Robust Source Coding with Generalised T-Codes

Ulrich Günther

A thesis submitted in partial fulfilment of the requirements
for the degree of
Doctor of Philosophy
in
Computer Science

The University of Auckland, 1998

Abstract

This thesis presents a range of novel and improved results in the area of source coding with T-Codes, based on a thorough review of the recursive structure of the T-Codes.

T-Codes were introduced by Mark Titchener in 1984. They are variable-length codes, similar to the well-known Huffman codes and may thus be used for compression by source coding. T-Codes have also been noted for their self-synchronisation properties, which result from their recursive construction.

This thesis reviews and formalises the theory of generalised T-Codes and their recursive construction. It extends an existing recursive storage concept, the binary depletion codes, as the "T-depletion codes". These are shown to provide a unique fixed-length representation for T-Codes, which may be used in a recursive encoder/decoder scheme for T-Codes.

It is shown that the T-depletion code format accommodates unique representations for all proper prefixes of T-Code codewords, thus covering the complete state space of an encoder or decoder. For use in decoder applications, the thesis derives a recursive conversion between T-depletion codes and a contiguous integer index.

A new simplified method for the calculation of the expected synchronisation delay (ESD) is introduced. Finally, the thesis develops a recursive search algorithm to find the T-Code set that most efficiently encodes a source.

Preface and Acknowledgements

The present thesis is, if the author has counted correctly, the fourth PhD thesis in the area of T-Codes, and the first in the area of generalised T-Codes.

While a PhD thesis is always an individual's research record, few if any are written in isolation. This thesis is no exception. In fact, given the circumstances under which the project was conducted, I owe thanks to rather more people than the average PhD student.

The list should perhaps start with Gavin Higgie, who first introduced me to T-Codes in a graduate class that he was teaching. Then it was Gary Bold who alerted me to the fact that Mark Titchener had returned to New Zealand and was teaching at Auckland University's Tamaki Campus. Mark was very supportive from the start and right throughout the thesis process, and his support was not restricted to university matters alone. I hope very much that this thesis rewards his confidence in me. Throughout my time at Tamaki, Mark has been a continuous source of new ideas. While he often had a hard time convincing a sceptic like me, I must say that his "gut feel" has often proved to be more reliable than my (and other's) doubts. Many of the new and extended results in this thesis have their origin in our brainstorming sessions.

Radu Nicolescu helped a great deal in formalising the notation used in this thesis, and his uniqueness theorem proved to be an extremely useful tool for this thesis. His interest in my work was always appreciated. Peter Hertling deserves a special mention for his suggestions for cleaning up some of the proofs in Chapters 4 and 6.

Professor Cris Calude deserves thanks for his ongoing interest in my work, and his help and feedback regarding several of the publications that this thesis is based on. Professor Reinhard Klette was kind enough to read some early chapters of this thesis - his comments on style and structure have had a profound influence on the shape of this thesis. Professor Clark Thomborson also gave helpful comments — his assistance with administrative and policy matters and an always open ear for PhD students' problems have been a great reassurance.

Thanks go to the Department of Computer Science, in particular its present head Professor Peter Gibbons, for providing me with a departmental stipend and with the necessary hardware to keep my tutoring load under control so I could write this thesis. Penny Barry also deserves a mention for making the administrative side of this run so smoothly. Bruce Benson, Paul Burkimsher, Paul Bonnington, Rob

Burrowes, Mano Manoharan, Clare West, Gary Wong, and Edouard Poor have had their part in saving the computational side from the brink of disaster a few times.

Working at Tamaki Campus has been a great experience. Its interdisciplinary family atmosphere has been a source of ideas. My special thanks go to Alastair McNaughton whose comments made me re-think aspects of the search algorithm in Chapter 10. As a result, the algorithm's usability for real-world problems has improved significantly. Many other staff from computer science, mathematics, physics, and statistics have had input into the notation used.

Being one of the first PhD students at Tamaki campus meant that I often did not fit into established categories. Professor Chris de Freitas offered a lot of support during the early stages of my research, while he was Head of Division. Steve Chaney and others from ITSS often put in unscheduled jobs to keep me online. The administrative staff at Tamaki have also suffered much under me and deserve a mention.

Library support is vital for any research, especially when most of the relevant literature is located on another campus, and one's field of research spans several disciplines. Lynley Stone and her team at the Tamaki Library have bent over backwards many times to make the physical distance disappear. Thanks here go especially to Brenda Dwane, Danielle Carter, Hester Mountifield, and Russell Tuffery for going the extra mile on so many occasions.

Thanks also go to the many staff and fellow students who have helped to keep me sane. My motivation also owed a lot to my friends in amateur radio and elsewhere.

Finally, I am indebted to my family for their continuous love and support over time and distance.

Auckland, February 1998 Ulrich Günther

Contents

CHAPTER 1

Introduction

This introduction sets the scope for the rest of the thesis. It gives some motivation and introduces basic notation and conventions.

1.1 Motivation

Communication of information through space or time is a fundamental engineering problem. Whether we wish to transmit telemetry data from a satellite to earth, a voice signal over a telephone line, or store a file on a computer disk for later retrieval, we can always identify three essential components:

1. an information source which has information that is to be transmitted or stored;

2. an information sink that receives and utilises the information in some way.

3. a channel connecting the source and the sink.

The channel carries the source's information in some way or another to the sink. In the context of this thesis, a "channel" denotes both a channel through space as well as a channel through time. For example, we would regard a computer disk as a channel through time: information is first written to it and later retrieved.

All real-life channels are in some way *bandlimited*. In the case of channels through space this means that a channel can only pass a limited amount of information to the sink during a given time interval. In the case of channels through time, the limited *bandwidth* manifests itself — if we take the disk example — in the form of unwelcome error messages such as "disk full".

The question that usually arises is: "Given an information source and sink, and a channel, how can the interface between the channel and the source/sink be designed such that we can communicate the maximum amount of information through the channel, with a minimum number of errors, and at minimum cost?"

There is no single answer to this question, as the above goals are generally in conflict with each other. Any individual solution will always be a compromise, depending on the particular source, sink, and channel involved. Also, the number of errors that may be acceptable varies from application to application.

Thus, the best that can be achieved is an addition to the collection of known compromises that have proven to be useful. This thesis attempts to do just that.

The particular communication scenario on which this thesis is based assumes that the sink can accommodate a modest amount of errors, such that no extra channel coding (such as checksums, etc.) is required. Our scenario further assumes that a high (but not necessarily maximal) amount of information needs to be passed through the channel. At the same time, we require that the cost at the source and sink be low.

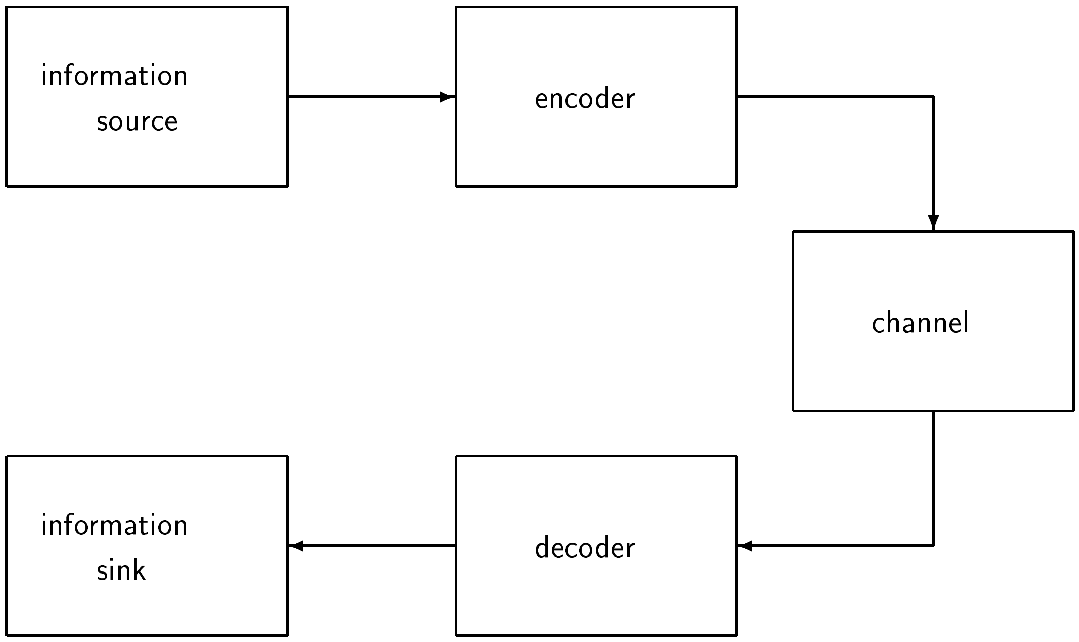


Figure 1.1. *Our model of a communication process involving an information source, an encoder, a channel, a decoder, and an information sink.*

As mentioned above, practical applications that fall within this category are, e.g., digital telephony and digital image/video transmission. In these cases, the ultimate information sink is the human ear or eye. Both the ear and the eye are known to integrate well over a limited amount of errors.

In our communication scenario, the interface between source and channel is called the encoder, and the interface between channel and sink is called the decoder. Figure 1.1 illustrates this concept.

1.2 How this Thesis is Organised

This thesis is a collection of solutions to various problems related to source coding with T-Codes. As such, its chapters do not all have to be read in sequence. The main results presented are:

1. a treatment of T-depletion (Chapter 4) and pseudo-T codes (Chapter 6). These discussions build on all chapters up to Chapter 4.

2. a contiguous-range index conversion for T-depletion codes (Chapter 5), which builds on the T-depletion codes.

3. a novel storage scheme for variable-length codes (Chapter 6), based on the T-depletion codeword format and the pseudo-T codes.

4. a novel way for the calculation of the expected synchronisation delay (ESD) for T-Code sets (Chapter 9), based on the hierarchical coding aspect of T-Codes discussed in Chapter 7. The treatment of this topic uses results from all chapters up to Chapter 4, as well as the discussion of T-Code synchronisation in Chapter 8.

5. a search algorithm to find a T-Code set that provides the best possible coding efficiency for a given source probability distribution. This is discussed in Chapter 10 and uses the results from Chapters 1 to 4, and Chapter 6.

A summary, including an outlook on possible avenues for future research, is given in Chapter 11.

1.3 Basic Notation and Conventions

A significant part of this thesis is devoted to developing a theoretical framework for discussing the construction, storage, handling, and performance of T-Code sets. While digital communication provides the motivation, this theoretical model of T-

Codes is much more general and not restricted to binary codes. This both motivates and necessitates a somewhat more mathematically formal approach.

The notation that will be used here represents a compromise between formal correctness, tradition, and space efficiency. It was chosen against the background of this thesis, which includes previous works, but also other ongoing work at the time. The author is well aware that some of it, at least in the context of this thesis, may seem a bit cumbersome and overly complicated. However, in most cases this was necessary to avoid ambiguities in other circumstances.

Only the basic notation is introduced here. Notation specific to more advanced concepts will be introduced together with the concepts, i.e., when it is required.

The end of a proof for a theorem, lemma, or corollary is marked by a square box at the end of a line. $\square$

The cardinality of a set X is denoted as $\#X$. The subtraction of sets is denoted by the backslash "$\backslash$", i.e., $X \backslash Y$ denotes the set that contains all elements of X that are not in the set Y. Similarly, the union of sets will be denoted by the "$\cup$"-symbol, such that $X \cup Y$ denotes the set of all elements that are in X or Y. The "$\cup$"-symbol may also be used in the form $\bigcup_i X_i$ which denotes the union of all sets X_i. For some set X, $\mathcal{P}(X)$ denotes the set of all subsets of X. The "$|$" symbol in the set notation used stands for "for which": e.g., $\{n|n > 3\}$ is read as "the set of all n for which n is greater than 3".

Expression from logic used include $\vee$ (logical OR), $\wedge$ (logical AND), $\forall$ ("for all…"), and $\exists$ ("there exists at least one…").

All code sets to be discussed are based on a finite alphabet S, which we regard as a set of $\#S$ *symbols*. In many examples, S will be the binary alphabet $S = \{0, 1\}$. The binary alphabet is generally chosen in order to keep examples compact and easy to understand. However, unless specified otherwise, there is no principal limit on the cardinality of S.

Symbols may be concatenated to form strings. The empty string is denoted as λ. The concatenation of two strings x and y is denoted as xy. The concatenation of n strings x, where n is a non-negative integer, is denoted as x^n. $x^0 = \lambda$ for any string x. The length of a string x measured in symbols from S is denoted $|x|$, and $|\lambda| = 0$.

For simplicity, this thesis does not explicitly distinguish between alphabet symbols and strings - a symbol is simply regarded as a string with length 1.

In the context of strings, $S^\star$ denotes the set of all finite strings that can be generated by concatenation of symbols from S. $S^\star$ contains the empty string λ. Where λ is explicitly excluded, the remaining set is denoted as S^+.

A *code set* C in the context of this thesis is a finite subset of S^+. Its elements will be called *codewords* or *words*. Similarly, $C^\star$ and C^+ denote the set of finite strings generated by concatenation of elements from C, including/excluding λ.

Furthermore, following a convention similar to Hamming [23], the term "symbol" will also be used in the context of an information source which emits "source symbols" (as opposed to the "channel (alphabet) symbols" from S). These source symbols are encoded as codewords, which in turn are strings from S^+. However, in each instance it should be obvious from the context which type of "symbol" is being referred to.

If for a string $x \in C^\star$ there is another string $y \in C^\star$ such that $z = xy \in C^\star$, we say that "x is a prefix of z (over C)". This is denoted as $x \preceq_C z$. If we demand that $y \in C^+$, i.e., $y \neq \lambda$, this is denoted as $x \prec_C z$ and we say that "x is a proper prefix of z (over C)". Similarly, if we wish to express that x is not a prefix of z over C, and $x \neq z$, then this is denoted as $x \not\preceq_C z$. It should be noted here that this notation is only valid in circumstances where z has a unique decoding over C. This is the case whenever C is prefix-free (see the following section).

Similarly, we define suffixes: if for a string $x \in C^\star$ there is another string $y \in C^\star$ such that $z = xy \in C^\star$, we say that "y is a suffix of z (over C)". This is denoted as $z \succeq_C y$. Proper suffixes ($x \neq \lambda$) are similarly denoted $z \succ_C y$ etc.

Provided that a unique decoding over C exists for some string $x \in C^\star$, we can define the length of x, measured in codewords from C. We denote this length as $|x|_C$.

As mentioned in the introduction, the communication model used in this thesis can be applied to both space and time. For the sake of simplicity and without loss of generality, the terminology here will generally assume communication in space. That is, we will presume that the information source and sink are spatially separated, and that the channel propagation delay is negligible. Hence, e.g., the term "symbol rate" — rather than "symbol density" — will be used.

1.4 Code Sets and their Properties

In the context of the problems addressed in this thesis, we may generally assume that the information source, its statistics, and the channel alphabet S are known.

1.4.1 Unique and Instantaneous Decodability, Prefix-Freeness, and Completeness of Codes

In general, a code C is a subset of S^+. We may use C to encode a set X, comprising all source symbols σ_i, by defining a mapping ϵ :

$$\epsilon : \ X \longrightarrow C. \tag{1.1}$$

We call $\epsilon(\sigma_i)$ the **encoding** of σ_i.

Some authors [2] will not call C a code just because C is a subset of S^+, but require further that C be *uniquely decodable* as well. This is not an unreasonable requirement in many practical cases, where a series of words from C form a symbol stream over S. Unique decodability is guaranteed if C is **prefix-free**, i.e., if there is no word in C that is the prefix of another word in C.

Hamming [23] actually calls this property "instantaneous" and explicitly distinguishes it from "uniquely decodable". The example he gives for a uniquely decodable but non-instantaneous code, however, assumes that one can somehow go to the end of a transmitted message and then decode backwards. This in turn requires a way of signaling to the receiver that a message has ended, which is not always possible. It is therefore often easier — and for our purposes entirely sufficient — to simply test for prefix-freeness and infer unique decodability from this.

The prefix-freeness of the T-Codes plays an important role in this thesis, and is proven in Chapter 2. However, as mentioned before, the convention adopted here is that any subset of S^+ may be called a code. The reason for this should become obvious in Chapter 4, where the pseudo-T codes are introduced, which correspond to a subset of S^+ that is not prefix-free.

A prefix-free code C satisfies the Kraft inequality:

$$\sum_{x \in C} \#S^{-|x|} \leq 1 \tag{1.2}$$

Another property of some prefix-free codes is that of **completeness**. Some authors also call complete codes **exhaustive** [13, 26]. This thesis uses the term "complete" as it seems to be the predominant term used in the information sciences.

A prefix-free, complete code C may be characterised in two ways:

1. for any string $x \in S^+ \backslash C$, the code set $(C \cup \{x\})$ is not prefix-free.

2. any sufficiently long[1] string in S^+ may be written as a unique concatenation xy of a codeword $x \in C$ and a string $y \in S^\star$.

The latter property is particularly important, as it guarantees that *any* sufficiently long input into a decoder for a complete code will result in some valid output.

If C is represented by a "decoding tree" (see Chapter 3), a prefix-free code corresponds to a decoding tree where only the leaf nodes are codewords, and a complete code is synonymous to a decoding tree where all branch nodes are each fully populated with $\#S$ outgoing branches.

For prefix-free complete codes, the Kraft inequality becomes an equality:

$$\sum_{x \in C} \#S^{-|x|} = 1. \tag{1.3}$$

1.4.2 Coding Efficiency and Compression

The Shannon theorem [39, 40] places an upper bound ρ_m on the number of symbols from the channel alphabet S that we can communicate through a channel with bandwidth ω in a given amount of time, with signal-to-noise ratio η:

$$\rho_m \leq \omega \log_{\#S}(1 + \eta). \tag{1.4}$$

The number ρ of symbols from S that have to be communicated depend on several factors:

- the rate at which the source emits source symbols;
- the (possibly conditional) probability of a source symbol being emitted;

[1] In this context, a string is "sufficiently long" if but not necessarily only if it is at least as long as the longest codeword in C.

- which $C \subset S^+$ is chosen to encode the source.

In many practical circumstances, the first two of these are determined by the source and cannot be controlled. In fact, for many practical applications, the probability of occurrence of a source symbol can only be estimated. However, the encoding is often open to choice, and it is this parameter that can sometimes lead to a considerable reduction in the symbol rate on the channel.

Similarly, the bandwidth ω and the SNR η are often hard constraints. Thus, choosing a suitable encoding method is in some cases the *only* way of controlling the number of symbols from S that are communicated.

The "standard" way of encoding a source is to use block codes, where $|\epsilon(\sigma_i)|$ is constant, i.e., independent of σ_i. This does not pay any attention to the source's statistics, however.

"Source coding", on the other hand, takes source statistics into account, thus resulting in "compression" when compared to a block encoding.

In 1952, Huffman [30] presented an algorithm for the construction of variable-length codes from the probabilities of occurrence of the source symbols. These "Huffman codes", arguably the best-known example of source coding, assume that the source symbols are mutually independent, i.e., that the probability of occurrence of the next symbol that the source will emit does not depend on previously emitted symbols.

However, for most real-life sources, there is some correlation between source symbols. This renders the Huffman codes suboptimal, and approaches such as the Lempel-Ziv algorithm [54] frequently lead to an overall better compression. However, in a number of cases there are good arguments for adhering to an algorithm that encodes each source symbol individually. For simplicity, or just as a best estimate, one can often assume that the source symbols are independent, which usually still results in a significant compression gain.

The general source coding model for T-Codes is based on this same assumption and is discussed in Chapter 10.

1.4.3 Other Aspects of Coding

Coding efficiency and compression ratios are only one aspect under one may wish to select a code for a particular application. There are other aspects, too, two of which play an important role in this thesis:

1. robustness of codes. In many practical applications, the channel introduces errors into the received symbol stream. A source code that inherently helps a decoder to recover from errors may be desirable in some applications.

2. encoder and decoder complexity. It is generally desirable to have codes which can be encoded and decoded quickly with as few computational resources (such as storage) as possible. This is a issue especially when hardware implementations are involved.

1.5 A Brief History of T-Codes

As a subset of all possible Huffman code sets, T-Codes have inadvertently been in use for a long time. However, their special properties, in particular self-synchronisation and the T-augmentation construction, were first recognised and presented [44] by Mark Titchener while he was a graduate student. He later completed a PhD [46] on the subject, in which he also introduced a fixed -length representation. Together with Jeff Hunter, he developed the theory of T-Code synchronisation [51], which is revisited and expanded in this thesis.

Mark Titchener's ideas have been picked up by several people. In 1989, a UK-based research group, M. Darnell, B. Honary, and F. Zolghadr, suggested the use of T-Codes in the construction of a statistical real-time channel evaluation system [7]. Gavin Higgie completed a PhD on T-Codes in 1991 [26], in which he compares the synchronisation performance of a large number of possible T-Code sets, and suggests some hardware solutions for the practical implementation of T-Code decoders. Some of his graduate students have since presented theses or project work on T-Codes [10, 24, 25, 5], mostly in the area of fax and image coding. Gavin Higgie himself has also remained active in the area, elaborating further on his thesis topic [27].

The original T-Code concept as presented by Mark Titchener restricted itself to "simple" T-augmentation. In 1986, he came to the conclusion that the important results, in particular with regard to self-synchronisation, held true for a much wider class of codes. This resulted much later in a paper introducing the generalised T-Codes [49], and forms the basis of the research for this thesis.

Radu Nicolescu [36] has contributed significantly to the understanding of T-Code codeword structure by proving the uniqueness of the longest codewords in T-Code sets.

T-Codes have always offered "spin-offs" for general variable-length coding. Gavin Higgie developed a decoder model for general variable-length codes that "traps" the final codeword between pointers [26]. Mark Titchener realised that the synchronisation behaviour of most variable-length codes could be approximated by that of similar T-Code sets [50]. Finally, the generalised T-depletion code format presented in this thesis and in [18, 19] offers a convenient storage and handling format for general variable-length codes.

CHAPTER 2

An Introduction to T-Codes

This chapter introduces the concept of T-augmentation and shows how it may be used to obtain T-Code sets. T-Code properties such as prefix-freeness, completeness and the cardinality of T-Code sets are discussed. A conversion method for a more restricted notation used by other authors is also given.

2.1 T-Augmentation

T-augmentation is the central “tool” in the construction of T-Code sets. However, since its application is not restricted to T-Code sets, we may define it separately:

Definition 2.1.1 (T-Augmentation)

The mapping

$$\alpha(X, p, k) : \mathcal{P}(S^{\star}) \times S^{\star} \times \mathbb{N} \rightarrow \mathcal{P}(\mathrm{S}^{\star})$$

is called a **T-augmentation** *of X iff $p \in X$, $k \in \mathbb{N}^{+}$, and*

$$\alpha(X, p, k) = \{x | x = p^{k'} s \text{ where } s \in X \backslash \{p\} \wedge 0 \leq k' \leq k\} \cup \{p^{k+1}\}. \qquad (2.1)$$

If $\alpha(X,p,k)$ is a T-augmentation, we write $X_{(p)}^{(k)} = \alpha(X,p,k)$. We call $X_{(p)}^{(k)}$ a **T-augmented set**. *The string p is said to be the* **T-prefix**, *and the integer k is called the* **T-expansion parameter** *for the T-augmentation.*

This definition goes beyond the original set "augmentation" algorithm proposed by Titchener in [44]. His original definition did not include the T-expansion parameter k that he added later [49]. We propose to call the original algorithm "simple T-augmentation" and may treat it simply as a special case of the above definition, with $k = 1$. The algorithm proposed above is also known as "generalised T-augmentation".

Simple T-augmentation and the resulting simple T-Code sets have been investigated by others who developed applications such as encoders and decoders [26], real-time channel evaluators [7], and image coding [24] techniques.

Example 2.1.2 (T-Augmentation)
Consider the set

$$X = \{a, ba, bb, bc, ca, cb, cc\}$$

based on $S = \{a, b, c\}$ and choose, say, $p = bb$ and $k = 2$ for the T-augmentation. Thus, the T-augmented set

$$\begin{aligned} X_{(bb)}^{(2)} \quad = \quad & \{a, ba, bc, ca, cb, cc, bba, \\ & bbba, bbbc, bbca, bbcb, \\ & bbcc, bbbba, bbbbba, \\ & bbbbbb, bbbbbc, bbbbca, \\ & bbbbcb, bbbbcc\}. \end{aligned}$$

2.2 T-Code Sets

As mentioned above, T-augmentation is central in the construction of T-Code sets. T-Code sets are defined as follows:

Definition 2.2.1 (T-Code Sets)

We define any finite alphabet S to be a **T-Code set** *at* **T-augmentation level** *0. A set $X \subset S^+$ that can be derived from S by a finite series of n T-augmentations with T-prefixes $p_1, p_2, \ldots, p_n$ and T-expansion parameters $k_1, k_2, \ldots, k_n$ respectively, such that*

$$X = \left[\ldots\left[\left[S_{(p_1)}^{(k_1)}\right]_{(p_2)}^{(k_2)}\right]\ldots\right]_{(p_n)}^{(k_n)}$$

is called a **T-Code set** *at* **T-augmentation level**[1] *n. We write*

$$S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)} = X.$$

This definition leads to a few corollaries:

Corollary 2.2.2

Let $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ be a T-Code set at T-augmentation level n, and let $p_{n+1} \in S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ and $k_{n+1} \in \mathbb{N}^+$. The set

$$S_{(p_1,p_2,\ldots,p_{n+1})}^{(k_1,k_2,\ldots,k_{n+1})} = \left[S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}\right]_{(p_{n+1})}^{(k_{n+1})} \tag{2.2}$$

is a T-Code set at T-augmentation level $n+1$.

Proof: follows immediately from Definition 2.2.1. $\square$

Corollary 2.2.3

Let $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ be a T-Code set at T-augmentation level n, and let $p_{n+1} \in S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$

[1] referring to a T-augmentation level always carries an implicit reference to a particular set of T-prefixes and T-expansion parameters (see Chapter 3).

and $k_{n+1} \in \mathbb{N}^+$. *Then*

$$S^{(k_1,k_2,\ldots,k_{n+1})}_{(p_1,p_2,\ldots,p_{n+1})} = \left\{p^{k'_{n+1}}s | s \in S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} \backslash \{p_{n+1}\} \wedge 0 \leq k'_{n+1} \leq k_{n+1}\right\} \cup \{p^{k_{n+1}+1}_{n+1}\}. \tag{2.3}$$

Proof: follows immediately from Definitions 2.1.1 and 2.2.1. □

Another notion that will be used throughout this thesis is that of *intermediate* T-Code sets:

Definition 2.2.4 (Intermediate T-Code Sets)
Let $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ *be a T-Code set at T-augmentation level* n. *For* $m \leq n$, *the T-Code set* $S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)}$ *is called an* **intermediate T-Code set** *(of* $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$*).*

Note that the final set $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ counts as an intermediate T-Code set. This inclusion is of a technical nature and simplifies a number of proofs in this thesis.

Definition 2.2.5 (Simple T-Code Sets)
T-Code sets that can be generated entirely by simple T-augmentation are called "simple T-Code sets".

The following example illustrates the construction of a T-Code set:

Example 2.2.6 (Construction of a T-Code set)
Table 2.1 shows the construction and the intermediate T-Code sets for the binary T-Code set $S^{(1,1,3)}_{(0,1,01)}$ *at the T-augmentation levels from 0 to 3. The concept of T-augmentation may be understood as follows: a new column/T-Code set at level* $i+1$ *may be derived from at T-Code set at level* i *to the left by copying the codeword list into a new column a total of* $k_{i+1}+1$ *times. The copies may be indexed by* k'_{i+1} *such that* $k'_{i+1} = 0, \ldots, k_{i+1}$. *Each copy is then prefixed with* $p^{k'_{i+1}}_{i+1}$. *Finally, all codewords of the form* $p^{k'_{i+1}}_{i+1}$ *are removed from the list.*

	T-augmentation level			
n	0	1	2	3
k_n	$k_0 = 1$	1	1	3
set	S	$S_{(0)}^{(1)}$	$S_{(0,1)}^{(1,1)}$	$S_{(0,1,01)}^{(1,1,3)}$
	0	$\cancel{0}$	–	–
	1	1	$\cancel{1}$	–
		00	00	00
		01	01	$\cancel{0}\cancel{1}$
			–	–
			11	11
			100	100
			101	101
				–
				–
				0100
				$\cancel{0}\cancel{1}\cancel{0}\cancel{1}$
				–
				0111
				01100
				01101
				–
				–
				010100
				$\cancel{0}\cancel{1}\cancel{0}\cancel{1}\cancel{0}\cancel{1}$
				–
				010111
				0101100
				0101101
				–
				–
				01010100
				01010101
				–
				01010111
				010101100
				010101101

Table 2.1. The columns in the table list the codewords in the intermediate T-Code sets: S (T-augmentation level 0), $S_{(0)}^{(1)}$ (T-augmentation level 1), $S_{(0,1)}^{(1,1)}$ (T-augmentation level 2), and $S_{(0,1,01)}^{(1,1,3)}$ (the final set at T-augmentation level 3).

2.3 Basic Properties of T-Code Sets

This section explores and proves two basic properties of T-Code sets: prefix-freeness and completeness. Together, these two properties ensure the unique and instantaneous decodability of any semi-infinite string in $S^{+\infty}$. We also derive an expression for the cardinality of T-Code sets.

2.3.1 Prefix-Freeness

A code set C is called prefix-free if none of its codewords is the prefix of another, or, more formally:

$$\forall x \in C, y \in C: \; x \preceq_S y \Rightarrow x = y. \tag{2.4}$$

Theorem 2.3.1 (Prefix-Freeness of T-Code Sets)

T-Code sets are prefix-free.

Proof: by induction over the T-augmentation level n. The alphabet S is by definition prefix-free. By induction hypothesis, $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ is prefix-free. By Corollary 2.2.3, every codeword $x \in S_{(p_1,p_2,\ldots,p_{n+1})}^{(k_1,k_2,\ldots,k_{n+1})}$ is of the form $x = p_{n+1}^{k'_{n+1}} y$ where $y \in S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ and $0 \leq k'_{n+1} \leq k_{n+1}$. Presume that there is a codeword $x' \in S_{(p_1,p_2,\ldots,p_{n+1})}^{(k_1,k_2,\ldots,k_{n+1})}$ such that x' is a proper prefix of x. If we write x' in its appropriate form, i.e., $x' = p_{n+1}^{k''_{n+1}} y'$ where $y' \in S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$, then there are three possibilities:

1. $x = p_{n+1}^{k'_{n+1}} y$, $x' = p_{n+1}^{k''_{n+1}} y'$, and $k'_{n+1} = k''_{n+1}$. Thus $y' \prec_S y$, which violates the prefix-freeness of $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$.

