are configured and their compression efficiency is compared to the M coder of CABAC. Moreover, a scheme for multiplexing the output of the binary coders, which allows a particularly high degree of parallelization, is presented.

1.1 Main contributions

- A canonical representation for V2V codes is defined and an algorithm for creating actual V2V codes from a canonical representation is presented. The so-called *prefix merge* algorithm is developed, which derives prefix-free codes for given code word lengths and given numbers of ones and zeros.
- The concept of prime and composite V2V codes is introduced. It is shown that the redundancy of composite V2V codes always lies between the redundancies of the included prime V2V codes. This can be used to sort out composite V2V codes in algorithms for code design. Furthermore, an algorithm for testing whether a canonical V2V codes is composite is developed.
- Instead of using numeric probability values in algorithms for code design, variable probability values are employed, which leads to the concept of polynomial-based algorithms. Such algorithms yield as result the set of all codes that can occur for all possible probability values. For each code in the set, a probability interval with analytical exact interval boundaries is also yielded. Such a probability interval specifies all probability values for that the original code design algorithm yields the same code, which also is the code that is associated with the interval.
- Numerous optimal limited-size V2V codes are derived along with exact probability intervals for which they are optimal. They are the basis for a low redundancy PIPE coding setup. Furthermore, so-called *systematic V2V codes* with simple structure are presented.
- A concept for designing PIPE coders based on a candidate set of V2V codes is presented. It allows the optimization of the PIPE coders to a given training set of e.g. H.265/HEVC-encoded bit streams. Furthermore, the so-called successive removal algorithm is presented, which has a low complexity, but may yield coders with a higher redundancy.
- Based on a training set of H.265/HEVC-encoded bit streams, the redundancy of the M coder of CABAC is derived and compared to the various PIPE coders.
- A multiplexing technique for PIPE coding, which is based on fixed-length bit sequences, is presented. It introduces numerous options for increasing the achievable throughput by parallel processing.

1.2 Organization of the thesis

In Chapter 2, the PIPE coding concept is reviewed. Based on the CABAC scheme, various options for replacing the M coder with a number of parallelizable binary coders are discussed. The influence on the redundancy is derived when the binary coders use ideal arithmetic coding and an example is given of how to realize binary coders with variable-length codes. Furthermore, the need for multiplexing of the output of the parallel binary coders is discussed.

In Chapter 3, the concept of variable-to-variable length (V2V) codes is analyzed. Several properties are derived and a canonical representation, the so-called *canonical V2V codes*, is defined. Based on this representation, the *prefix merge* algorithm is introduced, which allows the generation of a V2V code from a canonical representation. The canonical representation turns out to be valuable for efficiently designing V2V codes. Furthermore, the concept of prime and composite V2V codes is proposed and several properties are derived.

In Chapter 4, a new principle for designing variable-length codes is presented where the symbol probabilities are given as univariate polynomials. It can be applied to any algorithm that only uses summing, multiplication and comparison operations on probabilities. This is demonstrated for the Huffman algorithm, the Tunstall algorithm, and for the package merge algorithm. The result of running such an algorithm is a set of probability intervals and for each probability interval one associated code. For each such probability interval, the associated code is identical the code that is produced by executing the original algorithm for an arbitrary value in the associated probability interval. A polynomial-based algorithm yields the exact and unique number of different codes, the underlying original algorithm can produce for the probability interval.

In Chapter 5, the question of how to design optimal V2V codes is addressed. Of interest are the codes which have certain size limitations like a maximum source or code tree height. With such constraints, codes can be designed to suit a particular application. Procedures for finding such codes by exploiting the properties derived in Chapter 3 are described using the concept of polynomial-based algorithms of Chapter 4. A broad variety of optimal codes is derived, which can be used to design an actual PIPE coder. In addition, the so-called systematic V2V codes are discussed. They have simple construction rules and allow for particularly efficient software and hardware implementations.

In Chapter 6, the aspects of selecting V2V codes for a PIPE coder setup are studied.

Based on a training set of H.265/HEVC bit streams, a configurable bit rate criterion is derived, which is used for the subsequent V2V code selection procedure. Based on this criterion, several PIPE coders are constructed from the V2V codes of Chapter 5 by either exhaustive search or by a less complex V2V code selection algorithm. The bit rate overhead of these PIPE coders is compared to the M coder. Furthermore, the aspects of complexity-scalable entropy coding are discussed and a concept for increasing the throughput of a PIPE coder is reviewed.

In Chapter 7, multiplexing in PIPE coding is addressed. First, the basic principle of decoder-synchronized encoding is reviewed and based on this, the chunk-based multiplexing is developed and studied. The influence on memory utilization and parallelization options is compared for both schemes. An experimental evaluation concludes the chapter.

In Chapter 8, the insights gained in this thesis are summarized and several aspects for further research are discussed.

2

Probability interval partitioning entropy coding

Probability interval partitioning entropy coding (PIPE) is a lossless source coding technique [6, 7, 8] for arbitrary binary and nonbinary information for which statistical dependencies are known. The PIPE coding scheme is able to exploit these dependencies by context modeling based on a priori domain knowledge coming from the particular source coding application. The subsequent probability estimation is mainly based on universal prediction techniques [10]. So far, the process is basically the same as in CABAC of H.264/AVC and H.265/HEVC. The actual entropy coding is carried out in the coding stage that follows the probability estimation, and which shall be denoted *binary coding engine* (BCE) throughout this text. CABAC uses the M coder as binary coding engine, which is based on binary arithmetic coding, while PIPE coding uses a technique based on a partitioning of the probability interval, which shall be denoted *P coder* throughout this text.

2.1 Overview

A block diagram of a PIPE coder is depicted in Fig. 2.1. The encoder consists of a binarizer and context modeler, a probability estimator, and the P encoder. The decoder consists of corresponding stages. Input to the PIPE encoding process is a sequence of symbols of various types depending on the source coding application. In the case of a video codec like H.264/AVC or H.265/HEVC these are e.g. motion vector differences, quantized transform coefficient levels, control flags, etc. The binarization and context modeling stage converts the input symbols into a sequence of binary symbols,

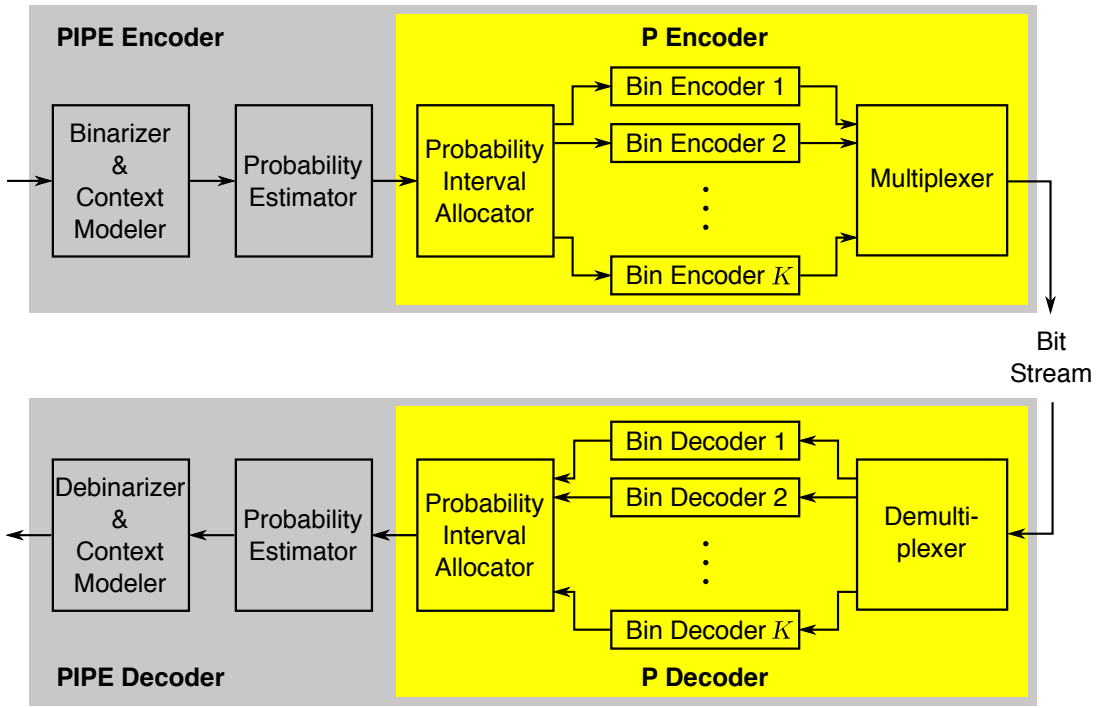

Figure 2.1: Block diagram of the PIPE coding scheme.

so-called *modeled bins*, and also carries out context modeling such that each bin is associated with one of many statistical models, which are usually referred to as *context models* or *probability models*. At this stage, dependencies are exploited by assigning modeled bins with similar statistical properties to the same probability model based on causal information. This may be any information which was previously encoded and which is thus available in the decoder where exactly the same context modeling needs to be carried out. This can be understood as an abstraction layer since the subsequent probability estimation is done independently for each of the probability models. I.e., only the modeled bins associated with a probability model as well as model-intrinsic parameters (like initialization values) are used to carry out the probability estimation. In order to avoid signaling cost for estimated parameters, backward-adaptive techniques are usually employed. The probability estimator of CABAC [5], for example, uses a state transition table. The underlying concept is probability estimation based on weighting past symbols. Further approaches with a higher complexity demonstrate that an improvement of the compression efficiency can be achieved [11, 12].

A sequence of modeled bins produced by the binarization stage can be described as a realization $\{w_i\}$ of a binary random process $\{W_i\}$. The probability estimator

derives conditional pmfs

$$p_{W_i}(w_i|w_{i-1},\ldots,w_1) = \Pr(W_i = w_i|W_{i-1} = w_{i-1},\ldots,W_1 = w_1). \tag{2.1}$$

For notational convenience, we shall denote a sequence $x_i, x_{i-1}, \ldots, x_1$ by $\mathbf{x}_i$ so that (2.1) can be written as

$$p_{W_i}(w_i|\mathbf{w}_{i-1}) = p_{W_i}(w_i|w_{i-1},\ldots,w_1). \tag{2.2}$$

For entropy coding, it is sufficient to design a coding system for symbols with a probability of one in the interval $(0, 0.5]$ since symbols with probability of one in the interval $(0.5, 1)$ can be negated before entropy encoding. Such a conversion is done in the probability estimation stage where each modeled bin w_i is converted into a so-called *coding bin* g_i according to the relationship

$$g_i = w_i \oplus u_i \tag{2.3}$$

where $\oplus$ denotes the *exclusive disjunction* and with

$$u_i = \begin{cases} 1 & \text{if } p_{W_i}(1|\mathbf{w}_{i-1}) > 0.5 \\ 0 & \text{otherwise.} \end{cases} \tag{2.4}$$

Consequently, $\{W_i\}$ is converted into a new random process $\{G_i\}$ with pmfs

$$p_{G_i}(x|\mathbf{g}_{i-1}) = \begin{cases} 1 - p_{W_i}(x|\mathbf{w}_{i-1}) & \text{if } u_i = 1 \\ p_{W_i}(x|\mathbf{w}_{i-1}) & \text{otherwise.} \end{cases} \tag{2.5}$$

The conditional probability of one $p_{G_i}(1|\mathbf{g}_{i-1})$ is in the interval $(0, 0.5]$ for all i. Therefore, $g_i = 1$ is also denoted *less probable symbol* (LPS) and $g_i = 0$ is denoted *more probable symbol* (MPS). u_i specifies the value of the MPS of coding bin g_i. In the decoder, w_i is restored by applying (2.3) to g_i, which also requires u_i. The output of the probability estimator are symbols g_i with associated conditional pmfs p_{G_i}, which are forwarded to the binary coding engine. Actually, the implementation of the probability estimator in CABAC is based on g_i and u_i instead of w_i. I.e., the probability estimator directly assigns to each symbol g_i a probability estimate $p_{G_i}(1|\mathbf{g}_{i-1}) \leq 0.5$, and also takes care of the exclusive disjunction in (2.3) when necessary.

The objective of the binary coding engine is to approach the entropy rate [13]

$$\bar{H}(\{G_i\}) = \lim_{n\to\infty} \frac{1}{n} H(G_1, G_2, ..., G_n). \tag{2.6}$$

This is a straightforward task since conditional pmfs are given and can directly be used to encode the coding bins. The backward-adaptive probability estimation process changes the conditional pmfs of $\{G_i\}$ from bin to bin, which must be taken into account in the binary coding engine. In CABAC this is achieved with the M coder by simply changing the probability for interval subdivision from bin to bin. The use of variable-length codes instead of the M coder is however difficult. The PIPE coding concept overcomes this problem by a subdivision of the binary probability interval $(0, 0.5]$ into K disjoint subintervals $I_k = (p_k, p_{k+1}]$ with $0 < k \leq K + 1$, $p_1 = 0$, and $p_{K+1} = 0.5$. Each symbol g_i can then be uniquely associated with one of the subintervals, depending on whether $p_{G_i}(1|\mathbf{g}_{i-1})$, i.e., the conditional probability of symbol one is in the subinterval. One distinct encoder, denoted *bin encoder*, is associated with each subinterval. An important aspect of the PIPE coding concept is that bin encoders do *not* regard the conditional pmfs of the individual symbols. Instead, all symbols associated with a particular subinterval are treated as if they come from a binary i.i.d. process. Each bin encoder produces code bits, which are multiplexed into a single bit stream by the multiplexer.

Complexity considerations

The M coder of CABAC approaches the entropy rate very closely while already having a relatively low complexity. However, the sequential nature of the M coder makes it difficult to realize a very high throughput or parallel processing. An obvious option for increasing the throughput is a higher operating frequency of the employed hardware architecture. Apart from this, universal concepts as e.g. speculative decoding [14, 15] give a rather small additional speedup. Even specific techniques like the *fast renormalization* [16] only achieve moderate improvements. Due to these limitations, a lot of research targets throughput and parallelization questions.

PIPE coding is an approach that addresses these problems since it transforms the task of encoding $\{G_i\}$ into a number of independent encoding operations, which can run in parallel. The number of parallel coders can be adapted to the requirements of the application and each parallel coder can process multiple symbols at once.

In order to take full advantage of PIPE coding, the binarization, context modeling, and probability estimation need to be adjusted so the P decoder is able to deliver sequences of symbols instead of individual symbols. Hence, no dependencies may exist between symbols of such a sequence of symbols.

2.2 Average code length per symbol

To evaluate a binary coding engines, the so-called *average code length per symbol* shall be defined. It corresponds to the expected description length per symbol used in [13, Eq. (5.39)], which is lower bounded by the entropy rate (2.6). Let B be a binary coding engine and let $\ell_B(\{G_i\}, (g_1, g_2, \ldots, g_n))$ be the so-called *encoded length* of a sequence $g_1, g_2, \ldots, g_n$ from random process $\{G_i\}$ when encoded with binary coding engine B. The *average code length per symbol* of encoding the random process $\{G_i\}$ with the binary coding engine B shall be defined as

$$\bar{L}(\{G_i\}, B) = \lim_{n\to\infty} \frac{1}{n} \sum_{(g_1,g_2,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1, g_2, \ldots, g_n)\ell_B(\{G_i\}, (g_1, g_2, \ldots, g_n)) \quad (2.7)$$

where $\mathcal{B} = \{0, 1\}$ is the binary alphabet with $\mathcal{B}^n$ being the n-fold Cartesian power of $\mathcal{B}$ and where $p_{\{G_i\}}$ is the joint pmf of $\{G_i\}$. Note that the limit does not need to exist. When

$$\ell_B(\{G_i\}, (g_1, g_2, \ldots, g_n)) = -\log_2 p_{\{G_i\}}(g_1, g_2, \ldots, g_n), \quad (2.8)$$

the average code length per symbol equals the entropy rate (2.6) of $\{G_i\}$.

2.3 Ideal binary arithmetic bin coders

In this section, an ideal binary arithmetic coder that operates at a fixed probability shall be used as a binary coding engine. Let $\hat{B}(Y)$ be an ideal binary arithmetic coder that can encode a binary random variable Y with zero redundancy. Such a binary coding engine $\hat{B}(Y)$ produces $-\log_2 p_Y(0)$ bits for encoding a zero and $-\log_2 p_Y(1)$ bits for encoding a one. Consequently, the encoded length of a sequence of bits only depends on the number of ones and zeros contained in the sequence as well as on p_Y. With $M(g_1, g_2, \ldots, g_n)$ being the number of ones in the sequence $g_1, g_2, \ldots, g_n$ and with $N(g_1, g_2, \ldots, g_n)$ being the number of zeros in the sequence $g_1, g_2, \ldots, g_n$, the encoded length of sequence $g_1, g_2, \ldots, g_n$ encoded with $\hat{B}(Y)$ is given as

$$\ell_{\hat{B}(Y)}(g_1, g_2, \ldots, g_n) = -M(g_1, g_2, \ldots, g_n)\log_2 p_Y(1) - N(g_1, g_2, \ldots, g_n)\log_2 p_Y(0). \quad (2.9)$$

Note that, in contrast to (2.8), it doesn't depend on the joint pmf of $\{G_i\}$. The average code length per symbol for the ideal binary arithmetic coder is yielded by substituting

(2.9) in (2.7). It can be written as

$$\begin{aligned}
&\bar{L}(\{G_i\}, \hat{B}(Y)) = \\
&- \log_2 p_Y(1) \lim_{n\to\infty} \frac{1}{n} \sum_{(g_1,g_2,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1, g_2, \ldots, g_n) M(g_1, g_2, \ldots, g_n) \\
&- \log_2 p_Y(0) \lim_{n\to\infty} \frac{1}{n} \sum_{(g_1,g_2,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1, g_2, \ldots, g_n) N(g_1, g_2, \ldots, g_n).
\end{aligned} \tag{2.10}$$

For $x \in \mathcal{B}$ define

$$\bar{p}_{\{G_i\}}(x) = \begin{cases} \lim\limits_{n\to\infty} \frac{1}{n} \sum\limits_{(g_1,g_2,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1, g_2, \ldots, g_n) M(g_1, g_2, \ldots, g_n) & \text{if } x = 1 \\ \lim\limits_{n\to\infty} \frac{1}{n} \sum\limits_{(g_1,g_2,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1, g_2, \ldots, g_n) N(g_1, g_2, \ldots, g_n) & \text{otherwise.} \end{cases} \tag{2.11}$$

Note that the limit in (2.11) exists if and only if the limit in (2.10) exists. Substituting (2.11) in (2.10) yields

$$\bar{L}(\{G_i\}, \hat{B}(Y)) = -\bar{p}_{\{G_i\}}(0) \log_2 p_Y(0) - \bar{p}_{\{G_i\}}(1) \log_2 p_Y(1). \tag{2.12}$$

Note that

$$\begin{aligned}
&\bar{p}_{\{G_i\}}(1) + \bar{p}_{\{G_i\}}(0) \\
&= \lim_{n\to\infty} \frac{1}{n} \sum_{(g_1,g_2,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1, g_2, \ldots, g_n) \left(M(g_1, g_2, \ldots, g_n) + N(g_1, g_2, \ldots, g_n)\right) \\
&= \lim_{n\to\infty} \frac{1}{n} \sum_{(g_1,g_2,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1, g_2, \ldots, g_n) n \\
&= \lim_{n\to\infty} \sum_{(g_1,g_2,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1, g_2, \ldots, g_n) = 1
\end{aligned} \tag{2.13}$$

and therefore $\bar{p}_{\{G_i\}}$ can be seen as pmf of a binary random variable. Thus, combining (2.12) and (2.13), we have proved the following Lemma.

Lemma 2.1. *If the pmf $\bar{p}_{\{G_i\}}$ exists for $\{G_i\}$, an ideal arithmetic coder operating at an arbitrary fixed pmf p_Y produces an average code length per symbol for $\{G_i\}$ equal to the cross entropy $H(Z;Y) = -p_Z(1) \log_2 p_Y(1) - p_Z(0) \log_2 p_Y(0)$, where Z shall be a binary random variable with pmf $\bar{p}_{\{G_i\}}$.*

□

Moreover, from (2.13) follows that (2.12) can be written as

$$\bar{L}(\{G_i\}, \hat{B}(Y)) = \bar{p}_{\{G_i\}}(0) \log_2 \frac{p_Y(1)}{p_Y(0)} - \log_2 p_Y(1). \tag{2.14}$$

The cross entropy $H(Z;Y)$ as used in Lemma 2.1 can be written as

$$H(Z;Y) = H(Z) + D(Z\|Y) \tag{2.15}$$

where $D(Z\|Y)$ is the Kullback-Leibler divergence [13, Eq. (2.26)].

Lemma 2.2. *If the pmf $\bar{p}_{\{G_i\}}$ exists for $\{G_i\}$, the average code length per symbol $\bar{L}(\{G_i\}, \hat{B}(Y))$ that an ideal binary arithmetic coder $\hat{B}(Y)$ produces for random process $\{G_i\}$ is minimal when $p_Y = \bar{p}_{\{G_i\}}$.*

Proof. Let Z be a binary random variable with pmf $\bar{p}_{\{G_i\}}$. The cross entropy is minimized when $D(Z\|Y) = 0$, which is fulfilled if and only if $p_Y = \bar{p}_{\{G_i\}}$ according to [13, Theorem 2.6.3]. □

2.4 Optimal probability interval derivation

In order to use ideal arithmetic bin coders with the PIPE coding concept, the probability intervals and the representative probabilities at which the bin coders operate need to be derived such that the average code length per symbol is minimized. For given probability intervals, the representative probabilities can unambiguously be derived by (2.11) due to Lemma 2.2. This takes into account which values of $p_{G_i}(1|\mathbf{g}_{i-1})$ fall in a particular interval and how the values are distributed. When, on the other hand, the representative probabilities are given, the optimal interval boundaries can be derived as follows. For two adjacent representative probabilities, the interval boundary in between these two values can unambiguously be derived by equating (2.12) for both values. Since (2.14) is a linear equation, there exists a unique solution which is the intersection point of two straight lines. These relationships between interval boundaries and representative probabilities suggest the use of the Lloyd-Max algorithm [17, 18] for iteratively deriving good probability intervals. An algorithm based on this idea is presented in [9].

In the case of CABAC, $p_{G_i}(1|\mathbf{g}_{i-1})$ can only attain one of 63 values (denoted *states*), which translates the problem of finding optimal intervals into deriving subsets of the states. Interestingly, an optimal solution to this problem can be found in publications related to context quantization [19, 20]. In these papers, the quantization of context states is carried out by first mapping each context state to a pmf. The subsequent quantization is then based on pmfs which is the same problem as finding optimal probability intervals for the PIPE coding system. Greene et al. [19] show that an optimal solution for this problem can be efficiently found by dynamic programming.

2.5 Bin coders based on variable-length codes

An alternative to binary arithmetic coding for bin coders are variable-length codes. In order to efficiently use variable-length codes for source coding, the individual symbol probabilities should not exceed $p = 0.5$. A simple and well-known example how to achieve this for a binary source is to regard fixed-length words of the source and assign a Huffman code [21] to all possible fixed-length words. The resulting code is e.g. known as fixed-to-variable length (F2V) code. By properly choosing the word length, a symbol probability of $p \leq 0.5$ can always be ensured.

In general, the source words don't need to be of fixed length. When also allowing shorter source words than of the fixed length, the redundancy of the resulting code can greatly be reduced as will be shown in Ch. 3. Such codes are denoted variable-to-variable length (V2V) codes (or dual-tree codes [22]). A binary tree is used to parse the binary source into source words. This binary tree corresponds to a prefix-free code and is commonly denoted source tree or parse tree. The prefix-free code that is used to assign code words to the source words is commonly denoted code tree. A further special case of V2V codes that use fixed-length code words are the so-called variable-to-fixed length (V2F) codes. The famous algorithm by Tunstall [23] yields optimal V2F codes. However, for comparable source and code tree size, Tunstall codes tend to have a higher redundancy than F2V or V2V codes as shown in Ch. 5.

2.6 Multiplexing

Distributing the bins to different coders creates a number of bit streams. These can either be stored and transmitted independently, or they can be multiplexed into a single bit stream. Since having a number of streams is for the most application undesirable, multiplexing is usually used for PIPE coding. Amongst the available multiplexing techniques, the ones that can be implemented without extra signaling cost of where to find which bits in the multiplexed bit stream are of particular interest. For V2V bin coders, an intuitive concept for this is the *decoder-synchronized encoding* [24]. The basic problem is that a bin encoder can output a codeword earliest when all bins which are encoded in this code word are known. However, the decoder requires the code word when the first bin is requested. In the decoder-synchronized encoding, the encoder interleaves code words of the various bin encoders so that the decoder can read a code word when the first bin of the code word is needed.

Binary arithmetic coding that operates at a fixed probability can also be used for

implementing bin coders. Of particular interest in this context is the so-called multi-MPS (more probable symbol) coding concept [25]. It allows the joint encoding and decoding of MPS sequences with low complexity. Decoder-synchronized encoding is in principle possible as well for binary arithmetic bin coders as described by [26]. However, it is difficult to realize in an encoder since the position of a code bit in the bit stream becomes known some time before the value of the code bit. The number of these *pending bits* can dynamically grow due to the phenomenon of *outstanding bits* as they can occur in arithmetic coding [5].

For bin coders where decoder-synchronized encoding is feasible, a number of further optimizations can be applied. It is e.g. possible to interleave fixed-length bit sequences instead of only single code words while still being decoder-synchronized [27, 28]. This may greatly increase throughput and parallelization capabilities.

Another important technique is the so-called low-delay control (see *fill symbol control* in [24]). In an encoder, it can happen that a source word takes, by chance, a very long time to complete. Low-delay control artificially completes code words when they are pending in the encoder for a long time by adding flushing bins. The decoder must detect such situations as well and discard corresponding flushing bins. Flushing bins also occur when the encoding of a sequence of bins is finished since all pending code words must be completed. The various aspects of multiplexing in the PIPE coding system are discussed in Ch. 7.

2.7 State of the art PIPE coding

The PIPE coding concept was originally developed as an alternative [29] to the state of the art CABAC scheme as found in the video compression standards H.264/AVC and H.265/HEVC. Many of the ideas behind the PIPE coding concept can be found in several former data compression schemes as discussed in the following.

A very early approach, described by Fano [30], splits a Markov source into a set of memoryless sources and assigns individual codes to each of them. This scheme is basically identical to conditional Huffman codes [8]. Since it produces one code word for each input symbol, the minimum achievable bit rate is limited to one bit per symbol and thus, it requires a multi-symbol alphabet preferably with a large number of source letters to compress well. The work by Ono et al. [24] addresses this problem by grouping symbols of a binary source. This introduces new difficulties related to the interleaving of the resulting code words for which Ono et al. present a solution as well [24]. The so-called decoder-synchronized encoding interleaves the output of several

parallel coders in a way that allows low complexity decoding. The work by Howard [26] generalizes the coding system of Ono et al. by allowing various families of run-length codes. Even if it is not explicitly discussed, the coding concept reveals the idea of partitioning the probability interval. Kiely and Klimesh [31, 32, 33] generalize the coding schemes by Ono et al. and Howard by introducing the separation of modeling and coding. Instead of considering a Markov source that is split into a set of memoryless sources, they formulate the idea of partitioning the unit interval into several narrow subintervals. Any probability modeling scheme may then be combined with the coding scheme and binary input symbols are decomposed into subsets according to the probability intervals. This is essentially the idea of the PIPE concept. Variations of the scheme by Ono et al. can also be found in the work of Macq et al. [34, 35] and Nguyen-Phi et al. [36] where the coding concept is combined with a probability estimation scheme based on a finite-state machine.

2.8 Chapter Summary

In this chapter, the PIPE coding concept is reviewed and the differences to CABAC are described. The aspects of using a plurality of fixed probability bin coders (as present in PIPE coding) instead of binary arithmetic coding (as present in CABAC) are discussed. In particular, the options for establishing parallel processing in a PIPE coding system are pointed out.

The average code length per symbol turned out to be a suitable quantity for evaluating the compression efficiency of a binary coding engine. It has the entropy rate as lower bound so that the difference between average code length per symbol and the entropy rate directly reveals how far from optimal a binary coding engine is.

It was shown that a random variable can be derived from the random process of the coding bins $\{G_i\}$ for that an ideal binary arithmetic bin coder, which operates at a fixed probability, produces an average code length equal to the average code length per symbol that it produces for $\{G_i\}$. This observation is the basis for a discussion of strategies for jointly deriving optimal ideal binary arithmetic bin coders and associated probability intervals for the use in a P coder. One approach is based on the Lloyd-Max algorithm, while another approach employs dynamic programming according to an algorithm by Greene et al.

An important aspect of the PIPE coding concept is the multiplexing of the output of the individual bin coders for that the state-of-the-art approach, the decoder-synchronized encoding, is shortly reviewed.

An alternative to using fixed-probability binary arithmetic coding for the bin coders is to use variable-length coding. However, this requires the joint processing of coding bins, which leads to the concept of V2V codes. An in-depth analysis of the properties of V2V codes is the topic of the next chapter.

3

Variable-to-variable length codes

Variable-to-variable length (V2V) codes are a two stage coding concept for binary sources. Each stages uses a binary tree as prefix-free code, both trees having the same number of leaf nodes. There exists a bijective mapping between the leaf nodes of the binary tree of the first stage and the leaf nodes of the binary tree of the second stage. Encoding of binary input symbols is carried out by parsing them with the binary tree of the first stage into a sequence of so-called *source words* and writing the corresponding so-called *code words*[1] of the binary tree of the second stage to the bit stream. Decoding of the bit stream uses the same procedure, but with the two binary trees exchanged. A prefix-free code can be seen as a full binary tree, i.e., every node has 0 or 2 children. The binary symbols visited when traversing the tree from the root node to a leaf node form a binary word and a prefix-free code is fully described by the set of its binary words. In the context of V2V codes, the prefix-free code of the first stage is denoted *source tree* and it is unambiguously described by the set of source words $\mathcal{S} = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$. The prefix-free code of the second stage is denoted *code tree* and it is unambiguously described by the set of code words $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$. Since the source and code tree have the same number of leaf nodes, $|\mathcal{S}| = |\mathcal{C}|$. The symbol $\mathcal{S}$ shall interchangeably be used to refer to a source tree as well as to the set of source words of a source tree. Analogously, the symbol $\mathcal{C}$ shall interchangeably be used to refer to a code tree as well as to the set of code words of a code tree.

[1] Although the term *code words* is generally used for prefix-free codes, it shall only be used for the binary tree of the second stage of a V2V code in this work.

Source tree

The source tree is used to convert the binary input into a sequence of source words by parsing binary source symbols until a source word σ_i is matched and by repeating this process until the end of the binary sequence is reached. A potential incomplete source word can then be completed by adding arbitrary stuffing bits until a code word is complete. Depending on the application, it may be necessary to also signal the number of added stuffing bits in order to be able to remove them at the decoder. However, this shall be neglected in this work. For the subsequent discussion, the binary input is assumed to come from an i.i.d. binary random process $\{X\}$. Encoding $\{X\}$ with a source tree $\mathcal{S}$ creates a random variable S over the source words $\mathcal{S}$ and an i.i.d. random process $\{S\}$ over $\mathcal{S}$. The pmf $p_S(\sigma)$ depends on the number of ones and zeros in the source word σ and on the pmf of X. Let x be a binary word. With $M(x)$ being the number of ones, and $N(x)$ being the number of zeros in x, the source word pmf $p_S(\sigma)$ is given as

$$p_S(\sigma) = p_X(1)^{M(\sigma)} p_X(0)^{N(\sigma)}. \tag{3.1}$$

The expectation value of the lengths of the source words shall be denoted *average source word length* and it is defined as

$$\bar{\ell}(S) = \sum_{\sigma \in \mathcal{S}} p_S(\sigma)\ell(\sigma) \tag{3.2}$$

where $\ell(\sigma)$ is the length of source word σ. If a symbol of $\{S\}$ contains on average $\bar{\ell}(S)$ symbols of $\{X\}$, such a symbol of $\{S\}$ must on average also contain $\bar{\ell}(S)$ times the average information of symbols of $\{X\}$, i.e., the entropy

$$H(S) = \bar{\ell}(S)H(X). \tag{3.3}$$

Code tree

For encoding $\{S\}$, a code word is assigned to each source word using a code tree. The source words are bijectively mapped to the code words using a mapping function

$$k : \mathcal{S} \leftrightarrow \mathcal{C}. \tag{3.4}$$

Such a function is uniquely described by an index permutation function π which bijectively maps to source word with index i a code word with index $j = \pi(i)$. A random variable C can be derived from S by simply replacing the alphabet $\mathcal{S}$ with $\mathcal{C}$ using the mapping function k. Consequently, this yields an i.i.d. random process $\{C\} = k(\{S\})$

over $\mathcal{C}$, which produces the sequence of code words. The concatenation of the output of $\{C\}$ is the entropy coded representation of the binary source $\{X\}$. The source tree $\mathcal{S}$, the code tree $\mathcal{C}$, and the permutation function π constitute a V2V code $v = (\mathcal{S}, \pi, \mathcal{C})$, a system for encoding binary sources.

3.1 State of the art V2V coding

Most of the literature related to V2V codes came up independently of the work around the PIPE coding concept. A comprehensive survey can be found in [37]. Some of the most relevant studies in this area are by Fabris [38], Freeman [22, 39], and Stubley [40, 41, 42]. These works mainly address the question of generating suboptimal V2V codes with possibly low redundancy since no fast algorithm is known to derive optimal V2V codes so far.

Fabris starts the analysis with a concatenation of Tunstall (V2F) [23] and Huffman (F2V) [21] codes which are both known to be asymptotically optimal (when the average code word length goes to infinity). However, if the maximum code word length (or some other size-limiting criterion) is fixed, there usually exist V2V codes which compress closer to the entropy than a Tunstall or Huffman code. Based on this observation, Fabris develops a greedy algorithm which starts with a V2V code consisting of a Tunstall source tree and derives a suitable Huffman code tree. Then he modifies the two trees in order to reduce the redundancy and thus yield a good V2V code which is however not always optimal. Since the code tree of an optimal V2V code is always a Huffman code, finding an optimal source tree appears to be the more challenging problem. Approaches for finding source trees are described by e.g. Tunstall, Stubley, or Freeman. They are based on growing the source tree from the root. In the approach by Freeman, leaf node splitting decisions are based on deriving the Huffman code length for each potential split and selecting the one minimizing the resulting redundancy. A further approach by Freeman is based on using a heuristic for how well a particular leaf node of the source tree is matched to the potential redundancy of a Huffman code word, i.e., the source leaf node probability is close to 2^{-i} for some $i \in \mathbb{Z}^+$. One approach by Stubley is based on the same idea but uses a different heuristic in order to be able to extend the scheme towards an adaptive V2V code which also incorporates probability modeling. Although only the exhaustive search finds optimal V2V codes, Stubley shows [40] that the suboptimal codes of some heuristic algorithms almost achieve the same low redundancy as optimal codes. Kiely and Klimesh [31, 32, 33] describe a coding concept which is related to V2V coding. It is based on the

idea that a code tree may not only use bits which are directly written to the bit stream, but also symbols which are encoded with other available V2V codes.

Designing V2V codes that have a low redundancy is not the only aspect of interest. Other parameters, like the achievable throughput, can be important as well. Senecal et al. [43] present a scheme for increasing the throughput of V2V codes by source string merging which increases the average number of bits per code word of a V2V code.

Relation to unequal letter cost coding

The design of V2V codes is related to coding with unequal letter cost. Many approaches for V2V code design are based on somehow determining a source tree and deriving for it an optimal code tree with the Huffman or package merge algorithm. An alternative approach is to start with a code tree and find for it an optimal source tree. This would make sense if the number of code trees to test in an exhaustive search is much smaller than the number of source trees to test. However, it would also require an algorithm which is able to find an optimal source tree for a given code tree. This is related to the concept of coding with unequal letter cost [44, 45, 46]. To the authors best knowledge, no polynomial-time algorithm is known for this problem. However, if this problem becomes solved in future, design of optimal V2V codes may become simpler as well.

3.2 Average code length and redundancy of V2V codes

The expected length of a code word is given as

$$\bar{\ell}(C) = \sum_{\sigma \in \mathcal{S}} p_S(\sigma) \ell(k(\sigma)) \tag{3.5}$$

and it shall be denoted *average code word length*. The average number of code word bits per binary source symbol of a V2V code v is given as ratio between average code word length and average source word length

$$\bar{\ell}_v(X) = \frac{\bar{\ell}(C)}{\bar{\ell}(S)} = \frac{\sum\limits_{\sigma \in \mathcal{S}} p_S(\sigma) \ell(k(\sigma))}{\sum\limits_{\sigma \in \mathcal{S}} p_S(\sigma) \ell(\sigma)}. \tag{3.6}$$

To distinguish it from the average lengths of source words and code words, it is denoted *average code length* of a V2V code. The redundancy of the code tree is given

as

$$R(C) = \bar{\ell}(C) - H(S) = \bar{\ell}(C) - H(C). \tag{3.7}$$

It can be rewritten as

$$\begin{aligned} R(C) &= \sum_{\sigma \in \mathcal{S}} p_S(\sigma)\ell(k(\sigma)) + \sum_{\sigma \in \mathcal{S}} p_S(\sigma)\log_2(p_S(\sigma)) \\ &= \sum_{\sigma \in \mathcal{S}} p_S(\sigma)\Big(\ell(k(\sigma)) + \log_2(p_S(\sigma))\Big) \\ &= \sum_{\sigma \in \mathcal{S}} p_S(\sigma)\Big(\log_2 2^{\ell(k(\sigma))} + \log_2(p_S(\sigma))\Big) \\ &= \sum_{\sigma \in \mathcal{S}} p_S(\sigma)\log_2 \frac{p_S(\sigma)}{2^{-\ell(k(\sigma))}}. \end{aligned} \tag{3.8}$$

A random variable $\hat{S}$ and accordingly an i.i.d. random process $\{\hat{S}\}$ shall be defined over $\mathcal{S}$, which a prefix-free code with code words $\mathcal{C}$ can encode with zero redundancy. I.e., its pmf is given as

$$p_{\hat{S}}(\sigma) = 2^{-\ell(k(\sigma))}. \tag{3.9}$$

The redundancy (3.8) can be rewritten as Kullback-Leibler divergence [13] (or relative entropy) of $\{\hat{S}\}$ from $\{S\}$

$$R(C) = \sum_{\sigma \in \mathcal{S}} p_S(\sigma)\log_2 \frac{p_S(\sigma)}{p_{\hat{S}}(\sigma)} = D(S\|\hat{S}). \tag{3.10}$$

It is well-known that $D(S\|\hat{S}) \geq 0$ with equality if and only if

$$p_S(\sigma) = 2^{-\ell(k(\sigma))} \tag{3.11}$$

for all σ (see [13, Theorem 2.6.3]). The average code length of V2V code v as given in (3.6) can be rewritten as

$$\bar{\ell}_v(X) = \frac{\bar{\ell}(C)}{\bar{\ell}(S)} \tag{3.12}$$

$$= \frac{H(S) + D(S\|\hat{S})}{\bar{\ell}(S)} \tag{3.13}$$

$$= H(X) + \frac{D(S\|\hat{S})}{\bar{\ell}(S)} \tag{3.14}$$

where (3.3) is substituted into (3.13). From (3.14) can be seen that $\bar{\ell}_v(X)$ is lower-bounded by the entropy of the binary source $H(X)$. The redundancy of a V2V code is given as

$$R_v(X) = \bar{\ell}_v(X) - H(X) \tag{3.15}$$

$$= \frac{D(S\|\hat{S})}{\bar{\ell}(S)} \tag{3.16}$$

$$= \frac{R(C)}{\bar{\ell}(S)}. \tag{3.17}$$

3.3 Nonzero redundancy of finite size V2V codes

It can be shown that there exists no V2V code (of finite size) with zero redundancy for binary source symbols with $0 < p_X(1) < 0.5$. Stubley shows in [40, Sec. 7.1.3] that there exist particular values of $p_X(1)$ for which the redundancy of a V2V code cannot be zero. In the following, it is proven by contradiction that the redundancy of a V2V code cannot be zero for $0 < p_X(1) < 0.5$ and $p_X(1) \in \mathbb{R}$.

Lemma 3.1. *Let $\mathcal{S} = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ be the set of source words of a source tree. There exists exactly one source word σ_α that only consists of ones and one source word σ_β that only consists of zeros.*

Proof. Since a source tree is a full binary tree, each node has 0 or 2 child nodes. Consequently, it is possible to traverse the tree from root by always selecting 1 as next symbol, yielding σ_α or by always selecting 0 as next symbol, yielding σ_β. □

Assume that there exists a redundancy-free V2V code with $0 < p_X(1) < 0.5$. Let σ_α and σ_β be as in Lemma 3.1 with source word lengths $a = \ell(\sigma_\alpha)$ and $b = \ell(\sigma_\beta)$. Substituting the probabilities of source words (3.1) for σ_α and σ_β into (3.11) yields

$$2^{-c} = p_X(1)^a \qquad \Longleftrightarrow \qquad 2^{-\frac{c}{a}} = p_X(1) \tag{3.18}$$

$$2^{-d} = (1 - p_X(1))^b \qquad \Longleftrightarrow \qquad 1 - 2^{-\frac{d}{b}} = p_X(1) \tag{3.19}$$

where c and d are the lengths of the code words to be associated with σ_α and σ_β, respectively. The following Lemma is taken from [47].

Lemma 3.2. *Let $a_1, a_2, ..., a_n, b_1, b_2, ..., b_n \in \mathbb{Q}^+$ and $k \in \mathbb{Z}^+$. A sum $\sum_{i=1}^{n} a_i \sqrt[k]{b_i} = M \in \mathbb{Z}$ cannot occur if at least one $\sqrt[k]{b_i}$ is irrational.*

Proof. See [47]. □

Theorem 3.1. *Let σ_α and σ_β be source words according to Lemma 3.1 with length $a = \ell(\sigma_\alpha) > 0$ and $b = \ell(\sigma_\beta) > 0$. For a binary random variable X, there exists no $p_X(1) \in \mathbb{R}$ with $0 < p_X(1) < 1/2$ for which* (3.11) *is fulfilled for σ_α and σ_β. Consequently, the redundancy of the V2V code cannot be zero.*

Proof. Theorem 3.1 requires $p_X(1) > 0$, which is always fulfilled for (3.18) and (3.19), and $p_X(1) < 1/2$ which yields

$$2^{-\frac{c}{a}} < \frac{1}{2} \qquad \Longleftrightarrow \qquad \frac{c}{a} > 1 \tag{3.20}$$

$$1 - 2^{-\frac{d}{b}} < \frac{1}{2} \qquad \Longleftrightarrow \qquad \frac{d}{b} < 1. \tag{3.21}$$

Substituting (3.18) in (3.19) yields

$$2^{-\frac{d}{b}} + 2^{-\frac{c}{a}} = 1 \tag{3.22}$$

$$\Longleftrightarrow \quad \sqrt[ab]{2^{-ad}} + \sqrt[ab]{2^{-bc}} = 1. \tag{3.23}$$

From Lemma 3.2 follows that the sum on the left-hand side of (3.23) cannot become 1 if at least one of the roots is irrational. Since $d/b < 1$ by (3.21), $2^{-d/b}$ must be irrational. Consequently, (3.23) has no solution for $0 < p_X(1) < 1/2$ and the Theorem is proven. □

3.4 Asymptotic optimal V2V codes

Although V2V codes of finite size are guaranteed to have nonzero redundancy (for $p_X(1) \neq 1/2$), there are several well-known approaches of designing V2V codes where the redundancy approaches zero when the size of the code is allowed to go to infinity. Kraft's inequality [13, Theorem 5.2.1] proves the existence of a prefix-free code $\mathcal{C}$ for which

$$H(S) \leq \bar{\ell}(C) < H(S) + 1 \tag{3.24}$$

holds. Well-known approaches for generating codes that satisfy (3.24) are Huffman's algorithm [21] or the Shannon-Fano code [13]. Substituting (3.7) in (3.24) yields

$$R(C) < 1. \tag{3.25}$$

I.e., when the code tree fulfills (3.24) and when the expected source word length goes to infinity, the redundancy (3.17) asymptotically approaches zero.

3.5 Negated source trees

Next, consider for an arbitrary source tree, a corresponding source tree where each symbol of each code word is negated. This corresponding source tree is denoted *negated source tree* in the following. From (3.1) follows that the probability of a source word is identical to the probability of the corresponding negated source word when $p_X(1)$ and $p_X(0)$ are swapped. Consequently, when for each element of a set of source trees, the negated source tree is also element of the set, it is sufficient to only regard the probability interval $[0, 0.5]$ for algorithms that are based on source word probabilities. This property of sets of source trees shall be denoted *negated source tree property*.

3.6 Canonical source trees

The redundancy of a V2V code (3.17) is influenced by the source tree only through the source word probabilities and the source word lengths. The source word probabilities only depend on the numbers of ones and zeros of a source word according to (3.1). I.e., the order of ones and zeros in a source word is not important. Consequently, there may exist different source trees that do not differ in their source word probabilities.

Definition 3.1. Two source trees are *equivalent* if the multiset of source word probability polynomials (3.1) of both trees is identical.

An example for equivalent source trees is given in Fig. 3.1 where the two source words 11 and 00 are identical while the remaining three source words differ only in the order of ones and zeros. In order to efficiently handle equivalent source trees, a canonical representation of source words and source trees is defined.

Definition 3.2. The canonical representation of a source word σ shall be defined as $\mathrm{Cw}(\sigma) = (M(\sigma), N(\sigma))$. I.e., it consists of the number of ones $M(\sigma)$ and the number

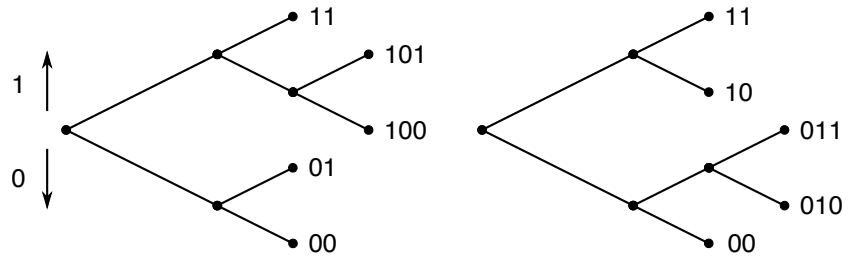

Figure 3.1: Example for two equivalent source trees.

of zeros $N(\sigma)$ of source word σ. The canonical representation of a source word shall interchangeably be referred to as *canonical source word*.

Definition 3.3. The canonical representation of a source tree $\mathcal{S}$ shall be defined as $\mathrm{Cs}(\mathcal{S}) = \{\mathrm{Cw}(\sigma) \mid \sigma \in \mathcal{S}\}_b$. I.e., it consists of the multiset[1] of canonical source words of source tree $\mathcal{S}$. The canonical representation of a source tree shall interchangeably be referred to as *canonical source tree*.

Lemma 3.3. *For $0 < p_X(1) < 1$, two source trees are equivalent according to Def. 3.1, if and only if they have the same canonical representation.*

Proof. Clearly, if two source trees have the same canonical representation, they are equivalent. To prove the other direction of the claim, it obviously suffices to show that if two source words σ_1 and σ_2 have the same source word probability polynomials, then one has $\mathrm{Cw}(\sigma_1) = \mathrm{Cw}(\sigma_2)$. The associated source word probability polynomials as defined in (3.1) are equal if and only if

$$p_X(1)^{M(\sigma_1)}(1-p_X(1))^{N(\sigma_1)} = p_X(1)^{M(\sigma_2)}(1-p_X(1))^{N(\sigma_2)} \tag{3.26}$$

holds. (3.26) can be rewritten as

$$\begin{aligned} & p_X(1)^{M(\sigma_1)}(1-p_X(1))^{N(\sigma_1)} - p_X(1)^{M(\sigma_2)}(1-p_X(1))^{N(\sigma_2)} = 0 \\ \iff\ & p_X(1)^{M(\sigma_1)}(1-p_X(1))^{N(\sigma_1)} \left(1 - p_X(1)^{M(\sigma_2)-M(\sigma_1)}(1-p_X(1))^{N(\sigma_2)-N(\sigma_1)}\right) = 0. \end{aligned} \tag{3.27}$$

(3.27) holds if any of the three product terms on the left-hand side equals zero. $p_X(1)^{M(\sigma_1)}$ equals zero if and only if $p_X(1) = 0$ and $(1-p_X(1))^{N(\sigma_1)}$ equals zero if and only if $p_X(1) = 1$. Both values are not allowed for $p_X(1)$. Consequently, (3.27)

[1] Multiset definitions are subscripted with a small b like e.g. $\{1,1,2\}_b$ in order to distinguish them from set definitions.

holds if and only if

$$\begin{aligned} 1 - p_X(1)^{M(\sigma_2)-M(\sigma_1)}(1-p_X(1))^{N(\sigma_2)-N(\sigma_1)} &= 0 \\ \iff p_X(1)^{M(\sigma_2)-M(\sigma_1)}(1-p_X(1))^{N(\sigma_2)-N(\sigma_1)} &= 1 \end{aligned} \tag{3.28}$$

holds. Since $p_X(1) < 1$ and since $1 - p_X(1) < 1$, the product on the left-hand side of (3.28) can only equal 1 if the exponents of both product terms equal zero. Consequently, $M(\sigma_2) = M(\sigma_1)$ and $N(\sigma_2) = N(\sigma_1)$ and thus $\mathrm{Cw}(\sigma_1) = \mathrm{Cw}(\sigma_2)$. □

An algorithm that is based on source word probabilities (as e.g. Huffman's algorithm) yields exactly the same results for any two equivalent source trees and thus can be directly applied to the canonical representation that both source trees share. Consequently, instead of applying an algorithm to a set of source trees, it can be applied to the corresponding set of canonical source trees, which is usually much smaller.

A similar concept is known for code trees (or in general, prefix-free codes), where only the code word lengths are relevant for the redundancy. Neither the numbers of ones and zeros, nor their order is important. Accordingly, the situation is a bit more relaxed for code trees than for source trees, where the number of ones and zeros *is* important.

The canonical Huffman codes [48, 49, 50] take advantage of this property to efficiently represent and transmit Huffman codes. However, this concept works for arbitrary prefix-free codes. Since only the code word lengths are of interest, a canonical representation of a prefix-code is given as the multiset of code word lengths. Moreover, it is well known that for any multiset of code word lengths, a prefix-free code can be constructed when the Kraft inequality [51] holds. A corresponding algorithm is given in [13, Sec. 5.2]. The canonical representation of a code tree $\mathcal{C}$ shall be given as the multiset of code word lengths

$$\mathrm{Cc}(\mathcal{C}) = \{\ell(x) \mid x \in \mathcal{C}\}_b. \tag{3.29}$$

3.7 Prefix merge algorithm

Generating a source tree from its canonical representation is a similar problem as generating a prefix-free code from code word lengths. However, it has the further constraint that the numbers of ones and zeros of the source words must be correct. While it is trivial to derive the canonical representation of a source tree, the reverse way is not very intuitive. In the following, an algorithm is presented that allows the

construction of a source tree $\mathcal{S}'$ when only the canonical representation $\mathrm{Cs}(\mathcal{S})$ of a source tree $\mathcal{S}$ is known. The algorithm ensures that $\mathrm{Cs}(\mathcal{S}) = \mathrm{Cs}(\mathcal{S}')$ holds while $\mathcal{S} = \mathcal{S}'$ needs not to hold. It is based on merging source words that only differ in the last bit (i.e., with identical prefixes) and is thus denoted *prefix merge* algorithm.

Lemma 3.4. *For a source tree $\mathcal{S}$, let*

$$\hat{\mathcal{S}} = \{\hat{\sigma} \in \mathcal{S} \mid \forall \sigma \in \mathcal{S} : \ell(\hat{\sigma}) \geq \ell(\sigma)\} \tag{3.30}$$

be the subset of source words of maximum length. Let $x \in \hat{\mathcal{S}}$ be a source word with the maximum number of ones amongst the source words of maximum length. I.e.,

$$\forall \sigma \in \hat{\mathcal{S}} : M(x) \geq M(\sigma) \tag{3.31}$$

holds for x. Then, the leaf node associated with x has a sibling that is a leaf node as well and the associated source word, which shall be denoted y, has canonical representation $\mathrm{Cw}(y) = (M(x) - 1, N(x) + 1)$.

Proof. Since a source tree is a full binary tree, each leaf node must have a sibling. Obviously, each leaf node at the highest depth must have a sibling which is a leaf node as well. Since the leaf node associated with source word x is at the highest depth, it must have a sibling that is a leaf node and that has an associated source word y of the same length as x. This proves the existence of y.

Since two source words whose associated leaf nodes are siblings differ only in the last bit, their number of ones must differ exactly by 1. Since x has the maximum number of ones amongst all source words of the same length, y must have a one less and a zero more than x. □

By only employing the canonical representation $\mathrm{Cs}(\mathcal{S})$ of a source tree $\mathcal{S}$, Lemma 3.4 identifies the canonical representations of two source words x and y for which the associated leaf nodes are siblings. These leaf nodes can be *pruned*, which means that they are removed and their common parent node becomes the new leaf node. In terms of the canonical source tree $\mathrm{Cs}(\mathcal{S})$, this pruning corresponds to replacing $\mathrm{Cw}(x)$ and $\mathrm{Cw}(y)$ with the canonical representation $\mathrm{Cw}(z)$ of their common prefix, denoted z, which is the source word, associated with the new leaf node. When one of the canonical source word $\mathrm{Cw}(x)$, $\mathrm{Cw}(y)$, or $\mathrm{Cw}(z)$ is known, the other two can be derived since x has a one more than z and y has a zero more than z. Consequently, a pruning step is unambiguously described by one of these three canonical source words.

