Journal of Universal Computer Science, vol. 2, no. 11 (1996), 769-795
submitted: 29/3/96, accepted: 17/10/96, appeared: 28/11/96 © *Springer Pub. Co.*

Data Compression and Serial Communication with Generalized T-Codes

Ulrich Günther
(The University of Auckland, New Zealand
ulrich@tcode.tcs.auckland.ac.nz)

Abstract: Variable-length T-Codes may be used to provide robust compression for data communication and storage on noisy channels or media. Over the past twelve years, a number of papers on T-Codes have been published in various journals and as technical reports. During this time, notation and scope of the T-Codes have changed considerably, giving rise to a more integrated theory of T-Codes as recursive codes. This paper presents all known core principles of T-Code theory by taking a recursive approach throughout. A sufficient condition for information sources, ensuring decoder self-synchronisation for the T-encoded symbol stream, is introduced. By example of a recursive program, the paper shows how a suitable T-Code set for encoding a given memoryless source can be found.

Key Words: T-Codes, coding, synchronisation, T-augmentation, recursive code, coding efficiency, redundancy, string decomposition

Category: H.3.3

1 Introduction

It is now twelve years since Mark Titchener, then a postgraduate student, developed the concept of a recursive coding system which became known as the "T-Codes" [Titchener 1984], [Titchener 1985], [Titchener 1986]. A comparatively late arrival on the scene of digital communications, T-Codes nevertheless offer a range of interesting features in a combination that other coding systems do not exhibit. Some of these features are explored here in detail, including the feature that has perhaps attracted most interest: the inherent self-synchronisation in the face of errors [Titchener and Hunter 1985]. Self-synchronisation is of general significance for variable-length codes, and has been discussed by a number of investigators, including [Gilbert 1960], [Ferguson and Rabinowitz 1984], [Maxted and Robinson 1985], [Montgomery and Abrahams 1986], and [Takishima, Wada, and Murakami 1994].

In the first section, we will introduce the concepts of recursive coding, simple T-augmentation and simple T-Codes, which are then expanded to the generalized T-augmentation and generalized T-Codes. The second section briefly discusses these concepts in a graphical tree picture. The third section deals with the self-synchronisation property of T-Codes. After having established the theoretical basis for this useful property, we show how to find the most efficient T-Code set for the encoding of a given memoryless information source. Finally, we show how to T-decompose strings, which is a possible method of relaying code set information in a compact way.

2 T-Code Set Construction by T-Augmentation

The following sections give an introduction to the recursive coding concept [Section 2.1] on which the T-Code construction properties are based, introduce T-augmentation and T-Code sets [Section 2.2], and discuss the set size lemma [Section 2.2].

2.1 Recursive Coding and Variable-Length Codes

What is recursive coding? Imagine a piece of English text (the works of Shakespeare come to mind as an example). We could split the works of Shakespeare into several books, and refer to them as "Macbeth", "Romeo and Juliet" and so on. The titles are themselves a kind of encoding as there seems to be a general understanding of what, e.g., "Romeo and Juliet" stands for. Thus we may regard Shakespeare's works as a concatenation of books. Then again, we could regard the works of Shakespeare as a concatenation of chapters (or acts), and refer to individual chapters instead of books. We observe that, if we printed all of Shakespeare's works in a single volume, any boundary between two books is also a boundary between two chapters, but not vice versa. We could now regard each chapter as a concatenation of sentences, which makes the works of Shakespeare a concatenation of sentences. A boundary between two books is also a boundary between chapters which is also a boundary between sentences. The reverse is not true, however: a boundary between two sentences does not generally mark the boundary between two chapters or even books, and a boundary between chapters is generally no boundary between books. Continuing down the same path, we know that sentences are made up of words which are made up of letters. If we regard letters as, e.g., ASCII characters, we can even split them up into bits.

In the last paragraph, we regard Shakespeare's works as an example for recursive coding over several levels. Shakespeare's works, viewed from this aspect, also provide an example for variable-length codes. Words in the English language are made up from a varying number of letters, and sentences from a varying number of words. The same applies to sentences and chapters, chapters and books, and so on. At the lowest coding level (letters as ASCII characters with 8 bits each), however, we have an example of a block code, where each codeword (ASCII letter) contains a fixed number of (binary) alphabet symbols (bits).

We could also imagine a situation where the letters are encoded with a Huffman or a similar binary variable-length code [see Huffman 1952]. In this case, frequent letters such as "e" or "r" are represented by short codewords, and letters such as "w" or "x" with a low frequency of occurrence are represented by longer codewords. The total number of bits needed to represent Shakespeare's works may thus be reduced significantly compared to the ASCII encoding. Therefore, variable-length codes are very popular as the permit data compression for storage or serial communication of data.

Now imagine the works of Shakespeare being sent between two computers on a serial cable which is prone to bit errors. Bit errors can come as bit inversion, bit deletion or bit insertion. Unless there is a mechanism to ensure that bit errors don't propagate very far, the works of Shakespeare as received by the second computer may have very little literary quality left... In the case of ASCII encoding, only bit deletion or bit insertion cause any lasting difficulty. However, with a variable-length encoding, any type of bit error may cause major problems as boundaries between individual codewords are not evenly spaced.

Encoding the works of Shakespeare with T-Codes is a possible way of keeping the effect of bit errors restricted, while still being able to compress the data for efficient transmission.

We will now show how T-Code sets may be constructed, highlighting their recursive structure. Readers who are familiar with the construction of Huffman codes will probably miss the source probabilities at this stage — this will be discussed later.

2.2 T-Augmentation

How does one construct a T-Code set? The construction starts from a base alphabet S. All T-Code properties presented here hold for any finite alphabet S with at least two symbols, i.e., $\#S \geq 2$. However, in our examples we will assume for simplicity that S is the binary alphabet $\{0, 1\}$. Let us first consider what we will call "simple T-augmentation":

Example 1. [Simple T-augmentation]. We write down the alphabet $S = \{0, 1\}$ (which in itself is a T-Code set at "0'th T-augmentation level") in a single column in a table (shown here below the double line):

	T-augmentation level			
n	0	1	2	3
k_n	n/a	-	-	-
set	S	-	-	-
	0			
	1			

The line labeled k_n may be ignored for the time being, its significance will be discussed later when we will discuss generalized T-augmentation. Now we will *T-augment* the set to *T-augmentation level 1*. The first step is to copy the current set twice into the next column:

	T-augmentation level			
n	0	1	2	3
k_n	n/a	1	-	-
set	S	-	-	-
	0	0		
	1	1		
		0		
		1		

The next step is to select a codeword from the first copy of the set, e.g., the 1. This codeword — called the "T-prefix" — is eliminated from the first copy of the list and prefixed (concatenated on the left-hand side) to the second copy:

	T-augmentation level			
n	0	1	2	3
k_n	n/a	1	-	-
set	S	$S_{(1)}$	-	-
	0	0		
	1	~~1~~		
		10		
		11		

This concludes the first simple T-augmentation. Our T-Code set at T-augmentation level 1 is now $S_{(1)} = \{0, 10, 11\}$, where the subscript indicates the *T-prefix* chosen.

We note that $S_{(1)} = \{0, 10, 11\}$ is prefix-free, i.e., none of the codewords is the prefix of another. It is also complete, which means that any arbitrary (infinite) bit stream has a unique decoding over $S_{(1)}$ and no ambiguities arise (cf., e.g., [Bell, Cleary, and Witten 1990], p.208). Complete code sets have also been referred to as "exhaustive" (cf., e.g., [Gilbert and Moore 1959], [Higgie 1991], [Roberts 1993]).

With the present set, only three codewords are available. If one would like to encode an ASCII file, for example, 256 codewords would be needed to encode all possible 8-bit ASCII characters. Therefore, we may wish to increase the number of available codewords by T-augmenting again:

	T-augmentation level			
n	0	1	2	3
k_n	n/a	1	1	-
set	S	$S_{(1)}$	$S_{(1,10)}$	-
	0	0	0	
	1	$\cancel{1}$	–	
		10	$\cancel{10}$	
		11	11	
			100	
			–	
			1010	
			1011	

This time we chose 10 as our T-prefix, copied the set over to the next column and duplicated it, eliminated the T-prefix from the first part of the list and prefixed it to the second part. The set at T-augmentation level 2 is now $S_{(1,10)} = \{0, 11, 100, 1010, 1011\}$, and the subscript vector lists the T-prefixes used in the two T-augmentations that we performed. This set is again complete and prefix-free. Note that the choice of T-prefix is arbitrary — we could have just as well taken any of the other codewords from $S_{(1)}$. This would have neither changed the completeness nor the prefix-freeness of the set nor the total number of codewords in the set (five). Only the codeword length distribution of the set can be manipulated this way. As we will see, this is a useful tool when we wish to match the probability distribution of a given source. A further T-augmentation to T-augmentation level 3 might look like this:

	T-augmentation level			
n	0	1	2	3
k_n	n/a	1	1	1
set	S	$S_{(1)}$	$S_{(1,10)}$	$S_{(1,10,0)}$
	0	0	0	$\cancel{0}$
	1	$\cancel{1}$	–	–
		10	$\cancel{10}$	–
		11	11	11
			100	100
			–	–
			1010	1010
			1011	1011
				00
				–
				–
				011
				0100
				–
				01010
				01011

By now, it should be apparent that we used 0 as the T-prefix for this T-augmentation, and that the resulting set $S_{(1,10,0)}$ is once again prefix-free and complete. The number of codewords in this set is 9.