2. $x = p_{n+1}^{k'_{n+1}} y$, $x' = p_{n+1}^{k''_{n+1}} y'$, and $k'_{n+1} > k''_{n+1}$. This implies that y' is a proper prefix of $p_{n+1}^{k'_{n+1}-k''_{n+1}} y$. By Corollary 2.2.3, $y' \neq p_{n+1}$ because $k''_{n+1} < k'_{n+1} \leq$

k_{n+1}. Hence, we require either $y' \prec_S p_{n+1}$ or $p_{n+1} \prec_S y'$. Neither of these two options is feasible because they violate the prefix-freeness of $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ and hence the induction hypothesis.

3. $x = p_{n+1}^{k'_{n+1}} y$, $x' = p_{n+1}^{k''_{n+1}} y'$, and $k'_{n+1} < k''_{n+1}$. This implies that $p_{n+1}^{k''_{n+1}-k'_{n+1}} y'$ is a proper prefix of y, which makes p_{n+1} a proper prefix of y. This is not compatible with the induction hypothesis either.

Hence, $S^{(k_1,k_2,\ldots,k_{n+1})}_{(p_1,p_2,\ldots,p_{n+1})}$ is prefix-free. □

As mentioned before, the property of prefix-freeness is so fundamental that some authors regard it as a necessary condition for a set of strings to be called a "code"[2].

2.3.2 Completeness

Code set completeness may be regarded as complementary to the prefix-freeness of a code. A prefix-free code C is called complete if it is not possible to add another word from S^+ to the code without violating its prefix-freeness, or, more formally:

$$\forall x \in S^+ \backslash C : \quad C \cup \{x\} \text{ is not prefix-free.} \tag{2.5}$$

A brief note on terminology: complete code sets are also sometimes called "exhaustive", in particular in an engineering context [13, 26, 38], however, this thesis will use the term "complete" since it seems to be the more common usage in the information sciences.

Completeness in conjunction with prefix-freeness of a code C implies unique and instantaneous decodability, i.e., any string in $S^{+\infty}$ can be written as a unique sequence of codewords from C.

Theorem 2.3.2 (Completeness of T-Code Sets)

T-Code sets are complete.

Proof: by induction over the T-augmentation level n. The alphabet S is by default complete — any string from S^+ that we could add starts with a symbol from S and hence the resulting set would not be prefix-free. Our induction hypothesis is now that $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ is complete. Let us presume that there exists a finite string $x \in S^+ \backslash S_{(p_1,p_2,\ldots,p_{n+1})}^{(k_1,k_2,\ldots,k_{n+1})}$ such that $S_{(p_1,p_2,\ldots,p_{n+1})}^{(k_1,k_2,\ldots,k_{n+1})} \cup \{x\}$ is still prefix-free. Unique and instantaneous decodability of $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ implies that we have three cases

1. x is a proper prefix of a codeword in $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$. In this case, x violates the prefix-freeness of $S_{(p_1,p_2,\ldots,p_{n+1})}^{(k_1,k_2,\ldots,k_{n+1})}$ because all proper prefixes of codewords in $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ are also proper prefixes of codewords in $S_{(p_1,p_2,\ldots,p_{n+1})}^{(k_1,k_2,\ldots,k_{n+1})}$.

2. x has a proper prefix that is a codeword in $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)} \backslash \{p_{n+1}\}$. Since all codewords of $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ except p_{n+1} are also codewords in $S_{(p_1,p_2,\ldots,p_{n+1})}^{(k_1,k_2,\ldots,k_{n+1})}$, this would violate the prefix-freeness of $S_{(p_1,p_2,\ldots,p_{n+1})}^{(k_1,k_2,\ldots,k_{n+1})}$.

3. p_{n+1} is a proper prefix of x. Now we may remove copies of p_{n+1} from the left of x until

   - we have removed $k_{n+1}+1$ copies of p_{n+1}, in which case $p_{n+1}^{k_{n+1}+1}$ is a proper prefix of x, or
   - the remainder x' after the removal of $k'_{n+1} \leq k_{n+1}$ copies of p_{n+1} no longer has a decoding over $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$, in which case x' is a proper prefix of a codeword $s \in S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$. Hence, x is a proper prefix of $p_{n+1}^{k'_{n+1}} s$, or
   - the remainder x' after the removal of $k'_{n+1} \leq k_{n+1}$ copies of p_{n+1} no longer starts with p_{n+1}, but with some other codeword $s \in S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$. Then $p_{n+1}^{k'_{n+1}} s$ is a proper prefix of x.

In all these cases, the prefix-freeness of T-Code sets leads to the desired contradiction. Hence, T-Code sets are complete. $\square$

2.3.3 Cardinality

The cardinality of a T-Code set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ is determined by the number of alphabet symbols, $\#S$, and the T-expansion parameters used in the construction of the T-Code set [49, 17]:

Theorem 2.3.3

The cardinality of T-Code sets at T-augmentation level 1 and above is given by

$$\#S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)} = 1 + (\#S - 1)\prod_{i=1}^{n}(k_i + 1). \tag{2.6}$$

Proof: by induction over the T-augmentation level n. By Corollary 2.2.3, the assertion holds for $n = 1$. In a T-augmentation from $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ to $S_{(p_1,p_2,\ldots,p_{n+1})}^{(k_1,k_2,\ldots,k_{n+1})}$, we need to account for the following cases:

- the codeword $p_{n+1}^{k_{n+1}+1}$,
- $\#S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)} - 1$ codewords from $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$, each prefixed with between 0 and k_{n+1} copies of p_{n+1}. This is a total of $\left(\#S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)} - 1\right)(k_{n+1} + 1)$ codewords[2].

This yields a total of

$$\begin{aligned}
\#S_{(p_1,p_2,\ldots,p_{n+1})}^{(k_1,k_2,\ldots,k_{n+1})} &= 1 + \left(\#S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)} - 1\right)(k_{n+1} + 1)) \\
&= 1 + \left(1 + (\#S - 1)\prod_{i=1}^{n}(k_i + 1) - 1\right)(k_{n+1} + 1)) \\
&= 1 + (\#S - 1)\prod_{i=1}^{n+1}(k_i + 1).
\end{aligned} \tag{2.7}$$

□

[2] The unique decodability of strings composed from T-Code codewords ensures that we do not count codewords in $S_{(p_1,p_2,\ldots,p_{n+1})}^{(k_1,k_2,\ldots,k_{n+1})}$ twice. For a more elaborate and wide-ranging proof of this assertion that does not rely on the cardinality of T-Code sets, see Theorem 4.2.3.

The minimum cardinality at a particular T-augmentation level is given by the simple T-Code sets, where $k_i = 1$ for $1 \leq i \leq n$, and thus [46]

$$\#S^{1,1,\ldots,1}_{p_1,p_2,\ldots,p_n} = 1 + (\#S - 1)2^n. \tag{2.8}$$

In the case of a binary coding alphabet ($\#S = 2$), this simplifies to

$$\#S^{1,1,\ldots,1}_{p_1,p_2,\ldots,p_n} = 1 + 2^n. \tag{2.9}$$

2.4 Notation Conversion

In 1985, Titchener devised a notation that permitted the unique description of any simple T-Code set [45], based on the binary depletion codes (an advanced form of which will be discussed in Chapter 4). It has subsequently been used among others by Titchener, Higgie [26, 27], as well as Honary, Zolghadr and Darnell [7]. The notation also states the alphabet, followed by square brackets with a list of T-prefixes. These T-prefixes are not given by their literal reading but rather by an index corresponding to their position on a list similar to the lists in the columns of Table 2.1. The codeword in the top line of the list is given the index 0, and counting all lines, including the empty lines (due to previously removed T-prefixes), one finally arrives at the appropriate index.

If we restrict ourselves to simple T-Codes, this notation is indeed appropriate. However, it needs to be adapted to work with generalised T-Codes where the result of a T-augmentation no longer depends on the T-prefixes alone. Depending on the T-expansion parameter used, positions on a codeword list may be occupied by different codewords. Such an adaptation was performed for a suite of C tools (TCODE — T-Code Online Development Environment [52]), but proved to be too cumbersome for theoretical work.

Hence, a more comprehensive and improved notation was required for generalised T-Codes. It is used throughout this thesis and has already been introduced in this chapter. The following example shows how to convert between the two notations:

Example 2.4.1 (Notation Conversion)

1. *The simple T-Code set* $S^{(1,1,1)}_{(1,10,0)}$ *from Table 2.2 is denoted* $S[1,2,0]$ *in Titchener's 1985 notation. Note that it is not simply possible to take the "binary value" of the strings as this might suggest. For example,* $S^{(1,1,1)}_{(0,00,1)}$ *would be denoted* $S[0,2,1]$.

2. *In the other direction, we convert as follows: consider, e.g.,* $S[1,3,2]$. *With* $p_1 = 1$, *we get* $S[1] = S^{(1)}_{(1)} = \{0, -, 10, 11\}$, *where the "-" indicates a "deleted" position that needs to be counted. Thus,* $p_2 = 11$ *and*

$$S[1,3] = S^{(1,1)}_{(1,11)} = \{0, -, 10, -, 110, -, 1110, 1111\}.$$

Finally, we get $p_3 = 10$ *and hence*

$$\begin{aligned} S[1,3,2] &= S^{(1,1,1)}_{(1,11,10)} \\ &= \{0, -, -, -, 110, -, 1110, 1111, 100, -, \\ &\quad 1010, -, 10110, -, 101110, 101111\} \end{aligned}$$

2.5 Discussion

This chapter has shown how T-Code sets may be constructed, and that they possess the basic properties of prefix-freeness and completeness that ensure unique and

	T-augmentation level			
n	0	1	2	3
k_n	n/a	1	1	1
index	S	$S_{(1)}^{(1)}$	$S_{(1,10)}^{(1,1)}$	$S_{(1,10,0)}^{(1,1,1)}$
0	0	0	0	$\cancel{0}$
1	1	$\cancel{1}$	–	–
2		10	$\cancel{1}\cancel{0}$	–
3		11	11	11
4			100	100
5			–	–
6			1010	1010
7			1011	1011
8				00
9				–
10				–
11				011
12				0100
13				–
14				01010
15				01011

Table 2.2. Indices of T-Code codewords in a series of simple T-Code sets.

instantaneous decodability. These features are desirable for codes in practical applications, for example in communications or data compression. Furthermore, the cardinality of T-Code sets has been shown to depend only on the alphabet size and the T-expansion parameters.

Having discussed the "basics" of T-Codes, we now turn our attention towards the recursive features of T-Codes. The reader may have noticed that all the significant proofs in this chapter have been conducted by induction over the T-augmentation level n — a method that seems to be tailor-made for T-Code sets with their recursive construction by T-augmentation. It is therefore not surprising that proofs by induction will be used repeatedly throughout this thesis.

CHAPTER 3

T-Prescriptions

The process of T-augmentation in the construction of T-Code sets may be regarded as a recursive "copy-and-prefix" process. This chapter illustrates this construction using decoding trees as a graphical representation. A question of uniqueness arises: is it possible that multiple sets of T-prefixes and T-expansion parameters yield identical sets? The answer is yes, under certain well-defined circumstances [36]. The notion of a T-prescription is introduced, and equivalence criteria for T-prescriptions are discussed.

3.1 Variable-Length Codes as Trees

Decoding trees and search trees are common in computer science. Consider a prefix-free variable-length code (e.g., a Huffman code or a T-Code) based on an alphabet S with $\#S$ symbols. The decoding tree starts at a **root node**, which we denote by λ (the string associated with the beginning of the decoding is the empty string). Unless the code set is empty, the root node is a **branch node**. A branch node gives rise to between 1 and $\#S$ *outgoing* **branches** connecting the node with other nodes, and (except for the root node) exactly one *incoming* branch. Each outgoing branch from a branch node is associated with a unique symbol from S. Outgoing branches end either at other branch nodes, or at **leaf nodes** (terminal nodes), characterised by having no outgoing branches. Leaf nodes correspond to a completely decoded codeword.

A decoder for the code may be thought of as a state machine automaton, starting at the root node. Upon receipt of the first symbol from S, the automaton follows the branch associated with that symbol to the next node. If this node is a branch node, the decoder will wait until the next symbol is received. It then exits the branch node via the appropriate branch to the next node. This is repeated until the node reached is a leaf node, at which stage the decoder outputs the source symbol corresponding to the decoded codeword.

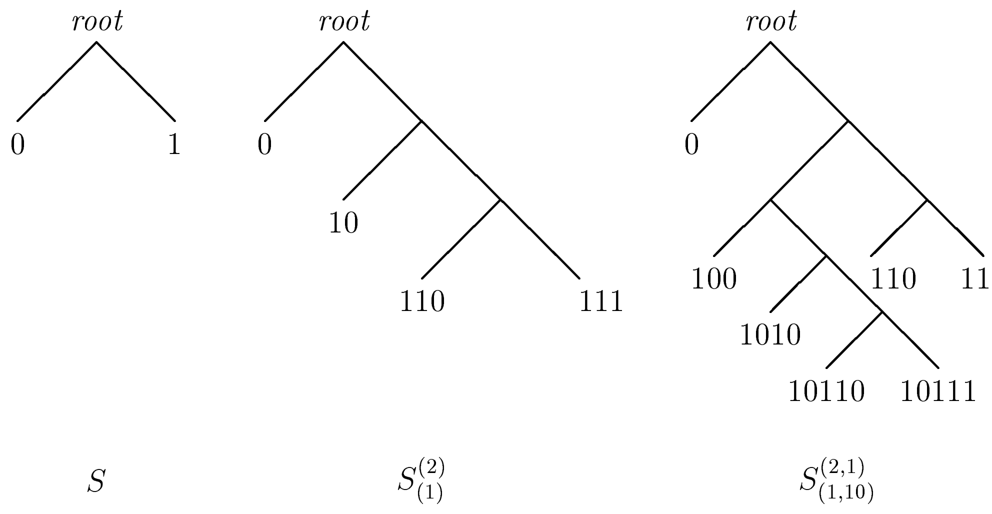
Commonly, the trees are drawn such that they indicate the "direction" of the decoding, e.g., the decoder moves from the root node at the top to the leaf nodes further "down" in the tree. Here, we will adopt the convention that (for binary trees) a branch to the left will correspond to the receipt of a 0, while a branch to the right will correspond to the receipt of a 1.

3.2 T-Codes as Trees

For T-Codes, trees help to understand T-augmentation as a process of appending k copies of an existing tree to the original copy via one of its leaf nodes, p. The following example illustrates this:

Example 3.2.1 (T-Augmentation of Trees)

Consider the T-augmentation sequence of $S = \{0, 1\}$ *via* $S_{(1)}^{(2)}$ *to* $S_{(1,10)}^{(2,1)}$ *in Figure 3.1.*



.

Figure 3.1. Recursive growth of a decoding tree under two T-augmentations.

In the first T-augmentation, the tree for S is copied twice and the two copies are appended to the original tree (and each other) via the respective leaf nodes corresponding to the codeword 1*. The second T-augmentation links one copy of the tree for $S^{(2)}_{(1)}$ with the original via the leaf node* 10*.*

We can now see the significance of the T-prefixes and the T-expansion parameters: the choice of the T-prefix controls the "direction" in which the tree "grows" as a result of the T-augmentation. The T-expansion parameter controls the size (i.e., the number of nodes) of the T-augmented tree. Together, the T-prefix and the T-expansion parameter control the depth and codeword length distribution of the tree.

3.3 T-Prescriptions: Construction Strategies for T-Code Sets

The previous section discussed how the choice of T-prefixes and T-expansion parameters affects the size and shape of a T-Code decoding tree. This raises another question: is it possible to construct the same T-Code set using two different sets of T-prefixes and T-expansion parameters? The answer is yes — the reader may wish to verify that, e.g., the T-Code sets $S_{(0)}^{(3)}$ and $S_{(0,00)}^{(1,1)}$ are identical.

Nicolescu [36] showed that this is indeed the case under certain well-defined circumstances. To separate a T-Code set from a particular way of constructing it, Nicolescu introduced the notion of a "T-prescription":

Definition 3.3.1 (T-Prescriptions)

For a given T-Code set C, the system $(S,(p_1,p_2,\ldots,p_n),(k_1,k_2,\ldots,k_n))$ is called a **T-prescription** *of C if $C = S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$.*

Note that this definition permits several T-prescriptions for the same set. Nicolescu further proved that if one prescription was known, all other prescriptions could be derived from it. This is possible by searching for "factors" in T-prefixes and T-expansion parameters:

Theorem 3.3.2 (Expansion)

Consider a T-prescription $(S,(p_1,p_2,\ldots,p_n),(k_1,k_2,\ldots,k_n))$ of $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$. If for some k_m with $1 \leq m \leq n$, (k_m+1) is not prime, i.e., there exist two positive integers $k_m^{\dagger}$ and $k_m^{\dagger\dagger}$ such that

$$k_m + 1 = (k_m^{\dagger}+1)(k_m^{\dagger\dagger}+1), \tag{3.1}$$

then the T-prescription

$$(S,(p_1,p_2,\ldots,p_{m-1},p_m,p_m^{k_m^{\dagger}+1},p_{m+1},\ldots,p_n),(k_1,k_2,\ldots,k_{m-1},k_m^{\dagger},k_m^{\dagger\dagger},k_{m+1}\ldots,k_n))$$

is also a T-prescription of $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$.

Proof: it suffices to show that two successive T-augmentations of a T-Code set $S^{(k_1,k_2,\ldots,k_{m-1})}_{(p_1,p_2,\ldots,p_{m-1})}$, with $k^{\dagger}_m$, $k^{\dagger\dagger}_m$ as the successive T-expansion parameters and p_m, $p_m^{k^{\dagger}_m+1}$ as the T-prefixes, yield the same final set as a single T-augmentation of $S^{(k_1,k_2,\ldots,k_{m-1})}_{(p_1,p_2,\ldots,p_{m-1})}$ with k_m and p_m respectively:

$$
\begin{aligned}
&S^{(k_1,k_2,\ldots,k_{m-1},k^{\dagger}_m,k^{\dagger\dagger}_m)}_{(p_1,p_2,\ldots,p_{m-1},p_m,p_m^{k^{\dagger}_m+1})} \\
&= \left\{x|x = p_m^{(k^{\dagger}_m+1)i}s \text{ where } s \in S^{(k_1,k_2,\ldots,k_{m-1},k^{\dagger}_m)}_{(p_1,p_2,\ldots,p_{m-1},p_m)} \backslash \left\{p_m^{k^{\dagger}_m+1}\right\} \wedge 0 \leq i \leq k^{\dagger\dagger}_m\right\} \\
&\quad \cup \left\{p_m^{(k^{\dagger}_m+1)(k^{\dagger\dagger}_m+1)}\right\} \\
&= \left\{x|x = p_m^{(k^{\dagger}_m+1)i}s \text{ where } s \in \left\{y|y = p_m^{j}s' \wedge s' \in S^{(k_1,k_2,\ldots,k_{m-1})}_{(p_1,p_2,\ldots,p_{m-1})} \backslash \{p_m\}\right.\right. \\
&\quad \left.\left. \wedge\, 0 \leq j \leq k^{\dagger}_m\right\} \wedge\, 0 \leq i \leq k^{\dagger\dagger}_m\right\} \cup \left\{p_m^{(k^{\dagger}_m+1)(k^{\dagger\dagger}_m+1)}\right\} \\
&= \left\{x|x = p_m^{(k^{\dagger}_m+1)i}p_m^{j}s' \text{ where } s' \in S^{(k_1,k_2,\ldots,k_{m-1})}_{(p_1,p_2,\ldots,p_{m-1})} \backslash \{p_m\} \wedge 0 \leq j \leq k^{\dagger}_m \wedge 0 \leq i \leq k^{\dagger\dagger}_m\right\} \\
&\quad \cup \left\{p_m^{k_m+1}\right\} \\
&= \left\{x|x = p_m^{i'}s' \text{ where } s' \in S^{(k_1,k_2,\ldots,k_{m-1})}_{(p_1,p_2,\ldots,p_{m-1})} \backslash \{p_m\} \wedge 0 \leq i' \leq (k^{+}_m+1)k^{\dagger\dagger}_m + k^{+}_m\right\} \\
&\quad \cup \left\{p_m^{k_m+1}\right\} \\
&= \left\{x|x = p_m^{i'}s' \text{ where } s' \in S^{(k_1,k_2,\ldots,k_{m-1})}_{(p_1,p_2,\ldots,p_{m-1})} \backslash \{p_m\} \wedge 0 \leq i' \leq k_m\right\} \cup \left\{p_m^{k_m+1}\right\} \\
&= S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)} \qquad (3.2)
\end{aligned}
$$

This concludes the proof of Theorem 3.3.2. □

The proof above may also be used to prove the "reverse" theorem:

Theorem 3.3.3 (Contraction)
Consider a T-prescription $(S,(p_1,p_2,\ldots,p_n),(k_1,k_2,\ldots,k_n))$ *of* $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$. *If for some* p_m, p_{m+1} *with* $1 \le m < n$,

$$p_{m+1} = p_m^{k_m+1}, \tag{3.3}$$

then the T-prescription

$$(S,(p_1,p_2,\ldots,p_{m-1},p_m,p_{m+2},\ldots,p_n),(k_1,k_2,\ldots,k_{m-1},(k_m+1)(k_{m+1}+1)-1,k_{m+2},\ldots,k_n))$$

is also a T-prescription of $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$.

Proof: see above. □

The above theorems confirm Nicolescu's results from [36]. They will also be used in Chapter 7, where his uniqueness theorem is discussed against the background of hierarchical coding. In Chapter 10, the above results are applied to achieve a considerable reduction in computational complexity for a search algorithm.

Whenever there are several T-prescriptions for a set, we can find one that is "canonical" and another one that is "anti-canonical":

Definition 3.3.4 (Canonical and Anti-Canonical T-Prescriptions)
A T-prescription $(S,(p_1,p_2,\ldots,p_n),(k_1,k_2,\ldots,k_n))$ *of* $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ *is called* **canonical** *if there is no* m *with* $1 \le m < n$ *for which* $p_{m+1} = p_m^{k_m+1}$. *It is called* **anti-canonical** *if there is no* m *with* $1 \le m \le n$ *for which* (k_m+1) *is not prime.*

Nicolescu also showed that the canonical and the anti-canonical T-prescriptions of $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ are unique, i.e., there is only one canonical and one anti-canonical T-prescription for each T-Code set. Note that the canonical and the anti-canonical T-prescription may be identical, in which case no other T-prescription of $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ exists.

3.4 Discussion

The structure of T-Code sets as the product of a recursive "copy-and-prefix" process is now well understood. The existence of multiple T-prescriptions for some T-Code sets introduces an element of ambiguity. However, by demanding that only the canonical or anti-canonical form of a T-prescription be used, one can avoid this ambiguity. This is utilised in a set searching algorithm presented in Chapter 10, where it helps to avoid repeated processing of the same T-Code set under different T-prescriptions.

However, the existence of multiple T-prescriptions is also a useful tool when testing assertions about T-Code sets: all assertions that relate to the set must be independent of the T-prescription chosen to obtain it.

CHAPTER 4

T-Depletion Codes

In some practical applications, e.g., in an encoder or decoder, it is desirable to have a fixed-length format for the storaged and handling of codewords from a variable-length code. This chapter shows that the recursive construction of T-Code sets gives rise to an inherent recursive structure in T-Code codewords. This structure may be exploited to efficiently represent T-Code codewords in a fixed-length format, the T-depletion codeword. Algorithms that convert between the variable-length T-Code codewords and their corresponding T-depletion codewords may be regarded as encoders and decoders, and an example for each is presented here.

4.1 Representing Variable-Length Codes in a Fixed-Length Format

By their nature, variable-length codes, such as Huffman codes, are somewhat cumbersome to handle in computers with a fixed-length word architecture. For this reason, we require fixed-length representations for variable-length codes, and conversions between variable-length code and fixed-length code formats.

For the purposes of this chapter, an encoder is regarded as an algorithm or device that converts source data from a fixed-length input format into a unique variable-length output format. Similarly, a decoder is regarded as an algorithm or device that performs the inverse operation, i.e., that converts variable-length input data uniquely into some fixed-length output (which may then be used to, e.g, address (index) a look-up table containing the corresponding source symbols). In particular, we consider the case where a sequence of fixed-length inputs is encoded into a series of concatenated variable-length outputs, yielding a continuous symbol stream. This symbol stream is then communicated through a channel, which may introduce symbol errors into the stream. At the other end of the channel, the stream is decoded and reconverted into a sequence of fixed-length outputs.

One way of storing variable-length codewords in a fixed-length format is to store both the codeword string and its length, where the length of the field used to store the codeword string is determined by the length of the longest codeword in the set. In Section 4.2.4, an example illustrates that this representation is not very space-efficient compared to the representation proposed in this chapter. The "traditional" way of representing strings in the PASCAL programming language (and the problems this created) is a good example for this approach.

The representation of the variable-length codewords is a particular problem in encoders. In decoders, such as the universal decoders for variable-length codes proposed by Tanaka [42] or Chung [6], an enumeration of the codewords can bypass the need to handle codewords in their literal form. However, if the code set used is not fixed, it may need to be communicated to the decoder. In this case, a general variable-length code may still require a format similar to the one above to communicate the codewords and their "translations". Canonical Huffman codes as treated by Hirschberg and Lelewer [29] simplify the decoder considerably by establishing a convention that permits the derivation of the decoding tree from the code's code length distribution. However, (canonical) Huffman codes are not necessarily robust, in particular with respect to synchronisation.

Apart from addressing this practical problem, the fixed-length T-depletion representation for T-Code codewords introduced in this chapter also provides an insight into the structure of T-Code codewords. It may also be used to derive synchronisation information in a T-Code decoder (cf. Chapter 8). Furthermore, as we will see in Chapter 6, the T-depletion representation is general enough to accommodate arbitrary variable-length codes.

The predecessor to the T-depletion codes, the binary depletion codes, were initially proposed by Titchener [45], based on simple T-Codes. The term "depletion" originates from a depletion algorithm that is applied to an initially complete list of binary words of $n+1$ bits (in the case of a binary alphabet S). Starting from the initial list of 2^{n+1} binary words, individual words are deleted from the list according to a regular pattern. Titchener showed that the remaining words on the thus depleted list corresponded uniquely to T-Code codewords in a simple T-Code set at T-augmentation level n, with individual bits indicating the absence or presence of T-prefixes in the codeword concerned. He thus showed that the depletion algorithm in the fixed-length domain was equivalent to the T-augmentation algorithm in the variable-length domain.

Zolghadr, Honary, and Darnell used binary depletion codes in the implementation of a real-time channel evaluation system [7]. It is now convenient to extend the notion of these depletion codes to accommodate generalised T-augmentation. The following sections thus develop a general formal definition for T-depletion codes that permits the representation of generalised T-Codes based on binary and non-binary alphabets.

The next section shows how the T-depletion code format may be derived from the general form of T-Code codewords.

4.2 The Structure of T-Code Codewords and T-Depletion Codes

4.2.1 The Structure of T-Code Codewords

We begin by proving two lemmata that will be used in the proof of the central theorem of this section, and in the proof of Theorem 6.1.2.

Lemma 4.2.1

Consider a string of the form

$$x = p_n^{k'_n} p_{n-1}^{k'_{n-1}} \dots p_1^{k'_1}. \tag{4.1}$$

where $0 \leq k'_i \leq k_i$ *for* $i = 1, \dots, n$. *Then* x *is a proper prefix of a codeword in the set* $S_{(p_1,p_2,\dots,p_n)}^{(k_1,k_2,\dots,k_n)}$, *i.e., there exists a string* $y \in S^+$ *such that* $xy \in S_{(p_1,p_2,\dots,p_n)}^{(k_1,k_2,\dots,k_n)}$.

Proof: by induction over n. For $n = 0$, $x = \lambda$ (where λ denotes the empty string), which is clearly a proper prefix of all elements of S. Now consider

$$x = p_{n+1}^{k'_{n+1}} p_n^{k'_n} p_{n-1}^{k'_{n-1}} \dots p_1^{k'_1}. \tag{4.2}$$

By induction hypothesis, the word $p_n^{k'_n} p_{n-1}^{k'_{n-1}} \dots p_1^{k'_1}$ is a proper prefix of a codeword $z \in S_{(p_1,p_2,\dots,p_n)}^{(k_1,k_2,\dots,k_n)}$. We now distinguish two cases: if $z \neq p_{n+1}$, then $p_{n+1}^{k'_{n+1}} z$ is a codeword in $S_{(p_1,p_2,\dots,p_{n+1})}^{(k_1,k_2,\dots,k_{n+1})}$ and hence x is a proper prefix of it. If $z = p_{n+1}$, then x is a proper prefix of $p_{n+1}^{k_{n+1}+1} \in S_{(p_1,p_2,\dots,p_{n+1})}^{(k_1,k_2,\dots,k_{n+1})}$. □

We can now use this result to prove the following lemma:

Lemma 4.2.2

For any string $x \in S^\star$ and a given T-Code set $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$, there exists at most one set $\{k'_1,\ldots,k'_n\}$ with $0 \leq k'_i \leq k_i$ for $i = 1,\ldots,n$ such that x may be written in the form of Equation (4.1).

Proof: x for which no appropriate sets $\{k'_1,\ldots,k'_n\}$ exist trivially satisfy the lemma.

Strings x for which an appropriate set $\{k'_1,\ldots,k'_n\}$ exists are obviously more interesting. In this case, we may use induction over n to prove that $\{k'_1,\ldots,k'_n\}$ is indeed the only set that satisfies Equation (4.1).

For $n = 0$, only the empty word λ can be written in the form of Equation (4.1), and this representation is unique.

For the induction step we assume that a word $x \in S^\star$ can be written in two forms satisfying Equation (4.1):

$$x = p_{n+1}^{k'_{n+1}} p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1}. \tag{4.3}$$

and

$$x = p_{n+1}^{k''_{n+1}} p_n^{k''_n} p_{n-1}^{k''_{n-1}} \ldots p_1^{k''_1}, \tag{4.4}$$

such that $0 \leq k'_i \leq k_i$, $0 \leq k''_i \leq k_i$ for $i = 1,\ldots,n+1$. We need to show that necessarily $k'_i = k''_i$ for all $i = 1,\ldots,n+1$.