Let $\mathrm{Cs}(\mathcal{S}_n)$ be a canonical source tree and let $\mathrm{Cw}(x_n)$, $\mathrm{Cw}(y_n)$, and $\mathrm{Cw}(z_n)$ be according to the description above, but subscripted with an index n. The canonical

source tree after pruning shall be denoted $\mathrm{Cs}(\mathcal{S}_{n+1})$. Next, two properties are derived for this relationship.

Firstly, although the relationship is established purely on canonical source trees and canonical source words, it can be used to manipulate *real*[1] source words and source trees. I.e., when a source tree is given, which has canonical representation $\mathrm{Cs}(\mathcal{S}_n)$, a pruning step can be applied, which results in a source tree with canonical representation $\mathrm{Cs}(\mathcal{S}_{n+1})$.

Secondly, a pruning step can also be applied in reverse order, which shall be denoted a *growing* step. More precisely, if a canonical source tree $\mathrm{Cs}(\mathcal{S}_{n+1})$ is given, it can be converted into $\mathrm{Cs}(\mathcal{S}_n)$ by extending the leaf node associated with the canonical source word $\mathrm{Cw}(z_n)$ with a one branch and a zero branch. I.e., canonical source word $\mathrm{Cw}(z_n)$ is duplicated and a one is appended to the first copy while a zero is appended to the second copy. This can also be used to grow a real source trees.

The prefix merge algorithm uses both properties. In a first step, for the given canonical source tree, a sequence of pruning steps is derived until the canonical source tree is fully pruned, which means that only two canonical source words of length one are left. These are the *merge by prefix* steps where the name of the algorithm comes from. In the second step, the sequence of pruning steps is used in reverse order to grow a real source tree that has the desired canonical representation. Equivalently, the tree can be created during the pruning procedure by converting the individual pruning steps into subtrees and merging these.

An algorithmic description is as follows:

1. Initialize an empty list and add all canonical source words to it.
2. Associate with each canonical source word in the list, a tree only consisting of a root node.
3. Out of all canonical source words of maximum length in the list, find a source word x which has the largest number of ones. Find for this source word a possible sibling y. I.e., with a ‘one’ less and with a ‘zero’ more. This corresponds to Lemma 3.4.
4. Remove x and y from the list and create a new canonical source word z consisting of the common prefix of x and y. I.e., with as many ‘ones’ as y and as many ‘zeros’ as x.

[1] ‘Real’ meaning that all source words are exactly known.

5. Associate with z a tree consisting of a root node, a '1' branch and a '0' branch.
6. Append the tree of x to the '1' branch and the tree of y to the '0' branch.
7. If the list is empty, terminate. The tree associated with z is the desired source tree.
8. Add z to the list and continue with step 3.

Let $\mathcal{S}^c$ be a canonical source tree and let function $\mathrm{Rs}(\cdot)$ yield a source tree $\mathrm{Rs}(\mathcal{S}^c)$, that is generated by applying the prefix merge algorithm to S^c. Although the algorithm allows a degree of freedom in creating siblings in step 3, which may lead to different source trees, it shall be assumed that function $\mathrm{Rs}(\cdot)$ always yields the same $\mathrm{Rs}(\mathcal{S}^c)$ for a given canonical source tree $\mathcal{S}^c$. In other words, each canonical source tree $\mathcal{S}^c$ is unambiguously associated with one particular source tree $\mathrm{Rs}(\mathcal{S}^c)$.

Furthermore, let $\mathcal{X}$ be a set of source trees and let a function $\mathrm{Eq}(\cdot)$ be defined as

$$\mathrm{Eq}(\mathcal{X}) = \{\mathrm{Rs}(\mathrm{Cs}(\mathcal{S})) \mid \mathcal{S} \in \mathcal{X}\}. \tag{3.32}$$

I.e., it yields another set of source trees, derived from the canonical representations of the set of source trees $\mathcal{X}$. The set $\mathrm{Eq}(\mathcal{X})$ shall be denoted *canonical representation* of a set of source trees $\mathcal{X}$. An algorithm that can be carried out with canonical source trees yields identical results for $\mathcal{X}$ and $\mathrm{Eq}(\mathcal{X})$ while $\mathrm{Eq}(\mathcal{X})$ may contain fewer elements than $\mathcal{X}$. This property is exploited in the V2V code design strategies of Ch. 5.

3.8 Canonical V2V codes

As discussed in the previous section, for algorithms that are based on source word probabilities, source word lengths, and code word lengths of V2V codes, canonical source trees and canonical code trees can be used. Consequently, a canonical representation shall be defined for V2V codes. It is denoted $\mathrm{Cv}(v)$ for a given V2V code $v = (\mathcal{S}, \pi, \mathcal{C})$, and simply consists of a canonical source tree, a permutation function, and a canonical code tree. More precisely,

$$\mathrm{Cv}(v) = (\mathrm{Cs}(\mathcal{S}), \pi, \mathrm{Cc}(\mathcal{C})). \tag{3.33}$$

Tab. 3.1 shows the canonical representation of an exemplary V2V code as depicted in Fig. 3.2. A column of the table contains numbers of ones and zeros of a source word σ_i along with the length of the corresponding code word c_j. I.e., the bijective

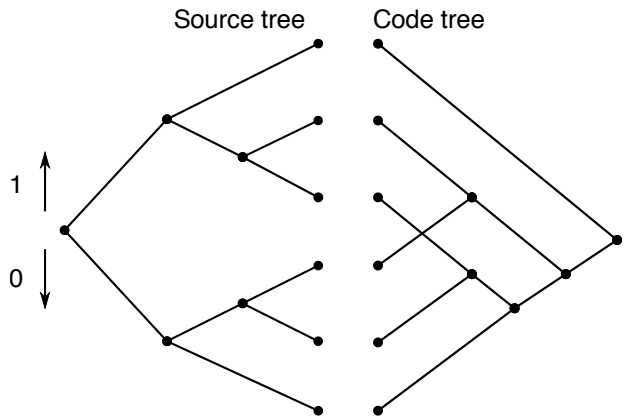

Figure 3.2: Examplary V2V code.

$M(\sigma_i)$	2	2	2	1	1	0
$N(\sigma_i)$	0	1	1	2	2	2
$\ell(c_j) = \ell(k(\sigma_i))$	1	3	3	4	4	3

Table 3.1: Tabular representation of a canonical V2V code for source words $\mathcal{S} = \{\sigma_1, \sigma_2, \ldots\}$ and associated code words $\mathcal{C} = \{c_1, c_2, \ldots\}$.

mapping of source word indexes i to code word indexes j results implicitly from the table structure. Note that there are several columns with identical values which could be used to further compress the table format. However, for better readability, the format as exemplified in Tab. 3.1 is used for presenting V2V codes in the following.

3.9 Prime and composite V2V codes

In this section, the concept of *prime* and *composite* V2V codes is introduced.

Definition 3.4. A V2V code shall be *composite* if two prefixes a and b of nonzero length exist for which all of the following conditions are fulfilled:

- Two or more source words have prefix a
- Two or more code words have prefix b
- All source words with prefix a are associated with code words with prefix b
- All code words with prefix b are associated with source words with prefix a

A V2V code shall be *prime* if it is not composite.

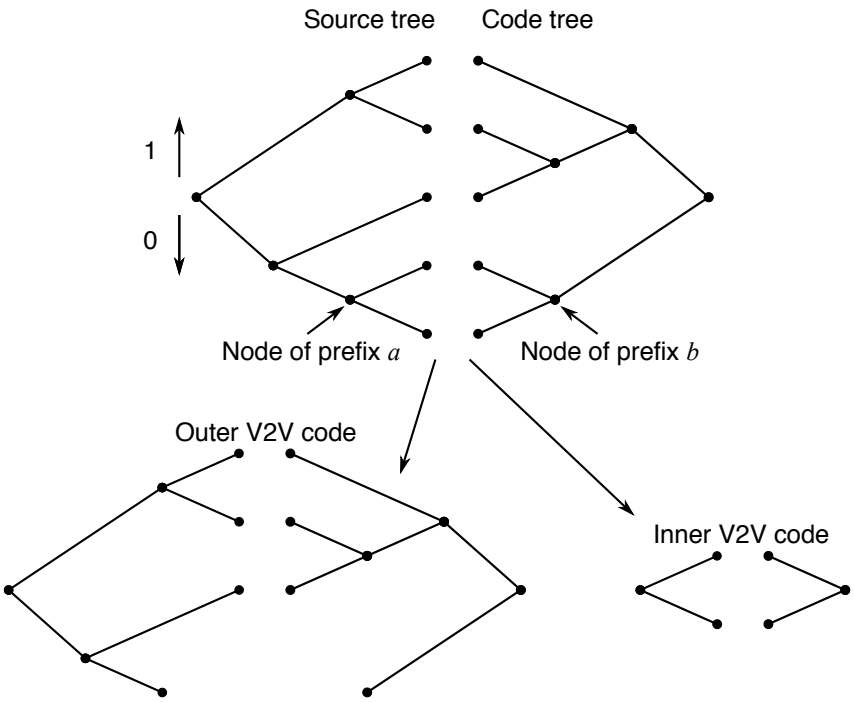

Figure 3.3: Example for a composite V2V code.

A composite V2V code v according to Def. 3.4 can be split into two V2V codes. An example is depicted in Fig. 3.3. Prefix a corresponds to the internal node where the source tree is split and prefix b corresponds to the internal node where the code tree is split. The two separated subtrees constitute the so-called *inner* V2V code. In the remaining source and code tree, the two internal nodes where splitting was carried out are now leaf nodes. These are associated with each other and the resulting code shall be denoted *outer* V2V code. In the example of Fig. 3.3 the outer and inner V2V codes are prime since they cannot be split further. In principle, it is possible that inner and outer V2V codes are composite and further splitting can be applied repeatedly until all resulting outer and inner V2V codes are prime. This is similar to the prime factors of a natural number. However, a V2V code is not uniquely described by its prime V2V codes since it needs to be specified how the codes are connected.

Encoding with a composite V2V code is identical to encoding with the outer V2V code and switching to the inner V2V code after a source word a is encoded. After encoding a source word with the inner V2V code, it is switched back to the outer V2V code. In terms of the average code length, this corresponds to elongating source word a by the average source word length of the inner V2V code and code word b is

elongated by the average code word length of the inner V2V code.

Consider the multiset of prime V2V codes, that results from repeatedly splitting a V2V code. The question arises, whether this multiset is unique for the V2V code. It turns out, that this is the case, because all potential splitting points can be determined, based on the original (unsplit) V2V code. Splitting at these points always results in the same subtrees, regardless of the order in which the points are processed. Therefore, always the same multiset of prime V2V codes is yielded.

3.9.1 Suboptimality of composite V2V codes

Let a composite V2V code v with source tree $\mathcal{S}$ and code tree $\mathcal{C}$ be split into inner and outer V2V codes according to Def. 3.4 (with prefixes a and b). $\mathcal{S}_I$ and $\mathcal{C}_I$ shall be the source and code trees of the inner V2V code, and $\mathcal{S}_O$ and $\mathcal{C}_O$ shall be the source and code trees of the outer V2V code. Let $x^\frown y$ denote the concatenation of two binary words x and y. Then, $\mathcal{S}$ can be written as

$$\mathcal{S} = (\mathcal{S}_O \setminus \{a\}) \cup \{a^\frown\sigma \mid \sigma \in \mathcal{S}_I\} \tag{3.34}$$

and $\mathcal{C}$ can be written as

$$\mathcal{C} = (\mathcal{C}_O \setminus \{b\}) \cup \{b^\frown c \mid c \in \mathcal{C}_I\}. \tag{3.35}$$

Let S, S_I, C_I, S_O, and C_O be the random variables associated with $\mathcal{S}$, $\mathcal{S}_I$, $\mathcal{C}_I$, $\mathcal{S}_O$, and $\mathcal{C}_O$, respectively, for encoding a random variable X. The average code length (3.6) of v can be expressed as

$$\begin{aligned}
\bar{\ell}_v(X) &= \frac{\bar{\ell}(C)}{\bar{\ell}(S)} \\
&= \frac{\sum\limits_{\sigma\in\mathcal{S}_O\setminus\{a\}} p_S(\sigma)\ell(k(\sigma)) + \sum\limits_{\sigma\in\mathcal{S}_I} p_S(a^\frown\sigma)\ell(k(a^\frown\sigma))}{\sum\limits_{\sigma\in\mathcal{S}_O\setminus\{a\}} p_S(\sigma)\ell(\sigma) + \sum\limits_{\sigma\in\mathcal{S}_I} p_S(a^\frown\sigma)\ell(a^\frown\sigma)} \\
&= \frac{\bar{\ell}(C_O) - p_{S_O}(a)\ell(b) + \sum\limits_{\sigma\in\mathcal{S}_I} p_{S_O}(a)p_{S_I}(\sigma)\big(\ell(b)+\ell(k(\sigma))\big)}{\bar{\ell}(S_O) - p_{S_O}(a)\ell(a) + \sum\limits_{\sigma\in\mathcal{S}_I} p_{S_O}(a)p_{S_I}(\sigma)\big(\ell(a)+\ell(\sigma)\big)} \\
&= \frac{\bar{\ell}(C_O) - p_{S_O}(a)\ell(b) + p_{S_O}(a)\ell(b)\sum\limits_{\sigma\in\mathcal{S}_I} p_{S_I}(\sigma) + p_{S_O}(a)\sum\limits_{\sigma\in\mathcal{S}_I} p_{S_I}(\sigma)\ell(k(\sigma))}{\bar{\ell}(S_O) - p_{S_O}(a)\ell(a) + p_{S_O}(a)\ell(a)\sum\limits_{\sigma\in\mathcal{S}_I} p_{S_I}(\sigma) + p_{S_O}(a)\sum\limits_{\sigma\in\mathcal{S}_I} p_{S_I}(\sigma)\ell(\sigma)}.
\end{aligned} \tag{3.36}$$

With

$$\sum_{\sigma \in \mathcal{S}_I} p_{S_I}(\sigma) = 1, \tag{3.37}$$

(3.36) becomes

$$\bar{\ell}_v(X) = \frac{\bar{\ell}(C_O) + p_{S_O}(a)\bar{\ell}(C_I)}{\bar{\ell}(S_O) + p_{S_O}(a)\bar{\ell}(S_I)}. \tag{3.38}$$

An intuitive interpretation can be given for (3.38) as follows. $p_{S_O}(a)\bar{\ell}(S_I)$ corresponds to the elongation of source word a and $p_{S_O}(a)\bar{\ell}(C_I)$ corresponds to the elongation of code word b.

Lemma 3.5. *The average code length of a composite V2V code is in between the average code lengths of the outer and the inner V2V code it can be split into.*

Proof. Let a, b, c, d be real and positive. The mediant inequality states that if $a/b \leq c/d$, then

$$\frac{a}{b} \leq \frac{a+c}{b+d} \leq \frac{c}{d} \tag{3.39}$$

where $(a+c)/(b+d)$ is denoted mediant of a/b and c/d. A proof can be found in the related literature (see [52] for an overview) and is shortly reviewed in Appendix B.1. Since $\bar{\ell}(C_O)$, $\bar{\ell}(S_O)$, $\bar{\ell}(C_I)$, $\bar{\ell}(S_I)$, and $p_{S_O}(a)$ are real and positive, $\bar{\ell}_v(X)$ is the mediant of the average code lengths of the inner and outer V2V code. Consequently,

$$\frac{\bar{\ell}(C_O)}{\bar{\ell}(S_O)} \leq \bar{\ell}_v(X) \leq \frac{\bar{\ell}(C_I)}{\bar{\ell}(S_I)} \tag{3.40}$$

holds if

$$\frac{\bar{\ell}(C_O)}{\bar{\ell}(S_O)} \leq \frac{\bar{\ell}(C_I)}{\bar{\ell}(S_I)} \tag{3.41}$$

and

$$\frac{\bar{\ell}(C_O)}{\bar{\ell}(S_O)} \geq \bar{\ell}_v(X) \geq \frac{\bar{\ell}(C_I)}{\bar{\ell}(S_I)} \tag{3.42}$$

holds if

$$\frac{\bar{\ell}(C_O)}{\bar{\ell}(S_O)} \geq \frac{\bar{\ell}(C_I)}{\bar{\ell}(S_I)}, \tag{3.43}$$

which completes the proof. □

Lemma 3.5 states that each composite V2V code consists of at least one prime V2V code that has a lower or equal average code length as the composite V2V code. This property may be useful when searching a candidate set of V2V codes for minimum redundancy codes. Whenever a composite V2V code occurs and when it is

guaranteed that the prime V2V codes it consists of are in the candidate set, the composite V2V code can be disregarded as optimal V2V code. Because one of its prime codes has either lower or the same average code length as the composite V2V code.

As discussed in the following chapters, a search for optimal V2V codes is often based on a set of candidate source trees and optimal code trees are created with the Huffman [21] or package merge algorithm [53]. In such a case, when a composite V2V code occurs, it is sufficient if the source trees of its prime V2V codes are in the candidate set. This is because V2V codes with the same source trees as the prime V2V codes have lower or the same redundancy as the prime V2V codes. Consequently, composite V2V codes can be disregarded as well.

3.9.2 Primality test for V2V codes

In order to benefit from the fact that composite V2V codes can be disregarded in many V2V code design schemes, primality of the related codes must be tested. This can be done by iterating over all possible subtrees of the source tree and searching for a corresponding subtree in the code tree such that both subtrees form an inner V2V code according to Def. 3.4.

3.9.3 Primality of canonical V2V codes

The concept of prime and composite V2V codes can also be applied to canonical V2V codes. When for a canonical V2V code, one particular V2V code can be found that is composite, the canonical V2V code is considered composite and can be disregarded in many algorithms according to the argumentation above. However, it turns out that deciding whether a composite V2V code exists for a given canonical representation is a difficult task.

For the most applications it is not necessary to determine for each canonical V2V code whether it is prime or composite. Instead, it may be sufficient when some of the composite V2V codes can be detected and sorted out. If there would be a simple test, which is able to detect particular composite V2V codes only, this can already reduce the overall complexity. In the following, such a test is developed. It is based on the fact that source and code trees can be grown from the leaf nodes, starting with the longest source and code words. For the source tree, the prefix merge algorithm can be applied as described in Sec. 3.7. For the code tree, an even simpler method can be employed, which is derived from the prefix merge algorithm.

i	1	2	3	4	5
$M(\sigma_i)$	2	1	1	1	0
$N(\sigma_i)$	0	1	1	2	3
$\ell(c_j) = \ell(k(\sigma_i))$	2	3	3	2	2

Table 3.2: Exemplary canonical V2V code for primality testing.

An algorithmic description is as follows:

1. Initialize an empty list and add all canonical code words to it.
2. Associate with each canonical code word in the list, a tree only consisting of a root node.
3. Out of all canonical code word of maximum length in the list, select two arbitrary code words x and y.
4. Remove x and y from the list and create a new canonical code word z which is by 1 shorter than x (or y).
5. Associate with z a tree consisting of a root node and two branches.
6. Append the tree of x to the one branch and the tree of y to the other branch.
7. If the list is empty, terminate. The tree associated with z is the desired code tree.
8. Add z to the list and continue with step 3.

The above algorithm has the freedom to select two of potentially many code words in step 3. The same is true for the prefix merge algorithm, but with some more restrictions regarding the numbers of ones and zeros (see step 3 of the prefix merge algorithm in Sec. 3.7). Having in mind which leaf nodes of the source tree are associated with leaf nodes of the code tree, this freedom can be used to somehow skilfully select source and code words for merging so that inner V2V codes are formed. If this succeeds, the corresponding canonical V2V code is composite. A full primality test may be very complex using this concept since the degree of freedom in step 3 of both algorithms may lead to a large number of possible different V2V codes.

An example for the concept is given in the following for an exemplary canonical V2V code as depicted in Tab. 3.2. It corresponds to the V2V code depicted in Fig. 3.3 which it is known to be composite. From the code word lengths, it is clear that $k(\sigma_2)$

and $k(\sigma_3)$ are merged first since they are the two longest code words. According to the prefix merge algorithm, σ_4 and σ_5 are merged first in the source tree. So far, no other choices for merging were possible. When continuing the procedure, all of the remaining code words are of length 2 and an arbitrary pair could be merged. Since σ_4 and σ_5 are already merged in the source tree, merging $k(\sigma_4)$ and $k(\sigma_5)$ creates an inner V2V code. This proves that the canonical V2V code is composite.

The strategy of selecting source and code words for merging in order to find inner V2V codes is not further specified so far. It may be possible to describe an algorithm that is able to securely find inner V2V codes if they exist. However, when only using a primality test to reduce the computational complexity of an algorithm for e.g. V2V code design, a complete and costly primaliy test can usually be relinquished.

3.10 Chapter summary

In this chapter, the properties of variable-to-variable length codes are studied for binary i.i.d. sources. It is shown that the redundancy of a V2V code cannot be zero for a binary i.i.d. source except for the case where the probability of one of the source is 0.5. Furthermore, by analyzing the properties of the average code length of V2V codes, a canonical representation for source trees, code trees, and finally for V2V codes is defined. The prefix merge algorithm was developed in order to be able to construct a source tree from a canonical representation. It is based on merging particular source words by their prefix.

One important observation of this chapter is that V2V codes can be composed of other V2V codes. This lead to the concept of composite and prime V2V codes. Moreover, it was possible to show that if a V2V code is composite, its average code length lies between the average code lengths of the V2V codes it is composed of. This property can be useful for finding minimum redundancy V2V codes. For example, composite V2V codes can be removed from a candidate set when the optimal codes are known to be prime, which is often the case.

Several aspects of primality testing for V2V codes were discussed. It turned out that a simple primality test could not be found. In particular, primality testing of canonical V2V code seems to be a difficult task. However, it was possible to derive a simple procedure for primality testing from the prefix merge algorithm. It can detect for some canonical V2V codes that there exists a corresponding V2V code that is surely composite. Canonical V2V codes and the concept of prime and composite V2V codes are the basis for the V2V code design algorithms of Ch. 4 and Ch. 5.

4

Joint V2V code and probabiltiy interval design

The main focus of the current chapter lies on the design of code trees for a given source tree $\mathcal{S}$. A binary i.i.d. random process $\{X\}$ is assumed, which shall be encoded with the source tree. This constitutes an i.i.d. random process $\{S\}$ over $\mathcal{S}$. In general, all the well-known techniques for encoding non-binary i.i.d. sources with variable-length codes can be used, like Huffman coding [21] or the package merge algorithm [53]. Most such algorithms construct a prefix-free code based on a sequence of decisions by comparing sums of leaf node probabilities $p_S(\sigma_i)$. The intuitive approach is to select a predefined pmf p_X for the binary i.i.d. source $\{X\}$, which determines the source word pmf p_S and e.g. a Huffman code can be generated. The solution found in this way is optimal for the predefined pmf p_X. For a different pmf of the binary source $\{X\}$, the procedure needs to be repeated, potentially yielding a different V2V code. In order to yield a set of V2V codes, many different binary pmfs must be evaluated while it remains unclear, whether the found V2V codes are optimal for further, untested pmfs. To overcome this problem, a technique is developed, which use $p_X(1)$ as variable. Consequently, the source word probabilities are polynomials in $p_X(1)$. As stated above, the Huffman and package merge algorithm are based on the comparison of sums of source word probabilities. It turns out that all these operations can also be done with polynomials where decisions automatically yield probability intervals for which the decisions are valid. When e.g. used with the Huffman algorithm, the result is a set of subintervals and associated Huffman codes that are optimal for the subintervals. This also yields the exact number of optimal code trees that exist for a given source tree. The same idea can be applied to Tunstall codes, where the code

tree is given in advance and a source tree is derived.

For Huffman coding, a related concept is introduced by Longo and Galasso [54] where so-called *attraction regions* are defined. Attraction regions are source symbol probabilities that share the same Huffman code. Longo and Galasso prove that attraction regions are always convex by employing the *sibling property* [55] of Huffman codes. Analogously, in [56] so-called *Tunstall regions* are defined and it is shown that a Tunstall code is always optimal for one single uninterrupted probability interval.

4.1 Basic principle of polynomial-based algorithms

Before discussing polynomial-based versions of the various algorithms for code generation, the basic principle of making an algorithm polynomial-based is described. For this purpose, it is sufficient to analyze which operations are applied to symbol probabilities and whether these operations can be carried out when the symbol probabilities are polynomials. Fortunately, most of the algorithms for code generation apply very few and simple operations on symbol probabilities.

Summation and multiplication

Obviously, for polynomials, summation and multiplication can be carried out yielding as a result a new polynomial.

Comparison

The Huffman algorithm is based on repeatedly finding the two smallest probabilities of a set of probabilities. For the package merge algorithm, a set of probabilities must be sorted as described in Sec. 4.5. For the Tunstall algorithm, the highest probability out of a set of probabilities needs to be derived. All these operations can be reduced to one or more 'larger than' or 'smaller than' comparison operations where for two probabilities, it is decided which of both is larger or smaller. When these probabilities are given as polynomials P_a and P_b in p, a decision, which of both is larger is only valid for particular values of p (except if P_a a nd P_b are identical polynomials). More precisely, whenever polynomials P_a and P_b intersect, the decision changes. The other way around, in between two intersections, the decision is the same. Consequently, an interval can be specified for which a decision is valid. Boundaries of such intervals can only reside at values of p where the polynomials intersect. I.e., at the roots of

$P_a - P_b = 0$.

In the following, a procedure is described how an algorithm is made polynomial-based when it applies no other operations than summation, multiplication, and 'larger than' or 'smaller than' comparisons to symbol probabilities.

An initial probability interval needs to be defined for which results of the algorithm are of interest. E.g., $I =]0, 0.5[$. The algorithm is assumed to be prepared in a way such that sorting operations are translated into sequences of comparisons. Whenever, in the algorithm, a comparison is required, the following steps are applied.

1. For the requested comparison, the two involved polynomials shall be P_a and P_b and the comparison shall be $P_a < P_b$.
2. Find all real roots of $P_a - P_b = 0$ which are in the interval I and subdivide I at the real roots.
3. For each subinterval derive for an arbitrary value in the subinterval, whether P_a is smaller than P_b.
4. Continue the algorithm for each of the subintervals separately and independently with the subinterval being used as interval I.

The above procedure is carried out whenever a comparison is required and each time, the probability interval may be subdivided into further subintervals and the algorithm is continued for each of the subintervals with the associated decision results.

4.1.1 Finite-state machine-based algorithms

The procedure described in the previous section, although simple to explain, may be complicated to implement in software because of step 4 where the state of the whole program must be duplicated a number of times and for each duplicate, the program has to continue independently. A way to address this problem is to implement the algorithm as a finite-state machine. This corresponds to creating a number of *state transitions*, each consisting of a portion of the algorithm so that decisions are carried out between state transitions. Furthermore, a *state variable* is maintained, which contains the whole state of the algorithm, i.e., all variables required to carry out the algorithm, since state transitions cannot store variables. A so-called *state processor* maintains a queue of state transitions that need to be processed and for each state transition, a copy of the state variable. To start the algorithm, the state transition that corresponds to the

beginning of the algorithm is queued and a state variable is initialized and associated with this state transition. The operation of the state processor is as follows. From the queue, it removes one state transition and calls it with the associated state variable as parameter. When this state transition returns, it either signals that the algorithm is finished, or it specifies a state transition to be processed next. In the latter case, the state transition may have requested a decision. If so, the state processor carries out the decision and passes the decision result to the next state transition to be processed and queues the next state transition.

This is the point where an algorithm can be made polynomial-based. The result of the polynomial-based decision is a number of subintervals, each one being associated with a decision result. The state processor creates duplicates of the next state transition and of the state variable so that one copy of each is available for each subinterval. The decision results and subintervals are stored in the state variable so that the next state transition can evaluate the decision result. All state transitions created in this way are queued along with the associated state variables, which effectively corresponds to the forking of the program flow as it is required for polynomial-based algorithms. The state processor sequentially continues the processing of the queued items, which may lead to further duplications. The results of the algorithm for the different subintervals are returned by the terminating states.

An advantage of this concept is that forking of the program flow doesn't need to be kept in mind when implementing an algorithm since the algorithm always receives only one result for a decision. On the other hand, the state processor doesn't need to know anything about the algorithm, since it only needs to process queued state transitions, evaluate decisions, and create duplicates of the transition states if necessary. The difficulty in this concept lies in the conversion of an algorithm to a number of state transitions so that all decisions happen between state transitions. Apart from this, nothing more needs to be done to make any algorithm polynoial-based.

For rather simple algorithms, polynomial-based versions are often easy to achieve without the finite-state machine concept. However, for more complicated algorithms, like a polynomial-based implementation of the extremely complex *lazy reverse run-length package merge* algorithm [57, p. 201] as derived in Sec. 4.5, the concept is very useful.

4.1.2 Grouping of identical polynomials

Sometimes it is helpful to know whether two polynomials are identical. E.g. for a polynomial-based sorting algorithm, the number of elements involved in the actual sorting can be reduced when identical polynomials are treated as one polynomial because all of them will anyway appear in an uninterrupted sequence in the sorted result. It is easy to decide whether two polynomials are identical by comparing the polynomials coefficients. More generally, a *sort key* can be defined for polynomials as e.g. the list of the polynomials coefficients or a hash of it. It allows all the well-known techniques for determining identical items as e.g. by sorting or by employing a hash table. This concept is used in the *runlength* extension of the package merge algorithm in Sec. 4.5.

4.1.3 Precise polynomial root handling

Real roots of a polynomial may be irrational numbers which don't have a closed-form expression. Using a rational approximation of such a root is usually sufficient for practical applications but the results are not exact. However, it is possible to circumvent this problem. Instead of storing polynomial roots as as rational approximations, they can be stored as polynomial along with a so-called isolation interval with rational boundaries (see [58, 59]) such that the interval only contains the one real root of the associated polynomial. Having polynomial roots stored in this form, it is always possible to correctly judge whether two roots are identical or which of them is lower or higher. More precisely, assume two polynomial roots stored in this way, i.e., given as polynomial P_a with isolation interval I_a and polynomial P_b with isolation interval I_b. When the isolation intervals do not overlap, it is clear which of the roots must be smaller. If they overlap, the roots of $P_a P_b$ in the intersection of both intervals are derived. If there is only one root with multiplicity equal to the sum of the multiplicities of both roots, the roots are identical. If not, isolation interval refinement is carried out on I_a and I_b until the intervals are disjoint.

4.1.4 Implementation

A handy way of operating with polynomials in a software implementation is to use a *symbolic computation* software platform. The algorithms described in the following are implemented in the programming language *Python* [60] by using the symbolic mathematics framework *SymPy* [61]. It provides functionality for working with polynomials

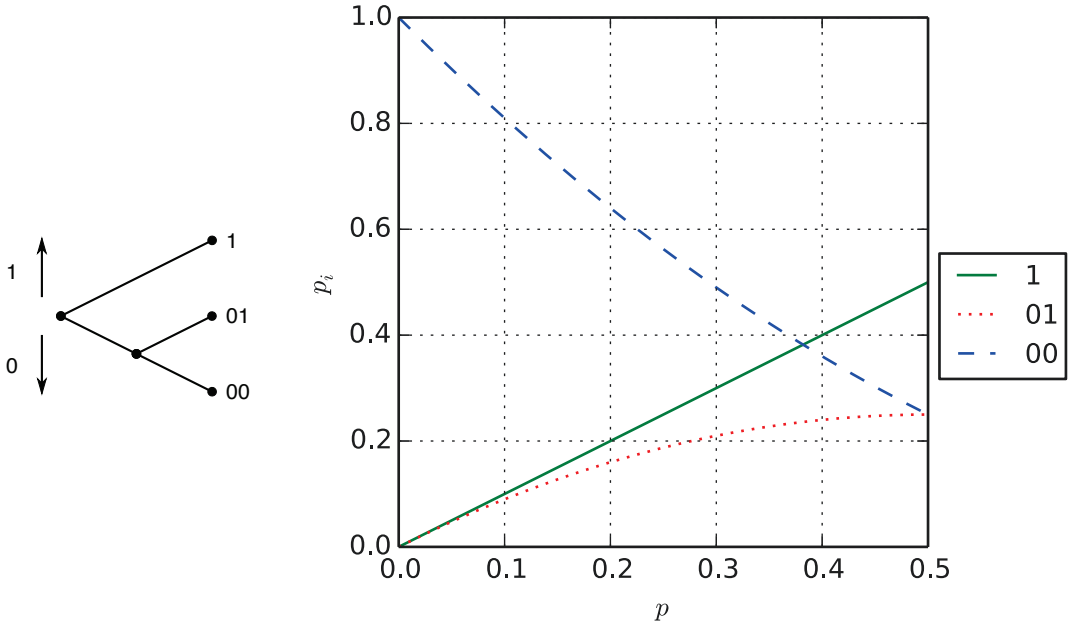

Figure 4.1: Left: Source tree, Right: Probability polynomials of source words

and also implementations of polynomial root isolation algorithms according to [58, 59].

4.2 Polynomial-based Huffman algorithm

Huffman's algorithm constructs a prefix-free code based on the order of symbol probabilities and of sums of symbol probabilities. The subsequent discussion assumes a non-binary i.i.d. source $\{S\}$ as created by a source tree $\mathcal{S}$ when parsing a binary i.i.d. source $\{X\}$ into source words. The symbol probabilities are then source word probabilities which are polynomials in $p_X(1)$. For notational convenience, $p_X(1)$ is abbreviated with p and $p_S(\sigma_i)$ is abbreviated with p_i. As stated in the previous section, deriving an order of a set of polynomials by value is only possible in intervals of p where the order of the values of the polynomials doesn't change. I.e., whenever the graphs of any two polynomials in the set intersect, these two polynomials switch their order. An example for this is given in Fig. 4.1. Here, the graphs of source word polynomials for the depicted source tree are given. In this case, source word probabilities of source words "1" and "00" intersect inside of the interval $[0, 0.5]$. The value p of the intersection is given as root of the difference of both polynomials p and $(1-p)^2$ which is located at $p = (3-\sqrt{5})/2 \approx 0.382$. Furthermore, the polynomials of code words "00" and "01" intersect at $p = 1/2$. This is, in general, also a relevant root. However, the

current example shall be restricted to the interval $[0, 0.5]$ as it would be appropriate for the use in the PIPE coding system. The intersection at $p = 1/2$ is thus located at the interval boundary and can be neglected. Two subintervals are spanned in this way. $I_1 = [0, 0.382[$ and $I_2 = [0.382, 0.5]$. For any value of p in such an interval, the order of source word probabilities is the same and the first decision in Huffman's algorithm, which of the symbols to merge is also the same. In the example of Fig. 4.1, Huffman's algorithm merges source words "00" and "1" for values of $p \in I_1$ and the source words "00" and "01" for values of $p \in I_2$.

After merging two source words, their polynomials graphs are replaced with their sum and the procedure is repeated for each subinterval which may involve further subdivisions until for every interval, a Huffman code is completed.

The detailed description of the algorithm is as follows:

1. Start with a set of symbols for which the Huffman codes and intervals shall be derived with each symbol having an associated probability polynomial. Create an empty list that represent the progress of Huffman's algorithm, able to store binary trees.

2. Initialize the list of trees with one tree for each symbol, only consisting of a single node which represents the symbol. Associate an initial interval, e.g., $[0, 0.5]$ with the list of trees.

3. Derive the probability polynomials p_i for each of the trees in the list of trees by summing over the trees leaf node probability polynomials.

4. Derive all values of p where any two of the trees polynomials have the same value and which are in the interval I. These are the roots of all differences of the trees polynomials $p_i - p_k \, \forall \, i \neq k$ in I.

5. Subdivide the interval I into non-overlapping subintervals according to the roots found in step 4 neglecting intervals of zero width. Associate with each subinterval, a copy of the list of trees which belongs to the original interval I.

6. For each subinterval, find the two trees with the two lowest polynomial values p_i and p_k for an arbitrary value of p in the subinterval. In the list associated with the subinterval, merge the two trees by making their root nodes children of a newly created root node.

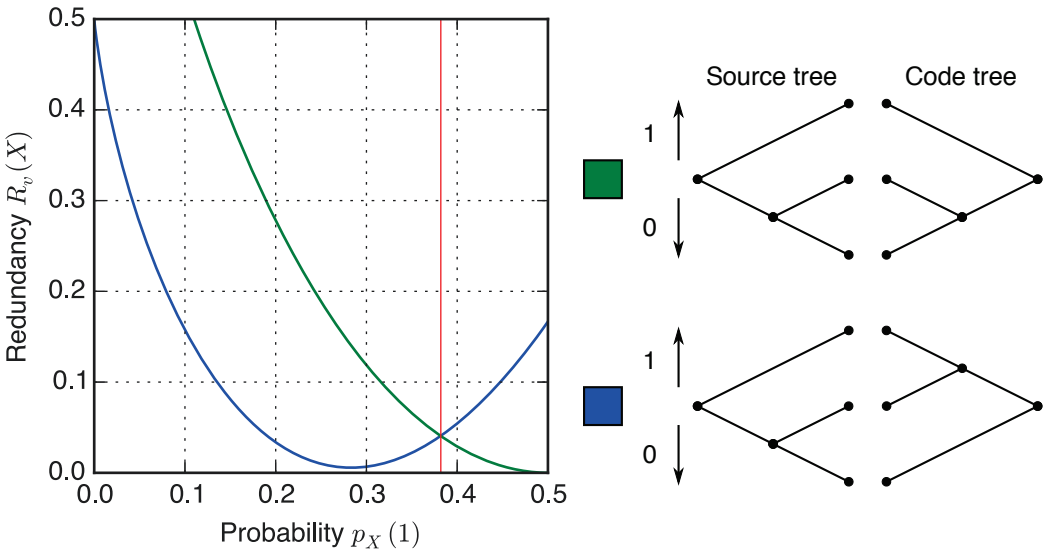

Figure 4.2: The two minimum redundancy V2V codes for the given source tree and their redundancy. The red line shows the position $p_X(1) = (3-\sqrt{5})/2 \approx 0.382$ where the probability interval is split into two subintervals.

7. For each subinterval separately, continue at step 3 until the Huffman tree is complete. I.e., the list of trees contains only one tree.

The above algorithm yields a set of disjoint intervals, each being associated with a Huffman code. It can, however, happen that the Huffman trees of two neighboring subintervals differ from each other, but assign the same code word lengths to the source symbols. I.e., the average code word length polynomials (3.5) are equal for both Huffman codes. In this case, the intervals are merged, keeping an arbitrary one of the two Huffman codes. This ensures that the algorithm yields the exact number of Huffman codes with distinct average code word length that exist for a source with polynomial-based symbol probabilities. In the case of V2V codes, this yields a uinque optimal set of code trees for a given source tree. Fig. 4.2 shows the two V2V codes that result from running the polynomial-based Huffman algorithm for the source tree of Fig. 4.1 as well as their redundancy according to (3.17). The two probability intervals are $[0, (3-\sqrt{5})/2[$ and $[(3-\sqrt{5})/2, 0.5]$.

4.3 Merging V2V codes with probability intervals

The polynomial-based Huffman algorithm as presented in the previous section derives a set of intervals with associated code trees for one predefined source tree. For finding the optimal V2V codes for a set of source trees, the outcome of the polynomial-based Huffman algorithm for all source trees needs to be merged. In other words, select from all V2V codes yielded from applying the polynomial-based Huffman algorithm to all source trees, the codes that have minimum average code length for some probability interval and specify these intervals. The selected V2V codes and their intervals correspond to the envelope of the redundancy (3.15) of all V2V codes of the set and deriving it basically corresponds to intersecting the average code length of all V2V codes, deriving intersection points and selecting minimum redundancy codes along with intervals.

An algorithmic description of such a merging procedure is given as follows:

1. Create a set, denoted *candidate set*, containing all intervals with associated V2V codes to be merged together.

2. Create an empty set, denoted *currently best set*, for storing intervals with associated V2V codes. Remove a number of non-overlapping and contiguous intervals with associated V2V codes[1] from the candidate set and add them to the currently best set.

3. Remove an arbitrary interval and associated V2V code from the candidate set. The interval shall be denoted *current interval*.

4. Subdivide each interval in the currently best set that partly overlaps with the current interval, so that the resulting subintervals either don't overlap with the current interval, or they are fully covered by it. When an interval is subdivided, its associated V2V code is associated with each of the subintervals.

5. Subdivide the current interval so that the resulting subintervals are congruent with these intervals in the currently best set that fall inside the current interval. The subintervals created in this way are denoted *current subintervals*.

[1] This can e.g. be the set of intervals and associated V2V codes yielded by one execution of the polynomial-based Huffman algorithm.

6. For each pair of congruent intervals (one interval from the currently best set and the congruent interval from the current subintervals), find these values of p in the interval for which the average code length (3.6) of both associated V2V code equals. If such values of p exist, both congruent intervals are subdivided at these values.

7. For each pair of congruent intervals, calculate the average code length for an arbitrary value of p in the interval for both associated V2V codes. If the calculated average code length of the V2V code associated with the current subinterval is the smaller one of both values, flip the association of the two V2V codes (so that the V2V code with lower average code length is located in the currently best set).

8. Merge all neighboring subintervals whose associated V2V codes have the same canonical representation keeping an arbitrary one of the associated V2V codes. Discard the current subintervals.

9. Continue with step 3 until the candidate set is empty. The currently best set contains the merged V2V codes.

Remark: Finding values of p for which the average code length of two V2V codes v_a and v_b is identical (as required in step 6) corresponds to finding solutions for

$$\begin{aligned} \bar{\ell}_{v_a}(X) - \bar{\ell}_{v_b}(X) &= 0 \\ \iff \frac{\bar{\ell}(C_a)}{\bar{\ell}(S_a)} - \frac{\bar{\ell}(C_b)}{\bar{\ell}(S_b)} &= 0 \\ \iff \frac{\bar{\ell}(C_a)\bar{\ell}(S_b) - \bar{\ell}(C_b)\bar{\ell}(S_a)}{\bar{\ell}(S_a)\bar{\ell}(S_b)} &= 0 \\ \iff \bar{\ell}(C_a)\bar{\ell}(S_b) - \bar{\ell}(C_b)\bar{\ell}(S_a) &= 0 \end{aligned} \tag{4.1}$$

where S_x and C_x are random variables associated with source and code tree of V2V code v_x. As can be seen from (4.1), the problem is a polynomial root finding problem, which can be treated as discussed in Sec. 4.1.

The above algorithm derives a number disjoint subintervals and selects for each interval the V2V code of minimum redundancy out of all candidate V2V codes.

4.4 Polynomial-based Tunstall algorithm

The idea of using polynomials in p instead of constant values can also be applied to Tunstall's algorithm [23]. Tunstall codes are so-called variable-to-fixed-length (V2F)

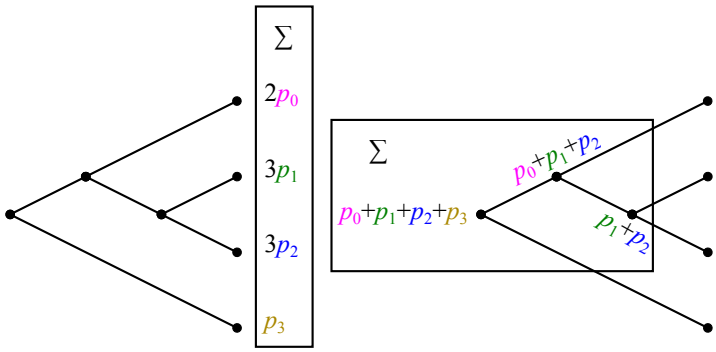

Figure 4.3: Example for rewriting the average source word length as sum of internal node probabilities.

codes, which means that all code words are of the same length. I.e., the code tree is a perfect binary tree.

Review of Tunstall codes

Tunstall's algorithm yields optimal V2F codes in an elegant way and with low computational complexity as follows. When the code tree is a perfect binary tree of a given depth d, the average code length according to (3.6) can be written as

$$\bar{\ell}_v(X) = \frac{d}{\bar{\ell}(S)} \tag{4.2}$$

and an optimal V2F code is found by maximizing the average source word length (3.2). In order to yield an intuitive view of Tunstall's algorithm, $\bar{\ell}(S)$ shall first be rewritten as the sum of the probabilities of all internal nodes of the source tree including the root node. An example for this is given in Fig. 4.3. The average source word length is the weighted sum of the source word length with the weights being the leaf node probabilities. I.e., the sum over the terms in the left-hand box in Fig. 4.3. Each of the source word probabilities appears with the multiplicity equal to the length of the source words. When rewriting each term as sum of leaf node probabilities (e.g., $p_1 + p_1 + p_1$ instead of $3p_1$), there are as many summands for each term as nodes are visited when traversing from the leaf node to the root node (including the root node). When moving a summand to each of these internal nodes, each internal node yields exactly one summand of each leaf node connected via its child branches. Thus, the summands of an internal node equal the probability of the internal node. And since there were no summands added or removed, the sum over all terms must still equal the average source word length. An example of this is given on the right-hand side of Fig. 4.3.

Tunstall codes can be derived by making use of this representation of the average source word length in a source tree growing approach. When starting with a root node with two connected leaf nodes for growing a source tree, a sequence of tree growing steps can be carried out until the source tree has as many leaf nodes as the perfect binary code tree of the predefined depth d. If the right tree growing decisions are made, the result will yield a source tree with maximum expected source word length. From the observation that the expected source word length equals the sum of the internal nodes probabilities follows that a particular source tree growing step will increase the expected source word length by the probability of this leaf node because it becomes an internal node. Consequently, the leaf node with the highest increase of the expected source word length, when extended, is the one with the highest leaf node probability. An extension step corresponds to replacing the largest value from the set of leaf node probabilities with two smaller values. Consequently, extension decisions cannot become incorrect because of subsequent extension decisions and the resulting source tree must have maximum average code word length.

Polynomial-based Algorithm

As discussed above, Tunstall codes are grown by repeatedly extending the source words with the highest probability yielding an optimal V2F code. When the source word probabilities are polynomials in p, the source word with the hightst probability may change depending on p. As in the polynomial-based Huffman algorithm, probability intervals can be derived such that for each interval, one particular source word has a higher probability than all other source words for any value of p in the interval.

An algorithmic description for jointly deriving Tunstall codes and intervals for which they are optimal V2F codes is as follows:

1. Start with a binary tree of height 1 and associate with it an interval I for which a set of Tunstall codes shall be derived, e.g., $I = [0, 0.5]$.

2. Derive the probability polynomials p_i for each leaf node of the tree.

3. Derive all values of p where any two probability polynomials have the same value and which are in the interval I. These are the roots of all differences of the probability polynomials $p_i - p_k \,\forall\, i \neq k$ in I.

4. Subdivide the interval I into non-overlapping subintervals according to the roots

found in step 3 neglecting intervals of zero width. Associate with each subinterval, a copy of the binary tree which belongs to the original interval I.

5. For each subinterval, find the leaf node whose probability polynomial p_i has the highest value for an arbitrary value of p in the subinterval. Extend this leaf node by adding two branches with two new leaf nodes.

6. For each subinterval separately, continue at step 2 until the binary tree has as many leaf nodes as desired. E.g., for a Tunstall code with length k code words, stop when the binary tree has 2^k leaf nodes.

7. Merge all neighboring subintervals whose binary trees have identical average source word length polynomials when used as source tree. Associate with the merged interval an arbitrary one of the original intervals binary trees.

The above algorithm yields a set of binary trees with associated intervals which are the source trees of Tunstall codes.

Using the example of Tunstall codes, the property that polynomial-based algorithms yield a unique number of results (with intervals) for a given algorithm is demonstrated. Tab. 4.1 shows the counts of possible canonical[1] Tunstall codes for given

Code word length	2	3	4	5	6
Number of codes	2	5	12	28	61

Table 4.1: Counts of possible canonical Tunstall codes for given code word lengths.

fixed code word length ranging from 2 to 6. This results from the polynomial-based implementation but is an intrinsic property of Tunstall codes. *No other canonical Tunstall codes exist for the given code word lengths.*

4.5 Polynomial-based package merge algorithm

The *package merge* algorithm [53] by Larmore and Hirschberg can be understood as generalization of the Huffman algorithm. For an i.i.d. source $\{S\}$ over $\mathcal{S}$, it finds an optimal prefix-free code when the maximum code word length is restricted to a predefined value d with $\lceil \log_2(|\mathcal{S}|) \rceil \leq d \leq H$ where H is the height of the corresponding Huffman code. When $d = H$, the resulting code is a Huffman code.

[1] With canonical Tunstall code, the canonical representation as defined for V2V codes is meant.

Although it is possible to make the package merge algorithm polynomial-based, it turns out to have by far more computational complexity than the polynomial-based Huffman algorithm. However, many effort has been spent to reduce the complexity of package merge [62, 63, 64, 65, 66, 57]. An in-depth discussion about the differences between the several package merge variants can be found in [57] and in [66]. Basically, the variants can be categorized into two types. The ones based on the original package merge and the ones based on the so-called *reverse package merge* approach [63]. The variants based on original package merge have particularly low complexity when $d \approx H$ while the variants based on reverse package merge perform best when $d \approx \lceil \log_2(|\mathcal{S}|) \rceil$ [66]. Since it is usually of interest to have code (and source) trees of strongly limited height (as discussed in the next chapter), the reverse package merge variants seem to be the more attractive ones in the context of this work. Furthermore, several optimizations can be applied which lead to the *lazy reverse run-length package merge* algorithm [57, p. 201] for which a polynomial-based version is derived. However, any of the variants yields identical results.

Review of the reverse package merge algorithm

In the following, the reverse package merge algorithm is shortly reviewed based on the presentation in [57]. Any of the package merge variants derives as output for each source symbol an optimal code word length, given the length constraint d. These code word lengths represent a canonical code tree, which can be converted into a code tree as described in [13, Sec. 5.2]. The reverse package merge algorithm initially assigns the maximum allowed code word length d to each source symbol. These code word lengths are then interatively shortened in a greedy manner until the Kraft inequality [13] is fulfilled with equality. For an i.i.d. source $\{S\}$ over $\mathcal{S} = \{\sigma_1, \sigma_2, ..., \sigma_n\}$ with each σ_i having an associated code word length ℓ_i, which is initially set to $\ell_i = d \forall \sigma_i \in \mathcal{S}$, the sum in the Kraft inequality $\mathcal{K}_0$ is initially given as

$$\mathcal{K}_0 = \sum_{i=1}^{|\mathcal{S}|} 2^{-\ell_i} = |\mathcal{S}| 2^{-d}. \tag{4.3}$$

$|\mathcal{S}|$ may not exceed 2^d in order to ensure that $\mathcal{K}_0$ doesn't exceed 1. If $|\mathcal{S}| = 2^d$, the sum in the Kraft inequality is 1 and no code word can be shortened. The interesting case is, when $|\mathcal{S}| < 2^d$ and consequently $\mathcal{K}_0 < 1$. Here, code words can be shortened until $\mathcal{K}_0$ equals 1. Shortening a code word from length x to $x - 1$ increases the Kraft sum by 2^{-x}. I.e., shortening a length d code word by 1 adds 2^{-d} to the Kraft sum while

shortening a length $d-1$ code word by 1 adds $2 \cdot 2^{-d}$ to the Kraft sum which is twice as much. In general, each length-one code word shortening increases the Kraft sum by a power of 2 times 2^{-d} and consequently, the total required Kraft sum increase can be expressed as an integer multiple of 2^{-d}. More precisely, let

$$\tilde{\mathcal{K}}_0 = \lambda\, 2^{-d} = 1 - \mathcal{K}_0 \tag{4.4}$$

be the value required to increase the Kraft sum to 1 where λ is the integer multiplier for 2^{-d}. Then

$$\lambda = 2^d - |\mathcal{S}|. \tag{4.5}$$

Carrying out the reverse package merge algorithm corresponds to successively shortening code words and decreasing λ to 0. Shortening a length x code word by 1 reduces λ by 2^{d-x}, i.e., a power of 2. λ can be interpreted as money which can buy code word shortenings. The optimal shortening steps minimize the average code length (3.6). Since the average source word length $\bar{\ell}(S)$ is fixed, the average code word length

$$\bar{\ell}(C) = \sum_{i=1}^{|\mathcal{S}|} p_S(\sigma_i)\ell_i \tag{4.6}$$

can be minimized instead of (3.6). From (4.6) can be seen that shortening a code word length ℓ_i by 1 decreases $\bar{\ell}(C)$ by $p_S(\sigma_i)$ independently of the length ℓ_i. The decrease of λ, however depends on the current length of a code word. Interpreting λ as money to buy code word shortenings, the 'price' of shortening a code word doubles with each shortening step already applied to this code word. The reverse package merge algorithm selects the code words to shorten by maximizing the associated decrease of $\bar{\ell}(C)$ for a given price. Furthermore, it has to ensure that exactly all of the money is spent in the end. I.e., λ has to reduce to 0. To accomplish this, the binary representation of λ is derived where the bits shall be given as $b_1, b_2, \cdots, b_k$ with b_1 being the LSB and b_k being the MSB. Bits of value 1 can buy code word shortenings. E.g., $b_1 = 1$ could buy a shortening of a length d code word while $b_2 = 1$ could buy a shortening of a length $d-1$ code word or, alternatively, two length d code words. Conversely, $b_1 = 1$ cannot buy a length $d-1$ shortening. For this reason, bits of λ are spent starting from the LSB. E.g. if $b_1 = 1$, out of all length d code words, the one with the hightest associated source word probability is shortened by 1. Next, if $b_2 = 1$, it can either shorten the length $d-1$ code word with highest associated source word probability, or the two length d code words with highest associated source word probabilities, depending of which leads to the larger decrease of $\bar{\ell}(C)$. The reverse package

merge algorithm maintains for each bit b_i of λ, a row r_i of purchasable items, ordered by the decrease of $\bar{\ell}(C)$ which the item would lead to (the so-called item probability) in decreasing order. A purchasable item contains either a shortening of a length $d+1-i$ code word or two items of the previous row r_{i-1}. The algorithmic description of the whole reverse package merge algorithm is as follows:

1. Initialize two empty rows $\tilde{r}$ and r_1. Initialize a bit index j with 1.
2. For each source word σ_i, create an item consisting of a list of indexes and an item probability where the list of indexes is initialized with index i and the item probability is set to the source word probability $p_S(\sigma_i)$. Add all items to the list $\tilde{r}$ and sort list $\tilde{r}$ by the item probability of the items in descending order.
3. Add a copy of all items in list $\tilde{r}$ to list r_j such that list r_j is sorted afterwards (this corresponds to interleaving elements of two sorted lists such that the resulting list is sorted).
4. When bit $b_j = 1$, remove the first item of row r_j and for each index i in the list of indexes, shorten the code word associated with source word σ_i by 1. Note that the list of indexes may contain a particular index more than once. In this case, the associated code word is shortened by 1 for each occurrence of the index.
5. If b_j is the MSB of λ, terminate. Otherwise, increase the bit index j by 1 and initialize an empty row r_j (the next row).
6. Remove the first two items of row r_{j-1} and merge them by summing their item probability and by concatenating their lists of indexes. Add the merged item to row r_j (a merged item is also denoted *package* where the package merge algorithm is named after).
7. If row r_{j-1} contains more than one item, continue with step 6. Otherwise proceed with step 3.