In fact, the T-augmentation process may be repeated over and over again to generate large sets with an unrestricted number of codewords. All of those sets will be complete and prefix-free. Note that this is a property of the algorithm, and not of the particular T-prefixes that we have chosen. All T-Code sets that are derived entirely by simple T-augmentation are called "simple T-Code sets".

Simple T-augmentation as described above was the original "augmentation" algorithm proposed by Titchener in [Titchener 1984], and has since been investigated by others including [Higgie 1991] and [Roberts 1993]. As we shall see, it is a special case of the generalized T-augmentation.

To generalize T-augmentation, we expand the notation for simple T-Code sets by adding a superscript for each T-augmentation level to our set notation. This superscript is known as *T-expansion factor* or *T-expansion parameter*, and for a single T-augmentation it is denoted as k. k is a positive integer. In a set constructed by multiple T-augmentations, the T-expansion factors form a (superscript) vector $(k_1, k_2, \ldots, k_n)$ similar to the (subscript) T-prefix vector $(p_1, p_2, \ldots, p_n)$. Accordingly, we also use the notation $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$.

But why do we need a T-expansion factor? So far, when T-augmenting, we have copied the existing set twice to the next column. In generalized T-augmentation, we copy the set $k+1$ times. However, the multiple copies alone do not complete the T-augmentation. We also need to remove the T-prefix for this T-augmentation from all copies of the set in the new column, except for the last copy. The T-prefix is then attached once to the second copy of the set, twice to the third, three times to the fourth and so forth. If we set $k = 1$, we copy the set twice. Hence simple T-augmentation is merely a special case of the generalized T-augmentation, and we may use the simplified notation $S_{(p_1,p_2,\ldots,p_n)} = S_{(p_1,p_2,\ldots,p_n)}^{(1,1,\ldots,1)}$. So $S_{(1,10,0)}$ becomes $S_{(1,10,0)}^{(1,1,1)}$ in the generalized T-augmentation notation.

Example 2. [Generalized T-Augmentation]. Consider the set $S_{(1,10)}^{(1,1)}$ at the second T-augmentation level from our previous example above. Let us use 0 as the third-level T-prefix again, but this time with a T-expansion parameter $k_3 = 3$ rather than $k_3 = 1$ as before. We obtain:

	T-augmentation level			
n	0	1	2	3
k_n	n/a	1	1	3
set	S	$S_{(1)}^{(1)}$	$S_{(1,10)}^{(1,1)}$	$S_{(1,10,0)}^{(1,1,3)}$
	0	0	0	$\cancel{0}$
	1	$\cancel{1}$	–	–
		10	$\cancel{1}\cancel{0}$	–
		11	11	11
			100	100
			–	–
			1010	1010
			1011	1011
				$\cancel{0}\cancel{0}$
				–
				–
				011
				0100
				–
				01010
				01011
				$\cancel{0}\cancel{0}\cancel{0}$
				–
				–
				0011
				00100
				–
				001010
				001011
				0000
				–
				–
				00011
				000100
				–
				0001010
				0001011

The resulting set is denoted $S_{(1,10,0)}^{(1,1,3)}$. We note that this set is also prefix-free and complete.

We can now summarize and formally define generalized T-augmentation:

Definition 1. [Generalized T-augmentation]. Consider an alphabet S, a set $S_1 \subset S^+$, a string $p \in S_1$, and a positive integer k. The operation that generates the set S_2 according to the rule

$$S_2 = \bigcup_{i=0}^{k} \{p^i s | s \in S_1 \backslash \{p\}\} \cup \{p^{k+1}\} \tag{1}$$

is called T-augmentation. The string p is called T-prefix and the integer k is called T-expansion parameter or T-expansion factor. The set S_2 may be denoted as $S_2 = [S_1]_{(p)}^{(k)}$.

As in our examples, we denote multiple successive T-augmentations using subscript vectors for the T-prefixes and superscript vectors for the T-expansion factors. We now define T-Code sets:

Definition 2 T-Code sets. . Consider a series of $n \geq 0$ successive T-augmentations of an alphabet S using the T-prefixes $p_1, p_2, \ldots, p_n$, and T-expansion parameters $k_1, k_2, \ldots, k_n$ respectively. The resulting set is denoted $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ and is referred to as a T-Code set at the n'th T-augmentation level.

Note that according to this definition, an alphabet is always a T-Code set at 0'th T-augmentation level. We now define simple T-augmentation as a special case of T-augmentation:

Definition 3 Simple T-augmentation and Simple T-Code sets. The $n+1$'th T-augmentation from T-augmentation level n to T-augmentation level $n+1$ is said to be simple iff $k_{n+1} = 1$. A T-Code set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ for which $k_1 = k_2 = \ldots = k_n = 1$ is called a simple T-Code set. It may be denoted $S_{(p_1,p_2,\ldots,p_n)}$.

Terminology: with reference to a T-Code set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ at the n'th T-augmentation level, a T-Code set $S_{(p_1,p_2,\ldots,p_i)}^{(k_1,k_2,\ldots,k_i)}$ where $i \leq n$ may also be referred to as an "intermediate" T-Code set. **Abbreviated notation:** a T-Code set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ may be written as $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)} = S_{\mathbf{p}}^{\mathbf{k}} = S_{\mathbf{p}:n}^{\mathbf{k}:n}$ where $\mathbf{p} = (p_1, p_2, \ldots, p_n)$ and $\mathbf{k} = (k_1, k_2, \ldots, k_n)$. Intermediate T-Code sets at T-augmentation levels i, $0 \leq i \leq n$, may be written as $S_{\mathbf{p}:i}^{\mathbf{k}:i}$. For simple T-Code sets, we use the abbreviated notation $S_{(p_1,p_2,\ldots,p_n)} = S_{\mathbf{p}} = S_{\mathbf{p}:n}$ etc.

In some circumstances, a special class of T-Code sets are of interest:

Definition 4 Minimal and Strictly Minimal T-Code Sets. . A T-Code set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ for which all the T-prefixes $p_1, p_2, \ldots, p_n$ used in its construction are of a length that is smaller than or equal to the length of the shortest codeword in $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$, i.e.,

$$\forall\, m \leq n,\ s \in S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)} \ \ |s| \geq |p_m|, \tag{2}$$

is said to be "minimal". Furthermore, if this is also the case for all intermediate T-Code sets, i.e.,

$$\forall\, m \leq n,\ s \in S_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)} \ \ |s| \geq |p_m|, \tag{3}$$

$S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ is said to be "strictly minimal".

2.3 T-Code Set Size

The following lemma on the number of strings in $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ was proved by Titchener [Titchener 1986], initially for simple T-Code sets. In its expanded version for generalized T-Code sets [Titchener 1995], the lemma may be stated as follows:

Lemma 5. *[Number of Codewords in S and $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$]. Consider an alphabet S and a prefix-free set of codewords (strings) $S' \subset S^+$ containing $\#S'$ codewords. A series of $n \geq 0$ successive T-augmentations with the T-expansion parameters $k_1, k_2, \ldots, k_n$ will generate a set of*

$$\#S'^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} = 1 + (\#S' - 1)\prod_{i=1}^{n}(k_i + 1) \tag{4}$$

codewords.