We distinguish three cases:

1. $k'_{n+1} = k''_{n+1}$: in this case, $p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1} = p_n^{k''_n} p_{n-1}^{k''_{n-1}} \ldots p_1^{k''_1}$ and the induction hypothesis applies such that $k'_i = k''_i$ for $i = 0,\ldots,n$.

2. $k'_{n+1} > k''_{n+1}$: in this case, p_{n+1} is a prefix of $p_n^{k''_n} p_{n-1}^{k''_{n-1}} \ldots p_1^{k''_1}$. However, by Lemma 4.2.1, this is a prefix of a codeword in $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$. Thus, p_{n+1} would have to be a prefix of a codeword in $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$, which violates the prefix-freeness of that set.

3. $k'_{n+1} < k''_{n+1}$: this case also violates the prefix-freeness of $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$, by the same argument as before.

This proves Lemma 4.2.2. $\square$

We are now ready to prove the central theorem of this section: the existence of a unique form for T-Code codewords in $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$:

Theorem 4.2.3 (Decomposition of T-Code Codewords)
For all codewords $x \in S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$, a decomposition

$$x = p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1} k'_0, \tag{4.5}$$

exists such that $0 \leq k'_i \leq k_i$ for $i = 0, 1, \ldots, n$ and $k_0 = \#S-1$. This decomposition of x is unique, i.e., there exists exactly one set of k'_i for which Equation (4.5) is satisfied.

Proof: **Existence**: by induction over n. For $n = 0$, the T-Code set is the alphabet S itself. As we may represent each alphabet symbol by an integer $0 \leq k'_0 \leq k_0$, codewords x_0 at T-augmentation level 0 are of the general form $x = k'_0$, which satisfies the theorem.

By the induction hypothesis, the codewords of $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ satisfy the general form

$$x = p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1} k'_0, \tag{4.6}$$

for some k'_i, $i = 0, 1, \ldots, n$, such that $0 \leq k'_i \leq k_i$. The T-augmentation of $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ with p_{n+1} and k_{n+1} prefixes these codewords with $k'_{n+1} \leq k_{n+1}$ copies of p_{n+1}. The codewords in $S^{(k_1,k_2,\ldots,k_n,k_{n+1})}_{(p_1,p_2,\ldots,p_n,p_{n+1})}$ thus satisfy the general form

$$x = p_{n+1}^{k'_{n+1}} p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1} k'_0. \tag{4.7}$$

Thus, we have shown the existence of the decomposition.

Uniqueness: write a codeword $x \in S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ in the form of Equation 4.5:

$$x = p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1} k'_0. \tag{4.8}$$

The uniqueness of k'_0 is clear as it is simply the last symbol in x. The uniqueness of $k'_1, \ldots, k'_n$ follows from Lemma 4.2.2 by applying it to the string $p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1}$. This concludes the proof of Theorem 4.2.3. □

In fact, the representation in form of Equation (4.5) is unique not only for codewords in $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$, but also for all codewords in its intermediate T-Code sets:

Theorem 4.2.4 (Decomposition for Intermediate T-Code Sets)
For any $m \leq n$, the decomposition of $x \in S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)}$ as

$$x = p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1} k'_0 \tag{4.9}$$

with $0 \leq k'_i \leq k'_i$ exists and is unique. It satisfies $k'_i = 0$ for $i > m$.

Proof: The existence of a decomposition with $k'_i = 0$ for $i > m$ follows from Theorem 4.2.3 for $k'_i = 0$ for $i > m$. Again, the uniqueness of k'_0 is clear. The uniqueness of $k'_1, \ldots, k'_n$ follows from Lemma 4.2.2. □

Definition 4.2.5 (T-expansion indices and literal symbol)
Let $x = p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1} k'_0$ with $x \in S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)}$ for $m \leq n$ and $k'_i \leq k_i$ for $i = 0, \ldots, n$. For $i \geq 1$, we call k'_i the i'th **T-expansion index** *of x. k'_0 is called the* **literal symbol**.

Example 4.2.6 (Decomposition of T-Code codewords)
Table 4.1 shows the decomposition of the codewords from the binary ($S = \{0, 1\}$) T-Code set $S^{(1,1,3)}_{(0,1,01)}$.

T-Code	*structure*
00	$(01)^0 1^0 0^1 0$
11	$(01)^0 1^1 0^0 1$
100	$(01)^0 1^1 0^1 0$
101	$(01)^0 1^1 0^1 1$
0100	$(01)^1 1^0 0^1 0$
0111	$(01)^1 1^1 0^0 1$
01100	$(01)^1 1^1 0^1 0$
01101	$(01)^1 1^1 0^1 1$
010100	$(01)^2 1^0 0^1 0$
010111	$(01)^2 1^1 0^0 1$
0101100	$(01)^2 1^1 0^1 0$
0101101	$(01)^2 1^1 0^1 1$
01010100	$(01)^3 1^0 0^1 0$
01010101	$(01)^3 1^0 0^1 1$
01010111	$(01)^3 1^1 0^0 1$
010101100	$(01)^3 1^1 0^1 0$
010101101	$(01)^3 1^1 0^1 1$

Table 4.1. Codeword decomposition for the T-Code set $S_{(0,1,01)}^{(1,1,3)}$.

Nicolescu [36] showed that the decomposition of one of the longest codewords in a T-Code set may be used to represent the set itself. The longest codewords contain all T-prefixes, and their T-expansion indices equal the T-expansion factors of the corresponding set. Chapter 7 discusses T-decomposition, an algorithm that permits the recovery of a T-prescription[1] from any one of the longest codewords. Hence, it is possible to communicate the whole T-Code set to a decoder by simply sending one of the longest codewords.

[1] The T-prefixes and T-expansion parameters cannot necessarily be uniquely determined this way. Consider, e.g., $S_{(0)}^{(3)} = S_{(0,00)}^{(1,1)}$ for $S = \{0, 1\}$.

4.2.2 T-Depletion Codes

Presume that — for a given T-Code set — the T-prefixes and T-expansion parameters are known. Thus, we may specify any codeword in such a set simply by stating the corresponding T-expansion indices $k'_1, k'_2, \ldots, k'_n$ and literal symbol k'_0. For example, if the T-Code set $S^{(1,1,3)}_{(0,1,01)}$ from the previous examples was given, and we specified that $k'_0 = 1$, $k'_1 = 0$, $k'_2 = 1$, and $k'_3 = 2$, it would be unambiguous that this combination would refer to $(01)^2 1^1 0^0 1 = 010111$. T-depletion codewords, which may be defined in terms of multibase numbers, implement this format:

Definition 4.2.7 (Multibase Numbers)
A vector $(k'_n, k'_{n-1}, \ldots, k'_1, k'_0)$ is called a **multibase number** *with base $(k_n + 1, k_{n-1} + 1, \ldots, k_1 + 1, k_0 + 1)$ if for all i, $0 \leq i \leq n$*

$$0 \leq k'_i \leq k_i. \tag{4.10}$$

Definition 4.2.8 (T-Depletion Codewords)
For a given T-Code set $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$, a multibase number $(k'_n, k'_{n-1}, \ldots, k'_1, k'_0)$ is called the **T-depletion codeword** *$d_n(x)$ corresponding to $x \in S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)}$ if $m \leq n$, $0 \leq k'_i \leq k_i$ for $i = 0, \ldots, n$, and*

$$x = p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1} k'_0. \tag{4.11}$$

The notion of multibase numbers will probably be familiar to the reader: for example, a 24-hour clock may be represented by a multibase number with base $(24, 60, 60)$, i.e., one index running from 0 to 23 for the hours, and two indices running from 0 to 59 each for the minutes and seconds.

T-Code	*T-depletion codeword*
	(k'_3, k'_2, k'_1, k'_0)
00	$(0,0,1,0)$
11	$(0,1,0,1)$
100	$(0,1,1,0)$
101	$(0,1,1,1)$
0100	$(1,0,1,0)$
0111	$(1,1,0,1)$
01100	$(1,1,1,0)$
01101	$(1,1,1,1)$
010100	$(2,0,1,0)$
010111	$(2,1,0,1)$
0101100	$(2,1,1,0)$
0101101	$(2,1,1,1)$
01010100	$(3,0,1,0)$
01010101	$(3,0,1,1)$
01010111	$(3,1,0,1)$
010101100	$(3,1,1,0)$
010101101	$(3,1,1,1)$

Table 4.2. Codewords and T-depletion code elements for the T-Code set $S^{(1,1,3)}_{(0,1,01)}$.

Note that, by Theorem 4.2.4, a unique **T-depletion codeword** exists for each codeword from the intermediate T-Code sets $S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)}$.

T-depletion codewords are illustrated in the following example:

Example 4.2.9 (T-Depletion Codewords)
The codewords from the binary ($S = \{0,1\}$) T-Code set $S^{(1,1,3)}_{(0,1,01)}$ and their associated T-depletion codewords are listed in Table 4.2.

Since the decomposition of T-Code codewords reflects the recursive construction of the T-Codes, the T-depletion codewords reflect it, too: consider, for example, a T-depletion codeword $d_n(x) = (k'_n, k'_{n-1}, \ldots, k'_1, k'_0)$ of a T-Code codeword $x \in S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$, i.e., $x = p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1} k'_0$. We may split x recursively into k'_n copies of the n'th level T-prefix p_n, and the codeword $p_{n-1}^{k'_{n-1}} \dots p_1^{k'_1} k'_0$, all of which are codewords in $S^{(k_1,k_2,\dots,k_{n-1})}_{(p_1,p_2,\dots,p_{n-1})}$, etc. Similarly, the T-depletion codeword $d_n(x)$ may be interpreted as accounting for k'_n copies of $d_{n-1}(p_n)$, and one copy of $d_{n-1}(p_{n-1}^{k'_{n-1}} \dots p_1^{k'_1} k'_0)$.

This may be utilised in the construction of encoders and decoders, which may be regarded as converters between T-Code and T-depletion codewords, i.e., between the variable-length format and the fixed-length format.

4.2.3 Conversion between Variable-Length T-Code Codewords and T-Depletion Codewords

Table 4.2 lists the T-Code codewords from $S^{(1,1,3)}_{(0,1,01)}$ and their corresponding T-depletion codewords, but does not explain how to obtain one format from the other. We will now discuss these conversions:

Encoders: Converting T-Depletion into T-Code Codewords

The conversion algorithm here is in principle the same algorithm used for the binary depletion codes: Theorem 4.2.3 provides us with an easy way of converting T-depletion codewords into their variable-length T-Code codewords counterparts in $S^{(k_1,k_2,\dots,k_n)}_{(p_1,p_2,\dots,p_n)}$. We only need to concatenate the appropriate number of T-prefixes and the literal symbol. Since all T-prefixes and the literal symbol may be found in at least one of the intermediate sets $S^{(k_1,k_2,\dots,k_m)}_{(p_1,p_2,\dots,p_m)}$, $m < n$, each of these strings has a T-depletion code associated with it. Hence the concatenation of the T-prefixes and the literal symbol may be done recursively.

If a T-Code set is to be stored by a decoder in a fixed-length format, this may thus be achieved by representing it by the T-depletion codewords for its T-prefixes and their respective T-expansion parameters. This information suffices to permit the recursive encoding or decoding of every codeword in the set.

Decoders: Converting T-Codes into T-Depletion Codewords

The conversion in the other direction, i.e., obtaining a T-depletion codeword from a variable-length T-Code codeword, may be used in decoders. The pseudo-code in Figure 4.1 implements such a recursive decoder. A T-Code codeword at T-augmentation level n is read left-to-right as a concatenation of codewords from the intermediate set at level $n-1$. This concatenation sequence consists of up to k_n T-prefixes and another codeword from the intermediate set. To obtain the number of T-prefixes k'_n, and the T-depletion codeword elements for the remaining suffix (a codeword from the intermediate set at level $n-1$), the decoder routine calls itself recursively as a decoder over the intermediate set. Only at the lowest level, representing codewords from S (i.e., T-augmentation level 0), the routine reads symbols from the decoder's input.

Example 4.2.10 (Decoding T-Code into T-Depletion Codewords)
The T-depletion codeword format for codewords from $S_{(0,1,01)}^{(1,1,3)}$ *is* (k'_3, k'_2, k'_1, k'_0). *We follow the entries in this format as we decode the codeword* 01010101 *from* $S_{(0,1,01)}^{(1,1,3)}$. *The following "snapshots" of the global T-depletion codeword register may be taken at the point indicated in the pseudo-code listing in Figure 4.1:*

- *initial state:* $(-,-,-,-)$, *not initialised. As the procedure calls itself recursively, the T-expansion indices are initialised top-down, i.e.,* $(0,-,-,-)$, *then* $(0,0,-,-)$, $(0,0,0,-)$, *and* $(0,0,0,0)$ *before the first snapshot is taken.*

```
program    conversion;
var
           global x:  string;
           global i:  integer;
           global (k'_n, k'_{n-1}, ..., k'_1, k'_0):  TDepletionCodeWord;

procedure  tconvert(m:  integer);

begin
           k'_m := 0;  {Clear T-expansion index}
           if ( m > 0 ) then begin    {Decode at lower level}
              loop:
              {Check if next lower-level codeword is p_m}
              tconvert(m - 1);    {Decode next lower-level codeword}
              if ( (k'_m, k'_{m-1}, ..., k'_1, k'_0) ≠ d_{m-1}(p_m) ) then break;
              {Found p_m: is it the (k_m + 1)'th copy of p_m?}
              if ( k'_m < k_m ) then
                 k'_m := k'_m + 1;  {increment T-expansion index k'_m}
              else break;  {it's the (k_m + 1)'th copy:}
                 {end of codeword at level m found}
              end loop;
           end else begin    {symbol level reached - read next symbol}
           i := i + 1;
           k'_0 := x[i];
           {The snapshots in Example 4.2.10 are taken at this point}
           end;
end;

begin
           i := 0;  {initialise string pointer}
           x :=ATCodeCodeWord;  {this is the codeword we want to convert}
           tconvert(n);  {run conversion}
           output((k'_n, k'_{n-1}, ..., k'_1, k'_0));  {output result}
end.
```

Figure 4.1. A T-Code decoder: the pseudo-code routine above converts the T-Code codeword x from $S_{(p_1,p_2,\dots,p_n)}^{(k_1,k_2,\dots,k_n)}$ recursively into a T-depletion codeword $d_n(x) = (k'_n, k'_{n-1}, \dots, k'_1, k'_0)$. Note that the T-expansion indices k'_i are determined in decreasing order of i, which permits the use of a single storage register (global variable) in this recursive routine.

- *decode* $x[1] = 0$*: "snapshot"* $(0,0,0,0)$*. Upon return to the calling procedure, it is determined that the* (0) *is a copy of* p_1*, i.e.,* $(0) = d_0(p_1) = (0)$*. Thus* k'_1 *is incremented to* $k'_1 = 1$*. The next run through the loop resets* k'_0 *to* 0 *before the second snapshot is taken.*

- *decode* $x[2] = 1$*: "snapshot"* $(0,0,1,1)$*. Upon return to the calling procedure, it is determined that* $(1) \neq d_0(p_1) = (0)$*. Thus, the procedure itself returns to its own calling level, where* $(1,1)$ *is determined not to be a copy of* p_2*, i.e.,* $(1,1) \neq d_1(p_2) = (0,1)$*, and program execution returns to the next higher level. There, it is found that* $(0,1,1) = d_2(p_3)$*, and* k'_3 *is incremented to* $k'_3 = 1$*. In the next run of the loop,* k'_2, k'_1*, and* k'_0 *are successively reset to* 0*:* $(1,0,0,0)$ *is the state of the register just before the third snapshot.*

- *decode* $x[3] = 0$*: "snapshot"* $(1,0,0,0)$*, another copy of* p_1*. Increment* k'_1 *to* $k'_1 = 1$ *and reset* k'_0*:* $(1,0,1,0)$ *just before the fourth snapshot.*

- *decode* $x[4] = 1$*: "snapshot"* $(1,0,1,1)$*,* $(1,1) \neq d_0(p_1)$ *and* $(1,1) \neq d_0(p_2)$*, but* $(0,1,1) = d_2(p_3)$*, i.e., we have found another copy of* p_3*. Increment* k'_3 *to* $k'_3 = 2$ *and reset* k'_0, k'_1, k'_2*:* $(2,0,0,0)$ *before the fifth snapshot.*

- *decode* $x[5] = 0$*: "snapshot"* $(2,0,0,0)$*, a copy of* p_1*. Increment* k'_1 *and reset* k'_0*, etc.*

- *decode* $x[6] = 1$*: "snapshot"* $(2,0,1,1)$*. As before, increment* k'_3 *and reset* k'_2, k'_1, k'_0 *etc. Now,* $k'_3 = k_3$*, i.e., no more instances of* p_3 *can follow.*

- *decode* $x[7] = 0$*: "snapshot"* $(3,0,0,0)$*, a copy of* p_1*. Increment* k'_1 *and reset* k'_0 *as before:* $(3,0,1,0)$ *just before the eighth snapshot.*

- *decode* $x[8] = 1$*: "snapshot"* $(3,0,1,1)$*. What seems like another copy of* p_3 *is in fact the remainder of the codeword: as* $k'_3 = k_3$*, the* 1 *must be the literal,* *i.e.,* $k'_0 = 1$. *The program returns to its top level and outputs* $(3, 0, 1, 1)$. *We combine* $p_3^3 p_2^0 p_1^1 k'_0 = (01)^3 1^0 0^1 1 = 01010101$ *to verify our result.*

In a full decoder, we may use the fixed-length representation of the final T-depletion codeword output as an address into a look-up table containing the real (fixed-length) decoding of the corresponding T-Code codeword.

4.2.4 Storage Requirements for T-Depletion Codewords

As mentioned above, conventional variable-length encoders often require the storage of both the codeword string and length. In these cases, the size of the field used to store the codeword string is often based on the length of the longest codeword in the code set. This format permits the representation of both final and intermediate encoder states: an incompletely encoded codeword may be represented by the full codeword string accompanied by a length value that is shorter than the string. Let us consider the storage cost associated with a codeword:

Example 4.2.11 (Storage Requirements)
Consider the codeword 01010111 *from a variable-length binary code whose longest codeword is, e.g., nine bits long. The codeword may be represented in a fixed-length binary register by, e.g., a four bit length field and the codeword string itself, padded with* 0*'s to the right (the dot indicates the border between length field and the padded string):*

$$1000.010101110$$

A partially encoded (or transmitted) codeword, e.g., its first five bits, can be represented as follows:

$$0101.010101110$$

As multibase numbers, T-depletion codewords for a given T-Code set may also be stored in a fixed-length format. The storage resource requirement of this format is given by the total number of m-ary register cells required to store all entries of the multibase number. The storage of a single entry with base $(k_i + 1)$ requires $\lceil \log_m(k_i + 1) \rceil$ m-ary register cells[2]. To store an arbitrary T-depletion codeword from $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ in an m-ary register, we hence require

$$L = \sum_{i=0}^{n} \lceil \log_m(k_i + 1) \rceil \tag{4.12}$$

m-ary register cells. In the equation above, (k_i+1) is the i'th base of our multibase number.

Example 4.2.12 (Storage Requirements for T-Depletion Codewords) *Consider the binary set $S_{(0,1,01)}^{(1,1,3)}$ from the previous examples, and presume that we wish to store a T-depletion codeword from this set in a binary ($m = 2$) register. Then $L = 5$, i.e., we require a 5-bit register.*

If we used the format from Example 4.2.11 to store one of the codewords in $S_{(0,1,01)}^{(1,1,3)}$, we would require nine bits to store the string, and another four to store the length information, i.e., a total of thirteen bits.

Note that if $\log_m(k_i + 1) \notin \mathbb{N}$ for some i, there is some inherent inefficiency in the m-ary representation of the multibase number $(k'_n, k'_{n-1}, \ldots, k'_1, k'_0)$. However, this inefficiency is always less than $n + 1$ digits in the m-ary representation.

The **binary depletion codewords** used in [45, 26, 7] implicitly assume that $k'_n = k'_{n-1} = \ldots = k'_1 = 1$, and $m = 2$. In this case, the presence or absence of the T-prefix p_i in a T-Code codeword is indicated by a single bit in the T-depletion codeword. A binary T-Code codeword from such a **simple T-Code set** at T-augmentation level n may thus be represented by a binary register with $n + 1$ bits.

[2] $\lceil y \rceil$ denotes the smallest integer that is larger than or equal to y

4.3 Discussion

This chapter introduced a fixed-length representation of T-Code codewords, the T-depletion code format. It can be derived from the recursive structure of T-Code codewords. The T-depletion codes are thus an efficient encoding for T-Code codewords as they replace the actual codewords by their structural "blueprints". Converters between T-Code and T-depletion code formats may be regarded as encoders and decoders for T-Codes.

However, the multibase number format associated with the T-depletion codewords also permits the representation of multibase numbers that do not correspond to T-Code codewords. As we shall see in Chapter 6, these numbers may be interpreted as incomplete codewords or intermediate decoder states — a feature that can ultimately be used to represent arbitrary variable-length codes in a T-depletion format. Unfortunately though, for a T-Code decoder operating into a look-up table, these "other" multibase numbers pose a problem, which is discussed in the next chapter.

CHAPTER 5

Contiguous Range Index Conversion

The underlying multibase number representation of T-depletion codewords may be used as an address into a look-up table. Such a look-up table may be required, for example, in a decoder as described in the previous chapter. This chapter shows this addressing scheme to be wasteful, and proposes a simple algorithm to convert T-depletion codewords into a contiguous range index.

5.1 Simple Addressing

The previous chapter introduced T-depletion codewords — in essence a multibase number format that can be used to represent T-Code codewords by means of recording the "literal symbol" and the number of T-prefix copies introduced by each T-augmentation. This gives rise to a simple indexing scheme, where we obtain a unique index I for every codeword $x \in S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ as[1]

$$I(x) = k'_0 + \sum_{i=1}^{n} \left[k'_i \prod_{j=0}^{i-1} (k_j + 1) \right]. \tag{5.1}$$

[1] This index is used in the notation for simple T-Code sets discussed in 2.4

In a decoder of the type discussed in the previous chapter, this index may be used as an address into a look-up table where the final decodings of the T-Code codewords are stored. The number of distinct addresses possible under this scheme may be obtained by setting $k'_0 = k_0$ and $k'_i = k_i$ for all i in the equation above. Comparing this with Theorem 2.3.3 on the cardinality of T-Code sets reveals that the number of addresses possible under the above scheme exceeds the number of codewords in the corresponding T-Code set for all $n > 1$.

Hence, there must be multibase numbers that either do not correspond to a T-Code codeword, or that correspond to the same codeword as some other multibase number. The next chapter proves that the former is the case and the latter impossible. However, as the decoding algorithm presented in the last chapter yields only one distinct output for each T-Code codeword, we may assume for the moment that our decoders use this algorithm as their "front end".

Ideally, the decoder should not require more than one address per codeword, and the required index range is given by the T-Code set's cardinality. However, the comparison of Equation (5.1) with Theorem 2.3.3 shows that the range of I may be close to twice the real requirement[2]. In a look-up table, the cost of providing extra storage at addresses that are never accessed may be considered significant.

Rather than using the simple index from Equation (5.1), we would thus prefer a contiguous range integer index. That is, an index that maps, say, all integers between 0 and $\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1$ uniquely to the T-Code codewords in $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$. This chapter presents such an index.

[2]In the case of simple binary T-Codes.

5.2 From T-Depletion Codes to Contiguous Indices

We are looking for a contiguous index starting at 0 and running up to $\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1$. That is, we require an algorithm that takes a T-depletion codeword as its input and outputs the corresponding index. Similarly, we require an algorithm that maps an index straight to its corresponding T-depletion codeword.

To accomplish this, it is useful to define the concept of parents and descendants of T-Code codewords:

Definition 5.2.1 (Parents and Descendants)
Consider a codeword $x \in S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)}$ and a codeword $y \in S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$, where $n > m$ and

$$y = p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_{m+1}^{k'_{m+1}} x \tag{5.2}$$

such that $k'_{m+1}, \ldots, k'_n$ are the respective T-expansion indices in the T-depletion codeword of y. Then x is called a **parent** *of y, and y is called a* **descendant** *of x.*

In the following, the proposed contiguous range integer index of the codeword $x \in S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ will be denoted $I_n(x)$. We also adopt the convention that if $x \in S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ but $x \notin S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)}$ for $m < n$, the index $I_m(x)$ will be the index of the parent of x in $S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)}$, i.e., for $x = p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1} k'_0$ we have

$$I_m(x) = I_m(p_m^{k'_m} p_{m-1}^{k'_{m-1}} \ldots p_1^{k'_1} k'_0). \tag{5.3}$$

Assume that we wish to convert the codeword x with its associated T-depletion codeword

$$d_n(x) = (k'_n, k'_{n-1}, \ldots, k'_1, k'_0)$$

into its contiguous range integer index $I_n(x)$. We may now propose an index and show that it satisfies the criteria of uniqueness, contiguity, and the parent-descendant relationship of Equation (5.3).

Proposition 5.2.2

The mapping $I_0 : S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)} \to \mathbb{N}$, *with*

$$I_0(x) = k'_0, \tag{5.4}$$

is a contiguous integer index at T-augmentation level 0.

This is the trivial case and requires no further discussion. We adopt $I_0(x)$ as our index at T-augmentation level 0 and propose a derivation for indices at higher T-augmentation levels:

Proposition 5.2.3

Let $I_m(x)$ *be the contiguous integer index at T-augmentation level* m *with* $0 \leq m < n$. *The mapping* $I_{m+1} : S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)} \to \mathbb{N}$, *with*

$$I_{m+1}(x) = \begin{cases} I_m(x) + k'_{m+1}\left(\#S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)} - 1\right) \\ \quad \text{for } I_m(x) \leq I_m(p_{m+1}) \vee k'_{m+1} = k_{m+1} \\ \\ I_m(x) + k'_{m+1}\left(\#S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)} - 1\right) - 1 \\ \quad \text{for } I_m(x) > I_m(p_{m+1}) \wedge k'_{m+1} < k_{m+1} \end{cases} \tag{5.5}$$

is a contiguous integer index at T-augmentation level $m+1$.

Proof: by induction over n.

Uniqueness: $I_0(x)$ is obviously unique for all codewords in S. We also note that Equation (5.5) does not permit negative indices. Presume that there are two distinct codewords $y, y^* \in S_{(p_1,p_2,\ldots,p_{n+1})}^{(k_1,k_2,\ldots,k_{n+1})}$ such that $I_{n+1}(y) = I_{n+1}(y^*)$. The T-expansion indices of y and $y*$ at level $n+1$ are k^*_{n+1} and k^{**}_{n+1} respectively. Without loss of generality, we may assume that $I_n(y) \geq I_n(y^*)$. There are four cases:

1. $I_{n+1}(y) = I_n(y) + k^*_{n+1}(\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1)$ and $I_{n+1}(y^*) = I_n(y^*) + k^{**}_{n+1}(\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1)$: with the indices at level $n+1$ presumed equal, we may write

$$I_n(y) - I_n(y^*) = (k^{**}_{n+1} - k^*_{n+1})(\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1).$$

For the term on the left hand side, the properties of the index at level n imply that $I_n(y) - I_n(y^*) \leq (\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1)$. Since the right hand side is a multiple of $(\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1)$, the left hand side can only either be zero or $\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1$. For the first case, we require $k^{**}_{n+1} = k^*_{n+1}$ and $I_n(y) = I_n(y^*)$, which is impossible because it implies $y = y^*$. The second case implies $k^{**}_{n+1} = k^*_{n+1} + 1$, $I_n(y) = (\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1)$, and $I_n(y^*) = 0$. Since this rules out $k^{**}_{n+1} = k^*_{n+1}$, we require both $I_n(y) \leq I_n(p_{n+1})$ and $I_n(y^*) \leq I_n(p_{n+1})$. With $I_n(y) = \#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1$, we get $I_n(p_{n+1}) = \#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1$, i.e., p_{n+1} must be the parent of y in $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ such that $y = p_{n+1}^{k^*_{n+1}+1}$. However, we concluded that $k^*_{n+1} < k_{n+1}$, such that y cannot be in $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$.

2. $I_{n+1}(y) = I_n(y) + k^*_{n+1}(\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1)$ and $I_{n+1}(y^*) = I_n(y^*) + k^{**}_{n+1}(\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1) - 1$. Following the same argument as above, we get

$$I_n(y) - I_n(y^*) + 1 = (k^{**}_{n+1} - k^*_{n+1})(\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1).$$

In order to match the term on the right hand side, the term on the left hand side can only assume the values of 0 or $(\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1)$. The first case implies $I_n(y) = I_n(y^*) - 1$ which contradicts our assumption that $I_n(y) \geq I_n(y^*)$. In the second case, we require $k^{**}_{n+1} = k^*_{n+1} + 1$, and either

- $I_n(y) = \#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 2$ and $I_n(y^*) = 0$, or

- $I_n(y) = \#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1$ and $I_n(y^*) = 1$.

As $k^{**}_{n+1} = k^*_{n+1}+1$ implies that $k^*_{n+1} < k_{n+1}$, we require $I_n(y) \leq I_n(p_{n+1})$. At the same time we know that $I_n(y^*) > I_n(p_{n+1})$. As all indices must be non-negative, the first option is ruled out by $I_n(y^*) = 0 > I_n(p_{n+1})$. The second option requires $I_n(p_{n+1})$ to simultaneously be 0 at least as large as $\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1$, which is impossible.

3. $I_{n+1}(y) = I_n(y) + k^*_{n+1}(\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1) - 1$ and $I_{n+1}(y^*) = I_n(y^*) + k^{**}_{n+1}(\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1)$. This yields

$$I_n(y) - I_n(y^*) - 1 = (k^{**}_{n+1} - k^*_{n+1})(\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1).$$

Here, the left hand side can only be zero because of the range of the indices. It follows that $k^{**}_{n+1} = k^*_{n+1}$, which implies that $k^{**}_{n+1}, k^*_{n+1} < k_{n+1}$ and hence $I_n(y^*) \leq I_n(p_{n+1})$. It also implies that $I_n(y) = I_n(y^*) + 1$. Since $I_n(y) > I_n(p_{n+1})$ we get $I_n(y^*) = I_n(p_{n+1})$, i.e., p_{n+1} is the parent of y^* in $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$. This means that the parent of y^* in $S^{(k_1,k_2,\ldots,k_{n+1})}_{(p_1,p_2,\ldots,p_{n+1})}$ is $p_{n+1}^{k^{**}_{n+1}+1}$. However, since $k^{**}_{n+1} < k_{n+1}$, this string is not a codeword in $S^{(k_1,k_2,\ldots,k_{n+1})}_{(p_1,p_2,\ldots,p_{n+1})}$.