Several optimizations can be applied to the reverse package merge algorithm in order to reduce the computational complexity.

Lazy and runlength extension of reverse package merge

The probably most relevant reduction of computational complexity can be achieved with the so-called *lazy* technique [57]. Instead of computing all items of all rows r_j,

the creation of items is done on demand. I.e., when a requests to remove items occurs in step 4, this item is derived, including the generation of all possibly related items. To accomplish this, for each row a copy of row $\tilde{r}$ is maintained in order to be able to track for which row, items stemming from row $\tilde{r}$ have already been removed. This row is denoted $\tilde{r}_j$. Whenever a request to remove the first item of row r_j occurs (without having precomputed row r_j), the question arises what the first item of row r_j can be. It is either a package of the first two items of row r_{j-1} or the first item of row $\tilde{r}_j$, depending on which of both has the higher item probability. This involves deriving (but not yet removing) the first two packages of row r_{j-1} which is done by recursively repeating this procedure.

The resulting algorithm is the *lazy reverse package merge* algorithm and it avoids the creation of a large number of items which are not needed. A further improvement which can be applied is the so-called *runlength* concept. Roughly speaking, it can be seen as an improved sorting technique where source symbols which have identical probability are treated together for sorting since such symbols will always appear in an uninterrupted sequence in the sorted result. This, however, requires to determine the groups of symbols of the same probability in advance. As discussed in the next section, this is a beneficial improvement for a polynomial-based implementation of reverse package merge.

Polynomial-based implementation

The lazy reverse runlength package merge algorithm can be made polynomial-based by implementing it as finite-state machine according to the concept in Sec. 4.1.1. The resulting implementation is very extensive and a detailed description of it would go beyond the scope of this work. Instead, a number of specific aspects is discussed.

Firstly, a concept for sorting source word probability polynomials $p_S(\sigma_i)$ as required in step 2 of the algorithm is described. Before carrying out the actual sorting, identical polynomials are determined according to the *runlength* concept as described in Sec. 4.1.2. The yielded identical polynomials can be treated as a group during the subsequent sorting algorithm. This happens without subdividing intervals. The runlength extension is particularly useful in the case of source trees. For a length k source word, only $k+1$ different probability polynomials exist and consequently, for a source tree of height h, the number of different source word polynomials is upper-bounded by

$$\hat{N}_h = \sum_{n=1}^{h} n + 1 = h + \sum_{n=1}^{h} n = \frac{h^2 + 3h}{2}. \tag{4.7}$$

The number of possible different source words of length h is 2^h and consequently, for a source tree of height h, the number of different source words is upper-bounded by

$$\tilde{N}_h = \sum_{n=1}^{h} 2^n = 2^{h+1} - 2. \tag{4.8}$$

However, for a single source tree of height h, the number of source words cannot exceed

$$\bar{N}_h = 2^h, \tag{4.9}$$

which is a much better upper bound. When comparing $\hat{N}_k$ with $\bar{N}_k$, it becomes clear that the runlength extension can greatly reduce the complexity of the reverse package merge algorithm for source trees (for $k > 3$).

The *lazy* concept corresponds to only requiring a few of the largest polynomials out of a number of polynomials. For the remaining polynomials, their order is unimportant. To take advantage of this property, an appropriate sorting algorithm must be used. The so-called *heap sort* algorithm seems perfectly suited for this purpose. It is known to be asymptotically optimal in terms of computational complexity and furthermore, the sorted items are derived in descending order. In the case where only a number of largest items is required, the heap sort algorithm can be terminated after sufficient items are derived.

4.6 Chapter summary

In this chapter, a concept for jointly deriving V2V codes and associated probability intervals is presented. It is based on the idea to use a variable probability of one of the binary source to be encoded instead of a fixed value. It turned out that many algorithms for code tree design can also be carried out when the probability of one of the binary source is a variable. The main difference to conventional algorithms is that probabilities occur as polynomials in the probability of one and that polynomial roots need to be evaluated whenever a comparison of two probabilities is required by the algorithm.

An interesting property of polynomial-based algorithms is that they partition the probability interval into a set of disjoint and contiguous subintervals for which analytical exact interval boundaries are also yielded by the algorithm. Such a set of intervals and associated codes could directly be used in a P coder.

For some algorithms, like the Huffman algorithm or the Tunstall algorithm, a polynomial-based version is relatively simple to achieve while for some other algorithms this is much more complicated. For this purpose, the concept of finite-state machine-based algorithms was introduced. The algorithm, from which a polynomial-based version shall be desired, a finite-state machine-based implementation with particular properties is created. Such an implementation can transparently be converted into a polynomial-based version. This idea is applied to the very complex lazy reverse runlength package merge algorithm in order to obtain a polynomial-based implementation. The algorithms derived in this chapter are the basis for the derivation of V2V codes in the next chapter.

5

Design of limited size V2V codes

The concept of systematic V2V codes as presented in Sec. 5.5.2 is already published in [67, 68, 69].

As discussed in Ch. 3, V2V codes can be arbitrarily large and approach the source entropy arbitrarily close. For practical applications, restrictions may be introduced to the properties of the V2V codes to be used. These could be of various kinds as a limited source or code tree height, a limited number of leaf nodes for source and code tree, a minimum number of source symbols per code word and so forth. Given such constraints, the target of V2V code design is to find minimum redundancy codes. For the most types of V2V codes, efficient algorithms for finding optimal codes are not known. The literature related to V2V code design usually deals with efficient but suboptimal algorithms. In contrast to this, the design of optimal codes is addressed in this chapter, employing exhaustive search wherever necessary. Particularly for the PIPE coding system, a set of appropriate V2V codes needs to be designed only once, covering the whole probability interval. Therefore, designing V2V codes by exhaustive search is a one-time effort which is worth to invest since the resulting coding system is more efficient than a system with suboptimal V2V codes.

In the following, the polynomial-based algorithms as presented in the previous chapter are used to derive the different kinds of limited size V2V codes along with associated probability intervals.

5.1 The set of all possible V2V codes

In order to discuss the various types of restrictions which can apply to V2V codes, the set of all possible V2V codes without any restrictions shall be defined first. For this

purpose, let $\mathcal{P}$ be the set of all possible full binary trees and for $i \in \mathbb{N}$, let $\mathcal{P}_i$ be the set of all full binary trees with exactly i leaf nodes. Then

$$\mathcal{P} = \bigcup_{i \in \mathbb{N}} \mathcal{P}_{i+1}. \tag{5.1}$$

Note that $\mathcal{P}_0 = \mathcal{P}_1 = \emptyset$. As described in Ch. 3, a V2V code consists of a source tree $\mathcal{S} \in \mathcal{P}_i$ and a code tree $\mathcal{C} \in \mathcal{P}_i$, which are full binary trees with the same number i of leaf nodes, and a bijective mapping implemented by a permutation function $\pi \in \Pi_i$, where

$$\Pi_i = \{\pi_1, \pi_2, \ldots, \pi_{i!}\} \tag{5.2}$$

is the set of all possible permutations of i items. Thus, the set of all V2V codes with i leaf nodes is defined as

$$\mathcal{V}_i = \mathcal{P}_i \times \Pi_i \times \mathcal{P}_i \tag{5.3}$$

with each element of $\mathcal{V}_i$ being a triple of the form

$$v = (\mathcal{S}, \pi, \mathcal{C}). \tag{5.4}$$

The set of all possible V2V codes $\mathcal{V}$ is then given as

$$\mathcal{V} = \bigcup_{i \in \mathbb{N}} \mathcal{V}_{i+1}. \tag{5.5}$$

Given $\mathcal{S} \in \mathcal{P}_i$ and $\mathcal{C} \in \mathcal{P}_i$, one can find a (not necessarily unique) $\pi \in \Pi_i$, such that the average code length, defined as in (3.6), of the triple $(\mathcal{S}, \pi, \mathcal{C})$ is less or equal than the average code length of any triple $(\mathcal{S}, \pi', \mathcal{C})$, $\pi' \in \Pi_i$: Simply sort the source words by their probabilities and the code words by their lengths and assign source words with higher probabilities to code words with shorter lengths. Designing V2V codes of limited size (in some sense), can be seen as defining a finite subset of $\mathcal{V}$, denoted $\mathcal{V}'$, which contains all V2V codes that meet the size criteria. Optimal V2V codes are then found by minimizing the average code length over $\mathcal{V}'$ for a given p, i.e., by solving

$$v_{opt}(X) = \arg\min_{v \in \mathcal{V}'} \bar{\ell}_v(X). \tag{5.6}$$

This concept is basically an exhaustive search. While it is also possible to define infinite subsets of $\mathcal{V}$, for infinite subsets $\mathcal{V}' \subseteq V$ a code $v \in V'$ with minimal redundancy does not need to exist. An example for this phenomenon is the set of all so-called fixed-to-variable length (F2V) codes, defined in Sec. 5.4.1, since, as argued in Sec. 3.4, the infimum of the redundancies of all F2V codes is zero. A way to deal with infinite sets

of V2V codes (i.e., subsets of $\mathcal{V}$) is to split them into finite subsets by introducing some parameter as, e.g. the source tree height in the case of F2V codes. When an efficient algorithm for deriving optimal codes for these finite subsets is found, it is commonly referred to as optimal algorithm for the associated infinite subset. An example for this is the Tunstall algorithm which finds optimal variable-to-fixed-length (V2F) codes as discussed in 5.4.2.

5.2 V2V codes with Huffman or package-merge code trees

Finding optimal V2V codes in a finite set $\mathcal{V}'$ of V2V codes (i.e., a subset of $\mathcal{V}$) can be carried out by solving (5.6). Huffman's algorithm, or the more general package merge algorithm may be beneficial in solving (5.6) when $\mathcal{V}'$ has particular properties. Let

$$\mathrm{St} : \mathcal{V} \to \mathcal{P}, (\mathcal{S}, \pi, \mathcal{C}) \mapsto \mathcal{S} \tag{5.7}$$

yield the source tree of a V2V code. Consequently, $\mathrm{St}(\mathcal{V}')$ is the set of all source trees occurring in V2V codes in $\mathcal{V}'$. Let $\mathrm{Hu}(\mathcal{S}, X)$ yield a V2V code with a Huffman code as code tree for a given source tree $\mathcal{S}$ and binary random variable X. Analogously, let $\mathrm{Pm}(\mathcal{S}, X, d)$ be a V2V code with a code tree derived by the package merge algorithm with maximum code tree height d for a given source tree $\mathcal{S}$ and binary random variable X. The set

$$\mathcal{V}'_H(X) = \mathrm{Hu}(\mathrm{St}(\mathcal{V}'), X) \tag{5.8}$$

contains all V2V codes constructed by combining each source tree in V2V codes of $\mathcal{V}'$ with the corresponding Huffman code tree for a given binary random variable X. When

$$\mathrm{Hu}(\mathrm{St}(\mathcal{V}'), X) \subseteq \mathcal{V}', \tag{5.9}$$

(5.6) can be written as

$$v'_{opt}(X) = \underset{v \in \mathrm{Hu}(\mathrm{St}(\mathcal{V}'), X)}{\arg\min} \bar{\ell}_v(X), \tag{5.10}$$

since it is well-known that for a given source tree $\mathcal{S}$, Huffman's algorithm [21] leads to a minimum redundancy V2V code. In other words, instead of solving (5.6) over $\mathcal{V}'$, it is solved over a subset of $\mathcal{V}'$ containing only V2V codes that have Huffman codes as code trees. When

$$|\mathrm{Hu}(\mathrm{St}(\mathcal{V}'), X)| \ll |\mathcal{V}'|, \tag{5.11}$$

solving (5.10) is much less complex than solving (5.6) while still yielding an optimal result. I.e., (5.10) can be seen as an algorithmic optimization over (5.6) which is

Name	S3_1							S3_2			S3_3						S3_4					S3_5	
I_k	$(0, 0.25]$							$(0.25, 0.29]$			$(0.29, 0.33]$						$(0.33, 0.43]$					$(0.43, 0.5]$	
$M(\sigma_i)$	3	2	2	2	1	1	0	2	1	0	2	2	2	1	1	0	3	2	1	1	0	1	0
$N(\sigma_i)$	0	1	1	1	2	2	2	0	1	1	0	1	1	2	2	2	0	1	1	1	2	0	1
$\ell(k(\sigma_i))$	1	3	3	3	5	5	4	1	2	2	1	3	3	4	4	3	2	3	2	2	3	1	1

Table 5.1: Descriptive names, intervals I_k, and canonical source and code words of optimal V2V codes with maximum source tree height 3.

applicable only if the property defined by (5.9) holds. This property on subsets of $\mathcal{V}$ will be referred to as *Huffman property* of a set of V2V codes. Following the same argumentation for the package merge algorithm, the *package-merge property* can be defined.

5.3 Evaluation method

When the optimal codes are derived for a set of V2V codes using polynomial-based algorithms according to Ch. 4, each of the codes is associated with a probability interval. A common way to evaluate the compression performance of the codes is to compare their redundancy. To give an example, the optimal V2V codes where the source tree height doesn't exceed 3 are evaluated. They are listed in Tab. 5.1 and their redundancy is depicted in Fig. 5.1. The different graphs can unambiguously be associated with the intervals in Tab. 5.1. Since a graph belongs to the interval where it has the lowest redundancy (amongst all graphs), the interval boundaries must be located at probabilities where two neighboring graphs intersect. In the following diagrams, only the envelope of the graphs belonging to the optimal codes of a particular set of V2V codes are depicted. In this way, a comparison of two or more sets of V2V codes is possible without making the diagrams too confusing. The canonical representations of many V2V codes presented in the following sections are listed in Appendix A where also the associated interval boundaries can be looked up.

5.4 Selected sets of V2V codes

In the following, selected subsets of the set of all possible V2V codes $\mathcal{V}$ are discussed. The target is to derive and evaluate codes that are particularly suitable for the use in

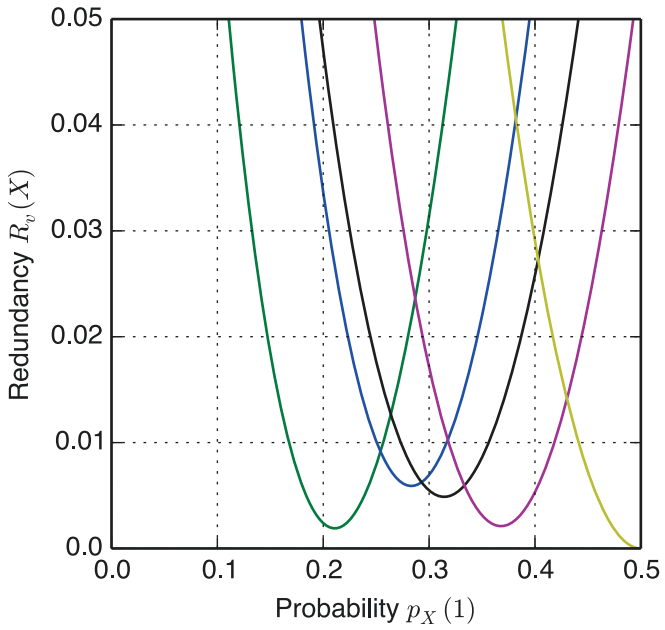

Figure 5.1: Redundancy of optimal V2V codes with maximum source tree height 3.

the PIPE coding concept. The canonical representations of the V2V codes discussed in this section are listed in Appendix A.1 together with the associated probability intervals.

5.4.1 Fixed-to-variable length codes

A widely known set of V2V codes is the set of fixed-to-variable length codes for which source trees must be perfect binary trees. Optimal F2V codes are determined by applying Huffman's algorithm to a given perfect binary source tree. The redundancy of optimal F2V codes approaches zero when the source tree height goes to infinity. Such codes may be beneficial in applications where encoder complexity shall be low because the number of symbols that are used to form a code word is fixed. I.e., given a sequence of input symbols, it can be determined in advance which of them will be used for a code word, independently of the values of the symbols. Fig. 5.2 shows the redundancy envelopes of optimal F2V codes derived by the polynomial Huffman algorithm. It is interesting to see that the redundancy of F2V codes with source tree height k is not always smaller than the redundancy of F2V codes of smaller source tree height. I.e., it may sometimes be beneficial to use a shorter source tree height than the maximum allowed.

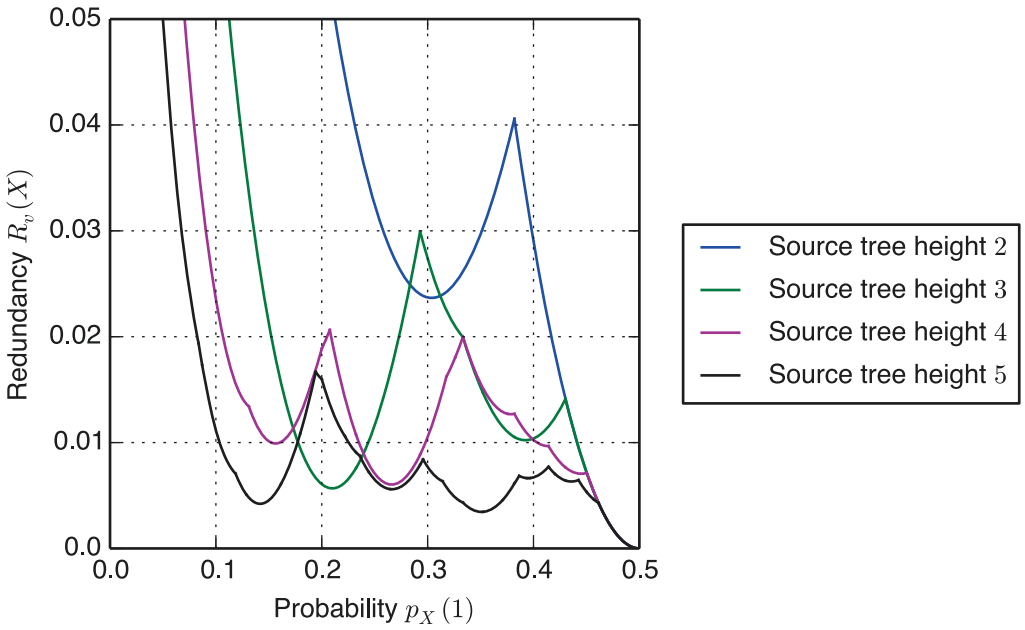

Figure 5.2: Envelopes of the redundancy of F2V codes of source tree height 2 to 5.

5.4.2 Variable-to-fixed length codes

Variable-to-fixed length (V2F) codes have code trees which are perfect binary trees. Optimal V2F codes are known as Tunstall codes after an algorithm by Tunstall [23]. They are also asymptotically optimal [70, 39]. Fixed-length code words may be advantageous because the length of the code words is fixed and it can be determined in advance at which position in the bit stream a particular code word is located. The redundancy envelope of Tunstall (V2F) codes with code word length ranging from 2 to 6 is depicted in Fig. 5.3. Note that, as expected, the interval boundaries associated with Tunstall codes with code tree height 2 and 3 as depicted in Fig. 5.3 and as listed in Tab. A.10 and Tab. A.11 match the Tunstall regions in [56, Fig. 4] for $j = 2$ and $j = 6$, respectively.

The redundancy of V2F codes tends to be higher than of F2V codes with comparable size. Particularly, V2F codes seem to have a systematic redundancy peak which moves from around $p = 0.2$ towards $p = 0$ for increasing code tree height.

5.4.2.1 Redundancy drawback of V2F and F2V codes

V2F and F2V codes have a systematic disadvantage in terms of redundancy compared to V2V codes. More precisely, from a V2F code which has an average code

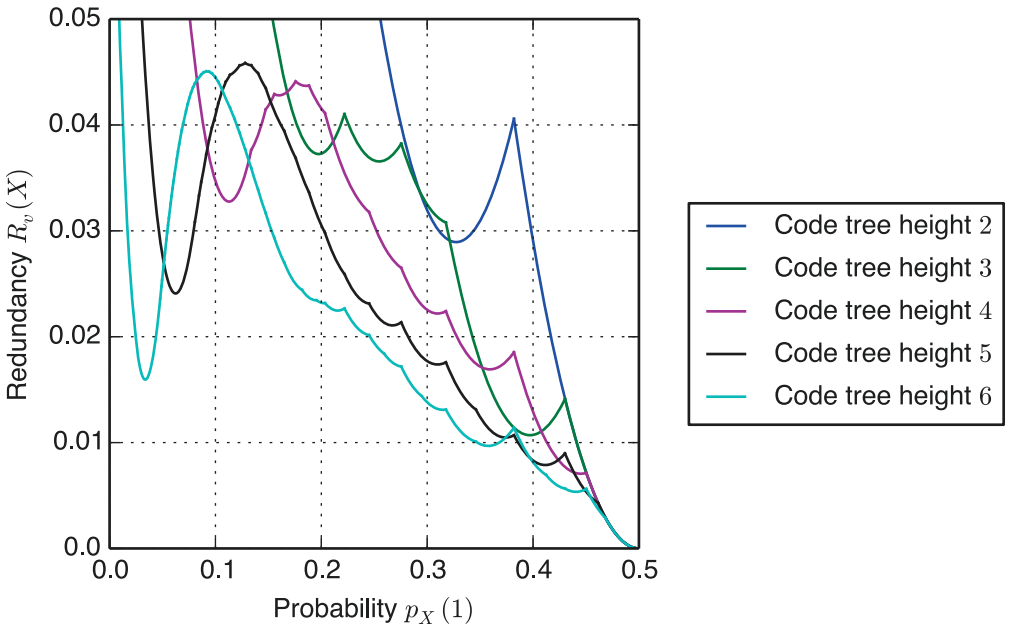

Figure 5.3: Envelopes of the redundancy of V2F (Tunstall) codes with code tree height ranging from 2 to 6.

length $\bar{\ell}_v(X) < 1$, a V2V code with smaller average code length can be derived. This works as follows. Any source tree has (at least) two source words σ_a and σ_b where the associated leaf nodes are siblings. Since all code words are of the same length, the two source words can always be associated with two code words where the leaf nodes are siblings as well. Consequently, the siblings in the source tree and the siblings in the code tree form an inner V2V code according to Sec. 3.9. This shows that canonical V2F codes are always composite (except the trivial V2F code with source and code tree height equal to 1). Since the average code length of this inner V2V code equals 1, the average code length of the corresponding outer V2V code must be smaller than of the original V2F code according to Lemma 3.5. For F2V codes, the same argumentation can be used with source and code tree exchanged.

5.4.3 Leaf-limited V2V codes

The next subset of $\mathcal{V}$ to be discussed contains all possible V2V codes with the number of leaf nodes $|\mathcal{S}|$ (of source tree and code tree) not exceeding a predefined maximum count x. V2V codes in this set

$$\tilde{\mathcal{V}}_x = \mathcal{V}_2 \cup \mathcal{V}_3 \cup \cdots \cup \mathcal{V}_x \tag{5.12}$$

are denoted leaf-limited variable-to-variable length (LV2V) codes. This class of codes is addressed in the most of the literature related to V2V codes [39, 22, 40, 38, 43, 37]. When analyzing the definition of LV2V codes, it becomes clear that each source tree occurring in LV2V codes in $\tilde{\mathcal{V}}_x$ can be combined with any applicable code tree and any applicable permutation. Thus, $\tilde{\mathcal{V}}_x$ has the Huffman property (i.e., (5.9) holds for $\tilde{\mathcal{V}}_x$) and optimal LV2V codes with maximum number of leaf nodes x are found by solving (5.10) over $\mathrm{Hu}(\mathrm{St}(\tilde{\mathcal{V}}_x), X)$ for a given binary i.i.d. source X.

Optimal leaf-limited V2V codes

To the authors best knowledge, an efficient algorithm for finding leaf-limited V2V codes with minimum redundancy is unknown. For finding optimal codes in $\tilde{\mathcal{V}}_x$, an exhaustive search is applied to the different source trees $\tilde{\mathcal{P}}_x = \mathrm{St}(\tilde{\mathcal{V}}_x)$ in $\tilde{\mathcal{V}}_x$ since code trees are Huffman codes. Consequently, the count of different source trees $\tilde{\mathcal{P}}_x$ is most relevant for the complexity of the exhaustive search. For $n \in \mathbb{N}$ let

$$C_n = \frac{2n!}{n!(n+1)!},$$

the so called Catalan number, see [71, p. 114]. Then it is well-known that the number of full binary trees with n leaf nodes equals C_{n-1} [71, p. 127]. The number of distinct source trees in $\tilde{\mathcal{V}}_x$ is thus given as

$$|\tilde{\mathcal{P}}_x| = \sum_{n=1}^{x-1} \mathrm{C}_n = \sum_{n=1}^{x-1} \frac{(2n)!}{(n+1)!n!}. \tag{5.13}$$

Tab. 5.2 shows the numbers of source trees in $\tilde{\mathcal{P}}_x$ for $2 < x < 14$. Canonical source

$\lvert\tilde{\mathcal{P}}_3\rvert$	$\lvert\tilde{\mathcal{P}}_4\rvert$	$\lvert\tilde{\mathcal{P}}_5\rvert$	$\lvert\tilde{\mathcal{P}}_6\rvert$	$\lvert\tilde{\mathcal{P}}_7\rvert$	$\lvert\tilde{\mathcal{P}}_8\rvert$	$\lvert\tilde{\mathcal{P}}_9\rvert$	$\lvert\tilde{\mathcal{P}}_{10}\rvert$	$\lvert\tilde{\mathcal{P}}_{11}\rvert$	$\lvert\tilde{\mathcal{P}}_{12}\rvert$	$\lvert\tilde{\mathcal{P}}_{13}\rvert$
3	8	22	64	196	625	2055	6917	23713	82499	290511

Table 5.2: Counts of source trees with the number of leaf nodes not exceeding x with $1 < x < 10$.

trees can be used here in order to further reduce complexity of the exhaustive search. However, it turned out, that on a modern computer, an exhaustive search over a set of source trees whose cardinality is approximately $|\tilde{\mathcal{P}}_{13}|$ can be carried out easily. The redundancy envelopes for codes with a maximum number of leaf nodes ranging from 3 to 9 are depicted in Fig. 5.4. Since $\tilde{\mathcal{V}}_x \subset \tilde{\mathcal{V}}_{x+1}$, all optimal LV2V codes in $\tilde{\mathcal{V}}_x$ are in $\tilde{\mathcal{V}}_{x+1}$ as well and thus the redundancy is a monotonically decreasing function of x.

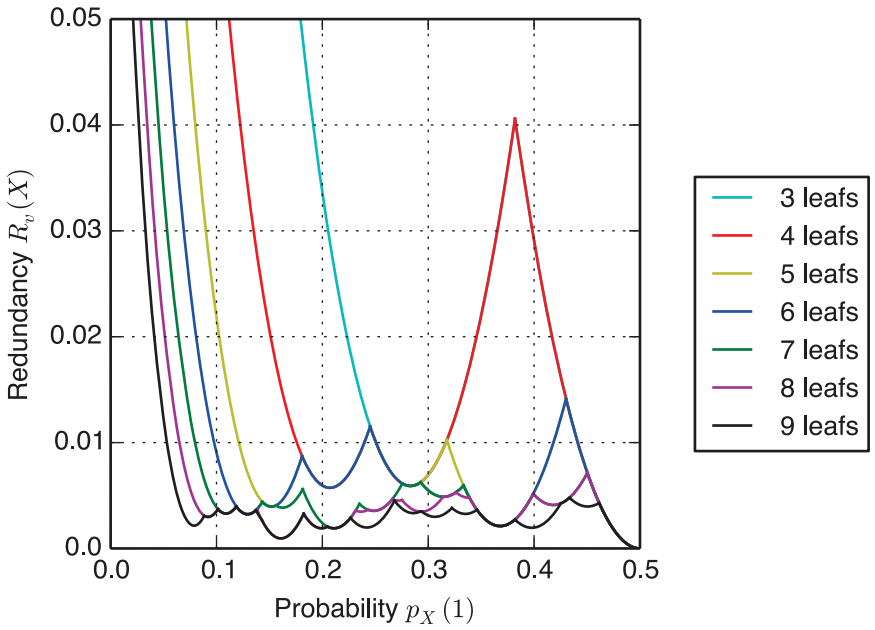

Figure 5.4: Envelopes of the redundancy of LV2V codes with the maximum number of leaf nodes ranging from 3 to 9.

5.4.4 Height-limited V2V codes

Two further parameters of V2V codes, for which it may be desirable to limit them, are the maximum height of the source and code tree. V2V codes with limited maximum source tree height are defined as

$$\hat{\mathcal{V}}_x = \{(\mathcal{S}, \pi, \mathcal{C}) \in \mathcal{V} \mid \forall s \in \mathcal{S} : \ell(s) \leq x\}. \tag{5.14}$$

V2V codes of this type shall be denoted source-height-limited variable-to-variable length (SV2V) codes. Analogously, V2V codes with limited code word length are given as

$$\bar{\mathcal{V}}_x = \{(\mathcal{S}, \pi, \mathcal{C}) \in \mathcal{V} \mid \forall c \in \mathcal{C} : \ell(c) \leq x\} \tag{5.15}$$

and these are denoted code-height-limited variable-to-variable length (CV2V) codes.

5.4.4.1 Optimal SV2V codes

Again, it is obvious that $\hat{\mathcal{V}}_x$ has the Huffman property (i.e., (5.9) holds for $\hat{\mathcal{V}}_x$) and optimal SV2V codes with maximum source tree height k are found by solving (5.10) over $\mathrm{Hu}(\mathrm{St}(\hat{\mathcal{V}}_x), X)$ for a given binary i.i.d. source X. In order to evaluate the complexity of

solving (5.10) over $\hat{\mathcal{V}}_x$, the number of different source trees in

$$\hat{\mathcal{P}}_x = \mathrm{St}(\hat{\mathcal{V}}_x) = \{y \in \mathcal{P} \mid \forall z \in y : \ell(z) \leq x\}, \tag{5.16}$$

i.e., $|\hat{\mathcal{P}}_x|$ turns out to be most relevant. $\hat{\mathcal{P}}_x$ can be derived from $\hat{\mathcal{P}}_{x-1}$ using the concept found in [72, p. 717]. The argumentation is as follows. A full binary tree y of height h with $h > 0$ can be split into two sub trees y_0 and y_1 by removing the root node and making the two connected child nodes the root nodes of the sub trees. The heights $h_{1,2}$ of the two sub trees $y_{1,2}$ then fulfill $0 \leq h_{1,2} < h$. When a tree in $\hat{\mathcal{P}}_{x+1}$ is split in this way, the resulting sub trees *must* be in

$$\hat{\mathcal{P}}_x^* = \hat{\mathcal{P}}_x \cup \{P^*\} \tag{5.17}$$

where P^* shall be a tree only consisting of a root node. Consequently, a tree in $\hat{\mathcal{P}}_{x+1}$ can be represented as the two subtrees which result from splitting, i.e., an ordered pair of trees in $\hat{\mathcal{P}}_x^*$. This leads to

$$|\hat{\mathcal{P}}_{x+1}| = |\hat{\mathcal{P}}_x^*|^2 = (1 + |\hat{\mathcal{P}}_x|)^2. \tag{5.18}$$

With the number of full binary trees $|\hat{\mathcal{P}}_1|$ of height 1 set to 1, the first six iterations of (5.18) are given in Tab. 5.3. A lower bound on $|\hat{\mathcal{P}}_x|$ can be defined using the

$\lvert\hat{\mathcal{P}}_1\rvert$	$\lvert\hat{\mathcal{P}}_2\rvert$	$\lvert\hat{\mathcal{P}}_3\rvert$	$\lvert\hat{\mathcal{P}}_4\rvert$	$\lvert\hat{\mathcal{P}}_5\rvert$	$\lvert\hat{\mathcal{P}}_6\rvert$	$\lvert\hat{\mathcal{P}}_7\rvert$
1	4	25	676	458329	210066388900	44127887745906175987801

Table 5.3: Count of full binary trees $|\hat{\mathcal{P}}_k|$ with $1 \leq k \leq 7$.

approximation

$$|\hat{\mathcal{P}}_x| = (1 + |\hat{\mathcal{P}}_{x-1}|)^2 > |\hat{\mathcal{P}}_{x-1}|^2. \tag{5.19}$$

Starting with $|\hat{\mathcal{P}}_2| = 4$, the lower bound is given as

$$|\hat{\mathcal{P}}_x| > 2^{2^{x-1}} \tag{5.20}$$

for $x > 2$. (5.20) shows that $|\hat{\mathcal{P}}_x|$ has at least double-exponential growth $\mathcal{O}(2^{2^n})$ which makes solving of (5.10) practically infeasible for large x. For source trees of height not exceeding $x = 6$, it may however be feasible to derive the optimal V2V codes.

As discussed in Sec. 3.6, for the redundancy of a V2V code it is sufficient to only regard one of multiple equivalent source trees in a set of source trees. For source trees of SV2V codes, it is thus sufficient to regard the set $\mathrm{Eq}(\hat{\mathcal{P}}_x)$ with the same canonical representations as $\hat{\mathcal{P}}_x$. However, it is difficult to calculate the number $|\mathrm{Eq}(\hat{\mathcal{P}}_x)|$ of

$\lvert\mathrm{Eq}(\hat{\mathcal{P}}_1)\rvert$	$\lvert\mathrm{Eq}(\hat{\mathcal{P}}_2)\rvert$	$\lvert\mathrm{Eq}(\hat{\mathcal{P}}_3)\rvert$	$\lvert\mathrm{Eq}(\hat{\mathcal{P}}_4)\rvert$	$\lvert\mathrm{Eq}(\hat{\mathcal{P}}_5)\rvert$	$\lvert\mathrm{Eq}(\hat{\mathcal{P}}_6)\rvert$
1	4	21	253	12360	4450860

Table 5.4: Counts of distinct full binary trees $\lvert\mathrm{Eq}(\hat{\mathcal{P}}_x)\rvert$ with $1 \leq x \leq 6$.

source trees with *distinct* multisets of leaf node probabilities. Tab. 5.4 shows these numbers which are derived by simulation. Since a simple closed-form expression for deriving $\lvert\mathrm{Eq}(\hat{\mathcal{P}}_x)\rvert$ is not known, few can be said about the order with which the size of $\lvert\mathrm{Eq}(\hat{\mathcal{P}}_x)\rvert$ grows in x. However, when comparing the cardinality of $\hat{\mathcal{P}}_x$ and $\mathrm{Eq}(\hat{\mathcal{P}}_x)$ for maximum source tree height $x = 6$, the number of elements is reduced by a factor of more than 47196 which makes the problem of finding optimal V2V codes for $\hat{\mathcal{P}}_6$ much easier. For $x > 6$ it can be guessed that already $\lvert\mathrm{Eq}(\hat{\mathcal{P}}_7)\rvert$ may be to large for exhaustive search since $\lvert\hat{\mathcal{P}}_7\rvert > 4.4 \cdot 10^{22}$.

The recursive code construction scheme described above can also be applied to canonical source trees. Joining two source trees at their roots (creating a new root) corresponds to prefixing all source words of the one source tree with a 0 and all source words of the other source tree with a 1. This operation can be applied to canonical source trees by appropriately increasing counts of ones or zeros. Consequently, a better approximation of $\lvert\mathrm{Eq}(\hat{\mathcal{P}}_7)\rvert$ is given as

$$\lvert\mathrm{Eq}(\hat{\mathcal{P}}_7)\rvert \leq (1 + \lvert\mathrm{Eq}(\hat{\mathcal{P}}_6)\rvert)^2 = 4450861^2 > 1.9 \cdot 10^{13} \tag{5.21}$$

which still seems too big for an exhaustive search.

The redundancy envelopes of optimal SV2V codes with maximum source tree height ranging from 2 to 5 are depicted in Fig. 5.5. These codes tend to have a higher redundancy than LV2V codes for probabilities close to 0, which makes sense because a source word may be longer for an LV2V code than for an SV2V code that performs similarly good for higher probabilities. E.g., when comparing the SV2V codes of maximum depth 4 with the LV2V codes with up to 7 leafs, they perform similarly good (in terms of redundancy) for probabilities greater than 0.15 while the LV2V codes perform much better at lower probabilities. Assuming the case where $p = 0$, the cheapest way of encoding symbols is to map as many zeros as possible to a code word of length 1. For the SV2V codes of maximum depth 3, this would mean, three zeros can be encoded with a 1 bit code word, yielding a redundancy of $1/3$ bit while with an LV2V code of up to 7 leafs a code word of six zeros can be realized which corresponds to $1/6$ bit redundancy.

It is interesting to see that F2V codes of a particular source tree height mostly have

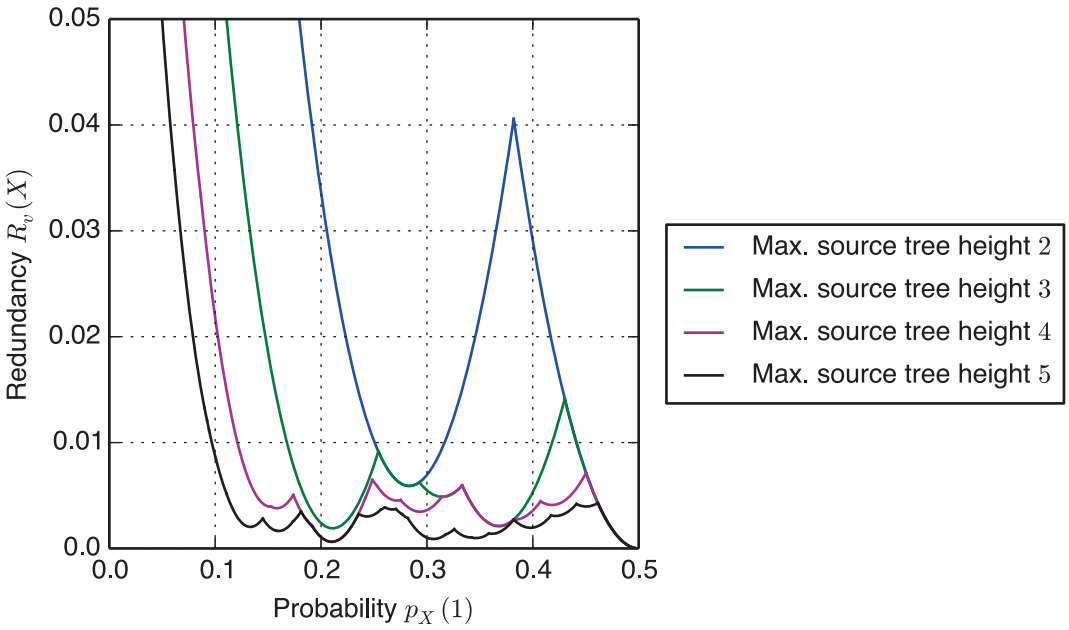

Figure 5.5: Envelopes of the redundancy of SV2V codes with maximum source tree height 2 to 5.

a higher redundancy than SV2V codes of the same maximum source tree height. This can be observed when comparing the redundancy charts of SV2V codes of a maximum depth k in Fig. 5.5 with the corresponding F2V codes of Fig. 5.2. I.e., in some cases it may reduce the redundancy when shorter source words than these of the maximum allowed length are used.

5.4.4.2 Optimal CV2V codes

CV2V codes are created by using the package merge algorithm instead of the Huffman algorithm for code tree generation. The source tree height is only limited by the number of code words and the number of code words is only limited by the maximum code tree height. More precisely, for a maximum code tree height d, the maximum possible number of leaf nodes (of source and code tree) is 2^d. Obviously, when a code tree of height d has 2^d leaf nodes, all of them must be at the same level, which corresponds to a perfect binary code tree. In this case, the optimal source tree can be derived using Tunstall's algorithm and the exhaustive search only needs to be applied to all possible source trees with up to $2^d - 1$ leafs. According to (5.13) the valid source trees for CV2V codes of maximum code tree height d are $\tilde{\mathcal{P}}_{2^d-1}$. The cardinalities of $\tilde{\mathcal{P}}_{2^d-1}$

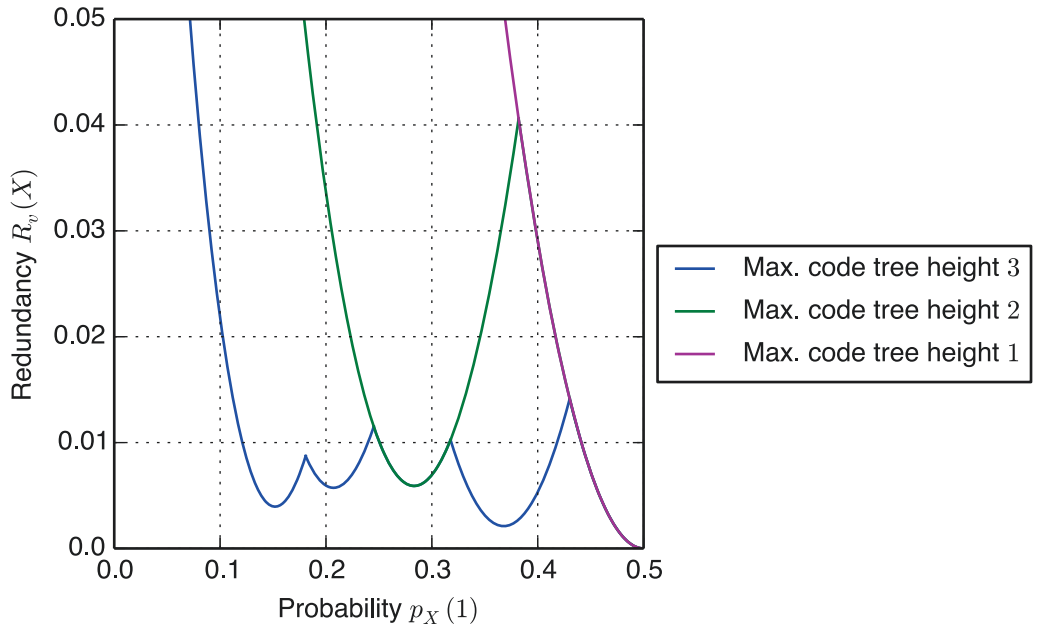

Figure 5.6: Envelopes of the redundancy of CV2V codes with maximum code tree height 1 to 3.

grow much faster in d than source trees for SV2V codes as can be seen in Tab. 5.5 for $1 < d < 7$. Consequently, only very small optimal CV2V codes can be derived

$\lvert\hat{\mathcal{P}}_3\rvert$	$\lvert\hat{\mathcal{P}}_7\rvert$	$\lvert\hat{\mathcal{P}}_{15}\rvert$	$\lvert\hat{\mathcal{P}}_{31}\rvert$	$\lvert\hat{\mathcal{P}}_{63}\rvert$
3	196	3707851	5175497420902740	$> 3.2 \cdot 10^{34}$

Table 5.5: Counts of source trees $\lvert\hat{\mathcal{P}}_{2^d-1}\rvert$ for code trees of maximum height $1 < d < 7$.

by exhaustive search. Fig. 5.6 shows CV2V codes with maximum code tree height ranging from 1 to 3.

5.4.4.3 Optimal SCV2V codes

Next, V2V codes with limited source and code tree height are discussed. Such codes shall be denoted SCV2V codes. Limiting source and code tree height of a V2V code is of particular interest for an implementation of a V2V encoder or decoder that is based on a lookup table. For example, a V2V encoder may use k input symbols to access an array of size 2^k. The addressed element stores the corresponding code word and the length of the source word. Then it removes the source word from the

		Max. code tree height c				
		2	3	4	5	6
Max. source tree height s	2	3	4	x	x	x
	3	3	20	21	21	21
	4	x	67	252	253	253
	5	x	127	4369	12359	12360
	6	x	159	31865	2167044	4450859

Table 5.6: Counts of distinct canonical source trees that can occur in SCV2V codes for given maximum source and code tree height.

input sequence and outputs the code word. Using this approach, an array of size 2^k is able to implement all V2V codes with maximum source tree height of k. The same concept can be used for a decoder implementation when source and code tree are exchanged. In a practical coding application, it is usually desired to have a good trade-off between lookup table sizes in encoder and decoder and the corresponding redundancy.

Limiting source and code tree height reduces the counts of applicable source trees for exhaustive search compared to limiting only one of the two parameters. The counts of distinct (canonical) source trees that can occur in SCV2V codes are given in Tab. 5.6. They are derived by intersecting sets of canonical source trees for SV2V codes and CV2V codes. Not all of the s and c combinations make sense. A code tree of height c has at least $c+1$ leaf nodes which requires a source tree of height $\lceil \log_2(c+1) \rceil$ or more. Analogously, source trees of height s require code trees of height $\lceil \log_2(s+1) \rceil$ or more. Fields for which these relationships are not fulfilled in Tab. 5.6 are marked with an 'x'. Figs. 5.7 and 5.8 show redundancy envelopes of a variety of different SCV2V codes. They are derived by applying the polynomial-based lazy reverse run-length package merge algorithm to the sets of source trees, which is, in the case of maximum source and code tree height equal to 6, a substantial computational effort. However, these codes (with maximum source and code tree height of 6) show a remarkably low redundancy which makes them interesting for practical applications.

Memory requirements of SCV2V codes

The memory required to implement a particular SCV2V encoder with maximum source tree height s and maximum code tree height c can be derived as follows. For each

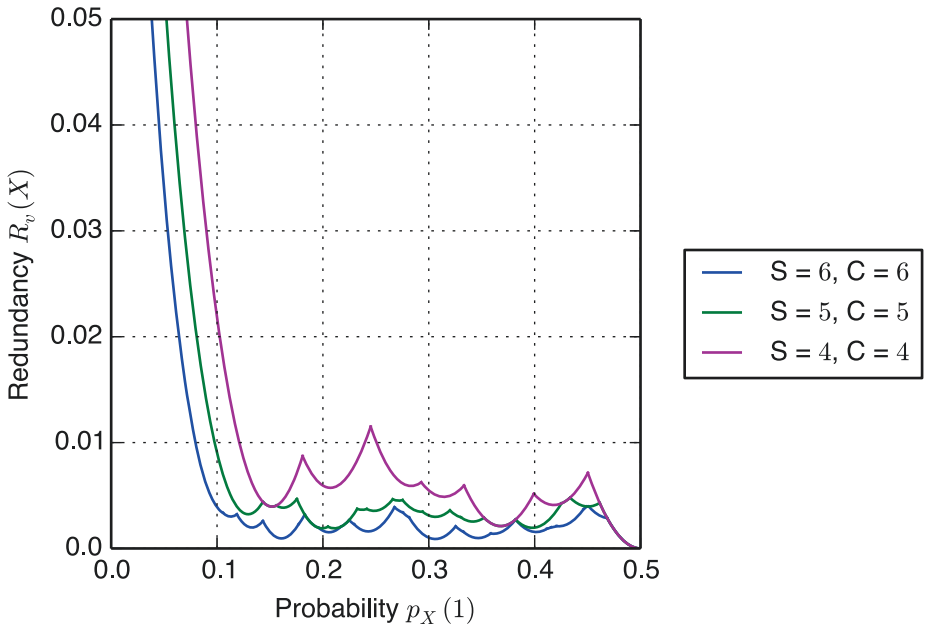

Figure 5.7: Envelopes of the redundancy of SCV2V codes with maximum source and code tree height S and C.

element in the table, a code word, its length and the source word length needs to be stored. The length of a code word can be stored by using only one additional bit. This is done by storing the code word left-aligned in the lookup table and adding a 1 bit to it. The remaining bits up to $c+1$ in total are filled with 0 bits. To unambiguously determine a code word represented in this way, from the first $c+1$ bits of a lookup table element, as many bits are removed from the right until a 1 is removed. The remaining bits are the code word. Furthermore, the length of the source word requires $\lceil log_2(s) \rceil$ bits. This amounts to

$$\mathcal{M}_{enc} = 2^s(c + 1 + \lceil log_2(s) \rceil) \tag{5.22}$$

bits. A decoder works in the same way, but with source and code tree exchanged. Consequently, the memory for a decoder implementation is given as

$$\mathcal{M}_{dec} = 2^c(s + 1 + \lceil log_2(c) \rceil). \tag{5.23}$$

Memory requirements for SCV2V encoders is listed in Tab. 5.7. For the decoder, the same values apply with s and c exchanged. As in Tab. 5.6, combinations of s and c which don't make sense are marked with in 'x' in Tab. 5.7 as well. In summary, SCV2V codes seem to be most suitable for lookup table-based encoding and decoding applications.

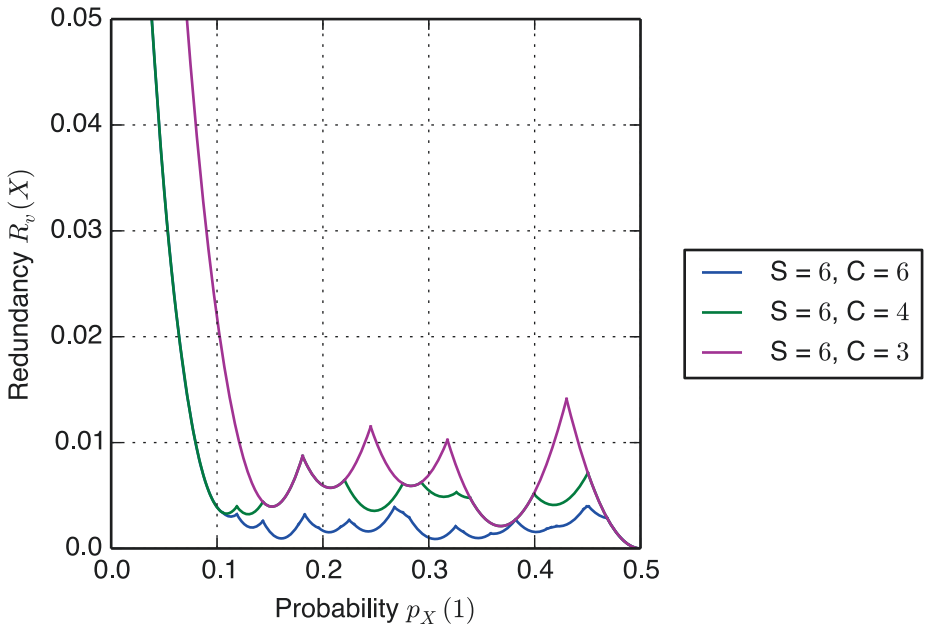

Figure 5.8: Envelopes of the redundancy of SCV2V codes with maximum source and code tree height S and C.

		Max. code tree height c				
		2	3	4	5	6
Max. source tree height s	2	2	2.5	x	x	x
	3	5	6	7	8	9
	4	x	12	14	16	18
	5	x	28	32	36	40
	6	x	56	64	72	80

Table 5.7: Memory required for implementing a lookup table-based SCV2V encoder for given maximum source and code tree heights in bytes.

5.5 Particular V2V codes

Several further types of V2V codes can be found in the recent literature. In the following, two kinds of V2V codes, which appear in the context of the standardization of H.265/HEVC, are evaluated and compared to the size-limited V2V codes of the previous sections.