Proof: It follows from [Definition 1] that a prefix-free set S' generates

$$\#S'^{(k)}_{(p)} = \#S'(k+1) - k = (k+1)(\#S' - 1) + 1 \tag{5}$$

codewords under a T-augmentation with T-expansion parameter k. By induction over $n \geq 1$ we obtain

$$\begin{aligned} \#S'^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)} &= (k_1+1)(k_2+1)\ldots(k_n+1)(\#S'-1)+1 \\ &= 1 + (\#S' - 1)\prod_{i=1}^{n}(k_i + 1) \end{aligned} \tag{6}$$

as the number of codewords in the set $S'^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$.□

In the case of n simple T-augmentations of an alphabet S, the lemma simplifies to:

$$\#S_{\mathbf{p}:n} = 1 + (\#S - 1)2^n. \tag{7}$$

3 T-Code Sets Represented by Trees

Readers that are familiar with coding issues will know that prefix-free code sets based on symbols from an alphabet S may be represented by "decoding trees". Starting at the "root" of the tree, each symbol that is received causes the decoder to either transit along a "branch" of the tree to another "node" of the tree, or to conclude that a complete codeword has been received. There are two types of nodes in a decoding tree: "branch nodes" and "leaf nodes". Branch nodes are characterized by up to $\#S$ outgoing branches associated with the receipt of symbols from S. Leaf nodes, on the other hand, have no outgoing branches attached to them. Once the decoder has reached a leaf node, it knows that it has found a codeword boundary in the received symbol stream and that the next symbol in the stream belongs to the following codeword. The same may be the case if the decoder reaches a branch node and the next symbol received has no outgoing branch associated with it. A complete code is characterized by the fact that all branch nodes in the tree have exactly $\#S$ outgoing branches.

Just like other prefix-free, complete code sets, a T-Code set may be represented graphically as such a tree. In fact, the recursive nature of the code set construction may be visualized by drawing the T-Code set as a tree.

Example 3 . [T-Code set construction in the tree picture]. Consider [Fig. 1]. The original set $S = \{0, 1\}$ is first augmented with T-prefix 1 and T-expansion factor 2. Two additional copies of the tree representing A, connected to each other at the 1-node of the upper tree, are affixed to the 1-node of the original tree. Thus we get the set

$$S_{(1)}^{(2)} = \{0, 10, 110, 111\}$$

at T-augmentation level $n = 1$. At T-augmentation level $n = 2$, we augment this set to get the new set

$$S_{(1,10)}^{(2,1)} = \{0, 110, 111, 100, 1010, 10110, 10111\}.$$

In terms of trees, it involves copying the tree from T-augmentation level 1 twice, and attaching the root of one copy to the "10"-node of the other. The "10"-node of the upper tree is now a branch node and no longer a leaf node — a necessary condition for $S_{(1,10)}^{(2,1)}$ to be prefix-free.

Working in the "tree picture" emphasizes the recursive nature of the T-Code construction. This recursive nature is also the "secret" behind the self-synchronisation properties of T-Codes, which we will discuss in the following section.

4 Self-Synchronisation of T-Codes

Self-synchronisation is the property of a code to "pick up the pieces" after one or more symbol errors have occurred on the communication channel (e.g., a serial interface line). These may be errors due to inversion, deletion or insertion of a symbol, and generally cause the decoder to loose alignment with respect to the location of codeword boundaries in the symbol stream. Self-synchronisation means that the code itself permits the decoder to find an unambiguously correct codeword boundary in the course of normal decoding, thus re-establishing correct word alignment.

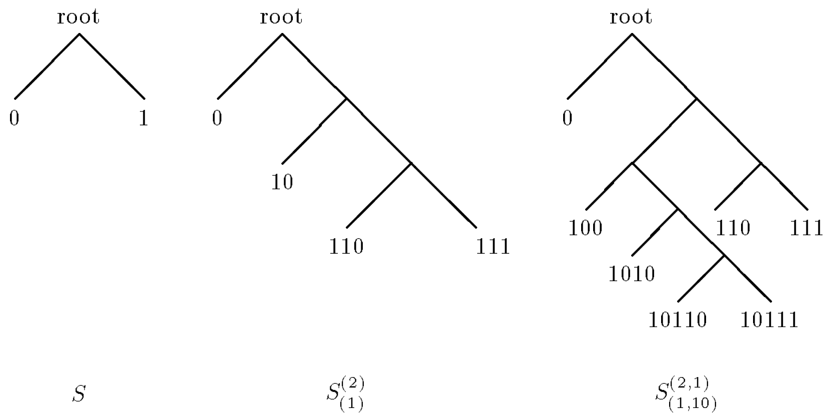

Figure 1: Two T-augmentations from $S = \{0, 1\}$ via $S^{(2)}_{(1)}$ to $S^{(2,1)}_{(1,10)}$, using T-prefixes $p_1 = 1$ and $p_2 = 10$, and T-expansion parameters $k_1 = 2$ and $k_2 = 1$: during the first T-augmentation, the basic tree gets copied $k_1 + 1 = 3$ times, and the copies get attached to each other in hierarchical order at their p_1-nodes. During the second T-augmentation, the tree from T-augmentation level 1 gets copied twice, and the root of one of the copies gets attached to the p_2-node of the other copy.

At first glance, variable-length codes seem to present the decoder with a difficult task, as it cannot expect codewords of a fixed length. Therefore, a single symbol error will generally result in the loss of synchronisation with the decoder now erroneously decoding entirely different codewords with different lengths. Thus, variable-length code synchronisation is not only required at the start of the decoding process, but also during the decoding process whenever a symbol error results in a loss of synchronisation.

4.1 The Self-Synchronisation Mechanism of T-Codes

As we will show, T-Code sets are inherently statistically self-synchronisable as a result of their structure, provided that the probability of occurrence for certain codewords is greater than zero.

Definition 6 Statistically Self-Synchronisable Codes. . Consider a finite code C based on an alphabet S, i.e., $C \subset S^+$. C is called self-synchronisable with respect to a given source if the probability of synchronising after the reception of less than N symbols converges to 1 as N goes to infinity:

$$\lim_{N\to\infty} P_{\text{synch}}(N) = 1. \tag{8}$$

An initial proof for the self-synchronisability of (then simple) T-Codes was devised by Titchener and Hunter [Titchener and Hunter 1985]. This proof simply assumed a random input into the decoder, and solved the problem as an absorbing Markov chain. While this proof could be expanded to included generalized T-Codes, the recursive structure of the codes allows us to take a somewhat simpler, inductive approach, and spell out a sufficient condition for the source that ensures that a T-encoded symbol stream will be statistically synchronisable.

Theorem 7. *[T-Codes are statistically synchronisable]. A T-Code set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ is statistically self-synchronising if the probability $P(p_j, j-1)$ of decoding the T-prefix p_j, $1 \leq j \leq n$, in a decoding of the source symbol stream over $S_{\mathbf{p}:j-1}^{\mathbf{k}:j-1}$ is smaller than 1:*

$$P(p_j, j-1) < 1. \tag{9}$$

Proof (essentially by induction): Consider a decoder that starts decoding a symbol stream. The decoder operates with a top-level T-Code set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$. As mentioned before, obtaining synchronisation is equivalent to finding a codeword boundary between codewords from that set in the symbol stream. Due to the recursive structure of T-Code sets, every codeword boundary with respect to the set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ in the symbol stream is also a codeword boundary with respect to $S_{(p_1,p_2,\ldots,p_{n-1})}^{(k_1,k_2,\ldots,k_{n-1})}$ and all other sets at the lower T-augmentation levels, i.e., $S_{(p_1,p_2,\ldots,p_i)}^{(k_1,k_2,\ldots,k_i)}$ where $i < n$. The same applies to all of the intermediate sets, such that a codeword boundary with respect to $S_{(p_1,p_2,\ldots,p_j)}^{(k_1,k_2,\ldots,k_j)}$ is also always a codeword boundary with respect to $S_{(p_1,p_2,\ldots,p_i)}^{(k_1,k_2,\ldots,k_i)}$ if $i < j$. The following example illustrates this:

Example 4. [Common boundaries]. Consider the binary T-Code set $S_{(1,10,0)}^{(1,1,3)}$ from [Example 2]. If we choose an arbitrary message encoded with $S_{(1,10,0)}^{(1,1,3)}$, let's say, e.g., 100100101001110101111011, we can decode it as a message in all of the intermediate sets at T-augmentation levels 0 to 2:

<table>
<tr><td>S</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td></td></tr>
<tr><td>$S_{(1)}^{(1)}$</td><td colspan="2">1 0</td><td>0</td><td colspan="2">1 0</td><td>0</td><td colspan="2">1 0</td><td colspan="2">1 0</td><td>0</td><td colspan="2">1 1</td><td colspan="2">1 0</td><td colspan="2">1 0</td><td colspan="2">1 1</td><td colspan="2">1 1</td><td>0</td><td colspan="2">1 1</td><td></td></tr>
<tr><td>$S_{(1,10)}^{(1,1)}$</td><td colspan="3">1 0 0</td><td colspan="3">1 0 0</td><td colspan="4">1 0 1 0</td><td>0</td><td colspan="2">1 1</td><td colspan="4">1 0 1 0</td><td colspan="2">1 1</td><td colspan="2">1 1</td><td>0</td><td colspan="2">1 1</td><td></td></tr>
<tr><td>$S_{(1,10,0)}^{(1,1,3)}$</td><td colspan="3">1 0 0</td><td colspan="3">1 0 0</td><td colspan="4">1 0 1 0</td><td colspan="3">0 1 1</td><td colspan="4">1 0 1 0</td><td colspan="2">1 1</td><td colspan="2">1 1</td><td colspan="3">0 1 1</td><td></td></tr>
</table>

Here, the vertical bars indicate the position of the codeword boundaries. As we can see, each codeword boundary at a higher level set is also a boundary in the decoding with respect to all lower sets. This is because we can "spell" all codewords in a T-Code set uniquely using codewords from any of its intermediate sets.