4. $I_{n+1}(y) = I_n(y) + k^*_{n+1}(\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1) - 1$ and $I_{n+1}(y^*) = I_n(y^*) + k^{**}_{n+1}(\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1) - 1$. Again,

$$I_n(y) - I_n(y^*) = (k^{**}_{n+1} - k^*_{n+1})(\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1).$$

Since $I_n(y^*) > I_n(p_{n+1})$, the left hand side can only be zero, which implies that y and y^* have the same parent in $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$. Furthermore, we have $k^{**}_{n+1} = k^*_{n+1}$ which means that $y = y^*$. This contradicts our assumption that they are distinct.

Contiguity: the minimum value of $I_{n+1}(x)$ is given by the minimum value of $I_n(x)$, which is 0. The maximum value is given by the maximum value of

$I_n(x) + k_{n+1}(\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} - 1)$, which is $\#S^{(k_1,k_2,\ldots,k_{n+1})}_{(p_1,p_2,\ldots,p_{n+1})} - 1$. Since $I_{n+1}(x)$ exists for every $x \in S^{(k_1,k_2,\ldots,k_{n+1})}_{(p_1,p_2,\ldots,p_{n+1})}$, uniqueness of $I_{n+1}(x)$ implies contiguity.

Parent-Descendant-Relationship: this is satisfied because the $I_n(x)$ depends only on the T-expansion indices at levels 0 to n.

Thus, $I_n(x)$ is a contiguous integer index for $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$. □

The motivation behind Propositions 5.2.2 and 5.2.3 lies in the copying process of the T-augmentation. During a T-augmentation from $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ to $S^{(k_1,k_2,\ldots,k_{n+1})}_{(p_1,p_2,\ldots,p_{n+1})}$, the T-prefix p_{n+1} is removed from the original list and all but the "last" copy of the list (for which $k'_{n+1} = k_{n+1}$). Accordingly, all codewords with a parent whose index falls below that of p_{n+1}, are simply given an "index offset" (T-expansion index times length of the list without T-prefix). Those with a parent index above that of p_{n+1} are treated the same way, except that they are also shifted by one position to compensate for the removal of p_{n+1} from the original codeword list. Codewords in the last copy ($k'_{n+1} = k_{n+1}$) are never shifted, because the codeword $p_{n+1}^{k_{n+1}+1}$ needs to be accounted for.

The conversion is illustrated in the following example, based on our example set $S^{(1,1,3)}_{(0,1,01)}$ (cf. Table 2.1).

Example 5.2.4 (Index Conversion of a T-Depletion Codeword)
The T-depletion code $d_3(010100) = (2, 0, 1, 0)$ *of the codeword* 010100 *may be converted as follows:*

We start with the binary alphabet $S = \{0, 1\}$ *as the T-Code set at T-augmentation level 0. The indices of the codewords in this set are:*

codeword	I_0
0	0
1	1

The codeword that our T-depletion code refers to ends in a 0*, and thus has the index* $I_0(x) = 0$*. The next step is to find the indices for* $S^{(1)}_{(0)}$*. From the table above, the index for* $p_1 = 0$ *is* 0*, and with* $k_1 = 1$ *we obtain the following "copies" of* S*:* 1 *and* 00, 01*. The codewords in the latter copy have the T-prefix* p_1 *with* $k'_1 = k_1 = 1$*. The new indices are:*

codeword	I_1
1	0
00	1
01	2

We see here that the index of the codeword 1 *has dropped by one, because its original index* $I_0(1) = 1$ *was larger than that of the T-prefix* p_1*,* $I_0(0) = 0$*. The index for* 01 *is not adjusted downwards because it is the "last" copy of* S*. The index to watch, however, is* $I_1(00)$*, as* 00 *is the suffix of our codeword. It has been calculated using equation 5.5:*

$$\begin{aligned} I_1(x) &= I_1(00) = I_0(x) + k'_1(\#S - 1) \\ &= I_0(0) + 1(2-1) = 0 + 1(2-1) \\ &= 1. \end{aligned} \tag{5.6}$$

In the next step, we take the T-prefix p_2 *and note its index,* $I_1(p_2) = I_1(1) = 2$*. The copies of* $S^{(1)}_{(0)}$ *in the next table consist of the codewords* 00, 01 *(zero'th copy) and* 11, 100, 101 *(first copy). The indices for* $S^{(1,1)}_{(0,1)}$ *are:*

codeword	I_2
00	0
01	1
11	2
100	3
101	4

We can see that the removal of 1 *from the zero'th copy has affected the indices for* 00 *and* 01 *because their indices* $I_1(00) = 1$ *and* $I_1(01) = 2$ *were larger than* $I_1(p_2) = I_1(1) = 0$. $I_2(x)$ *is calculated using* $k'_2 = 0$*:*

$$\begin{aligned} I_2(x) &= I_2(00) = I_1(x) + k'_2\left(\#S^{(1)}_{(0)} - 1\right) - 1 \\ &= I_1(00) + k'_2\left(\#S^{(1)}_{(0)} - 1\right) - 1 \\ &= 1 + 0(3-1) - 1 \\ &= 0. \end{aligned} \tag{5.7}$$

At the third T-augmentation level, we have $k_3 = 3$ *and thus a total of four copies (counting from zero):* 00, 11, 100, 101 *(zero'th copy),* 0100, 0111, 01100, 01101 *(first copy),* 010100, 010111, 0101100, 0101101 *(second copy), and* 01010100, 01010101, 01010111, 010101100, 010101101 *(third copy). The index of the T-prefix,* $I_2(p_3) = I_2(01) = 1$. *The final indices are thus*

codeword	I_3	*codeword*	I_3
00	0	010111	9
11	1	0101100	10
100	2	0101101	11
101	3	01010100	12
0100	4	01010101	13
0111	5	01010111	14
01100	6	010101100	15
01101	7	010101101	16
010100	8		

The index for 010100 *in particular is given by*

$$I_3(010100) = I_2(010100) + k'_3\left(\#S^{(1,1)}_{(0,1)} - 1\right)$$

$$\begin{aligned} &= I_2(00) + 2\,(5-1) \\ &= 0 + 8 \\ &= 8, \end{aligned} \tag{5.8}$$

using $k_3' = 2$. Note that this time, $I_2(00) < I_2(p_3) = I_2(01)$, such that there is no need to adjust the index downwards. The reader may verify that the other indices may be obtained from the T-depletion codes in table 4.2 in the same manner.

The calculation of an index requires the knowledge of the indices for the T-prefixes at all lower levels, i.e., $I_0(p_1), I_1(p_2), \ldots, I_{n-1}(p_n)$. These may be obtained either in advance (suitable for a hardware or software implementation) or recursively during the indexing process (suitable for software implementations only, where speed is not an issue).

5.3 From Contiguous Indices to T-Depletion Codes

Given an index $I_n(x)$ for a codeword $x \in S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$, one may ask how to find the associated T-depletion codeword. The following recursive rules reverse the indexing process:

$$k_m' = \begin{cases} \left\lfloor \frac{I_m(x)}{\#S_{(p_1,p_2,\ldots,p_{m-1})}^{(k_1,k_2,\ldots,k_{m-1})} - 1} \right\rfloor & \text{for } \left\lfloor \frac{I_m(x)}{\#S_{(p_1,p_2,\ldots,p_{m-1})}^{(k_1,k_2,\ldots,k_{m-1})} - 1} \right\rfloor \leq k_m \\ k_m \text{ otherwise} \end{cases} \tag{5.9}$$

$$I_{m-1}(x) = \begin{cases} I_m(x) - k'm(\#S^{(k_1,k_2,\ldots,k_{m-1})}_{(p_1,p_2,\ldots,p_{m-1})} - 1) \text{ for} \\ \qquad \begin{array}{c} I_m(x) - k'm(\#S^{(k_1,k_2,\ldots,k_{m-1})}_{(p_1,p_2,\ldots,p_{m-1})} - 1) \\ < I_{m-1}(p_m) \end{array} \\ \qquad \text{or} \\ \qquad k_m = k'_m \\ I_m(x) - k'm(\#S^{(k_1,k_2,\ldots,k_{m-1})}_{(p_1,p_2,\ldots,p_{m-1})} - 1) + 1 \text{ otherwise} \end{cases} \tag{5.10}$$

5.4 Discussion

The contiguous integer index introduced in the last section provides a tool for the conversion of T-depletion codes into an address that is suitable for an efficient look-up table. A conversion in the reverse direction is also possible. A decoder that decodes, say, T-Code codewords representing ASCII characters, may thus be built as a three stage device: T-Code codeword to T-depletion codeword conversion, followed by T-depletion codeword to index conversion, followed by the final look-up table. Under resource considerations this is not an inefficient design, as the first two conversions require resources that that depend on the number of T-augmentation levels and the T-expansion parameters involved. This becomes negligible compared to the size of the look-up table as the cardinality of the T-Code set increases.

At first glance, the "holes" in the simple addressing scheme are unused resources. However, as the next chapter shows, they have a well-defined significance.

CHAPTER 6

Storing Arbitrary Variable-Length Codes in T-Depletion Code Format

The previous chapter has left an open question: if some multibase numbers in a T-depletion code format do not actually correspond to a valid T-depletion codeword for a T-Code set, then what is their significance? This chapter shows that they have in fact a well-defined role to play: they correspond uniquely to the proper prefixes of T-Code codewords and may thus be interpreted as intermediate decoder states. This opens a way to store arbitrary variable-length codewords in T-depletion code format, leaving us with another — as yet still open — problem.

6.1 Pseudo-T Codewords

The contiguous-range indexing algorithm presented in the last chapter was developed to overcome one problem: not all multibase numbers corresponding to the T-depletion code format for a given T-Code set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ yield a valid T-Code codeword if used as input to the encoder algorithm presented in Chapter 4. The index conversion provides a "workaround" which can be used at the decoder side, i.e., the focus so far has been on those multibase numbers that correspond to T-Code codewords. In this chapter, we will explain the significance of those multibase numbers that do *not* correspond to T-Code codewords.

The reader may have noticed that Theorem 4.2.3 demands that all T-Code codewords conform to a certain format, that is, the format specified by Equation (4.5). However, the theorem does not claim that all strings satisfying this format are necessarily T-Code codewords. Given a T-Code set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$, consider a string x for which "T-expansion indices" k'_i with $0 \le k'_i \le k_i$ exist such that x may be written in the form given by Equation (4.5).

For example, the string $x = 01010110$ satisfies Equation (4.5) for $S_{(0,1,01)}^{(1,1,3)}$ if it is written as $x = (01)^3 1^1 0^0 0$, i.e., if we choose $k'_3 = 3$, $k'_2 = 1$, $k'_1 = 0$, and $k'_0 = 0$. Thus, x may be represented in the T-depletion code format. However, it is evident from Table 2.1 that $x \notin S_{(0,1,01)}^{(1,1,3)}$. Such strings will be referred to as "pseudo-T codewords":

Definition 6.1.1 (Pseudo-T Codewords)
Strings $x \notin S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ that may be written in the form of Equation (4.5) for some k'_i with $0 \le k'_i \le k_i$ for $i = 0, \ldots, n$ are called **pseudo-T codewords** *for $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$. We also define the empty string λ to be a pseudo-T codeword. The set of all pseudo-T codewords for $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ is denoted $\Phi\left(S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}\right)$.*

This definition provides for the existence of pseudo-T codewords, but tells little about their properties. The following theorem provides us with an alternative to this definition and an insight into the structure of pseudo-T codewords.

Theorem 6.1.2 (Decomposition of Pseudo-T Codewords)
The set of all pseudo-T codewords for $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ may be written as

$$\Phi\left(S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}\right) = \{x | x = p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1} \lambda, 0 \le k'_i \le k_i, i = 1, \ldots, n\}. \tag{6.1}$$

Proof: by induction over n. We note that for $n = 0$, where $\Phi(S) = \{\lambda\}$, the assertion is true.

We now require proof that

$$\Phi\left(S^{(k_1,k_2,\ldots,k_{n+1})}_{(p_1,p_2,\ldots,p_{n+1})}\right) \subseteq \{x | x = p_{n+1}^{k'_{n+1}} p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1}\lambda, 0 \leq k'_i \leq k_i, i = 1, \ldots, n+1\}$$

and

$$\Phi\left(S^{(k_1,k_2,\ldots,k_{n+1})}_{(p_1,p_2,\ldots,p_{n+1})}\right) \supseteq \{x | x = p_{n+1}^{k'_{n+1}} p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1}\lambda, 0 \leq k'_i \leq k_i, i = 1, \ldots, n+1\}.$$

"$\subseteq$": let $x \in \Phi\left(S^{(k_1,k_2,\ldots,k_{n+1})}_{(p_1,p_2,\ldots,p_{n+1})}\right)$. We need to show that there are k'_i with $0 \leq k'_i \leq k_i$ such that $x = p_{n+1}^{k'_{n+1}} p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1}$.

If $x = \lambda$, then this is true if $k'_i = 0$ for all i. If $x \neq \lambda$, then by Definition 6.1.1, x satisfies Equation (4.5) such that

$$x = p_{n+1}^{k''_{n+1}} p_n^{k''_n} p_{n-1}^{k''_{n-1}} \ldots p_1^{k''_1} k''_0. \tag{6.2}$$

There are two cases:

1. $p_n^{k''_n} p_{n-1}^{k''_{n-1}} \ldots p_1^{k''_1} k''_0 \in S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$: since $x \notin S^{(k_1,k_2,\ldots,k_{n+1})}_{(p_1,p_2,\ldots,p_{n+1})}$, Equation (2.1) implies that this is only possible if $k''_{n+1} < k_{n+1}$ and $p_n^{k''_n} p_{n-1}^{k''_{n-1}} \ldots p_1^{k''_1} k''_0 = p_{n+1}$. In this case, choose $k'_{n+1} = k''_{n+1} + 1$ and $k'_i = 0$ for $i \leq n$ to satisfy the assertion.

2. $p_n^{k''_n} p_{n-1}^{k''_{n-1}} \ldots p_1^{k''_1} k''_0 \in \Phi\left(S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}\right)$: by induction hypothesis, there are k'_i such that
$$p_n^{k''_n} p_{n-1}^{k''_{n-1}} \ldots p_1^{k''_1} k''_0 = p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1} \tag{6.3}$$
and hence choose $k'_{n+1} = k''_{n+1}$ to satisfy the assertion.

"$\supseteq$": here, the aim is to show that any word $x = p_{n+1}^{k'_{n+1}} p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1}$ with $0 \leq k'_i \leq k_i$ for $i \leq n+1$ is not in $S^{(k_1,k_2,\ldots,k_{n+1})}_{(p_1,p_2,\ldots,p_{n+1})}$ but nevertheless is either the empty string λ or satisfies Equation (4.5) for some $0 \leq k''_i \leq k_i$ for $i \leq n+1$. Lemma 4.2.1 and the prefix-freeness of $S^{(k_1,k_2,\ldots,k_{n+1})}_{(p_1,p_2,\ldots,p_{n+1})}$ guarantee that $x \notin S^{(k_1,k_2,\ldots,k_{n+1})}_{(p_1,p_2,\ldots,p_{n+1})}$.

If $k'_i = 0$ for all i, then $x = \lambda$ and the assertion is trivially satisfied. If $k'_i > 0$ for some i, then define m such that $k'_m > 0$ and $k'_i = 0$ for all $i < m$. Since $p_m \in S^{(k_1,k_2,\ldots,k_{m-1})}_{(p_1,p_2,\ldots,p_{m-1})}$, Theorem 4.2.3 may be used to write

$$p_m = p_{m-1}^{k''_{m-1}} \ldots p_1^{k''_1} k''_0 \tag{6.4}$$

for some k''_i with $0 \le k''_i \le k_i$ for $i = 0, \ldots, m-1$. Now set $k''_m = k'_m - 1$ and $k''_i = k'_i$ for $i > m$ to satisfy Equation (4.5).

This concludes the proof of Theorem 6.1.2. □

Together, Theorems 4.2.3 and 6.1.2 yield an alternative expression for the T-Code set $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ in a non-recursive form:

$$S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} = \bigcup_{\substack{0 \le k'_1 \le k_1, \ldots \\ \ldots, 0 \le k'_n \le k_n, \\ k'_0 \in S}} \{p_n^{k'n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1} k'_0\} \setminus \bigcup_{\substack{0 \le k'_1 \le k_1, \ldots \\ \ldots, 0 \le k'_n \le k_n}} \{p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1}\}. \tag{6.5}$$

What is the significance of pseudo-T codes? Comparing Equation (6.1) with Equation (4.5), we find that for all $k'_0 \in S$ and $x \in \Phi\left(S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}\right)$, strings of the form xk'_0 satisfy the general form of T-Code codewords in Equation (4.5). This motivates the following theorem:

Theorem 6.1.3 (Pseudo-T Codewords are Proper Prefixes of T-Codes) *Every pseudo-T codeword for $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ is a proper prefix of a T-Code codeword in $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$. Conversely, every proper prefix of a T-Code codeword in $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ is a pseudo-T codeword for $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$. That is,*

$$\Phi\left(S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}\right) = \left\{x \in S^\star | \exists y \in S^+ \textit{ s.t. } xy \in S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}\right\} \tag{6.6}$$

Proof: the inclusion

$$\Phi\left(S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}\right) \subseteq \left\{x \in S^\star | \exists y \in S^+ \text{ s.t. } xy \in S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}\right\}$$

follows from Theorem 6.1.2 and Lemma 4.2.1. The inclusion

$$\Phi\left(S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}\right) \supseteq \left\{x \in S^\star | \exists y \in S^+ \text{ s.t. } xy \in S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}\right\}$$

may be proven by induction as follows:

Let x be a T-Code codeword in $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$, and let $x[1:i]$ for $i = 0, \ldots, |x| - 1$ denote its proper prefix comprising the first i symbols in x, where $x[1:0] = \lambda$.

By Theorem 4.2.3, x has the form of Equation (4.5) and thus $x[1:|x|-1]$ is a pseudo-T codeword by Theorem 6.1.2.

By the induction hypothesis, let $x[1:i]$ with $|x[1:i]| > 0$ be a pseudo-T codeword. Then, by Definition 6.1.1, it may be written in the form of Equation (4.5). Thus, by Theorem 6.1.2, $x[1:i-1]$ is a pseudo-T codeword. □

Every pseudo-T-Code codeword may be written in the form of Equation (4.1) and, with the exception of the empty word λ, every pseudo-T-Code codeword may be written in the form of Equation (4.5). It follows from Lemma 4.2.2 that both of these forms are unique.

In the decoding tree of a T-Code set, the T-Code codewords thus occupy the leaf (terminal) nodes of the tree, whereas the pseudo-T codewords represent all the internal (branch) nodes of the decoding tree. As a decoder traverses the tree, it will encounter these branching nodes. Pseudo-T codewords may thus be interpreted as intermediate decoder states that occur in an incomplete decoding. In fact, the reader may verify from the "snapshots" in Example 4.2.10 that this is in fact the case for the decoder algorithm presented in Chapter 4. As the next section shows, this enables us to represent any finite and prefix-free variable-length code by a T-depletion codeword.

6.2 Pseudo-T Codewords and Variable-Length Codes

If any prefix of a T-Code codeword may be represented in a T-depletion codeword format, then any prefix-free variable-length code C can be represented in this way, provided there exists a T-Code set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ such that all codewords in C are prefixes of codewords in $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$.

Definition 6.2.1 (Covering Code Sets)

Let $C, C' \subset S^+$ be finite and prefix-free variable-length codes. C' is said to be a **covering set** *of C iff*

$$\forall x \in C : \ \exists\, y \in S^\star, z \in C' \ s.t.\ xy = z. \tag{6.7}$$

In this case, we also say that "C' covers *C".*

The concept of covering code sets may be visualised by considering the decoding trees of the two sets involved. However, we must ask whether there is a covering T-Code set for every code C.

Theorem 6.2.2 (Existence of Covering T-Code Sets)

For any finite and prefix-free code $C \subset S^+$, there exists a T-Code set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ that covers C.

Proof: a covering T-Code set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ for a given code C may always be found with the following algorithm:

1. start with S at T-augmentation level $m = 0$, and define some lexicographical order on the elements of C.

2. if $S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$ covers C, finish.

3. from the lexicographical order on C, select the first element $x \in C$ that is not a prefix of any codeword in $S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$. Since $S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$ is complete and prefix-free, there exists a unique codeword $x' \in S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$ such that x' is a proper prefix of x. Set $p_{m+1} = x'$, $k_{m+1} = 1$, and T-augment to $S_{(p_1,p_2,\ldots,p_{m+1})}^{(k_1,k_2,\ldots,k_{m+1})}$. Increment m and continue with step 2.

This algorithm is guaranteed to terminate, because

- in the third step, $|x|$ (the length of x) is finite. However, since T-Code sets are complete, the length of x' in step 3 increases with each T-augmentation (as long as x is not a prefix of x'). Thus, $|x'|$ will eventually reach or exceed $|x|$, such that x becomes a prefix of x', i.e., the number of T-augmentations for a given x is finite.

- the number of strings in C (i.e., the number of possible x) is finite, such that the total number of T-augmentations must be finite.

This proves the existence of covering T-Code sets. □

It is obvious that there is no unique covering T-Code set for a given C. For example, any T-augmentation of a covering T-Code set yields another covering T-Code set. Restricting T-expansion parameters to a value of 1, or using a particular lexicographical order on C in the above algorithm, are also arbitrary constraints which, if changed, may lead to vastly different solutions.

Under "storage cost" considerations, the covering T-Code set of our choice would obviously be a T-Code set for which the storage cost of the T-depletion code format is minimised. The algorithm proposed in the proof above, however, does not work optimally in this sense. This is illustrated by the following example:

Example 6.2.3 (Inefficiency of Covering T-Code Sets)
Let $S = \{0, 1\}$ be the binary alphabet. Consider the code

$$C = \{0000, 0001, 00100, 00101, 0011, 01, 100, 101, 11\}.$$

C is actually a T-Code set, $S_{(0,1,00)}^{(1,1,1)}$, and from Section 4.2.4 we know that its T-depletion codewords require 4 bits of storage. However, using the above algorithm (presuming a lexicographical ordering of the codewords in C as listed above), we obtain the covering T-Code set $S_{(0,00,001,1)}^{(1,1,1,1)}$, which requires 5 bits of storage.

An "optimal" algorithm in the sense of finding a covering T-Code set with minimal storage cost remains an open problem.

6.3 Discussion

The fact that all multibase numbers in the T-depletion codeword format have a well-defined significance is perhaps somewhat surprising. However, it is also quite satisfying as it clearly addresses the question of storage efficiency. The decoder algorithm presented in Chapter 4 uses only a single multibase register — we now understand that this is the minimum storage required to represent all decoder states.

The representation of arbitrary variable-length codes in a T-depletion code format is potentially interesting both from an application point of view and in attempting to explain the self-synchronisation capabilities of variable-length codes [50].

CHAPTER 7

Hierarchical Coding Alphabets and T-Codes

In the previous chapters, the recursive nature of the T-Code set construction has been a dominant topic. This chapter shows that the hierarchy of T-Code sets leads to a natural hierarchy of decodings in strings composed of T-Code codewords. T-decomposition, a convenient way to recover T-Code set information from one of the longest codewords in a set, is discussed as an application.

7.1 Hierarchical Coding Alphabets

The notion of hierarchical coding alphabets is by no means a feature connected exclusively with T-Codes. In fact, there are many instances in which hierarchical coding appears implicitly:

Consider, for example, the "Collected Works of William Shakespeare" or similar literary works stored as a single text file on a computer. Such a file may be divided into volumes, chapters, paragraphs, sentences, words, bytes and bits, which may be regard as hierarchical layers of encoding.

A boundary between two volumes coincides with a boundary between two chapters, a boundary between two chapters always coincides with a boundary between two paragraphs, and so on. In the other direction of this hierarchy, however, boundaries are not generally shared across the hierarchy: a boundary between bits is generally not also a boundary between bytes — a principle that holds at each layer of the coding hierarchy: an upper layer (e.g., words) shares its codeword boundaries with the lower layer(s) (e.g., bytes and bits), but the reverse is not true.

Definition 7.1.1 (Hierarchical Codes)
A series of code sets $C_1, C_2, C_3, \ldots C_n$, for which $C_{i+1} \subset C_i^+$ for all $i < n$, is called a code hierarchy*. The code sets $C_1, C_2, C_3, \ldots C_n$ are called* hierarchical codes *or* hierarchical coding alphabets*.*

Thus, we may think of a "higher level" coding alphabet as being composed of symbols from a "lower level" code set.

Lemma 7.1.2 (T-Codes are Hierarchical Codes)
A T-Code set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ and its intermediate set are hierarchical codes.

Proof: follows from Definition 2.1.1. □

This conclusion (pre-empted somewhat by the previous chapters) has relatively far-reaching consequences. For example, a string of codewords from $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ may be viewed as a string of codewords from any one of its intermediate T-Code sets. As we shall see in the next chapter, this is the basis of the T-Code self-synchronisation theory.

The same concept may be applied to boundaries between codewords:

S	$1 \mid 0 \mid 0 \mid 1 \mid 0 \mid 0 \mid 1 \mid 0 \mid 1 \mid 0 \mid 0 \mid 1 \mid 1 \mid 1 \mid 0 \mid 1 \mid 0 \mid 1 \mid 1 \mid 1 \mid 1$
$S_{(1)}^{(1)}$	$1\ 0 \mid 0 \mid 1\ 0 \mid 0 \mid 1\ 0 \mid 1\ 0 \mid 0 \mid 1\ 1 \mid 1\ 0 \mid 1\ 0 \mid 1\ 1 \mid 1\ 1$
$S_{(1,10)}^{(1,1)}$	$1\ 0\ 0 \mid 1\ 0\ 0 \mid 1\ 0\ 1\ 0 \mid 0 \mid 1\ 1 \mid 1\ 0\ 1\ 0 \mid 1\ 1 \mid 1\ 1$
$S_{(1,10,0)}^{(1,1,3)}$	$1\ 0\ 0 \mid 1\ 0\ 0 \mid 1\ 0\ 1\ 0 \mid 0\ 1\ 1 \mid 1\ 0\ 1\ 0 \mid 1\ 1 \mid 1\ 1$

Table 7.1. 0-, 1-, 2-, and 3-boundaries between codewords from intermediate T-Code sets in a binary string.

Definition 7.1.3 (n-Boundary)

*Consider a finite string $x \in {S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}}^+$. A codeword boundary between two codewords in the decoding of x over $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ is called an n-**boundary**.*

Example 7.1.4 (n-Boundaries)

Table 7.1 shows the 0-, 1-, 2-, *and* 3-*boundaries in the binary string*

$$100100101001110101111.$$

The following corollary establishes the hierarchical relationship:

Lemma 7.1.5 (Hierarchy of n-Boundaries)

For every $m \leq n$, an n-boundary is also an m-boundary.

Proof: follows from Definition 2.1.1. □

In the reverse direction, the following lemma applies:

Lemma 7.1.6 (A Sufficient Condition for n-Boundaries)

In a finite string $x \in {S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}}^+$, every $(n-1)$-boundary following a codeword other than p_n is also a n-boundary.

Proof: follows from Definition 2.1.1. □

We may ask under which conditions strings in $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)^+}$ are also strings in $S_{(p_1,p_2,\ldots,p_{n+1})}^{(k_1,k_2,\ldots,k_{n+1})^+}$. The following lemma provides a sufficient condition:

Lemma 7.1.7 (A Sufficient Condition for Hierarchy in Strings)
Consider a finite string $x \in S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)^+}$. If the last codeword in the decoding of x over $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ is not p_{n+1}, then $x \in S_{(p_1,p_2,\ldots,p_{n+1})}^{(k_1,k_2,\ldots,k_{n+1})^+}$.

Proof: presume that the condition is satisfied, i.e., the last codeword in the decoding of x over $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ is not p_{n+1}. Then we may split the decoding of x into substrings of the form $p_{n+1}^{k'}y'$, where k is some non-negative integer and $y' \in S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)} \backslash \{p_{n+1}\}$. It suffices to show that each substring of this form is in $S_{(p_1,p_2,\ldots,p_{n+1})}^{(k_1,k_2,\ldots,k_{n+1})^+}$.

We distinguish two cases:

1. $k' \leq k_{n+1}$ for all substrings involved: in this case, $p_{n+1}^{k'}y' \in S_{(p_1,p_2,\ldots,p_{n+1})}^{(k_1,k_2,\ldots,k_{n+1})}$ for all substrings and the assertion is true.

2. $k' > k_{n+1}$ for some of the substrings: in this case, we may factor out one or more codewords of the form $p_{n+1}^{k_{n+1}+1}$ from the substring(s) concerned such that the previous case is satisfied. □

A practical application of the hierarchical nature of T-Codes is the *T-decomposition* [1] of strings from S^+. It permits the construction of the entire code set T-Code set from the knowledge of one of its longest codewords. In a communication situation, it is thus sufficient to transmit any one of the longest codewords to the receiver — rather than having to transmit the whole code set. The next section discusses *T-decomposition* in detail.

[1] the T-decomposition algorithm here should not be confused with the "decomposition" of codewords as described in Chapter 4. The latter is not an algorithm, but merely states the fact that a string may be written as a concatenation of the substrings specified.

7.2 T-Decomposition of Strings in $S^\star$

The original T-decomposition algorithm was found empirically by Mark Titchener [48]. On his suggestion, Nicolescu investigated and eventually proved the existence and uniqueness of a T-decomposition for every string $x \in S^\star$ in his "Uniqueness Theorem for T-Codes" [36]. These results may be stated in a modified form as follows:

Theorem 7.2.1 (Uniqueness Theorem for T-Codes)

String defines set: *there exists a known mapping*

$$\Delta_S : S^\star \longrightarrow \{X | X \text{ is a T-Code set over } S\}$$

such that for any $a \in S$ and $x \in S^\star$, the string xa is one of the longest codewords in the T-Code set $\Delta_S(x)$.

Set uniqueness: *Δ_S is 1-to-1 and onto.*

Proof: Part 1: existence of Δ_S. It suffices to show the existence of an unambiguous algorithm that derives a set of T-prefixes and T-expansion parameters from every string $x \in S^\star$ such that

$$x = p_n^{k_n} p_{n-1}^{k_{n-1}} \dots p_1^{k_1}. \tag{7.1}$$

This is the appropriate form for the longest codewords in $S_{(p_1,p_2,\dots,p_n)}^{(k_1,k_2,\dots,k_n)}$ with the last symbol removed (cf. Equation (4.5)). Note that n itself may also be determined by the algorithm. Provided that such an algorithm exists, we may use it to perform the mapping Δ_S.