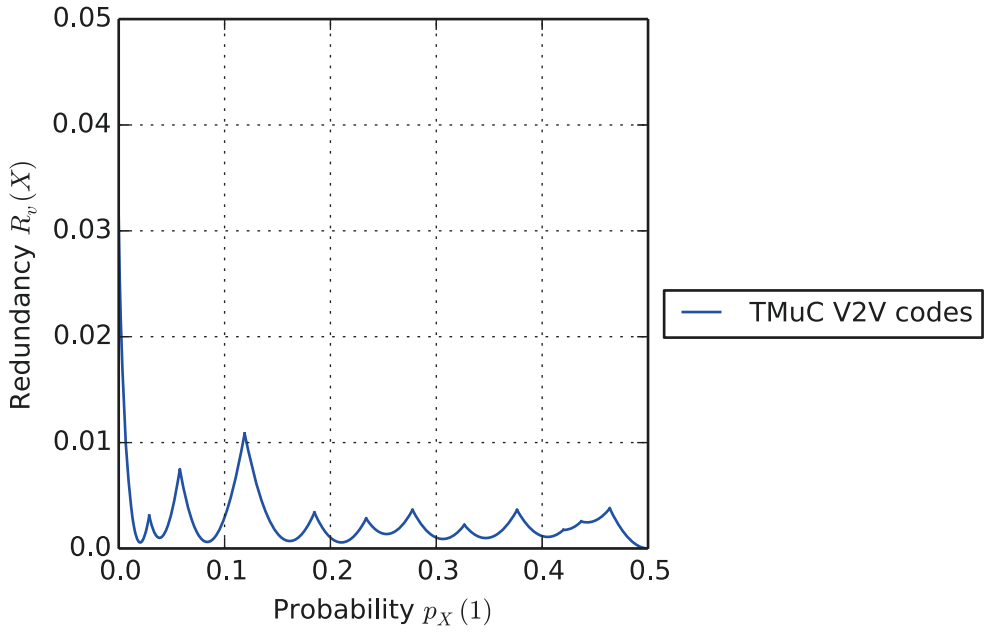

Figure 5.9: Envelopes of the redundancy of the 12 TMuC V2V codes.

5.5.1 HEVC-related V2V codes

The PIPE coding system, as originally proposed [29] as entropy coding engine for the video compression standard H.265/HEVC, uses twelve V2V-based binary coders. The algorithms for the design of these V2V codes is described in [9] and it is similar to the exhaustive search for LV2V codes. The PIPE coding system and the set of 12 V2V codes are included in the so-called *Test Model under Consideration (TMuC)* version 0.2 [73] as used during the standardization process of H.265/HEVC. This set of V2V codes shall be denoted *TMuC V2V codes* in this text and its redundancy envelope is depicted in Fig. 5.9. The canonical representations and associated probability intervals are listed in Appendix A.2.1. When compared to the SCV2V codes with maximum source and code tree height of 6, the TMuC V2V codes have a very similar redundancy for $p_X(1) > 0.1$. However, they have source tree heights of up to 12 and code tree heights of up to 11 to achieve this. For $p_X(1) \leq 0.1$, SCV2V codes are not satisfactory.

5.5.2 Systematic V2V codes

Next, the so-called *systematic V2V codes* [69] are discussed which have a more or less simple construction rule. The idea behind such codes is to implement them using a simple logic and to avoid the requirement of a lookup table. However, since these codes are still V2V codes, an implementation based on a lookup table is still possible if desired. Systematic V2V codes in combination with the PIPE coding concept are proposed [67, 68] as a replacement of the CABAC entropy coding engine of the H.265/HEVC standard. They are less complex than the TMuC V2V codes, but have a slightly increased redundancy as discussed later. Furthermore, having only 8 instead of 12 V2V codes is also a reduction of the complexity.

The systematic V2V codes of [69] are mainly based on two construction rules. The so-called *unary-to-rice codes* and the *bin-pipe codes*. These codes are inspired by an analysis of the structure of exhaustively derived V2V codes. It turns out that for the two extreme cases $p_X(1) = 0$ and $p_X(1) = 0.5$, the V2V codes tend to develop a particular structure. This is, for $p_X(1) = 0$, a string of ones of maximum length, mapped to a code word of length 1, leading to the unary-to-rice codes and for $p_X(1) = 0.5$, most of the source and code words are of same length. Only two source words are associated with code words of a by 1 differing length.

Unary-to-rice codes

A unary-to-rice code of degree k has a truncated unary source tree with $2^k + 1$ leaf nodes. The code word of the truncated source word (only consisting of ones) is of length 1, all other code words are of length $k + 1$. Unary-to-rice codes are a special case of the so-called *block MELCODE*[1] by Ono et al. [24], which has a Golomb code [75] as code tree. Furthermore, unary-to-rice codes appear in [34] and [35] where they are denoted *truncated run-length* (TRL) codes. An example is given in Tab. 5.8. A decoder implementation of such a code becomes relatively simple and could be according to the following rule:

If the first symbol is 1, decode 2^k ones. Otherwise, if the first symbol is 0, interpret the next k symbols as binary representation of integer j and decode j ones followed by one zero.

[1] The name "MELCODE" is presumably derived from "Mitsubishi Electric Company" (see Ch. 20.3.1 of [74]).

Source word	Code word
0	0000
10	0001
110	0010
1110	0011
11110	0100
111110	0101
1111110	0110
11111110	0111
11111111	1

Table 5.8: Unary-to-rice code of degree $k = 3$.

An encoder implementation could be similarly simple by maintaining a counter which counts occurrences of ones until a zero occurs or until k ones are received. Particularly for large k, unary-to-rice codes are very big and a counter-based encoder or decoder as outlined above is very memory-efficient. A detailed discussion of a counter-based implementation is given in [67].

An interesting property of unary-to-rice codes is that they are optimal for $p_X(1) \to 0$ because the length of the source word consisting of only ones is maximized and the length of the associated code word is minimized [40, Sec. 7.1.3]. Consequently, the redundancy of a unary-to-rice code for a binary random variable X_0 with $p_{X_0}(1) \to 0$ is given as

$$R_v(X_0) = \frac{1}{2^k}. \tag{5.24}$$

Fig. 5.10 shows the redundancy of unary-to-rice codes for $k = 0, ..., 6$. These codes are the ideal supplement to SCV2V codes because they close the gap for $p_X(1) < 0.1$.

Bin-pipe codes

While unary-to-rice codes are particularly suitable for probabilities close to zero, bin-pipe codes have a low redundancy for probability around $1/2$.

The source tree of a bin-pipe code of degree k has a source word consisting of k ones and an associated code word consisting of $k - 1$ zeros. Furthermore, it has a source word consisting of $k - 1$ zeros which has as associated code word consisting of k ones. All other source words are of minimum possible length such that the soruce tree is a full binary tree and the associated code words equal the source word. An

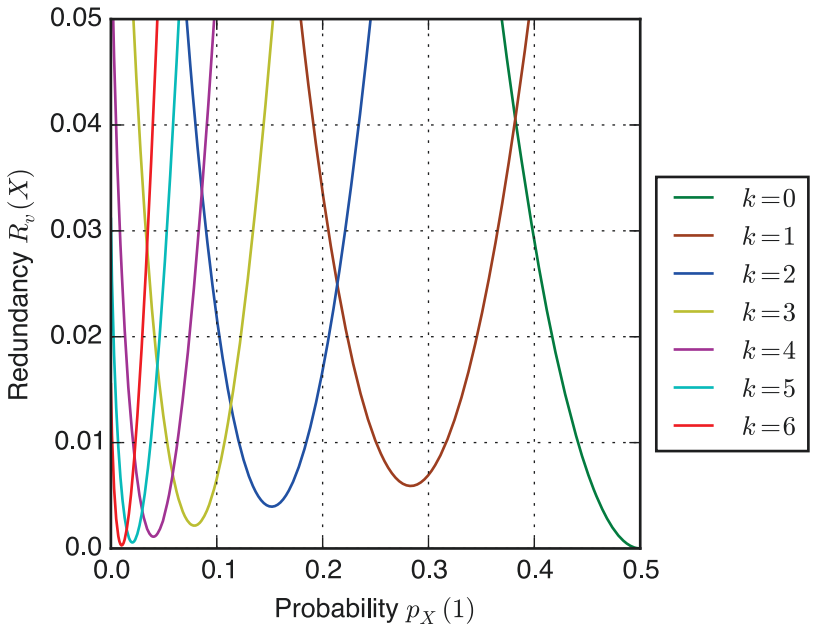

Figure 5.10: Redundancy of unary-to-rice codes.

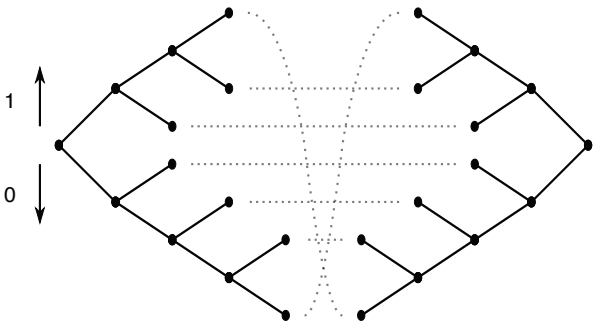

Figure 5.11: Source and code tree of bin-pipe 4.

Source word	Code word
0000	111
0001	0001
001	001
01	01
10	10
110	110
111	0000

Table 5.9: Bin-pipe 4 source and code words.

example for this is given in Fig. 5.11 and Tab. 5.9. The redundancy of bin-pipe codes is depicted in Fig. 5.12. The *double truncated unary* structure also allows for efficient, counter-based implementations in the encoder and the decoder as described in [67]. Furthermore, note that bin-pipe code with $k = 2$ equals the unary-to-rice code with $k = 1$.

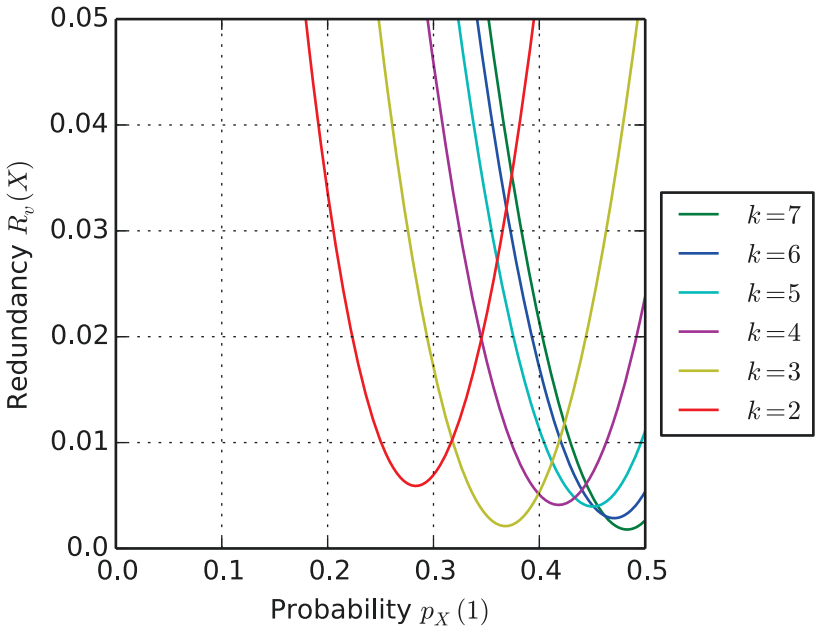

Figure 5.12: Redundancy of bin-pipe codes.

Proposed setup of systematic V2V codes

In this section, an exemplary PIPE coding setup using systematic V2V codes is discussed. In principle, with unary-to-rice codes, it is possible to cover the whole probability interval as can be seen from Fig. 5.10.

Since the smallest occurring probability in H.265/HEVC is $p \approx 0.02$, the unary-to-rice V2V codes with $k < 6$ shall be used. Only using these codes, however, results in a relatively high overall redundancy. Consequently, for some of the 'redundancy peaks' between unary-to-rice codes with $0 \leq k \leq 2$, further V2V codes are added. In order to also have systematic V2V codes to fill these gaps, and to not add too many further codes, the bin-pipe code with $k = 3$ is selected for reducing the redundancy peak between unary-to-rice codes with $k = 0$ and $k = 1$. For the second-largest redundancy gap, unfortunately, no bin-pipe code is able to fill it. Instead, a F2V code with source word length 3 is selected, which is denoted *three bin code* with source and code word table according to Tab. 5.10. The three bin code is in fact one of the optimal F2V codes and turns out to also have a structure that follows a simple construction rule. The length of a code word equals one plus two times the number of ones of a source word. This property can be used in encoder or decoder implementations to avoid the use of lookup tables. More detail about a simple algorithmic implementation of the

Source word	Code word
000	0
001	100
010	101
100	110
011	11100
101	11101
110	11110
111	11111

Table 5.10: Source and code words of the 'three bin code'.

three bin code can be found in [67].

Fig. 5.13 shows the redundancy envelope of the set of six unary-to-rice codes[1] plus the bin-pipe code with $k = 3$ and the three bin code. The redundancy peaks in

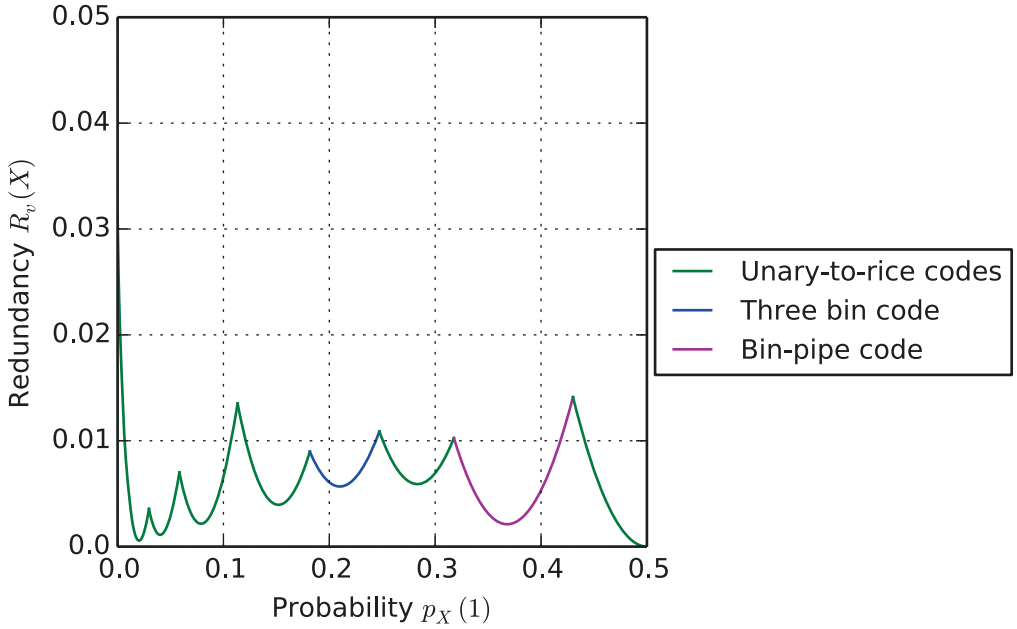

Figure 5.13: Redundancy envelope of the systematic V2V codes.

between two neighboring V2V codes redundancy curves are now relatively balanced such that a further V2V code would not contribute much to a redundancy reduction. This configuration is designed to have low redundancy over the whole probability in-

[1] Two of the unary-to-rice codes are also bin-pipe codes.

terval and is proposed to the H.265/HEVC video compression standard [67, 68]. The canonical representations of the systematic V2V codes of Fig. 5.13 and the associated probability intervals are listed in Appendix A.2.2.

5.6 Chapter summary

In this chapter, V2V codes with minimum redundancy for several constraints are derived. The basic principle is to determine a candidate set of canonical source trees and to derive associated code trees using the polynomial-based algorithms of the previous chapter. From the set of V2V codes found in this way, the ones with minimum redundancy are selected.

The investigated constraints are a limited number of leaf nodes of source and code tree, a limited maximum source tree height, a limited maximum code tree height, and combinations of them. It was shown that each particular class of codes has its own complexity, primarily determined by the cardinality of the associated candidate set of source trees. However, by exploiting the insights gained in Ch. 3, the cardinality of some candidate sets of source trees could greatly be reduced. Furthermore, it was pointed out that a limited source and code tree height is of particular interest for lookup table-based implementations of V2V codes. Consequently, a wide variety of such codes was derived.

The analysis of exhaustively derived V2V codes reveals several structural properties from which construction rules for V2V codes can be derived. This leads to the concept of systematic V2V codes. Of particular interest amongst the systematic V2V codes are the unary-to-rice codes, which appear several times in the related literature. It was shown that they are almost the only option for sources with a particularly small probability of one. To derive a P coder with a low redundancy, it makes sense to combine V2V codes that are found by exhaustive search with unary-to-rice codes as will be shown in the next chapter.

6

Application of PIPE coding to H.265/HEVC

The concept of complexity-scalable entropy coding of Sec. 6.3 is already published in [68, 76, 77].

In this chapter, various PIPE coding configurations are designed for the video coding standard H.265/HEVC in order to explore strengths and weaknesses of the concept. The focus lies on redundancy, complexity, and throughput. In Sec. 6.1, the V2V codes designed in Ch. 5 are used to create a PIPE coding setup with low redundancy. Various strategies for the selection of V2V codes for a P coder setup are presented. Furthermore, the redundancy of the M coder of CABAC is analyzed and compared to the various P coders. The aspects of complexity and throughput are discussed in Sec. 6.2 for CABAC and PIPE coding. The focus lies on the differences between the M and P coder with respect to the capabilities for parallelization and joint processing of consecutive bins. Based on this, an exemplary complexity-scalable configuration is presented in Sec. 6.3, where the trade-off between compression efficiency and complexity can be configured according to the needs of the coding application. Techniques for increasing the throughput of PIPE coding are discussed in Sec. 6.4.

6.1 P coder design

In this section, the aspects of selecting appropriate V2V codes for a P coder are discussed. One possible strategy for P coder design is to first preassign the number of desired bin coders and then select appropriate V2V codes for them out of a candidate

set. The statistical properties of the coding bins $\{G_i\}$ are given as conditional pmfs $p_{G_i}(\cdot|\mathbf{g}_{i-1})$ during encoding or decoding in the P coder. The coding bins are distributed to the individual V2V codes according to their conditional probabilities of one so that each V2V coder receives symbols with similar conditional pmfs. This is an important aspect since the V2V codes of the previous chapters are optimized for i.i.d. sources. Moreover, the calculation of the average code length of a V2V code according to (3.6), which is the basis for the most algorithms of the previous chapters, requires (3.1) to hold. An example for a situation where (3.1) does not hold is a stationary Markov source where the source word probabilities differ from (3.1) as shown by Wiegand and Schwarz in [8, Sec. 3.3.2].

A typical P coder is usually designed to operate close to the entropy rate (2.6). I.e., the average code length per symbol (2.7) of the coding bins $\{G_i\}$ encoded with a P coder shall be minimized. For this purpose, the average code length per symbol shall be derived for PIPE coding in the following. In CABAC and PIPE coding, the values that the conditional probability of one $p_{G_i}(1|\mathbf{g}_{i-1})$ of a coding bin g_i can attain are in the set

$$\mathcal{Q} = \{\omega_k | k \in \{0, 1, \ldots, 62\}\} \tag{6.1}$$

with

$$\omega_k = \frac{1}{2}\alpha^k \tag{6.2}$$

and where $\alpha = \sqrt[63]{3/80} \approx 0.949$ is a parameter of the probability estimator [5]. Let B_k be a bin coder and let $\ell_{B_k}(\{G_i\}, (g_1, g_2, \ldots, g_n))$ be the encoded length of a sequence $g_1, g_2, \ldots, g_n$ from random process $\{G_i\}$ when encoded with bin coder B_k. Moreover, assume a P coder B' that employs a separate bin coder B_k for each ω_k so that the encoded length $\ell_{B'}(\{G_i\}, (g_1, g_2, \ldots, g_n))$ of a sequence of coding bins $g_1, g_2, \ldots, g_n$ can be expressed as

$$\ell_{B'}(\{G_i\}, (g_1, g_2, \ldots, g_n)) = \sum_{k=0}^{62} \ell_{B_k}(\zeta(\{G_i\}, (g_1, g_2, \ldots, g_n), \omega_k)) \tag{6.3}$$

where $\zeta(\{G_i\}, (g_1, g_2, \ldots, g_n), \omega_k)$ shall be defined as the subsequence of coding bins in the sequence $g_1, g_2, \ldots, g_n$ for that the associated conditional probability of one $p_{G_1}(1),\ p_{G_2}(1|g_1), \ldots,\ p_{G_n}(1|\mathbf{g}_{n-1})$ equals ω_k. The average code length per symbol

(2.7) that P coder B' produces when encoding $\{G_i\}$ is given as

$$\bar{L}(\{G_i\}, B') = \lim_{n\to\infty} \frac{1}{n} \sum_{(g_1,g_2,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1, g_2, \ldots, g_n) \sum_{k=0}^{62} \ell_{B_k}(\zeta(\{G_i\}, (g_1, g_2, \ldots, g_n), \omega_k)). \tag{6.4}$$

Note that the limit does not need to exist.

Next, let $\tilde{B}$ be a P coder for that ideal binary arithmetic coders are used as bin coders B_k. More precisely, bin coder B_k shall be an ideal binary arithmetic coder $\hat{B}(Y_k)$ for a random variable Y_k. Substituting (2.9) in (6.3) yields

$$\ell_{\tilde{B}}(\{G_i\}, (g_1, g_2, \ldots, g_n)) = -\sum_{k=0}^{62} \big(M(\zeta(\{G_i\}, (g_1, g_2, \ldots, g_n), \omega_k)) \log_2 p_{Y_k}(1) + N(\zeta(\{G_i\}, (g_1, g_2, \ldots, g_n), \omega_k)) \log_2 p_{Y_k}(0)\big), \tag{6.5}$$

which is the encoded length of sequence $g_1, g_2, \ldots, g_n$ when encoded with the ideal binary arithmetic P coder $\tilde{B}$. The average code length per symbol (6.4) for encoding $\{G_i\}$ with P coder $\tilde{B}$ is given as

$$\begin{aligned}\bar{L}(\{G_i\}, \tilde{B}) = &\\ -\sum_{k=0}^{62} \lim_{n\to\infty} \frac{1}{n} &\sum_{(g_1,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1, \ldots, g_n) M(\zeta(\{G_i\}, (g_1, \ldots, g_n), \omega_k)) \log_2 p_{Y_k}(1)\\ -\sum_{k=0}^{62} \lim_{n\to\infty} \frac{1}{n} &\sum_{(g_1,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1, \ldots, g_n) N(\zeta(\{G_i\}, (g_1, \ldots, g_n), \omega_k)) \log_2 p_{Y_k}(0).\end{aligned} \tag{6.6}$$

Let $\mathcal{W}_M(\{G_i\}, \omega_k)$ be defined as

$$\mathcal{W}_M(\{G_i\}, \omega_k) = \lim_{n\to\infty} \frac{1}{n} \sum_{(g_1,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1, \ldots, g_n) M(\zeta(\{G_i\}, (g_1, \ldots, g_n), \omega_k)) \tag{6.7}$$

and let $\mathcal{W}_N(\{G_i\}, \omega_k)$ be defined as

$$\mathcal{W}_N(\{G_i\}, \omega_k) = \lim_{n\to\infty} \frac{1}{n} \sum_{(g_1,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1, \ldots, g_n) N(\zeta(\{G_i\}, (g_1, \ldots, g_n), \omega_k)). \tag{6.8}$$

(6.7) and (6.8) can be interpreted as the average relative frequency of ones and zeros respectively, that occur in coding bins of $\{G_i\}$ for that the conditional probability of one

equals ω_k. Substituting (6.7) and (6.8) in (6.6) yields

$$\bar{L}(\{G_i\},\tilde{B}) = -\sum_{k=0}^{62}\big(\mathcal{W}_M(\{G_i\},\omega_k)\log_2 p_{Y_k}(1) + \mathcal{W}_N(\{G_i\},\omega_k)\log_2 p_{Y_k}(0)\big). \tag{6.9}$$

Let

$$\begin{aligned}\mathcal{W}_\ell(\{G_i\},\omega_k) &= \mathcal{W}_M(\{G_i\},\omega_k) + \mathcal{W}_N(\{G_i\},\omega_k) &(6.10)\\ &= \lim_{n\to\infty}\frac{1}{n}\sum_{(g_1,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1,\ldots,g_n)\ell(\zeta(\{G_i\},(g_1,\ldots,g_n),\omega_k)) &(6.11)\end{aligned}$$

which can be interpreted as the average relative frequency of coding bins in $\{G_i\}$ for that the conditional probability of one equals ω_k. Using (6.10) to expand (6.9) yields

$$\begin{aligned}&\bar{L}(\{G_i\},\tilde{B})\\ &= -\sum_{k=0}^{62}\mathcal{W}_\ell(\{G_i\},\omega_k)\left(\frac{\mathcal{W}_M(\{G_i\},\omega_k)}{\mathcal{W}_\ell(\{G_i\},\omega_k)}\log_2 p_{Y_k}(1) + \frac{\mathcal{W}_N(\{G_i\},\omega_k)}{\mathcal{W}_\ell(\{G_i\},\omega_k)}\log_2 p_{Y_k}(0)\right)\end{aligned} \tag{6.12}$$

Let $\mathrm{RV}(x)$ be a binary random variable with pmf $p_{\mathrm{RV}(x)}(1) = x$ and let

$$J(\{G_i\},\omega_k) = \mathrm{RV}\left(\frac{\mathcal{W}_M(\{G_i\},\omega_k)}{\mathcal{W}_\ell(\{G_i\},\omega_k)}\right) \tag{6.13}$$

be a binary random variable so that (6.12) can be written as a weighted cross entropy

$$\begin{aligned}\bar{L}(\{G_i\},\tilde{B}) &= -\sum_{k=0}^{62}\mathcal{W}_\ell(\{G_i\},\omega_k)\big(p_{J(\{G_i\},\omega_k)}(1)\log_2 p_{Y_k}(1) + p_{J(\{G_i\},\omega_k)}(0)\log_2 p_{Y_k}(0)\big)\\ &= \sum_{k=0}^{62}\mathcal{W}_\ell(\{G_i\},\omega_k)H(J(\{G_i\},\omega_k);Y_k).\end{aligned} \tag{6.14}$$

Note that $H(X;Y)$ shall denote the cross entropy of a random variable Y with respect to a random variable X. When, for a particular ω_k, (6.7) and (6.8) are nonzero, the question arises whether

$$\frac{\mathcal{W}_M(\{G_i\},\omega_k)}{\mathcal{W}_\ell(\{G_i\},\omega_k)} = \omega_k \tag{6.15}$$

holds. It may be a reasonable assumption because of the following argumentation. A coding bin with conditional probability of one equal to ω_k occurs with relative frequency $\mathcal{W}_\ell(\{G_i\},\omega_k)$ in $\{G_i\}$. Since its probability of one equals ω_k, a coding bin of value one and with conditional probability of one equal to ω_k occurs with relative frequency of

$\omega_k \cdot \mathcal{W}_\ell(\{G_i\}, \omega_k)$, which equals $\mathcal{W}_M(\{G_i\}, \omega_k)$. Therefore, it shall be assumed, that (6.15) holds. Consequently, (6.14) can be written as

$$\bar{L}(\{G_i\}, \tilde{B}) = \sum_{k=0}^{62} \mathcal{W}_\ell(\{G_i\}, \omega_k) H(\Omega_k; Y_k). \tag{6.16}$$

A function $Q(\cdot)$ shall be defined that derives a random variable $Q(\{G_i\})$ over Ω with pmf

$$p_{Q(\{G_i\})}(\omega_k) = \mathcal{W}_\ell(\{G_i\}, \omega_k) \tag{6.17}$$

from the random process $\{G_i\}$. Note that

$$\begin{aligned}
&\sum_{k=0}^{62} p_{Q(\{G_i\})}(\omega_k) = \sum_{k=0}^{62} \lim_{n\to\infty} \frac{1}{n} \sum_{(g_1,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1,\ldots,g_n)\ell(\zeta(\{G_i\},(g_1,\ldots,g_n),\omega_k)) \\
&= \lim_{n\to\infty} \frac{1}{n} \sum_{(g_1,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1,\ldots,g_n) \sum_{k=0}^{62} \ell(\zeta(\{G_i\},(g_1,\ldots,g_n),\omega_k)) \\
&= \lim_{n\to\infty} \frac{1}{n} \sum_{(g_1,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1,\ldots,g_n) n = 1
\end{aligned} \tag{6.18}$$

and therefore $p_{Q(\{G_i\})}$ can be seen as pmf of a random variable. $Q(\{G_i\})$ can be interpreted as the distribution of the conditional probabilities of one of $\{G_i\}$. Substituting (6.17) in (6.16) yields

$$\bar{L}(\{G_i\}, \tilde{B}) = \bar{\mathcal{L}}_{\tilde{B}}(Q(\{G_i\})) = \sum_{k=0}^{62} p_{Q(\{G_i\})}(\omega_k) H(\Omega_k; Y_k). \tag{6.19}$$

Interestingly, the average code length per symbol of random process $\{G_i\}$ for P coder $\tilde{B}$ can be expressed as function of random variable $Q(\{G_i\})$ and random variables Y_k. When $Y_k = \Omega_k$, (6.19) becomes

$$\bar{H}(\{G_i\}) = \bar{\mathcal{H}}(Q(\{G_i\})) = \sum_{k=0}^{62} p_{Q(\{G_i\})}(\omega_k) H(\Omega_k), \tag{6.20}$$

which is the entropy rate (2.6) of $\{G_i\}$. Note, that (6.19) only holds when ideal binary arithmetic bin coders are used. When V2V codes shall be employed as bin coders, it turns out that the situation is more complicated. This is because the encoded length of a sequence $g_1, g_2, \ldots, g_n$ that is encoded with a V2V code cannot be expressed as a function of the number of ones and zeros in the sequence because the order of ones and zeros has an influence on the resulting encoded length. However, notice that the cross entropy in (6.19) is also the average code length

$$\bar{\ell}_{\hat{B}(Y_k)}(\Omega_k) = H(\Omega_k; Y_k) \tag{6.21}$$

of an ideal binary arithmetic coder. Consequently, the average code length per symbol can be seen as weighted average code lengths per symbol of the ideal binary arithmetic coders.

The bin sequences that occur for a particular ω_k in practical applications are very likely not distinguishable from a realization of a binary i.i.d. source. Therefore, a reasonable approximation of (6.19) for bin coders of arbitrary kind (e.g. ideal binary arithmetic bin coders or V2V codes) may be obtained by replacing the cross entropy with the average code length $\bar{\ell}_{B_k}(\Omega_k)$ of bin codes B_k for Ω_k. I.e., for a P coder B' with bin coders B_k, the approximation of (6.19) is given as

$$\bar{L}(\{G_i\}, B') = \bar{\mathcal{L}}_{B'}(Q(\{G_i\})) = \sum_{k=0}^{62} p_{Q(\{G_i\})}(\omega_k)\bar{\ell}_{B_k}(\Omega_k). \tag{6.22}$$

For applications like H.264/AVC or H.265/HEVC $Q(\{G_i\})$ can usually not exactly be derived, since it reflects all details of binarization, context modeling and probability estimation. As an alternative, an estimated version of $Q(\{G_i\})$ shall be derived, based on selected realizations of $\{G_i\}$. These realizations shall be extracted from a training set of bit streams that are encoded with H.265/HEVC. By properly selecting this training set, the resulting P coder can be tuned to particular properties of the bit streams to be encoded. These properties could be resolution, picture quality, prediction structures, etc., insofar as they produce different characteristics in the distribution of the conditional pmfs.

A typical encoder employs rate-distortion optimization and needs to know the expected encoded lengths of coding bins. The exact code lengths depend on the binary coding engine employed. However, an encoder usually calculates the code lengths directly from the estimated conditional pmfs of the coding bins and therefore they don't depend on the binary coding engine. This is a reasonable strategy since the divergence of the exact code lengths from the ones derived from the conditional pmfs is usually very low. Therefore, the P coder design can be carried out based on bit streams that are derived using a conventional H.265/HEVC encoder.

The design strategy shall be as follows. From the training set of bit streams, realizations of $\{G_i\}$ are extracted. E.g., one realization per slice or one realization per bit stream. In the simplest case, only one realization $g_1, g_2, \ldots, g_n$ is used to calculate an estimate for $Q(\{G_i\})$ that shall be denoted $Q(\{G_i\}, (g_1, g_2, \ldots, g_n))$. This corresponds to the relative frequencies of ω_k in all conditional pmfs of the realization, i.e.,

$$p_{Q(\{G_i\},(g_1,g_2,\ldots,g_n))}(\omega_k) = \frac{\ell(\zeta(\{G_i\}, (g_1, g_2, \ldots, g_n), \omega_k))}{n}. \tag{6.23}$$

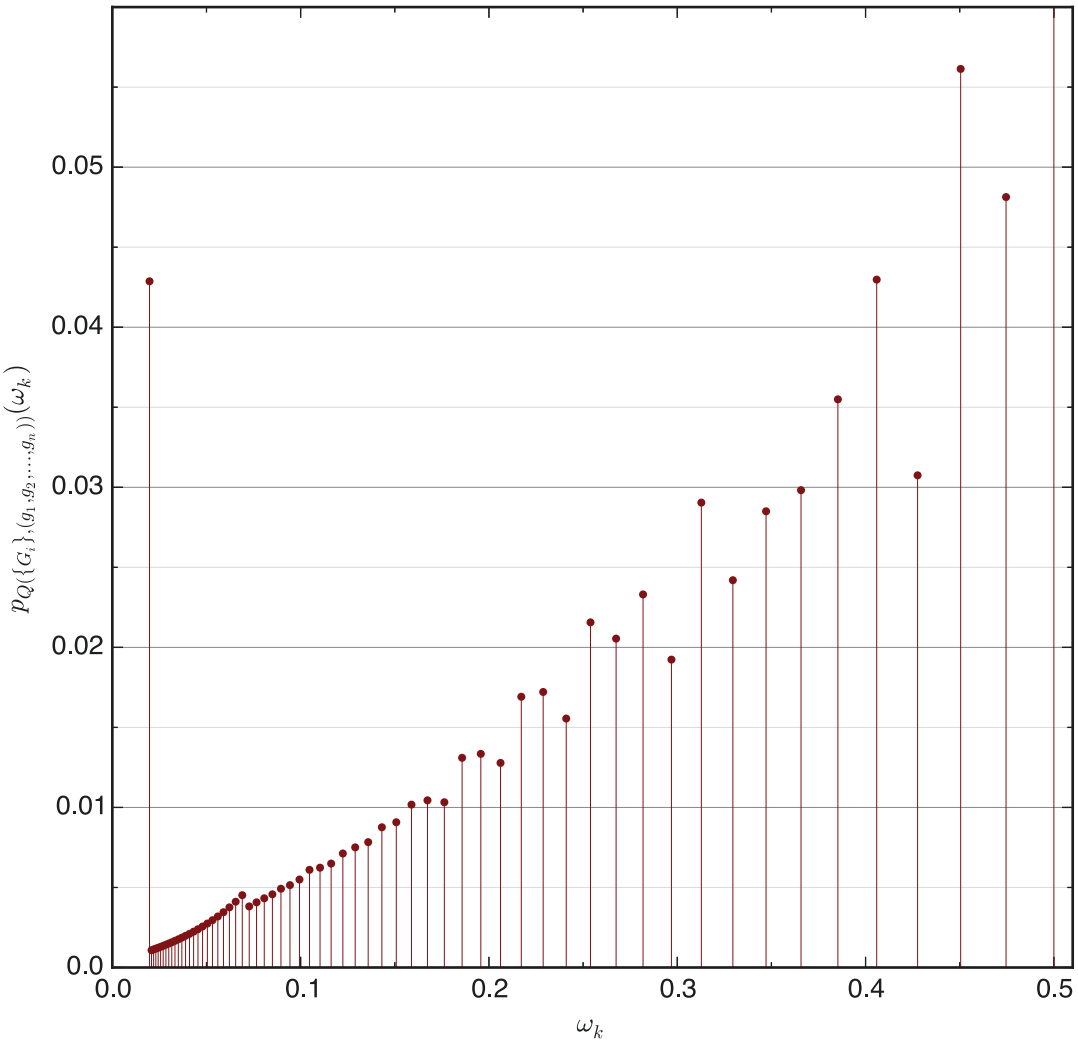

Figure 6.1: Exemplary estimated pmf, derived from test sequence 'Traffic' at QP 22 with hierarchical B frames (with $p_{Q(\{G_i\},(g_1,g_2,\ldots,g_n))}(\omega_0) \approx 0.283$).

An example for an estimated pmf according to (6.23) is depicted in Fig. 6.1. Note that $p_{Q(\{G_i\},(g_1,g_2,\ldots,g_n))}(0.5) \approx 0.283$ lies outside of the chart because of its large value. This is caused by the so-called bypass bins, which are by the design of binarization and context modeling of H.265/HEVC very frequent. Analogously, for the smallest occurring value $\omega_{62} \approx 0.02$, the probability $p_{Q(\{G_i\},(g_1,g_2,\ldots,g_n))}(\omega_{62}) \approx 0.043$ is also relatively large, which can be interpreted as a clipping effect in the probability estimation [5].

Combining (6.17), (6.23), and (6.11) yields

$$p_{Q(\{G_i\})}(\omega_k) = \lim_{n\to\infty} \sum_{(g_1,g_2,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1,g_2,\ldots,g_n) p_{Q(\{G_i\},(g_1,g_2,\ldots,g_n))}(\omega_k), \quad (6.24)$$

which reveals that $p_{Q(\{G_i\})}$ is a weighted sum of $p_{Q(\{G_i\},(g_1,g_2,\ldots,g_n))}$ of all realizations in $\mathcal{B}^n$. To derive an approximation for $p_{Q(\{G_i\})}$, only the realizations that are extracted from the training set shall be used. Furthermore, a weight shall also be specified for each realization. Assume that n different realizations for $\{G_i\}$ are extracted from the training set. Let p_{Q_j} be a pmf according to (6.23) derived from realization j with $j = 1,\ldots,n$ and let τ_j be the weight associated with it so that

$$\sum_{j=1}^{n} \tau_j = 1. \quad (6.25)$$

The resulting estimated pmf shall be given as

$$p_{\bar{Q}}(\omega_k) = \sum_{j=1}^{n} \tau_j p_{Q_j}(\omega_k) \quad (6.26)$$

and it shall be denoted *weighted pmf*. The average code length per symbol (6.22) for a P coder B' with bin coders B_k becomes

$$\bar{\mathcal{L}}_{B'}(\bar{Q}) = \sum_{k=0}^{62} p_{\bar{Q}}(\omega_k) \bar{\ell}_{B_k}(\Omega_k). \quad (6.27)$$

Note that (6.22) and (6.27) are based on the assumption that for each ω_k with $k = 0,1,\ldots,62$, a separate PIPE code B_k is used. However, in an actual P coder, several consecutive k are usually associated with one bin coder. I.e., such a bin coder receives a sequence of coding bins that have varying conditional pmfs. Depending on the occurrence order of the different conditional pmfs, one or another conditional pmf may dominate for different sections of a sequence of coding bins. To simplify matters, it shall be assumed that (6.27) also holds when one bin coder is associated with several ω_k. In particular, when a bin coder is associated with a narrow probability interval, the assumption should not lead to a noticeable error.

In the following, an exemplary weighted pmf shall be described and used for the design of a P coder for H.265/HEVC. The training set consists of bit streams encoded with the H.265/HEVC reference software HM 16.2 [78] under the common test conditions as specified in [79]. They specify

- 24 different test sequences, which are categorized in 6 classes A-F characterizing different resolutions and content types

- 8 different test conditions based on ‘Intra’, ‘Random access’, and ‘Low delay’ prediction structures

- 4 different quality settings corresponding to quantization parameter (QP) values $22, 27, 32,$ and 37

From the $24 \cdot 8 \cdot 4 = 768$ possible combinations 680 are specified as test bit streams. One realization $\{g_i\}$ is extracted for each bit stream by concatenating the sequences of coding bins of all frames in the bit stream. For each of these realization, a pmf p_{Q_j} is derived and an associated weight τ_j is assigned to it. If weighted equally, the derived pmfs p_{Q_j} have equal influence regardless of the resolution, frame rate, bit rate and sequence length of the underlying bit stream. I.e., longer realizations $\{g_i\}$ have the same influence as shorter ones. If it is desired to have a larger influence for particular aspects like e.g. resolution or QP, this can be achieved by adjusting the weights. In the following, an exemplary set of weights shall be derived where the influence of the various aspects reflected in the training set are balanced. Let a *subset weight* be defined as the sum of the weights that are associated with bit streams in a subset of the training set. The weights are derived as follows:

1. Six first level subsets are formed by distinguishing the classes A-D. Each of these subsets shall have equal subset weight.

2. Each first level subset is subdivided into two or three[1] second-level subsets by distinguishing the test conditions ‘Intra’, ‘Random access’, and ‘Low delay’. The weight of a first level subset is evenly distributed to its second-level subsets.

3. The weight of a second level subset is evenly distributed to the bit streams in the subset.

The weights produced by this procedure are given in Tab. 6.1 and the resulting weighted pmf $p_{\bar{Q}}(\omega_k)$ is depicted in Fig. 6.2. It shall be the basis for designing a P coder by minimizing (6.27). Although the optimization is based on average code length per symbol, the redundancy is a useful quantity to select a candidate set of V2V codes because it reveals how close a code operates at the entropy. The average code length can easily be derived by adding the entropy according to (3.15).

[1] For class A, no ‘Low delay’ bit streams, and for class E, no ‘Random access’ bit streams exist.

Class	Class weight	Test condition	Test condition weight	Number of sequences	Sequence Weight
A	$1/6$	Intra	$1/12$	32	$1/384$
		Random access	$1/12$	32	$1/384$
B	$1/6$	Intra	$1/18$	40	$1/720$
		Random access	$1/18$	40	$1/720$
		Low delay	$1/18$	80	$1/1440$
C	$1/6$	Intra	$1/18$	32	$1/576$
		Random access	$1/18$	32	$1/576$
		Low delay	$1/18$	64	$1/1152$
D	$1/6$	Intra	$1/18$	32	$1/576$
		Random access	$1/18$	32	$1/576$
		Low delay	$1/18$	64	$1/1152$
E	$1/6$	Intra	$1/12$	24	$1/288$
		Low delay	$1/12$	48	$1/576$
F	$1/6$	Intra	$1/18$	32	$1/576$
		Random access	$1/18$	32	$1/576$
		Low delay	$1/18$	64	$1/1152$

Table 6.1: Exemplary weight configuration for the training set according to the common test conditions [79].

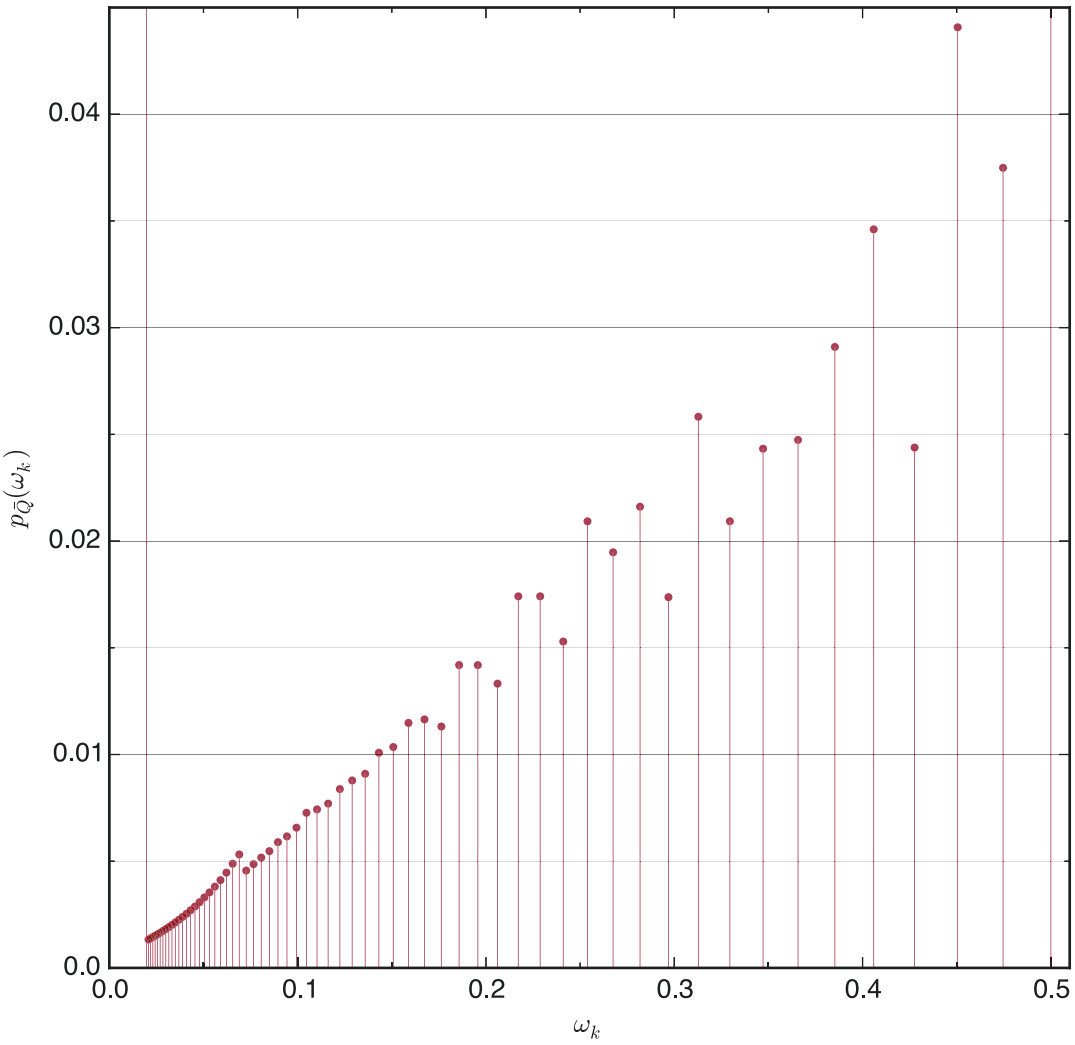

Figure 6.2: Weighted pmf according to Tab. 6.1 with $p_{\bar{Q}}(\omega_0) \approx 0.302$ and $p_{\bar{Q}}(\omega_{62}) \approx 0.055$.

6.1.1 V2V code selection strategies

In this section, strategies for selecting V2V codes for a P coder are discussed. Let $\mathcal{V}_C = \{v_1, v_2, \ldots, v_n\}$ be the initial candidate set of n V2V codes from which m codes shall be selected for a P coder. I.e., a P coder is unambiguously associated with a subset of $\mathcal{V}_C$. The $\binom{n}{m}$ possible different subsets of $\mathcal{V}_C$ that have cardinality m shall be denoted $\mathcal{V}_m^i$ with $i = 1, 2, \ldots, \binom{n}{m}$. Note that given $\mathcal{V}_m^i$, for each $\omega_k \in \Omega$, we can choose a V2V code $v \in \mathcal{V}_m^i$ that has minimal average code length for ω_k. The resulting P coder corresponding to subset $\mathcal{V}_m^i$ shall be denoted P_m^i. Its associated average code length $\bar{\ell}_{P_m^i}(X)$ equals the envelope of the average code lengths of the involved V2V codes. The optimal P coder $P_m^{i_{opt}}$ with m bin coders is found by minimizing (6.27) over all $\mathcal{V}_m^i$ with $i = 1, 2, \ldots, \binom{n}{m}$ and for a given weighted pmf $p_{\bar{Q}}(\omega_k)$. This corresponds to an exhaustive search and depending on m and n, the procedure may have a high computational complexity. For m close to n or m close to 0, the number of P coders to test is low and can usually be carried out. For m close to $\frac{n}{2}$, the number of combinations has its maximum and the above minimization is much more complex.

6.1.1.1 Proposed algorithm

A less complex, but suboptimal alternative to the exhaustive minimization approach of the previous section is as follows. For each k with $0 \leq k < 63$, the V2V code with the lowest average code length for ω_k is selected out of the candidate set of V2V codes $\mathcal{V}_C$. This yields a number of V2V codes which already could be used as P coder. However, it will likely have an undesirably large number of V2V codes. Note that this step doesn't require the weighted pmf $p_{\bar{Q}}(\omega_k)$. The idea behind the algorithm is to successively remove individual V2V codes until the desired number of V2V codes is reached. In each removal step, the increase of the average code length per symbol (6.27) shall be minimized. This is done as follows:

1. Select for each k with $0 \leq k < 63$ the V2V code out of $\mathcal{V}_C$ that minimizes the average code length for Ω_k. These V2V codes form the initial set of V2V codes $\hat{\mathcal{V}}_m$ with m elements (which corresponds to a P coder with m bin coders).

2. Derive $\hat{\mathcal{V}}_{m-1}$ by removing from $\hat{\mathcal{V}}_m$ the V2V code that causes the smallest increase of (6.27) for the weighted pmf $p_{\bar{Q}}(\omega_k)$ when removed.

3. Continue with step 2 with m reduced by 1 until the desired number of V2V codes is achieved.

The algorithm is referred to as *successive removal algorithm*.

6.1.2 Evaluation

To compare the exhaustive search to the successive removal algorithm, several exemplary P coders are derived and evaluated. The 24 optimal SCV2V codes with maximum source and code tree height of 6 (see Sec. 5.4.4.3) and the unary-to-rice codes of degree $3 \leq k \leq 5$ (see Sec. 5.5.2) shall be used as candidate set of V2V codes. It is referred to as *S6C6+UR*. The SCV2V codes are used because they are a good trade-off between size and redundancy and the unary-to-rice codes provide a low redundancy for particularly small ω_k. As can be seen in Fig. 5.10, the probability where a unary-to-rice-code has minimum redundancy decreases with increasing degree k (except for $k = 0$). Furthermore, when comparing Fig. 5.10 with Fig. 6.2, it can be seen that the unary-to-rice code of degree $k = 5$ has the lowest redundancy for the smallest occurring probability $\omega_{62} \approx 0.2$ so that codes of larger degree can be disregarded in advance. Similarly, codes of degree $k < 3$ can also be neglected since they clearly have a larger redundancy than the SCV2V codes of maximum source and code tree height of 6 (cf. Fig. 5.7 and Fig. 5.10).

Applying step 1 of the successive removal algorithm yields 20 V2V codes as depicted in Tab. 6.2 along with the associated probabilities. In order to evaluate the compression efficiency of a P coder for a given weighted pmf, the overhead of (6.22) over (6.20) is of interest. Let Q' be a random variable over $\mathbb{Q}$ like e.g. the distribution of conditional probabilities of one $Q(\{G_i\})$ of random process $\{G_i\}$. For Q', the overhead shall be defined as

$$\eta_P(Q') = 100\% \cdot \left(\frac{\bar{\mathcal{L}}_P(Q')}{\bar{\mathcal{H}}(Q')} - 1 \right) \tag{6.28}$$

and it describes the percentaged overhead of the average code length per symbol for a given P coder P (6.22) relative to the entropy rate (6.20). For the weighted pmf $\bar{Q}$ and for the codes listed in Tab. 6.2 it amounts to remarkably low 0.189%. The overhead of the P coders derived by the successive removal algorithm is depicted in Fig. 6.3 together with several P coders derived by exhaustive search. Additionally, one P coder that uses optimal binary arithmetic bin coders is also depicted, for which optimal probability intervals are derived with dynamic programming according to Greene et al. [19]. Note that the overhead for P coders with one, two, and three V2V codes are listed in Tab. 6.3 because they are outside of the chart. The associated codes are listed in Appendix B.2. Although the overhead increases with each step, this increase

UR_5	$\omega_{55} - \omega_{62}$
UR_4	$\omega_{42} - \omega_{54}$
UR_3	$\omega_{32} - \omega_{41}$
S6C6_1	$\omega_{30} - \omega_{31}$
S6C6_2	$\omega_{28} - \omega_{29}$
S6C6_3	$\omega_{24} - \omega_{27}$
S6C6_4	$\omega_{20} - \omega_{23}$
S6C6_6	$\omega_{18} - \omega_{19}$
S6C6_7	ω_{17}
S6C6_8	ω_{16}
S6C6_9	ω_{15}
S6C6_10	$\omega_{12} - \omega_{14}$
S6C6_13	$\omega_{9} - \omega_{11}$
S6C6_14	ω_{8}
S6C6_15	ω_{7}
S6C6_16	ω_{6}
S6C6_17	$\omega_{4} - \omega_{5}$
S6C6_20	ω_{3}
S6C6_22	ω_{2}
S6C6_24	$\omega_{0} - \omega_{1}$

Table 6.2: V2V codes after step 1 of the successive removal algorithm.

Number of V2V codes	1	2	3
S6C6+UR (successive removal)	25.79%	4.16%	2.50%
S6C6+UR	13.43%	3.67%	1.77%
Systematic V2V codes	15.12%	4.51%	1.96%
TMuC V2V codes	13.43%	3.83%	1.68%
Optimal arithmetic bin coders	13.80%	3.49%	1.42%

Table 6.3: Overhead values that are outside of the chart in Fig. 6.3.