A T-Code decoder operating over $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ may thus be thought of as implicitly decoding over all intermediate sets at the same time. In its unsynchronised state, the decoder is only in synchronisation with the codeword boundaries in S. In the following paragraphs, we will show that the decoder may now use the information received to try and determine whether a boundary in S is also a boundary in the higher T-augmentation level sets, up to a maximum level $j \leq n$. If such a common boundary is found, the decoder is at "synchronisation level" j, i.e., in synchronism with respect to $S^{(k_1,k_2,\ldots,k_j)}_{(p_1,p_2,\ldots,p_j)}$.

The decoder using $S^{(k_1,k_2,\ldots,k_j)}_{(p_1,p_2,\ldots,p_j)}$ for decoding now knows that all further codeword boundaries decoded over $S^{\mathbf{k}:j}_{\mathbf{p}:j}$ will be correct ones (provided that no more symbol errors occur). Again, the decoder may use the received information to determine if any of these boundaries is shared with sets at higher T-augmentation levels, and so forth. Once the decoder has determined a codeword boundary in $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$, it is fully synchronised. If it cannot unequivocally determine that a codeword boundary with respect to an intermediate set $S^{\mathbf{k}:j}_{\mathbf{p}:j}$ is also a boundary with respect to a higher set, at least one more codeword must be decoded over $S^{\mathbf{k}:j}_{\mathbf{p}:j}$ before unambiguous synchronisation with respect to a set at a higher level can be established. The synchronisation process may thus be viewed as a sequence of up to n transitions between increasing synchronisation levels.

This leaves us with the question: "When is the decoder able to establish that a boundary with respect to a set $S^{(k_1,k_2,\ldots,k_i)}_{(p_1,p_2,\ldots,p_i)}$ also a boundary with respect to a set $S^{(k_1,k_2,\ldots,k_i,\ldots,k_j)}_{(p_1,p_2,\ldots,p_i,\ldots,p_j)}$ at a higher T-augmentation level j?" Instead, we may ask "when is a boundary with respect to a set $S^{\mathbf{k}:j}_{\mathbf{p}:j}$ also a boundary with respect to a set $S^{(k_1,k_2,\ldots,k_j,k_{j+1})}_{(p_1,p_2,\ldots,p_j,p_{j+1})}$ at the next T-augmentation level?", where $j \geq i$. The initial question may then be answered by induction.

The set $S^{(k_1,k_2,\ldots,k_j)}_{(p_1,p_2,\ldots,p_j)}$, $j \geq i$, shares all but one of its codewords (the T-prefix p_{j+1}) with the set $S^{(k_1,k_2,\ldots,k_{j+1})}_{(p_1,p_2,\ldots,p_{j+1})}$. The remaining codewords in $S^{(k_1,k_2,\ldots,k_{j+1})}_{(p_1,p_2,\ldots,p_{j+1})}$, when read as messages over $S^{(k_1,k_2,\ldots,k_j)}_{(p_1,p_2,\ldots,p_j)}$, also all end in one of these shared codewords — with the exception of the codeword $p_{j+1}^{k_{j+1}+1}$, the only codeword in $S^{(k_1,k_2,\ldots,k_{j+1})}_{(p_1,p_2,\ldots,p_{j+1})}$ that ends in p_{j+1}. Moreover, the T-augmentation algorithm ensures that all $S^{(k_1,k_2,\ldots,k_j)}_{(p_1,p_2,\ldots,p_j)}$-codewords boundaries inside a $S^{(k_1,k_2,\ldots,k_{j+1})}_{(p_1,p_2,\ldots,p_{j+1})}$-codeword are boundaries following the T-prefix p_{j+1}.

Thus, if the decoder can exclude the possibility that the $S^{(k_1,k_2,\ldots,k_j)}_{(p_1,p_2,\ldots,p_j)}$-codeword stream preceding the $S^{(k_1,k_2,\ldots,k_j)}_{(p_1,p_2,\ldots,p_j)}$-codeword boundary in question ends in p_{j+1}, this boundary is definitely also a valid boundary with respect to $S_{(p_1,p_2,\ldots,p_{j+1})}^{(k_1,k_2,\ldots,k_{j+1})}$. The decoder can then switch (at least) to synchronisation level $j+1$. If the decoder cannot unambiguously exclude the possibility that the $S_{(p_1,p_2,\ldots,p_j)}^{(k_1,k_2,\ldots,k_j)}$-codeword stream *might* or does end in the T-prefix p_{j+1}, the decoder cannot switch to synchronisation level $j+1$: we have a situation that we will refer to as a **blocking condition**.

In the case of a blocking condition, the $S_{(p_1,p_2,\ldots,p_j)}^{(k_1,k_2,\ldots,k_j)}$-boundary may be either "in the middle" of an $S_{(p_1,p_2,\ldots,p_{j+1})}^{(k_1,k_2,\ldots,k_{j+1})}$-codeword following a T-prefix p_{j+1}, or at the end of an $S_{(p_1,p_2,\ldots,p_{j+1})}^{(k_1,k_2,\ldots,k_{j+1})}$-codeword (i.e., at a valid codeword boundary with respect to $S_{(p_1,p_2,\ldots,p_{j+1})}^{(k_1,k_2,\ldots,k_{j+1})}$). However, the decoder lacks the necessary information to determine which of these two cases it has encountered. To get this information, it must decode at least one more codeword over $S_{(p_1,p_2,\ldots,p_j)}^{(k_1,k_2,\ldots,k_j)}$ before it can switch to synchronisation level $j+1$ or higher. (The blocking condition replaces the "prefix" and "suffix" conditions used earlier, e.g., in [Titchener 1986]).

For the practical implementation of a decoder, it is of interest to investigate the circumstances under which the decoder might encounter a blocking condition. Consider a decoder that has decoded the last codeword s at synchronisation level i, i.e., $s \in S_{\mathbf{p}:i}^{\mathbf{k}:i}$. Furthermore, let us presume that the decoder has not encountered any blocking conditions with respect to the prefixes $p_{i+1}, \ldots p_j$, $i < j$, as a result of decoding s. If this decoder is to encounter a blocking condition with respect to p_{j+1}, the following necessary condition, referred to as the **blocking pre-condition**, must be satisfied:

$$p_{j+1} \succeq_{S_{\mathbf{p}:i}^{\mathbf{k}:i}} s, \tag{10}$$

i.e., the last codeword decoded, $s \in S_{\mathbf{p}:i}^{\mathbf{k}:i}$, either equals p_{j+1}, or is a suffix of p_{j+1} if p_{j+1} is decoded as a string over $S_{\mathbf{p}:i}^{\mathbf{k}:i}$. It simply means that unless the last codeword s matches the end of p_{j+1}, we do not have a blocking condition. The blocking pre-condition stated above is stricter than that formulated by Titchener [Titchener 1986], insofar as we require that s be a suffix of p_{j+1} not only over S, but also over $S_{\mathbf{p}:i}^{\mathbf{k}:i}$. The justification for this is that p_{j+1} is a codeword in $S_{\mathbf{p}:j}^{\mathbf{k}:j}$, and that it thus has a unique spelling (decoding) in $S_{\mathbf{p}:j}^{\mathbf{k}:j}$ and all lower sets, including $S_{\mathbf{p}:i}^{\mathbf{k}:i}$. This unique decoding over $S_{\mathbf{p}:i}^{\mathbf{k}:i}$ must end in s in case of a blocking condition.