Consider the following algorithm:

1. Set $m = 0$.

2. Decode xa as a string of codewords from $S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$.

3. If xa decoded into a single codeword from $S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$, set $n = m$ and finish.

4. Otherwise, set the T-prefix p_{m+1} to be the second-to-last codeword in the decoding over $S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$.

5. Count the number of adjacent copies of p_{m+1} that immediately precede the second-to-last codeword. Add 1 to this number, and define it to be the T-expansion parameter k_{m+1}.

6. T-augment with p_{m+1} and k_{m+1}.

7. Increment m by 1 and goto step 2 above.

All of this is straightforward, provided that the decoding in the second step exists. This needs to be proven. For $m = 0$ this is trivial. For $m > 0$, we need to show that xa will decode as a string of codewords from $S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$, provided it is a string of codewords over $S_{(p_1,p_2,\ldots,p_{m-1})}^{(k_1,k_2,\ldots,k_{m-1})}$.

We may write xa as follows:

$$xa = y' p_m^{k_m} y, \tag{7.2}$$

where $y \in S_{(p_1,p_2,\ldots,p_{m-1})}^{(k_1,k_2,\ldots,k_{m-1})}$ and $y' \in S_{(p_1,p_2,\ldots,p_{m-1})}^{(k_1,k_2,\ldots,k_{m-1})^\star}$, such that y' does not end in p_m — all copies of p_m at this position are combined in the string $p_m^{k_m}$. For xa to be a string in $S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)^+}$, it suffices to show that it is a concatenation of strings in $S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)^+}$. As $p_m^{k_m} y \in S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$, and y' ends in a codeword from $S_{(p_1,p_2,\ldots,p_{m-1})}^{(k_1,k_2,\ldots,k_{m-1})} \backslash \{p_m\}$, this condition is satisfied by Lemma 7.1.6. We now define Δ_S as the above algorithm, which concludes the proof of Part 1.

Part 2: Δ_S is 1-to-1 and onto. From Theorem 4.2.3, we know that for any given T-Code set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$, there is exactly one $x \in S^\star$ for which $\Delta_S(x) = S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$. We also know that the above algorithm is unambiguous (this follows from the unique decodability). Hence, Δ_S is onto.

Δ_S is 1-to-1 if and only if no two distinct T-Code sets over S share any of their longest codewords. We may prove this as follows: presume that there are two distinct T-Code sets at T-augmentation levels m and n, that share the same longest codeword xa. We further presume that their T-prescriptions are in the canonical form with T-prefixes $p_1, p_2, \ldots, p_m$ and $\pi_1, \pi_2, \ldots, \pi_n$, and with T-expansion parameters $k_1, k_2, \ldots, k_m$ and $\kappa_1, \kappa_2, \ldots, \kappa_n$ respectively.

The T-augmentation levels m and n are bounded by the length $|x|$ of x, since each T-augmentation must contribute at least one alphabet symbol to x.

According to Theorem 4.2.3, there would have to be two distinct decompositions of xa and thus of x itself:

$$x = p_m^{k_m} p_{m-1}^{k_{m-1}} \ldots p_1^{k_1} \tag{7.3}$$

and

$$x = \pi_n^{\kappa_n} \pi_{n-1}^{\kappa_{n-1}} \ldots \pi_1^{\kappa_1}. \tag{7.4}$$

Since $p_1, \pi_1 \in S$, we may conclude that $p_1 = \pi_1$. Without loss of generality, we may assume that $\kappa_1 \geq k_1$. If $\kappa_1 > k_1$, p_2 must end in p_1. However, the only codeword at the second T-augmentation level that ends in p_1 is $p_1^{k_1+1}$ which implies that the first T-prescription is not canonical. Hence, $k_1 = \kappa_1$

Since xa may now be decoded over $S_{(p_1)}^{(k_1)}$, we may apply a similar argument to conclude that $p_2 = \pi_2$ and $k_2 = \kappa_2$, etc. Because there are only finitely many T-augmentation levels to consider, we may conclude that the two T-prescriptions must be identical. This contradicts our initial assumption that they are distinct.

$\square$

Definition 7.2.2 (T-Decomposition)
The algorithm introduced in the above proof is called T-decomposition.

T-decomposition has several interesting implications. From a theoretical point of view, it is remarkable that the longest codewords are unique to each T-Code set. Moreover, not only do the longest codewords act as "fingerprints" for a T-Code set, it is also possible to reconstruct the set as such from the knowledge of the alphabet and any one of the longest codewords. From a practical point of view, this permits the efficient communication of the whole decoding tree by sending just a single codeword. Inherent in this, however, is the observation that the longest codewords carry all the information about the tree itself.

Example 7.2.3 (T-Decomposition)
Let $x = 011000101010$ *and* $a = 1$*, and and let* $xa = 0110001010101$ *be the longest codeword in some T-Code set. Decoded over* $S = \{0, 1\}$*, we obtain the following codeword boundaries indicated by a dot:*

$$xa = 0.1.1.0.0.0.1.0.1.0.1.0.1.$$

from which we identify $p_1 = 0$ *and* $k_1 = 1$*. Decoded over* $S^{(1)}_{(0)}$ *we obtain*

$$xa = 01.1.00.01.01.01.01.$$

i.e., $p_2 = 01$ *and* $k_2 = 3$*. Hence, decoded over* $S^{(1,3)}_{(0,01)}$*, we get*

$$xa = 011.00.01010101.$$

such that $p_3 = 00$*,* $k_3 = 1$*, and* $p_4 = 011$ *with* $k_4 = 1$*. The reader may wish to verify that* $xa = 0110001010101$ *is indeed one of the longest codewords of* $S^{(1,3,1,1)}_{(0,01,00,011)}$.

7.3 Discussion

As we have seen, the recursive construction of the T-Code sets leads to a coding hierarchy involving all of the intermediate T-Code sets. This hierarchy is consistent with the results on T-Code codeword structure that were presented in Chapter 4. It also forms the basis for the T-Code synchronisation mechanism which will be discussed in Chapter 8.

We have seen that is possible to encode any arbitrary T-Code set in the form of one of its longest codewords. The coding hierarchy ensures the decodability of this encoding, i.e., it ensures that we can reconstruct the T-Code set from the knowledge of one of its longest codewords. The unique relationship between strings in S^+ and T-Code sets further suggests that this is a very efficient encoding.

The next chapter shows how the coding hierarchy may be used to explain T-Code self-synchronisation.

CHAPTER 8

T-Code Self-Synchronisation

One of the distinguishing properties of T-Codes is their strong tendency towards self-synchronisation. This chapter discusses concepts of general variable-length code synchronisation. It then revisits the T-Code self-synchronisation model, which is based on the hierarchical coding model introduced in Chapter 7. The treatment includes a proof of statistical self-synchronisability for T-Code sets, using a "small" decoder model. Options for faster synchronisation with more sophisticated decoders are discussed. For further treatment, two standard decoder models are proposed.

8.1 Synchronisation Concepts

The general model of our communication process was discussed in Chapter 1: information is emitted by an information source and subsequently encoded by an encoder. In our model, the encoder substitutes each source symbol by a codeword — a finite string over the channel alphabet S. We further assumed that all codewords are from a finite, prefix-free subset $C \subset S^+$ which we called our code set. Assuming that the source emits an infinite string of source symbols, we thus get a stream of symbols from S at the encoder's output.

This symbol stream is then passed through the channel which may introduce symbol errors by insertion, deletion, or corruption of individual symbols. The symbol stream is then decoded by a decoder that at most possesses the a-priori information on the code used by the encoder and, of course, any information that is received through the channel. There are no side channels in our model through which the decoder could receive additional information. We also presume that the decoder is a hard-decision decoder, i.e., recognises exactly $\#S$ different symbols on the channel.

The concept of synchronisation generally aims at establishing the correct location of codeword boundaries in the received symbol stream. While the above model is consistent with that of other authors, there is no consensus in the literature as to what it is exactly that constitutes "synchronisation". There are a number of different aspects to consider:

- It is possible to simply consider the correctness of the decoder output as an indicator for synchronisation. This can be problematic under certain circumstances, as correct output is sometimes possible even if the decoder is not operating at the intended codeword boundaries. Consider, for example, a decoder decoding the binary codeword 00 in a bitstream that consists entirely of zeros. In the case of a T-Code decoder, there is the possibility of an "error echo" (cf., e.g., [47, 17]).

- The above concept may be extended by demanding that the decoder resolve the intended codeword boundaries correctly. This concept is assumed by several authors, including Gilbert and Moore [13], Ferguson and Rabinowitz [9] and Montgomery and Abrahams [34]. To confirm synchronisation here, we effectively require an external observer with additional information on the correct position of the codeword boundaries. For example, a decoder that starts decoding at an arbitrary position in a symbol stream may be synchronised from the outset. However, in our communication model, there is no possibility for the decoder to immediately establish this fact.

- A third approach — perhaps the most conservative — is taken by Wei and Scholtz [53]:

  > "...synchronization is achieved when the receiver can indicate the first symbol of some codeword in the received symbol stream with zero probability of error."

  This approach does not require an independent observer, provided that the decoder is able to derive synchronisation information from the received symbol stream. In other words, Wei and Scholtz require that the decoder be *aware* of its current synchronisation status. In this case, however, it is assumed that the received symbol stream is sufficiently error-free such that the decoder can establish synchronisation with absolute confidence.

- Yet another approach to synchronisation is possible by only considering the decoder side: if the decoder can derive synchronisation information from the symbol stream, it is usually possible to model the synchronisation mechanism as a state machine automaton. This approach has been used by a number of authors, including Gilbert [12], Neumann [35], Maxted and Robinson [33], Perrin and Schuetzenberger [37], and Takishima, Wada, and Murakami [41]. It is also the approach that has traditionally been used for the treatment of T-Code synchronisation.

The concept of decoder "awareness" raises the question of decoder intervention: some coding schemes (especially when block codes are involved) require decoder intervention to synchronise. That is, once the decoder has sufficient information to indicate the first symbol of a codeword, it must intervene to realign itself with that codeword boundary.

An example for this is the 7Eh flag used as a synchronising token in HDLC frames on Ethernets [31]. In this case, the decoder is made aware of its lack of synchronisation and intervenes to restore correct alignment with the new codeword boundary. In comma-free coding [15] or bounded delay codes [14], a decoder operating without correct codeword boundary alignment will also become aware of this fact and realign itself accordingly.

However, even if the decoder is able to derive its own synchronisation status, explicit decoder intervention is not always required. Unlike block codes, variable-length codes are often "statistically synchronisable", i.e., self-synchronising (see Gilbert and Moore [13], Neumann [35], Ferguson and Rabinowitz [9], Maxted and Robinson [33], Takishima, Wada, and Murakami [41]). Provided that we have a sufficiently well-defined concept of what constitutes synchronisation, we may define statistically synchronisable codes along the lines of Wei and Scholtz [53]:

Definition 8.1.1 (Statistically Synchronisable Codes)
Consider a prefix-free code C, and a source encoded with codewords from C. Let $P_{synch}(q)$ denote the probability that the decoder automaton, starting in an unsynchronised state, will be in its synchronised state after receiving q symbols. If

$$\lim_{q\to\infty} P_{synch}(q) = 1, \tag{8.1}$$

then C is called **statistically synchronisable**.

Note that this definition does not explictly require that the decoder be aware of its synchronisation status. It does however require a definition of what constitutes "synchronisation".

8.2 Defining a Synchronisation Model

As discussed above, the concept of "synchronisation" is not uniquely defined. It is thus essential to specify a synchronisation model for use in this thesis. This synchronisation model assumes that

1. "synchronisation" denotes the final state of an automaton. We also demand that the state of the automaton be visible, i.e., that the decoder is aware of its current synchronisation status and can tell us whether it thinks that it is synchronised.

2. the symbol error rate is sufficiently low such that the synchronisation process takes place in an error-free part of the symbol stream. This is equivalent to the case of a decoder starting at an arbitrary position in an error-free symbol stream.

3. coincidental operation at the correct codeword boundaries is not a sufficient criterion for synchronisation.

4. the decoder is totally unsynchronised at the start of the synchronisation process.

This approach has the advantage that it is — apart from the assumption about the error-freeness of the symbol stream — conservative. In particular, the inherent decoder awareness implies that the expected synchronisation delay (ESD) obtained for a code under this model will generally exceed the ESD for a model that merely requires operation with respect to the intended codeword boundaries.

8.3 The T-Code Self-Synchronisation Mechanism

From Chapter 7, we already know that the T-Codes are hierarchical codes. We recall also that a string of codewords from a T-Code set may be regarded as a string of codewords from any of the intermediate T-Code sets, the lowest of which is the alphabet S itself. The boundary between individual codewords from an intermediate set at T-augmentation level m is referred to as an m-boundary. We also recall that the decoding algorithm presented in Chapter 4 exploits this hierarchy by decoding codewords recursively.

It is this hierarchy that not only implies statistical synchronisability for T-Codes, but also permits the decoder to determine its current synchronisation status.

As discussed before, we assume that decoder starts at an arbitrary position in a symbol stream, and that the decoder will not encounter any errors during the synchronisation process.

We may assume that the decoder is in synchronisation with respect to S at the start of the synchronisation process. In the case of a hierarchical code such as a T-Code set, we may regard the symbol stream as a sequence of codewords from each of the hierarchical levels, in our case the T-augmentation levels. Hence, the decoder may find itself synchronised with respect to some intermediate T-Code set at, say, T-augmentation level m, at some stage during the process. Since this T-Code set shares all of its codeword boundaries (m-boundaries) with all of the intermediate T-Code sets at lower levels, a decoder that has identified an m-boundary must be synchronised with respect to these sets at lower levels, too.

Definition 8.3.1 (Synchronisation Level)
Consider a T-Code decoder for a T-Code set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$*. Let* $S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$ *be the highest-level intermediate T-Code set of* $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ *for which the decoder has been able to derive synchronisation. Then the decoder is said to operate at* **synchroni-**

sation level m.

Note that the above definition explicitly includes the T-prescription being used. For T-Code sets with multiple T-prescriptions, we thus have the choice between multiple sets of synchronisation levels. This is not problematic as long as we restrict ourselves to a single one of these T-prescriptions for our discussion.

For the time being, we assume that our decoder is a recursive decoder of the type presented in Chapter 4. That is, the decoder operates over all intermediate T-Code sets simultaneously.

To derive synchronisation information, we require the decoder to perform an additional task: whenever a T-depletion codeword at level $m < n$ is returned by the decoder routine to the calling routine, the decoder should be able to:

1. compare the T-depletion codeword returned with the T-prefix p_{m+1}, and

2. if the match is negative and the decoder operates at synchronisation level m, increment the decoder's synchronisation level to $m+1$.

With the synchronisation level initially set to 0, the decoder may indeed derive synchronisation information this way because all but one of the codewords x at T-augmentation level $m+1$ are of the form

$$x = p_{m+1}^{k'_{m+1}} y, \tag{8.2}$$

where $y \in S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)} \backslash \{p_{m+1}\}$. Thus, if the codeword decoded at synchronisation level m does not match p_{m+1}, it will be the suffix of a codeword in $S_{(p_1,p_2,\ldots,p_{m+1})}^{(k_1,k_2,\ldots,k_{m+1})}$, and the decoder has found an $(m+1)$-boundary. Titchener and Hunter [51] called this the "prefix condition".

Since this comparison is performed as part of the normal decoding process, it is guaranteed that the detection of an n-boundary will terminate the decoding of the current (possibly corrupt) codeword at level n, because a negative comparison with p_n coincides with the return to the top level of the recursive decoder routine. Explicit intervention by the decoder is thus not required — it is already decoding at the correct synchronised position.

Theorem 8.3.2 (T-Codes are Statistically Synchronisable)
Consider a T-Code decoder that is receiving a semi-infinite sequence of codewords from a T-Code set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$. For $m \leq n$ let $P(x,m)$ denote the probability of occurrence of a codeword x from $S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$ in the sequence and let $P(x,n) > 0$ for all $x \in S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$. Further presume that the decoder starts decoding at synchronisation level $\Lambda = 0$ at some arbitrary point σ_0 within the sequence (i.e., not necessarily at an n-boundary). Let $P(\Lambda = n, \sigma_0 + \sigma)$ denote the probability that the decoder will operate at synchronisation level $\Lambda = n$ after reception of σ further symbols from S. Then

$$\lim_{\sigma \to \infty} P(\Lambda = n, \sigma_0 + \sigma) = 1, \tag{8.3}$$

which implies that $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ is a statistically synchronisable code.

Proof: by induction over Λ. Since the decoder starts at synchronisation level 0, all we have to show is that if the decoder operates at some synchronisation level $\Lambda = m < n$, it will eventually encounter a codeword from $S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)} \backslash \{p_{m+1}\}$ which enables it to switch to synchronisation level $\Lambda = m+1$. This is guaranteed as $P(x,m) > 0$ for all $x \in S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$, which in turn is guaranteed as $P(x,n) > 0$ for all $x \in S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$. □

Viewed externally and leaving the synchronisation information aside, our decoder cannot be distinguished from any other decoder for variable-length codes. Hence, T-Codes are statistically synchronisable irrespective of the decoder's construction.

Example 8.3.3 (Simple Synchronisation)

Consider the binary T-Code set

$$S^{(2,1,1)}_{(1,0,010)} = \{00, 0110, 0111, 10, 110, 111, 01000, 010010, 0100110, 0100111, 01010, 010110, 010111\}.$$

Further imagine that we wish to synchronise on the bit stream

$$0110101\ldots$$

Initially, we are in synchronism only with respect to $S = \{0, 1\}$*. We decode the first codeword over* S *(the end of the codeword is marked by a period):*

$$0.110101\ldots$$

Since $0 \neq p_1$*, the decoder can switch to synchronisation level 1. This means the next codeword boundary we need to look at follows a codeword from* $S^{(2)}_{(1)} = \{0, 10, 110, 111\}$*:*

$$0.110.101\ldots$$

Since $110 \neq p_2$*, the decoder can switch to synchronisation level 2. Finally, we decode over* $S^{(2,1)}_{(1,0)} = \{10, 110, 111, 00, 010, 0110, 0111\}$*:*

$$0.110.10.1\ldots$$

Since $110 \neq p_3$*, the decoder can switch to synchronisation level 3 and is now fully synchronised.*

8.3.1 Synchronising Earlier

If the decoder is allowed to be "more sophisticated", it may be able to derive synchronisation information earlier. This was discussed by Titchener and Hunter [51] who proposed a test for a "suffix condition".

For example, an m-boundary that is also an $(m+1)$-boundary could also be an $(m+2)$-boundary and so forth. A decoder that has arrived at an m-boundary could test whether it is also an $(m+1)$-boundary. If yes, the decoder could further test whether it is also an $(m+2)$-boundary, and so forth. Thus, the decoder could synchronise earlier.

The coding hierarchy of the T-Codes suggests an inductive approach to earlier synchronisation, whereby the synchronisation level is increased incrementally until the decoder can no longer conclude that the codeword boundary found is a codeword boundary at the next higher level.

Let m denote the synchronisation level that the decoder achieved before the decoding of the last codeword x such that $x \in S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$. Now assume that the decoder has been able to determine that the codeword boundary following x is an m'-boundary, where $m < m' < n$. We may ask whether there is a sufficient condition that tells us whether an m'-boundary is also an $(m'+1)$-boundary.

Alternatively, we may look for a necessary condition that an m'-boundary must satisfy to possibly *not* qualify as $(m'+1)$-boundary, i.e., a condition that blocks the decoder from immediate further synchronisation. We recall that an m'-boundary that is not an $(m'+1)$-boundary would have to follow a copy of the T-prefix $p_{m'+1}$, and hence may define a blocking condition as follows:

Definition 8.3.4 (Blocking Condition)

A decoder at synchronisation level m' that cannot — within its capability for assessing the information available to it from the communication channel — exclude *the possibility that an m'-boundary that it has reached marks the end of $p_{m'+1}$, is said to have encountered a* **blocking condition**.

In other words, a blocking condition prevents a decoder from increasing its synchronisation level and requires the decoding of at least one more codeword at level m'.

As mentioned in the above definition, the occurrence of a blocking condition is potentially dependent on the decoder's degree of sophistication. We formulate a necessary condition for the occurrence of a blocking condition as follows:

Theorem 8.3.5 (Blocking Pre-Condition)
Consider a decoder for $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ that has decoded a codeword $x \in S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$ and has established synchronisation to level m' with $m \leq m' < n$ at the codeword boundary following x. The decoder may encounter a blocking condition with respect to $p_{m'+1}$ only if

$$p_{m'+1} \succeq_{S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}} x, \tag{8.4}$$

i.e., if x is a suffix of $p_{m'+1}$ in the decoding of $p_{m'+1}$ over $S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$.

Proof: the codeword boundaries preceding and succeeding x are adjacent m-boundaries. A blocking condition can only occur if x could be marking the end of $p_{m'+1}$.

However, the boundary preceding $p_{m'+1}$ would have to be an m'-boundary and hence by hierarchical inference an m-boundary. As $p_{m'+1} \in S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)+}$, all m-boundaries within $p_{m'+1}$ must also correspond to m-boundaries in the received symbol stream. Hence, x must be a suffix of $p_{m'+1}$ in the decoding of $p_{m'+1}$ over $S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$. □

Note that the blocking pre-condition proposed here is stricter than the "suffix condition" proposed by Titchener and Hunter [51],

$$p_{m'+1} \succeq_S x, \tag{8.5}$$

which only requires x to be a suffix of $p_{m'+1}$ over S. The blocking pre-condition introduced here is also much easier to implement in a recursive decoder than the "suffix condition" — all that is required is a comparison between the T-depletion code for x and the relevant entries in the T-depletion code for $p_{m'+1}$.

If the blocking pre-condition for synchronisation level $m'+1$ is not satisfied, the decoder has obviously reached that synchronisation level. This tends to be the case in most situations. Of more interest, however, are cases where the blocking pre-condition *is* satisfied. A decoder may now utilise all available information received since the start of the synchronisation process in order to determine whether there is actually a blocking condition or not.

Assume that the decoder encounters a blocking pre-condition following the reception of a string $d \in S^+$ after the start of the synchronisation process. This marks a blocking condition under the following three mutually exclusive and complementary circumstances:

Case 1: where $x = p_{m'+1}$. (This is the trivial case.)

Case 2: where $p_{m'+1} \succ_S d \succeq_S x$, and d is "boundary compatible" with $p_{m'+1}$. That is, the symbol stream d received since the start of the decoding is a suffix (over S) of $p_{m'+1}$, and $p_{m'+1}$ is longer than d and x. Furthermore, known codeword boundaries in d up to level m must be "compatible" with the corresponding boundaries in $p_{m'+1}$: each symbol boundary in $p_{m'+1}$ is a j-boundary but not a $(j+1)$-boundary at some known level $j < m'$. Similarly, the boundaries immediately preceding and following $p_{m'+1}$ at at least m'-boundaries. The corresponding boundaries in d must be "compatible", i.e., it must be possible that they are boundaries at the same levels. See subsection 8.3.2 for a more detailed explanation and an example. In this case, symbols transmitted before the start of the synchronisation process could complement the received symbols to yield $p_{m'+1}$.

Case 3: where $d \succ_S p_{m'+1} \succ_{S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)}} x$, i.e., $p_{m'+1}$ is a suffix (over S) of d, and $p_{m'+1}$ and d are "boundary compatible".

A decoder that can "remember" d (and thus the confirmed codeword boundaries for the various synchronisation levels attained) may now be able to establish whether any of these three cases is satisfied, or whether the blocking pre-condition raised a "false alarm". If a decoder lacks this memory and logic, it must err on the side of caution in which case the blocking pre-condition becomes a full blocking condition. Hence we see once more that the delay in the synchronisation process may depend to an extent on the actual decoder implementation.

It is thus essential to nominate which decoder model is being assumed if one wishes to make statements on parameters such as the expected synchronisation delay of a T-Code set.

In this thesis, the treatment is generally restricted to two models:

a minimal decoder. This model assumes that the decoder is merely able to compare a codeword $x \in S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)}$ with p_{m+1}. The outcome of this comparison is a Boolean value which may be used to increment the synchronisation level. In the context of the work by Titchener and Hunter, this corresponds to a decoder that only tests for the "prefix condition". This is the minimum functionality required in a decoder in order to be able to derive synchronisation information. We have already used this model above in our proof of statistical synchronisability.

a maximal decoder. Here, we assume that the decoder is able to utilise all information contained in d and in the prescription for $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ in order to determine its current synchronisation level.

The following example demonstrates the advantages of the maximal decoder model:

Example 8.3.6 (Early Synchronisation)

Consider once more the T-Code set and the bit stream from Example 8.3.3. Initially, we are in synchronism only with respect to $S = \{0, 1\}$*. We decode the first codeword over* S *(the end of the codeword is marked by a period):*

$$0.110101\ldots$$

Since $0 \neq p_1$*, the decoder can switch to synchronisation level 1. However, we do get a blocking condition with respect to level 2 because* $0 = p_2$*. This means the next codeword boundary we need to look at still follows a codeword from* $S_{(1)}^{(2)} = \{0, 10, 110, 111\}$*:*

$$0.110.101\ldots$$

Since $110 \neq p_2$*, the decoder can now switch to synchronisation level 2. Also,* 110 *is not a suffix of* $p_3 = 010$ *over* $S_{(1)}^{(2)}$*, so the decoder can switch to synchronisation level 3 and is fully synchronised, two bits "earlier" than in the previous example.*

8.3.2 Boundary Compatibility

The "boundary compatibility" criterion is based on the boundary information that the decoder derives as it synchronises into d. For each symbol boundary in d, the decoder is able to derive maximum and minimum values for the level j for which the boundary is a j-boundary but not a $(j+1)$-boundary. The (known) boundaries within $p_{m'+1}$ have to be compatible with this, i.e., they must fall within the range established by the decoder.

Example 8.3.7 (Boundary Compatibility)
Consider the binary T-Code set $S^{(2,1,1,1)}_{(0,01,1,001)}$ *and the strings* 101001 *and* 10101001*, which mark the first bits in two different synchronisation scenarios.*

Synchronising into the first string, the decoder is able to establish the following minimum/maximum values for j *at each symbol boundary, denoted here by subscripts and superscripts:*

$$ {}^4_0 1^4_1 0^0_0 1^4_1 0^0_0 0^0_0 1^4_? $$

Before the first bit, the decoder is completely unsynchronised, i.e., the boundary preceding it could be anything between a 0*-boundary and a* 4*-boundary. After the first bit, the decoder is able to tell that it has reached a* 1*-boundary, which may also be a boundary at a higher level. At this point, we have a blocking pre-condition with respect to* $p_2 = 01$.

Closer inspection reveals that the boundaries (denoted by subscripts) in and around p_2 *are:* ${}_1 0_0 1_1$. *Since the first corresponding boundary is not contained in our above string, we must establish whether the* 0*-boundary in the middle of* p_2 *falls within the range* 0 *to* 4*, and whether the* 1*-boundary at the end falls within the range* 1 *to* 4*. They both do, and hence we have a blocking condition.*

We then decode the next codeword at level 1, which is $01 = p_2$. *In our string, its internal boundary is at least and at most a* 0*-boundary, whereas the boundary at its end is at least a* 1*-boundary and at most a* 4*-boundary. Since the codeword causes a blocking condition, the decoder remains at level 1.*

The next codeword is 001*, which permits a transition to level 3, but it causes a blocking pre-condition with respect to level 4. The internal boundaries within* 001 *are both* 0*-boundaries. The boundaries in and around* p_4 *are* ${}_3 1_2 0_0 1_1 0_0 0_0 1_3$. *All of these fall within the range established above, which means there is a blocking condition. Thus, the decoder remains at level 3 at the end of the string, i.e., we may replace the question mark with a 3.*

Now consider the second string:

$$ {}_0^4 1_1^4 0_0^0 1_1^4 0_0^0 1_1^4 0_0^0 0_0^0 1_?^4 $$

It differs from the first string only in that there is an extra codeword $p_2 = 01$ *decoded at level 1, and we once again get a blocking pre-condition when we decode the* 001 *codeword at the end. Now, however, the* 3*-boundary at the beginning of* p_4 *is incompatible with the boundary after the second bit in our string, which is at least and at most a* 0*-boundary. Hence, there is no blocking condition, and the decoder can advance to synchronisation level 4.*

8.4 Generalised vs. simple T-Codes

The T-Code self-synchronisation mechanism as discovered and explained by Titchener and Hunter does not have to be changed when generalised rather than simple T-Codes are involved [50] as the hierarchical model on which the self-synchronisation is based remains the same. A blocking condition does not depend on which T-prefix string causes it, and which position that string has in the actual codeword at the higher level.

Furthermore, if a maximal decoder is used, the self-synchronisation mechanism does not change with the type of T-prescription used. In this case, the overall synchronisation behaviour of a T-Code set remains the same under expansion and contraction of the associated T-prescription (cf. Chapter 3). If for some T-augmentation level m, we have $p_m^{k_m+1} = p_{m+1}$, the absence of a blocking condition with respect to p_m implies the absence of a blocking condition with respect to p_{m+1}. Hence, a decoder that operates at level i will never have to operate at level $m+1$. Conversely, a decoder transition from a synchronisation level below m to level $m+1$ will always be prevented by a blocking condition with respect to p_m.

8.5 Discussion

Applying the strict hierarchical coding model presented in Chapter 7 to explain the T-Code self-synchronisation has several advantages. Firstly, it permits the rather simple proof of statistical synchronisability given in this chapter. It does not explicitly require the results by Gilbert and Moore [13] — an alternative would have been to show that the greatest common divisor of codeword lengths in any T-Code set is 1.

Secondly, the hierarchical structure of the T-Codes offers the opportunity to "keep an eye" on the synchronisation process — we can derive synchronisation information straight from the decoder. This enables us to take a more conservative approach towards concepts such as statistical synchronisability and expected synchronisation delay. Many other variable-length codes may be self-synchronising, too, but have no obvious way of letting the decoder derive information on its synchronisation status.

This chapter has used the hierarchical coding model to suggest the stricter criterion for the "suffix condition", implemented here in the form of the blocking condition criteria and the blocking pre-condition.