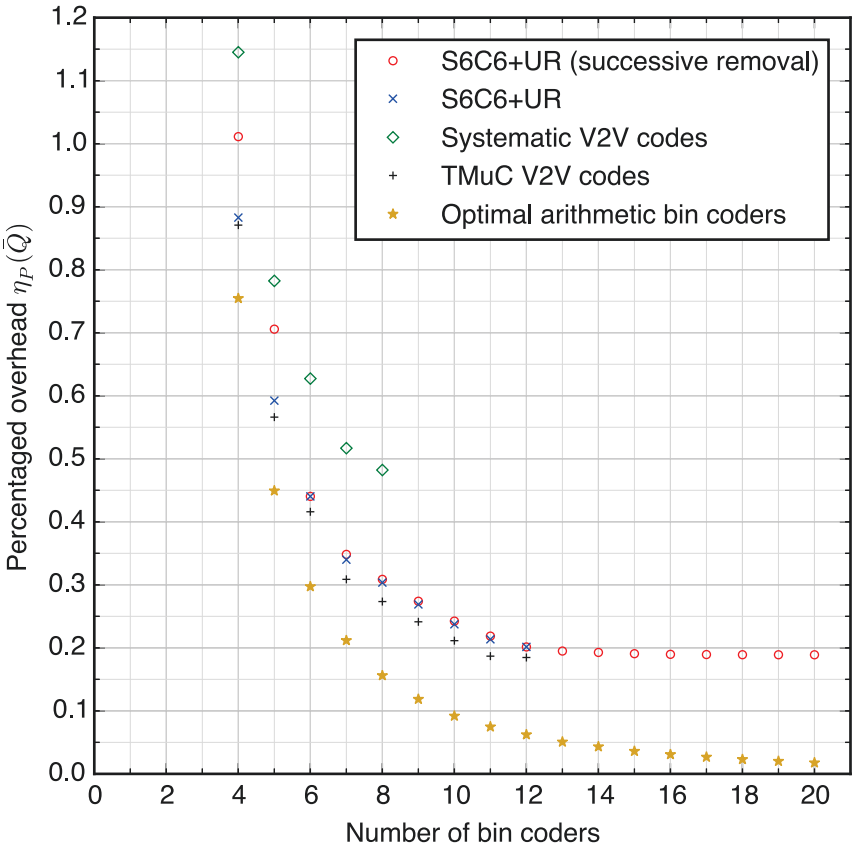

Figure 6.3: Percentaged overhead $\eta_P(\bar{Q})$ of the different P coders (P coders with one, two, and tree codes are outside of the chart and have an overhead as given in Tab. 6.3).

is negligible for P coders with 12 to 20 V2V codes. Consequently, P coders with more than 12 V2V codes are of limited interest for practical implementations.

Exhaustive search is applied to S6C6+UR codes, to the 12 TMuC V2V codes of Fig. 5.9, and to the 8 systematic V2V codes of Fig. 5.13. The maximum number of V2V codes per P coder is limited to 12 for the S6C6+UR codes since no noticable reduction of the redundancy can be expected as the results of the successive removal algorithm show. Codes derived by exhaustive search are listed in Appendix B.3.

When comparing the P coders derived from the S6C6+UR codes, the successive removal algorithm yields almost as good results as exhaustive search for P coders with 6 or more V2V codes. For P coders with less V2V codes, the exhaustive search yields much better results. When comparing the three different types of P coders derived by exhaustive search (S6C6+UR, systematic, and TMuC), the following can be observed:

With an increasing number of V2V codes of a P coder, the overhead shows an almost exponential decay. An overhead of less than 1% is rather easy to achieve with only 4 V2V codes. An increase to 8 V2V codes allows an overhead of approximately 0.3%. The addition of further V2V codes up to a total number of 12 reduces the overhead to approximately 0.2%. A further reduction cannot be achieved with S6C6+UR codes, which suggests that longer maximum source and code word lengths may be required. The TMuC-based P coders with 4 to 12 V2V codes show overhead values that are constantly lower by approximately 0.03% than the corresponding S6C6+UR-based P coders. This seems to be a relatively small advantage of the TMuC-based P coders over the S6C6+UR-based coders although the TMuC codes are allowed to have much longer source and code words. The P coders based on systematic V2V codes have a substantially larger overhead than the other types but still stay below 0.5% for 8 V2V codes.

6.1.3 Comparison to the M coder

The overhead of the various P coders derived in the previous sections shall be compared to the M coder as used in CABAC of H.264/AVC or H.265/HEVC. Let X be a binary random variable for which symbol 1 shall be the LPS with $p_{LPS} = p_X(1)$ and for which symbol 0 is the MPS with $p_{MPS} = p_X(0)$. Binary arithmetic coding can be considered redundancy-free[1] when the coding interval subdivision is carried out exactly in proportion to the probability values of the pmf $p_X(1)$ and $p_X(0)$ of the binary source X to be encoded. In practical implementations (like the M coder) the interval subdivision position differs slightly from the desired exact position so that the coder has zero redundancy for a slightly different pmf $p_{\tilde{X}}$ (according to the ratio of the actual subintervals). Consequently, the redundancy of encoding an individual symbol with a binary arithmetic coder equals the relative entropy $D(X\|\tilde{X})$. Note that the pmf of $\tilde{X}$ usually changes from symbol to symbol.

In the M coder, the multiplication, which is required for the interval subdivision, is replaced with a table-lookup. The resulting redundancy is studied in [80, Sec. 6.3] for various configurations. In the configuration used in H.264/AVC and H.265/HEVC, the coding interval width is represented as the *range* value r, which is an integer in the interval $[256, 510]$. For the interval subdivision, r is quantized to four cells with the cell index given as

$$r_{idx} = (r >> 6)\&3, \tag{6.29}$$

[1] Bits required to terminate an arithmetic code word are neglected.

which is an integer in the interval $[0, 3]$. The exact subinterval width $r_{LPS} = r \cdot p_X(1)$ of the LPS is approximated with a precalculated value $\tilde{r}_{LPS}$ that is read from a lookup table using r_{idx}. I.e., all values of r for which (6.29) yields the same index r_{idx} use the same value $\tilde{r}_{LPS}$. The pmf $p_{\tilde{X}}$ is then given as

$$p_{\tilde{X}}(1) = \frac{\tilde{r}_{LPS}}{r}. \tag{6.30}$$

The average code length of encoding a symbol from X is the cross entropy

$$\begin{aligned} \ell'_M(X, r) &= H(X; \tilde{X}) && (6.31)\\ &= D(X \| \tilde{X}) + H(X) \\ &= -p_X(1) \log_2 \frac{\tilde{r}_{LPS}}{r} - p_X(0) \log_2 \left(1 - \frac{\tilde{r}_{LPS}}{r}\right). && (6.32) \end{aligned}$$

After encoding a symbol, one of the two subintervals becomes the coding interval, potentially also being renormalized (i.e. multiplied with a power of 2) to ensure that r remains an integer in the interval $[256, 510]$. Consequently, the value of r fluctuates in the interval $[256, 510]$.

Let r_i be the range value before encoding coding bin g_i and let $\mathrm{Rn}(\cdot)$ be a function that unambiguously derives $r_i = \mathrm{Rn}(\mathbf{g}_{i-1})$ from the sequence of previous coding bins $\mathbf{g}_{i-1} = g_{i-1}, g_{i-2}, \ldots, g_1$. As in H.264/AVC and H.265/HEVC, r_1 shall be 510. Furthermore, let $\vartheta(\{G_i\}, (g_1, g_2, \ldots, g_n), \omega_k, r)$ be the subsequence of coding bins in the sequence $g_1, g_2, \ldots, g_n$ for that both of the following conditions are fulfilled:

1. The conditional probability of one $p_{G_i}(1|\mathbf{g}_{i-1})$ for coding bin g_i equals ω_k
2. The range value $r_i = \mathrm{Rn}(\mathbf{g}_{i-1})$ as present before encoding g_i equals r

Next, the average code length per symbol (2.7) of the M coder shall be studied. Let $L_M(r, \omega_k)$ represents the lookup table of the M coder that stores the precalculated LPS range values for probability ω_k and range value r as used in H.264/AVC and H.265/HEVC. For $n \to \infty$, the encoded length that the M coder produces for a sequence of coding bins $g_1, \ldots, g_n$ can be expressed as

$$\begin{aligned} \ell_M(\{G_i\}, (g_1, \ldots, g_n)) = -\sum_{r=256}^{510} \sum_{k=0}^{62} \Bigg(& M(\vartheta(\{G_i\}, (g_1, \ldots, g_n), \omega_k, r)) \log_2 \frac{L_M(r, \omega_k)}{r} \\ & + N(\vartheta(\{G_i\}, (g_1, \ldots, g_n), \omega_k, r)) \log_2 \frac{r - L_M(r, \omega_k)}{r} \Bigg). \end{aligned} \tag{6.33}$$

Substituting (6.33) in (2.7) yields

$$\bar{L}(\{G_i\}, M) = -\sum_{r=256}^{510}\sum_{k=0}^{62}$$
$$\left(\lim_{n\to\infty}\sum_{(g_1,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1,\ldots,g_n)\frac{M(\vartheta(\{G_i\},(g_1,\ldots,g_n),\omega_k,r))}{n}\log_2\frac{L_M(r,\omega_k)}{r}+\right.$$
$$\left.\lim_{n\to\infty}\sum_{(g_1,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1,\ldots,g_n)\frac{N(\vartheta(\{G_i\},(g_1,\ldots,g_n),\omega_k,r))}{n}\log_2\frac{r-L_M(r,\omega_k)}{r}\right), \tag{6.34}$$

which is the average code length per symbol that the M coder produces when encoding $\{G_i\}$. Note that the limit does not need to exist. Let $\mathcal{U}_M(\{G_i\},\omega_k,r)$ be defined as

$$\mathcal{U}_M(\{G_i\},\omega_k,r) = \lim_{n\to\infty}\sum_{(g_1,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1,\ldots,g_n)\frac{M(\vartheta(\{G_i\},(g_1,\ldots,g_n),\omega_k,r))}{n} \tag{6.35}$$

and let $\mathcal{U}_N(\{G_i\},\omega_k,r)$ be defined as

$$\mathcal{U}_N(\{G_i\},\omega_k,r) = \lim_{n\to\infty}\sum_{(g_1,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1,\ldots,g_n)\frac{N(\vartheta(\{G_i\},(g_1,\ldots,g_n),\omega_k,r))}{n}. \tag{6.36}$$

(6.35) and (6.36) can be interpreted as the average relative frequency of ones and zeros respectively, that occur in coding bins of $\{G_i\}$ for that the conditional probability of one equals ω_k and for that the range value before encoding equals r. Let $\mathcal{U}_\ell(\{G_i\},\omega_k,r)$ be defined as the sum

$$\begin{aligned}\mathcal{U}_\ell(\{G_i\},\omega_k,r) =& \mathcal{U}_M(\{G_i\},\omega_k,r) + \mathcal{U}_N(\{G_i\},\omega_k,r)\\ =& \lim_{n\to\infty}\sum_{(g_1,\cdots,g_n)\in\mathcal{B}^n} p_{\{G_i\}}(g_1,\ldots,g_n)\frac{\ell(\vartheta(\{G_i\},(g_1,\ldots,g_n),\omega_k,r))}{n}.\end{aligned} \tag{6.37}$$

Substituting (6.36) and (6.35) in (6.34) and expanding it with (6.37) yields

$$\begin{aligned}&\bar{L}(\{G_i\}, M)\\ &= -\sum_{r=256}^{510}\sum_{k=0}^{62}\left(\mathcal{U}_M(\{G_i\},\omega_k,r)\log_2\frac{L_M(r,\omega_k)}{r} + \mathcal{U}_N(\{G_i\},\omega_k,r)\log_2\frac{r-L_M(r,\omega_k)}{r}\right)\\ &= -\sum_{r=256}^{510}\sum_{k=0}^{62}\mathcal{U}_\ell(\{G_i\},\omega_k,r)\cdot\\ &\left(\frac{\mathcal{U}_M(\{G_i\},\omega_k,r)}{\mathcal{U}_\ell(\{G_i\},\omega_k,r)}\log_2\frac{L_M(r,\omega_k)}{r} + \frac{\mathcal{U}_N(\{G_i\},\omega_k,r)}{\mathcal{U}_\ell(\{G_i\},\omega_k,r)}\log_2\frac{r-L_M(r,\omega_k)}{r}\right).\end{aligned} \tag{6.38}$$

Following the same argumentation as for (6.15), it shall be assumed that

$$\frac{\mathcal{U}_M(\{G_i\},\omega_k,r)}{\mathcal{U}_\ell(\{G_i\},\omega_k,r)}=\omega_k \tag{6.39}$$

holds. Consequently, (6.38) can be rewritten as

$$\begin{aligned}&\bar{L}(\{G_i\},M)\\&=-\sum_{r=256}^{510}\sum_{k=0}^{62}\mathcal{U}_\ell(\{G_i\},\omega_k,r)\left(\omega_k\log_2\frac{L_M(r,\omega_k)}{r}+(1-\omega_k)\log_2\frac{r-L_M(r,\omega_k)}{r}\right)\\&=\sum_{r=256}^{510}\sum_{k=0}^{62}\mathcal{U}_\ell(\{G_i\},\omega_k,r)H\left(\Omega_k;\mathrm{RV}\left(\frac{L_M(r,\omega_k)}{r}\right)\right),\end{aligned} \tag{6.40}$$

which reveals that the average code length per symbol of the M coder can be written as a sum of weighted cross entropies.

A function $\mathrm{Rg}(\cdot)$ shall be defined that yields a random variable $\mathrm{Rg}(\{G_i\})$ over $\mathcal{I}=\{256,257,\ldots,510\}$. In a similar way as $Q(\{G_i\})$ for the conditional probabilities of one, random variable $\mathrm{Rg}(\{G_i\})$ shall describe the *distribution of range values* that occur for $\{G_i\}$ as follows. Random variable $Q(\{G_i\})$ as defined in (6.17) and random variable $\mathrm{Rg}(\{G_i\})$ shall have the joint pmf

$$p_{Q(\{G_i\}),\mathrm{Rg}(\{G_i\})}(\omega_k,r)=\mathcal{U}_\ell(\{G_i\},\omega_k,r). \tag{6.41}$$

When $\mathrm{Rg}(\{G_i\})$ is marginalized out, (6.41) equals the distribution of conditional probabilities of one (6.17) of $\{G_i\}$. By marginalizing out $Q(\{G_i\})$ in (6.41), the pmf

$$p_{\mathrm{Rg}(\{G_i\})}(r)=\lim_{n\to\infty}\frac{1}{n}\sum_{(g_1,\cdots,g_n)\in\mathcal{B}^n}p_{\{G_i\}}(g_1,\ldots,g_n)\sum_{k=0}^{62}\ell(\vartheta(\{G_i\},(g_1,\ldots,g_n),\omega_k,r)). \tag{6.42}$$

is derived. Note that

$$\begin{aligned}&\sum_{r=256}^{510}p_{\mathrm{Rg}(\{G_i\})}(r)\\&=\lim_{n\to\infty}\frac{1}{n}\sum_{(g_1,\cdots,g_n)\in\mathcal{B}^n}p_{\{G_i\}}(g_1,\ldots,g_n)\sum_{k=0}^{62}\sum_{r=256}^{510}\ell(\vartheta(\{G_i\},(g_1,\ldots,g_n),\omega_k,r))\\&=\lim_{n\to\infty}\frac{1}{n}\sum_{(g_1,\cdots,g_n)\in\mathcal{B}^n}p_{\{G_i\}}(g_1,\ldots,g_n)n=1\end{aligned} \tag{6.43}$$

and therefore $p_{\mathrm{Rg}(\{G_i\})}$ can be seen as pmf of a random variable. Substituting (6.41) in (6.40) yields

$$\bar{L}(\{G_i\}, M) = \sum_{r=256}^{510} \sum_{k=0}^{62} p_{Q(\{G_i\}),\mathrm{Rg}(\{G_i\})}(\omega_k, r) H\left(\Omega_k; \mathrm{RV}\left(\frac{L_M(r,\omega_k)}{r}\right)\right), \quad (6.44)$$

which reveals that the average code length per symbol of the M coder can be calculated through the joint pmf of $Q(\{G_i\})$ and $\mathrm{Rg}(\{G_i\})$. When $Q(\{G_i\})$ and $\mathrm{Rg}(\{G_i\})$ are statistically independent, (6.44) becomes

$$\begin{aligned}\bar{L}(\{G_i\}, M) &= \sum_{r=256}^{510} \sum_{k=0}^{62} p_{Q(\{G_i\})}(\omega_k) p_{\mathrm{Rg}(\{G_i\})}(r) H\left(\Omega_k; \mathrm{RV}\left(\frac{L_M(r,\omega_k)}{r}\right)\right) \\ &= \sum_{r=256}^{510} p_{\mathrm{Rg}(\{G_i\})}(r) \sum_{k=0}^{62} p_{Q(\{G_i\})}(\omega_k) H\left(\Omega_k; \mathrm{RV}\left(\frac{L_M(r,\omega_k)}{r}\right)\right). \quad (6.45)\end{aligned}$$

Due to the fluctuation of the range values r, it may be justified to assume statistical independence. However, this depends on the coding application.

Next, the fluctuation of the range values shall be studied for independent $Q(\{G_i\})$ and $\mathrm{Rg}(\{G_i\})$. For this purpose, the coding bins shall be assumed to come from an artificial source as follows. Assume a random variable Q' over $\mathfrak{Q}$ like e.g. the distribution of conditional probabilities of one $Q(\{G_i\})$ of random process $\{G_i\}$. Let $K(Q')$ be a random variable over

$$\mathfrak{Q} \times \mathfrak{B} = \{(\omega_0, 0), (\omega_0, 1), (\omega_1, 0), (\omega_1, 1), (\omega_2, 0), (\omega_2, 1), \ldots\} \quad (6.46)$$

with pmf

$$p_{K(Q')}((\omega_k, g)) = \begin{cases} p_{Q'}(\omega_k)\omega_k & \text{if } g = 1 \\ p_{Q'}(\omega_k)(1-\omega_k) & \text{otherwise.} \end{cases} \quad (6.47)$$

To use $K(Q')$ for evaluating a binary coding engine (like the M coder), a pair (ω_k, g) is drawn from $K(Q')$. Then, ω_k is used as conditional pmf and g is used as coding bin to be encoded. This is equivalent to first drawing a pmf from Q' and then using it to draw a coding bin. The average code length of encoding a bin from $K(Q')$ with a binary coding engine B is given as

$$\bar{L}(K(Q'), B) = \sum_{k=0}^{62} p_{Q'}(\omega_k) \bar{\ell}_B(\Omega_k). \quad (6.48)$$

(6.48) equals the average code length per symbol (6.22) of the random process from that Q' is derived. I.e., the conditional pmfs occur in the same proportion as in a random process from which Q' is derived while no dependencies exist between consecutive coding bins. This is a useful property because coding bins from $K(Q(\{G_i\}))$ can

be encoded instead of encoding random process $\{G_i\}$ when dependencies between coding bins are not relevant or shall not be present. Apart from this, the statistical properties of $K(Q')$ are rather simple since a sequence of ω_k and a sequence of b extracted from a sequence of pairs drawn from $K(Q')$ both behave like realizations of an i.i.d. process.

$K(Q(\{G_i\}))$ shall be used for analyzing the fluctuation of the range values r of the M coder since no dependencies exist between consecutive coding bins that could *disturb* the fluctuation. It can be observed that the values that occur for r when encoding source $K(Q(\{G_i\}))$ with the M coder can be modeled as a Markov chain where the allowed values for r are the states. Each element in $\mathcal{Q} \times \mathcal{B}$ corresponds to a state transition. I.e., for a given starting state r, an unambiguous target state can be derived for each element in $\mathcal{Q} \times \mathcal{B}$. An intermediate state $T'(r, (\omega_k, g))$ for starting state r and $(\omega_k, g) \in \mathcal{Q} \times \mathcal{B}$ shall be defined as

$$T'(r, (\omega_k, g)) = \begin{cases} L_M(r, \omega_k) & \text{if } g = 1 \\ r - L_M(r, \omega_k) & \text{otherwise.} \end{cases} \tag{6.49}$$

$T'(r, (\omega_k, g))$ is the range value after encoding (ω_k, g) prior to renormalization. To derive the target state $T(r, (\omega_k, g))$, the intermediate state $T'(r, (\omega_k, g))$ doubled until it is in the interval $[256, 510]$. Note that several elements in $\mathcal{Q} \times \mathcal{B}$ may have the same target state so that they contribute to the same state transition. Based on the weighted pmf $\bar{Q}$, a distribution of the range values shall be derived by simulation from $K(\bar{Q})$. Since it is the equivalent to $\mathrm{Rg}(\{G_i\})$ for $\{G_i\}$, it shall be described as random variable over $\mathcal{I} = \{256, 257, \ldots, 510\}$, denoted $\mathrm{Rg}(K(\bar{Q}))$. Its pmf corresponds to the steady-state probabilities of the Markov chain. Fig. 6.4 shows such a pmf that is derived by simulation for the weighted average pmf $p_{\bar{Q}}$ as depicted in Fig. 6.2. A sequence of 10 million pairs is drawn from $K(\bar{Q})$ (using a pseudo-random number generator). The coding bins g are encoded with the M coder, using the ω_k as conditional pmfs. The occurring range values are tracked and a relative frequency is calculated for each possible value of r.

Note that the bypass bins are neglected since the M coder encodes them with zero redundancy and since they don't alter r. The pmf is decomposed into 9 lines[1] according to the multiplicity of prime factor 2 in r in order to try an explanation of the behavior of the pmf. It is interesting to see that odd values for r (line 0) almost always have a lower probability than all other values. Furthermore, with increasing

[1] The lines between the points shall only improve the readability.

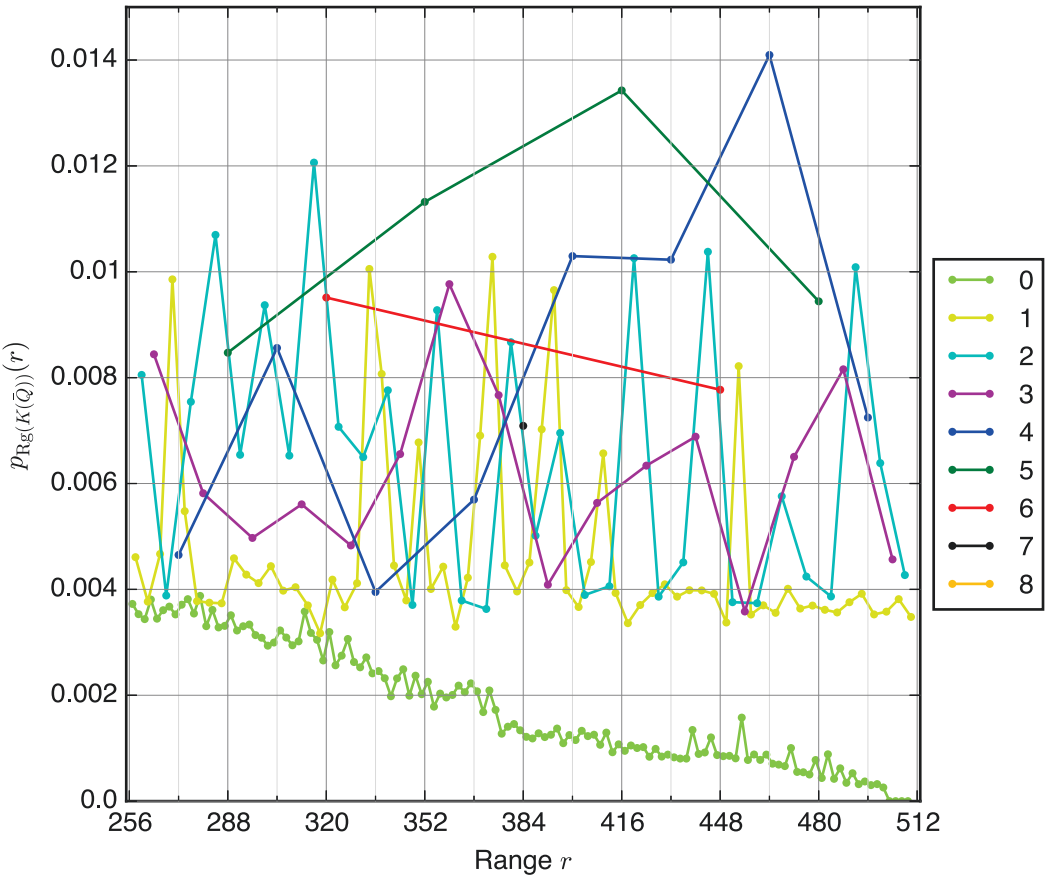

Figure 6.4: Pmf $p_{\mathrm{Rg}(K(\bar{Q}))}$ derived (by simulation) from the weighted pmf of Fig. 6.2 (excluding bypass bins), decomposed into 9 categories according to the power of prime factor 2 in r. Note that $p_{\mathrm{Rg}(K(\bar{Q}))}(256) = 0.0375$ is outside of the chart.

r, the probability of the odd values decreases. In order to explain this behavior, the question arises, how a particular value of r can occur. The two possiblities are either by interval subdivision or by renormalization. Since renormalization is a multiplication with a power of 2, odd values of r cannot occur after a renormalization. Instead, odd values are the result of a subdivision of an even larger value of r. The larger the odd value is, the less values of r are available that are even larger. Higher multiplicities of prime factor 2 in r, however, don't seem to reveal a clear behavior so that other effects may be relevant here. An exception is $p_{\mathrm{Rg}(K(\bar{Q}))}(256) = 0.0375$, which has a much larger probability than all other values of r.

The average code length of encoding $K(Q')$ with the M coder can be calculated from Q' as follows. Using (6.31), the average code length of the coding bins for that

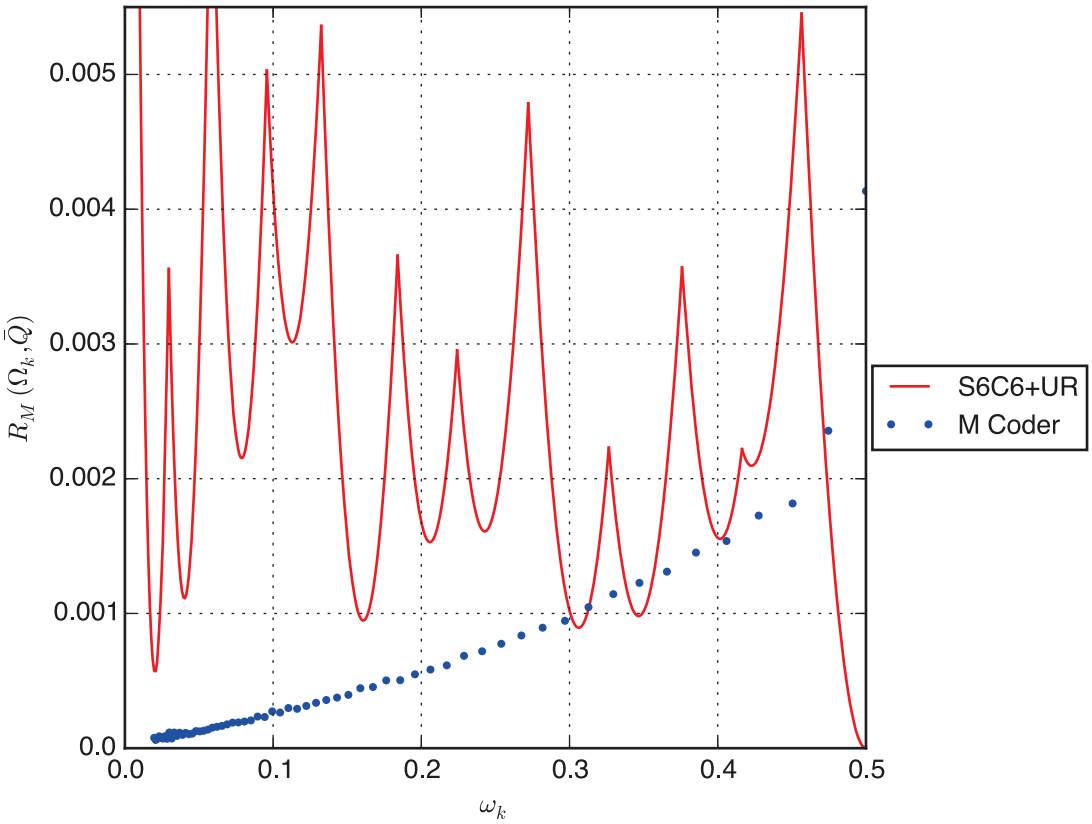

Figure 6.5: Redundancy $R_M(\Omega_k, \bar{Q})$ of the M coder for which the range values are distributed according to pmf $p_{\mathrm{Rg}(\bar{Q})}$ of Fig. 6.4 in comparison to 12 S6C6+UR V2V codes selected by exhaustive search (see Fig. 6.3).

a particular ω_k occurs amongst all coding bins of $K(Q')$ is given as weighted cross entropy

$$\bar{\ell}_M(\Omega_k, Q') = \sum_{r \in \mathcal{I}} p_{\mathrm{Rg}(K(Q'))}(r) H\left(\Omega_k; \mathrm{RV}\left(\frac{L_M(r, p_{\Omega_k}(1))}{r}\right)\right). \tag{6.50}$$

Note that $\bar{\ell}_M(\Omega_k, Q')$ is not the average code length for encoding a sequence of symbols from random variable Ω_k since $p_{\mathrm{Rg}(K(Q'))}$ is only valid for source $K(Q')$. From (6.50), a redundancy can be derived as

$$R_M(\Omega_k, Q') = \bar{\ell}_M(\Omega_k, Q') - H(\Omega_k). \tag{6.51}$$

Fig. 6.5 shows the redundancy of the M coder per state based on $\bar{Q}$. The range values are distributed according to Fig. 6.4, which is based on the weighted pmf of Fig. 6.2.

Additionally, the redundancy of a P coder that uses 12 bin coders from the S6C6+UR V2V codes is depicted. The V2V codes are selected by exhaustive search (see Fig. 6.3). It can be seen that the redundancy of the M coder increases with increasing probability ω_k. In comparison to the depicted P coder, the redundancy of the M coder is almost always lower, except for a few cases like for $\omega_0 = 0.5$. However, the distance between the redundancies of M and P coder is relatively small so that the P coder may be a suitable alternative to the M coder. The average code length per symbol (6.48) for encoding $K(Q')$ with the M coder is given as

$$\bar{L}(K(Q'), M) = \sum_{k=0}^{62} p_{Q'}(\omega_k)\bar{\ell}_M(\Omega_k, Q'). \tag{6.52}$$

Analogously to the overhead of the P coder, a percentaged overhead of the average code length per symbol produced by the M coder (6.52) relative to the optimal length (6.20) can be defined as

$$\eta_M(Q') = 100\% \cdot \left(\frac{\bar{L}(K(Q'), M)}{\bar{\mathcal{H}}(Q')} - 1 \right) \tag{6.53}$$

for a given distribution of conditional probabilities of one Q'. When using the weighted average pmf $p_{\bar{Q}}$ as depicted in Fig. 6.2, it is important to properly regard the bypass bins from $p_{\bar{Q}}(0.5)$ because they are encoded with zero redundancy in the M coder while the remaining bins for $p_{\bar{Q}}(0.5)$ are not. In the case of Fig. 6.2, the bypass bins occur with $p_{\bar{Q}}^{bypass}(0.5) = 0.267$ and the non-bypass bins occur with $p_{\bar{Q}}^{non-bypass}(0.5) = 0.035$. Counting the bypass bins with 1 bit per coding bin in (6.52), the overhead of the M coder amounts to 0.094% for the weighted average pmf of Fig. 6.2 and the M coder of Fig. 6.5. When comparing this value to the overhead of the various P coders given in Fig. 6.3, the M coder has approximately only half the overhead than the best depicted P coder. Furthermore, it has approximately the same overhead as a P coder that uses 10 ideal binary arithmetic bin coders. To achieve a P coder with an overhead that is comparable to the M coder, it is likely that more than 10 V2V codes of very low redundancy are necessary. In general, the number of bin coders and the maximum source and code word lengths are traded off against the overhead of the P coder.

6.2 Complexity and throughput considerations

Due to the large structural similarity between PIPE coding and CABAC, most of the literature about complexity and throughput aspects of CABAC also apply to PIPE coding. Most of the known CABAC optimizations, don't interact with the internal structure

of the M coder and can also be applied to the P coder of PIPE coding. Conversely, optimizations that target the M coder usually don't interact with the other stages of the CABAC system.

Several approaches for efficient implementations of CABAC exist which are mainly focused on the decoder. Some of them apply to hardware and software implementations, like the fast renormalization technique [16], while others mainly address hardware implementations [81, 14, 15]. Many of these techniques evolved after the standardization of H.264/AVC and the challenges of efficiently implementing CABAC became well understood. For example, hardware decoder implementations usually employ a pipelined CABAC decoder where the process of decoding a bin is divided into several pipeline stages like e.g. context selection, context model loading, binary arithmetic decoding, and context model update [15]. When the context selection for a particular bin depends on the previously decoded bin, the context selection stage is blocked until the binary arithmetic decoding of the previous bin is finished. This can take several cycles since all pipeline stages must be passed to decode the previous bin. In H.264/AVC, many situations exist where pipeline stages are interrupted. Without going into detail, most pipeline stalls can be traced back to dependencies between consecutive bins. Reducing these dependencies increases the throughput. In H.265/HEVC, much care has been taken to find a good trade off between compression efficiency and the reduction of entropy coding dependencies. A comprehensive discussion of this topic can be found in [82] and [83].

When designing a CABAC or PIPE coding system, several parameters can be used to trade off the complexity against the compression efficiency. The most relevant parameters which apply to both, CABAC and PIPE coding are:

Binarization rules The complexity of the rules for deriving the binarized representations of syntax elements.

Bins per syntax element The average number of bins resulting from the binarization of a syntax element.

Context derivation rules The complexity of the derivation rules for allocating probability models to bins.

Probability model memory The total number of required probability models.

Bin-to-bin dependencies Binarization, context modeling and probability estimation for a bin can depend on a number of immediately preceding bins. Removing

such dependencies allows the joint processing of numbers of bins but usually also has the disadvantage to reduce the compression efficiency.

For the best compression efficiency, binarization, context derivation, and probability modeling must carefully be designed to achieve the best trade-off between modeling accuracy and context dilution. When the best trade-off is found, a further reduction of the complexity of any of the above-mentioned parameters usually leads to a decreased compression efficiency.

The bin-to-bin dependencies are particularly problematic for decoder implementations. In the encoder, no feedback loops exist between binarization, context modeling, probability estimation, and bin encoding. I.e., each stage can operate independently of the subsequent stages. This is different in the decoder where often a bin has to be decoded completely before being able to carry out context modeling and probability estimation of the next bin. In general, if for a sequence of consecutive bins no dependencies exist amonst the bins in the entropy coding stage, this sequence can jointly be passed from stage to stage in encoder and decoder. Depending on the stage, it may also be possible to process the bins in parallel. In the case of CABAC, however, there always exist bin-to-bin dependencies in the M coder which leave few options for parallel processing. I.e., if context modeling and probability estimation are configured to deliver sequences of jointly processable bins, the M coder still has to sequentially process each bin. The question arises, how this throughput limiting property of the M coder can be removed. A solution to this problem is to replace the M coder with the P coder (see Fig. 2.1), which corresponds to using PIPE coding instead of CABAC. When the P coder is properly configured, sequences of bins can also be jointly processed in the P encoder and decoder. As discussed in Sec. 7.2, the chunk-based multiplexing of PIPE coding can be configured in a way so that each bin decoder has a guaranteed number of readily decoded bins available. The probability interval allocator must then read and arrange the bins of the individual bin decoders according to the requested bin sequence. As a proof of concept, a pipelined hardware implementation of such a P decoder is presented in [68] where the bin buffers, which contain the readily decoded bins per bin decoder, also act as pipeline registers. The multi bin probability interval allocator is implemented in a separate pipeline stage. For single bin decoding, it only accounts for approximately 20% of the P coder chip area (for the PIPE coder setup used in [68]). When extending this stage to simultaneous decoding of N bins, the related chip area will approximately be $0.8 + 0.2N$ times the area of the P decoder for single bins. I.e., the required chip area grows linearly with the number of simultaneously decodable bins.

When dependencies between bins are not removed, speculative computation can be used to simultaneously process multiple bins. In this case, the chip area grows exponentially for the P and M decoder. However, speculation is required in the probability interval allocation stage of the P decoder only which should require less chip area than a speculative multi bin M decoder [68].

6.3 Complexity-scalable entropy coding

The basic idea of complexity scalable PIPE coding is published in [68] where it is proposed as entropy coder for the H.265/HEVC video compression standard. In H.264/AVC, two entropy coding engines are available. CABAC and context-adaptive variable length coding (CAVLC) which is less complex and less efficient than CABAC. Depending on the available resources, one or another can be selected. For compatibility reasons, both entropy coders are often required in an encoder or decoder implementation. Since CABAC and CAVLC have few in common, an implementation that supports both is complex. An alternative is to use a complexity-scalable configuration of CABAC or PIPE coding. This can be achieved by designing two or more coder setups with different complexity operation points so that a less complex setup can be achieved by selectively replacing features of a more complex setup with less complex alternatives. This simplifies implementations where more than one complexity operation point is required. If a high throughput operation point is desired, it can be achieved by configuring all stages of the entropy coder to allow the joint processing of bin sequences by removing bin-to-bin dependencies. Since the M coder of CABAC can hardly process sequences of bins jointly, the P coder of PIPE coding may be more suitable for such a configuration, as discussed in the previous section.

The concept is demonstrated by a PIPE coding setup with three operation points for H.265/HEVC video coding. The first operation point is a high efficiency (HE) mode with good compression performance and high complexity. The second operation point is a low complexity (LC) mode with lower coding efficiency and significantly reduced complexity and which allows a high achievable throughput. The HE mode is optimized to deliver a compression efficiency close to state of the art CABAC while in the LC mode, many simplifications are introduced, as well as the ability of jointly processing multiple bins. The third operation point is a medium complexity (MC) mode which lies between the HE and LC mode in terms of complexity and compression efficiency.

In the following, principles for deriving an MC or LC mode from the HE mode are discussed. They involve selective simplifications until the desired MC or LC operation

point is reached while as much of the compression efficiency as possible is retained. For the LC mode, more simplifications are applied than for the MC mode. The parameters as listed in Sec. 6.2 can be adjusted according to the following principles:

Simplified context derivation The HE context derivation rules are simplified for the LC mode. For most bins, the rules are disabled, which corresponds to selecting always the same probability model for a given bin. For some bins where this would lead to an unacceptable increase of the bit rate, more sophisticated rules may be used.

Binarization reordering The bins are reordered so that the length of bin sequences that don't have bin-to-bin dependencies in the LC mode is increased. Of particular interest are reorderings that can also be applied to the HE mode without major disadvantages. In this way, the differences between HE and LC binarization can be minimized.

Simplified probability estimator In the HE mode, a probability model is updated after each bin encoded with it. This introduces bin-to-bin dependencies if a probability model is used for consecutive bins. Reducing the frequency of probability model updates or fully disabling them for the LC mode reduces the complexity and the bin-to-bin dependencies.

With the above principles, a PIPE coding setup with two or more operation points can be derived, which are optimized to different needs like low complexity, high efficiency, high throughput, etc.

Exemplary complexity-scalable PIPE coder

The concept of complexity-scalable entropy coding is demonstrated with the PIPE coding setup as it is proposed as entropy coder for the H.265/HEVC video coding standard [68]. It is based on the working draft 3 (WD3) of H.265/HEVC [84] which was the current version at the time of the proposal [68]. Since then, many changes were incorporated into H.265/HEVC and the final standard is quite different from WD3. However, no fundamental changes were made in the entropy coding stage and the concept of how complexity-scalability can be implemented remains applicable. Therefore, proposal [68] still gives an appropriate demonstration of a complexity-scalable entropy coder. A software implementation, also available for download (accompanying [68]), is used for the experimental evaluation. It is based on the HEVC Test Model 3.2 (HM

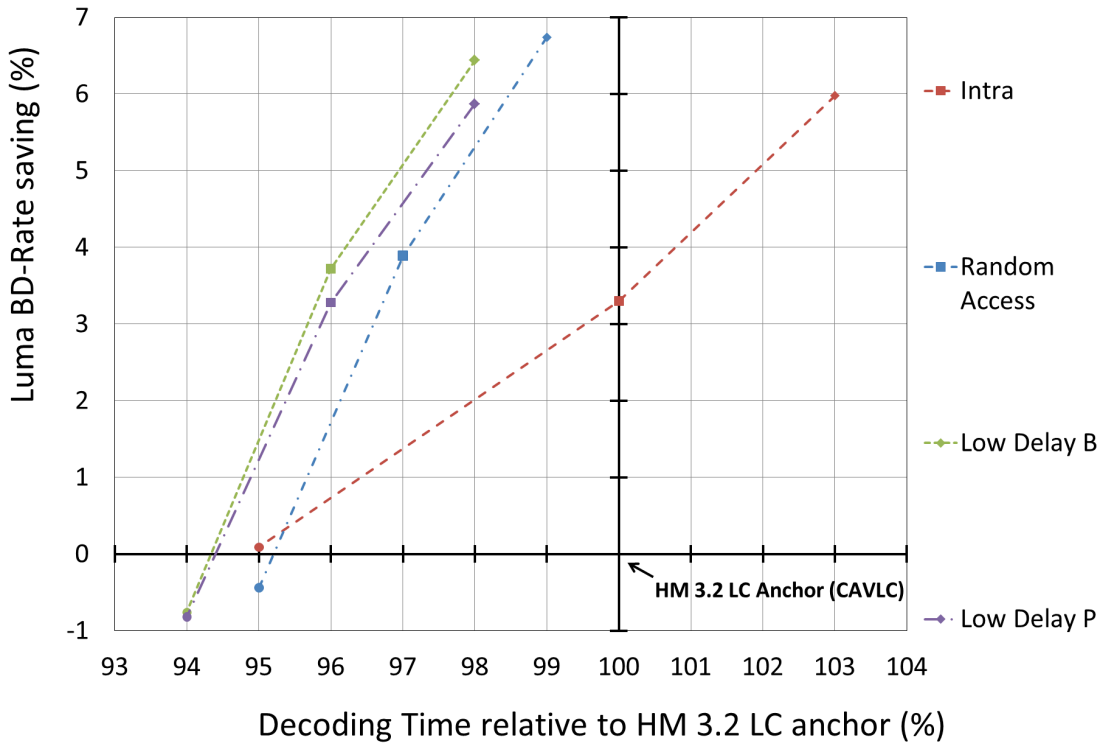

Figure 6.6: Luma BD-rate compared to decoding time (chart taken from Fig. 4 in [68]).

3.2) [85, 86, 87]. For details about the configuration of the HE, ME, and LC mode, the reader is referred to [68]. An impression of the trade off between coding efficiency and computational complexity is shown in Fig. 6.6 (which is taken from [68]). It compares the decoding time and the so-called *Bjøntegaard delta bit rate* (BD rate) [88] for the three complexity-scalable configurations against the HM 3.2 reference software. Note that the HM 3.2 reference software is not strongly optimized for speed and the decoding time may behave different in an optimized software or hardware implementation. However, it is reasonable to assume that the tendency stays the same. Each of the depicted lines consists of three data points corresponding to the LC, MC, and HE configuration, which belong to the lowest, medium, and highest BD rate savings, respectively. It can be seen that complexity scalability is achieved in a BD rate range of approximately 7%. The observed differences in the decoding time are rather low so that the decoding time reduction seem not to justify the BD rate increase. However, many of the complexity reductions are rather useful for parallelized hardware implementations. This is not reflected in the decoding time measurements of Fig. 6.6 since they are based on a single-threaded software implementation.

6.4 Throughput optimized V2V coder setup

The work by Roth et al. [89] shows how the throughput of a software implementation of PIPE coding can be increased by V2V code concatenation. It is based on the fact that the optimal lookup table size for a V2V coder implementation depends on the utilized software or hardware platform. By concatenating V2V codes, the average (and maximum) source and code word length can be increased while the average code length of the V2V code remains unchanged. Concatenation means in this context, that a composite V2V code is created from as many copies of a given V2V code as possible so that a predetermined maximum code tree height is not exceeded. When encoding or decoding of a code word is a single table lookup[1], the throughput is larger if the average source and code word lengths of a V2V codes are longer and concatenation is one way of increasing V2V code length. An obvious alternative is to design V2V codes with longer source and code words, which would also likely have a lower average code length. For the effects on the throughput, it should not matter whether concatenated or otherwise derived V2V codes are used and the runtime analysis in [89] can be assumed to be a generally valid demonstration of the throughput behavior of a lookup table-based single-threaded PIPE decoder software implementation. Roth et al. [89] report parsing time savings of up to 15% and decoding time savings of up to 5% compared to CABAC for the use of an 8 bit lookup table per bin decoder, which corresponds to a maximum code word length of 8 bits. The PIPE decoder uses the chunk-based multiplexing as discussed in Ch. 7 with 8 bit chunks. Low-dealy control is enabled with a ring buffer size of 512 chunks. It is not stated in [89], but the implementation is based on HEVC Test Model 7.0 (HM 7.0) [90, 91, 92], which is also used for the CABAC reference[2]. Note, that CABAC in HM 7.0 contains some speed optimizations (e.g. using the fast renormalization [16]) so that the runtime comparisons should be reasonably fair. These throughput improvements are remarkable, since they are achieved without employing a parallelized software implementation. It can be assumed that a parallelized hardware implementation may further increase the throughput.

[1] Assuming that a table lookup approximately consumes a constant time.

[2] According to a personal communication with the authors of [89]

6.5 Chapter summary

In this chapter, several P coders were designed. The average code length per symbol was used as minimization criterion for P coder design in order to approach the entropy rate of the coding bins. It turned out that a distribution of conditional probabilities of one could be derived, which covers almost all properties of the coding bins that are relevant for P coder design. Such a distribution was estimated based on a training set of H.265/HEVC-encoded bit streams, which are selected to reflect a wide range of different properties related to the video content type, the encoder settings, etc.

An algorithm for selecting V2V codes for a P coder out of a candidate set was proposed. It is based on successively removing V2V codes from the candidate set so that the increase of the average code length per symbol is minimized in each step. The algorithm yields good results when the number of desired bin coders is not too small. Furthermore, it can be used for a pre-examination in order to find out whether exhaustive search may further improve a P coder that uses many bin coders.

It was demonstrated that a P coder with 12 V2V codes of the S6C6+UR candidate set has a percentaged overhead of only 0.2%. When 10 ideal binary arithmetic bin coders are used, a percentaged overhead is less than 0.1%. An analysis in this chapter shows that this value is comparable to the percentaged overhead of the M coder as used in H.265/HEVC.

Based on a study of complexity and throughput aspects of the PIPE coding system, the concept of complexity-scalable entropy coding is developed. In each of the stages of a PIPE coder, a different trade-off between complexity and coding efficiency can be configured. In particular, the options for parallel processing can be greatly improved. In contrast to the M coder of CABAC, the P coder allows a large degree of parallel processing. This is an important aspect since prallelization is most efficient when all stages of a PIPE coder allow parallel processing. This includes a highly parallelizable multiplexing scheme, which is the topic of the next chapter.

7

Multiplexing in PIPE coding

The concept presented in Sec. 7.2 is already published in [28] and [93].

The PIPE coding system uses a plurality of independent bin coders. Each of the bin encoders produces output bits, which contain the encoded bins. In the PIPE decoder, each of the bin decoders receives the code bits, the corresponding bin encoder has produced. This requires an appropriate multiplexing of coded bits as already outlined in Sec. 2.6. In this chapter, various aspects of multiplexing in the PIPE coding concept are discussed. Firstly, the decoder-synchronized encoding is shortly reviewed and based on this, the so-called *chunk-based multiplexing* is developed.

7.1 Review of decoder-synchronized encoding

In PIPE encoding, the probability estimator produces a sequence of bins with probability estimates, which are distributed to the bin encoders by the probability interval allocator (see Fig. 2.1). Consequently, the bin encoders receive bins in some seemingly unstructured order. Binarization, context modeling, and probability estimation is structured in a way that the decoder is always able to reproduce the probability estimate of the next bin to be decoded. Therefore, the probability interval allocator in the decoder receives bin decoding requests from the probability estimator in the same seemingly unstructured order as in the encoder. Decoder-synchronized encoding interleaves the code words of the bin encoders in a way such that the bin decoders can read them from the bit stream when the first bin encoded in a code word is requested by the probability estimator. This technique appears several times in the related literature [24, 27, 26]. While it is probably the simplest interleaving scheme for the decoder, the encoder is rather complicated. Basically, a bin encoder has to reserve a slot for a code word in

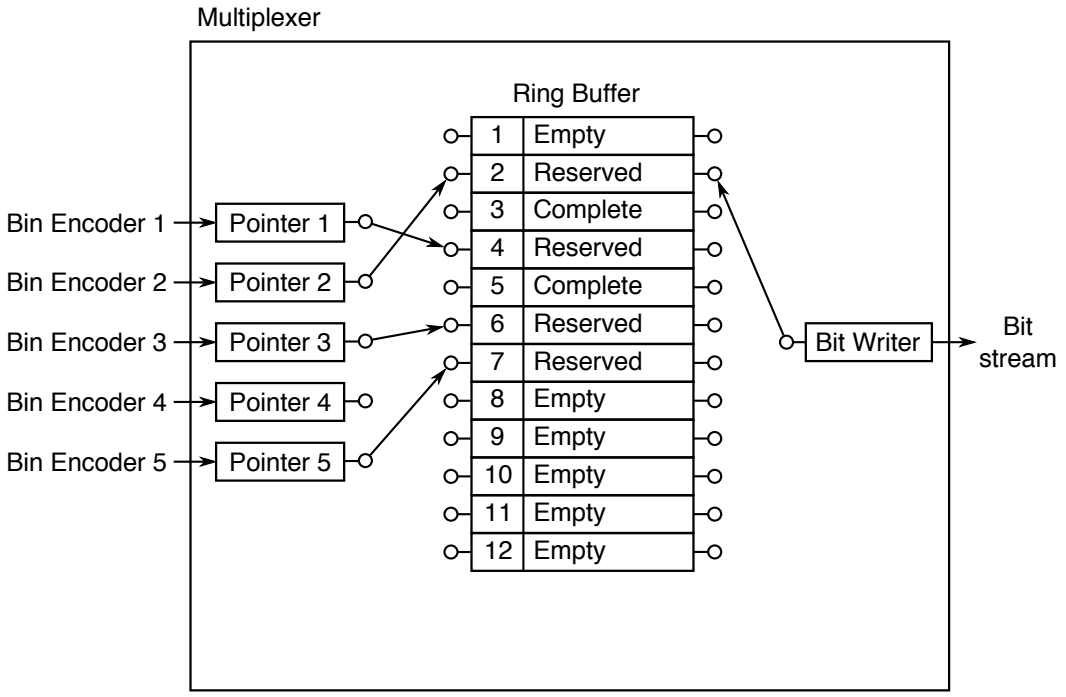

Figure 7.1: Exemplary multiplexer for decoder-synchronized encoding.

the bit stream when it receives the first bin of it. At this time, the encoder cannot know how long the code word will be and how long it takes until it receives the last bin of the code word. A bin encoder which uses fixed-length code words can reserve a slot of the correct length for the code word since it knows how long the code word will be. However, codes with fixed-length code words, like Tunstall codes, tend to have a higher redundancy than V2V codes of similar size (see Sec. 5.4.2). Regardless of whether fixed or variable-length code words are used, there is no guarantee that a code word completes in a certain time span which means that the required code word buffer is not limited (or only limited by other application-specific parameters like maximum stream length, etc.). Since this is undesirable for practical encoder implementations, low-delay techniques like e.g. the so-called fill-symbol control as proposed in [24] are usually employed. They avoid buffer overflows by artificial code word completion. This makes the decoder slightly more complicated as it has to detect and regard artificially completed code words. An exemplary decoder-synchronized encoder is depicted in Fig. 7.1. Each element of the ring buffer is in one of the three possible states *Empty*, *Reserved*, or *Complete*. In *Reserved* state, a buffer element is also able to store an incomplete source word and in *Complete* state, a buffer element is also able to store a

code word. A bit writer points to the buffer element to be written to the bit stream next. For each bin encoder, a pointer is maintained, which points to a buffer element where the next code word of this bin encoder has to be placed. In the beginning of the encoding process, all buffer elements are marked *Empty*, the pointers of the bin encoders are set to an invalid state, and the bit writer points to an arbitrary buffer element (e.g. the first element). During the encoding process, when a bin encoder receives the first bin of a code word, it reserves the *next free* element in the ring buffer. The *next free* element in the ring buffer is found by starting at the buffer element, the bit writer points to and going down (potentially including wraparounds) until an *Empty* buffer element is reached. This buffer element is marked *Reserved*, the bin is stored in this element, and the *Pointer* associated with the corresponding bin encoder is set to this element. Further bins that occur in the corresponding bin encoder are collected in this reserved buffer element until the code word is complete. If this happens, the buffer element is marked *Complete* and the contained source word is replaced with the corresponding code word. The pointer of the corresponding bin encoder is invalidated. Whenever the bit writer points to a *Complete* buffer element, it writes this stored code word to the bit stream, sets the buffer element to *Empty*. Afterwards, the bit writer is set to point to the next element (potentially including a wraparound) and if this element is *Complete* as well, the procedure is repeated until the bit writer points to either a *Reserved* or an *Empty* element.

Fig. 7.1 shows an exemplary state in which the multiplexer could be. A number of buffer elements are *Reserved* and in-between them are *Complete* buffer elements. All other buffer elements are *Empty*. The number of *Reserved* buffer elements is lower or equal to the number of bin encoders since when a code word is finished in a bin encoder, the corresponding pointer is invalidated (pointing nowhere). The number of *Complete* code words is not limited and depends on how long it takes until the oldest *Reserved* buffer element is completed. When all *Empty* elements are exhausted, no new buffer element can be reserved and the procedure cannot continue. This has to be avoided by the encoding application, e.g. by choosing the ring buffer size appropriately.