The sufficient condition in [Theorem 7] demands that the T-prefixes at every T-augmentation level should occur with a probability of less than 1. This ensures that when such additional decodings are made over $S_{(p_1,p_2,\ldots,p_j)}^{(k_1,k_2,\ldots,k_j)}$ as a result of a blocking condition, the decoder will eventually encounter a $S_{(p_1,p_2,\ldots,p_j)}^{(k_1,k_2,\ldots,k_j)}$-codeword other than p_{j+1}. This codeword will not satisfy the blocking pre-condition, thus permitting the decoder to switch to synchronisation level $j+1$ or perhaps further. By induction, the decoder will thus eventually reach the final synchronisation level n. This concludes the proof of [Theorem 7].□

For transparency, we may write the synchronisation mechanism for $S_{\mathbf{p}:n}^{\mathbf{k}:n}$ in pseudo-code:

```
<START>
           j = 0; d = λ; status=unsynchronised;
LABEL1:    s = DecodeNextWord(S_{p:j}^{k:j});
           d = Concatenate(d, s);
LABEL2:    if (blocking condition for p_{j+1}) goto LABEL1;
           j = j + 1;
           if (j < n) goto LABEL2;
           status=synchronised;
<END>
```

In this code, the variable j indicates the synchronisation level that the decoder operates at. The string of already decoded symbols is stored in d.

When testing for the occurrence of a blocking condition, it is probably easier in a practical implementation to first test for the occurrence of a blocking pre-condition. As we shall see, this requires fewer comparisons. In practice, most codewords $s \in S_{\mathbf{p}:i}^{\mathbf{k}:i}$, $i > 0$, tend not to cause blocking conditions and tend not to satisfy the blocking pre-condition. Hence it may be prudent for a decoder to test the last decoded word s for blocking pre-conditions rather than for blocking conditions. The associated pseudo-code may be written as:

```
<START>
           j = 0; d = λ; status=unsynchronised;
LABEL1:    s = DecodeNextWord(S_{p:j}^{k:j});
           i = j;
           d = Concatenate(d, s);
LABEL2:    if (p_{j+1} ⪰_{S_{p:i}^{k:i}} s) then
           {
               if (blocking condition for p_{j+1}) goto LABEL1;
           }
           j = j + 1;
           if (j < n) goto LABEL2;
           status=synchronised;
<END>
```

Note that we have introduced a new variable i into the program. It records the T-augmentation level at which the last codeword s was decoded.

Whenever a blocking pre-condition is encountered, the decoder must determine whether there is in fact a blocking condition. In this process, the decoder may encounter three mutually exclusive cases in which a blocking condition exists:

- Case 1: $s = p_{j+1}$. This is the trivial case and the only type of blocking condition that its possible between adjacent levels, i.e., when $j = i$.

- Case 2: $d \succeq_S p_{j+1} \succ_{S^{\mathbf{k}:i}_{\mathbf{p}:i}} s$, i.e., p_{j+1} is a suffix (over S) of the symbol stream d received since the beginning of the synchronisation process, and p_{j+1} is longer than s. This is only possible if the number of symbols in S received since the beginning of the synchronisation process is greater than or equal to the length of p_{j+1}: $|d| \geq |p_{j+1}|$.
- Case 3: $p_{j+1} \succ_S d \succeq_S s$, i.e., the symbol stream d received since the start of the decoding is a suffix (over S) of p_{j+1}, and p_{j+1} is longer than s. This is only possible if the number of symbols received since the beginning of the synchronisation process is less than the length of p_{j+1}: $|d| < |p_{j+1}|$. In this situation, it is possible that "lost" symbols prior to the start of the synchronisation process would have complemented the received symbols to give p_{j+1}, and we have a blocking condition.

If none of the three cases holds, there is no blocking condition and the decoder may switch to synchronisation level $j+1$ or even further. Accordingly, we may refine our pseudo-code above:

<START>
    $j = 0$; $d = \lambda$; status=unsynchronised;
LABEL1:
    $s = \text{DecodeNextWord}(S^{\mathbf{k}:j}_{\mathbf{p}:j})$;
    $i = j$;
    $d = \text{Concatenate}(d, s)$;
LABEL2:
    if ($p_{j+1} \succeq_{S^{\mathbf{k}:i}_{\mathbf{p}:i}} s$) then
    {
        if ($s = p_{j+1}$) then goto **LABEL1**;
        if ($d \succeq_S p_{j+1} \succ_{S^{\mathbf{k}:i}_{\mathbf{p}:i}} s$) then goto **LABEL1**;
        if ($p_{j+1} \succ_S d \succeq_S s$) then goto **LABEL1**;
    }
    $j = j + 1$;
    if ($j < n$) goto **LABEL2**;
    status=synchronised;
<END>

It should now be apparent why it is prudent to test for the blocking pre-condition first: each test for a blocking condition requires four comparisons, whereas a test for a blocking pre-condition requires only a single comparison.

Let us consider an example:

Example 5 . [T-Code self-synchronisation]. Assume that the following bitstream is taken from a message encoded with the binary T-Code set $S^{(1,1,3)}_{(1,10,0)}$ from the table on page 7. The bitstream starts at an arbitrary point, such that the decoder has no initial synchronisation information above the bit level:

$$\ldots 11010111000101001010010001001010100101001010110 \ldots$$

The decoder now starts in synchronism with $S = \{0, 1\}$, i.e., at synchronisation level 0 and finds the first codeword boundary at that level, marked by a dot:

$$\ldots 1.1010111000101001010010001001010100101001010110 \ldots$$

The codeword decoded over $S^{\mathbf{k}:0}_{\mathbf{p}:0} = S$, a 1, is also the T-prefix $p_1 = 1$ for $S^{(1)}_{(1)}$, so we have a blocking condition according to Case 1: it is unclear at this stage whether the 1 is the second bit of the codeword 11 in $S^{(1)}_{(1)} = \{0, 10, 11\}$ or whether it is the first bit of the codewords 10 or 11 in that set. Thus it is ambiguous as to whether the decoder has found a codeword boundary in a higher set, i.e., it cannot switch beyond S for the time being. The decoder now detects the next codeword in S, in this case the next bit:

$$\ldots 1.1.010111000101001010010001001010100101001010110 \ldots$$

This is also a 1, which still doesn't allow the decoder to unambiguously decide whether it has found a codeword boundary in $S^{(1)}_{(1)}$ or a higher set. Either, the first two bits that it decoded are a 11 codeword in $S^{(1)}_{(1)}$, or the first bit is a 11 with the leading 1 missing, and the second 1 is the first bit of a 10 or 11 (at this stage, the decoder does not know that the next bit is a 0). The decoder keeps on decoding in S and "peels" the next bit off:

$$\ldots 1.1.0.10111000101001010010001001010100101001010110 \ldots$$

This time, it has clearly found a codeword boundary in $S^{(1)}_{(1)}$ — a 0 always marks the end of a codeword in $S^{(1)}_{(1)}$. More formally, $0 \neq p_1$. For the decoder, this means that it is now in synchronism with respect to $S^{(1)}_{(1)}$, i.e., it has reached synchronisation level 1. The decoder may now check whether it can go to a higher synchronisation level. The last codeword decoded in S, $s = 0$, is a suffix of the level 2 T-prefix $p_2 = 10$, i.e., we have a blocking pre-condition because $10 \succeq_s 0$. The level 2 T-prefix p_2 is also a suffix of previously received symbol stream $d = 110$, and thus we have a blocking condition according to Case 2 above. The decoder is confined to $S^{(1)}_{(1)}$ for decoding, and it decodes a 10 as the next code word:

$$\ldots 1.1.0.10.111000101001010010001001010100101001010110 \ldots$$

As the 10 is the T-prefix for the next T-augmentation, we have another blocking condition which prevents the decoder from switching to $S^{(1,1)}_{(1,10)}$, and $S^{(1)}_{(1)}$ remains the set over which the next codeword is decoded:

$$\ldots 1.1.0.10.11.1000101001010010001001010100101001010110 \ldots$$

The 11 marks a codeword boundary in $S^{(1,1)}_{(1,10)}$. Thus the decoder is now definitely in synchronism with respect to $S^{(1,1)}_{(1,10)}$. The 11 does not cause a blocking condition with respect to $p_3 = 0$ either, which means that the decoder may switch to synchronisation level 3, i.e., it is now fully synchronised with respect to $S^{(1,1,3)}_{(1,10,0)}$. The remaining codeword boundaries are

$$\ldots 1.1.0.10.11.100.01010.01010.0100.0100.1010.100.1010.01010.11.0 \ldots$$

In this case, we have lost 7 bits to the synchronisation process.

See [Titchener 1986] and [Titchener and Hunter 1985] for an introduction to the synchronisation process for simple T-Codes, and [Titchener 1995] for an extension of this theory to generalized T-Codes. The paper by [Günther and Titchener 1995b] presents a method of calculating the expected synchronisation delay of a given generalized T-Code set, while [Higgie 1991] offers a comparison of the synchronisation performance of simple T-Code sets based on simulations.