The amount of synchronisation information that may be utilised depends on two factors: firstly, on the information present in the symbol stream; secondly, on the decoder's sophistication, i.e., the extent to which the decoder is able to utilise the information contained in the incoming symbol stream. Hence, it is prudent to define the type of decoder that is assumed when discussing T-Code synchronisation-related issues. The method of calculation of the T-Code synchronisation delays in the next chapter assumes a maximal decoder.

It is further interesting to consider the work of authors who have suggested methods for modifying Huffman codes to improve their synchronisation properties. Perrin and Schuetzenberger [37], for example, give an example of a "synchronising prefix code", which is identical to the binary T-Code set $S_{(0,1)}^{(1,1)}$. The same code set is proposed by Takishima, Wada, and Murakami [41]. They also conclude that two other codes (sets C_{11} and C_{13} in their paper), which in our notation are the T-Code sets $S_{(0)}^{(3)}$ and $S_{(0,01)}^{(1,1)}$, have good synchronisation properties.

CHAPTER 9

Calculating the Expected Synchronisation Delay

The T-Code self-synchronisation mechanism permits the calculation of an expected synchronisation delay (ESD), i.e., an expectation value for the number of symbols that an unsynchronised decoder has to receive before it can confirm that it is synchronised. This chapter presents a new approach to calculating the ESD. The computational complexity of the method presented is also discussed.

9.1 Modelling the T-Code Self-Synchronisation Mechanism as a Discrete Markov Chain

The expected synchronisation delay (ESD)[1] is the number of alphabet symbols that a totally unsynchronised decoder needs to receive before it can determine that it is

[1]the term *expected synchronisation delay* is that used by Titchener and Hunter. Another expression that essentially describes the same quantity (subject to the definition of *synchronisation* that is used) is *synchronization acquisition delay* [16]. Higgie [26] uses the term *average synchronisation delay (ASD)*, but also mentions ESD and *average re-synchronisation delay.*

synchronised. The problem of ESD calculation for T-Code sets was first discussed by Titchener and Hunter [51], and this discussion initially follows in their footsteps.

The T-Code self-synchronisation mechanism, as discussed in the previous chapter, may be modelled as a discrete time Markov chain. For this purpose, we require a finite set of discrete states, events that bring about transitions between these states, and transition probabilities associated with these events.

In our case, the synchronisation levels represent the states. During the synchronisation process, transitions between synchronisation levels take place and eventually terminate the synchronisation process at the highest possible synchronisation level. We now define the events that cause transitions, and also their associated probabilities.

Up to this point, the ESD calculation presented here follows the model presented by Titchener and Hunter [51]. However, Titchener and Hunter took a symbol-oriented approach: that is, the principal event that causes transitions is the reception of an individual symbol. As a result, additional states are required at most synchronisation levels. The number of additional states that are required depends on the T-prefixes chosen for each level and grows as the T-prefixes become more complex with an increase in T-augmentation level. Titchener and Hunter give an example of a T-Code set at T-augmentation level 11 that requires a total of sixty states [51].

This increases the total number of possible transitions, which in turn imposes a large degree of computational complexity on the ESD calculation — this is noticeable especially for large T-Code sets [46].

The approach presented here is codeword-oriented rather than symbol-oriented and, by comparison, is based on $n+1$ states for a T-Code set, i.e., 12 for the above example. It was proposed by the author together with Mark Titchener in [21].

A further abridged description appeared in [22]. This method is based on the synchronisation mechanism discussed in the last chapter, i.e., assuming a maximal decoder model, and was chosen primarily for completeness. Less sophisticated decoders may be treated similarly.

A codeword-oriented approach has the advantage of being less complex. In addition, it is possible to accommodate widely varying codeword probabilities.

9.2 Calculating the Expected Synchronisation Delay (ESD)

The expected synchronisation delay (ESD) is defined as the average number of symbols in S that the decoder has to receive before it can conclude that it has achieved synchronisation with respect to the highest-level set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$, and is a function of $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$. Without loss of generality, we will use assume that the T-prefixes and T-expansion parameters for $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ constitute a canonical T-prescription (if several T-prescriptions for $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ exist).

For simplicity, however, we will mostly drop the explicit reference to $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ in our notation, as it is unambiguous that all quantities to be discussed are functions of $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ or its intermediate T-Code sets (and hence expressible as functions of $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$) and its associated source probabilities.

We calculate the ESD for $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$, $\mathrm{ESD}(S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)})$, as the sum over the expected delays at the intermediate synchronisation levels:

$$\mathrm{ESD}(S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}) = \sum_{m=0}^{n-1} P_v(m)\,\tau(m). \tag{9.1}$$

In this equation, $P_v(m)$ is the **visitation probability** , the probability that the decoder will ever decode a codeword while being at synchronisation level m during the process of synchronisation. This is equivalent to saying that $P_v(m)$ is the á priori probability of encountering a blocking condition with respect to p_{m+1} during the synchronisation process.

The **expected level synchronisation delay** $\tau(m)$ is the average number of symbols from S decoded while the decoder operates at level m.

We first show how to calculate the visitation probability $P_v(m)$ and then calculate $\tau(m)$ in Section 9.2.2.

9.2.1 Calculating the Visitation Probability $P_v(m)$

In our synchronisation model, the decoder always starts at synchronisation level $m = 0$, with visitation probability $P_v(0) = 1$. For $m > 0$, the decoder may "skip" one or more of the levels if it decodes a codeword at a level below m that does not cause a blocking condition with respect to a T-prefix below p_{m+2}. In such cases $P_v(m) \leq 1$ (of course, if we used a very simple decoder model, the decoder could only advance by one synchronisation level at at time, and $P_v(m) = 1$ for all m).

The visitation probability $P_v(m)$ may be expressed as a recursive sum[2]:

$$P_v(m) = \sum_{i=0}^{m-1} T_{im} P_v(i), \tag{9.2}$$

where T_{im} denotes[3] the probability that a transition from level i will carry the decoder through to level $m > i$.

No transition from any level i carries the decoder beyond synchronisation level n, thus we have the trivial identity

$$\sum_{m=0}^{n} T_{im} = 1, \tag{9.3}$$

[2]Equation (9.2) may also be expressed as an eigenvalue problem if the T_{im} are redefined to include "transitions" from a particular level to itself.

[3]Note that the T_{im} we use here should not be confused with the T_{im} used in [51, 46]: in our case, the indices i and m refer to synchronisation levels, not base alphabet symbols.

where we define $T_{im} = 0$ for $m \leq i$. The latter is a consequence of our assumption that there will be no further symbol errors during the course of synchronisation. [4]

Calculating T_{im}:

The T_{im} may be determined as follows. The receipt of a codeword $x \in S_{(p_1,p_2,\ldots,p_i)}^{(k_1,k_2,\ldots,k_i)} \backslash p_{i+1}$ by the decoder may precipitate a transition to various levels $m > i$, or may leave the decoder in $S_{(p_1,p_2,\ldots,p_i)}^{(k_1,k_2,\ldots,k_i)}$, depending on x and in some cases on symbols received prior to x. If x precipitates a transition (under at least some circumstances), it is a **transition codeword**.

We define $P(x, i, m)$ to be the probability that the receipt of x in at synchronisation level i precipitates a transition to $S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$. Any codeword $x \neq p_{i+1}$ in $S_{(p_1,p_2,\ldots,p_i)}^{(k_1,k_2,\ldots,k_i)}$ causes a transition to a higher level set, and we get

$$T_{im} = \sum_{x \in S_{(p_1,p_2,\ldots,p_i)}^{(k_1,k_2,\ldots,k_i)} \backslash \{p_{i+1}\}} \frac{P(x,i)P(x,i,m)}{1 - P(p_{i+1}, i)} \tag{9.4}$$

where $P(x, i)$ is the probability of decoding the codeword x in a decoding over $S_{(p_1,p_2,\ldots,p_i)}^{(k_1,k_2,\ldots,k_i)}$. The denominator $1 - P(p_{i+1}, i)$ reflects the probability of decoding the T-prefix p_{i+1} as a next codeword, which has the effect of delaying the transition to a higher set. It is required here for normalisation purposes.

If $p_{m+1} \not\preceq_{S_{(p_1,p_2,\ldots,p_i)}^{(k_1,k_2,\ldots,k_i)}} x$ (Theorem (8.3.5)), there is no blocking pre-condition and $P(x, i, m) = 0$. If $p_{m+1} \not\preceq_{S_{(p_1,p_2,\ldots,p_i)}^{(k_1,k_2,\ldots,k_i)}} x$ for all $i \leq m < n$, we have $P(x, i, m) = 0$ and $P(x, i, n) = 1$ respectively. In practice, this is the case for most combinations of x, i and m, and corresponds to a reduction in the computational effort (see

[4] Note that $P_v(n)$ and T_{in} have no relevance to the calculation of the ESD (see Equation (9.1)). However, as T-Codes are statistically synchronisable, we have $P_v(n) = 1$, which may be used in practice to check the feasibility of the $P_v(m)$ and T_{im} calculated.

section 9.4.3). We now calculate $P(x,i,m)$ for those x for which a blocking pre-condition occurs.

Accounting for Blocking Conditions

Possible blocking conditions have to be investigated whenever a blocking pre-condition is encountered, i.e., when for some level $i < m$

$$p_{m+1} \succeq_{S^{(k_1,k_2,\ldots,k_i)}_{(p_1,p_2,\ldots,p_i)}} x,$$

but $p_{m'+1} \not\succeq_{S^{(k_1,k_2,\ldots,k_i)}_{(p_1,p_2,\ldots,p_i)}} x$ for all m' where $i \leq m' < m$.

In this case, the probability $P(x,i,m)$ for a transition from level i to level m, upon receipt of x, equals the probability that a blocking condition with respect to p_{m+1} occurs. In cases where the blocking pre-condition $p_{m+1} \succeq_{S^{(k_1,k_2,\ldots,k_i)}_{(p_1,p_2,\ldots,p_i)}} s$ is not satisfied, this probability is zero.

We know which x, i, and m satisfy the blocking pre-condition because all of the intermediate code sets are known. However, we do not know exactly which — or even how many — symbols precede x in an actual decoding situation.

Determining the probability of a blocking condition following the receipt of x thus requires determining the possible symbol combinations that may precede x *and* would cause a blocking condition. The probability of a blocking condition is then simply the probability that the decoder encounters one of these situations.

Where the blocking pre-condition holds, the decoder may encounter one of three distinct cases (see Section 8.3.1):

Case 1: $x = p_{m+1}$. Here, a blocking condition exists irrespective of any symbols received prior to x, and $P(x,i,m) = 1$.

Case 2: $p_{m+1} \succ_S d \succ_S x$ and boundary compatibility between p_{m+1} and d, corresponding to the situation where decoding has started *after* the beginning of a string that could be p_{m+1}.

Case 3: $d \succeq_S p_{m+1} \succ_{S^{(k_1,k_2,\ldots,k_i)}_{(p_1,p_2,\ldots,p_i)}} x$ and boundary compatibility between p_{m+1} and d. This accounts for the case where decoding has started *before* the beginning of a string that could be p_{m+1}.

While the first case is trivial, calculating $P(x, i, m)$ for the last two cases requires more effort as we need to account for all possible strings z that could precede x in a real decoding situation, such that $d = zx$. We know that the decoder is at synchronisation level i at the boundary between z and x. Hence, z must be a **synchronising string** for level i, i.e., it must synchronise the decoder exactly to level i but not further.

Synchronising strings: According to [49], a string that synchronises a decoder to level i is of the form

$$x_s(i) = p_1^{m_1} p_2^{m_2} \ldots p_i^{m_i} x', \tag{9.5}$$

where $m_1, m_2, \ldots, m_i \in \mathbb{N}$ and $x' \in \left\{ S^{(k_1,k_2,\ldots,k_{i-1})}_{(p_1,p_2,\ldots,p_{i-1})} \cup \{\lambda\} \right\} \backslash \{p_i\}$. Titchener's equation implies that only the decoding of T-prefixes causes blocking conditions. For some T-Code sets, this is indeed the case, namely those sets for which the T-prefixes are chosen such that blocking conditions can only occur if $x = p_{m+1}$.

However, this is not always the case. In the very case that we are discussing at the moment, we suspect that a transition codeword x decoded at level m could be the end of p_{m+1}. Note that x does not have to be a T-prefix at any T-augmentation level for this to occur. If we cannot rule out that m is the end of a copy of p_{m+1} in a practical decoding situation, we have a blocking condition, and x gets inserted into the synchronising string. If we continue decoding and the decoder eventually synchronises to a level beyond $m+1$, then x becomes part (but not suffix of) the particular instance of the synchronising string for that level. Unless x is a T-prefix of the set at a lower level, the resulting synchronising string does not satisfy the above equation.

A necessary and sufficient form of Titchener's equation may be obtained by modifying it such that transitions during the synchronisation process are attributed explicitly to the transition codewords that cause them.

Let $x(i_j, i_{j'})$ be a codeword in $S_{(p_1,p_2,\ldots,p_{i_j})}^{(k_1,k_2,\ldots,k_{i_j})}$ that causes a transition between levels i_j and $i_{j'}$. For $i_1 < \ldots < i_j < \ldots < i$, we get:

$$x_s(i) = p_1^{m_0} x(0, i_1) \ldots p_{i_{j-1}}^{m_{j-1}} x(i_{j-1}, i_j) p_{i_j}^{m_j} x(i_j, i). \tag{9.6}$$

where $m_0, m_1, \ldots, m_j \in \mathbb{N}$ Note that the $x(i_j, i_{j'})$ may depend to an extent on the substring to the left of their position in $x_s(i)$. $x(i_j, i_{j'}) \neq p_{i_{j'}+1}$ are subject to blocking conditions that depend on previous symbols. This needs to be taken into account.

The strings described by the previous equation include only such strings that *just* synchronise the decoder to level i. That is, no proper prefix of an $x_s(i)$ that satisfies Equation (9.6) is a synchronising string. To include all potential candidates for blocking strings, we must thus allow for the possibility that a run of T-prefixes p_{i+1} has held the decoder at level i. That is, we are interested in all strings that synchronise the decoder *to level i but not further.*

Allowing for this, Equation (9.6) becomes[5]:

$$x_s(i) = p_1^{m_0} x(0, i_1) \ldots p_{i_{j-1}+1}^{m_{j-1}} x(i_{j-1}, i_j) p_{i_j+1}^{m_j} x(i_j, i) p_{i+1}^{m_{j+1}}. \tag{9.7}$$

We now attempt a "naive" definition of a probability of occurrence for x_s as the

[5] we also keep in mind that λ is a "string" that synchronises a decoder to level 0

product of the probability of occurrence of its components:

$$\begin{aligned} P(x_s(i)) &= P(p_1,0)^{m_0} P(x(0,i_1),0) \cdot \ldots \\ &\quad \ldots \cdot P(p_{i_{j-1}+1}, i_{j-1})^{m_{j-1}} P(x(i_{j-1},i_j), i_{j-1}) P(p_{i_j+1}, i_j)^{m_j} \\ &\quad \cdot P(x(i_j,i), i_j) P(p_{i+1}, i)^{m_{j+1}}. \end{aligned} \tag{9.8}$$

The " problem" with this definition is that if we added the $P(x_s(i))$ for all synchronising strings $x_s(i)$, the sum would generally not add up to one, and thus $P(x_s(i))$ does not constitute a "meaningful" probability. This has two reasons:

- some synchronising strings in the form of Equation (9.7) are prefixes of others. E.g., we can take any $x_s(i)$ and append a copy of p_{i+1}, and thus obtain a another synchronising string. Thus, the $P(x_s(i))$ do not refer to mutually exclusive events.
- not all combinations of codewords at ascending levels yield synchronising strings.

The "problem", however, is merely a lack of normalisation:

- consider that in our scenario, all synchronising strings are followed by x, a codeword other than p_{i+1}. If we account for x, then no synchronising string followed by x is a prefix of another synchronising string followed by x. Hence, the terms $P(x_s(i))P(x,i)$ describe mutually exclusive events. However, as $P(x,i)$ is a common factor in $P(x_s(i))P(x,i)$ for all synchronising strings $x_s(i)$, it is merely a normalisation factor which we may include if required or omit if not.
- the second point of concern was that $P(x_s(i))$ is not conditional on $x_s(i)$ being a synchronising string. Making the probability conditional is also only a matter of a multiplying all $P(x_s(i))$ by a common factor, namely $\left[\sum_{x_s(i)} P(x_s(i))\right]^{-1}$.

Hence, we could normalise the $P(x_s(i))$ if we wanted to. However, we shall see shortly that neither normalisation factor is required.

Blocking strings: Synchronising strings z that cause a blocking condition are called **blocking strings**. Thus, blocking strings are a subset of all synchronising strings, and $P(x, i, m)$ is the probability that we encounter a blocking string rather than another synchronising string in an actual decoding situation.

Calculating $P(x, i, m)$: $P(x, i, m)$ is the probability that we have a blocking string given that we have a synchronising string. Hence, $P(x, i, m)$ is given by the cumulative probability of occurrence of all blocking strings $x_{sb}(i)$ divided by the cumulative probability of occurrence of all synchronising strings $x_s(i)$:

$$P(x, i, m) = \frac{\sum_{x_{sb}(i)} P(x_{sb}(i))}{\sum_{x_s(i)} P(x_s(i))}. \tag{9.9}$$

Our "normalisation problem" disappears at this point: the normalisation factors that we would require are the same for both numerator and denominator and hence cancel out.

Still, this equation poses a problem: there are infinitely many synchronising strings and generally also infinitely many blocking strings - so how do we calculate the sums? The problem is caused by the unbounded runs of T-prefixes in the synchronising strings, i.e., $m_0, \ldots, m_j$ are unbounded.

However, the problem is minor. Consider synchronising strings that are identical except for a run of $m_{j'}$ T-prefixes $p_{i_{j'}+1}$. For these strings, $P(x_s(i))$ only differs by a factor of $P(p_{i_{j'}+1}, i_{j'})^{m_{j'}}$, where $m_{j'}$ can generally take any value between zero and infinity. When taking the sum over $P(x_s(i))$ for these strings in Equation (9.9), we can factor the common part out and are left with a term that can be expressed as a geometric series:

$$\sum_{m_{j'}=0}^{\infty} P(p_{i_{j'}+1}, i_{j'})^{m_{j'}} = \frac{1}{1 - P(p_{i_{j'}+1}, i_{j'})}. \tag{9.10}$$

We can only do this if the actual number of T-prefixes in a run has no influence on whether the synchronising string is a blocking string or not. There is also the possibility of blocking conditions at lower levels caused by $x(i_j, i_{j'})$ that satisfy a blocking pre-condition. In the latter case, this could imply that for certain $p_{i_{j'}+1}$ and $m_{j'}$, the resulting string is not a synchronising string.

However, the two extra criteria that distinguish a blocking string from a mere synchronising string only apply to a finite portion of the synchronising string in question: the part where it "overlaps" what could be p_{m+1}. This finite substring may always be covered by finite runs of T-prefixes. The presence or absence of blocking conditions at lower levels are also brought about by finite substrings. In such cases we may thus treat an infinite run of T-prefixes as a combination of a finite run and an infinite one. Hence, we are able to group all infinte runs, and the number of relevant cases that we need to evaluate is indeed finite.

How can one find all relevant synchronising and blocking strings? The algorithm suggested here is a recursive search. Starting at level i and with $z = \lambda$, we recursively add T-prefixes and $x(i_j, i_{j'})$ to the left of z until we obtain synchronising strings.

When the length of the recursively constructed string exceeds $|p_{m+1}| - |x|$, we only add T-prefixes to the left if there is an unresolved blocking pre-condition due to some $x(i_j, i_{j'})$ that was added before. If not, we account for them collectively in the sum of probabilities.

All synchronising strings found are entered into a list, which also includes their unnormalised probability of occurrence (with the geometric series factor for T-prefix runs included). Each string is tested for the two blocking string criteria, and the sums in Equation (9.9) are taken, which yields $P(x,i,m)$ as desired.

A Feasibility Check

The treatment of blocking conditions is non-trivial. It may therefore be helpful to know that there is a way of checking the feasibility of a set of $P(x,i,m)$ that one has calculated: since no transition beyond the top level set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ is possible, we get the identity

$$\sum_{l=i}^{n} P(x,i,l) = 1, \tag{9.11}$$

where $P(x,i,i)$ denotes the probability of the decoder remaining at level i. For $x \neq p_{i+1}$, we have $P(x,i,i) = 0$, otherwise $P(x,i,m) = 0$ for $m > i$. For the "interesting" cases of multiple blocking conditions, we may hence check if

$$\sum_{l=i+1}^{n} P(x,i,l) = 1. \tag{9.12}$$

This also implies that, should we wish to determine the ESD of a set $S_{(p_1,p_2,\ldots,p_{n'})}^{(k_1,k_2,\ldots,k_{n'})}$ derived from $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ by further T-augmentations, all $P(x,i,n)$ and thus all T_{in} and $P_v(n)$ have to be recalculated.

9.2.2 Calculating $\tau(m)$

Once the visitation probabilities $P_v(m)$ have been determined, we calculate the expected level synchronisation delay $\tau(m)$ for each T-augmentation level $m < n$. This delay comprises two parts:

1. the **prefix delay**. This is the average delay introduced by the decoding of m consecutive T-prefixes p_{m+1} over $S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$ which prevents the decoder from making a transition to a higher set. The T-prefix delay is thus the sum of the lengths of the T-prefix strings each weighted by the probability of decoding

it m times:

$$\tau_p(m) = \sum_{m=1}^{\infty} |p_{m+1}| P(p_{m+1}, m)^m = |p_{m+1}| \frac{P(p_{m+1}, m)}{1 - P(p_{m+1}, m)}, \tag{9.13}$$

where we have used the identity $\frac{1}{1-\alpha} = \sum_{m=0}^{\infty} \alpha^m$.

2. the **suffix delay** arising from the decoding of a transition codeword, i.e., the average length of the transition codewords at level m. To obtain the suffix delay, we need to look at each individual codeword x in $S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)}$ except for the T-prefix p_{m+1}, and multiply its length by $P(x,m)/(1 - P(p_{m+1}, m)$. The suffix delay is thus given by

$$\tau_s(m) = \sum_{x \in S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)} \setminus p_{m+1}} |s| \frac{P(x, m)}{1 - P(p_{m+1}, m)}. \tag{9.14}$$

Adding the two parts gives the expected level synchronisation delay

$$\tau(m) = \tau_p(m) + \tau_s(m) = \sum_{x \in S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)}} |x| \frac{P(x, m)}{1 - P(p_{m+1}, m)}. \tag{9.15}$$

Taking the sum over all levels we get, according to Equation (9.1),

$$\mathrm{ESD}(S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}) = \sum_{m=0}^{n-1} P_v(m) \sum_{x \in S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)}} |x| \frac{P(x, m)}{1 - P(p_{m+1}, m)}. \tag{9.16}$$

In the case of a perfectly matched source, i.e., when $P(x,m) = \#S^{-|x|}$, we get:

$$\mathrm{ESD}(S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}) = \sum_{m=0}^{n-1} P_v(m) \sum_{x \in S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)}} \frac{|x|}{[\#S]^{|x|} \left(1 - \frac{1}{\#S^{|p_{m+1}|}}\right)}. \tag{9.17}$$

The next section illustrates this by way of two examples.

9.3 Two Examples: Calculating the ESD of Binary T-Code Sets

The two following examples show how the ESD for the binary sets $S^{(1,1,1,1,1)}_{(0,1,00,01,11)}$ and $S^{(2,1,1)}_{(0,001,01)}$ may be calculated using the above approach. $\mathrm{ESD}(S^{(1,1,1,1,1)}_{(0,1,00,01,11)})$ has been calculated by Titchener in [46] on page B4 to be 10.82292 bits[6] and may be used for comparison. Due to its relative simplicity, this example set allows us to emphasise some of the essential aspects of the ESD calculation. In contrast, the second example, $S^{(2,1,1)}_{(0,001,01)}$ introduces the treatment of non-trivial blocking conditions.

9.3.1 ESD($S^{(1,1,1,1,1)}_{(0,1,00,01,11)}$)

Consider the alphabet $S = \{0, 1\}$ and the simple T-Code set $S^{(1,1,1,1,1)}_{(0,1,00,01,11)}$ derived from S by simple T-augmentation with T-prefixes 0,1,00,01, and 11. $S^{(1,1,1,1,1)}_{(0,1,00,01,11)}$ and all its intermediate sets are listed in Table 9.1 on page 127.

We will now show how $P_v(m)$ and $\tau(m)$ may be calculated for each level $m = 0, \ldots, 4$. As $S = \{0, 1\}$ is a binary alphabet, $\#S = 2$. We also assume that the codewords in $S^{(1,1,1,1,1)}_{(0,1,00,01,11)}$ and its intermediate sets are optimally matched to the source, i.e., that $P(x, m) = 2^{-|x|}$.

- T-augmentation/synchronisation level 0. As mentioned in the previously, all decoding starts at this level, and $P_v(0) = 1$ by default. The $\tau(0)$ is given by:

$$\tau(0) = 1 \times \frac{1}{2^1\left(1 - \frac{1}{2^1}\right)} + 1 \times \frac{1}{2^1\left(1 - \frac{1}{2^1}\right)} = 2. \tag{9.18}$$

  In preparation for the next levels, we calculate the transition probabilities T_{0m}. As can be seen from the table, only the codeword 1 causes a transition, which encounters a blocking condition as $1 = p_2$. Thus $P(1, 0, 1) = 1$, and $P(x, 0, m) = 0$ for all other choices of x and m. Equation (9.4) yields

$$T_{01} = \sum_{x \in S^{(k_1,k_2,\ldots,k_0)}_{(p_1,p_2,\ldots,p_0)} \setminus p_1} \frac{P(x,0)P(x,0,1)}{1 - P(p_1,0)} = 1. \tag{9.19}$$

- T-augmentation/synchronisation level 1. We are now dealing with the inter-

[6] For most practical applications, a lesser degree of precision should be sufficient.

Intermediate T-Code sets of $S^{(1,1,1,1,1)}_{(0,1,00,01,11)}$ at T-augmentation levels 0-5										
$m=0$		$m=1$		$m=2$		$m=3$		$m=4$		$m=5$
S		$S^{(1)}_{(0)}$		$S^{(1,1)}_{(0,1)}$		$S^{(1,1,1)}_{(0,1,00)}$		$S^{(1,1,1,1)}_{(0,1,00,01)}$		$S^{(1,1,1,1,1)}_{(0,1,00,01,11)}$
0										
1	→	1								
		00	→	00						
		01	→	01	→	01				
				11	→	11	→	11		
				100	→	100	→	100	→	100
				101	→	101	→	101	→	101
						0000	→	0000	→	0000
						0001	→	0001	→	0001
						0011	→	0011	→	0011
						00100	→	00100	→	00100
						00101	→	00101	→	00101
								0101	→	0101
								0111	→	0111
								01100	→	01100
								01101	→	01101
								010000	→	010000
								010001	→	010001
								010011	→	010011
								0100100	→	0100100
								0100101	→	0100101
										1111
										⋮
										110100101

Table 9.1. Listing of the T-Code sets S, $S^{(1)}_{(0)}$, $S^{(1,1)}_{(0,1)}$, $S^{(1,1,1)}_{(0,1,00)}$, $S^{(1,1,1,1)}_{(0,1,00,01)}$, and $S^{(1,1,1,1,1)}_{(0,1,00,01,11)}$ based on the T-augmentation of $S = A = \{0,1\}$. The arrows indicate transitions during the synchronisation process. A transition in a particular line carries the decoder to the set pointed at by the last arrow in the line.

mediate set $S^{(k_1)}_{(p_1)} = S^{(1)}_{(0)}$. From Equation (9.2),

$$P_v(1) = \sum_{i=0}^{0} T_{i1} P_v(i) = T_{01} P_v(0) = 1. \tag{9.20}$$

With $p_2 = 1$,

$$\tau(1) = 1 + 1 + 1 = 3. \tag{9.21}$$

The situation with outgoing transitions from $S^{(1)}_{(0)}$ is a little more complicated than before, since there are now two transition codewords, 00 and 01. The codeword 00 marks a 2-boundary and thus permits the decoder to switch to $S^{(k_1,k_2,\ldots,k_2)}_{(p_1,p_2,\ldots,p_2)} = S^{(1,1)}_{(0,1)}$ for decoding. Since 00 is the T-prefix p_3 for the third level set $S^{(1,1,1)}_{(0,1,00)}$, the decoder encounters a blocking condition and cannot switch any further. Thus $P(00,1,2) = 1$. The codeword 01, however, marks a 3-boundary, where it encounters a blocking condition because $01 = p_4$. Hence $P(01,1,3) = 1$.

With $P(00,1) = P(01,1) = \frac{1}{4}$, the probability of decoding the T-prefix $p_2 = 1$ is $P(1,1) = \frac{1}{2}$. This yields $T_{12} = \frac{1}{2}$ and $T_{13} = \frac{1}{2}$. All other T_{1m} are zero.

- T-augmentation/synchronisation level 2. The decoding set is now $S^{(k_1,k_2)}_{(p_1,p_2)} = S^{(1,1)}_{(0,1)}$. As before, we first calculate $P_v(2)$:

$$P_v(2) = \sum_{i=0}^{1} T_{i2} P_v(i) = T_{12} P_v(1) = \frac{1}{2}, \tag{9.22}$$

where we have used the previous results, in particular that $T_{02} = 0$. $\tau(2)$ now has five terms:

$$\tau(2) = \frac{2}{3} + \frac{2}{3} + \frac{2}{3} + \frac{1}{2} + \frac{1}{2} = 3. \tag{9.23}$$

The transition codewords in $S^{(1,1)}_{(0,1)}$ come in three flavours: 01 marks a 3-boundary, 11 a 4-boundary, and 100 and 101 mark a 5-boundary, i.e., cause transitions to the final set $S^{(1,1,1,1,1)}_{(0,1,00,01,11)}$. Hence $P(01,2,3) = 1$, $P(11,2,4) = 1$, $P(100,2,5) = 1$, and $P(101,2,5) = 1$. The probabilities of occurrence for these transition codewords are $P(01,2) = \frac{1}{4}$, $P(11,2) = \frac{1}{4}$, $P(100,2) = \frac{1}{8}$ and $P(101,2) = \frac{1}{8}$. The probability of occurrence for the third level T-prefix is $P(00,2) = \frac{1}{4}$.This yields $T_{23} = \frac{1}{3}$, $T_{24} = \frac{1}{3}$, and $T_{25} = \frac{1}{3}$.