Low-delay control (see fill-symbol control in [24]) is a concept which also addresses this problem. It is simple to add to the described encoder but requires a substantial modification of the decoder. Whenever a bin encoder needs to reserve a buffer element and no *Empty* buffer element is available, the buffer element where the bit writer points to is completed by adding arbitrary flush bins to the corresponding bin encoder until the source word is complete. Preferably flush bins that lead to the short-

est possible code word. Then, this code word (and potentially further already complete code words) can be written to the bit streams and the corresponding buffer elements become *Empty*. In the decoder, the artificially added bins need to be detected and discarded. This can be achieved by tracking the ring buffer index for each code word as it occurs in the encoder. When a code word is read from the bit stream, the ring buffer index is increased by 1 or, in the case of a wraparound, it is set to 1. Immediately afterwards, it is checked whether one of the code words of the other bin decoders has the same index as the currently read code word. If so, the remaining bits of this code word are flush bits and need to be discarded.

7.1.1 Trade-off between bit rate and ring buffer size

Occasionally adding flush bins creates a bit rate overhead. The smaller the ring buffer is, the more often flush bits are generated. Furthermore, it depends on the source word lengths and on how the bins are distributed to the bin coders. A bin coder that processes only a small portion of the bins takes longer (on average) to complete a source word, which results in a higher probability of flushing bins. For a particular source word of a bin encoder, flushing bins can only occur when between the first and last bin of the source word, sufficient bins of other bin encoders occur in the sequence of coding bins such that the ring buffer overflows. When flushing occurs, the buffer element to be flushed contains an incomplete source word. This is one of all possible prefixes of the source words of the bin encoder (not including whole source words). Depending on the probability of a prefix, and on the length of the associated code word (derived by flushing), each prefix may have a different overhead. However, an exact calculation of the probability of a prefix is difficult so that it may be more suitable to derive it by simulation.

7.1.2 Disadvantages

Decoder-synchronized encoding has a number of disadvantages, which motivate the development of the so-called *chunk-based* multiplexing in the next section. The main issues are:

Sequential decoding The decoding of bins is a sequential operation, which can hardly be parallelized. More precisely, decoding of a code word is initiated when the first bin of the corresponding source word is requested by the probability interval allocator. While the code word is decoded, the operation of the probability

interval allocator is interrupted until the first bin is available. Decoding cannot start earlier since the bin decoder cannot know where the code word is located in the bit stream.

Ring buffer utilization A ring buffer element stores exactly one code word or an unambiguous reference to it. It depends on the PIPE coder setup how this is done in the most efficient way. For example, a unique index can be assigned to each code word of all coders. Or, if more efficient, the code words along with their length can also be stored directly in the ring buffer elements. In any case, the way of identifying a code word requires much more bits than the sum of the pure code word bits.

Variable length reading from the bit stream A decoding operation reads a variable number of bits from the bit stream where the length depends on the actual bits themselves. This requires a number of bit isolation and joining operations.

7.2 Chunk-based multiplexing

Chunk-based multiplexing is a technique that addresses the disadvantages of decoder-synchronized encoding as outlined in the previous section. Instead of interleaving code words, bit sequences of fixed-length (so-called chunks) are interleaved. The basic concept is published in [28] and [93] and is related to the scheme by Boliek et al. in [27]. The code words produced by a bin encoder are concatenated to a bit stream, which is then split into chunks. The chunks of all bin encoders are interleaved according to a predefined rule, similar as the code words are in decoder-synchronized encoding. This introduces a number of changes to the encoder and decoder. The ring buffer in the encoder operates on chunks instead of code words and each bin encoder can have more than one reservations at the same time. Furthermore, each bin encoder maintains a writing position that points to the next free bit inside of the reserved chunks. When the first bin of a source word arrives at a bin encoder, it checks the number of free bits in its reserved chunks and decides whether it reserves further chunks. A possible rule for this check could be as follows.

- When the first bin of a code word is requested, reserve chunks until the number of free bits is greater or equal to a predefined threshold n.

In other words, the threshold n is the minimum number of free bits available after the rule is executed. n must be large enough so that the next code word fits in the free

bits but can also be much larger. A reasonable choice is to set n to the length of the largest code word of the bin encoder as demonstrated in the following example. Fig.

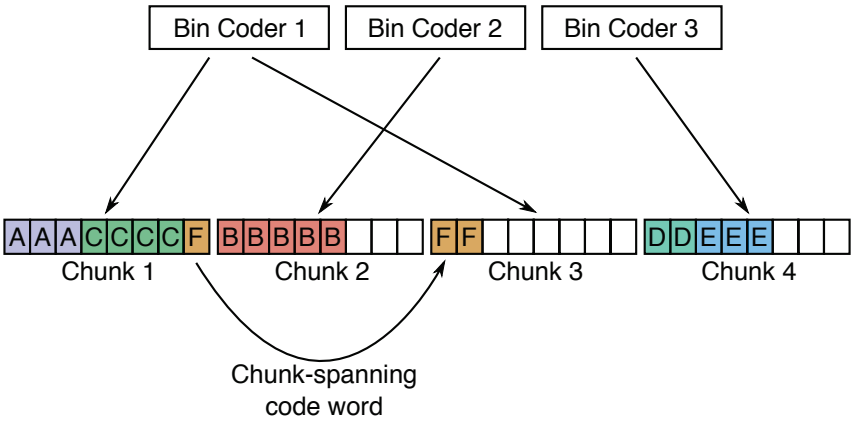

Figure 7.2: Exemplary multiplexing with chunks.

7.2 shows an exemplary chunk-based multiplexing of six code words $a, \ldots, f$ occurring in three bin encoders or decoders. The length of a chunk is chosen to be 8 bit. The bits of the code words are labeled with capital letters A, . . . ,F, respectively. The code words $a, \ldots, f$ are ordered according to the occurrence of the first bins of their source words. Code words $a, c,$ and f occur in bin coder 1, code word b occurs in bin coder 2 and code words d and e occur in bin coder 3. The encoder operates as follows. At the beginning of the encoding, no bin encoder has reserved chunks. Therefore, when the first bin of a source word occurs, chunks are reserved until n free bits are available according to the triggering rule. It is assumed that bin encoder 1 reserves chunk 1 after the first bin of code word a and bin encoder 2 reserves chunk 2 after the first bin of code word b. When further bins occur so that code word a is complete and the first bin of the next source word appears in bin encoder 1, the triggering rule applies. I.e., the number of free bits in chunk 1 is counted and compared to the threshold n, which is set to the length of the longest possible code word of bin encoder 1. If it would fit, no reservation is done and if not, chunks are reserved until it would fit. In the present example, it is assumed that it wouldn't fit and chunk 3 is reserved. Afterwards, code words d, e, and f follow where f is partly written to chunk 1 and partly to chunk 3. When all bits of a chunk are filled with code word bits, it is marked complete and the ring buffer can operate exactly in the same way as for decoder-synchronized encoding. Low-delay control can also be used with chunk-based multiplexing. When a reserved chunk has to be flushed because of a ring buffer overflow, a potentially pending code word in the reserved chunk is completed and written to the chunk. If further bits are

free in the chunk, they are filled with arbitrary bits and the chunk can be written to the bit stream. The decoder has to detect such situations and must discard remaining bits of a source word and also remaining bits in a chunk if present. On the one hand, decoding is slightly more complex because each bin decoder has to maintain a number of read chunks and a pointer to the next bit to read inside the chunks. On the other hand, the utilization of the ring buffer and the options for parallelization are improved as discussed in the following sections.

7.2.1 Ring buffer utilization

An advantage of chunk-based multiplexing is that each bin of the ring buffer stores code word bits without overhead. In contrast, decoder-synchronized encoding only stores one code word in a ring buffer element which is much more inefficient. In chunk-based multiplexing, complete chunks can be written to the bit stream without further processing while in decoder-synchronized encoding, the variable length code words need to be properly concatenated first. In other words, a sequence of consecutive complete chunks in the ring buffer is a piece of the bit stream, ready for transmission or storage. This allows to completely skip low-delay control in some cases. For example, in a video coding application, the entropy coder encodes bins from each frame (or slice) separately in order to be able to independently entropy decode each frame. When the encoder operates in a way that the code bits of a frame are stored in memory first, chunk-based multiplexing can be carried out in the final bit stream memory directly since chunks are reserved at the position in the memory where they finally have to be. This requires random-access writing to the bit stream memory in the encoder. Although, it may be worthwhile to accept this increased complexity in the encoder because the bit rate overhead associated with low delay control is avoided and the decoder does not need to handle low-delay control.

7.2.2 Parallel bin decoding

Increasing the threshold n for triggering chunks creates room for parallel code word decoding. When n is e.g. set to two times the length of the longest code word, at least two code words can be decoded. While the first bin of the first code word is required immediately, the bins of the second code word are needed sometime later. The bin decoder can decode the second code word independently of the further operation of the probability interval allocator, so to say, in the background. Consequently, it can carry out background decoding to always have readily decoded bins available and will

be able to immediately answer bin requests. This is of particular interest for hardware implementations of PIPE coding. In software, it is difficult to realize background decoding of code words because communication between threads of a multi-threaded implementation rather slow. For an in-depth discussion of hardware aspects, the reader is referred to [28].

7.2.3 Experimental evaluation

An evaluation of the bit rate overhead that is associated with chunk-based multiplexing is given in [28, Sec. 3.3.2]. The BD rate increase amounts to 0.21% for a P coder that uses the 8 systematic V2V codes as depicted in Fig. 5.13 with a chunk size of 8 bit and a ring buffer size of 512 chunks. The triggering threshold n is set to the smallest possible value, which is different for each bin coder. Note that the BD rate increase is measured relative to a PIPE coder which uses decoder-synchronized encoding without low delay constraint instead of chunk-based multiplexing. This should have virtually the same compression efficiency as chunk-based multiplexing without low-delay control, since the same code words are contained in the bit stream in both cases. This allows, however, a comparison of the decoding time of chunk-based multiplexing to decoder-synchronized encoding. A small speedup can be observed for chunk-based multiplexing. As discussed in Sec. 6.4, a more recent study by Roth et al. [89] demonstrates that substantial decoding time reductions can be achieved with chunk-based multiplexing when the average code word lengths of the V2V codes are high enough.

7.3 Chapter summary

In this chapter, a chunk-based multiplexing scheme for bin coder output was presented. It is a further development of the decoder-synchronized encoding. Instead of interleaving individual code words of the bin coders, bit sequences of fixed length are formed by each bin coder independently, which are then multiplexed.

Furthermore, chunk-based multiplexing allows to control when a bin decoder receives new chunks from the bit stream by employing a triggering threshold n. This is an important technique for allowing uninterrupted parallelized decoding on a large scale. When the triggering threshold is increased, the number of bins that can be present readily decoded in a bin decoder is increased as well. In other words, such a bin decoder is able to provide multiple bins in a single operation if required.

It was shown that interleaving of bit sequences of fixed length is also advantageous in an encoder implementation. The ring buffer is utilized in the best possible way since it stores portions of the bit stream. Moreover, in applications where a PIPE coder is terminated in manageable time intervals so that the resulting bit stream portion fits into memory, the ring buffer can completely be avoided. In this case, no bit rate overhead is caused by low-delay control. Furthermore, the demultiplexer is also less complex since it doesn't have to regard low-delay control.

8

Conclusions and Future Work

This work addresses the problem of designing V2V codes for the PIPE coding concept. Redundancy and parallelization capabilities are the main parameters of interest in this context. While many well-known algorithms are available for constructing code trees for minimum redundancy V2V codes with various constraints, the design of corresponding source trees is mainly based on exhaustive search. The concept of canonical V2V codes allows to reduce the complexity of this problem by eliminating equivalent source trees in the exhaustive search. Moreover, the classification into prime and composite V2V codes can further improve the efficiency of the search for minimum redundancy V2V codes by early termination.

The PIPE coding system requires a number of V2V codes that cover the probability interval $(0, 0.5]$, which leads to the idea of jointly deriving a set of such V2V codes. One handy way to achieve this is to use the probability as variable p, which leads to the concept of polynomial-based algorithms. Such an algorithm yields a set of contiguous subintervals of $(0, 0.5]$ with each subinterval having an associated V2V code that is the result of the algorithm for the subinterval. Less complex algorithms, like Huffman's or Tunstall's algorithm, can be made polynomial-based by small modifications. For more complex algorithms, like the lazy reverse runlength package merge algorithm, a finite-state machine-based implementation is much more manageable than directly introducing polynomials in the algorithm.

Since V2V codes can be arbitrarily large, their size needs to be constrained in design algorithms. Most literature about V2V code design limits the number of leaf nodes of source and code trees. However, for a lookup table-based implementation, codes for which only the source and code tree height is limited, utilize the lookup table in the best possible way. Since the complexity of searching minimum redundancy V2V

codes mainly depends on the cardinality of the candidate set of source trees, only the maximum height of a source tree cannot be increased arbitrarily. With an estimated number of up to $1.9 \cdot 10^{13}$ different canonical source trees, an exhaustive search for a maximum source tree height of 7 already seems unfeasible. While minimum redundancy V2V codes with a maximum source and code tree height of 6 already show a remarkably low redundancy for $p > 0.1$, they are not satisfactory for $p \leq 0.1$. In general, V2V codes seem to develop a particular structure for $p \to 0$, which leads to unary-to-rice codes. From a redundancy perspective, they seem to be the only option for very small p. Fortunately, they have a very systematic structure so that even very large codes can be implemented with low complexity. It turns out that also for larger values of p, V2V codes with a systematic structure can be found, which leads to the concept of systematic V2V codes.

The polynomial-based V2V code design algorithms tend to yield a strongly fragmented probability interval and a large number of V2V codes. It is intuitively clear that the envelope of the redundancy of these V2V codes cannot increase strongly when one of the V2V codes with a very narrow associated subinterval is removed. Consequently, it makes sense to only use a skillfully selected subset of V2V codes for a PIPE coder setup. In general, the size of the subset is traded of against the redundancy of the resulting PIPE coder, depending on the requirements of the application. Since the redundancy of a PIPE coder is given as a function of the probability, a minimization criterion needs to be defined like the average code length of the resulting PIPE coder for a given source. In practical applications, the properties of such a source is usually best described through a set of realizations like e.g. a set of H.265/HEVC-encoded bit streams. By appropriately weighting the encoded lengths of the individual realizations, the PIPE coder can be optimized to particular source characteristics.

A high degree of parallelization can be realized with a PIPE coder when each element in the processing chain is prepared for it. This is a difference to CABAC where the M coder withstands any attempts of large-scale parallelization. For allowing a high degree of parallel processing in the P coder, the individual bin coders must be able to run in parallel. This is mainly a question of the synchronization between bin coders, which is controlled by the multiplexing technique. Chunk-based multiplexing ensures that the bin decoders always have a sufficient amount of code bits available so that they can always have a sufficient amount of decoded bins available for the next processing stage. Consequently, requests for multiple bins can be answered immediately. Fortunately, chunk-based multiplexing is possible without introducing a noteworthy bit rate increase so that it is an option when the highest compression efficiency is needed.

8.1 Future Work

The problem of efficiently designing minimum redundancy V2V codes remains unsolved. The approach followed in this work is based on simplifying the exhaustive search and it may be interesting to advance this idea. One important tools for this purpose is the canonical representation of source trees, which greatly reduces the search space. The observation that minimum redundancy V2V codes cannot be composite, can be used for early termination. It could, however, also be useful for further reductions of the search space. For example, if two canonical source words that can be siblings are associated with the same code word lengths, a composite V2V code can be constructed from this canonical representation. Such source trees can be excluded from the search space. Moreover, consider the proportion of prime V2V codes amongst all V2V codes of a given source and code tree height. It can be speculated that the proportion of prime V2V codes decreases with increasing source and code tree height. This is because the number of possible leaf nodes increases exponentially with the source and code tree height, while the number of possible different source and code word lengths linearly increases. Consequently, the number of source and code words of the same length must strongly increase. If a test can be constructed for whether a V2V code is composite, based on the availability of sufficient source and code words of the same length, the size of the search space can be further reduced.

Several PIPE coder setups with a low overhead are presented in this work. The chunk-based multiplexing scheme enables the uninterrupted and parallelized decoding flow of bins. While this is possible without notably increasing the overhead, the question remains, whether this is also possible in the other stages of PIPE coding, like binarization, context modeling, and probability estimation. The ever-increasing video quality and associated data rates may in future create a need for parallel processing inside of the entropy coding engine.

Appendix A

Canonical V2V code tables

A.1 Optimal canonical V2V codes of limited size

A.1.1 Fixed-to-variable length codes

Name	F2V2_1				F2V2_2			
I_k	$(0, 0.38]$				$(0.38, 0.5]$			
$M(\sigma_i)$	2	1	1	0	2	1	1	0
$N(\sigma_i)$	0	1	1	2	0	1	1	2
$\ell(k(\sigma_i))$	1	2	3	3	2	2	2	2

Table A.1: Optimal F2V codes with source word length of 2.

Name	F2V3_1								F2V3_2								F2V3_3							
I_k	$(0, 0.2929]$								$(0.2929, 0.3333]$								$(0.3333, 0.4302]$							
$M(\sigma_i)$	3	2	2	2	1	1	1	0	3	2	2	2	1	1	1	0	3	2	2	2	1	1	1	0
$N(\sigma_i)$	0	1	1	1	2	2	2	3	0	1	1	1	2	2	2	3	0	1	1	1	2	2	2	3
$\ell(k(\sigma_i))$	1	3	3	3	5	5	5	5	2	2	3	3	4	4	4	4	2	3	3	3	3	3	4	4

Name	F2V3_4							
I_k	$(0.4302, 0.5]$							
$M(\sigma_i)$	3	2	2	2	1	1	1	0
$N(\sigma_i)$	0	1	1	1	2	2	2	3
$\ell(k(\sigma_i))$	3	3	3	3	3	3	3	3

Table A.2: Optimal F2V codes with source word length of 3.

Name	F2V4_1														
I_k	$(0, 0.1311]$														
$M(\sigma_i)$	4	3	3	3	3	2	2	2	2	2	2	1	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	2	3	3	3	3	4
$\ell(k(\sigma_i))$	1	3	3	3	4	6	7	7	7	7	7	9	9	9	10	10
Name	F2V4_2														
I_k	$(0.1311, 0.191]$														
$M(\sigma_i)$	4	3	3	3	3	2	2	2	2	2	2	1	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	2	3	3	3	3	4
$\ell(k(\sigma_i))$	1	3	3	4	4	5	6	6	6	6	6	8	8	8	9	9
Name	F2V4_3														
I_k	$(0.191, 0.2]$														
$M(\sigma_i)$	4	3	3	3	3	2	2	2	2	2	2	1	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	2	3	3	3	3	4
$\ell(k(\sigma_i))$	1	3	3	4	4	6	6	6	6	6	6	7	7	7	8	8
Name	F2V4_4														
I_k	$(0.2, 0.2076]$														
$M(\sigma_i)$	4	3	3	3	3	2	2	2	2	2	2	1	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	2	3	3	3	3	4
$\ell(k(\sigma_i))$	1	3	4	4	4	5	5	5	5	6	6	7	7	7	8	8
Name	F2V4_5														
I_k	$(0.2076, 0.3177]$														
$M(\sigma_i)$	4	3	3	3	3	2	2	2	2	2	2	1	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	2	3	3	3	3	4
$\ell(k(\sigma_i))$	2	3	3	3	3	5	5	5	5	5	5	6	6	6	7	7
Name	F2V4_6														
I_k	$(0.3177, 0.3333]$														
$M(\sigma_i)$	4	3	3	3	3	2	2	2	2	2	2	1	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	2	3	3	3	3	4
$\ell(k(\sigma_i))$	2	3	3	3	4	4	5	5	5	5	5	5	6	6	6	6

Table A.3: Optimal F2V codes with source word length of 4 (part 1).

Name	F2V4_7														
I_k	$(0.3333, 0.382]$														
$M(\sigma_i)$	4	3	3	3	3	2	2	2	2	2	2	1	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	2	3	3	3	3	4
$\ell(k(\sigma_i))$	3	3	3	4	4	4	4	4	4	4	4	5	5	5	6	6
Name	F2V4_8														
I_k	$(0.382, 0.4142]$														
$M(\sigma_i)$	4	3	3	3	3	2	2	2	2	2	2	1	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	2	3	3	3	3	4
$\ell(k(\sigma_i))$	3	3	4	4	4	4	4	4	4	4	4	4	5	5	5	5
Name	F2V4_9														
I_k	$(0.4142, 0.4503]$														
$M(\sigma_i)$	4	3	3	3	3	2	2	2	2	2	2	1	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	2	3	3	3	3	4
$\ell(k(\sigma_i))$	3	4	4	4	4	4	4	4	4	4	4	4	4	4	5	5
Name	F2V4_10														
I_k	$(0.4503, 0.5]$														
$M(\sigma_i)$	4	3	3	3	3	2	2	2	2	2	2	1	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	2	3	3	3	3	4
$\ell(k(\sigma_i))$	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

Table A.4: Optimal F2V codes with source word length of 4 (part 2).

Name	F2V5_1																									
I_k	$(0, 0.0836]$																									
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4
$\ell(k(\sigma_i))$	1	3	3	4	4	4	7	7	7	7	7	8	8	8	8	8	11	11	11	11	11	12	12	12	12	12	14
$M(\sigma_i)$	1	1	1	1	0																					
$N(\sigma_i)$	4	4	4	4	5																					
$\ell(k(\sigma_i))$	14	15	15	15	15																					

Table A.5: Optimal F2V codes with source word length of 5 (part 1).

Name	F2V5_2																								
I_k	$(0.0836, 0.0871]$																								
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4
$\ell(k(\sigma_i))$	1	3	4	4	4	4	6	6	6	6	6	7	7	7	7	7	10	10	10	10	10	11	11	11	11	13
$M(\sigma_i)$	1	1	1	1	0																					
$N(\sigma_i)$	4	4	4	4	5																					
$\ell(k(\sigma_i))$	13	14	14	14	14																					
Name	F2V5_3																								
I_k	$(0.0871, 0.1185]$																								
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4
$\ell(k(\sigma_i))$	1	3	4	4	4	4	6	6	6	6	7	7	7	7	7	7	9	9	9	9	9	10	10	10	10	12
$M(\sigma_i)$	1	1	1	1	0																					
$N(\sigma_i)$	4	4	4	4	5																					
$\ell(k(\sigma_i))$	12	13	13	13	13																					
Name	F2V5_4																								
I_k	$(0.1185, 0.1615]$																								
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4
$\ell(k(\sigma_i))$	1	4	4	4	4	4	6	6	6	6	6	6	6	6	6	6	8	8	8	8	8	9	9	9	9	11
$M(\sigma_i)$	1	1	1	1	0																					
$N(\sigma_i)$	4	4	4	4	5																					
$\ell(k(\sigma_i))$	11	12	12	12	12																					
Name	F2V5_5																								
I_k	$(0.1615, 0.1943]$																								
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4
$\ell(k(\sigma_i))$	1	4	4	4	4	4	6	6	6	6	6	6	6	6	6	6	8	8	8	8	9	9	9	9	9	10
$M(\sigma_i)$	1	1	1	1	0																					
$N(\sigma_i)$	4	4	4	4	5																					
$\ell(k(\sigma_i))$	10	11	11	11	11																					

Table A.6: Optimal F2V codes with source word length of 5 (part 2).

Name	F2V5_6																								
I_k	$(0.1943, 0.2]$																								
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4
$\ell(k(\sigma_i))$	2	3	3	3	4	4	5	5	5	6	6	6	6	6	6	6	8	8	8	8	8	8	8	8	8	9
$M(\sigma_i)$	1	1	1	1	0																				
$N(\sigma_i)$	4	4	4	4	5																				
$\ell(k(\sigma_i))$	9	10	10	10	10																				
Name	F2V5_7																								
I_k	$(0.2, 0.2219]$																								
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4
$\ell(k(\sigma_i))$	2	3	4	4	4	4	5	5	5	5	5	5	5	5	5	6	7	7	7	7	7	7	7	8	8	9
$M(\sigma_i)$	1	1	1	1	0																				
$N(\sigma_i)$	4	4	4	4	5																				
$\ell(k(\sigma_i))$	9	10	10	10	10																				
Name	F2V5_8																								
I_k	$(0.2219, 0.2358]$																								
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4
$\ell(k(\sigma_i))$	2	3	4	4	4	4	5	5	5	5	5	5	5	5	6	6	7	7	7	7	7	7	7	7	7	8
$M(\sigma_i)$	1	1	1	1	0																				
$N(\sigma_i)$	4	4	4	4	5																				
$\ell(k(\sigma_i))$	8	9	9	9	9																				
Name	F2V5_9																								
I_k	$(0.2358, 0.2957]$																								
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4
$\ell(k(\sigma_i))$	2	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	6	6	6	7	7	7	7	7	7	8
$M(\sigma_i)$	1	1	1	1	0																				
$N(\sigma_i)$	4	4	4	4	5																				
$\ell(k(\sigma_i))$	8	8	8	8	8																				

Table A.7: Optimal F2V codes with source word length of 5 (part 3).

Name	F2V5_10																								
I_k	$(0.2957, 0.3141]$																								
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4
$\ell(k(\sigma_i))$	3	3	4	4	4	4	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	7
$M(\sigma_i)$	1	1	1	1	0																					
$N(\sigma_i)$	4	4	4	4	5																					
$\ell(k(\sigma_i))$	7	8	8	8	8																					
Name	F2V5_11																								
I_k	$(0.3141, 0.3333]$																								
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4
$\ell(k(\sigma_i))$	3	4	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	7
$M(\sigma_i)$	1	1	1	1	0																					
$N(\sigma_i)$	4	4	4	4	5																					
$\ell(k(\sigma_i))$	7	7	7	7	7																					
Name	F2V5_12																								
I_k	$(0.3333, 0.382]$																								
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4
$\ell(k(\sigma_i))$	3	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	1	1	1	1	0																					
$N(\sigma_i)$	4	4	4	4	5																					
$\ell(k(\sigma_i))$	6	6	6	7	7																					
Name	F2V5_13																								
I_k	$(0.382, 0.3865]$																								
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4
$\ell(k(\sigma_i))$	3	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	1	1	1	1	0																					
$N(\sigma_i)$	4	4	4	4	5																					
$\ell(k(\sigma_i))$	6	6	6	6	6																					

Table A.8: Optimal F2V codes with source word length of 5 (part 4).

Name	F2V5_14																								
I_k	$(0.3865, 0.4142]$																								
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4
$\ell(k(\sigma_i))$	4	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6
$M(\sigma_i)$	1	1	1	1	0																					
$N(\sigma_i)$	4	4	4	4	5																					
$\ell(k(\sigma_i))$	6	6	6	6	6																					

Name	F2V5_15																								
I_k	$(0.4142, 0.4425]$																								
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4
$\ell(k(\sigma_i))$	4	4	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6
$M(\sigma_i)$	1	1	1	1	0																					
$N(\sigma_i)$	4	4	4	4	5																					
$\ell(k(\sigma_i))$	6	6	6	6	6																					

Name	F2V5_16																								
I_k	$(0.4425, 0.4614]$																								
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4
$\ell(k(\sigma_i))$	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	1	1	1	1	0																					
$N(\sigma_i)$	4	4	4	4	5																					
$\ell(k(\sigma_i))$	5	5	5	6	6																					

Name	F2V5_17																								
I_k	$(0.4614, 0.5]$																								
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	1	1	1	1	0																					
$N(\sigma_i)$	4	4	4	4	5																					
$\ell(k(\sigma_i))$	5	5	5	5	5																					

Table A.9: Optimal F2V codes with source word length of 5 (part 5).

A.1.2 Tunstall (variable-to-fixed length) codes

Name	V2F2_1				V2F2_2			
I_k	$(0, 0.38]$				$(0.38, 0.5]$			
$M(\sigma_i)$	3	2	1	0	2	1	1	0
$N(\sigma_i)$	0	1	1	1	0	1	1	2
$\ell(k(\sigma_i))$	2	2	2	2	2	2	2	2

Table A.10: Optimal Tunstall codes (V2F) with code word length of 2.

Name	V2F3_1								V2F3_2								V2F3_3							
I_k	$(0, 0.2219]$								$(0.2219, 0.2755]$								$(0.2755, 0.3177]$							
$M(\sigma_i)$	7	6	5	4	3	2	1	0	6	5	4	3	2	1	1	0	5	4	3	2	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	2	0	1	1	1	1	1	2	2
$\ell(k(\sigma_i))$	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

Name	V2F3_4								V2F3_5							
I_k	$(0.3177, 0.4302]$								$(0.4302, 0.5]$							
$M(\sigma_i)$	4	3	2	2	2	1	1	0	3	2	2	2	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	0	1	1	1	2	2	2	3
$\ell(k(\sigma_i))$	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

Table A.11: Optimal Tunstall codes (V2F) with code word length of 3.

Name	V2F4_1															
I_k	$(0, 0.1338]$															
$M(\sigma_i)$	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

Name	V2F4_2															
I_k	$(0.1338, 0.1474]$															
$M(\sigma_i)$	14	13	12	11	10	9	8	7	6	5	4	3	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2
$\ell(k(\sigma_i))$	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

Table A.12: Optimal Tunstall codes (V2F) with code word length of 4 (part 1).

Name	V2F4_3															
I_k	$(0.1474, 0.1556]$															
$M(\sigma_i)$	13	12	11	10	9	8	7	6	5	4	3	2	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2
$\ell(k(\sigma_i))$	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

Name	V2F4_4															
I_k	$(0.1556, 0.1757]$															
$M(\sigma_i)$	12	11	10	9	8	7	6	5	4	3	2	2	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2
$\ell(k(\sigma_i))$	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

Name	V2F4_5															
I_k	$(0.1757, 0.1883]$															
$M(\sigma_i)$	11	10	9	8	7	6	5	4	3	3	2	2	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2
$\ell(k(\sigma_i))$	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

Name	V2F4_6															
I_k	$(0.1883, 0.2035]$															
$M(\sigma_i)$	10	9	8	7	6	5	4	3	3	3	2	2	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2
$\ell(k(\sigma_i))$	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

Name	V2F4_7															
I_k	$(0.2035, 0.2451]$															
$M(\sigma_i)$	9	8	7	6	5	4	3	3	3	3	2	2	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2
$\ell(k(\sigma_i))$	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

Name	V2F4_8															
I_k	$(0.2451, 0.2755]$															
$M(\sigma_i)$	8	7	6	5	4	4	3	3	3	3	2	2	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

Table A.13: Optimal Tunstall codes (V2F) with code word length of 4 (part 2).

Name	V2F4_9															
I_k	$(0.2755, 0.3177]$															
$M(\sigma_i)$	7	6	5	4	4	4	3	3	3	3	2	2	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
Name	V2F4_10															
I_k	$(0.3177, 0.382]$															
$M(\sigma_i)$	6	5	4	4	4	3	3	3	3	2	2	2	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	1	2	2	2	2	2	2	2	2	3
$\ell(k(\sigma_i))$	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
Name	V2F4_11															
I_k	$(0.382, 0.4503]$															
$M(\sigma_i)$	5	4	3	3	3	3	2	2	2	2	2	2	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3
$\ell(k(\sigma_i))$	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
Name	V2F4_12															
I_k	$(0.4503, 0.5]$															
$M(\sigma_i)$	4	3	3	3	3	2	2	2	2	2	2	1	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	2	3	3	3	3	4
$\ell(k(\sigma_i))$	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

Table A.14: Optimal Tunstall codes (V2F) with code word length of 4 (part 3).

Name	V2F5_1																										
I_k	$(0, 0.0805]$																										
$M(\sigma_i)$	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	4	3	2	1	0																						
$N(\sigma_i)$	1	1	1	1	1																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						

Name	V2F5_2																										
I_k	$(0.0805, 0.0845]$																										
$M(\sigma_i)$	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	3	2	1	1	0																						
$N(\sigma_i)$	1	1	1	1	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						

Name	V2F5_3																										
I_k	$(0.0845, 0.0866]$																										
$M(\sigma_i)$	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	1	1	1	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						

Name	V2F5_4																										
I_k	$(0.0866, 0.0913]$																										
$M(\sigma_i)$	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	1	1	2	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						

Table A.15: Optimal Tunstall codes (V2F) with code word length of 5 (part 1).

Name	V2F5_5																									
I_k	$(0.0913, 0.0939]$																									
$M(\sigma_i)$	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	3	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	1	2	2	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						

Name	V2F5_6																									
I_k	$(0.0939, 0.0966]$																									
$M(\sigma_i)$	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	3	3	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	2	2	2	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						

Name	V2F5_7																									
I_k	$(0.0966, 0.1027]$																									
$M(\sigma_i)$	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	3	3	3	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	2	2	2	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						

Name	V2F5_8																									
I_k	$(0.1027, 0.1061]$																									
$M(\sigma_i)$	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	4	3	3	3	3	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	2	2	2	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						

Table A.16: Optimal Tunstall codes (V2F) with code word length of 5 (part 2).

Name	V2F5_9																										
I_k	$(0.1061, 0.1098]$																										
$M(\sigma_i)$	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	4	4	3	3	3	3	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	2	2	2	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						

Name	V2F5_10																										
I_k	$(0.1098, 0.1138]$																										
$M(\sigma_i)$	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	4	4	4	3	3	3	3	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	2	2	2	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						

Name	V2F5_11																										
I_k	$(0.1138, 0.1228]$																										
$M(\sigma_i)$	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	4	4	4	4	3	3	3	3	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	2	2	2	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						

Name	V2F5_12																										
I_k	$(0.1228, 0.128]$																										
$M(\sigma_i)$	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	5	4	4	4	4	4	3	3	3	3	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	2	2	2	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						

Table A.17: Optimal Tunstall codes (V2F) with code word length of 5 (part 3).

Name	V2F5_13																										
I_k	$(0.128, 0.1338]$																										
$M(\sigma_i)$	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	5	5	4	4	4	4	4	3	3	3	3	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	2	2	2	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						
Name	V2F5_14																										
I_k	$(0.1338, 0.1402]$																										
$M(\sigma_i)$	18	17	16	15	14	13	12	11	10	9	8	7	6	5	5	5	5	4	4	4	4	4	3	3	3	3	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	2	2	2	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						
Name	V2F5_15																										
I_k	$(0.1402, 0.1474]$																										
$M(\sigma_i)$	17	16	15	14	13	12	11	10	9	8	7	6	5	5	5	5	5	4	4	4	4	4	3	3	3	3	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	2	2	2	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						
Name	V2F5_16																										
I_k	$(0.1474, 0.1649]$																										
$M(\sigma_i)$	16	15	14	13	12	11	10	9	8	7	6	5	5	5	5	5	5	4	4	4	4	4	3	3	3	3	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	2	2	2	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						

Table A.18: Optimal Tunstall codes (V2F) with code word length of 5 (part 4).

Name	V2F5_17																										
I_k	$(0.1649, 0.1757]$																										
$M(\sigma_i)$	15	14	13	12	11	10	9	8	7	6	6	5	5	5	5	5	5	4	4	4	4	4	3	3	3	3	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	2	2	2	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						
Name	V2F5_18																										
I_k	$(0.1757, 0.1883]$																										
$M(\sigma_i)$	14	13	12	11	10	9	8	7	6	6	6	5	5	5	5	5	5	4	4	4	4	4	3	3	3	3	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	2	2	2	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						
Name	V2F5_19																										
I_k	$(0.1883, 0.2035]$																										
$M(\sigma_i)$	13	12	11	10	9	8	7	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4	3	3	3	3	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	2	2	2	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						
Name	V2F5_20																										
I_k	$(0.2035, 0.2219]$																										
$M(\sigma_i)$	12	11	10	9	8	7	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4	3	3	3	3	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	2	2	2	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						

Table A.19: Optimal Tunstall codes (V2F) with code word length of 5 (part 5).

Name	V2F5_21																										
I_k	$(0.2219, 0.2451]$																										
$M(\sigma_i)$	11	10	9	8	7	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4	3	3	3	3	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	2	1	1	0																						
$N(\sigma_i)$	2	2	2	2	2																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						

Name	V2F5_22																										
I_k	$(0.2451, 0.2755]$																										
$M(\sigma_i)$	10	9	8	7	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4	3	3	3	3	2	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	1	1	1	0																						
$N(\sigma_i)$	2	2	2	2	3																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						

Name	V2F5_23																										
I_k	$(0.2755, 0.3177]$																										
$M(\sigma_i)$	9	8	7	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4	3	3	3	3	2	2	2	2	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	1	1	1	0																						
$N(\sigma_i)$	2	3	3	3	3																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						

Name	V2F5_24																										
I_k	$(0.3177, 0.346]$																										
$M(\sigma_i)$	8	7	6	5	5	5	5	5	5	4	4	4	4	4	3	3	3	3	3	3	3	3	2	2	2	2	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	1	1	1	0																						
$N(\sigma_i)$	3	3	3	3	3																						
$\ell(k(\sigma_i))$	5	5	5	5	5																						

Table A.20: Optimal Tunstall codes (V2F) with code word length of 5 (part 6).

Name	V2F5_25																									
I_k	$(0.346, 0.382]$																									
$M(\sigma_i)$	7	6	5	5	5	5	5	5	4	4	4	4	4	3	3	3	3	3	3	3	3	2	2	2	2	2
$N(\sigma_i)$	0	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	2	1	1	1	0																					
$N(\sigma_i)$	3	3	3	3	3																					
$\ell(k(\sigma_i))$	5	5	5	5	5																					

Name	V2F5_26																									
I_k	$(0.382, 0.4302]$																									
$M(\sigma_i)$	6	5	5	5	5	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2
$N(\sigma_i)$	0	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	1	1	1	1	0																					
$N(\sigma_i)$	3	3	3	3	4																					
$\ell(k(\sigma_i))$	5	5	5	5	5																					

Name	V2F5_27																									
I_k	$(0.4302, 0.4614]$																									
$M(\sigma_i)$	6	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	2
$N(\sigma_i)$	0	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	1	1	1	1	0																					
$N(\sigma_i)$	4	4	4	4	4																					
$\ell(k(\sigma_i))$	5	5	5	5	5																					

Name	V2F5_28																									
I_k	$(0.4614, 0.5]$																									
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1	
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4	
$\ell(k(\sigma_i))$	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
$M(\sigma_i)$	1	1	1	1	0																					
$N(\sigma_i)$	4	4	4	4	5																					
$\ell(k(\sigma_i))$	5	5	5	5	5																					

Table A.21: Optimal Tunstall codes (V2F) with code word length of 5 (part 7).

Name	V2F6_1																									
I_k	$(0, 0.0478]$																									
$M(\sigma_i)$	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	9	8	7	6	5	4	3	2	1	0																	
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_2																									
I_k	$(0.0478, 0.049]$																									
$M(\sigma_i)$	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	6	5	4	3	2	1	1	0																	
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_3																									
I_k	$(0.049, 0.0496]$																									
$M(\sigma_i)$	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	7	6	5	4	3	2	2	1	1	0																	
$N(\sigma_i)$	1	1	1	1	1	1	1	1	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Table A.22: Optimal Tunstall codes (V2F) with code word length of 6 (part 1).

Name	V2F6_4																										
I_k	$(0.0496, 0.0509]$																										
$M(\sigma_i)$	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	6	5	4	3	2	2	2	1	1	0																	
$N(\sigma_i)$	1	1	1	1	1	1	1	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_5																										
I_k	$(0.0509, 0.0516]$																										
$M(\sigma_i)$	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	5	4	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	1	1	1	1	1	1	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_6																										
I_k	$(0.0516, 0.0523]$																										
$M(\sigma_i)$	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	4	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	1	1	1	1	1	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Table A.23: Optimal Tunstall codes (V2F) with code word length of 6 (part 2).

Name	V2F6_7																										
I_k	$(0.0523, 0.0537]$																										
$M(\sigma_i)$	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	1	1	1	1	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_8																										
I_k	$(0.0537, 0.0544]$																										
$M(\sigma_i)$	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	1	1	1	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_9																										
I_k	$(0.0544, 0.0552]$																										
$M(\sigma_i)$	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	1	1	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Table A.24: Optimal Tunstall codes (V2F) with code word length of 6 (part 3).

Name	V2F6_10																										
I_k	$(0.0552, 0.056]$																										
$M(\sigma_i)$	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	1	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_11																										
I_k	$(0.056, 0.0577]$																										
$M(\sigma_i)$	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_12																										
I_k	$(0.0577, 0.0586]$																										
$M(\sigma_i)$	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Table A.25: Optimal Tunstall codes (V2F) with code word length of 6 (part 4).

Name	V2F6_13																									
I_k	$(0.0586, 0.0595]$																									
$M(\sigma_i)$	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_14																									
I_k	$(0.0595, 0.0604]$																									
$M(\sigma_i)$	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_15																									
I_k	$(0.0604, 0.0614]$																									
$M(\sigma_i)$	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Table A.26: Optimal Tunstall codes (V2F) with code word length of 6 (part 5).

Name	V2F6_16																									
I_k	$(0.0614, 0.0635]$																									
$M(\sigma_i)$	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_17																									
I_k	$(0.0635, 0.0646]$																									
$M(\sigma_i)$	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_18																									
I_k	$(0.0646, 0.0658]$																									
$M(\sigma_i)$	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	19	18	17	16	15	14	13	12	11	10	9	8	7	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Table A.27: Optimal Tunstall codes (V2F) with code word length of 6 (part 6).

Name	V2F6_19																									
I_k	$(0.0658, 0.067]$																									
$M(\sigma_i)$	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	18	17	16	15	14	13	12	11	10	9	8	7	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_20																									
I_k	$(0.067, 0.0682]$																									
$M(\sigma_i)$	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	17	16	15	14	13	12	11	10	9	8	7	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_21																									
I_k	$(0.0682, 0.0695]$																									
$M(\sigma_i)$	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	16	15	14	13	12	11	10	9	8	7	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Table A.28: Optimal Tunstall codes (V2F) with code word length of 6 (part 7).

Name	V2F6_22																										
I_k	$(0.0695, 0.0723]$																										
$M(\sigma_i)$	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	15	14	13	12	11	10	9	8	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_23																										
I_k	$(0.0723, 0.0738]$																										
$M(\sigma_i)$	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	14	13	12	11	10	9	8	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_24																										
I_k	$(0.0738, 0.0754]$																										
$M(\sigma_i)$	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	13	12	11	10	9	8	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Table A.29: Optimal Tunstall codes (V2F) with code word length of 6 (part 8).

Name	V2F6_25																									
I_k	$(0.0754, 0.077]$																									
$M(\sigma_i)$	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	12	11	10	9	8	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_26																									
I_k	$(0.077, 0.0787]$																									
$M(\sigma_i)$	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	11	10	9	8	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_27																									
I_k	$(0.0787, 0.0805]$																									
$M(\sigma_i)$	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	10	9	8	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Table A.30: Optimal Tunstall codes (V2F) with code word length of 6 (part 9).

Name	V2F6_28																									
I_k	$(0.0805, 0.0825]$																									
$M(\sigma_i)$	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	9	8	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_29																									
I_k	$(0.0825, 0.0866]$																									
$M(\sigma_i)$	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_30																									
I_k	$(0.0866, 0.0889]$																									
$M(\sigma_i)$	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Table A.31: Optimal Tunstall codes (V2F) with code word length of 6 (part 10).

Name	V2F6_31																									
I_k	$(0.0889, 0.0913]$																									
$M(\sigma_i)$	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	8
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_32																									
I_k	$(0.0913, 0.0939]$																									
$M(\sigma_i)$	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	8	8
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_33																									
I_k	$(0.0939, 0.0966]$																									
$M(\sigma_i)$	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	8	8	8
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Table A.32: Optimal Tunstall codes (V2F) with code word length of 6 (part 11).

Name	V2F6_34																									
I_k	$(0.0966, 0.0996]$																									
$M(\sigma_i)$	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	8	8	8
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4
$N(\sigma_i)$	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																

Name	V2F6_35																									
I_k	$(0.0996, 0.1027]$																									
$M(\sigma_i)$	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	8	8	8	8
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4
$N(\sigma_i)$	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																

Name	V2F6_36																									
I_k	$(0.1027, 0.1061]$																									
$M(\sigma_i)$	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	8	8	8	8	8
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4
$N(\sigma_i)$	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																

Table A.33: Optimal Tunstall codes (V2F) with code word length of 6 (part 12).

Name	V2F6_37																									
I_k	$(0.1061, 0.1138]$																									
$M(\sigma_i)$	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	8	8	8	8	8	8
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																

Name	V2F6_38																									
I_k	$(0.1138, 0.1181]$																									
$M(\sigma_i)$	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	9	8	8	8	8	8	8	8
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																

Name	V2F6_39																									
I_k	$(0.1181, 0.1228]$																									
$M(\sigma_i)$	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	9	9	8	8	8	8	8	8	8
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																

Table A.34: Optimal Tunstall codes (V2F) with code word length of 6 (part 13).

Name	V2F6_40																									
I_k	$(0.1228, 0.128]$																									
$M(\sigma_i)$	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	9	9	9	8	8	8	8	8	8	8
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																

Name	V2F6_41																									
I_k	$(0.128, 0.1338]$																									
$M(\sigma_i)$	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	9	9	9	9	8	8	8	8	8	8	8
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																

Name	V2F6_42																									
I_k	$(0.1338, 0.1402]$																									
$M(\sigma_i)$	22	21	20	19	18	17	16	15	14	13	12	11	10	9	9	9	9	9	9	8	8	8	8	8	8	8
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																

Table A.35: Optimal Tunstall codes (V2F) with code word length of 6 (part 14).

Name	V2F6_43																									
I_k	$(0.1402, 0.1474]$																									
$M(\sigma_i)$	21	20	19	18	17	16	15	14	13	12	11	10	9	9	9	9	9	9	9	8	8	8	8	8	8	8
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_44																									
I_k	$(0.1474, 0.1556]$																									
$M(\sigma_i)$	20	19	18	17	16	15	14	13	12	11	10	9	9	9	9	9	9	9	9	8	8	8	8	8	8	8
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_45																									
I_k	$(0.1556, 0.1649]$																									
$M(\sigma_i)$	19	18	17	16	15	14	13	12	11	10	9	9	9	9	9	9	9	9	8	8	8	8	8	8	8	8
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Table A.36: Optimal Tunstall codes (V2F) with code word length of 6 (part 15).

Name	V2F6_46																									
I_k	$(0.1649, 0.1818]$																									
$M(\sigma_i)$	18	17	16	15	14	13	12	11	10	9	9	9	9	9	9	9	9	9	8	8	8	8	8	8	8	8
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	2	2	2	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_47																									
I_k	$(0.1818, 0.1955]$																									
$M(\sigma_i)$	17	16	15	14	13	12	11	10	9	9	9	9	9	9	9	9	9	8	8	8	8	8	8	8	8	8
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4	3
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	2	2	2	1	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	3																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_48																									
I_k	$(0.1955, 0.2035]$																									
$M(\sigma_i)$	16	15	14	13	12	11	10	9	9	9	9	9	9	9	9	9	8	8	8	8	8	8	8	8	7	
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4	3	3
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	2	2	2	2	1	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	2	2	3	3																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Table A.37: Optimal Tunstall codes (V2F) with code word length of 6 (part 16).

Name	V2F6_49																								
I_k	$(0.2035, 0.2219]$																								
$M(\sigma_i)$	15	14	13	12	11	10	9	9	9	9	9	9	9	9	9	8	8	8	8	8	8	8	8	7	7	7
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4	3	3	3	3
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	2	2	2	2	2	2	1	1	1	0																	
$N(\sigma_i)$	2	2	2	2	2	2	3	3	3	3																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_50																								
I_k	$(0.2219, 0.2451]$																								
$M(\sigma_i)$	14	13	12	11	10	9	9	9	9	8	8	8	8	8	8	8	8	7	7	7	7	7	7	7	6
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	2	2	2	2	2	2	1	1	1	0																	
$N(\sigma_i)$	3	3	3	3	3	3	3	3	3	3																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_51																								
I_k	$(0.2451, 0.2592]$																								
$M(\sigma_i)$	13	12	11	10	9	8	8	8	8	8	8	8	8	8	7	7	7	7	7	7	7	7	6	6	6	6	6
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	6	6	5	5	5	5	5	5	4	4	4	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	3
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	2	2	2	2	2	2	1	1	1	0																	
$N(\sigma_i)$	3	3	3	3	3	3	3	3	3	3																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Table A.38: Optimal Tunstall codes (V2F) with code word length of 6 (part 17).

Name	V2F6_52																									
I_k	$(0.2592, 0.2755]$																									
$M(\sigma_i)$	12	11	10	9	8	8	8	8	8	8	8	8	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	6	5	5	5	5	5	5	4	4	4	4	4	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	2	2	2	2	2	2	1	1	1	0																	
$N(\sigma_i)$	3	3	3	3	3	3	3	3	3	3																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_53																									
I_k	$(0.2755, 0.2947]$																									
$M(\sigma_i)$	11	10	9	8	8	8	8	8	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	5	5	4	4	4	4	4	4	4	4	4	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	2	2	2	2	2	2	1	1	1	0																
$N(\sigma_i)$	3	3	3	3	3	3	3	3	3	3																
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																

Name	V2F6_54																									
I_k	$(0.2947, 0.3177]$																									
$M(\sigma_i)$	11	10	9	8	8	8	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	5	4	4	4	4	4	4	4	4	4	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3	3
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	2	2	2	2	2	1	1	1	1	0																
$N(\sigma_i)$	3	3	3	3	3	3	3	3	3	4																
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																

Table A.39: Optimal Tunstall codes (V2F) with code word length of 6 (part 18).

Name	V2F6_55																									
I_k	$(0.3177, 0.346]$																									
$M(\sigma_i)$	10	9	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	4	4	4	4	4	4	4	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	2	2	2	2	2	1	1	1	1	0																	
$N(\sigma_i)$	3	3	3	3	3	4	4	4	4	4																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_56																									
I_k	$(0.346, 0.382]$																									
$M(\sigma_i)$	9	8	7	7	7	7	7	7	7	7	6	6	6	6	6	6	6	5	5	5	5	5	5	5	4	4	4
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	4	4	4	4	4	4	4	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2
$N(\sigma_i)$	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	2	2	2	2	2	1	1	1	1	0																	
$N(\sigma_i)$	3	3	3	3	3	4	4	4	4	4																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_57																									
I_k	$(0.382, 0.4123]$																									
$M(\sigma_i)$	8	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4	4	4	4	4	4	4	4
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	4	4	4	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2
$N(\sigma_i)$	2	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	4	4	4	4
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	2	2	2	2	1	1	1	1	1	0																	
$N(\sigma_i)$	4	4	4	4	4	4	4	4	4	5																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Table A.40: Optimal Tunstall codes (V2F) with code word length of 6 (part 19).

Name	V2F6_58																										
I_k	$(0.4123, 0.4302]$																										
$M(\sigma_i)$	7	6	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4	4	4	4	4	4	4	4	4
$N(\sigma_i)$	0	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	4	4	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	
$N(\sigma_i)$	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	4	4	4	4	4	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	
$M(\sigma_i)$	2	2	2	2	1	1	1	1	1	0																	
$N(\sigma_i)$	4	4	4	4	4	4	4	4	4	5																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_59																										
I_k	$(0.4302, 0.4503]$																										
$M(\sigma_i)$	7	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4	4	4	4	4	4	4	4	4	4
$N(\sigma_i)$	0	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	4	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	
$N(\sigma_i)$	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	4	4	4	4	4	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	
$M(\sigma_i)$	2	2	2	2	1	1	1	1	1	0																	
$N(\sigma_i)$	4	4	4	4	4	4	4	4	4	5																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Name	V2F6_60																										
I_k	$(0.4503, 0.4684]$																										
$M(\sigma_i)$	7	6	5	5	5	5	5	5	4	4	4	4	4	4	4	4	4	4	4	4	4	4	3	3	3	3	
$N(\sigma_i)$	0	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	
$M(\sigma_i)$	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	2	
$N(\sigma_i)$	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	4	4	4	4	4	4	4	4	4	4	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	
$M(\sigma_i)$	2	2	2	2	1	1	1	1	1	0																	
$N(\sigma_i)$	4	4	4	4	5	5	5	5	5	5																	
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																	

Table A.41: Optimal Tunstall codes (V2F) with code word length of 6 (part 20).

Name	V2F6_61																								
I_k	$(0.4684, 0.5]$																								
$M(\sigma_i)$	6	5	5	5	5	5	5	4	4	4	4	4	4	4	4	4	4	4	4	4	4	3	3	3	3	3
$N(\sigma_i)$	0	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	2	2
$N(\sigma_i)$	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	4	4	4	4	4	4	4	4	4	4	4
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	2	2	2	1	1	1	1	1	1	0																
$N(\sigma_i)$	4	4	4	5	5	5	5	5	5	6																
$\ell(k(\sigma_i))$	6	6	6	6	6	6	6	6	6	6																

Table A.42: Optimal Tunstall codes (V2F) with code word length of 6 (part 21).

A.1.3 Leaf-limited V2V codes

Name	L3_1			L3_2	
I_k	$(0, 0.38]$			$(0.38, 0.5]$	
$M(\sigma_i)$	2	1	0	1	0
$N(\sigma_i)$	0	1	1	0	1
$\ell(k(\sigma_i))$	1	2	2	1	1

Table A.43: Optimal LV2V codes with up to 3 source/code words.