4.2 Error Echo and Error Bound

As we have seen, the T-Code self-synchronisation mechanism depends on a sequence of switching processes to higher synchronisation levels, i.e., correct word alignment with respect to T-Code sets at higher T-augmentation levels. A synchronisation process with respect to a set $S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$ starts at T-augmentation level 0 and carries the decoder through to T-augmentation level n. During the process, a decoder at synchronisation level j may switch to level $j+1$ or further unless it decodes the T-prefix p_{j+1} (a Case 1 blocking condition). A synchronising string in S^+ is therefore of the form [see Titchener 1995]:

$$p_1^\star p_2^\star \ldots p_n^\star s, \tag{11}$$

where $s \in S^{(k_1,k_2,\ldots,k_n)}_{(p_1,p_2,\ldots,p_n)}$. The expression $p_j^\star$ represents a series of zero or more successive T-prefixes p_j. A series of p_j may contain successive $p_j^{k_j+1}$-codewords, and may be written as $(p_j^{k_j+1})^\star p_j^l$, where $0 \leq l < k_j + 1$. A decoder operating over $S^{(k_1,k_2,\ldots,k_j)}_{(p_1,p_2,\ldots,p_j)}$, but synchronised only with respect to $S^{(k_1,k_2,\ldots,k_{j-1})}_{(p_1,p_2,\ldots,p_{j-1})}$ and not with respect to $S^{(k_1,k_2,\ldots,k_j)}_{(p_1,p_2,\ldots,p_j)}$, will nevertheless decode the correct sequence of $p_j^{k_j+1}$-codewords. Decoding errors may only occur at the end of the series, where the decoder may be in error by presuming that the following non-p_j-codeword has exactly l T-prefixes p_j. This happens because any T-prefixes p_j preceding the series are "obscured" to the insufficiently synchronised decoder. However, they may have been part of a previous $p_j^{k_j+1}$-codeword preceding the start of the decoding/resynchronisation. Any synchronising sequence, according to [Equation 11], consists of at most n such T-prefix series. Presuming that the end of each of these series causes a separate character error at their end, the maximum number of character errors due to synchronisation is n. Together with a possible decoding error at the start of the synchronisation process (due to missing or corrupted symbols prior to the symbol with which the decoder starts synchronising), this yields an error bound of at most $n+1$ wrongly decoded characters.

An intriguing aspect of this is the so-called "error echo", where character errors may occur long after the actual symbol error in the symbol stream, with correct decoding in between. It is best demonstrated in an example:

Example 6 . [Error Echo]. As a very simple case, consider the set $S_{(1)}^{(1)} = \{0, 10, 11\}$ and the bitstream

$$\ldots 111111111111100 \ldots,$$

with the code assignments $a = 0$, $b = 10$ and $c = 11$. The decoding of the correct bitstream is "*cccccba*". With the first bit in error, the bitstream becomes

$$\ldots 011111111111100 \ldots,$$

and decodes as "*accccccaa*". We see that there is a decoding error at the point where the bit error occurs, followed by a number of correctly decoded T-prefix-T-prefix words, and finally another error — the so-called "error echo".

5 Matching T-Codes to an Information Source

In the previous sections, we have focussed on the structure and properties of T-Code sets without paying any attention to the statistics of an information source that we may wish to encode using T-Codes. However, this is of significance for practical implementations, in particular where good compression is required. We presume here that our information source is memoryless and that each of the m source characters x_j, $j = 1, \ldots, m$, occurs with a certain probability $P(x_j)$. The encoding of x_j, $E(x_j)$, is a codeword from a variable-length code based on a finite alphabet S.

The practical value of this variable-length code is at least partly determined by the redundancy that is left in an encoded symbol stream. The redundancy of the code with respect to the source is given by

$$r(E) = \sum_{j=1}^{m} P(x_j) \left[|E(x_j)| + \log_{\#S} P(x_j) \right], \tag{12}$$

where $|E(x_j)|$ is the length of the encoding of x_j, measured as the number of alphabet symbols in $E(x_j)$. Huffman [Huffman 1952] introduced the now well-known algorithm for maximizing the coding efficiency of memoryless information sources, by deriving the code set directly from the probability distribution of the source. No such direct algorithm is known for T-Code sets. To find the T-Code set which best matches a given source, we may perform a search of all feasible sets and pick one of those that minimize the redundancy.

The search algorithm presented in this paper works recursively. In principle, the routine takes a given T-Code set as input (initially the base alphabet S) and runs through all feasible T-prefixes and T-expansion parameters. It then performs a T-augmentation for each of the combinations and calculates the redundancy of the resulting T-augmented set. This redundancy is compared with the lowest redundancy found so far, which is then updated if required. The routine then calls itself with the T-augmented set as an input.

Before discussing the pseudo-code, it is prudent to first consider a few computational shortcuts. This is required because the number of possible T-prefix/T-expansion parameter combinations is unlimited, and we cannot possibly scan an unlimited number of T-Code sets. As we shall see, the number of feasible sets, i.e., sets that could possibly yield a minimum redundancy, is nevertheless limited. It thus pays to consider carefully which T-prefix/T-expansion parameter combinations are feasible and under which circumstances unnecessary computations may be avoided.

Firstly, we notice that the redundancy depends on $|E(x_j)|$ rather than the literal reading of $E(x_j)$. It is therefore sufficient to generate code length distributions of T-Code sets rather than the full sets, an approach which we will call "virtual T-augmentation". The code length distribution of a T-Code set $S_{\mathbf{p}:i+1}^{\mathbf{k}:i+1}$ is easily calculated if k_{i+1}, $|p_{i+1}|$, and the code length distribution of $S_{\mathbf{p}:i}^{\mathbf{k}:i}$ are known. Let $N(S_{\mathbf{p}:i}^{\mathbf{k}:i}, l)$ be the code length distribution of $S_{\mathbf{p}:i}^{\mathbf{k}:i}$, i.e., the number of codewords of length l in the T-Code set $S_{\mathbf{p}:i}^{\mathbf{k}:i}$, and define $N(S_{\mathbf{p}:i}^{\mathbf{k}:i}, l) = 0$ for $l \leq 0$. Furthermore, we define $N'(S_{\mathbf{p}:i}^{\mathbf{k}:i}, l)$ as

$$N'(S_{\mathbf{p}:i}^{\mathbf{k}:i}, l) = \begin{cases} N(S_{\mathbf{p}:i}^{\mathbf{k}:i}, l) & : 0 < l \neq |p_{i+1}| \\ N(S_{\mathbf{p}:i}^{\mathbf{k}:i}, l) - 1 & : l = |p_{i+1}| \end{cases} \tag{13}$$

It follows from [Definition 2.3] that the code length distribution of $S_{\mathbf{p}:i+1}^{\mathbf{k}:i+1}$ is given by

$$N(S_{\mathbf{p}:i+1}^{\mathbf{k}:i+1}, l) = \sum_{k'=0}^{k_{i+1}-1} N'(S_{\mathbf{p}:i}^{\mathbf{k}:i}, l - k'|p_{i+1}|) + N(S_{\mathbf{p}:i}^{\mathbf{k}:i}, l - k_{i+1}|p_{i+1}|). \tag{14}$$

The first term on the right hand side of this equation accounts for the codewords in the "first k_{i+1} copies of $S_{\mathbf{p}:i}^{\mathbf{k}:i}$" if we use a T-augmentation table as in [Example 2.1]. The second term accounts for the "last copy" that has k_{i+1} T-prefixes attached to its codewords.

[Equation 14] gives rise to a second shortcut. Since $N(S_{\mathbf{p}:i+1}^{\mathbf{k}:i+1}, l)$ depends only on $|p_{i+1}|$, but not on p_{i+1} itself, the redundancy in T-Code sets depends only on the length of the prefixes chosen, but not on their literal reading. This implies that two T-Code sets with identical T-expansion parameters and identical prefix lengths have the same redundancy. Instead of performing virtual T-augmentations for each T-prefix, our routine will only have to consider virtual T-augmentations for each prefix length.

The assignment between source characters and codewords is made in order of the source characters' probabilities, such that for two arbitrary source characters x_j and $x_{j'}$

$$P(x_j) \geq P(x_{j'}) \iff |E(x_j)| \leq |E(x_{j'})|. \tag{15}$$

If a T-Code set has more codewords than there are source characters, some codewords will remain unassigned if they are above a certain length. A further T-augmentation of this set using the T-prefix p and the T-expansion parameter k cannot yield a better redundancy unless we are able to encode at least one characters with a shorter codeword after the T-augmentation than before. As the T-augmentation itself removes a short codeword (the T-prefix), we may require that at least two new codewords be generated that satisfy the following conditions:

- both codewords must be generated by prefixing an existing codeword with the string p^k, i.e., they must contain the maximum number of T-prefixes p. If this is not the case, codewords of equivalent or shorter length (and hence the same or better contributions to the coding efficiency) can be generated in a T-augmentation with T-expansion parameters less than k.
- both codewords must be shorter than the length of the longest codeword used in the non-T-augmented set.