- T-augmentation/synchronisation level 3, with $S_{(p_1,p_2,\ldots,p_3)}^{(k_1,k_2,\ldots,k_3)} = S_{(0,1,00)}^{(1,1,1)}$.

$$\begin{aligned} P_v(3) &= \sum_{i=0}^{2} T_{i3} P_v(i) \\ &= T_{13} P_v(1) + T_{23} P_v(2) \\ &= \frac{1}{2} + \frac{1}{3}\frac{1}{2} = \frac{2}{3}. \end{aligned} \tag{9.24}$$

$\tau(3)$ is contributed to by nine codewords:

$$\tau(3) = \frac{2}{3} + \frac{2}{3} + \frac{1}{2} + \frac{1}{2} + \frac{1}{3} + \frac{1}{3} + \frac{1}{3} + \frac{5}{24} + \frac{5}{24} = \frac{15}{4}. \tag{9.25}$$

All codewords in $S_{(0,1,00)}^{(1,1,1)}$ cause transitions to the final set, with the exception of the fourth level T-prefix 01 and the fifth level T-prefix 11 — the only other transition codeword. Hence $P(11,3,4) = 1$, and thus $T_{34} = \frac{1}{3}$ and $T_{35} = \frac{2}{3}$.

- T-augmentation/synchronisation level 4 with $S_{(p_1,p_2,\ldots,p_4)}^{(k_1,k_2,\ldots,k_4)} = S_{(0,1,00,01)}^{(1,1,1,1,1)}$.

$$P_v(4) = \frac{7}{18}. \tag{9.26}$$

$S_{(0,1,00,01)}^{(1,1,1,1,1)}$ is also the largest intermediate set, so the $\tau(4)$ is a correspondingly large expression:

$$\begin{aligned} \tau(4) &= \frac{2}{3} + \frac{1}{2} + \frac{1}{2} + \frac{1}{3} + \frac{1}{3} + \frac{1}{3} \\ &\quad + \frac{5}{24} + \frac{5}{24} + \frac{1}{3} + \frac{1}{3} + \frac{5}{24} \\ &\quad + \frac{5}{24} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{7}{96} \\ &\quad + \frac{7}{96} \\ &= \frac{225}{48} \end{aligned} \tag{9.27}$$

$T_{45} = 1$ as all transitions from this level go to the top level set.

Having calculated all contributing terms, we may now take the sum over m:

$$\begin{aligned}\mathrm{ESD}(S^{(1,1,1,1,1)}_{(0,1,00,01,11)}) &= 1 \times 2 \\ &\quad + 1 \times 3 \\ &\quad + \frac{1}{2} \times 3 \\ &\quad + \frac{2}{3} \times \frac{15}{4} \\ &\quad + \frac{7}{18} \times \frac{225}{48} \\ &= \frac{1039}{96} \\ &= 10.82291\overline{6} \end{aligned} \tag{9.28}$$

This agrees precisely with the value obtained in [46]. A quick consistency check also yields $P_v(5) = P_v(2)T_{25} + P_v(3) + T_{35} + P_v(4)T_{45} = 1$.

9.3.2 $\mathrm{ESD}(S^{(2,1,1)}_{(0,001,01)})$

The T-Code set $S^{(2,1,1)}_{(0,001,01)}$ and its intermediate sets are listed in Table 9.2 on page 131. Unlike the previous example, we are faced with non-trivial blocking conditions. Again, we shall assume that $P(x,m) = 2^{-|x|}$. As before, we start at level 0:

- T-augmentation level 0. By default, $P_v(0) = 1$, and $\tau(0) = 2$ just like in the previous example. Again, we calculate the T_{0m} for use at higher levels. This requires the calculation of $P(x, 0, m)$. The only transition codeword is 1, which always marks a 1-boundary. At first glance we may suspect that 1 also marks a 2-boundary or a 3-boundary depending on the previously decoded symbols. This ambiguity arises because 1 is a suffix (over S) of the T-prefixes $p_2 = 001$ and $p_3 = 01$ and thus satisfies the blocking pre-condition with respect to these. In the case of p_2, $x = 1 \neq p_2$, and thus we search for synchronising and blocking strings. We find that $z = \lambda$, $z = 0$,

T-Code sets at T-augmentation levels 0-3						
$m=0$		$m=1$		$m=2$		$m=3$
S		$S^{(2)}_{(0)}$		$S^{(2,1)}_{(0,001)}$		$S^{(2,1,1)}_{(0,001,01)}$
0						
1	$\rightarrow$	1	$\overset{?}{\rightarrow}$	1	$\overset{?}{\rightarrow}$	1
		01	$\rightarrow$	01		
		000	$\rightarrow$	000	$\rightarrow$	000
		001				
				0011	$\rightarrow$	0011
				00101	$\rightarrow$	00101
				001000	$\rightarrow$	001000
				001001	$\rightarrow$	001001
						011
						01000
						010011
						0100101
						01001000
						01001001

Table 9.2. Listing of the T-Code sets S, $S^{(2)}_{(0)}$, $S^{(2,1)}_{(0,001)}$, and $S^{(2,1,1)}_{(0,001,01)}$, based on the T-augmentation of $S=\{0,1\}$. The arrows indicate possible transitions during the synchronisation process. A transition in a particular line carries the decoder to the set pointed at by the last arrow in the line. Transitions marked with a "?" indicate a blocking pre-condition.

$z=00$, and $z=0^{\star}00$ are the relevant synchronising strings for level 0, and all of them are blocking strings. Thus $P(1,0,1)=1$. By implication, $P(1,0,2)=P(1,0,3)=0$. Hence, we have $T_{01}=1$ and $T_{02}=T_{03}=0$.

- T-augmentation level 1. Using the previous results, $P_v(1)=1$, and $\tau(1)=2$. There is only one (trivial) blocking condition to take into account at this level, due to $01=p_3$, which leads to $P(01,1,2)=1$, and thus $T_{12}=\frac{2}{7}$. Note that the 1, if decoded as a codeword at the first level, does not cause a blocking pre-condition. While it is a suffix of the third-level T-prefix $p_3=01$ when p_3 is decoded over S, it is not a suffix under a decoding with respect to $S^{(2)}_{(0)}$. $T_{13}=\frac{5}{7}$.

- T-augmentation level 2. $P_v(2)=\frac{2}{7}$, and $\tau(2)=\frac{21}{8}$. $T_{23}=1$.

Summing up, we get

$$\mathrm{ESD}(S_{(0,001,01)}^{(2,1,1)}) = 1 \times 2 + 1 \times 2 + \frac{2}{7} \times \frac{21}{8} = 4.75. \qquad (9.29)$$

Our consistency check shows again that $P_v(3) = 1$ for the $P_v(m)$ and T_{im} used.

9.4 Special Cases and Computational Complexity

9.4.1 T-Expansion Parameters

Expansion parameters have no special significance in the algorithm presented above, as the T-prefix delay accounts for repetitive T-prefix patterns. Longer codewords due to T-expansion parameters $k_m > 1$ are also treated in exactly the same way as before in calculating the suffix delay. This does not come as a surprise since the T-Code synchronisation mechanism does not depend on T-expansion parameters.

9.4.2 Mismatched Sources

If the source is not matched perfectly to the T-Code set used, i.e., when $P(x, n) \neq \#S^{-|x|}$ for some $x \in S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$, we have to calculate the $P(x, m)$ for all x and all m from the (known) source statistics for $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$. Since every codeword in a set $S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$ has a unique decoding over $S_{(p_1,p_2,\ldots,p_{m-1})}^{(k_1,k_2,\ldots,k_{m-1})}$, we may compute $P(x, m-1)$ from the relevant probabilities at level m, using the relationship

$$P(x, m-1) = \frac{\sum_{x' \in S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}} P(x', m)|x'|_{x,m-1}}{\sum_{x' \in S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}} P(x', m)|x'|_{m-1}}, \qquad (9.30)$$

where $|x'|_{x,m-1}$ denotes the number of occurrences of $x \in S_{(p_1,p_2,\ldots,p_{m-1})}^{(k_1,k_2,\ldots,k_{m-1})}$ in the $S_{(p_1,p_2,\ldots,p_{m-1})}^{(k_1,k_2,\ldots,k_{m-1})}$-decoding of x', and $|x'|_{m-1}$ denotes the total number of codewords from $S_{(p_1,p_2,\ldots,p_{m-1})}^{(k_1,k_2,\ldots,k_{m-1})}$ in the $S_{(p_1,p_2,\ldots,p_{m-1})}^{(k_1,k_2,\ldots,k_{m-1})}$-decoding of x'.

The numerator of this expression may be thought of as the number of occurrences of x in an average codeword from $S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)}$. The denominator may be thought of as the "length" of that average codeword, measured in codewords over $S^{(k_1,k_2,\ldots,k_{m-1})}_{(p_1,p_2,\ldots,p_{m-1})}$, thus normalising the number of occurrences of x to a probability.

A recursive top-down application of Equation (9.30), starting with $m = n$ yields the desired probabilities for all intermediate sets.

9.4.3 Computational Complexity

The computational complexity of our new algorithm has practical significance as it governs both the speed of calculation and its memory requirements. Both the speed and the memory requirements may be specified in "Big-O"-notation (see, e.g., [1]).

For the computation of $\tau(m)$ for each m, the delay contribution of each codeword in the intermediate sets up to $S^{(k_1,k_2,\ldots,k_{n-1})}_{(p_1,p_2,\ldots,p_{n-1})}$ needs to be taken into account. Each intermediate set $S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)}$ contains at most half the number of codewords of the next higher set $S^{(k_1,k_2,\ldots,k_{m+1})}_{(p_1,p_2,\ldots,p_{m+1})}$ plus p_{m+1}. Thus the total number of codewords that have to be inspected for each level is always less than $2\#S^{(k_1,k_2,\ldots,k_{n-1})}_{(p_1,p_2,\ldots,p_{n-1})} + n$. Memory is required for the probabilities $P(x, m)$. Summing up over all values of m, the total memory requirement and the execution time of the $\tau(m)$ calculations are thus each of order $O(\#S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}) = O(2^n)$.

The computation of the $P_v(m)$ consists mostly of the computation of the set transition probabilities T_{im}.

The matrix T_{im} itself is of order $O(n^2)$. For each T_{im}, $\#S^{(k_1,k_2,\ldots,k_i)}_{(p_1,p_2,\ldots,p_i)}$ codewords need to be inspected to obtain the $P(x, i, m)$. This part of algorithm does not require any significant amount of memory since we have already accounted for the $P(x, m)$ elsewhere. The execution time is roughly the same for most $P(x, i, m)$, which yields $O(2^{n-1})$ using similar arguments to those presented above. Blocking pre-conditions are rather rare for large sets and the number of relevant synchronising and blocking strings small compared to the general size of the set. Hence they do not require a lot of processing compared to the rest of the set, such that we may assume that their contribution in terms of required storage and execution time is usually negligible.

The execution time for the computation of the $P_v(m)$ is therefore of order $O(2^n n^2)$. We may therefore claim that the total algorithm is of order $O(2^n n^2)$, provided that the source is perfectly matched.

In the case of a mismatched source, an individual calculation for $P(x, m-1)$ is of order $O(\#S^{(k_1,k_2,\ldots,k_m)}_{(p_1,p_2,\ldots,p_m)}) \approx O(2^m)$. For the whole intermediate set $S^{(k_1,k_2,\ldots,k_{m-1})}_{(p_1,p_2,\ldots,p_{m-1})}$, approximately 2^m of these calculations have to be performed, i.e., for the whole intermediate set we have an operation of $O(2^{2m})$. This implies that the algorithm for all intermediate sets is of order $O(2^{2n})$. For large sets with mismatched sources, this is the dominant operation.

9.5 Discussion

The algorithm presented here permits a precise computation of the ESD of any T-Code set based on any alphabet, driven by a source with arbitrary source statistics.

The algorithm has the potential to be implemented as a computer program, and would provide an efficient engineering tool for T-Code communication applications.

CHAPTER 10

Approaches to Source Coding With T-Codes

The classic area of application for variable-length codes is data compression by source coding. Huffman codes are perhaps the best-known example for this. This chapter discusses the problem of finding the T-Code set that offers the best compression for a given source statistic.

10.1 Source Coding

Source coding assumes the following situation: an information source emits a finite number N_s of distinct source symbols σ_i in a continuous stream. $P(\sigma_i)$ denotes the probability that the next source symbol emitted by the source is σ_i. Each σ_i is assigned a unique variable-length codeword $x(\sigma_i)$ from a prefix-free code set $C \in S^+$.

The expected redundancy r of this encoding is given by the difference between the weighted mean of the codeword lengths of the encoded source and the source's entropy (cf. Shannon [39]):

$$r = \sum_i P(\sigma_i) \left[|x(\sigma_i)| + \log_{\#S} P(\sigma_i) \right] \tag{10.1}$$

The simple source coding model generally assumes that $P(\sigma_i)$ is independent of previously emitted symbols — an assumption that does not hold in many practical cases. Still, in many cases this presents a fair approximation and significant compression gains may be achieved by minimising r.

Huffman [30] introduced his now famous algorithm for the construction of a prefix-free variable-length code with minimal redundancy, assuming such a simple source coding model. His algorithm has found many practical applications. It is used in the `pack` command under UNIX and for the compression of DCT coefficients in the popular JPEG image compression.

The Huffman algorithm starts with given source probabilities $P(\sigma_i)$ and yields a variable-length code set over S with minimal r. This code set is generally not the only possible one that minimises r. One of the reasons for this is that r does not depend on the code itself, but merely on the code length distribution, i.e., the histogram function of codeword lengths. This is a result of the $|x(\sigma_i)|$-term in Equation (10.1). Codes that share the same code length distribution are hence equivalent in the sense of Equation (10.1).

Capocelli, Giancarlo, and Taneja [4] presented bounds on the redundancy of Huffman codes. Similar work was also undertaken by Gallager [11] and Johnsen [32].

10.2 Source Coding with T-Codes

Source coding with T-Codes requires a different approach than Huffman coding. Whereas the Huffman algorithm is constrained only by $P(\sigma_i)$ and the requirement to minimise r, the requirement that the final code be a T-Code set places an additional constraint on the process. This renders the Huffman algorithm unusable for our purposes.

Unfortunately, it also means that there is not always a T-Code set for which r can be minimised to the same value as for a Huffman code for the same source, as illustrated in the following example:

Example 10.2.1 (T-Code sets with Non-Minimal Redundancy)
Consider a source with six symbols and associated probabilities $\frac{1}{4}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{8}$, $\frac{1}{8}$, *and* $\frac{1}{8}$ *respectively. If we were to use a block code for encoding, we would require a 3-bit code. The Huffman code*

$$C = \{00, 01, 100, 101, 110, 111\}$$

yields minimum redundancy with an average codeword length of 2.5 bits, such that the block code features a redundancy of $r = 0.5$ *bits. The closest T-Code set we can obtain here is, e.g., the set* $S_{(0,1)}^{(1,1)}$ *which has* $r = 0.125$.

Thus, there is often a trade-off in efficiency when a T-Code set is used for source coding. This trade-off may be more than compensated by other benefits such as good self-synchronisation.

This poses the question as to how we may find a T-Code set with minimal r for a given source.

For many practical applications it may be sufficient — and quicker to resolve — to simply soften the criterion somewhat and look for a T-Code set with a "small" r, rather than the smallest r achievable.

To date, the only strategy that has been identified for minimising r is to perform an exhaustive search of all feasible code length distributions. This approach was first presented by the author in [17]. This chapter presents a much improved version of this algorithm.

10.3 The Search Algorithm

The search algorithm presented here operates mainly as a recursive "divide-and-conquer" algorithm. This section discusses the general structure of the algorithm, whereas the following section (10.4) takes a detailed look at the techniques used.

Since the code length distribution determines the redundancy, the algorithm simply works with the set distributions rather than with the full code sets. As the number of possible distributions is effectively unlimited, the algorithm uses several feasibility criteria that permit us to use a "branch-and-bound" technique [28, 3], which limits the number of sets that we need to search. These will be discussed in the next section.

The algorithm consists of two parts (see also Figure 10.1 above and Figure 10.2 overleaf):

1. Initialisation: sets up global parameters such as the variables for recording the best redundancy and distribution found so far. It also sets the "seed" distribution of the alphabet (i.e., $\#S$ codewords of length 1, zero codewords for all other lengths) and uses it as the base distribution for the recursive matching subroutine.

2. Recursive matching subroutine. If the base distribution is large enough to encode the source, the procedure first calculates the redundancy associated with the base distribution and updates the global records on the best redundancy set, if applicable.

   Starting with the shortest available T-prefix length, the procedure loops through all combinations of available T-prefix lengths and T-expansion parameters that meet the feasibility criteria that we have yet to discuss (see below). For each combination, it creates a code length distribution corresponding to a T-augmentation from the base distribution. Any distributions

```
program   recursivesearch
var
          global bestredundancy: float;
          global bestset: TCodeSet;
          global P(σ_i): array[1..N_s] of float;
          baseset: TCodeSet;
          { a TCodeSet here consists of a code length distribution}
          { and a T-prescription, with T-prefix lengths }
          { rather than full T-prefixes }
procedure recursive_match(baseset); { forward declaration, see Figure 10.2 }

{ main program code starts here }
begin
          baseset:=alphabet;
          { the code length distribution of baseset consists of }
          { #S symbols of length 1 and no symbols of other lengths. }
          { The T-prescription is empty. }
          bestset:=none;
          { we have yet to find a matching set }
          bestredundancy:=infinity;
          { any large enough set will match this! }
          input(P(σ_i));
          recursive_match(baseset);
          { call the recursive routine }
          output(bestset,bestredundancy)
end.
```

Figure 10.1. pseudo code (part 1) of a recursive search algorithm used to find the T-Code set distribution with the lowest redundancy for a given source probability distribution: main routine. Some details have been omitted to underline the basic concept. The recursive matching procedure is listed in part 2 (see Figure 10.2).

```
procedure  recursive_match(baseset)
var
           newredundancy: float;
           tmpset: TCodeSet;
           p: integer; {for T-prefix length}
           k: integer; {for T-expansion parameter}

begin
           { first calculate redundancy for the baseset }
           { provided it is large enough to encode the source }
           if (size(baseset) >= N_s) then begin
              newredundancy:=redundancy(baseset,P(σ_i));
              { update records for best set if appropriate }
              if (newredundancy < bestredundancy) then begin
                 bestset:=baseset;
                 bestredundancy:=newredundancy;
              end;
           end
           { now loop through all feasible T-prefix lengths }
           { choose shortest available codeword first.}
           p:=shortest(basedistribution);
           { However, p must be as least as long as the T-prefix }
           { of the last virtual T-augmentation. }
           if (p<lastp(baseset)) then begin
              p:=lastp(baseset);
              { if no codewords of that length exist }
              { increase p until we find one }
              while (basedistribution[p]==0) do p:=p+1;
           end;
           while feasible(p) begin
              { loop through all feasible k }
              k:=1;
              while feasible(p,k) do begin
                 tmpset:=v_taugment(baseset,p,k);
                 recursive_match(tmpset);
                 { get next higher k }
                 k:=next_prime(k+1)-1;
              end;
              { get next higher p }
              { p:=nextp(p,baseset);
           end;
end;
```

Figure 10.2. pseudo code (part 2) of a recursive search algorithm used to find the T-Code set distribution with the lowest redundancy for a given source probability distribution. This listing describes the recursive routine. The individual feasibility criteria are discussed in the main text of this chapter. Some details have been omitted to underline the basic concept.

thus obtained are used by the procedure as the base distributions for recursive calls to itself.

At the end of the algorithm's run, all feasible distributions have been generated, and the best achievable redundancy is known. A set of T-prefix lengths and T-expansion parameters for a code set with this redundancy is also returned.

10.4 Feasibility Criteria and Simplifications

This section discusses the techniques that are used to ensure that

- the search algorithm terminates,
- the search of multiple equivalent code length distributions is avoided,
- the code length distributions are restricted to codeword lengths of interest,
- code length distributions that cannot yield a minimum redundancy are avoided if possible.

10.4.1 Virtual T-Augmentation

Since the code length distribution is the determining factor in the calculation of r, two code sets with the same code length distribution have the same redundancy. Hence, rather than searching for a "best set", we may restrict ourselves to searching for the best distribution and a "recipe" that permits the construction of a set which features that distribution. For this purpose, the algorithm uses a technique called "virtual T-augmentation".

Let δ_C denote the code length distribution function of a code C, such that $\delta_C(l)$ denotes the number of codewords of length l in the code set C, and define $\delta_C(l) = 0$

for $l \leq 0$. Then

$$\delta_S(l) = \begin{cases} \#S & \text{if } l = 1 \\ 0 & \text{otherwise} \end{cases} \tag{10.2}$$

and

$$\delta_{S^{(k_1,k_2,\ldots,k_{n+1})}_{(p_1,p_2,\ldots,p_{n+1})}}(l) = \begin{cases} \delta_{S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}}(l) - 1 & \text{if } l = |p_{n+1}| \\ \sum_{k'_{n+1}=0}^{k_{n+1}} \delta_{S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}}(l - k'_{n+1}|p_{n+1}|) & \text{otherwise} \end{cases} \tag{10.3}$$

The last equation defines the *virtual T-augmentation.* Note that only the length of the T-prefixes plays a role here — if $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ contains more than one codeword of the intended T-prefix length $|p_{n+1}|$, then $\delta_{S^{(k_1,k_2,\ldots,k_{n+1})}_{(p_1,p_2,\ldots,p_{n+1})}}(l)$ does *not* depend on *which* of these is chosen as the T-prefix in a corresponding "real" T-augmentation. On the other hand, the choice of T-expansion parameters has a significant influence on $\delta_{S^{(k_1,k_2,\ldots,k_{n+1})}_{(p_1,p_2,\ldots,p_{n+1})}}(l)$.

Virtual T-augmentation permits us to manipulate T-Code set code length distributions rather than full sets, which is exactly what is required if we wish to minimise the redundancy r.

10.4.2 Avoiding Multiple Equivalent T-Prescriptions

As mentioned above, two code sets will have the same redundancy r if they have the same code length distribution. Hence, it is sufficient to generate the code length distribution for just one such set to obtain r for all sets having the same distribution. In particular, two T-Code sets share the same distribution if they have been generated by equivalent T-prescriptions (cf. 3) — in this case we are simply generating the distribution for the same set.

An obvious way of preventing this duplication of distributions is to specify that all set T-prescriptions must be in their anti-canonical form (see Chapter 3). Even though we are not dealing with full T-Code sets here, we know that set distributions corresponding to a non-anti-canonical T-prescription can also be generated by a corresponding anti-canonical T-prescription, and hence we can restrict our search to the latter. It suffices to demand that the T-expansion parameters used must be one less than a prime number. Since the number of primes in any larger finite interval of $\mathbb{N}$ is significantly less than the number of natural numbers in that interval, this constraint saves a significant amount of computation time.

10.4.3 Non-Decreasing T-Prefix Lengths

Limiting the T-prescriptions to the anti-canonical form is only one way of eliminating duplicate code length distributions. Further savings are possible if the T-prefixes are used in ascending order of length:

Consider the form of codewords in $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ as presented in Theorem 4.2.3,

$$x = p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1} k'_0, \tag{10.4}$$

and the form of the associated pseudo-T codewords from Theorem 6.1.2

$$x_\phi = p_n^{k'_n} p_{n-1}^{k'_{n-1}} \ldots p_1^{k'_1} \lambda, \text{ with } 0 \le k'_i \le k_i \text{ for } i = 1, \ldots, n. \tag{10.5}$$

Given a fixed set of T-prefix lengths and T-expansion parameters for these two equations, neither of the code length distributions that these two equations give rise to is changed by "swapping" the order of any two adjacent substrings of the form $p_{m-1}^{k'_{m-1}}$ and $p_m^{k'_m}$ for $2 \le m \le n$. I.e., for "adjacent" T-augmentations,

$$\delta_{S_{(p_1,p_2,\ldots,p_{n-1},p_n)}^{(k_1,k_2,\ldots,k_{n-1},k_n)}} = \delta_{S_{(p_1,p_2,\ldots,p_n,p_{n-1})}^{(k_1,k_2,\ldots,k_n,k_{n-1})}}, \tag{10.6}$$

provided that both $S_{(p_1,p_2,\ldots,p_{n-1},p_n)}^{(k_1,k_2,\ldots,k_{n-1},k_n)}$ and $S_{(p_1,p_2,\ldots,p_n,p_{n-1})}^{(k_1,k_2,\ldots,k_n,k_{n-1})}$ exist.

Here, we have two cases to consider:

- $|p_n| \le |p_{n-1}|$. In this case, $p_n \in S_{(p_1,p_2,\ldots,p_{n-2})}^{(k_1,k_2,\ldots,k_{n-2})}$ and both sets exist. If we demand that T-prefixes should be used in ascending order of length for all virtual T-augmentations, we generate the distribution for the second set and hence, by equivalence, cover the first set, too.

- $|p_n| > |p_{n-1}|$. In this case, the latter set *may* not exist: if $p_n \notin S_{(p_1,p_2,\ldots,p_{n-2})}^{(k_1,k_2,\ldots,k_{n-2})}$, then p_n must be generated as a result of the $(n-1)$'th T-augmentation and hence $|p_n| > |p_{n-1}|$. If we demand non-decreasing T-prefix lengths for all virtual T-augmentations, we will generate the distribution for the first set. Whether the second set exists or not is thus unimportant as its possible code length distribution is covered by default. $\square$

From the above we may conclude that it is safe to require that the T-prefix length should be non-decreasing for all successive virtual T-augmentations.

10.4.4 Assignment of Codewords

When calculating the redundancy of a T-Code set, one must of course assign source symbols to codewords. This is obviously done in rank order of probability, i.e., the shortest codewords in the T-Code set get assigned to the highest probabilities such that

$$P(\sigma_i) > P(\sigma_{i'}) \Rightarrow |x(\sigma_i)| \leq |x(\sigma_{i'})|. \tag{10.7}$$

This implies that some of the longer codewords in the T-Code set may be unassigned. In turn, this yields a feasibility criterion for the choice of T-prefix lengths and T-expansion parameters.

10.4.5 Feasibility of a Virtual T-Augmentation

Presume that the present distribution $\delta_{S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}}$ contains a sufficient number of codewords to enable us to encode the source. Let L and ℓ be the length of the longest and shortest assigned codewords in the present distribution. Further let $|p_{n+1}|$ be the T-prefix length, and k_{n+1} the T-expansion parameter under consideration. Then we can require that

$$k_{n+1}|p_{n+1}| + \ell < L. \tag{10.8}$$

The left hand side of this inequality is the length of the shortest codeword that would be newly created in this (virtual) T-augmentation. If this length is not shorter than the longest codeword assigned so far, then no source symbol can be assigned a shorter codeword in the new distribution and hence r will not decrease. Nor would any of the newly created codewords in this distribution serve as sensible T-prefixes. Consequently, it does not make sense to further investigate this distribution.

As any codeword that is used as a T-prefix for a virtual T-augmentation is no longer available in the T-augmented set, the criterion may be tightened further: we may require that at any new T-augmentation produce at least two new codewords shorter than L — one to compensate for the loss of the T-prefix codeword, and one to achieve more efficient encoding as above. Hence, we define ℓ_2 as the length for which there are at least two codewords with length smaller than or equal to ℓ_2. The previous equation thus becomes:

$$k_{n+1}|p_{n+1}| + \ell_2 < L. \tag{10.9}$$

Note that this requirement puts a bound on the number of sets that need to be searched and hence guarantees that the search algorithm will terminate.

10.4.6 Redundancy Criterion

The feasibility criteria presented so far are independent of the probabilities of occurrence of the source symbols that we wish to encode. Given a certain number of source symbols and using only the feasibility criteria above, the search algorithm would thus always search the same number of distributions. The criterion introduced here is used to determine the feasibility of distributions on the basis of the source symbol probabilities:

Presume that we have established an upper bound for the redundancy of the most efficient T-Code set, e.g., from a set distribution whose redundancy we have previously calculated. As we calculate the redundancy for a newly generated distribution, we may without loss of generality add the $P(\sigma_i)|x(\sigma_i)|$ in order of decreasing $P(\sigma_i)$ and watch the sum's value after each addition. Once the sum exceeds the previously established bound, we know that the present set is not a contender for the most efficient encoding. This saves some work.

To improve on the previous bound, a T-augmentation leading to a more efficient set must change the code length distribution for codeword lengths up to the length at which the initial set's redundancy summation exceeded the bound. This requires a T-prefix of less than that length. Invoking the results on the length order of T-prefixes from above, we obtain a replacement value for L in Equation (10.9) for further T-augmentations with the present set distribution as the base distribution.

10.4.7 Maximum Feasible Codeword Length

Another significant saving can be made if we acknowledge that codewords above a certain length are simply of no interest. Given a coding alphabet with $\#S$ symbols, we know that any complete code set C over S with a maximal codeword length of $|\hat{x}|$ must have a minimum number of codewords:

$$\#C \geq (|\hat{x}| - 1)(\#S - 1) + \#S. \tag{10.10}$$

If we require exactly $N_s = \#C$ codewords, we obtain a bound on the maximum codeword length possible:

$$L_m = |\hat{x}| = \left\lceil \frac{N_s - 1}{\#S - 1} \right\rceil, \tag{10.11}$$

where $\lceil q \rceil$ denotes the smallest integer greater than or equal to q. Given N_s, we may choose $k = L_m - 1$ and any $p \in S$ to obtain a T-Code set $S_{(p)}^{(k)}$ such that the longest codewords in $S_{(p)}^{(k)}$ are of length L_m. Since $S_{(p)}^{(k)}$ has a sufficient number of codewords to encode the source, we may ask whether it is possible to encode the source more efficiently with a T-Code set that requires us to assign codewords longer than L_m.

Theorem 10.4.1 (Maximum Codeword Length)

There exists no source with N_s symbols such that the T-Code set with the lowest redundancy in an encoding of that source requires the assignment of codewords longer than L_m.

A formal proof of this theorem is an open problem. However, it is easy to give a "handwaving argument" for why the theorem is sensible: consider $S_{(p)}^{(k)}$ as above. $S_{(p)}^{(k)}$ is not only a T-Code set, but also the possible outcome of a Huffman code construction process. Of all Huffman codes possible for a source with N_s characters, $S_{(p)}^{(k)}$ is the one with the longest possible codewords. Since the Huffman code construction yields a minimum redundancy code for a given source, all possible Huffman codes for the source

- yield the same or a lower redundancy than $S_{(p)}^{(k)}$, and
- use only codewords up to length L_m.