Name	L4_1				L4_2			L4_3	
I_k	$(0, 0.25]$				$(0.25, 0.38]$			$(0.38, 0.5]$	
$M(\sigma_i)$	3	2	1	0	2	1	0	1	0
$N(\sigma_i)$	0	1	1	1	0	1	1	0	1
$\ell(k(\sigma_i))$	1	3	3	2	1	2	2	1	1

Table A.44: Optimal LV2V codes with up to 4 source/code words.

Name	L5_1					L5_2				L5_3			L5_4					L5_5	
I_k	$(0, 0.181]$					$(0.18, 0.25]$				$(0.25, 0.32]$			$(0.318, 0.43]$					$(0.43, 0.5]$	
$M(\sigma_i)$	4	3	2	1	0	3	2	1	0	2	1	0	3	2	1	1	0	1	0
$N(\sigma_i)$	0	1	1	1	1	0	1	1	1	0	1	1	0	1	1	1	2	0	1
$\ell(k(\sigma_i))$	1	3	3	3	3	1	3	3	2	1	2	2	2	3	2	2	3	1	1

Table A.45: Optimal LV2V codes with up to 5 source/code words.

Name	L6_1						L6_2					L6_3				L6_4			L6_5					
I_k	$(0, 0.1433]$						$(0.143, 0.181]$					$(0.18, 0.25]$				$(0.25, 0.29]$			$(0.2929, 0.3333]$					
$M(\sigma_i)$	5	4	3	2	1	0	4	3	2	1	0	3	2	1	0	2	1	0	2	2	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1	0	1	1	2	2	2
$\ell(k(\sigma_i))$	1	4	4	3	3	3	1	3	3	3	3	1	3	3	2	1	2	2	1	3	3	4	4	3

Name	L6_6					L6_7	
I_k	$(0.333, 0.43]$					$(0.43, 0.5]$	
$M(\sigma_i)$	3	2	1	1	0	1	0
$N(\sigma_i)$	0	1	1	1	2	0	1
$\ell(k(\sigma_i))$	2	3	2	2	3	1	1

Table A.46: Optimal LV2V codes with up to 6 source/code words.

Name	L7_1							L7_2						L7_3					L7_4						
I_k	$(0, 0.1187]$							$(0.1187, 0.1433]$						$(0.143, 0.156]$					$(0.1561, 0.1814]$						
$M(\sigma_i)$	6	5	4	3	2	1	0	5	4	3	2	1	0	4	3	2	1	0	4	4	3	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0	1	2	1	1	1	2
$\ell(k(\sigma_i))$	1	4	4	4	4	3	3	1	4	4	3	3	3	1	3	3	3	3	1	4	5	3	3	3	5

Name	L7_5							L7_6							L7_7			L7_8						
I_k	$(0.1814, 0.235]$							$(0.235, 0.2755]$							$(0.28, 0.29]$			$(0.2929, 0.3333]$						
$M(\sigma_i)$	3	2	2	2	1	1	0	5	4	3	2	2	1	0	2	1	0	2	2	2	1	1	0
$N(\sigma_i)$	0	1	1	1	2	2	2	0	1	1	1	1	2	1	0	1	1	0	1	1	2	2	2
$\ell(k(\sigma_i))$	1	3	3	3	5	5	4	2	4	3	3	3	4	2	1	2	2	1	3	3	4	4	3

Table A.47: Optimal LV2V codes with up to 7 source/code words (part 1).

Name	L7_9					L7_10							L7_11	
I_k	$(0.333, 0.399]$					$(0.3992, 0.4503]$							$(0.45, 0.5]$	
$M(\sigma_i)$	3	2	1	1	0	4	3	2	1	1	1	0	1	0
$N(\sigma_i)$	0	1	1	1	2	0	1	1	1	1	2	3	0	1
$\ell(k(\sigma_i))$	2	3	2	2	3	3	4	3	2	2	3	4	1	1

Table A.48: Optimal LV2V codes with up to 7 source/code words (part 2).

Name	L8_1								L8_2							L8_3						
I_k	$(0, 0.1013]$								$(0.1013, 0.1187]$							$(0.1187, 0.1383]$						
$M(\sigma_i)$	7	6	5	4	3	2	1	0	6	5	4	3	2	1	0	5	4	3	2	1	0	
$N(\sigma_i)$	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	
$\ell(k(\sigma_i))$	1	4	4	4	4	4	4	3	1	4	4	4	4	3	3	1	4	4	3	3	3	

Name	L8_4								L8_5								L8_6							
I_k	$(0.1383, 0.1823]$								$(0.1823, 0.2048]$								$(0.2048, 0.2314]$							
$M(\sigma_i)$	5	4	4	3	2	1	1	0	3	3	2	2	2	1	1	0	3	2	2	2	1	1	0	
$N(\sigma_i)$	1	0	2	2	1	1	1	2	0	1	1	1	2	2	2	2	0	1	1	1	2	2	2	
$\ell(k(\sigma_i))$	4	1	6	6	3	3	3	5	1	3	3	3	5	5	5	5	1	3	3	3	5	5	4	

Name	L8_7								L8_8								L8_9							
I_k	$(0.2314, 0.2431]$								$(0.2431, 0.266]$								$(0.266, 0.2752]$							
$M(\sigma_i)$	5	5	4	3	2	2	1	0	5	4	3	2	2	1	0		3	3	2	2	2	1	1	0
$N(\sigma_i)$	0	1	2	1	1	1	2	1	0	1	1	1	1	2	1		1	1	0	2	2	2	2	2
$\ell(k(\sigma_i))$	2	4	5	3	3	3	5	2	2	4	3	3	3	4	2		3	3	1	5	5	4	4	4

Name	L8_10								L8_11							L8_12								L8_13				
I_k	$(0.2752, 0.314]$								$(0.314, 0.3262]$							$(0.3262, 0.339]$								$(0.339, 0.399]$				
$M(\sigma_i)$	3	2	2	2	1	1	1	0	2	2	2	1	1	0		5	4	3	2	2	1	1	0	3	2	1	1	0
$N(\sigma_i)$	1	0	1	2	2	2	2	3	0	1	1	2	2	2		0	1	1	1	1	1	2	2	0	1	1	1	2
$\ell(k(\sigma_i))$	3	1	3	5	4	4	4	5	1	3	3	4	4	3		3	4	3	3	3	2	4	3	2	3	2	2	3

Name	L8_14							L8_15	
I_k	$(0.3992, 0.4503]$							$(0.45, 0.5]$	
$M(\sigma_i)$	4	3	2	1	1	1	0	1	0
$N(\sigma_i)$	0	1	1	1	1	2	3	0	1
$\ell(k(\sigma_i))$	3	4	3	2	2	3	4	1	1

Table A.49: Optimal LV2V codes with up to 8 source/code words.

Name	L9_1									L9_2								L9_3						
I_k	$(0, 0.0884]$									$(0.0884, 0.1013]$								$(0.1013, 0.1187]$						
$M(\sigma_i)$	8	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	6	5	4	3	2	1	0
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1
$\ell(k(\sigma_i))$	1	4	4	4	4	4	4	4	4	1	4	4	4	4	4	4	3	1	4	4	4	4	3	3

Name	L9_4						L9_5								L9_6							
I_k	$(0.1187, 0.1373]$						$(0.1373, 0.1428]$								$(0.1428, 0.1823]$							
$M(\sigma_i)$	5	4	3	2	1	0	6	5	4	4	3	2	1	1	0	5	4	4	3	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	1	2	0	2	2	1	1	1	2	1	0	2	2	1	1	1	2
$\ell(k(\sigma_i))$	1	4	4	3	3	3	4	6	1	6	6	3	3	3	6	4	1	6	6	3	3	3	5

Name	L9_7								L9_8							L9_9								
I_k	$(0.1823, 0.2048]$								$(0.2048, 0.2266]$							$(0.2266, 0.2681]$								
$M(\sigma_i)$	3	3	2	2	2	1	1	0	3	2	2	2	1	1	0	5	5	4	3	2	2	2	1	0
$N(\sigma_i)$	0	1	1	1	2	2	2	2	0	1	1	1	2	2	2	0	1	2	1	1	1	2	3	1
$\ell(k(\sigma_i))$	1	3	3	3	5	5	5	5	1	3	3	3	5	5	4	2	4	6	3	3	3	5	6	2

Name	L9_10									L9_11								L9_12								
I_k	$(0.2681, 0.2921]$									$(0.2921, 0.2941]$								$(0.2941, 0.3223]$								
$M(\sigma_i)$	3	3	2	2	2	1	1	1	0	3	2	2	2	1	1	1	0	4	3	2	2	2	1	1	1	0
$N(\sigma_i)$	1	2	0	1	3	2	2	2	3	1	0	1	2	2	2	2	3	1	2	0	1	2	2	2	2	3
$\ell(k(\sigma_i))$	3	5	1	3	6	4	4	4	6	3	1	3	5	4	4	4	5	4	5	1	3	4	4	4	4	5

Name	L9_13									L9_14					L9_15									
I_k	$(0.3223, 0.346]$									$(0.346, 0.382]$					$(0.382, 0.4253]$									
$M(\sigma_i)$	5	4	4	3	2	1	1	1	0	3	2	1	1	0	4	4	3	2	2	1	1	1	0	
$N(\sigma_i)$	0	1	1	2	1	1	1	2	3	0	1	1	1	2	0	1	2	1	2	1	1	3	3	
$\ell(k(\sigma_i))$	3	4	4	5	3	2	2	4	5	2	3	2	2	3	3	4	5	3	4	2	2	5	4	

Name	L9_16							L9_17								L9_18	
I_k	$(0.4253, 0.4334]$							$(0.4334, 0.4614]$								$(0.46, 0.5]$	
$M(\sigma_i)$	4	3	2	1	1	1	0	5	4	3	2	1	1	1	1	0	1	0
$N(\sigma_i)$	0	1	1	1	1	2	3	0	1	1	1	1	1	2	3	4	0	1
$\ell(k(\sigma_i))$	3	4	3	2	2	3	4	4	5	4	3	2	2	3	4	5	1	1

Table A.50: Optimal LV2V codes with up to 9 source/code words.

A.1.4 Source-height-limited V2V codes

Name	S2_1			S2_2	
I_k	$(0, 0.38]$			$(0.38, 0.5]$	
$M(\sigma_i)$	2	1	0	1	0
$N(\sigma_i)$	0	1	1	0	1
$\ell(k(\sigma_i))$	1	2	2	1	1

Table A.51: Optimal SV2V codes with maximum source tree height of 2.

Name	S3_1							S3_2			S3_3						S3_4					S3_5	
I_k	$(0, 0.2541]$							$(0.25, 0.29]$			$(0.2929, 0.3333]$						$(0.333, 0.43]$					$(0.43, 0.5]$	
$M(\sigma_i)$	3	2	2	2	1	1	0	2	1	0	2	2	2	1	1	0	3	2	1	1	0	1	0
$N(\sigma_i)$	0	1	1	1	2	2	2	0	1	1	0	1	1	2	2	2	0	1	1	1	2	0	1
$\ell(k(\sigma_i))$	1	3	3	3	5	5	4	1	2	2	1	3	3	4	4	3	2	3	2	2	3	1	1

Table A.52: Optimal SV2V codes with maximum source tree height of 3.

Name	S4_1											S4_2				
I_k	$(0, 0.0864]$											$(0.086, 0.153]$				
$M(\sigma_i)$	4	3	3	3	3	2	2	2	1	1	0	4	3	2	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	2	0	1	1	1	1
$\ell(k(\sigma_i))$	1	3	3	3	4	7	7	7	6	7	6	1	3	3	3	3

Name	S4_3												S4_4							
I_k	$(0.1528, 0.1739]$												$(0.1739, 0.191]$							
$M(\sigma_i)$	4	3	2	2	2	2	2	2	1	1	1	0	3	3	2	2	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	3	3	3	3	0	1	1	1	2	2	2	2
$\ell(k(\sigma_i))$	1	4	3	3	3	6	6	6	8	8	8	8	1	3	3	3	5	5	5	5

Name	S4_5											S4_6									
I_k	$(0.191, 0.2419]$											$(0.2419, 0.2485]$									
$M(\sigma_i)$	3	2	2	2	2	2	2	1	1	1	0	3	2	2	2	2	2	1	1	1	0
$N(\sigma_i)$	0	1	1	1	2	2	2	3	3	3	3	0	1	1	1	2	2	2	3	3	3
$\ell(k(\sigma_i))$	1	3	3	3	5	5	5	7	7	7	7	1	3	3	3	5	5	5	7	7	6

Table A.53: Optimal SV2V codes with maximum source tree height of 4 (part 1).

Name	S4_7												S4_8							
I_k	$(0.2485, 0.2526]$												$(0.2526, 0.2752]$							
$M(\sigma_i)$	4	3	3	3	3	2	2	2	2	2	1	1	0	3	3	2	2	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	3	3	2	1	1	0	2	2	2	2	2
$\ell(k(\sigma_i))$	2	3	3	3	3	5	5	5	5	5	6	6	4	3	3	1	5	5	4	4	4

Name	S4_9								S4_10						S4_11				
I_k	$(0.2752, 0.314]$								$(0.314, 0.3333]$						$(0.333, 0.382]$				
$M(\sigma_i)$	3	2	2	2	1	1	1	0	2	2	2	1	1	0	3	2	1	1	0
$N(\sigma_i)$	1	0	1	2	2	2	2	3	0	1	1	2	2	2	0	1	1	1	2
$\ell(k(\sigma_i))$	3	1	3	5	4	4	4	5	1	3	3	4	4	3	2	3	2	2	3

Name	S4_12									S4_13							S4_14	
I_k	$(0.382, 0.4073]$									$(0.4073, 0.4503]$							$(0.45, 0.5]$	
$M(\sigma_i)$	3	2	2	2	2	1	1	1	0	4	3	2	1	1	1	0	1	0
$N(\sigma_i)$	0	1	1	2	2	1	3	3	3	0	1	1	1	1	2	3	0	1
$\ell(k(\sigma_i))$	2	3	3	4	4	2	5	5	4	3	4	3	2	2	3	4	1	1

Table A.54: Optimal SV2V codes with maximum source tree height of 4 (part 2).

Name	S5_1																								
I_k	$(0, 0.0616]$																								
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3
$\ell(k(\sigma_i))$	1	3	3	4	4	4	7	7	7	7	7	8	8	8	8	8	11	11	12	12	12	12	11	11	11	11

Name	S5_2																				
I_k	$(0.0616, 0.0713]$																				
$M(\sigma_i)$	5	4	4	4	3	3	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3
$\ell(k(\sigma_i))$	1	4	4	4	3	3	7	7	7	7	7	8	7	7	11	11	11	11	11	11	11	11

Name	S5_3																				
I_k	$(0.0713, 0.0899]$																				
$M(\sigma_i)$	5	4	4	4	3	3	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	2	2	1	3	3	3	3	3	3	3	3	3
$\ell(k(\sigma_i))$	1	4	4	4	3	7	7	7	7	7	7	7	3	10	10	10	11	11	10	10	10	10

Table A.55: Optimal SV2V codes with maximum source tree height of 5 (part 1).

Name	S5_4																			
I_k	$(0.0899, 0.0923]$																			
$M(\sigma_i)$	5	4	4	4	3	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	2	1	2	3	3	3	3	3	3	3	3
$\ell(k(\sigma_i))$	1	4	4	4	3	7	7	7	7	7	7	3	7	10	10	10	10	10	10	10	10

Name	S5_5																						
I_k	$(0.0923, 0.0942]$																						
$M(\sigma_i)$	5	4	4	4	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1	1	1	1	0
$N(\sigma_i)$	0	1	1	1	2	2	2	2	2	1	1	2	3	3	3	3	3	3	3	4	4	4	4
$\ell(k(\sigma_i))$	1	4	4	4	7	7	7	7	7	3	3	6	10	10	10	10	10	10	10	12	12	13	13	12

Name	S5_6																
I_k	$(0.0942, 0.1005]$																
$M(\sigma_i)$	5	4	4	4	3	3	3	3	3	2	2	2	2	2	1	1	1	0
$N(\sigma_i)$	0	1	1	1	2	2	2	2	2	1	1	3	3	3	2	3	3	3
$\ell(k(\sigma_i))$	1	4	4	4	7	7	7	7	7	3	3	10	10	10	6	9	10	9

Name	S5_7												S5_8										
I_k	$(0.1005, 0.1107]$												$(0.1107, 0.145]$										
$M(\sigma_i)$	5	4	4	4	3	3	2	2	1	1	0		5	4	4	4	3	3	2	1	1	0	
$N(\sigma_i)$	0	1	1	1	2	2	1	2	1	2	2		0	1	1	1	2	2	2	1	1	2	
$\ell(k(\sigma_i))$	1	4	4	4	7	7	3	6	3	6	6		1	4	4	4	6	6	6	3	3	6	

Name	S5_9													S5_10											
I_k	$(0.145, 0.1575]$													$(0.1575, 0.1735]$											
$M(\sigma_i)$	4	4	3	3	3	2	2	2	2	1	1	1	0	4	4	3	3	2	2	2	2	1	1	1	0
$N(\sigma_i)$	0	1	2	2	2	1	1	3	3	1	3	3	3	0	1	2	2	1	1	2	3	1	3	3	3
$\ell(k(\sigma_i))$	1	4	6	6	6	3	3	9	9	3	8	8	8	1	4	6	6	3	3	6	8	3	8	8	8

Name	S5_11																	S5_12							
I_k	$(0.1735, 0.1811]$																	$(0.1811, 0.1867]$							
$M(\sigma_i)$	4	4	4	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	0	3	3	2	2	2	1	1	0
$N(\sigma_i)$	0	1	1	2	2	2	2	2	2	1	1	3	3	3	3	3	3	3	3	0	1	1	1	2	2	2	2
$\ell(k(\sigma_i))$	1	4	4	6	6	6	6	6	6	3	3	8	8	8	8	8	8	8	8	1	3	3	3	5	5	5	5

Name	S5_13										S5_14										
I_k	$(0.1867, 0.192]$										$(0.192, 0.2324]$										
$M(\sigma_i)$	3	3	3	2	2	2	1	1	1	0	3	2	2	2	2	2	2	1	1	1	0
$N(\sigma_i)$	0	1	2	1	1	3	2	2	2	3	0	1	1	1	2	2	2	3	3	3	3
$\ell(k(\sigma_i))$	1	3	6	3	3	7	5	5	5	7	1	3	3	3	5	5	5	7	7	7	7

Table A.56: Optimal SV2V codes with maximum source tree height of 5 (part 2).

Name	S5_15												
I_k	$(0.2324, 0.2355]$												
$M(\sigma_i)$	3	2	2	2	2	2	2	2	2	1	1	1	1	0
$N(\sigma_i)$	0	1	1	1	2	2	3	3	3	2	4	4	4	4
$\ell(k(\sigma_i))$	1	3	3	3	5	5	7	7	7	5	9	9	9	9

Name	S5_16																	
I_k	$(0.2355, 0.2375]$																	
$M(\sigma_i)$	5	4	3	3	3	3	2	2	2	2	2	2	2	2	2	1	1	1	0
$N(\sigma_i)$	0	1	1	1	2	2	1	1	2	2	3	3	3	3	3	4	4	4	3
$\ell(k(\sigma_i))$	2	4	3	3	5	5	3	3	5	5	7	7	7	7	7	8	9	9	6

Name	S5_17														S5_18											
I_k	$(0.2375, 0.2564]$														$(0.2564, 0.2603]$											
$M(\sigma_i)$	5	4	3	3	3	2	2	2	2	2	2	1	1	0	5	4	4	3	3	3	3	2	2	2	1	0
$N(\sigma_i)$	0	1	1	1	2	1	1	2	2	3	3	3	4	4	2	0	1	1	1	2	2	2	1	3	3	3	1
$\ell(k(\sigma_i))$	2	4	3	3	5	3	3	5	5	7	7	7	8	8	4	2	4	4	3	5	5	5	3	7	7	6	2

Name	S5_19													
I_k	$(0.2603, 0.2707]$													
$M(\sigma_i)$	3	3	3	3	3	3	2	2	2	2	2	1	1	0
$N(\sigma_i)$	1	1	2	2	2	2	0	3	3	3	3	3	3	2
$\ell(k(\sigma_i))$	3	3	5	5	5	5	1	7	7	7	7	6	6	4

Name	S5_20														S5_21										
I_k	$(0.2707, 0.2764]$														$(0.2764, 0.2819]$										
$M(\sigma_i)$	4	3	3	3	3	3	2	2	2	2	1	1	1	1	0	3	3	2	2	2	1	1	1	1	0
$N(\sigma_i)$	1	1	2	2	2	2	0	3	3	3	2	2	3	3	4	1	2	0	1	3	2	2	2	3	4
$\ell(k(\sigma_i))$	4	3	5	5	5	5	1	7	7	7	4	4	6	6	7	3	5	1	3	7	4	4	4	6	7

Name	S5_22												
I_k	$(0.2819, 0.3177]$												
$M(\sigma_i)$	4	4	3	3	3	3	2	2	2	1	1	1	0
$N(\sigma_i)$	1	1	2	2	2	2	0	3	3	2	2	2	3
$\ell(k(\sigma_i))$	4	4	5	5	5	5	1	6	6	4	4	4	5

Name	S5_23																								
I_k	$(0.3177, 0.3258]$																								
$M(\sigma_i)$	4	4	4	4	4	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	2	4	4	3
$\ell(k(\sigma_i))$	2	4	4	4	4	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	4	7	7	5

Table A.57: Optimal SV2V codes with maximum source tree height of 5 (part 3).

Name	S5_24																					
I_k	$(0.3258, 0.3421]$																					
$M(\sigma_i)$	5	4	4	4	4	3	3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	2	2	3	3	3	3	3	3	1	3	4	4	4
$\ell(k(\sigma_i))$	3	4	4	4	4	5	5	5	5	5	5	5	6	6	6	6	6	6	2	5	7	7	6

Name	S5_25																				
I_k	$(0.3421, 0.3586]$																				
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	2	2	2	2	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	2
$\ell(k(\sigma_i))$	3	4	4	4	4	4	5	5	5	5	5	5	5	5	4	6	6	6	6	5	5	3

Name	S5_26												S5_27								
I_k	$(0.3586, 0.382]$												$(0.382, 0.4173]$								
$M(\sigma_i)$	4	4	3	3	3	3	3	2	2	2	1	1	0	4	4	3	2	2	1	1	1	0
$N(\sigma_i)$	1	1	0	2	2	2	2	2	3	3	1	3	2	0	1	2	1	2	1	1	3	3
$\ell(k(\sigma_i))$	4	4	2	5	5	5	5	4	6	6	2	5	3	3	4	5	3	4	2	2	5	4

Name	S5_28											S5_29									S5_30	
I_k	$(0.4173, 0.4413]$											$(0.4413, 0.4614]$									$(0.46, 0.5]$	
$M(\sigma_i)$	4	3	2	2	2	2	1	1	1	1	0	5	4	3	2	1	1	1	1	0	1	0
$N(\sigma_i)$	0	1	1	2	3	3	1	1	4	4	4	0	1	1	1	1	1	2	3	4	0	1
$\ell(k(\sigma_i))$	3	4	3	4	5	5	2	2	6	6	5	4	5	4	3	2	2	3	4	5	1	1

Table A.58: Optimal SV2V codes with maximum source tree height of 5 (part 4).

A.1.5 Code-height-limited V2V codes

Name	C2_1			C2_2	
I_k	$(0, 0.38]$			$(0.38, 0.5]$	
$M(\sigma_i)$	2	1	0	1	0
$N(\sigma_i)$	0	1	1	0	1
$\ell(k(\sigma_i))$	1	2	2	1	1

Table A.59: Optimal CV2V codes with maximum code tree height of 2.

Name	C3_1					C3_2				C3_3			C3_4					C3_5	
I_k	$(0, 0.181]$					$(0.18, 0.25]$				$(0.25, 0.32]$			$(0.318, 0.43]$					$(0.43, 0.5]$	
$M(\sigma_i)$	4	3	2	1	0	3	2	1	0	2	1	0	3	2	1	1	0	1	0
$N(\sigma_i)$	0	1	1	1	1	0	1	1	1	0	1	1	0	1	1	1	2	0	1
$\ell(k(\sigma_i))$	1	3	3	3	3	1	3	3	2	1	2	2	2	3	2	2	3	1	1

Table A.60: Optimal CV2V codes with maximum code tree height of 3.

A.1.6 Source-and-code-height-limited V2V codes

Name	S6C6_1							S6C6_2										
I_k	$(0, 0.1056]$							$(0.1056, 0.1187]$										
$M(\sigma_i)$	6	5	4	3	2	1	0	6	5	4	4	3	3	3	2	2	1	0
$N(\sigma_i)$	0	1	1	1	1	1	1	0	1	1	1	1	1	2	2	2	2	1
$\ell(k(\sigma_i))$	1	4	4	4	4	3	3	1	4	4	4	4	4	6	6	6	6	3

Name	S6C6_3										S6C6_4							
I_k	$(0.1187, 0.1433]$										$(0.1433, 0.1771]$							
$M(\sigma_i)$	5	5	4	4	4	3	3	2	1	0	5	4	4	3	2	1	1	0
$N(\sigma_i)$	0	1	1	1	2	2	2	2	1	1	1	0	2	2	1	1	1	2
$\ell(k(\sigma_i))$	1	4	4	4	6	6	6	6	3	3	4	1	6	6	3	3	3	5

Name	S6C6_5														S6C6_6							
I_k	$(0.1771, 0.1828]$														$(0.1828, 0.1961]$							
$M(\sigma_i)$	5	5	5	4	4	4	4	3	3	3	2	2	1	1	0	3	3	2	2	2	1	1	0
$N(\sigma_i)$	1	1	1	0	2	2	2	2	2	2	2	2	1	2	2	0	1	1	1	2	2	2	2
$\ell(k(\sigma_i))$	4	4	4	1	6	6	6	6	6	6	6	6	3	5	5	1	3	3	3	5	5	5	5

Name	S6C6_7												S6C6_8						
I_k	$(0.1961, 0.2168]$												$(0.2168, 0.2248]$						
$M(\sigma_i)$	6	5	5	4	3	2	2	2	1	1	1	0	3	2	2	2	1	1	0
$N(\sigma_i)$	0	1	1	2	1	1	1	1	2	2	2	3	0	1	1	1	2	2	2
$\ell(k(\sigma_i))$	2	4	4	6	3	3	3	3	5	5	5	6	1	3	3	3	5	5	4

Table A.61: Optimal SCV2V codes with maximum source tree height of 6 and maximum code tree height of 6 (part 1).

Name	S6C6_9															
I_k	$(0.2248, 0.2296]$															
$M(\sigma_i)$	5	5	5	5	5	4	4	4	4	3	3	3	2	2	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	2	2	2	2	2	1
$\ell(k(\sigma_i))$	2	4	4	4	4	6	6	6	6	5	5	5	5	5	5	2

Name	S6C6_10													
I_k	$(0.2296, 0.2676]$													
$M(\sigma_i)$	5	5	5	5	4	4	4	3	3	2	2	2	1	0
$N(\sigma_i)$	0	1	1	1	2	2	2	2	2	1	2	2	3	1
$\ell(k(\sigma_i))$	2	4	4	4	6	6	6	5	5	3	5	5	6	2

Name	S6C6_11													
I_k	$(0.2676, 0.2755]$													
$M(\sigma_i)$	5	5	4	4	3	3	3	3	2	2	2	1	1	0
$N(\sigma_i)$	1	1	2	2	2	2	2	2	0	3	3	2	2	2
$\ell(k(\sigma_i))$	4	4	6	6	5	5	5	5	1	6	6	4	4	4

Name	S6C6_12														S6C6_13													
I_k	$(0.2755, 0.2815]$														$(0.2815, 0.3254]$													
$M(\sigma_i)$	5	4	4	3	3	3	3	2	2	2	1	1	1	0	4	4	3	3	3	3	2	2	2	1	1	1	0	
$N(\sigma_i)$	1	1	2	2	2	2	2	0	3	3	2	2	2	3	1	1	2	2	2	2	0	3	3	2	2	2	3	
$\ell(k(\sigma_i))$	4	4	6	5	5	5	5	1	6	6	4	4	4	6	4	4	5	5	5	5	1	6	6	4	4	4	5	

Name	S6C6_14																			
I_k	$(0.3254, 0.3327]$																			
$M(\sigma_i)$	5	4	4	4	4	3	3	3	3	3	3	3	2	2	2	2	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	2	2	3	3	3	3	1	3	3	3
$\ell(k(\sigma_i))$	3	4	4	4	4	5	5	5	5	5	5	5	6	6	6	6	2	5	5	5

Name	S6C6_15																				
I_k	$(0.3327, 0.3586]$																				
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	2	2	2	2	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	2
$\ell(k(\sigma_i))$	3	4	4	4	4	4	5	5	5	5	5	5	5	5	4	6	6	6	6	5	5	3

Name	S6C6_16														S6C6_17													
I_k	$(0.3586, 0.382]$														$(0.382, 0.4116]$													
$M(\sigma_i)$	4	4	3	3	3	3	3	2	2	2	1	1	0		5	5	4	4	4	3	3	2	2	1	1	1	0	
$N(\sigma_i)$	1	1	0	2	2	2	2	2	3	3	1	3	2		1	1	0	2	2	2	2	2	2	1	1	3	3	
$\ell(k(\sigma_i))$	4	4	2	5	5	5	5	4	6	6	2	5	3		5	5	3	6	6	5	5	4	4	2	2	5	4	

Table A.62: Optimal SCV2V codes with maximum source tree height of 6 and maximum code tree height of 6 (part 2).

Name	S6C6_18											S6C6_19														
I_k	$(0.4116, 0.4156]$											$(0.4156, 0.4207]$														
$M(\sigma_i)$	5	4	4	3	2	2	2	1	1	1	0	5	5	5	5	4	4	4	3	3	2	2	2	1	1	1	0
$N(\sigma_i)$	1	0	2	2	1	2	3	1	1	4	3	0	1	1	1	2	2	2	2	2	2	2	3	1	1	4	3
$\ell(k(\sigma_i))$	5	3	6	5	3	4	5	2	2	6	4	4	5	5	5	6	6	6	5	5	4	4	5	2	2	6	4

Name	S6C6_20															S6C6_21											
I_k	$(0.4207, 0.4462]$															$(0.4462, 0.4493]$											
$M(\sigma_i)$	5	5	5	4	4	3	2	2	2	2	1	1	1	1	0	5	5	4	3	2	2	1	1	1	1	0
$N(\sigma_i)$	0	1	1	2	2	2	1	2	3	3	1	1	4	4	4	0	1	2	1	1	3	1	1	2	4	4
$\ell(k(\sigma_i))$	4	5	5	6	6	5	3	4	5	5	2	2	6	6	5	4	5	6	4	3	5	2	2	3	6	5

Name	S6C6_22									S6C6_23											S6C6_24	
I_k	$(0.4493, 0.4511]$									$(0.4511, 0.4684]$											$(0.47, 0.5]$	
$M(\sigma_i)$	5	4	3	2	1	1	1	1	0	6	5	4	3	2	1	1	1	1	1	0	1	0
$N(\sigma_i)$	0	1	1	1	1	1	2	3	4	0	1	1	1	1	1	1	2	3	4	5	0	1
$\ell(k(\sigma_i))$	4	5	4	3	2	2	3	4	5	5	6	5	4	3	2	2	3	4	5	6	1	1

Table A.63: Optimal SCV2V codes with maximum source tree height of 6 and maximum code tree height of 6 (part 3).

Name	S6C4_1							S6C4_2							S6C4_3					S6C4_4			
I_k	$(0, 0.1187]$							$(0.1187, 0.1433]$							$(0.143, 0.181]$					$(0.18, 0.22]$			
$M(\sigma_i)$	6	5	4	3	2	1	0	5	4	3	2	1	0		4	3	2	1	0	3	2	1	0
$N(\sigma_i)$	0	1	1	1	1	1	1	0	1	1	1	1	1		0	1	1	1	1	0	1	1	1
$\ell(k(\sigma_i))$	1	4	4	4	4	3	3	1	4	4	3	3	3		1	3	3	3	3	1	3	3	2

Name	S6C4_5							S6C4_6			S6C4_7						S6C4_8							
I_k	$(0.2205, 0.2755]$							$(0.28, 0.29]$			$(0.2929, 0.3262]$						$(0.3262, 0.339]$							
$M(\sigma_i)$	5	4	3	2	2	1	0	2	1	0	2	2	2	1	1	0	5	4	3	2	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	1	0	1	1	0	1	1	2	2	2	0	1	1	1	1	1	2	2
$\ell(k(\sigma_i))$	2	4	3	3	3	4	2	1	2	2	1	3	3	4	4	3	3	4	3	3	3	2	4	3

Table A.64: Optimal SCV2V codes with maximum source tree height of 6 and maximum code tree height of 4 (part 1).

Name	S6C4_9					S6C4_10							S6C4_11	
I_k	$(0.339, 0.399]$					$(0.3992, 0.4503]$							$(0.45, 0.5]$	
$M(\sigma_i)$	3	2	1	1	0	4	3	2	1	1	1	0	1	0
$N(\sigma_i)$	0	1	1	1	2	0	1	1	1	1	2	3	0	1
$\ell(k(\sigma_i))$	2	3	2	2	3	3	4	3	2	2	3	4	1	1

Table A.65: Optimal SCV2V codes with maximum source tree height of 6 and maximum code tree height of 4 (part 2).

Name	S6C3_1					S6C3_2				S6C3_3			S6C3_4					S6C3_5	
I_k	$(0, 0.181]$					$(0.18, 0.25]$				$(0.25, 0.32]$			$(0.318, 0.43]$					$(0.43, 0.5]$	
$M(\sigma_i)$	4	3	2	1	0	3	2	1	0	2	1	0	3	2	1	1	0	1	0
$N(\sigma_i)$	0	1	1	1	1	0	1	1	1	0	1	1	0	1	1	1	2	0	1
$\ell(k(\sigma_i))$	1	3	3	3	3	1	3	3	2	1	2	2	2	3	2	2	3	1	1

Table A.66: Optimal SCV2V codes with maximum source tree height of 6 and maximum code tree height of 3.

Name	S5C5_1						S5C5_2					S5C5_3							S5C5_4							
I_k	$(0, 0.1433]$						$(0.143, 0.156]$					$(0.1561, 0.1754]$							$(0.1754, 0.2048]$							
$M(\sigma_i)$	5	4	3	2	1	0	4	3	2	1	0	4	4	3	2	1	1	0	3	3	2	2	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	0	1	1	1	1	0	1	2	1	1	1	2	0	1	1	1	2	2	2	2
$\ell(k(\sigma_i))$	1	4	4	3	3	3	1	3	3	3	3	1	4	5	3	3	3	5	1	3	3	3	5	5	5	5

Name	S5C5_5							S5C5_6							S5C5_7						
I_k	$(0.2048, 0.2324]$							$(0.2324, 0.2414]$							$(0.2414, 0.266]$						
$M(\sigma_i)$	3	2	2	2	1	1	0	5	4	3	3	2	2	1	0	5	4	3	2	2	1	0
$N(\sigma_i)$	0	1	1	1	2	2	2	0	1	1	1	1	2	2	1	0	1	1	1	1	2	1
$\ell(k(\sigma_i))$	1	3	3	3	5	5	4	2	4	3	3	3	5	5	2	2	4	3	3	3	4	2

Name	S5C5_8								S5C5_9								S5C5_10								
I_k	$(0.266, 0.2752]$								$(0.2752, 0.2941]$								$(0.2941, 0.32]$								
$M(\sigma_i)$	3	3	2	2	2	1	1	0	3	2	2	2	1	1	1	0	4	3	2	2	2	1	1	1	0
$N(\sigma_i)$	1	1	0	2	2	2	2	2	1	0	1	2	2	2	2	3	1	2	0	1	2	2	2	2	3
$\ell(k(\sigma_i))$	3	3	1	5	5	4	4	4	3	1	3	5	4	4	4	5	4	5	1	3	4	4	4	4	5

Table A.67: Optimal SCV2V codes with maximum source tree height of 5 and maximum code tree height of 5 (part 1).

Name	S5C5_11													
I_k	$(0.32, 0.3303]$													
$M(\sigma_i)$	5	4	4	4	4	3	3	3	2	2	1	1	1	0
$N(\sigma_i)$	0	1	1	1	1	2	2	2	2	2	1	2	2	3
$\ell(k(\sigma_i))$	3	4	4	4	4	5	5	5	4	4	2	4	4	5

Name	S5C5_12															
I_k	$(0.3303, 0.3522]$															
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	2	2	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2
$\ell(k(\sigma_i))$	3	4	4	4	4	4	5	5	5	5	4	4	4	4	4	3

Name	S5C5_13												S5C5_14								
I_k	$(0.3522, 0.382]$												$(0.382, 0.4253]$								
$M(\sigma_i)$	4	4	3	3	3	2	2	2	2	2	1	1	0	4	4	3	2	2	1	1	1	0
$N(\sigma_i)$	1	1	0	2	2	1	2	2	2	2	3	3	2	0	1	2	1	2	1	1	3	3
$\ell(k(\sigma_i))$	4	4	2	5	5	3	4	4	4	4	5	5	3	3	4	5	3	4	2	2	5	4

Name	S5C5_15							S5C5_16								S5C5_17	
I_k	$(0.4253, 0.4334]$							$(0.4334, 0.4614]$								$(0.46, 0.5]$	
$M(\sigma_i)$	4	3	2	1	1	1	0	5	4	3	2	1	1	1	1	0	1	0
$N(\sigma_i)$	0	1	1	1	1	2	3	0	1	1	1	1	1	2	3	4	0	1
$\ell(k(\sigma_i))$	3	4	3	2	2	3	4	4	5	4	3	2	2	3	4	5	1	1

Table A.68: Optimal SCV2V codes with maximum source tree height of 5 and maximum code tree height of 5 (part 2).

Name	S4C4_1					S4C4_2				S4C4_3					S4C4_4						S4C4_5				
I_k	$(0, 0.181]$					$(0.18, 0.25]$				$(0.245, 0.293]$					$(0.2929, 0.3333]$						$(0.333, 0.399]$				
$M(\sigma_i)$	4	3	2	1	0	3	2	1	0	3	2	2	1	0	2	2	2	1	1	0	3	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	0	1	1	1	1	0	2	2	1	0	1	1	2	2	2	0	1	1	1	2
$\ell(k(\sigma_i))$	1	3	3	3	3	1	3	3	2	3	1	4	4	2	1	3	3	4	4	3	2	3	2	2	3

Table A.69: Optimal SCV2V codes with maximum source tree height of 4 and maximum code tree height of 4 (part 1).

Name	S4C4_6							S4C4_7	
I_k	$(0.3992, 0.4503]$							$(0.45, 0.5]$	
$M(\sigma_i)$	4	3	2	1	1	1	0	1	0
$N(\sigma_i)$	0	1	1	1	1	2	3	0	1
$\ell(k(\sigma_i))$	3	4	3	2	2	3	4	1	1

Table A.70: Optimal SCV2V codes with maximum source tree height of 4 and maximum code tree height of 4 (part 2).

A.2 HEVC-related V2V codes

A.2.1 TMuC V2V codes

Name	TMuC_1																								
I_k	$(0, 0.0288]$																								
$M(\sigma_i)$	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	1	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	5	4	3	2	1	0																					
$N(\sigma_i)$	1	1	1	1	1	1																					
$\ell(k(\sigma_i))$	6	6	6	6	6	6																					

Name	TMuC_2																	
I_k	$(0.0288, 0.0578]$																	
$M(\sigma_i)$	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	1	6	6	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5

Name	TMuC_3																			
I_k	$(0.0578, 0.1189]$																			
$M(\sigma_i)$	11	10	10	9	9	8	8	8	7	7	6	5	4	3	3	2	2	2	1	1	0
$N(\sigma_i)$	1	1	2	2	2	0	2	2	2	2	2	1	1	1	1	1	1	2	2	2	2
$\ell(k(\sigma_i))$	5	5	8	8	8	1	8	8	8	8	8	4	4	4	4	4	4	7	7	7	7

Name	TMuC_4																					
I_k	$(0.1189, 0.1849]$																					
$M(\sigma_i)$	6	6	5	5	5	5	4	4	4	3	3	3	3	3	2	2	2	2	1	1	1	0
$N(\sigma_i)$	2	2	1	1	3	3	0	3	3	2	2	2	2	2	1	3	3	3	1	3	3	3
$\ell(k(\sigma_i))$	7	7	4	4	9	9	1	9	9	6	6	6	6	6	3	8	8	8	3	8	8	8

Name	TMuC_5															
I_k	$(0.1849, 0.2338]$															
$M(\sigma_i)$	5	4	4	3	3	3	2	2	2	2	2	2	1	1	1	1	0
$N(\sigma_i)$	2	3	3	0	3	4	1	1	1	2	2	4	3	3	3	3	4
$\ell(k(\sigma_i))$	6	8	8	1	8	10	3	3	3	5	5	10	7	7	7	7	9

Table A.71: TMuC V2V codes (part 1).

Name	TMuC_6																					
I_k	$(0.2338, 0.2775]$																					
$M(\sigma_i)$	7	7	7	6	5	5	5	5	5	5	5	4	4	4	3	3	3	2	2	2	1	0
$N(\sigma_i)$	0	1	2	3	1	1	1	1	2	2	2	3	3	3	2	2	2	1	2	3	3	1
$\ell(k(\sigma_i))$	3	5	7	8	4	4	4	4	6	6	6	8	8	8	5	5	5	3	5	7	6	2

Name	TMuC_7												
I_k	$(0.2775, 0.3265]$												
$M(\sigma_i)$	4	4	3	3	3	3	2	2	2	1	1	1	0
$N(\sigma_i)$	1	1	2	2	2	2	0	3	3	2	2	2	3
$\ell(k(\sigma_i))$	4	4	5	5	5	5	1	6	6	4	4	4	5

Name	TMuC_8																					
I_k	$(0.3265, 0.3763]$																					
$M(\sigma_i)$	5	4	4	4	4	4	3	3	3	3	3	3	3	3	2	2	2	2	2	1	1	0
$N(\sigma_i)$	0	1	1	1	1	1	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	2
$\ell(k(\sigma_i))$	3	4	4	4	4	4	5	5	5	5	5	5	5	5	4	6	6	6	6	5	5	3

Name	TMuC_9																					
I_k	$(0.3763, 0.4201]$																					
$M(\sigma_i)$	7	7	6	6	5	5	5	4	4	4	4	3	3	3	2	2	2	1	1	1	0	
$N(\sigma_i)$	3	4	2	5	1	4	5	0	3	4	6	2	3	6	1	2	5	1	1	4	3	
$\ell(k(\sigma_i))$	9	10	7	11	5	9	10	3	7	8	11	5	6	10	3	4	8	2	2	6	4	

Name	TMuC_10																			
I_k	$(0.4201, 0.4369]$																			
$M(\sigma_i)$	6	5	5	5	5	4	4	3	3	3	2	2	2	2	1	1	1	1	0	
$N(\sigma_i)$	2	0	1	1	3	2	3	2	3	4	1	2	3	5	1	1	4	4	4	
$\ell(k(\sigma_i))$	7	4	5	5	8	6	7	5	6	7	3	4	5	8	2	2	6	6	5	

Name	TMuC_11																					TMuC_12	
I_k	$(0.4369, 0.4638]$																					$(0.46, 0.5]$	
$M(\sigma_i)$	6	6	6	6	5	5	5	4	4	3	3	3	2	2	2	2	1	1	1	1	1	0	1	0
$N(\sigma_i)$	0	1	1	1	2	2	2	2	2	2	4	4	1	3	5	5	1	1	2	5	5	5	0	1
$\ell(k(\sigma_i))$	5	6	6	6	7	7	7	6	6	5	7	7	3	5	8	8	2	2	3	7	7	6	1	1

Table A.72: TMuC V2V codes (part 2).

A.2.2 Systematic V2V codes

Name	SYS_1																									
I_k	$(0, 0.0296]$																									
$M(\sigma_i)$	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	1	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
$M(\sigma_i)$	5	4	3	2	1	0																					
$N(\sigma_i)$	1	1	1	1	1	1																					
$\ell(k(\sigma_i))$	6	6	6	6	6	6																					

Name	SYS_2																	SYS_3								
I_k	$(0.0296, 0.0584]$																	$(0.0584, 0.1133]$								
$M(\sigma_i)$	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	8	7	6	5	4	3	2	1	0
$N(\sigma_i)$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1
$\ell(k(\sigma_i))$	1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	1	4	4	4	4	4	4	4	4

Name	SYS_4					SYS_5								SYS_6			SYS_7					SYS_8	
I_k	$(0.113, 0.182]$					$(0.1816, 0.2473]$								$(0.25, 0.32]$			$(0.318, 0.43]$					$(0.43, 0.5]$	
$M(\sigma_i)$	4	3	2	1	0	3	2	2	2	1	1	1	0	2	1	0	3	2	1	1	0	1	0
$N(\sigma_i)$	0	1	1	1	1	0	1	1	1	2	2	2	3	0	1	1	0	1	1	1	2	0	1
$\ell(k(\sigma_i))$	1	3	3	3	3	1	3	3	3	5	5	5	5	1	2	2	2	3	2	2	3	1	1

Table A.73: Systematic V2V codes.

Appendix B

Miscellaneous

B.1 Mediant inequality

The following Lemma is known as mediant inequality and can be found in the related literature. See [52] for details.

Lemma B.1. *Let a, b, c, d be real and positive. The mediant inequality states that if $a/b \leq c/d$, then*

$$\frac{a}{b} \leq \frac{a+c}{b+d} \leq \frac{c}{d} \tag{B.1}$$

where $(a+c)/(b+d)$ is denoted mediant of a/b and c/d.

Proof. (B.1) can be transformed into

$$\frac{a+c}{b+d} - \frac{a}{b} \geq 0 \tag{B.2}$$

and

$$\frac{c}{d} - \frac{a+c}{b+d} \geq 0. \tag{B.3}$$

(B.2) can be rewritten as

$$\frac{bc-ad}{b(b+d)} \geq 0 \tag{B.4}$$

$$\iff \frac{d}{b+d}\left(\frac{c}{d} - \frac{a}{b}\right) \geq 0. \tag{B.5}$$

Since $d/(b+d)$ is always greater 0, (B.5) holds if

$$\frac{c}{d} \geq \frac{a}{b} \tag{B.6}$$

holds, which is fulfilled by definition. Analogously, (B.3) can be rewritten as

$$\frac{bc-ad}{d(b+d)} \geq 0 \tag{B.7}$$

$$\iff \frac{b}{b+d}\left(\frac{c}{d}-\frac{a}{b}\right) \geq 0. \tag{B.8}$$

Since $b/(b+d)$ is always greater 0, (B.8) holds if (B.6) holds. Consequently, both (B.5) and (B.8) hold and the Lemma is proven. □

B.2 P coders derived by the successive removal algorithm

UR_5	$\omega_{55}-\omega_{62}$
UR_4	$\omega_{42}-\omega_{54}$
UR_3	$\omega_{32}-\omega_{41}$
S6C6_1	$\omega_{30}-\omega_{31}$
S6C6_2	$\omega_{28}-\omega_{29}$
S6C6_3	$\omega_{24}-\omega_{27}$
S6C6_4	$\omega_{20}-\omega_{23}$
S6C6_6	$\omega_{18}-\omega_{19}$
S6C6_7	$\omega_{16}-\omega_{17}$
S6C6_9	ω_{15}
S6C6_10	$\omega_{12}-\omega_{14}$
S6C6_13	$\omega_{9}-\omega_{11}$
S6C6_14	ω_{8}
S6C6_15	ω_{7}
S6C6_16	ω_{6}
S6C6_17	$\omega_{4}-\omega_{5}$
S6C6_20	ω_{3}
S6C6_22	ω_{2}
S6C6_24	$\omega_{0}-\omega_{1}$

UR_5	$\omega_{55}-\omega_{62}$
UR_4	$\omega_{42}-\omega_{54}$
UR_3	$\omega_{32}-\omega_{41}$
S6C6_1	$\omega_{30}-\omega_{31}$
S6C6_2	$\omega_{28}-\omega_{29}$
S6C6_3	$\omega_{24}-\omega_{27}$
S6C6_4	$\omega_{20}-\omega_{23}$
S6C6_6	$\omega_{18}-\omega_{19}$
S6C6_7	$\omega_{16}-\omega_{17}$
S6C6_10	$\omega_{12}-\omega_{15}$
S6C6_13	$\omega_{9}-\omega_{11}$
S6C6_14	ω_{8}
S6C6_15	ω_{7}
S6C6_16	ω_{6}
S6C6_17	$\omega_{4}-\omega_{5}$
S6C6_20	ω_{3}
S6C6_22	ω_{2}
S6C6_24	$\omega_{0}-\omega_{1}$

UR_5	$\omega_{55}-\omega_{62}$
UR_4	$\omega_{42}-\omega_{54}$
UR_3	$\omega_{32}-\omega_{41}$
S6C6_2	$\omega_{28}-\omega_{31}$
S6C6_3	$\omega_{24}-\omega_{27}$
S6C6_4	$\omega_{20}-\omega_{23}$
S6C6_6	$\omega_{18}-\omega_{19}$
S6C6_7	$\omega_{16}-\omega_{17}$
S6C6_10	$\omega_{12}-\omega_{15}$
S6C6_13	$\omega_{9}-\omega_{11}$
S6C6_14	ω_{8}
S6C6_15	ω_{7}
S6C6_16	ω_{6}
S6C6_17	$\omega_{4}-\omega_{5}$
S6C6_20	ω_{3}
S6C6_22	ω_{2}
S6C6_24	$\omega_{0}-\omega_{1}$

Table B.1: P coders with 19, 18, and 17 V2V codes derived by applying the successive removal algorithm to S6C6+UR codes.

UR_5	$\omega_{55} - \omega_{62}$
UR_4	$\omega_{42} - \omega_{54}$
UR_3	$\omega_{32} - \omega_{41}$
S6C6_2	$\omega_{28} - \omega_{31}$
S6C6_3	$\omega_{24} - \omega_{27}$
S6C6_4	$\omega_{20} - \omega_{23}$
S6C6_6	$\omega_{18} - \omega_{19}$
S6C6_7	$\omega_{16} - \omega_{17}$
S6C6_10	$\omega_{12} - \omega_{15}$
S6C6_13	$\omega_9 - \omega_{11}$
S6C6_15	$\omega_7 - \omega_8$
S6C6_16	ω_6
S6C6_17	$\omega_4 - \omega_5$
S6C6_20	ω_3
S6C6_22	ω_2
S6C6_24	$\omega_0 - \omega_1$

UR_5	$\omega_{55} - \omega_{62}$
UR_4	$\omega_{42} - \omega_{54}$
UR_3	$\omega_{32} - \omega_{41}$
S6C6_2	$\omega_{28} - \omega_{31}$
S6C6_3	$\omega_{24} - \omega_{27}$
S6C6_4	$\omega_{20} - \omega_{23}$
S6C6_7	$\omega_{16} - \omega_{19}$
S6C6_10	$\omega_{12} - \omega_{15}$
S6C6_13	$\omega_9 - \omega_{11}$
S6C6_15	$\omega_7 - \omega_8$
S6C6_16	ω_6
S6C6_17	$\omega_4 - \omega_5$
S6C6_20	ω_3
S6C6_22	ω_2
S6C6_24	$\omega_0 - \omega_1$

UR_5	$\omega_{55} - \omega_{62}$
UR_4	$\omega_{42} - \omega_{54}$
UR_3	$\omega_{32} - \omega_{41}$
S6C6_2	$\omega_{28} - \omega_{31}$
S6C6_3	$\omega_{24} - \omega_{27}$
S6C6_4	$\omega_{20} - \omega_{23}$
S6C6_7	$\omega_{16} - \omega_{19}$
S6C6_10	$\omega_{12} - \omega_{15}$
S6C6_13	$\omega_9 - \omega_{11}$
S6C6_15	$\omega_6 - \omega_8$
S6C6_17	$\omega_4 - \omega_5$
S6C6_20	ω_3
S6C6_22	ω_2
S6C6_24	$\omega_0 - \omega_1$

Table B.2: P coders with 16, 15, and 14 V2V codes derived by applying the successive removal algorithm to S6C6+UR codes.

UR_5	$\omega_{55} - \omega_{62}$
UR_4	$\omega_{42} - \omega_{54}$
UR_3	$\omega_{32} - \omega_{41}$
S6C6_2	$\omega_{28} - \omega_{31}$
S6C6_3	$\omega_{24} - \omega_{27}$
S6C6_4	$\omega_{20} - \omega_{23}$
S6C6_7	$\omega_{16} - \omega_{19}$
S6C6_10	$\omega_{12} - \omega_{15}$
S6C6_13	$\omega_9 - \omega_{11}$
S6C6_15	$\omega_6 - \omega_8$
S6C6_17	$\omega_4 - \omega_5$
S6C6_20	$\omega_2 - \omega_3$
S6C6_24	$\omega_0 - \omega_1$

UR_5	$\omega_{55} - \omega_{62}$
UR_4	$\omega_{42} - \omega_{54}$
UR_3	$\omega_{32} - \omega_{41}$
S6C6_2	$\omega_{26} - \omega_{31}$
S6C6_4	$\omega_{20} - \omega_{25}$
S6C6_7	$\omega_{16} - \omega_{19}$
S6C6_10	$\omega_{12} - \omega_{15}$
S6C6_13	$\omega_9 - \omega_{11}$
S6C6_15	$\omega_6 - \omega_8$
S6C6_17	$\omega_4 - \omega_5$
S6C6_20	$\omega_2 - \omega_3$
S6C6_24	$\omega_0 - \omega_1$

UR_5	$\omega_{55} - \omega_{62}$
UR_4	$\omega_{42} - \omega_{54}$
UR_3	$\omega_{32} - \omega_{41}$
S6C6_2	$\omega_{26} - \omega_{31}$
S6C6_4	$\omega_{20} - \omega_{25}$
S6C6_7	$\omega_{16} - \omega_{19}$
S6C6_10	$\omega_{12} - \omega_{15}$
S6C6_13	$\omega_9 - \omega_{11}$
S6C6_15	$\omega_5 - \omega_8$
S6C6_20	$\omega_2 - \omega_4$
S6C6_24	$\omega_0 - \omega_1$

Table B.3: P coders with 13, 12, and 11 V2V codes derived by applying the successive removal algorithm to S6C6+UR codes.