This is our third shortcut. It ensures that the search will terminate as it puts a limit on both T-prefix lengths and T-expansion parameters.

The memory and computing requirements of virtual T-augmentations may be limited by truncating the code length distributions involved. We may truncate them to an upper limit for the length of the longest codeword that could possibly be assigned to a source character. For a source with m characters, this limit is given by the longest codeword possible in a Huffman code set of such a source: for m source characters and the alphabet S, we can always Huffman encode such that, for all j,

$$|E(x_j)| \leq \left\lceil \frac{m-1}{\#S-1} \right\rceil, \tag{16}$$

where the symbols $\lceil$ and $\rceil$ indicate that the expression enclosed is rounded up to the nearest integer. This limit also holds for T-Codes: the T-Code set $S_{(p)}^{(k)}$ where $p \in S$ and $k = \lceil \frac{m-1}{\#S-1} \rceil$ permits such an encoding, and we thus have an upper limit for our redundancy. If it is at all possible to encode more efficiently, the number of codewords that are shorter than $\frac{m-1}{\#S-1}$ must be increased. A T-augmentation that generates only codewords longer than this cannot yield a lower redundancy. As a fourth shortcut, we may hence disregard codeword lengths over $\lceil \frac{m-1}{\#S-1} \rceil$.

A fifth shortcut arises from Nicolescu's paper on the uniqueness of T-Code set prescriptions [Nicolescu 1995]. He shows that if $p_{n+1} = p_n^{(k_n+1)}$,

$$S_{(p_1,p_2,\ldots,p_n,p_{n+1},\ldots,p_{n'})}^{(k_1,k_2,\ldots,k_n,k_{n+1},\ldots,k_{n'})} = S_{(p_1,p_2,\ldots,p_n,p_{n+2},\ldots,p_{n'})}^{(k_1,k_2,\ldots,k'_n,k_{n+2},\ldots,k_{n'})}, \tag{17}$$

where $k'_n = (k_n + 1)(k_{n+1} + 1) - 1$. Nicolescu's rule may be read in reverse: any T-augmentation with an expansion parameter k'_n may be split up into two or more successive T-augmentations — as long as we can factorize $k'_n + 1$. We may therefore restrict ourselves to virtual T-augmentations for which the T-expansion parameter is one less than a prime number. All other T-augmentations create distributions which are (if feasible) created as a matter of course anyway and do not need to be scanned twice.

Last but not least, it is obvious that a particular T-Code set is only worth investigating if the number of codewords in the set is sufficiently large to encode the source.

A recursive program was written which performs searches according to the above principles. Its pseudo-code is listed in [Figure 2]. A feasibility test is performed on each T-prefix/T-expansion parameter combination, using the shortcuts described above.

The early versions of the program were implemented using MATLAB® and Perl. Lately, a C version of the routine has been written, which implements all of the shortcuts described. In most practical applications, the source probabilities are not known to a high precision, such that real numbers in the program may be represented as floats rather than doubles. In its present version, the C routine was tested on a 90 MHz DEC Alpha workstation. The following example gives an indication of performance:

Example 7 . [T-Code versus Huffman code]. For a test, a binary alphabet and a source with 14 characters $x_0, x_1, \ldots, x_{13}$ were chosen. The source characters had the following probabilities of occurrence:

$P(x_1) = 0.15$	$P(x_2) = 0.15$	$P(x_3) = 0.14$	$P(x_4) = 0.14$	$P(x_5) = 0.13$
$P(x_6) = 0.12$	$P(x_7) = 0.1$	$P(x_8) = 0.03$	$P(x_9) = 0.02$	$P(x_{10}) = 0.01$
$P(x_{11}) = 0.005$	$P(x_{12}) = 0.003$	$P(x_{13}) = 0.001$	$P(x_{14}) = 0.001$	

A binary Huffman encoding of this source yields seven codewords of length 3, one each of lengths 4, 5, 6, 7, and 8, and two of length 9. The redundancy in the encoding is approximately 0.035 bits with an achievable compression down to about 79 per cent of the equivalent 4-bit block code. The T-Code matching yields one codeword of length 2, four of length 3, three of length 4, one of length 5, one of length 7, and four of length 8, plus some longer codewords that remain unused. The associated redundancy is approximately 0.1 bits, and compression is achievable down to about 80 per cent. The associated T-Code set may be constructed with T-prefixes of lengths 1, 1, and 5 respectively, with $k_1 = 2$, $k_2 = 2$, and $k_3 = 1$. To obtain this result, 184509929 calls to the recursive matching routine were required, which took 19130 seconds of CPU time on a 90 MHz DEC Alpha workstation.

The execution time of the routine remains a concern. However, it is conceivable that further shortcuts in program structure and the algorithm may be found.

```
<MAIN>
  {
    /* define alphabet size and source character probabilities */
    global const number_of_alphabet_symbols;
    global const source_probability_distribution;
    /* initialize records of best matching set */
    global best_redundancy=INFINITY;
    global best_code_distribution=();
    global best_set_parameters=();
    /* define start values:
       start distribution has number_of_alphabet_symbols
       symbols of length 1, and no parameters have been used so
       far as no virtual T-augmentation has taken place yet */
    local start_code_distribution=(number_of_alphabet_symbols);
    local start_set_parameters=();
    /* call recursive matching routine */
    recursive_match(start_code_distribution,start_set_parameters);
    /* print results */
    print(best_redundancy);
    print(best_code_distribution);
    print(best_set_parameters);
  }

<SUBROUTINE> recursive_match(base_code_distribution,base_set_parameters)
  {
    local p=minimum_length(base_code_distribution);
    /* use T-prefix length only if feasible */
    while(feasible(base_code_distribution,p,1))
    {
        local k=1; /* start with lowest k */
        /* use T-expansion parameter only if feasible */
        while(feasible(base_code_distribution,p,k))
        {
            /* perform virtual T-augmentation */
            local new_code_distribution=t_augment(base_code_distribution,p,k);
            local new_set_parameters=(base_set_parameters,(p,k));
            local new_redundancy=redundancy(new_code_distribution);
            /* does the new set have a lower redundancy? */
            if (new_redundancy<best_redundancy)
            {
                best_redundancy=new_redundancy;
                best_code_distribution=new_code_distribution;
                best_set_parameters=new_set_parameters;
            }
            /* recursively try to match new set */
            recursive_match(new_code_distribution,new_set_parameters);
            /* try a larger T-expansion parameter */
            k=next_prime_minus_one(k+1);
        }
        /* try a longer T-prefix */
        p=next_length(base_code_distribution,p);
    }
  }
```

Figure 2: pseudo-code for the recursive matching routine.

The fact that the choice of T-prefixes for minimum redundancy is only restricted to their length may permit to influence other aspects of the code. This could be used to, e.g., adapt the code to the requirements of a band-limited channel.

6 String Decomposition of the Longest Codewords

After "playing" with a few T-Code sets, the reader would probably find some of the properties of T-Code sets relating to their longest codewords:

- there are always two of them if a binary alphabet is used. More general, an alphabet with n characters results in T-Code sets with n longest codewords.
- the n longest codewords are all identical, except for the last symbol in each codeword. We call this last symbol the "literal symbol", or "literal bit" in the binary case.
- minimal T-Code sets have the shortest longest codewords for a given set size.

This is not coincidental. In fact, the definition of T-Code sets implies that the longest codewords $\hat{s}_c$ in a particular T-Code set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$ include all n T-prefixes, i.e., that they are of the form

$$\hat{s}_c = p_n^{k_n} p_{n-1}^{k_{n-1}} \ldots p_1^{k_1} c, \tag{18}$$

where $c \in S$ is the "literal symbol". This may be seen if the right hand side of the above equation is read from the right to the left: initially, our set is an alphabet and hence all elements, including the literal symbol "c", are "longest codewords". The maximum length string that can be prefixed to any of these in the first T-augmentation is $p_1^{k_1}$, such that $p_1^{k_1}c$ is one of the longest codewords in $S_{(p_1)}^{(k_1)}$. The maximum length string that can be prefixed $p_1^{k_1}c$ in the second T-augmentation is in turn $p_2^{k_2}$, and so forth.