As $S_{(p)}^{(k)}$ is a T-Code set, it sets a bound on the redundancy for the set we wish to find. The redundancy criterion we have introduced above may be applied here in a similar way — any improvement in the redundancy that could be achieved requires an increase in the number of codewords that are shorter than L_m. This then enables "codeword assignment swapping", i.e., the codeword(s) that are used as T-prefix(es) disappear, but their loss must be outweighed by a gain from the additional codewords created, i.e., it must be possible for source symbols with previously longer assignments to "move up" the tree. Since the longest possible assignment in a completely filled tree is of length L_m, any "better" tree cannot have any longer codewords assigned.

This leaves only those sets with codewords up to length L_m or less to consider.

□

Thus we may assume that the virtual T-augmentations in our algorithm do not need to keep track of codeword lengths larger than L_m.

Furthermore, it means that a virtual T-augmentation is not feasible unless

$$k|p| + \ell_2 < L_m. \tag{10.12}$$

This complements the already established rule for L: we now have a feasibility criterion that also works for sets that are too small to encode the source. For larger sets, we may simply replace L_m by L.

It should be noted that the other feasibility criteria mentioned in this chapter ensure on their own that the algorithm will terminate. However, in practice, the main benefit of the theorem is that it permits some savings to be made with respect to memory requirements and computation time, as the size of the distributions is limited.

10.4.8 Dropping the Logarithms

If we take a closer look at Equation (10.1), we notice that the term with the logarithms of the source symbol probabilities is constant. We may hence drop it when comparing the redundancies of two sets.

10.5 Performance of the Search Algorithm

One problem of the algorithm presented above is its execution time, which depends primarily on the number of set distributions that have to be searched. As a general rule, this increases with N_s. However, due to the redundancy bound feasibility criterion, it also has a strong dependence on the source symbol probabilities. Figure 10.3 shows the number of sets searched for N_s source symbol probabilities. Figure 10.4 shows the CPU time taken by a 250 MHz DEC AlphaServer 2100A for which the algorithm was implemented in C as a CMEX function for MATLAB. It is evident from this that the source probability distribution is itself the dominant factor in determining the execution time of the algorithm.

Figure 10.5 further shows that the "skew" in the distribution also has a direct influence on the number of sets searched.

10.6 Other Approaches

Mark Titchener [43] has suggested another approach for source coding with T-Codes. It exploits the fact that the largest efficiency gains are made with the shortest codewords. He proposes to encode the source as a Huffman code to obtain an optimal codeword length distribution. Starting with the shortest codewords in that distribution, one must then try to construct a T-Code set that matches it closely.

While this may not yield a T-Code set with the lowest achievable redundancy, it will generally yield one with a low redundancy, thus achieving the bulk of the compression savings. This is well justified in a number of practical situations, e.g., where only approximate source statistics are known and the additional inefficiency would be small compared to the errors in the symbol probabilities.

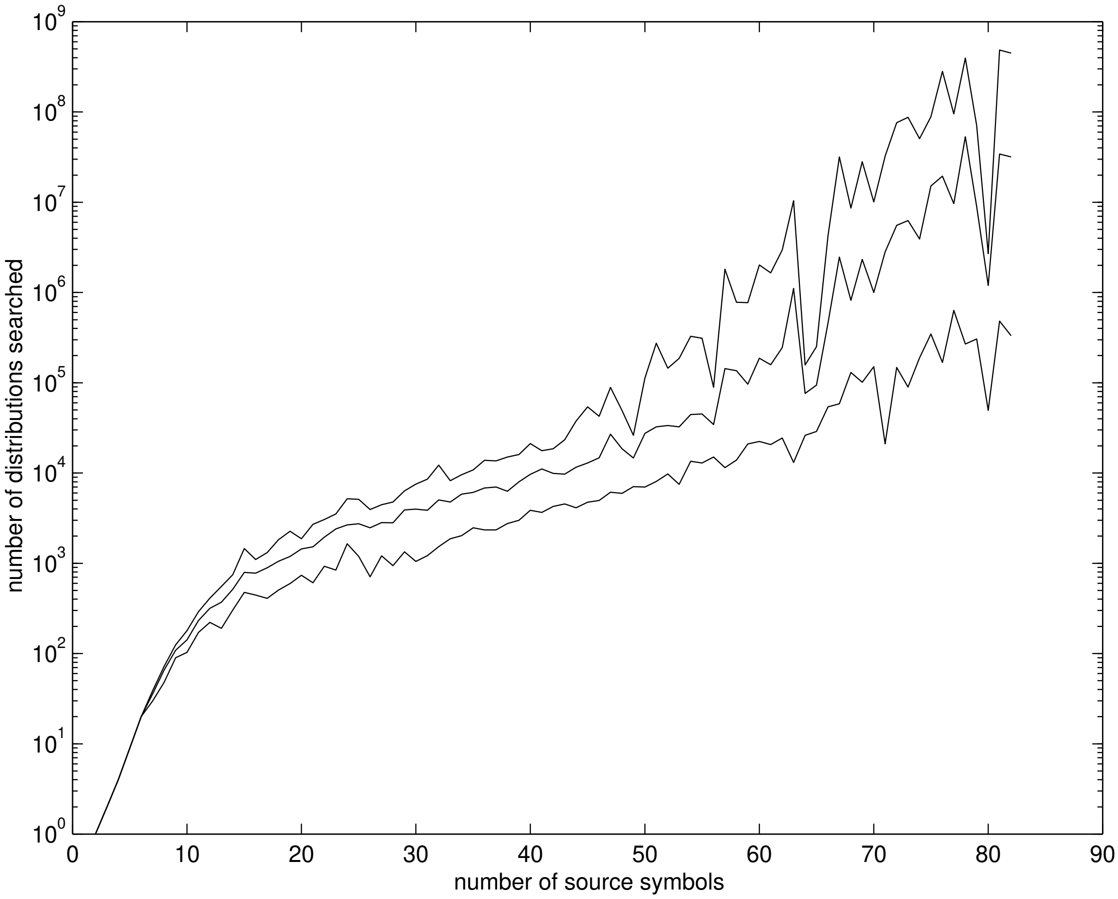


Figure 10.3. The execution time of the matching algorithm presented in this chapter depends primarily on the number of sets that the algorithm has to search. This number in turn depends on the number of source symbols, N_s, and their associated probabilities of occurrence. In the above plot, fifteen skewed probability distributions were randomly generated for each N_s and used as input to the matching algorithm (assuming a binary alphabet). The centre curve shows the average number of sets searched by the matching algorithm in this experiment. The top and bottom curves show the largest/smallest number of sets searched for each set of fifteen runs. It is evident that the source probability distribution itself has a paramount influence on the number of sets that need to be searched.

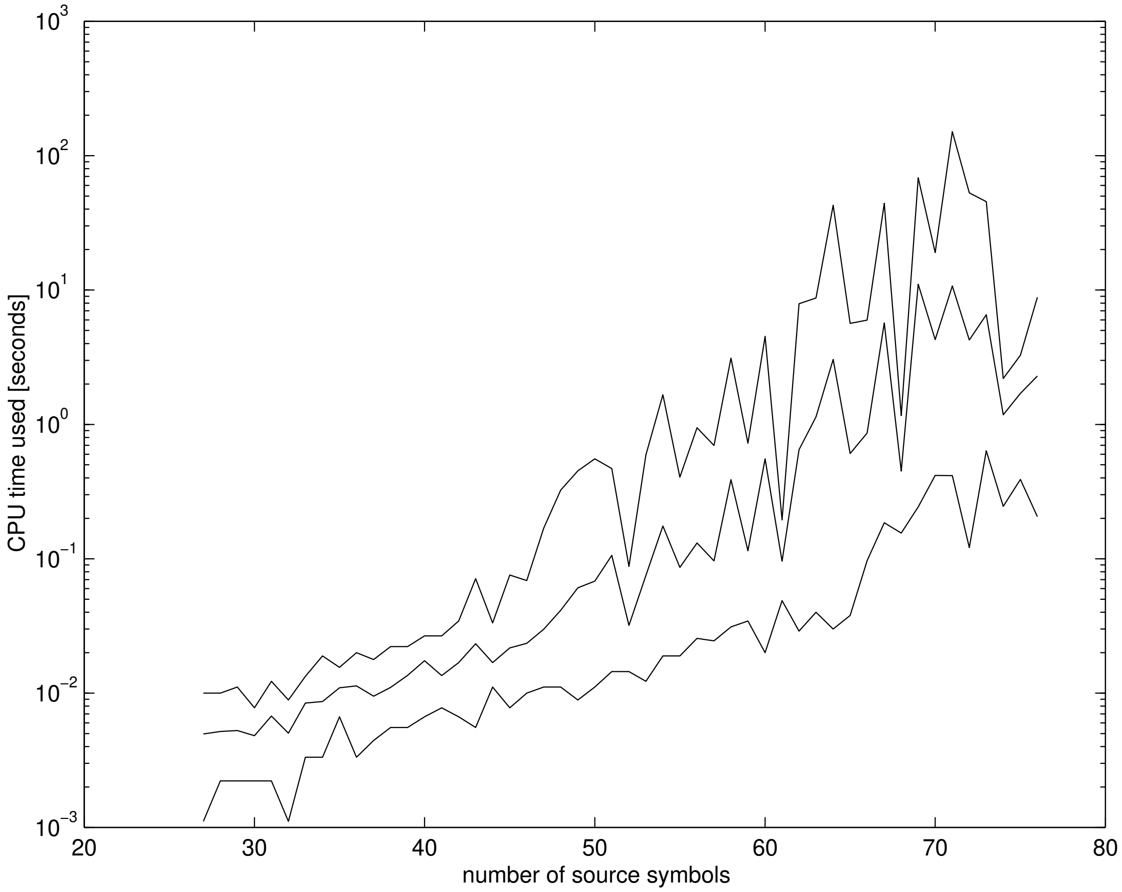


Figure 10.4. Execution time of the matching algorithm presented in this chapter for sources with N_s between 27 and 77. Fifteen skewed probability distributions were randomly generated for each N_s and used as input to the matching algorithm (assuming a binary alphabet). The centre curve shows the average CPU time taken by the matching algorithm in this experiment. The top and bottom curves show the highest/lowest CPU time for each set of fifteen runs. Note: CPU times shown are as reported by MATLAB (for distributions with N_s less than about 27, the minimum reported CPU time was often zero such that these runs had to be left out for scaling reasons).

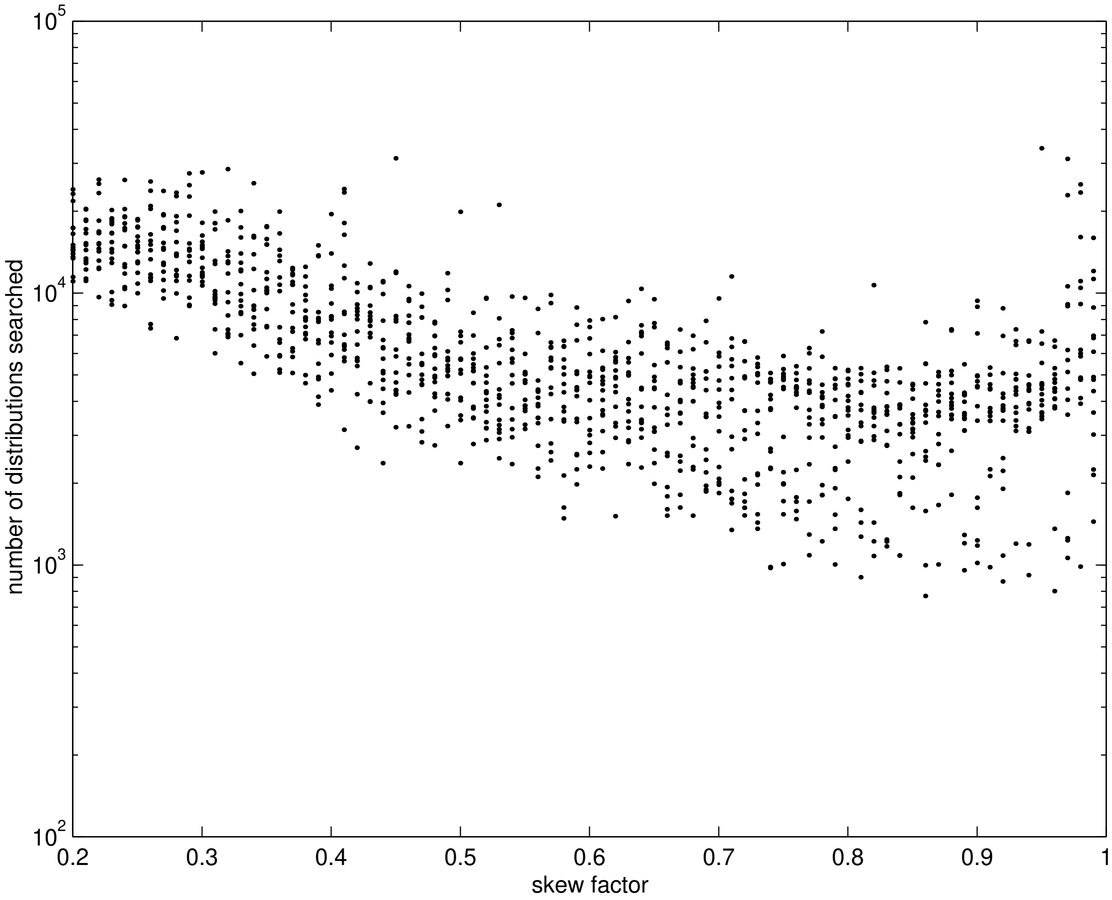


Figure 10.5. The number of sets that need to be searched to find the best match for a given source size N_s depends strongly on the source probabilities. For the plot above, probability distributions of size $N_s = 30$ were randomly generated with a "skew factor" bias. Low skew factors correspond to a small ratio between highest and lowest probability in the distribution. High skew factors correspond to a high ratio. For each skew factor (in 0.01 increments), fifteen distributions were generated and matched. It is evident that distributions with small skew factors generally require more sets to be searched.

10.7 Discussion

When we wish to find the most efficient T-Code set for a given set of source symbol probabilities, the constraints set by the T-augmentation construction prevent us from using the well-known Huffman code construction algorithm. To date, the only exact solution to this problem appears to be an exhaustive search of all feasible T-Code code length distributions.

The number of distributions that need to be searched for this purpose may be restricted by the use of a number of feasibility criteria in a branch-and-bound algorithm. The introduction of additional criteria has now led to the algorithm presented above, which is a substantial improvement on the algorithm presented by the author in [17]. In fact, it is now possible to match sources with several dozen symbols faster than a dozen symbols under the previous algorithm. The algorithm's output are the T-expansion parameters and T-prefix lengths of a T-Code set with minimal redundancy. This leaves a choice when it comes to picking the T-prefixes. It is thus possible to select a set on the basis of other properties, such as a minimal ESD.

While these recent improvements give the algorithm practical value, it still seems that the execution time can only be bounded by some exponential function. In other words, the algorithm seems to be of order $O(exp(N_s))$. This still compares unfavourably to the Huffman algorithm which is of order $O({N_s}^2)$.

Further improvements to the method presented are conceivable. For example, one approach could be in the form of new feasibility criteria — the author doubts that he has exhausted all possibilities. However, any savings obtained in this way will have to be weighed against the costs of such additional feasibility tests. Additional criteria are more likely to be of benefit if they lead to a "pruning" of the search tree close to the root, or if they rule out a large proportion of sets that would otherwise be searched. This also suggests that using a "breadth first" algorithm rather than the current "depth first" approach may result in a significant reduction in computational complexity.

Another improvement could derive from the parallelisation of the algorithm. Given a base distribution, the recursive calls for different T-augmentations may be processed in parallel, on a multi-processor machine or over a distributed network with a good load balancing scheme. The gains here might extend beyond those due to the extra processing power available: the initial shape of most T-Code code length distributions is formed during the first few T-augmentations, and these short codewords have the most significant influence on the code's redundancy. If these short distributions could be processed in parallel, updated (i.e., lower) bounds for the branch-and-bound could become available earlier and might save parallel searches from unnecessary recursions.

For some practical implementations, the algorithm suggested by Titchener may also prove to be the most economic. This could be the case especially when large sources are involved, the encoding speed is paramount, and the efficiency is only of secondary importance.

CHAPTER 11

Outlook and Conclusions

This chapter summarises the open problems that arise from this thesis and other research on T-Codes. Possible avenues of further research are also discussed.

11.1 Open Problems on T-Codes

The research undertaken for this thesis has left some open problems, some of which have already been mentioned:

- finding an optimal covering T-Code set for an arbitrary variable-length code such that the storage cost in T-depletion format is minimised. This problem is somewhat similar to the problem of matching a given source probability distribution which has been discussed in Chapter 10.

- T-Code self-synchronisation, still regarded as a partially open problem a few years ago, is now well understood in the context of hierarchical coding alphabets (see Chapters 8 and 9). However, some research still remains to be done in an associated area opened by Mark Titchener [50]: the synchronisation of general variable-length codes based on the T-Code model. Titchener suggests that the synchronisation process of a general variable-length code C is closely related to that of T-Code sets that share a large number of codewords with C. The more of the short (and hence more frequent) codewords are shared, the better will the T-Code set in question model the synchronisation behaviour of C. Finding the T-Code set that most closely mimics the synchronisation behaviour of a given variable-length code is an open problem. It is similar to the other problems mentioned above, except that the aim here is to minimise the ESD. An added problem here is that the automaton concept of T-Code synchronisation is not easily transferable to variable-length codes in general.

- finding the most efficient T-Code set for the encoding of a given source. An algorithm similar in execution time to the recursive bottom-up-top-down method used in the generation of Huffman codes would certainly be extremely useful. The present approaches (exhaustive search, as presented in the previous chapter, and approximation of Huffman code length distributions, as proposed by Titchener) are either too slow or not guaranteed to find the most efficient T-Code set. However, this should not detract from the usability of T-Codes for source coding purposes — the largest compression gains are made by coding the shortest codewords properly, which are relatively easy to match using both methods mentioned above.

- maximum feasible codeword length. The formal proof for Theorem 10.4.1 is still an open problem, although perhaps not a difficult one.

- improvements to the search algorithm presented in Chapter 10, in particular with respect to parallelisation.

- a practical implementation of the ESD calculation in Chapter 9 as a computer program.

The research for this thesis has also touched on and identified a number of other areas that in the author's opinion warrant further research:

- channel coding. The choice of T-prefixes in the T-augmentation process may be exploited to construct T-Code sets that permit matching an information source to the frequency response of the communication channel. Some preliminary work in this area was done by the author together with Mark Titchener and Radu Nicolescu [20].

- synchronisation and soft-decision decoding. It was noted in Chapter 8 that the synchronisation information that can be provided by a T-Code decoder is based on the assumption that no further symbol errors have occurred since the beginning of the synchronisation process.

  In practice, this is an answer to the question: "provided the last <number> symbols were received correctly, are you (the decoder) synchronised?" In other words, the decoder's answer is a conditional probability.

  In practical applications, it could be desirable to have a general confidence value rather than a conditional probability. This could be achieved by using soft-decision decoding (cf., e.g., [8]) to ascertain the probability that a bit is in error. Thus, it would be possible to associate a general confidence value with the decoder synchronisation level. This could be useful under circumstances where secondary symbol errors (that occur after the start of the synchronisation process) may lead to loss of synchronisation.

- string generation. Nicolescu's uniqueness result, as reconfirmed in Chapter 3, offers more than just a simple way of communicating a T-Code set between encoder and decoder. In particular, it establishes T-augmentation not only as a set generation algorithm, but also as a string generation algorithm. Nicolescu's result means that we can generate *any* finite string by means of a recursive string copying process. This may yield further insights into the structure and complexity of finite strings.

- compression: the source coding approach presented in Chapter 10 is only one approach towards compression. Lempel-Ziv compression [54], which searches for recurring patterns in a string, is an example of a different approach. The recursive structure of strings that can be read by T-decomposition, and the efficient nature of the T-depletion codes suggest that it may be possible to develop a similar but possibly more efficient algorithm for string compression.

- cryptography: the equivalence relations for T-prescriptions point at some potential for T-Codes in cryptographic applications. The significance of prime numbers in this context is also of some interest.

- error control techniques exploiting the good synchronisation behaviour of T-Codes.

11.2 Conclusion

This thesis has discussed a wide range of issues related to robust source coding with T-Codes, with a focus on the recursive structure of the codes. The main results presented in the thesis were:

- a confirmation of Nicolescu's work on the equivalence of multiple T-prescriptions.

- the derivation of the generalized T-depletion codes: a recursive, fixed-length, multibase number representation of T-Code codewords.

- the description of an encoder and decoder model that convert between T-depletion codewords and T-Code codewords and vice versa.

- an interpretation of the multibase number space spanned by the T-depletion codes as a representation for T-Code codewords and all their proper prefixes. The thesis further showed how this feature may be used to represent arbitrary variable-length codes as multibase numbers.
- an algorithm that converts between T-depletion codewords and a contiguous integer index, which may be of importance especially in low-cost decoders.
- a T-Code synchronisation theory that permits faster synchronisation detection under some circumstances.
- a recursive codeword-based algorithm for the calculation of the expected synchronisation delay (ESD) of T-Code sets that offers a lower computational complexity.
- a recursive search algorithm that yields the T-Code set with the minimum redundancy for a given source. It utilizes various equivalence and feasibility criteria to significantly restrict the search space.

However, as the above list of topics in the previous section shows, the T-Codes continue to offer a variety of interesting avenues for research. Some of these may lead to applications of great practical significance. Compression and encryption algorithms are but one promising area. Much work remains to be done.

Bibliography

[1] A. V. Aho and J. D. Ullman. *Foundations of Computer Science.* Computer Science Press, 1992.

[2] J. Berstel and D. Perrin. *Theory of Codes.* Academic Press Inc., 1985.

[3] E. J. Borowski and J. M. Borwein. *Collins Reference Dictionary Mathematics.* Collins, 1989.

[4] R. M. Capocelli, R. Giancarlo, and I. J. Taneja. Bounds on the Redundancy of Huffman Codes. *IEEE Trans. Inform. Theory*, 32(6):854–857, November 1986.

[5] R. W. M. Chan. *Variable Length Error Control Codes.* PhD thesis, The University of Auckland, 1996.

[6] K.-L. Chung. Efficient Huffman Decoding. *Information Processing Letters*, 61:97–99, 1997.

[7] B. Honary F. Zolghadr and M. Darnell. Statistical Real-Time Channel Evaluation (SRTCE) Technique Using Variable-Length T-Codes. *IEE Proceedings*, 136(4):259–266, August 1989.

[8] P. G. Farrell, B. K. Honary, and S. D. Bate. Adaptive Product Codes with Soft/Hard Decision Decoding. In H. J. Beker and F. C. Piper, editors, *Cryptography and Coding*, The Institute of Mathematics & ITS Applications Conference Series, pages 95–111. Oxford Science Publications, 1989.

[9] T. J. Ferguson and J. H. Rabinowitz. Self Synchronizing Huffman Codes. *IEEE Trans. Inform. Theory*, 30(4):687–693, July 1984.

[10] K. J. Frith. *Investigation into the Synchronisation Properties of the CCITT Fax Coding Techniques.* M.E. project report, The University of Auckland, Auckland, New Zealand, January 1993.

[11] R. G. Gallager. Variations on a Theme by Huffman. *IEEE Trans. Inform. Theory*, 24(6):668–674, November 1978.

[12] E. N. Gilbert. Synchronization of Binary Messages. *IRE Trans. Inform. Theory*, 10:933–967, 1960.

[13] E. N. Gilbert and E. F. Moore. Variable Length Binary Encodings. *Bell Syst. Tech. J.*, 38:933–967, July 1959.

[14] S. W. Golomb and B. Gordon. Codes with Bounded Synchronization Delay. *Inform. and Contr.*, 8:355–376, August 1965.

[15] S. W. Golomb, B. Gordon, and L. R. Welch. Comma-Free Codes. *Can. J. Math.*, 10(2):202–209, 1958.

[16] S. W. Golomb, R. E. Peile, and R. A. Scholtz. *Basic Concepts in Information Theory and Coding — The Adventures of Secret Agent 00111*. Plenum Publishing Corporation, 1994.

[17] U. Günther. Data Compression and Serial Communication with Generalized T-Codes. *Journal of Universal Computer Science*, 2(11):769–795, November 1996.

[18] U. Günther, P. Hertling, R. Nicolescu, and M. R. Titchener. Representing Variable-Length Codes in Fixed-Length T-Depletion Format in Encoders and Decoders. *Journal of Universal Computer Science*, 3(11):1207–1225, November 1997.

[19] U. Günther, P. Hertling, R. Nicolescu, and M. R. Titchener. Representing Variable-Length Codes in Fixed-Length T-Depletion Format in Encoders and Decoders. *CDMTCS Research Report 44*, The University of Auckland, Auckland, New Zealand, August 1997.

[20] U. Günther, R. Nicolescu, and M.R. Titchener. Even T-Code Sets. Tamaki Report Series 10, The University of Auckland, Auckland, New Zealand, December 1995.

[21] U. Günther and M. R. Titchener. Calculating the Expected Synchronization Delay for T-Code Sets. *IEE Proceedings — Communications*, 144:121, June 1997.

[22] U. Günther and M. R. Titchener. Calculating the Expected Synchronization Delay for T-Code Sets. In I. G. Richardson, editor, *Proceedings of AVSPN'97*, September 1997.

[23] R.W. Hamming. *Coding and Information Theory.* Prentice-Hall, second edition, 1986.

[24] N. G. Harlick. *A Comparison between JPEG Compressed Images Using Huffman and T-Codes and Transmitted over a UHF Radio Link.* M.Phil (Engineering) project report, The University of Auckland, Auckland, New Zealand, December 1993.

[25] N. G. Harlick. *Using T-Codes in an Implementation of JPEG Image Compression.* M.Phil (Engineering) project report, The University of Auckland, Auckland, New Zealand, December 1993.

[26] G. R. Higgie. *Analysis of the Families of Variable-Length Self-Synchronizing Codes called T-Codes.* PhD thesis, The University of Auckland, 1991.

[27] G. R. Higgie. Database of Best T-Codes. *IEE Proceedings — Computers and Digital Techniques*, 143:213–218, July 1996.

[28] F. S. Hillier and G. J. Lieberman. *Introduction to Operations Research.* McGraw-Hill, 5th edition, 1990.

[29] D. S. Hirschberg and D. A. Lelewer. Efficient Decoding of Prefix Codes. *Communications of the ACM*, 33(4):449–459, April 1990.

[30] D. Huffman. A Method for the Construction of Minimum Redundancy Codes. *Proc. Inst. Radio Eng.*, 40:1098–1101, September 1952.

[31] ISO/IEC. *Information Technology — Telecommunications and Information Exchange Between Systems — High-Level Data Link Control (HDLC) procedures — Frame structure*, December 1993. ISO/IEC Standard, reference number: ISO/IEC 3309:1993(E).

[32] O. Johnsen. On the Redundancy of Binary Huffman Codes. *IEEE Trans. Inform. Theory*, 26(2):220–222, 1980.

[33] J. C. Maxted and J. P. Robinson. Error Recovery for Variable Length Codes. *Bell Syst. Tech. J.*, 31(6), November 1985.

[34] B. L. Montgomery and J. Abrahams. Synchronization of Binary Source Codes. *IEEE Trans. Inform. Theory*, 32(6):849–854, November 1986.

[35] P. Neumann. Efficient Error-Limiting Variable Length Codes. *IEEE Trans. Inform. Theory*, 8:292–304, July 1962.

[36] R. Nicolescu. Uniqueness Theorems for T-Codes. *Tamaki Report Series 9*, The University of Auckland, September 1995.

[37] D. Perrin and M.-P. Schuetzenberger. Synchronizing Prefix Codes and Automata and the Road Coloring Problem. *Contemporary Mathematics*, 135:295–318, 1992.

[38] M. J. Roberts. *Techniques for Determining the Best T-Codes.* Master's thesis, The University of Auckland, October 1993.

[39] C. E. Shannon. A Mathematical Theory of Communications. *Bell Systems Technical Journal*, 27:379, July 1948.

[40] C. E. Shannon. A Mathematical Theory of Communications. *Bell Systems Technical Journal*, 27:623, October 1948.

[41] Y. Takishima, M. Wada, and H. Murakami. Error States and Synchronization Recovery For Variable Length Codes. *IEEE Trans. Commun.*, 42(2-4):783–792, February 1994.

[42] H. Tanaka. Data Structure of Huffman Codes and its Application to Efficient Encoding and Decoding. *IEEE Trans. Inform. Theory*, 33(1):154–156, January 1987.

[43] M. R. Titchener. verbal communication.

[44] M. R. Titchener. Technical Note: Digital Encoding by Way of New T-codes. *IEE Proceedings — Computers and Digital Techniques*, 131(4):151–153, 1984.

[45] M. R. Titchener. Construction and Properties of the Augmented and Binary-Depletion codes. *IEE Proceedings — Computers and Digital Techniques*, 132(3):163–169, May 1985.

[46] M. R. Titchener. *The Augmented and Binary Depletion T-codes.* PhD thesis, The University of Auckland, May 1986.

[47] M. R. Titchener. A Character Error Bound for the T-code Synchronization Process. *IEE Proceedings — Computers and Digital Techniques*, 134(3):155–158, May 1987.

[48] M. R. Titchener. Unequivocal Codes:- String Complexity and Compressibility. *Tamaki T-Code Project Series 1*, The University of Auckland, Auckland, New Zealand, August 1993. ISSN 1174-314X.

[49] M. R. Titchener. Generalized T-Codes: an Extended Construction Algorithm for Self-Synchronizing Variable-Length Codes. *IEE Proceedings — Computers and Digital Techniques*, 143(3):122–128, June 1996.

[50] M. R. Titchener. The Synchronization of Variable-Length Codes. *IEEE Trans. Inform. Theory*, 43(2):683–691, March 1997.

[51] M. R. Titchener and J. J. Hunter. Synchronization Process for the Variable-Length T-codes. *IEE Proceedings — Computers and Digital Techniques*, 133(1):54–64, 1985.

[52] M. R. Titchener and S. Wackrow. T-Code Online Development Environment. *Tamaki T-Code Project Series 8*, The University of Auckland, Auckland, New Zealand, October 1995. ISSN 1174-314X.

[53] V. K. W. Wei and R. A. Scholtz. On the Characterization of Statistically Synchronizable Codes. *IEEE Trans. Inform. Theory*, 26(6):733–735, November 1980.

[54] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Trans. Inform. Theory*, 24(5):530–536, September 1978.

Index