UR_5	$\omega_{55} - \omega_{62}$
UR_4	$\omega_{42} - \omega_{54}$
UR_3	$\omega_{32} - \omega_{41}$
S6C6_2	$\omega_{26} - \omega_{31}$
S6C6_4	$\omega_{18} - \omega_{25}$
S6C6_10	$\omega_{12} - \omega_{17}$
S6C6_13	$\omega_{9} - \omega_{11}$
S6C6_15	$\omega_{5} - \omega_{8}$
S6C6_20	$\omega_{2} - \omega_{4}$
S6C6_24	$\omega_{0} - \omega_{1}$

UR_5	$\omega_{55} - \omega_{62}$
UR_4	$\omega_{42} - \omega_{54}$
UR_3	$\omega_{29} - \omega_{41}$
S6C6_4	$\omega_{18} - \omega_{28}$
S6C6_10	$\omega_{12} - \omega_{17}$
S6C6_13	$\omega_{9} - \omega_{11}$
S6C6_15	$\omega_{5} - \omega_{8}$
S6C6_20	$\omega_{2} - \omega_{4}$
S6C6_24	$\omega_{0} - \omega_{1}$

UR_5	$\omega_{47} - \omega_{62}$
UR_3	$\omega_{29} - \omega_{46}$
S6C6_4	$\omega_{18} - \omega_{28}$
S6C6_10	$\omega_{12} - \omega_{17}$
S6C6_13	$\omega_{9} - \omega_{11}$
S6C6_15	$\omega_{5} - \omega_{8}$
S6C6_20	$\omega_{2} - \omega_{4}$
S6C6_24	$\omega_{0} - \omega_{1}$

Table B.4: P coders with 10, 9, and 8 V2V codes derived by applying the successive removal algorithm to S6C6+UR codes.

UR_5	$\omega_{47} - \omega_{62}$
UR_3	$\omega_{29} - \omega_{46}$
S6C6_4	$\omega_{18} - \omega_{28}$
S6C6_10	$\omega_{11} - \omega_{17}$
S6C6_15	$\omega_{5} - \omega_{10}$
S6C6_20	$\omega_{2} - \omega_{4}$
S6C6_24	$\omega_{0} - \omega_{1}$

UR_5	$\omega_{47} - \omega_{62}$
UR_3	$\omega_{29} - \omega_{46}$
S6C6_4	$\omega_{18} - \omega_{28}$
S6C6_10	$\omega_{11} - \omega_{17}$
S6C6_15	$\omega_{4} - \omega_{10}$
S6C6_24	$\omega_{0} - \omega_{3}$

UR_5	$\omega_{47} - \omega_{62}$
UR_3	$\omega_{29} - \omega_{46}$
S6C6_4	$\omega_{14} - \omega_{28}$
S6C6_15	$\omega_{4} - \omega_{13}$
S6C6_24	$\omega_{0} - \omega_{3}$

Table B.5: P coders with 7, 6, and 5 V2V codes derived by applying the successive removal algorithm to S6C6+UR codes.

UR_5	$\omega_{39} - \omega_{62}$
S6C6_4	$\omega_{14} - \omega_{38}$
S6C6_15	$\omega_{4} - \omega_{13}$
S6C6_24	$\omega_{0} - \omega_{3}$

UR_5	$\omega_{39} - \omega_{62}$
S6C6_4	$\omega_{9} - \omega_{38}$
S6C6_24	$\omega_{0} - \omega_{8}$

S6C6_4	$\omega_{9} - \omega_{62}$
S6C6_24	$\omega_{0} - \omega_{8}$

Table B.6: P coders with 4, 3, and 2 V2V codes derived by applying the successive removal algorithm to S6C6+UR codes.

S6C6_24	$\omega_{0} - \omega_{62}$

Table B.7: P coder with 1 V2V code derived by applying the successive removal algorithm to S6C6+UR codes.

B.3 P coders derived by exhaustive search

S6C6_13	$\omega_0 - \omega_{62}$

S6C6_2	$\omega_{14} - \omega_{62}$
S6C6_20	$\omega_0 - \omega_{13}$

UR_3	$\omega_{24} - \omega_{62}$
S6C6_10	$\omega_7 - \omega_{23}$
S6C6_24	$\omega_0 - \omega_6$

Table B.8: P coders with 1, 2, and 3 V2V codes derived by applying exhaustive search to S6C6+UR codes.

UR_4	$\omega_{34} - \omega_{62}$
S6C6_4	$\omega_{16} - \omega_{33}$
S6C6_13	$\omega_5 - \omega_{15}$
S6C6_24	$\omega_0 - \omega_4$

UR_4	$\omega_{36} - \omega_{62}$
S6C6_3	$\omega_{20} - \omega_{35}$
S6C6_10	$\omega_{11} - \omega_{19}$
S6C6_15	$\omega_4 - \omega_{10}$
S6C6_24	$\omega_0 - \omega_3$

UR_5	$\omega_{47} - \omega_{62}$
UR_3	$\omega_{29} - \omega_{46}$
S6C6_4	$\omega_{18} - \omega_{28}$
S6C6_10	$\omega_{11} - \omega_{17}$
S6C6_15	$\omega_4 - \omega_{10}$
S6C6_24	$\omega_0 - \omega_3$

Table B.9: P coders with 4, 5, and 6 V2V codes derived by applying exhaustive search to S6C6+UR codes.

UR_5	$\omega_{47} - \omega_{62}$
UR_3	$\omega_{29} - \omega_{46}$
S6C6_4	$\omega_{18} - \omega_{28}$
S6C6_10	$\omega_{12} - \omega_{17}$
S6C6_13	$\omega_7 - \omega_{11}$
S6C6_17	$\omega_3 - \omega_6$
S6C6_24	$\omega_0 - \omega_2$

UR_5	$\omega_{47} - \omega_{62}$
UR_3	$\omega_{29} - \omega_{46}$
S6C6_4	$\omega_{18} - \omega_{28}$
S6C6_10	$\omega_{12} - \omega_{17}$
S6C6_13	$\omega_8 - \omega_{11}$
S6C6_16	$\omega_5 - \omega_7$
S6C6_20	$\omega_2 - \omega_4$
S6C6_24	$\omega_0 - \omega_1$

UR_5	$\omega_{55} - \omega_{62}$
UR_4	$\omega_{42} - \omega_{54}$
UR_3	$\omega_{29} - \omega_{41}$
S6C6_4	$\omega_{18} - \omega_{28}$
S6C6_10	$\omega_{12} - \omega_{17}$
S6C6_13	$\omega_8 - \omega_{11}$
S6C6_16	$\omega_5 - \omega_7$
S6C6_20	$\omega_2 - \omega_4$
S6C6_24	$\omega_0 - \omega_1$

Table B.10: P coders with 7, 8, and 9 V2V codes derived by applying exhaustive search to S6C6+UR codes.

UR_5	$\omega_{55} - \omega_{62}$
UR_4	$\omega_{42} - \omega_{54}$
UR_3	$\omega_{32} - \omega_{41}$
S6C6_2	$\omega_{26} - \omega_{31}$
S6C6_4	$\omega_{18} - \omega_{25}$
S6C6_10	$\omega_{12} - \omega_{17}$
S6C6_13	$\omega_{8} - \omega_{11}$
S6C6_16	$\omega_{5} - \omega_{7}$
S6C6_20	$\omega_{2} - \omega_{4}$
S6C6_24	$\omega_{0} - \omega_{1}$

UR_5	$\omega_{55} - \omega_{62}$
UR_4	$\omega_{42} - \omega_{54}$
UR_3	$\omega_{32} - \omega_{41}$
S6C6_2	$\omega_{26} - \omega_{31}$
S6C6_4	$\omega_{20} - \omega_{25}$
S6C6_7	$\omega_{16} - \omega_{19}$
S6C6_10	$\omega_{12} - \omega_{15}$
S6C6_13	$\omega_{8} - \omega_{11}$
S6C6_16	$\omega_{5} - \omega_{7}$
S6C6_20	$\omega_{2} - \omega_{4}$
S6C6_24	$\omega_{0} - \omega_{1}$

UR_5	$\omega_{55} - \omega_{62}$
UR_4	$\omega_{42} - \omega_{54}$
UR_3	$\omega_{32} - \omega_{41}$
S6C6_2	$\omega_{26} - \omega_{31}$
S6C6_4	$\omega_{20} - \omega_{25}$
S6C6_7	$\omega_{16} - \omega_{19}$
S6C6_10	$\omega_{12} - \omega_{15}$
S6C6_13	$\omega_{9} - \omega_{11}$
S6C6_15	$\omega_{6} - \omega_{8}$
S6C6_17	$\omega_{4} - \omega_{5}$
S6C6_20	$\omega_{2} - \omega_{3}$
S6C6_24	$\omega_{0} - \omega_{1}$

Table B.11: P coders with 10, 11, and 12 V2V codes derived by applying exhaustive search to S6C6+UR codes.

SYS_7	$\omega_{0} - \omega_{62}$

SYS_4	$\omega_{10} - \omega_{62}$
SYS_8	$\omega_{0} - \omega_{9}$

SYS_3	$\omega_{22} - \omega_{62}$
SYS_6	$\omega_{6} - \omega_{21}$
SYS_8	$\omega_{0} - \omega_{5}$

Table B.12: P coders with 1, 2, and 3 V2V codes derived by applying exhaustive search to systematic V2V codes.

SYS_2	$\omega_{31} - \omega_{62}$
SYS_5	$\omega_{11} - \omega_{30}$
SYS_7	$\omega_{3} - \omega_{10}$
SYS_8	$\omega_{0} - \omega_{2}$

SYS_1	$\omega_{47} - \omega_{62}$
SYS_3	$\omega_{26} - \omega_{46}$
SYS_5	$\omega_{11} - \omega_{25}$
SYS_7	$\omega_{3} - \omega_{10}$
SYS_8	$\omega_{0} - \omega_{2}$

SYS_1	$\omega_{47} - \omega_{62}$
SYS_3	$\omega_{29} - \omega_{46}$
SYS_4	$\omega_{17} - \omega_{28}$
SYS_6	$\omega_{9} - \omega_{16}$
SYS_7	$\omega_{3} - \omega_{8}$
SYS_8	$\omega_{0} - \omega_{2}$

Table B.13: P coders with 4, 5, and 6 V2V codes derived by applying exhaustive search to systematic V2V codes.

SYS_1	$\omega_{47} - \omega_{62}$
SYS_3	$\omega_{29} - \omega_{46}$
SYS_4	$\omega_{20} - \omega_{28}$
SYS_5	$\omega_{14} - \omega_{19}$
SYS_6	$\omega_{9} - \omega_{13}$
SYS_7	$\omega_{3} - \omega_{8}$
SYS_8	$\omega_{0} - \omega_{2}$

SYS_1	$\omega_{55} - \omega_{62}$
SYS_2	$\omega_{42} - \omega_{54}$
SYS_3	$\omega_{29} - \omega_{41}$
SYS_4	$\omega_{20} - \omega_{28}$
SYS_5	$\omega_{14} - \omega_{19}$
SYS_6	$\omega_{9} - \omega_{13}$
SYS_7	$\omega_{3} - \omega_{8}$
SYS_8	$\omega_{0} - \omega_{2}$

Table B.14: P coders with 7 and 8 V2V codes derived by applying exhaustive search to systematic V2V codes.

TMuC_7	$\omega_{0} - \omega_{62}$

TMuC_4	$\omega_{11} - \omega_{62}$
TMuC_11	$\omega_{0} - \omega_{10}$

TMuC_3	$\omega_{23} - \omega_{62}$
TMuC_6	$\omega_{6} - \omega_{22}$
TMuC_12	$\omega_{0} - \omega_{5}$

Table B.15: P coders with 1, 2, and 3 V2V codes derived by applying exhaustive search to the TMuC V2V codes.

TMuC_2	$\omega_{34} - \omega_{62}$
TMuC_4	$\omega_{16} - \omega_{33}$
TMuC_7	$\omega_{5} - \omega_{15}$
TMuC_12	$\omega_{0} - \omega_{4}$

TMuC_1	$\omega_{47} - \omega_{62}$
TMuC_3	$\omega_{25} - \omega_{46}$
TMuC_5	$\omega_{12} - \omega_{24}$
TMuC_8	$\omega_{4} - \omega_{11}$
TMuC_12	$\omega_{0} - \omega_{3}$

TMuC_1	$\omega_{47} - \omega_{62}$
TMuC_3	$\omega_{28} - \omega_{46}$
TMuC_4	$\omega_{17} - \omega_{27}$
TMuC_6	$\omega_{10} - \omega_{16}$
TMuC_8	$\omega_{4} - \omega_{9}$
TMuC_12	$\omega_{0} - \omega_{3}$

Table B.16: P coders with 4, 5, and 6 V2V codes derived by applying exhaustive search to the TMuC V2V codes.

TMuC_1	$\omega_{47} - \omega_{62}$
TMuC_3	$\omega_{28} - \omega_{46}$
TMuC_4	$\omega_{20} - \omega_{27}$
TMuC_5	$\omega_{13} - \omega_{19}$
TMuC_7	$\omega_{7} - \omega_{12}$
TMuC_9	$\omega_{3} - \omega_{6}$
TMuC_12	$\omega_{0} - \omega_{2}$

TMuC_1	$\omega_{55} - \omega_{62}$
TMuC_2	$\omega_{42} - \omega_{54}$
TMuC_3	$\omega_{28} - \omega_{41}$
TMuC_4	$\omega_{20} - \omega_{27}$
TMuC_5	$\omega_{13} - \omega_{19}$
TMuC_7	$\omega_{7} - \omega_{12}$
TMuC_9	$\omega_{3} - \omega_{6}$
TMuC_12	$\omega_{0} - \omega_{2}$

TMuC_1	$\omega_{55} - \omega_{62}$
TMuC_2	$\omega_{42} - \omega_{54}$
TMuC_3	$\omega_{28} - \omega_{41}$
TMuC_4	$\omega_{20} - \omega_{27}$
TMuC_5	$\omega_{15} - \omega_{19}$
TMuC_6	$\omega_{12} - \omega_{14}$
TMuC_7	$\omega_{7} - \omega_{11}$
TMuC_9	$\omega_{3} - \omega_{6}$
TMuC_12	$\omega_{0} - \omega_{2}$

Table B.17: P coders with 7, 8, and 9 V2V codes derived by applying exhaustive search to the TMuC V2V codes.

TMuC_1	$\omega_{55} - \omega_{62}$
TMuC_2	$\omega_{42} - \omega_{54}$
TMuC_3	$\omega_{28} - \omega_{41}$
TMuC_4	$\omega_{20} - \omega_{27}$
TMuC_5	$\omega_{15} - \omega_{19}$
TMuC_6	$\omega_{12} - \omega_{14}$
TMuC_7	$\omega_{9} - \omega_{11}$
TMuC_8	$\omega_{6} - \omega_{8}$
TMuC_9	$\omega_{3} - \omega_{5}$
TMuC_12	$\omega_{0} - \omega_{2}$

TMuC_1	$\omega_{55} - \omega_{62}$
TMuC_2	$\omega_{42} - \omega_{54}$
TMuC_3	$\omega_{28} - \omega_{41}$
TMuC_4	$\omega_{20} - \omega_{27}$
TMuC_5	$\omega_{15} - \omega_{19}$
TMuC_6	$\omega_{12} - \omega_{14}$
TMuC_7	$\omega_{9} - \omega_{11}$
TMuC_8	$\omega_{6} - \omega_{8}$
TMuC_9	$\omega_{3} - \omega_{5}$
TMuC_11	ω_{2}
TMuC_12	$\omega_{0} - \omega_{1}$

TMuC_1	$\omega_{55} - \omega_{62}$
TMuC_2	$\omega_{42} - \omega_{54}$
TMuC_3	$\omega_{28} - \omega_{41}$
TMuC_4	$\omega_{20} - \omega_{27}$
TMuC_5	$\omega_{15} - \omega_{19}$
TMuC_6	$\omega_{12} - \omega_{14}$
TMuC_7	$\omega_{9} - \omega_{11}$
TMuC_8	$\omega_{6} - \omega_{8}$
TMuC_9	$\omega_{4} - \omega_{5}$
TMuC_10	ω_{3}
TMuC_11	ω_{2}
TMuC_12	$\omega_{0} - \omega_{1}$

Table B.18: P coders with 10, 11, and 12 V2V codes derived by applying exhaustive search to the TMuC V2V codes.

Glossary

Acronyms

AVC	Advanced Video Coding
BCE	Binary Coding Engine
BP	Bin-Pipe
CABAC	Context-based Adaptive Binary Arithmetic Coding
CV2V	Code tree height-limited Variable-to-Variable length
F2V	Fixed-to-Variable length
HEVC	High Efficiency Video Coding
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
ITU-T	International Telecommunication Union - Telecommunication Standardization Sector
LPS	Less Probable Symbol
LSB	Least Significant Bit
LV2V	Leaf-limited Variable-to-Variable length
MPS	More Probable Symbol
MSB	Most Significant Bit
PIPE	Probability Interval Partitioning Entropy

pmf	Probability Mass Function
SCV2V	Source and Code tree height-limited Variable-to-Variable length
SV2V	Source tree height-limited Variable-to-Variable length
TMuC	Test Model under Consideration
UR	Unary-to-Rice
V2F	Variable-to-Fixed length
V2V	Variable-to-Variable length

Symbols

α	Constant of the probability estimator of CABAC with value $\sqrt[63]{3/80}$
$\hat{B}(X)$	Bin encoder that is capable of optimally encoding random variable X
$\mathcal{B}$	The binary alphabet $\{0, 1\}$
$\tilde{B}$	P coder that uses ideal binary arithmetic bin coders
$\mathcal{C}$	Interchangeably used to refer to a code tree as well as to the set of code words of a code tree
C	Random variable over code words $\mathcal{C}$
c_i	Code word i
$\{G_i\}$	Binary random process that produces the coding bins
g_i	Coding bin associated with random variable G_i
I_k	Probability interval associated with bin coder k
$\mathcal{I}$	The set of all integers in the interval $[256, 510]$
Ω_k	Binary random variable with pmf $p_{\Omega_k}(1) = \omega_k$
ω_k	Possible values of the probability of the LPS in CABAC
$\mathcal{P}$	The set of all possible prefix-free codes

$\mathcal{P}_i$	The set of all possible prefix-free codes with i leaf nodes
$\hat{\mathcal{P}}_i$	The set of all possible prefix-free codes of height i or less
$\tilde{\mathcal{P}}_i$	The set of all possible prefix-free codes with up to i leaf nodes
Π_x	The set of all possible permutations of x items
$\mathcal{Q}$	Set of probability values ω_k that the states of CABAC can attain
Q'	Random variable over $\mathcal{Q}$ describing the distribution of conditional probabilities of one of a random process
r_i	Range value as present in the M coder before encoding coding bin g_i
$\hat{S}$	Random variable over $\mathcal{S}$ that can be encoded with zero redundancy with code tree $\mathcal{C}$
$\mathcal{S}$	Interchangeably used to refer to a source tree as well as to the set of source words of a source tree
σ_i	Source word i
S	Random variable over source words $\mathcal{S}$
u_i	Value of the MPS associated with coding bin g_i
$\mathcal{V}$	The set of all possible V2V codes
v	Variable-to-variable length code
$\bar{\mathcal{V}}_i$	The set of all possible V2V codes with a code tree height of i or less
$\hat{\mathcal{V}}_i$	The set of all possible V2V codes with a source tree height of i or less
$\mathcal{V}_i$	The set of all possible V2V codes with i leaf nodes
$\tilde{\mathcal{V}}_i$	The set of all possible V2V codes with up to i leaf nodes
$\{W_i\}$	Binary random process describing modeled bins
w_i	Modeled bin associated with random variable W_i
X_0	Binary random variable with $p_{X_0}(1) \to 0$

Functions

$\mathrm{Cc}(\mathcal{C})$	Canonical representation of code tree $\mathcal{C}$
$\mathrm{Cs}(\mathcal{S})$	Canonical representation of source tree $\mathcal{S}$
$\mathrm{Cv}(v)$	Canonical representation of a V2V code v
$\mathrm{Cw}(\sigma)$	Canonical representation of source word σ
$D(X\|\|Y)$	Kullback-Leibler divergence of Y from X
$\mathrm{Eq}(\mathcal{X})$	Canonical representation of a set of source trees $\mathcal{X}$
$\eta_M(Q')$	Percentaged overhead of the average code length per symbol of the M coder relative to the entropy rate of a random process with distribution of conditional probabilities of one according to Q'
$\eta_P(Q')$	Percentaged overhead of the average code length per symbol of the P coder relative to the entropy rate of a random process with distribution of conditional probabilities of one according to Q'
$\bar{H}(\{X_i\})$	Entropy rate of random process $\{X_i\}$
$H(X)$	Entropy of random variable X
$H(X;Y)$	Cross entropy of Y with respect to X
$\bar{\mathcal{H}}(Q')$	Entropy rate of a random process for that the conditional probabilities of one are distributed according to Q'
$\mathrm{Hu}(\cdot)$	V2V code with a code tree derived by the Huffman algorithm
$k(\sigma)$	Code word associated with source word σ of a V2V code
$K(Q')$	Random variable over $\mathcal{Q} \times \mathcal{B}$, derived from the distribution of conditional probabilities of one of a random process
$\bar{\ell}(X)$	Average length of words produced by a source X that has binary words as alphabet
$\bar{\ell}_B(X)$	Average code length of encoding random variable X with coder B
$\ell(x)$	Length of binary word x

$\ell_B(\{G_i\}, (g_1, g_2, \ldots, g_n))$ Encoded length of the sequence $g_1, g_2, \ldots, g_n$ of random process $\{G_i\}$ produced by binary coding engine B

$\bar{L}(K(Q'), B)$ Average code length per symbol of encoding $K(Q')$ with binary coding engine B

$L_M(r, \omega_k)$ Lookup table of the M coder that stores the precalculated LPS range values for probability ω_k and range r as specified in H.264/AVC and H.265/HEVC

$\bar{\ell}_M(\Omega_k, Q')$ Average code length produced by the M coder when encoding coding bins for that a particular ω_k occurs amongst all coding bins of $K(Q')$

$\ell'_M(X, r)$ Average code length of the M coder for encoding random variable X when the range value is r

$\bar{L}(\{G_i\}, B)$ Average code length per symbol of encoding random process $\{G_i\}$ with binary coding engine B

$\bar{\mathcal{L}}_B(Q')$ Average code length per symbol of encoding a random process for that the conditional probabilities of one are distributed according to Q' with binary coding engine B

$M(g_1, g_2, \ldots, g_n)$ Numbers of ones in the sequence of coding bins $g_1, g_2, \ldots, g_n$

$M(x)$ Numbers of ones in binary word x

$N(g_1, g_2, \ldots, g_n)$ Numbers of zeros in the sequence of coding bins $g_1, g_2, \ldots, g_n$

$N(x)$ Numbers of zeros in binary word x

$\bar{p}_{\{X_i\}}(\cdot)$ Average of conditional probability mass functions of random process $\{X_i\}$

$p_{X_i}(\cdot|\mathbf{x}_{i-1})$ Conditional probability mass function $p_{X_i}(\cdot|x_{i-1}, ..., x_1)$

$\pi(\cdot)$ Permutation function

$p_X(\cdot)$ Probability mass function of random variable X

$p_{\{X_i\}}(\cdot)$ Joint probability mass function of random process $\{X_i\}$

$\mathrm{Pm}(\cdot)$ V2V code with a code tree derived by the package merge algorithm

$Q(\{G_i\})$ Random variable that describes the distribution of the conditional probabilities of one of random process $\{G_i\}$

$Q(\{G_i\}, (g_1, g_2, \ldots, g_n))$ Random variable that describes the distribution of the conditional probabilities of one $p_{G_1}(1)$, $p_{G_2}(1|g_1), \ldots, p_{G_n}(1|\mathbf{g}_{n-1})$ that occur for the sequence $g_1, g_2, \ldots, g_n$ of random process $\{G_i\}$

$R(C)$ Redundancy of code tree C

$R_B(X)$ Redundancy of encoding random variable X with coder B

$\mathrm{Rg}(\{G_i\})$ Random variable over $\mathbb{I}$ that describes the distribution of the range values that occur for random process $\{G_i\}$

$\mathrm{Rg}(K(Q'))$ Random variable over $\mathbb{I}$ that describes the distribution of the range values that occur when encoding $K(Q')$

$\mathrm{Rn}(\mathbf{g}_{i-1})$ Range value as present in the M coder before encoding coding bin g_i

$\mathrm{Rs}(\cdot)$ Source tree generated with the prefix merge algorithm from a canonical representation

$\mathrm{RV}(x)$ Binary random variable with pmf $p_{\mathrm{RV}(x)}(1) = x$

$\mathrm{St}(v)$ Source tree of V2V code v

$\vartheta(\{G_i\}, (g_1, g_2, \ldots, g_n), \omega_k, r)$ Subsequence of coding bins in the sequence $g_1, g_2, \ldots, g_n$ for that the associated conditional probability of one equals ω_k and for that the associated range value equals r

$T(r, (\omega_k, g))$ State transition of the Markov chain over the range values of the M coder

$\mathcal{U}_\ell(\{G_i\}, \omega_k, r)$ Average relative frequency of coding bins of $\{G_i\}$ for that the conditional probability of one equals ω_k and for that the range value of the M coder prior to encoding equals r

$\mathcal{U}_M(\{G_i\}, \omega_k, r)$ Average relative frequency of ones that occur in coding bins of $\{G_i\}$ for that the conditional probability of one equals ω_k and for that the range value of the M coder prior to encoding equals r

$\mathcal{U}_N(\{G_i\}, \omega_k, r)$ Average relative frequency of zeros that occur in coding bins of $\{G_i\}$ for that the conditional probability of one equals ω_k and for that the range value of the M coder prior to encoding equals r

$\mathcal{W}_\ell(\{G_i\}, \omega_k)$ Average relative frequency of coding bins of $\{G_i\}$ for that the conditional probability of one equals ω_k

$\mathcal{W}_M(\{G_i\}, \omega_k)$ Average relative frequency of ones that occur in coding bins of $\{G_i\}$ for that the conditional probability of one equals ω_k

$\mathcal{W}_N(\{G_i\}, \omega_k)$ Average relative frequency of zeros that occur in coding bins of $\{G_i\}$ for that the conditional probability of one equals ω_k

$\zeta(\{G_i\}, (g_1, g_2, \ldots, g_n), \omega_k)$ Subsequence of coding bins in the sequence $g_1, g_2, \ldots, g_n$ for that the associated conditional probability of one $p_{G_1}(1)$, $p_{G_2}(1|g_1), \ldots, p_{G_n}(1|\mathbf{g}_{n-1})$ equals ω_k

Notation

$\mathbf{x}_i$	The sequence of symbols $x_i, x_{i-1}, \cdots, x_1$
$\mathcal{A}^n$	n-fold Cartesian power of set $\mathcal{A}$
$\lvert\mathcal{A}\rvert$	Number of elements in set or multiset $\mathcal{A}$
$\{\cdot\}_b$	A multiset
$x^\frown y$	Concatenation of two binary words x and y

Bibliography

[1] "Cisco Visual Networking Index: Forecast and Methodology, 2015-2019". *Cisco White Paper*. May 2015.

[2] ITU-T. "Rec. H.264: Advanced video coding for generic audiovisual services (AVC)". *International Telecommunication Union*. Mar. 2009.

[3] ITU-T. "Rec. H.265: High efficency video coding (HEVC)". *International Telecommunication Union*. Apr. 2013.

[4] H. Schwarz, T. Schierl, and D. Marpe. "Block Structures and Parallelism Features in HEVC". in *High Efficiency Video Coding: Algorithms and Architectures*. Ed. by V. Sze, M. Budagavi, and G. J. Sullivan. Integrated Circuits and Systems. *Springer*, 2014. Chap. 3, pp. 49–90. DOI: 10.1007/978-3-319-06895-4_3.

[5] D. Marpe, H. Schwarz, and T. Wiegand. "Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard". *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 620–636, July 2003. ISSN: 1051-8215. DOI: 10.1109/TCSVT.2003.815173.

[6] D. Marpe, H. Schwarz, T. Wiegand, and H. Kirchhoffer. "Entropy coding". *US Patent 8,907,823*. Dec. 2014.

[7] D. Marpe, H. Schwarz, and T. Wiegand. "Entropy coding in video compression using probability interval partitioning". in *Proceedings of the Picture Coding Symposium (PCS 2010), Nagoya, Japan*. Dec. 2010, pp. 66–69. DOI: 10.1109/PCS.2010.5702580.

[8] T. Wiegand and H. Schwarz. "Source Coding: Part I of Fundamentals of Source and Video Coding". *now publishers Inc.*, 2011. DOI: 10.1561/2000000010.

[9] D. Marpe, H. Schwarz, and T. Wiegand. "Novel entropy coding concept". *Joint Collaborative Team on Video Coding*. JCTVC-A032, Dresden, Germany, 1st Meeting, Mar. 2010.

[10] E. Meron and M. Feder. “Finite-memory universal prediction of individual sequences”. *IEEE Transactions on Information Theory*, vol. 50, no. 7, pp. 1506–1523, July 2004. ISSN: 0018-9448. DOI: 10.1109/TIT.2004.830749.

[11] C. C. Holt. “Forecasting seasonals and trends by exponentially weighted moving averages”. *International Journal of Forecasting*, vol. 20, no. 1, pp. 5–10, 2004. DOI: 10.1016/j.ijforecast.2003.09.015.

[12] E. Belyaev, M. Gilmutdinov, and A. Turlikov. “Binary Arithmetic Coding System with Adaptive Probability Estimation by ”Virtual Sliding Window””. in *2006 IEEE Tenth International Symposium on Consumer Electronics. ISCE ’06*. 2006, pp. 1–5. DOI: 10.1109/ISCE.2006.1689517.

[13] T. Cover and J. A. Thomas. “Elements of Information Theory”. Ed. by D. L. Schilling. Wiley Series in Telecommunications. *John Wiley & Sons, Inc.*, 1991. DOI: 10.1002/0471200611.

[14] P. Zhang. “Fast CABAC decoding architecture”. *Electronics Letters*, vol. 44, no. 24, pp. 1394–1395, Nov. 2008. ISSN: 0013-5194. DOI: 10.1049/el:20082126.

[15] Y. Yi and I.-C. Park. “High-Speed H.264/AVC CABAC Decoding”. *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 17, no. 4, pp. 490–494, Apr. 2007. ISSN: 1051-8215. DOI: 10.1109/TCSVT.2007.893831.

[16] D. Marpe, H. Kirchhoffer, and G. Marten. “Fast Renormalization for H.264/AVC Arithmetic Coding”. in *Proceedings of the 14th European Signal Processing Conference (EUSIPCO 2006), Florence, Italy*. Sept. 2006.

[17] S. Lloyd. “Least squares quantization in PCM”. *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, Mar. 1982. ISSN: 0018-9448. DOI: 10.1109/TIT.1982.1056489.

[18] J. Max. “Quantizing for minimum distortion”. *IRE Transactions on Information Theory*, vol. 6, no. 1, pp. 7–12, Mar. 1960. ISSN: 0096-1000. DOI: 10.1109/TIT.1960.1057548.

[19] D. Greene, F. Yao, and T. Zhang. “A linear algorithm for optimal context clustering with application to bi-level image coding”. in *International Conference on Image Processing (ICIP 98)*. Vol. 1. Oct. 1998, 508–511 vol.1. DOI: 10.1109/ICIP.1998.723548.

[20] S. Forchhammer, X. Wu, and J. Andersen. “Optimal context quantization in lossless compression of image data sequences”. *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 509–517, 2004. ISSN: 1057-7149. DOI: 10.1109/TIP.2003.822613.

[21] D. Huffman. “A Method for the Construction of Minimum-Redundancy Codes”. *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952. ISSN: 0096-8390. DOI: 10.1109/JRPROC.1952.273898.

[22] G. Freeman. “Asymptotic convergence of dual-tree entropy codes”. in *Proceedings of the Data Compression Conference. DCC ’91*. 1991, pp. 208–217. DOI: 10.1109/DCC.1991.213360.

[23] B. P. Tunstall. “Synthesis of noiseless compression codes”. PhD thesis. Georgia Institute of Technology, 1967. URL: http://hdl.handle.net/1853/15041.

[24] F. Ono, S. Kino, M. Yoshida, and T. Kimura. “Bi-level image coding with MELCODE-comparison of block type code and arithmetic type code”. in *IEEE Global Telecommunications Conference and Exhibition ’Communications Technology for the 1990s and Beyond’ (GLOBECOM). IEEE*, 1989, pp. 255–260. DOI: 10.1109/GLOCOM.1989.63977.

[25] D. Marpe, H. Kirchhoffer, M. Siekmann, and C. Bartnik. “Binary arithmetic coding scheme”. *WO Patent App. PCT/EP2013/051,045*. Aug. 2013.

[26] P. G. Howard. “Interleaving entropy codes”. in *Compression and Complexity of Sequences*. 1997, pp. 45–55. DOI: 10.1109/SEQUEN.1997.666902.

[27] M. Boliek, J. Allen, E. Schwartz, and M. Gormish. “Very high speed entropy coding”. in *IEEE International Conference on Image Processing (ICIP 1994), Austin, TX, USA*. Vol. 3. Nov. 1994, pp. 625–629. DOI: 10.1109/ICIP.1994.413814.

[28] H. Kirchhoffer, B. Bross, A. Henkel, D. Marpe, T. Nguyen, M. Preiss, M. Siekmann, J. Stegemann, and T. Wiegand. “CE1: Report of test results related to PIPE-based Unified Entropy Coding”. *Joint Collaborative Team on Video Coding*. JCTVC-G633, Geneva, Switzerland, 7th Meeting, Nov. 2011.

[29] M. Winken, S. Bosse, B. Bross, P. Helle, T. Hinz, H. Kirchhoffer, H. Lakshman, D. Marpe, S. Oudin, M. Preiss, H. Schwarz, M. Siekmann, K. Sühring, and T. Wiegand. “Description of video coding technology proposal by Fraunhofer HHI”. *Joint Collaborative Team on Video Coding*. JCTVC-A116, Dresden, Germany, Apr. 2010.

[30] R. M. Fano. “Transmission of Information: A Statistical Theory of Communication”. *The M.I.T. Press*, 1961.

[31] A. Kiely and M. Klimesh. “A new entropy coding technique for data compression”. *The Interplanetary Network Progress Report*, vol. 42-146, 2001.

[32] A. Kiely and M. Klimesh. “Memory-efficient recursive interleaved entropy coding”. *The Interplanetary Network Progress Report*, vol. 42-146, 2001.

[33] A. Kiely and M. Klimesh. “An adaptable binary entropy coder”. in *Proceedings of the Data Compression Conference. DCC 2001*. 2001, pp. 391–400. DOI: 10.1109/DCC.2001.917170.

[34] B. Macq, X. Marichal, and M. P. Queluz. “Entropy Coding of Segmentations Trees”. in *15th Symposium on Information Theory in the Benelux*. 1994, pp. 282–289.

[35] X. Marichal, B. Macq, and M. P. Queluz. “Generic coder for binary sources: the M-coder”. *Electronics Letters*, vol. 31, no. 7, p. 544, 1995. ISSN: 00135194. DOI: 10.1049/el:19950358.

[36] K. Nguyen-Phi and H. Weinrichter. “A new binary source coder and its application in bi-level image compression”. in *IEEE Global Telecommunications Conference (GLOBECOM'96)*. Vol. 3. *IEEE*, 1996, pp. 1483–1487. ISBN: 0-7803-3336-5. DOI: 10.1109/GLOCOM.1996.591888.

[37] J. Abrahams. “Code and parse trees for lossless source encoding”. in *Compression and Complexity of Sequences*. 1997, pp. 145–171. DOI: 10.1109/SEQUEN.1997.666911.

[38] F. Fabris. “Variable-length-to-variable length source coding: a greedy step-by-step algorithm”. *IEEE Transactions on Information Theory*, vol. 38, no. 5, pp. 1609–1617, 1992. ISSN: 0018-9448. DOI: 10.1109/18.149517.

[39] G. Freeman. “Divergence and the construction of variable-to-variable-length lossless codes by source-word extensions”. in *Proceedings of the Data Compression Conference. DCC '93*. 1993, pp. 79–88. DOI: 10.1109/DCC.1993.253142.

[40] P. R. Stubley. “Adaptive data compression using tree codes”. PhD thesis. University of Waterloo, 1992.

[41] P. R. Stubley. “On the redundancy of optimum fixed-to-variable length codes”. in *Proceedings of the Data Compression Conference. DCC '94*. 1994, pp. 90–97. DOI: 10.1109/DCC.1994.305916.

[42] P. R. Stubley. “Adaptive variable-to-variable length codes”. in *Proceedings of the Data Compression Conference. DCC ’94*. 1994, pp. 98–105. DOI: 10.1109/DCC.1994.305917.

[43] J. Senecal, M. Duchaineau, and K. Joy. “Length-limited variable-to-variable length codes for high-performance entropy coding”. in *Proceedings of the Data Compression Conference. DCC 2004*. 2004, pp. 389–398. DOI: 10.1109/DCC.2004.1281484.

[44] N. M. Blachman. “Minimum cost coding of information”. *Transactions of the IRE Professional Group on Information Theory*, vol. PGIT-3, no. 3, pp. 139–149, Mar. 1954. DOI: 10.1109/IREPGIT.1954.6373407.

[45] R. S. Marcus. “Discrete Noiseless Coding”. MA thesis. Massachusetts Institute of Technology, Dept. of Electrical Engineering, Cambridge, 1957. URL: http://hdl.handle.net/1721.1/35438.

[46] M. Golin and J. Li. “More Efficient Algorithms and Analyses for Unequal Letter Cost Prefix-Free Coding”. *IEEE Transactions on Information Theory*, vol. 54, no. 8, pp. 3412–3424, Aug. 2008. ISSN: 0018-9448. DOI: 10.1109/TIT.2008.926326.

[47] I. Boreico. “Linear Independence of Radicals”. *The Harvard College Mathematics Review*, vol. 2, no. 1, pp. 87–92, 2008.

[48] E. S. Schwartz and B. Kallick. “Generating a Canonical Prefix Encoding”. *Communications of the ACM*, vol. 7, no. 3, pp. 166–169, Mar. 1964. ISSN: 0001-0782. DOI: 10.1145/363958.363991.

[49] J. Connell. “A Huffman-Shannon-Fano code”. *Proceedings of the IEEE*, vol. 61, no. 7, pp. 1046–1047, July 1973. ISSN: 0018-9219. DOI: 10.1109/PROC.1973.9200.

[50] D. S. Hirschberg and D. A. Lelewer. “Efficient Decoding of Prefix Codes”. *Communications of the ACM*, vol. 33, no. 4, pp. 449–459, Apr. 1990. ISSN: 0001-0782. DOI: 10.1145/77556.77566.

[51] L. G. Kraft. “A device for quantizing, grouping, and coding amplitude-modulated pulses”. MA thesis. Massachusetts Institute of Technology. Dept. of Electrical Engineering, 1949. URL: http://hdl.handle.net/1721.1/12390.

[52] S. B. Guthery. “A Motif of Mathematics: History and Application of the Mediant and the Farey Sequence”. *CreateSpace Independent Publishing Platform*, 2010.

[53] L. L. Larmore and D. S. Hirschberg. “A fast algorithm for optimal length-limited Huffman codes”. *J. ACM*, vol. 37, no. 3, pp. 464–473, July 1990. ISSN: 0004-5411. DOI: 10.1145/79147.79150.

[54] G. Longo and G. Galasso. “An application of informational divergence to Huffman codes”. *IEEE Transactions on Information Theory*, vol. 28, no. 1, pp. 36–43, Jan. 1982. ISSN: 0018-9448. DOI: 10.1109/TIT.1982.1056452.

[55] R. Gallager. “Variations on a theme by Huffman”. *IEEE Transactions on Information Theory*, vol. 24, no. 6, pp. 668–674, Nov. 1978. ISSN: 0018-9448. DOI: 10.1109/TIT.1978.1055959.

[56] F. Fabris, A. Sgarro, and R. Pauletti. “Tunstall adaptive coding and miscoding”. *IEEE Transactions on Information Theory*, vol. 42, no. 6, pp. 2167–2180, Nov. 1996. ISSN: 0018-9448. DOI: 10.1109/18.556605.

[57] A. Moffat and A. Turpin. “Compression and Coding Algorithms”. *Kluwer Academic Publishers*, 2002. DOI: 10.1007/978-1-4615-0935-6.

[58] A. G. Akritas and A. W. Strzebonski. “A Comparative Study of Two Real Root Isolation Methods”. *Nonlinear Analysis: Modelling and Control*, vol. 10, no. 4, pp. 297–304, 2005.

[59] A. G. Akritas, A. W. Strzebonski, and P. S. Vigklas. “Improving the Performance of the Continued Fractions Method Using New Bounds of Positive Roots”. *Nonlinear Analysis: Modelling and Control*, vol. 13, no. 3, pp. 265–279, 2008.

[60] *Python programming language, Version 2.6.6*. http://www.python.org.

[61] *SymPy symbolic mathematics framework, Version 0.7.5*. http://www.sympy.org.

[62] J. Katajainen, A. Moffat, and A. Turpin. “A fast and space-economical algorithm for length-limited coding”. in *6th International Symposium on Algorithms and Computation, (ISAAC ’95), Cairns, Australia*. 1995, pp. 12–21. DOI: 10.1007/BFb0015404.

[63] A. Turpin and A. Moffat. “Practical Length-Limited Coding for Large Alphabets”. *The Computer Journal*, vol. 38, no. 5, pp. 339–347, 1995. DOI: 10.1093/comjnl/38.5.339.

[64] A. Turpin. “Efficient prefix coding”. PhD thesis. University of Melbourne, Dept. of Computer Science, 1998. URL: http://trove.nla.gov.au/work/32551427.

[65] A. Moffat and A. Turpin. “Efficient construction of minimum-redundancy codes for large alphabets”. *IEEE Transactions on Information Theory*, vol. 44, no. 4, pp. 1650–1657, July 1998. ISSN: 0018-9448. DOI: 10.1109/18.681345.

[66] M. Liddell and A. Moffat. “Incremental Calculation of Minimum-Redundancy Length-Restricted Codes”. *Communications, IEEE Transactions on*, vol. 55, no. 3, pp. 427–435, 2007. ISSN: 0090-6778. DOI: 10.1109/TCOMM.2007.892446.

[67] H. Kirchhoffer, D. Marpe, H. Schwarz, C. Bartnik, A. Henkel, M. Siekmann, J. Stegemann, and T. Wiegand. “Reduced complexity PIPE coding using systematic v2v codes”. *Joint Collaborative Team on Video Coding*. JCTVC-D380, Daegu, Korea, 4th Meeting, Jan. 2011.

[68] D. Marpe, H. Kirchhoffer, B. Bross, V. George, T. Nguyen, M. Preiss, M. Siekmann, J. Stegemann, and T. Wiegand. “Unified PIPE-based Entropy Coding for HEVC”. *Joint Collaborative Team on Video Coding*. JCTVC-F268, Torino, Italy, 6th Meeting, July 2011.

[69] H. Kirchhoffer, D. Marpe, C. Bartnik, A. Henkel, M. Siekmann, J. Stegemann, H. Schwarz, and T. Wiegand. “Probability interval partitioning entropy coding using systematic variable-to-variable length codes”. in *18th IEEE International Conference on Image Processing (ICIP 2011), Brussels, Belgium*. Sept. 2011, pp. 333–336. DOI: 10.1109/ICIP.2011.6116387.

[70] F. Jelinek and K. Schneider. “On variable-length-to-block coding”. *IEEE Transactions on Information Theory*, vol. 18, no. 6, pp. 765–774, Nov. 1972. ISSN: 0018-9448. DOI: 10.1109/TIT.1972.1054899.

[71] J. G. Michaels and K. H. Rosen. “Applications of discrete mathematics”. *McGraw-Hill, Inc.*, 1991.

[72] R. R. Coifman and M. V. Wickerhauser. “Entropy-based algorithms for best basis selection”. *IEEE Transactions on Information Theory*, vol. 38, no. 2, pp. 713–718, 1992. ISSN: 0018-9448. DOI: 10.1109/18.119732.

[73] JCT-VC. “Test Model under Consideration (TMuC) 0.2”. *Joint Collaborative Team on Video Coding*. Sept. 2010. URL: http://hevc.hhi.fraunhofer.de/svn/svn_HEVCSoftware/tags/0.2.

[74] D. Taubman and M. W. Marcellin. “JPEG2000: Image compression fundamentals, standards and practice”. *Boston: Kluwer Academic Publishers*, 2002. DOI: 10.1007/978-1-4615-0799-4.

[75] S. W. Golomb. “Run-length encodings (Corresp.)” *IEEE Transactions on Information Theory*, vol. 12, no. 3, pp. 399–401, July 1966. ISSN: 0018-9448. DOI: 10.1109/TIT.1966.1053907.

[76] M. Preiss, D. Marpe, B. Bross, V. George, H. Kirchhoffer, T. Nguyen, M. Siekmann, J. Stegemann, and T. Wiegand. “A unified and complexity scalable entropy coding scheme for video compression”. in *19th IEEE International Conference on Image Processing (ICIP 2012), Orlando, FL, USA*. Sept. 2012, pp. 729–732. DOI: 10.1109/ICIP.2012.6466963.

[77] T. Nguyen, D. Marpe, B. Bross, V. George, H. Kirchhoffer, M. Preiss, M. Siekmann, J. Stegemann, and T. Wiegand. “A complexity scalable entropy coding scheme for video compression”. in *Proceedings of the Picture Coding Symposium (PCS 2012), Kraków, Poland*. May 2012, pp. 421–424. DOI: 10.1109/PCS.2012.6213377.

[78] K. McCann, C. Rosewarne, B. Bross, M. Naccari, K. Sharman, and G. J. Sullivan. “High Efficiency Video Coding (HEVC) Test Model 16 (HM16) Improved Encoder Description”. *Joint Collaborative Team on Video Coding*. JCTVC-S1002, Strasbourg, France, 19th Meeting, Oct. 2014.

[79] F. Bossen. “Common test conditions and software reference configurations”. *Joint Collaborative Team on Video Coding*. JCTVC-L1100, Geneva, Switzerland, 12th Meeting, Jan. 2013.

[80] D. Marpe. “Adaptive Context-Based and Tree-Based Algorithms for Image Coding and Denoising”. PhD thesis. University of Rostock, 2004.

[81] J. Hahlbeck and B. Stabernack. “A 4k capable FPGA based high throughput binary arithmetic decoder for H.265/MPEG-HEVC”. in *IEEE International Conference on Consumer Electronics (ICCE 2014), Berlin, Germany*. Sept. 2014, pp. 388–390. DOI: 10.1109/ICCE-Berlin.2014.7034335.

[82] V. Sze, M. Budagavi, and G. J. Sullivan. “High Efficiency Video Coding: Algorithms and Architectures”. *Springer*, 2014. DOI: 10.1007/978-3-319-06895-4.

[83] V. Sze and M. Budagavi. “High Throughput CABAC Entropy Coding in HEVC”. *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1778–1791, Dec. 2012. ISSN: 1051-8215. DOI: 10.1109/TCSVT.2012.2221526.

[84] T. Wiegand, W.-J. Han, B. Bross, J.-R. Ohm, and G. J. Sullivan. “WD3: Working Draft 3 of High-Efficiency Video Coding”. *Joint Collaborative Team on Video Coding*. JCTVC-E603, Geneva, Switzerland, 5th Meeting, Mar. 2011.

[85] K. McCann, B. Bross, S.-i. Sekiguchi, and W.-J. Han. “HM3: High Efficiency Video Coding (HEVC) Test Model 3 Encoder Description”. *Joint Collaborative Team on Video Coding*. JCTVC-E602, Geneva, Switzerland, 5th Meeting, Mar. 2011.

[86] F. Bossen. “Common test conditions and software reference configurations”. *Joint Collaborative Team on Video Coding*. JCTVC-E700, Geneva, Switzerland, 5th Meeting, Mar. 2011.

[87] “HEVC Test Model Software 3.2”. *Joint Collaborative Team on Video Coding*. `https://hevc.hhi.fraunhofer.de/svn/svn_HEVCSoftware/tags/HM-3.2`, July 2011.

[88] G. Bjøntegaard. “Calcuation of average PSNR differences between RD-curves”. *ITU-T Q6/16, Video Coding Experts Group (VCEG)*. VCEG-M33, Austin, TX, USA, 2-4 April 2001, 2001.

[89] A. Roth, H. Kirchhoffer, D. Marpe, and T. Wiegand. “Increasing data throughput in PIPE coding using extended v2v-codes”. in *IEEE International Conference on Consumer Electronics (ICCE 2012), Berlin, Germany*. Sept. 2012, pp. 56–60. DOI: `10.1109/ICCE-Berlin.2012.6336524`.

[90] I.-K. Kim, K. McCann, K. Sugimoto, B. Bross, and W.-J. Han. “HM7: High Efficiency Video Coding (HEVC) Test Model 7 Encoder Description”. *Joint Collaborative Team on Video Coding*. JCTVC-I1002, Geneva, Switzerland, 9th Meeting, Apr. 2012.

[91] F. Bossen. “Common test conditions and software reference configurations”. *Joint Collaborative Team on Video Coding*. JCTVC-I1100, Geneva, Switzerland, 9th Meeting, Apr. 2012.

[92] “HEVC Test Model Software 7.0”. *Joint Collaborative Team on Video Coding*. `https://hevc.hhi.fraunhofer.de/svn/svn_HEVCSoftware/tags/HM-7.0`, May 2012.

[93] H. Schwarz, D. Marpe, T. Wiegand, H. Kirchhoffer, A. Henkel, C. Bartnik, M. Siekmann, and J. Stegemann. “Probability interval partitioning encoder and decoder”. WO Patent App. PCT/EP2011/055,528. Oct. 2011.

Theses

1. The redundancy of a V2V codes cannot be zero for an i.i.d. binary source X for which $0 < p_X(1) < 0.5$ with $p \in \mathbb{R}$.

2. A canonical representation for V2V codes can be defined from which the associated redundancy can unambiguously be derived.

3. From the canonical representation of a V2V code, an actual V2V code can be derived with the so-called *prefix merge* algorithm.

4. It can be shown that a V2V code may be composed of other V2V codes, which leads to the concept of prime and composite V2V codes.

5. A composite V2V code has a unique multiset of prime V2V codes from which it can be composed.

6. The redundancy of composite V2V codes always lies between the redundancies of the prime V2V codes which the composite V2V code consists of.

7. Any algorithm for code tree design that is based on summation, multiplication, and comparison of source word probabilities only, can also be carried out when the source word probabilities are polynomials in $p_X(1)$.

8. An algorithm for code tree design that allows the source word probabilities to be polynomials in $p_X(1)$ yields a number of results with associated contiguous, non-overlapping probability intervals with analytical exact interval boundaries.

9. From the canonical representation of a Tunstall code, an actual Tunstall code can always be derived that is composite.

10. For finding minimum redundancy V2V codes by exhaustive search over source trees, it is sufficient to only consider a set of canonical source trees.

11. There exist V2V codes that can be described by a simple construction rule and that have a relatively low redundancy, which leads to the concept of *systematic V2V codes*.

12. V2F and F2V codes have a systematic disadvantage in terms of redundancy when compared to V2V codes.

13. Minimum redundancy V2V codes with the source and code tree height limited to 6 have a low redundancy for $p_X(1) > 0.1$.

14. For $p_X(1) \to 0$, unary-to-rice codes are minimum redundancy V2V codes.

15. By combining unary-to-rice codes with minimum redundancy V2V codes for which the source and code tree height is limited to 6, a P coder can be designed for the H.265/HEVC video compression standard that has a percentaged overhead of only 0.3% when 8 bin coders are used.

16. When ideal binary arithmetic bin coders are used for PIPE coding in H.265/HEVC, the overhead is less than 0.3% for 6 bin coders and less than 0.1% for 10 bin coders.

17. The redundancy of the M coder of CABAC, as it is used in H.265/HEVC for non-bypass bins, has its highest value at $p_{LPS} = 0.5$ and it decreases with decreasing p_{LPS}.

18. The M coder of CABAC, as it is used in H.265/HEVC, has a percentaged overhead of approximately 0.094%.

19. The distribution of the range values of the M coder depends on the multiplicity of prime factor 2 in the range value.

20. The chunk-based multiplexing technique introduces numerous options for increasing the achievable throughput by parallel processing.

21. With the combination of chunk-based multiplexing and V2V codes, a nonparallelized software implementation of a decoder can run faster than an optimized implementation of the M coder.