Nicolescu [Nicolescu 1995] showed that the T-prefix part of the longest codeword, $p_n^{k_n} p_{n-1}^{k_{n-1}} \ldots p_1^{k_1}$, defines the set $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$. Nicolescu called this part the "T-handle" of $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$. In analogy with the vector notation used for T-Code sets, it is also denoted $\tilde{\mathbf{p}}^{\mathbf{k}}$, where the tilde indicates that the entries in the vectors are in reverse order. This result may be used to derive a set from one of its longest codewords $\hat{s}_c$ as follows:

1. start at T-augmentation level 0 with the set S..
2. decode $\hat{s}_c$ over the set at the current T-augmentation level.
3. if $\hat{s}_c$ decodes as a single codeword, $\hat{s}_c$ is the longest codeword in the set at the current T-augmentation level. Stop.

4. identify the second codeword from the right of the decoding of $\hat{s}_c$. This codeword is the T-prefix for the next T-augmentation level.
5. count the number of times that this T-prefix appears in the decoding left of the last codeword decoded. This is the T-expansion parameter for the next T-augmentation level.
6. create the next level set using the T-prefix and T-expansion factor found. Make it the current T-augmentation level set and continue at step 2.

Any arbitrary string in S^+ may be used as input for this algorithm.

Example 8 . [String Decomposition]. Consider the binary string

$$00101001001001001.$$

We first start decoding the string from the left, using S as our decoding alphabet:

$$0.0.1.0.1.0.0.1.0.0.1.0.0.1.0.\underline{0}.1$$

Naturally, we decode single bits only at this stage. The (underlined) second bit from the right, a "0", is the T-prefix p_1 for the first T-augmentation level. Looking further to the left, we also see that this prefix appears twice, and thus $k_1 = 2$. Hence our first level set is $S_{(0)}^{(2)} = \{1, 01, 000, 001\}$. We now use this set to decode the string. As before we start on the left:

$$001.01.001.001.\underline{001}.001$$

We can now see that $p_2 = 001$ (underlined) is the second level prefix. Looking to the left, we see that it occurs $k_2 = 3$ times. Thus the T-Code set at the second T-augmentation level is

$$\begin{aligned} S_{(0,001)}^{(2,3)} = \{&1, 01, 000, 0011, 00101, 001000,\\ &0010011, 00100101, 001001000, 0010010011,\\ &00100100101, 001001001000, 001001001001\} \end{aligned}$$

The string is now decoded using this set:

$$\underline{00101}.001001001001.$$

This gives us $p_3 = 00101$ and $k_3 = 1$, which leads to the final set:

$$\begin{aligned} S_{(0,001,00101)}^{(2,3,1)} = \{&1, 01, 000, 0011, 001000,\\ &0010011, 00100101, 001001000, 0010010011,\\ &00100100101, 001001001000, 001001001001,\\ &001011, 0010101, 00101000,\\ &001010011, 0010100101, 00101001000,\\ &001010010011, 0010100100101, 00101001001000,\\ &001010010010011, 0010100100100101,\\ &00101001001001000, 00101001001001001\}. \end{aligned} \tag{19}$$

In a practical application, string decomposition of the longest codewords permits the communication of the entire structure of a T-Code set to a receiver.

7 Discussion

As we have seen, T-Codes exhibit strong inherent self-synchronisation properties. This is an advantage over most Huffman codes when used for encoding over noisy or otherwise lossy communication channels. Huffman codes may have to be transformed in many cases to provide appreciable self-synchronisation properties [Ferguson and Rabinowitz 1984],[Takishima, Wada and Murakami 1994]. In many cases, Huffman codes may be transformed into T-Codes with equivalent redundancy, but improved synchronisation performance, or into codes that are very similar to T-Codes [Titchener 1995]. The source matching algorithm presented in this paper provides a practical tool for finding a suitable T-Code set for a given source. While the present algorithm may be slow, it is conceivable that further, less obvious “shortcuts” exist which may lead to a significant reduction in complexity of the source matching process and thus to a significant increase in speed.

The present state of research suggest that T-Codes in the form presented might be most useful in applications where the primary goals are a high data throughput (requiring compression) and a good long-term error performance. Digital telephony, video, and storage of image, video and audio data come to mind. While being self-synchronizing, the T-Codes introduced in this paper provide no immediate error correction or detection. However, in the applications mentioned this is usually of secondary importance as the human ear and eye are capable of integrating over short-term errors.

Having the choice between several T-Code sets with the same redundancy seems to provide additional possibilities for practical applications, such as influencing the spectral properties of the encoded bitstream. Nevertheless, the general principle that improvements in synchronisation performance affect the coding efficiency by adding redundancy seems to hold: the best matching T-Code sets generally exhibit more redundancy with respect to a given source than a Huffman encoding of the source. However, this is not always so, as the example in the program in [Section 5] shows.

Finally the inherent beauty of the T-Codes and their recursive construction suggests that many other aspects of T-Codes are yet to be discovered.

8 Acknowledgements

First and foremost, I owe thanks to Mark Titchener, inventor and pioneer of the T-Codes, whose research forms the foundation this paper builds on, and whose critical comments and suggestions have helped enormously. Thanks also go to Cris Calude for viewing this paper before submission. Finally, this paper was made possible with the support of the Computer Science Department, the Division of Science and Technology and the Graduate Research Fund of the University of Auckland, NZ.

References

[Bell, Cleary, and Witten 1990] Bell, T.C., Cleary, J.G., and Witten, I.H.: "Text Compression"; Prentice-Hall (1990)

[Ferguson and Rabinowitz 1984] Ferguson, T.J. and Rabinowitz, J.H.: "Self synchronizing Huffman codes"; IEEE Transactions on Information Theory, 30, 4 (July 1984), 687–693.

[Gilbert 1960] Gilbert, E.N.: "Synchronization of Binary Messages"; IRE Transactions on Information Theory, 10 (1960), 470-477.

[Gilbert and Moore 1959] Gilbert, E.N. and Moore, E.F.: "Variable Length Binary Encodings"; Bell Systems Technical Journal, 38 (July 1959), 933-967, July 1959.

[Günther and Titchener 1995b] Günther, U.M. and Titchener, M.R.: "Calculating the Expexted Synchronization Delay for T-Code Sets"; submitted to IEE Proceedings, October 1995.

[Higgie 1991] Higgie, G.R.: "Analysis of the familes of variable-length self-synchronizing codes called T-Codes"; PhD thesis, The University of Auckland, 1991.

[Huffman 1952] Huffman, D.A.: "A Method for the Construction of Minimum Redundancy Codes"; Proceedings of the IRE, 40 (September 1952), 1098–1101.

[Maxted and Robinson 1985] Maxted, J.C. and Robinson, J.P.: "Error Recovery for Variable Length Codes"; Bell Systems Technical Journal, 31, 6 (November 1985), 794-801.

[Montgomery and Abrahams 1986] Montgomery, B.L. and Abrahams, J.: "Synchronization of Binary Source Codes"; IEEE Transactions on Information Theory, 32, 6 (November 1986), 849-854.

[Nicolescu 1995] Nicolescu, R.: "Uniqueness Theorems for T-Codes"; Technical Report, Tamaki Report Series, 9 (September 1995), The University of Auckland.

[Roberts 1993] Roberts, M.J.: "Techniques for Determining the Best T-Codes"; ME thesis, The University of Auckland, October 1993.

[Takishima, Wada, and Murakami 1994] Takishima, Y., Wada, M., and Murakami, H.: "Error States and Synchronization Recovery For Variable Length Codes"; IEEE Transactions on Communications, 42, 2-4 (February 1994), 783-792.

[Titchener 1984] Titchener, M.R.: "Technical Note: Digital Encoding by Way of New T-codes"; IEE Proceedings Pt.E (Computers and Digital Techniques), 131, 4 (1984),151-153.

[Titchener 1985] Titchener, M.R. "Construction and Properties of the Augmented and Binary-Depletion codes"; IEE Proceedings Pt.E (Computers and Digital Techniques), 132, 3 (1985), 163-169.

[Titchener 1986] Titchener, M.R.: "The Augmented and Binary Depletion T-codes"; PhD thesis, The University of Auckland, May 1986.

[Titchener 1995] Titchener, M.R.: "Generalized T-Codes: an Extended Construction Algorithm for Self-Synchronizing Variable-Length Codes"; IEE Proceedings – Communications, 143, 3 (1996), 122-128.

[Titchener and Hunter 1985] Titchener, M.R. and Hunter, J.J.: "Synchronization Process for the Variable-Length T-codes"; IEE Proceedings Pt.E (Computers and Digital Techniques), 133, 1 (1985),54-64.